GBASE®

# GBase 8s SQL 指南：语法



www.gbase.cn

GBase 8s SQL 指南：语法，南大通用数据技术股份有限公司



## 版权声明



## 免责声明



## 通讯方式

南大通用数据技术股份有限公司

天津市高新区开华道 22 号普天创新产业园东塔 20-23 层

电话：400-013-9696　　邮箱：info@gbase.cn

## 商标声明

# 目　　录

# 关于本手册

本手册描述 GBase 8s 产品的 SQL 语言语句的语法、数据类型、表达式和 GBase 8s 内置函数。

本手册主要针对以下用户：

数据库用户

数据库管理员

数据库安全性管理员

数据库应用程序员

本手册假设您具有以下背景：

具有计算机、操作系统和操作系统提供的实用程序的工作知识。

使用过关系数据库或接触过数据库概念。

具有一些计算机编程经验

# 1. SQL 语法概述

本章提供了有关如何使用 SQL 语句、SPL 语句和语法段的信息。

本章组织为以下几节。

| 节 | 范围 |
| --- | --- |
| 如何输入 SQL 语句 | 如何使用语法图和描述以正确输入 SQL 语句 |
| 如何输入 SQL 注释 | 如何在 SQL 语句中输入注释 |
| SQL 语句的类别 | SQL 语句，按功能类别列出 |
| ANSI/ISO 的一致性和扩展 | SQL 语句，按与 ANSI/ISO 的一致性程度列出 |

## 1.1 如何输入 SQL 语句

SQL 语言是自由格式的（如同 C 或 PASCAL），通常忽略空格字符（例如 TAB 、LINEFEED 和各语句或语句元素之间额外的空格）。然而，必须至少有一个空白字符或其它定界符将关键字和标识与其它语法标记分开。

除了在带引号字符串中以外，SQL 是不区分大小写的；另见 标识符。在符合 ANSI 的数据库中，如果没有用双引号 (") 定界对象的 ***owner*** ，而且当初始化数据库服务器时没有将 **ANSIOWNER** 环境变量设置为 1 ，则数据库服务器会以大写字母存储 ***owner*** 名称。

在本手册中提供了语句描述以帮助您成功输入 SQL 语句。语句描述包括以下信息：

- 用来说明语句作用的简短介绍
- 用来显示如何正确输入语句的语法图
- 用来说明语法图中的每个输入参数的语法表
- 用法规则，通常带有说明这些规则的示例

对于某些语句，只为单独的子句提供了这些信息。

大多数语句描述的最后都带有本手册和其它手册中的相关信息的引用。

SQL 语句 提供了每个 SQL 语句的描述，是以字母顺序排列的。SPL 语句 使用相同格式描述了每个 SPL 语句。

输入 SQL 语句的主要帮助包括：

- 语法图和语法表的组合
- 出现在用法规则中的语法示例
- 对相关信息的引用

## 使用语法图和语法表

在您尝试使用本章中的语法图之前，先阅读介绍中的语法图一节会有所帮助。本节是理解语法图的关键，并说明了可出现在语法图中的元素和这些元素互连的路径。本节还包括说明典型语法图元素的示例。示例图后面的叙述显示了如何阅读该图以便成功输入语句。

语法图可以参考其他的语法部分或可以指定不同的限制。如果您正在使用应用程序接口（例如：ESQL/C），那么只有客户端应用和数据库服务器都支持的 SQL 语法规则是有效的。

当语法图包括输入规范（如标识、表达式、文件名、主机变量或其它项）时，语法图后跟着一个表，该表描述如何输入该项而不生成错误。每个语法表包含四列：

- **Element** 列列出了语法图中的每个变量项。
- **Description** 列简述了该项并标识缺省值（如果该项有缺省值的话）。
- **Restrictions** 列总结了该项的限制，例如值的可接受范围。（对于某些语法图，无法概括总结的限制显示在 **Usage** 注解中而不是在此列中。）
- **Syntax** 列指向给出该项的详细语法的 SQL 段。对于某些项（如主变量名称、路径名或文字字符），未提供页码索引。

这些图表通常提供给定的 SQL 语句中有效内容的直观注解，但对于某些语句，语法元素之间的相关性或限制仅在用法部分中的文本中作出标识。

## 使用示例

要理解某个语句的主要语法图和子图，请研究在每个语句的用法规则中显示的语法示例。这些示例有两个作用：

- 显示如何使用语句或子句完成特定任务
- 显示如何以具体的方法使用语句或其子句的语法

**提示：** 理解语法图的一个有效方法是，查找语法示例并将其与语法图中的关键字和参数作比较。通过将示例的具体元素映射到语法图的抽象元素，可以有效地理解和使用语法图。

对于在本手册示例中使用的约定的说明，请参阅介绍中的语法图的章节。

这些代码示例是用以说明有效语法的程序分段，而不是完整的 SQL 程序。在一些示例代码中，省略号（ . . . ）表示剩下的代码已省略。但为了节省空间，在程序分段的开头和结尾不显示省略号。

## 使用相关信息

为了帮助理解 SQL 语句描述中的概念和项，请查看每个语句结尾处的“相关信息”部分。

本部分指向本手册和其它手册中的相关信息，以帮助您理解讨论中的语句。本部分提供以下某些或全部信息。

- 相关语句的名称，这些相关语句可能包含词语中主题的更完整讨论

- 提供此语句中主题的展开讨论的其它手册的标题

**提示：** 如果您对 SQL 没有广泛的知识和经验，*GBase 8s SQL 教程指南*会给予您所需的基本 SQL 知识，供您理解并使用本手册中的语句描述。

## 1.2 如何输入 SQL 注释

您可以添加注释以阐明特定 SQL 语句的作用和影响。您还可以在程序开发期间使用注释符号以禁用个别语句，而无需从源代码中删除它们。

您的注释可帮助您或其他人理解程序、SPL 例程或命令文件中的语句角色。本手册中的代码示例有时包括阐明代码中的 SQL 语句角色的注释，但如果在写程序时经常使用注释，那么您自己的 SQL 程序将更容易阅读和维护。

下表显示了可在代码中输入的 SQL 注释指示符。这里列中的 ***Y*** 表示您可以将此符号与列标题中标识的产品或数据库类型一起使用。列中的 ***N*** 表示不能将此符号与指出的产品或指出的 ANSI 一致性状态的数据库一起使用。

| 注释符号 | ESQL/C | SPL 例程 | DB-Access | 符合 ANSI 的数据库 | 数据库不符合 ANSI | 描述 |
|---|---|---|---|---|---|---|
| 双连字符（--） | Y | Y | Y | Y | Y | 在单独一行中，双连字符放在注释的前面。要对多行做出注释，请将双连字符放在每个注释行的开头。 |
| 花括号 ( { . . . } ) | N | Y | Y | Y | Y | 花括号将注释括起来。{放在注释的前面}放在注释后面。花括号可以定界单行或多行注释，但不能嵌套注释。 |

| 注释符号 | ESQL/C | SPL 例程 | DB-Access | 符合 ANSI 的数据库 | 数据库不符合 ANSI | 描述 |
|---|---|---|---|---|---|---|
| 斜杠和星号 /* . . . */ | Y | Y | Y | Y | Y | C 语言样式的斜杠和星号( /* */ )成对定界符将注释括在其中。/* 放在注释前面，*/放在注释后面。这些符号可以定界单行或多行注释，但不能嵌套注释。 |

数据库服务器忽略注释中的字符。

优化程序伪指令 这一部分描述了上下文，在该上下文中注释内的信息可以影响 GBase 8s 的查询计划。

如果您使用的产品支持所有这些注释符号，则您选择的注释符号取决于对 ANSI/ISO 一致性的需求：

- 双连字符（ -- ）符合 SQL 的 ANSI/ISO 标准。
- 花括号（ { } ）是 ANSI/ISO 标准 的 GBase 8s 扩展。
- C 语言样式的斜杠和星号（ /* . . . */ ）符合 SQL-99 标准。

只要符合 ANSI/ISO 一致性，您对注释符号的选择纯属个人偏好。

在 DB-Access 中，当您使用 SQL 编辑器输入 SQL 语句和当您使用 SQL 编辑器或系统编辑器创建 SQL 命令文件时，可以使用这些注释符号中的任何符号。

SQL 命令文件是包含一条或多条 SQL 语句的操作系统文件。命令文件也称为命令脚本。关于命令文件的更多信息，请参阅 *GBase 8s SQL 教程指南* 中命令脚本的讨论。关于如何在 DB-Access 中使用 SQL 编辑器或系统编辑器创建和修改命令文件的信息,请参阅 *GBase 8s DB-Access 用户指南* 。

您可以在 SPL 例程的任意一行中使用任意一种注释符号。请参阅 *GBase 8s SQL 教程指南* 中关于如何注释和记录 SPL 例程的讨论。

在 GBase 8s ESQL/C 中，以连字符（--）开始的注释可以延伸到同一行的末尾。关于 GBase 8s ESQL/C 程序中特定于语言的注释符号的信息，请参阅 *GBase 8s ESQL/C 程序员手册* 。

## SQL 注释符号示例

这些示例说明了使用 SQL 注释指示符的不同方式。

以下示例使用了每一种注释符号，包括双连字符（--）、花括号（{}）、C 语言样式（/*...*/）的注释符号在 SQL 语句后注释。该注释与语句显示在同一行。

```
SELECT * FROM customer; -- Selects all columns and rows
SELECT * FROM customer; {Selects all columns and rows}
SELECT * FROM customer; /*Selects all columns and rows*/
```

以下三个示例与前面的示例使用了相同的 SQL 语句和相同的注释，但注释自成一行：

```
SELECT * FROM customer;
        -- Selects all columns and rows
SELECT * FROM customer;
        {Selects all columns and rows}
SELECT * FROM customer;
        /*Selects all columns and rows*/
```

以下示例中，用户输入与前面的示例中相同的 SQL 语句，但现在输入一条多行注释（或者对于双连字符有两条注释）：

```
SELECT * FROM customer;
    -- Selects all columns and rows
    -- from the customer table

SELECT * FROM customer;
        {Selects all columns and rows
        from the customer table}

SELECT * FROM customer;
        /*Selects all columns and rows
        from the customer table*/
```

一条 SQL 语句中出现任意三种样式的注释：

```
SELECT *            -- Selects all columns and rows
    FROM customer;  -- from the customer table

SELECT *            {Selects all columns and rows}
    FROM customer;  {from the customer table}

SELECT *            /*Selects all columns and rows*/
    FROM customer;  /*from the customer table*/
```

如果您使用花括号或 C 语言样式的注释被成对的开始和结束符号定界，那么结束注释符号必须与开始注释符号的样式相同。

### SQL 注释中的非 ASCII 字符

如果数据库语言环境支持非 ASCII 字符（包括多字节字符），则可以在 SQL 注释中输入非 ASCII 字符。关于 SQL 注释 GLS 方面的进一步信息，请参阅 *GBase 8s GLS 用户指南*。

## 1.3 SQL 语句的类别

传统上将 SQL 语句划分为以下逻辑类别：

**数据定义语句**

这些数据定义语言（DDL）语句可声明、重命名、修改或破坏数据库对象。

**数据操纵语句**

这些数据操纵语言（DML）语句可检索、插入、删除或修改数据值。

**游标操纵语句**

这些语句可声明、打开和关闭游标，游标是用于对多行数据操作的数据结构。

**动态管理语句**

这些语句支持内存管理并运行用户在运行时指定 DML 操作的详细信息。

**数据访问语句**

这些语句指定访问特权并支持多个用户对数据库的同时访问。

**数据完整性语句**

这些语句执行事务日志记录并支持数据库的参照完整性。

**优化语句**

这些语句可用于提高数据库操作的性能。

**例程定义语句**

这些语句可声明、定义、修改、执行或破坏数据库存储的用户定义的例程。

**客户机/服务器连接语句**

这些语句可打开或关闭数据库和客户机应用程序之间的连接。

**辅助语句**

这些语句可提供关于数据库的信息。（这也是一个剩余类别，用于不与其他语句紧密相关的语句。）

### 数据定义语言语句

SQL 数据定义语言（DDL）创建、修改、重命名或销毁数据库对象并对数据库中系统目录表中的行进行相应的修改。

- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER FUNCTION
- ALTER INDEX
- ALTER PROCEDURE
- ALTER ROUTINE
- ALTER SEQUENCE
- ALTER SECURITY LABEL COMPONENT
- ALTER TABLE
- ALTER TRUSTED CONTEXT
- ALTER USER
- CLOSE DATABASE
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE CAST
- CREATE DATABASE
- CREATE DEFAULT USER
- CREATE DISTINCT TYPE
- CREATE EXTERNAL TABLE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE GLOBAL TEMPRORY TABLE
- CREATE INDEX
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROLE
- CREATE ROUTINE FROM
- CREATE ROW TYPE
- CREATE SCHEMA
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- CREATE SEQUENCE
- CREATE SYNONYM
- CREATE TABLE
- CREATE TEMP TABLE
- CREATE TRIGGER
- CREATE TRUSTED CONTEXT
- CREATE USER
- CREATE VIEW
- CREATE MATERIALIZED VIEW
- CREATE XADATASOURCE
- CREATE XADATASOURCE TYPE
- DROP ACCESS_METHOD

- DROP AGGREGATE
- DROP CAST
- DROP DATABASE
- DROP FUNCTION
- DROP INDEX
- DROP OPCLASS
- DROP PROCEDURE
- DROP ROLE
- DROP ROUTINE
- DROP ROW TYPE
- DROP SECURITY
- DROP SEQUENCE
- DROP TRUSTED CONTEXT
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER
- DROP TYPE
- DROP USER
- DROP VIEW
- DROP MATERIALIZED VIEW
- DROP XADATASOURCE
- DROP XADATASOURCE TYPE
- RENAME COLUMN
- RENAME DATABASE
- RENAME INDEX
- RENAME SECURITY
- RENAME SEQUENCE
- RENAME TABLE
- RENAME TRUSTED CONTEXT
- RENAME USER
- TRUNCATE
- UPDATE STATISTICS

## 数据操纵语言语句

- DELETE
- INSERT
- LOAD
- MERGE
- SELECT
- UNLOAD
- UPDATE

**注：** SQL 的 ANSI/ISO 标准中的 DML 语句包括 DELETE 、INSERT 、MERGE 、SELECT 和 UPDATE 语句。MERGE 可模拟 INSERT 和 DELETE 或 UPDATE 。尽管 LOAD 和 UNLOAD 在功能上类似于 DML ，但这些 DB-Access 宏不在本手册中对“DML 语句”的大多数引用范围内。

## 数据完整性语句

- BEGIN WORK
- COMMIT WORK
- SAVEPOINT
- RELEASE SAVEPOINT
- ROLLBACK WORK
- SET Database Object Mode
- SET LOG
- SET Transaction Mode
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE

## 游标操纵语句

- CLOSE
- DECLARE
- FETCH
- FLUSH
- FREE
- OPEN
- OPEN FOR
- PUT
- SET AUTOFREE

## 动态管理语句

- ALLOCATE COLLECTION
- ALLOCATE DESCRIPTOR
- ALLOCATE ROW
- DEALLOCATE COLLECTION
- DEALLOCATE DESCRIPTOR
- DEALLOCATE ROW
- DESCRIBE
- DESCRIBE INPUT
- EXECUTE
- EXECUTE IMMEDIATE
- FREE
- GET DESCRIPTOR
- INFO
- PREPARE
- SET DEFERRED_PREPARE
- SET DESCRIPTOR
- REFRESH MATERIALIZED VIEW

## 数据访问语句

- GRANT
- GRANT FRAGMENT

- LOCK TABLE
- REVOKE
- REVOKE FRAGMENT
- SET ISOLATION
- SET LOCK MODE
- SET ROLE
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- SET Transaction Mode
- UNLOCK TABLE

## 优化语句

- SAVE EXTERNAL DIRECTIVES
- SET ENVIRONMENT
- SET EXPLAIN
- SET OPTIMIZATION
- SET PDQPRIORITY
- SET STATEMENT CACHE

## 例程定义语句

- ALTER FUNCTION
- ALTER PROCEDURE
- ALTER ROUTINE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DROP FUNCTION
- DROP PROCEDURE
- DROP ROUTINE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- SET DEBUG FILE TO

## 辅助语句

- GET DIAGNOSTICS
- INFO
- OUTPUT
- SET COLLATION
- SET DATASKIP
- SET ENCRYPTION PASSWORD
- SET USER PASSWORD
- WHENEVER

### 客户机/服务器连接语句

- CONNECT
- DATABASE
- DISCONNECT
- SET CONNECTION

## 1.4 ANSI/ISO 的一致性和扩展

以下列表显示了在入门级别上符合 ANSI SQL-92 标准的语句、符合 ANSI 但包括 GBase 8s 扩展的语句和 ANSI/ISO 标准的 GBase 8s 扩展语句。

### 符合 ANSI/ISO 的语句

- CLOSE
- COMMIT WORK
- RELEASE SAVEPOINT
- SET CONSTRAINTS（请参阅 SET Transaction Mode 语句）
- SET SESSION AUTHORIZATION
- SET TRANSACTION

### 具有 GBase 8s 扩展的符合 ANSI/ISO 的语句

- ALLOCATE DESCRIPTOR
- ALTER TABLE
- CONNECT
- COMMENT
- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE TRIGGER
- CREATE SCHEMA
- CREATE TABLE
- CREATE TEMP TABLE
- CREATE VIEW
- DEALLOCATE DESCRIPTOR
- DECLARE
- DELETE
- DESCRIBE
- DESCRIBE INPUT
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- GET DESCRIPTOR
- GET DIAGNOSTICS
- GRANT
- INSERT
- MERGE

- OPEN
- PREPARE
- REVOKE
- ROLLBACK WORK
- SAVEPOINT
- SELECT
- SET CONNECTION
- SET DESCRIPTOR
- UPDATE STATISTICS
- WHENEVER

## ANSI/ISO 标准的扩展语句

- ALLOCATE COLLECTION
- ALLOCATE ROW
- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER FUNCTION
- ALTER INDEX
- ALTER PROCEDURE
- ALTER ROUTINE
- ALTER SECURITY LABEL COMPONENT
- ALTER SEQUENCE
- ALTER TRUSTED CONTEXT
- ALTER USER
- BEGIN WORK
- CLOSE DATABASE
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE CAST
- CREATE DATABASE
- CREATE DEFAULT USER
- CREATE DISTINCT TYPE
- CREATE EXTERNAL TABLE
- CREATE FUNCTION FROM
- CREATE INDEX
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE PROCEDURE FROM
- CREATE ROLE
- CREATE ROUTINE FROM
- CREATE ROW TYPE
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- CREATE SEQUENCE
- CREATE SYNONYM

- CREATE TRUSTED CONTEXT
- CREATE USER
- CREATE XADATASOURCE
- CREATE XADATASOURCE TYPE
- DATABASE
- DEALLOCATE COLLECTION
- DEALLOCATE ROW
- DROP ACCESS_METHOD
- DROP AGGREGATE
- DROP CAST
- DROP DATABASE
- DROP FUNCTION
- DROP INDEX
- DROP OPCLASS
- DROP PROCEDURE
- DROP ROLE
- DROP ROUTINE
- DROP ROW TYPE
- DROP SECURITY LABEL
- DROP SECURITY LABEL COMPONENT
- DROP SECURITY POLICY
- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER
- DROP TRUSTED CONTEXT
- DROP TYPE
- DROP USER
- DROP VIEW
- DROP XADATASOURCE
- DROP XADATASOURCE TYPE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- FLUSH
- FREE
- GRANT FRAGMENT
- LOAD
- LOCK TABLE
- OUTPUT
- PUT
- RELEASE SAVEPOINT
- RENAME COLUMN
- RENAME DATABASE
- RENAME INDEX
- RENAME SECURITY LABEL
- RENAME SECURITY LABEL COMPONENT

- RENAME SECURITY POLICY
- RENAME SEQUENCE
- RENAME TABLE
- RENAME TRUSTED CONTEXT
- RENAME USER
- REVOKE FRAGMENT
- SAVE EXTERNAL DIRECTIVES
- SET AUTOFREE
- SET COLLATION
- SET CONSTRAINTS （请参阅 SET Database Object Mode 语句）
- SET Database Object Mode
- SET DATASKIP
- SET DEBUG FILE TO
- SET DEFERRED_PREPARE
- SET ENCRYPTION PASSWORD
- SET ENVIRONMENT
- SET EXPLAIN
- SET INDEXES （请参阅 SET Database Object Mode 语句）
- SET ISOLATION
- SET LOCK MODE
- SET LOG
- SET OPTIMIZATION
- SET PDQPRIORITY
- SET ROLE
- SET STATEMENT CACHE
- SET TRIGGERS （请参阅 SET Database Object Mode 语句）
- SET USER PASSWORD
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE
- TRUNCATE
- UNLOAD
- UNLOCK TABLE
- UPDATE STATISTICS

# 2. SQL 语句

本章节描述了由 GBase 8s 识别的 SQL 语句的语法和定义。

这些 SQL 语句描述按字母顺序显示。

对于某些语句，重要的语义详细信息显示在此文档集合的其它卷中，如交叉引用所指出的。

对于许多语句、语法图和/或紧跟语法图后的项表，包括了对 数据类型和表达式 或 其它语法段 中的语法段的引用。

当 SQL 语句的名称包括小写字符（如“ SET Database Object Mode ”）时。表示语句名称中的第一个大小写混合的字符串不是 SQL 关键字，而是表示两个或更多个不同的 SQL 关键字可以跟在前面的大写关键字后面。

对于语句描述的结构说明，请参阅 SQL 语法概述 。

## 2.1 ALLOCATE COLLECTION 语句

使用 ALLOCATE COLLECTION 语句为集合数据类型的变量（例如 LIST 、MULTISET 或 SET ）或未归类的集合变量分配内存。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *variable* | 要分配的已归类的或未归类的集合变量的名称 | 必须为未分配的 GBase 8s ESQL/C 集合类型主变量 | 特定于语言的名称规则 |

**用法**

该语句是 SQL ANSI/ISO 标准的扩展。在 ESQL/C 中使用此语句。

ALLOCATE COLLECTION 语句为可以存储 collection 数据类型的值的 ESQL/C 变量分配内存。

要为 GBase 8s ESQL/C 程序创建集合变量：

1. 在 GBase 8s ESQL/C 程序中作为客户机集合变量声明集合变量。

   集合变量可以是已归类或未归类的集合变量。
2. 使用 ALLOCATE COLLECTION 语句为集合变量分配内存。

如果分配内存成功，ALLOCATE COLLECTION 语句会将 **SQLCODE** （也就是 **sqlca.sqlcode**）置零（0）；如果失败，会将其设置成一个负数错误码。

当不再需要集合变量时，您必须使用 DEALLOCATE COLLECTION 语句显示地释放内存。在 DEALLOCATE COLLECTION 语句执行成功后，您可以重新使用该集合变量。

**提示：** ALLOCATE COLLECTION 语句仅为 GBase 8s ESQL/C 集合变量分配内存。要为 GBase 8s ESQL/C 行变量分配内存，请使用 ALLOCATE ROW 语句。

**示例**

以下示例显示如何使用 ALLOCATE COLLECTION 语句为未归类的集合变量 a_set 分配资源：

```
EXEC SQL BEGIN DECLARE SECTION;
client collection a_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_set;
```

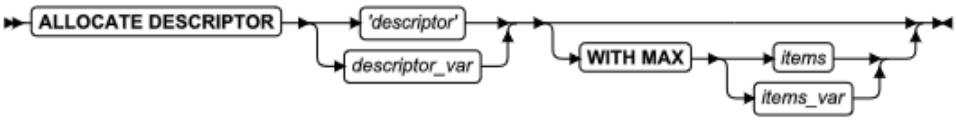
以下示例显示如何使用 ALLOCATE COLLECTION 语句为已归类的集合变量 a_typed_set 分配资源：

```
EXEC SQL BEGIN DECLARE SECTION;
client collection set(integer not null) a_typed_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_typed_set;
```

## 2.2 ALLOCATE DESCRIPTOR 语句

使用 ALLOCATE DESCRIPTOR 语句为系统描述符区域（SDA）声明和分配内存。在 ESQL/C 中使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 未分配的系统描述符区域的名称 | 包含在单引号（’）中。在 SDA 名称中必须是唯一的 | 引用字符串. |
| *descriptor_var* | 用来存储系统描述符区域名称的主机变量 | 必须包含未分配系统描述符区域的名称 | 特定于语言 |
| *items* | *descriptor* 中项描述符的数目。缺省值为 100 。 | 必须是大于零的无符号 INTEGER | 精确数值 |
| *items_var* | 包含项目数目的主变量 | 数据类型必须为 INTEGER 或 SMALLINT | 特定于语言 |

**用法**

ALLOCATE DESCRIPTOR 语句创建新的*系统描述符区域*，该区域是内存中的一个位置，存放 DESCRIBE 语句可以显示的信息，或存放关于查询的 WHERE 子句的信息。

系统描述符区域（SDA）包含一个或多个称为 *item descriptors* 的字段。每个项描述符都有一个数据库服务器可以接收或发送的数据值。项描述符也包含关于该数据的信息，例如数据类型、长度、小数位、精度和可以为 NULL 值。

系统描述符区域存放 DESCRIBE ... USING SQL DESCRIPTOR 语句获取的信息，或者存放关于一个动态执行语句中的 WHERE 子句的信息。

如果您分配一个系统描述符区域的名称与一个现有系统描述符区域的名称相同，则数据库服务器返回一条错误消息。如果您使用 DEALLOCATE DESCRIPTOR 语句释放了该描述符，则 ALLOCATE DESCRIPTOR 语句可以重新使用同样的描述符名称。

### WITH MAX 子句

您可以使用 WITH MAX 子句标识您需要的项描述符的最大数目。

当您使用此子句时，COUNT 字段设置为您指定的 ***items*** 数目。如果您不指定 WITH MAX 子句，COUNT 字段的缺省值为 100 。您可以使用 SET DESCRIPTOR 语句更改 COUNT 字段的值。

ALLOCATE DESCRIPTOR 语句示例

以下示例显示了有效的 ALLOCATE DESCRIPTOR 语句。每个示例都包含 WITH MAX 子句。此示例使用嵌入的变量名称标识系统描述符区域，并指定所需的项描述符：

```
EXEC SQL allocate descriptor :descname with max :occ;
```

下一示例使用加引号的字符串 desc1 作为系统描述符的标识，并且使用无符号整数 3 指定该 desc1 区域中所需的项描述符的最大数目：

```
EXEC SQL allocate descriptor 'desc1' with max 3;
```

## 2.3 ALLOCATE ROW 语句

使用 ALLOCATE ROW 语句为 *row* 变量分配内存。该语句是 SQL ANSI/ISO 标准的扩展。在 ESQL/C 中使用此语句。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *variable* | 要分配的已归类或未归类的 *row* 变量名称 | 必须为未分配的 GBase 8s ESQL/C row 类型主变量 | 特定于语言 |

**用法**

ALLOCATE ROW 语句为存储 row 类型数据的主变量分配内存。要创建 **row** 变量，ESQL/C 程序必须执行以下操作：

1. 声明 *row* 变量。row 变量可以是已归类或未归类的 *row* 变量。
2. 使用 ALLOCATE ROW 语句为 *row* 变量分配内存。

以下示例显示如何使用 ALLOCATE ROW 语句为已归类的 *row* 变量 a_row 分配资源：

```
EXEC SQL BEGIN DECLARE SECTION;
row (a int, b int) a_row;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate row :a_row;
```

如果内存分配操作成功，则 ALLOCATE ROW 语句会将 **SQLCODE**（**sqlca.sqlcode** 的内容）置零（0）；如果分配失败，会将其设置为一个负的错误码。

您必须使用 DEALLOCATE ROW 语句显示地释放内存。一旦您使用 DEALLOCATE ROW 语句释放了该 *row* 变量，您就可以重新使用该 *row* 变量。

**提示：** ALLOCATE ROW 语句仅为 GBase 8s ESQL/C *row* 变量分配内存。要为 GBase 8s ESQL/C *collection* 变量分配内存，请使用 ALLOCATE COLLECTION 语句。

当您在多次函数调用中使用同一 *row* 变量而未对其解除分配时，会导致客户机计算机上的内存泄露。因为没有办法确定指针在传递时是否是有效的，所以 GBase 8s ESQL/C 会假设它是无效的，并将其分配到新的内存位置。

## 2.4 ALTER ACCESS_METHOD 语句

可以使用 ALTER ACCESS_METHOD 语句更改一个或多个 **sysams** 系统目录表中用户定义的主或从的存取方法的属性。

**语法**

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *access_method* | 要更改的存取方法的名称 | 存取方法必须由先前的 CREATE ACCESS_METHOD 语句在 **sysams** 系统目录表中注册 | 标识符 |
| *owner* | 存取方法所有者的名称 | 必须拥有该存取方法 | 所有者名称 |
| *purpose_keyword* | 表示要更改的特征的关键字 | 关键字必须通过先前的 CREATE 或 ALTER ACCESS_METHOD 语句和存取 | 用途函数、标志和值 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 方法相关联 | |

**用法**

该语句是 SQL ANSI/ISO 标准的扩展。该语句无法修改内置的存取方法。

使用 ALTER ACCESS_METHOD 更改用户定义存取方法的定义。您无法修改内置的存取方法。

您必须是该存取方法的所有者或具有修改用户定义存取方法的 DBA 特权。在符合 ANSI 的数据库中，如果另一个用户是该存取方法的所有者，那么 DBA 必须限定该存取方法的名称。

当更改存取方法时，您同时更改了定义该存取方法的目的选项规范（目的函数、目的标志或目的值）。例如，您可以更改一个存取方法以分配一个新的用户定义的函数或方法名称，或为一个表的扫描成本提供乘数。

如果事务正在处理中，则数据库服务器将等待修改存取方法，直接提交或回滚该事务。该事务完成之前，其他任何用户都无法执行该存取方法。

**示例**

以下语句更改了 remote 用户定义的存取方法：

```
ALTER ACCESS_METHOD remote
ADD am_scancost = FS_scancost,
ADD am_rowids,
DROP am_getbyid,
MODIFY am_costfactor = 0.9;
```

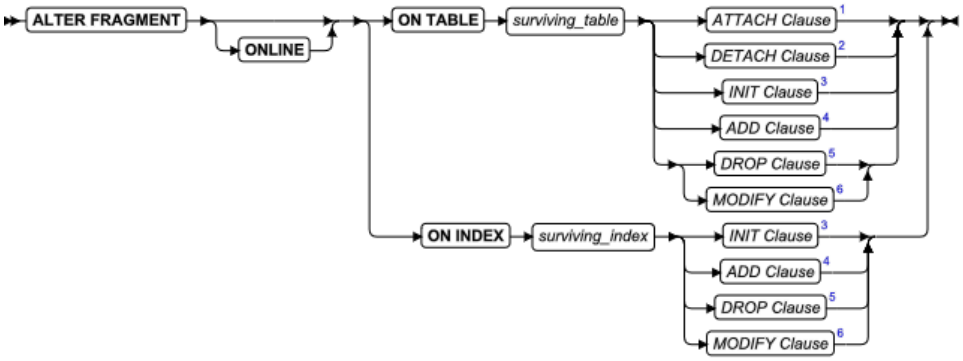
上述示例将对该存取方法进行以下修改：

- 添加一个称为 **FS_scancost( )** 的用户定义的函数或方法，它在 **sysams** 表中与 **am_scancost** 关键字向关联。
- 设置（添加） **am_rowids** 标记
- 删除与 **am_getbyid** 关键字相关联的用户定义的函数或方法
- 修改 **am_costfactor** 值

## 2.5 ALTER FRAGMENT 语句

可以使用 ALTER FRAGMENT 语句更改现有表或索引的分布策略或存储位置。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *surviving_index* | 用于修改分布或存储的索引 | 当该语句执行时必须存在 | 标识符 |
| *surviving_table* | 用于修改分布或存储的表 | 必须存在。请参阅 对 ALTER FRAGMENT 语句的限制 | 标识符 |

**用法**

ALTER FRAGMENT 语句仅适用于位于当前站点的表分片或索引分片。不会存储或更新任何远程信息。

要更改一个表的分片存储策略，您必须具有 Alter 或 DBA 特权。要更改一个索引的分段存储策略，您必须具有 Index 或 DBA 特权。

**Attention：** 此语句可能导致索引被删除或重建。执行更改操作之前，请仔细阅读 *GBase 8s 性能指南* 中的相应章节以查看影响和策略。

ALTER FRAGMENT 语句的子句支持以下任务。

| 子句 | 作用 |
|---|---|
| ATTACH | 将两个或多个具有相同模式的表组合到一个分片表中 |
| DETACH | 将一个表分片从分片存储策略中拆离，并将其置于一个新表中 |
| INIT | 提供以下选项：<br>• 定义并初始化一个表上的分片存储策略<br>• 更改对分片表达式求值的顺序<br>• 更改表或索引的分片存储策略<br>• 更改现有表的存储位置<br>• 将数据从现有的表分片移到另一个新的分片表中<br>• 更改数据库给表或索引生成的分片存储位置<br>• 更改表或索引的分片键或分片表达式 |
| ADD | 将另一个分片添加到一个现有分片存储列表 |
| DROP | 从一个分片存储列表删除一个现有分片 |

移除一个或多个创建内部分片的 dbspace 列表中的 dbspaces

MODIFY 更改现有区间、列表或基于表达式的分片表达

将现有的分片移动到不同的 dbspace 中去

用新的列表替换创建区间分片的 dbspace 当前列表

启用或禁用自动创建区间分片

使用 CREATE TABLE 语句或 ALTER FRAGMENT 语句的 INIT 子句来创建分片表。

在 **gspaces** 实用程序成功重命名 dbspace 后，只有新的名称能够引用重命名后的 dbspace。然而，表或索引的现有分片存储策略是由数据库服务器自动更新的，以使用新的 dbspace 名称替换旧的名称。您不需要采取任何额外的操作更新使用旧的 dbspace 名称定义的分布策略或存储位置，但是如果您要在一个 ALTER FRAGMENT 或 ALTER TABLE 语句中引用该 dbspace ，您必须使用新的名称。

如果您忽略可选 ONLINE 关键字，ALTER FRAGMENT 操作需要在参与该操作的所有表上都放置独占存取和独占锁。如果您启用 FORCE_DDL_EXEC 会话环境选项，那么您可以强制已打开的参与 ALTER FRAGMENT ON TABLE 操作的表或任一已放置锁的表的其他事务退出。如果服务器无法获得该表的独占访问和独占锁，那么服务器将会回滚已打开或表中已有锁的事务，直到满足 FORCE_DDL_EXE 选项指定的值。（有关更多信息，请参阅 FORCE_DDL_EXEC 环境选项 。）

## 对 ALTER FRAGMENT 语句的限制

您无法将 ALTER FRAGMENT 语句用于临时表、视图、或未在当前数据库注册的表。

如果您的表或索引尚未分片，则您可以使用的子句仅有 ATTACH 和 INIT 。

您无法将 ALTER FRAGMENT 用于属于表层次结构的类型表。

## ALTER FRAGMENT 和事务日志记录

如果您的数据库支持事务日志记录，ALTER FRAGMENT 将在一个单一事务内执行。如果分片存储策略使用大量记录，可能会耗尽日志空间或磁盘空间。（要修改分片存储策略，数据库服务器需要额外的磁盘空间，它将随后释放这些磁盘空间。）

如果您耗尽日志空间或磁盘空间，请尝试以下过程之一以减少您的日志空间或磁盘空间需求：

- 关闭日志记录，并在操作结束时将其重新打开。此过程间接地要求备份 **root** dbspace 。
- 将这些操作分割为多个 ALTER FRAGMENT 语句，每次仅移动较小部分的记录。

关于日志空间需求和磁盘空间需求的信息，请参阅 *GBase 8s 管理员指南* 。该指南也包含关于如何关闭日志记录的详细指示信息。关于备份的信息，请参阅 *GBase 8s 备份与恢复指南*。

## 决定分片中的行数

Dbspace 允许多少行，您就可以将多少行放入分片。

要查出一个分片中的行数：

1. 对该表运行 UPDATE STATISTICS FORCED 语句。此步骤会使用当前表的信息填充 **sysfragments** 系统目录表。
2. 查询 **sysfragments** 系统目录表以检查 **npused** 和 **nrows** 值。**npused** 列向您提供分片中使用的数据页数；**nrows** 字段向您提供分片中的行数。

## ALTER FRAGMENT 操作中的 ONLINE 关键字

ONLINE 关键字指示数据库服务器修改后台中表的存储，并且其它并发用户仍可以继续存取该表。

通过在 ALTER FRAGMENT 语句中使用 ONLINE 关键字，DBA 可以降低非独占存取错误的风险，可以提高分片表的可用性。该指示数据库服务器在内部 ATTACH 、DETACH 和 MODIFYT 操作以提交工作，如果没有错误，它将在该表上应用一个内部意向排他锁而非排他锁。

在 DETACH 和 MODIFY 操作中，在以下条件下，ONLINE 关键字可以降低 -710 错误的风险：

- AUTO_REPREPARE 配置参数设置为 1 ，
- IFX_ AUTO_REPREPARE 会话环境变量设置为 1 。

应用 ALTER FRAGMENT ONLINE FOR TABLE 语句有以下限制：

- ALTER FRAGMENT ONLINE 只有 ATTACH 、DETACH 和 MODIFY 选项是有效的。
- FOR TABLE 子句必须指定由范围区间架构分片的表。
- 正在修改的表不能被 LOCK TABLE 语句显示地锁定。
- ALTER FRAGMENT ONLINE 必须是该事务中首个修改任一数据库对象或表的语句。
- 在同一事务中 ALTER FRAGMENT ONLINE 语句后不能出现修改数据库中对象的操作。

## 自动重命名区间分片标识符

一些 ALTER FRAGMENT 操作可以更改分片表中现有的区间分片位置的顺序。在这些情况下，数据库服务器会自动修改受影响的区间分片的系统定义的名称。

对于由区间分片方案分区的表，添加、删除、附加或拆离分片或修改表的转换值的 ALTER FRAGMENT 操作可以更改现有的间隔分片的 **sysfragments.evalpos** 值，或者可以将间隔分片更改为范围分片。为了避免创建具有与 ALTER FRAGMENT 语句在分片列表中重新定位的间隔分片相同的系统生成的名称的新的间隔分片，数据库服务会自动使用与标识符名称不匹配的新标识符替换初始系统定义的名称。

以下一般的规则适用于系统生成的范围和区间分片名称：

- 对于区间分片：sys_**evalpos**
- 对于范围分片：sys_**evalpos**rg

此处 evalpos 是 sysfragments.evalpos 的数值（初始值），其中 0 是是指分片列表中第一个分片的 evalpos 值。

在重命名分片期间，当使用新的 **partition** 名称更改 **sysfragments** 系统目录表时将会在此分片上放置一个互斥锁， 并且对于初始位置在分片列表中的分片的新的 **evalpos** 值将会在 ALTER FRAGMENT 操作期间变更。

在创建新的区间分片时，要必须声明非唯一的分片名称，数据库服务器只能重命名在 ALTER FRAGMENT 操作中系统生成的重定位的区间分片的标识符。用户定义的重定位分片的标识符不会自动重命名。

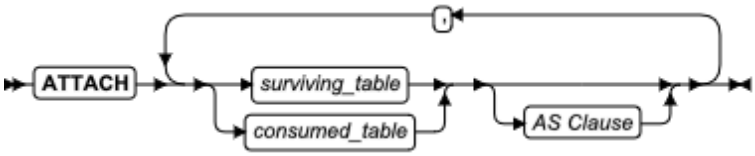
如果您希望在 ALTER FRAGMENT ONLINE ATTACH 语句执行期间或对使用区间分片表执行其它 ALTER FRAGMENT 操作时避免现有的分片的重命名，则您可以首先使用 ALTER FRAGMENT MODIFY 语句用用户定义的名称重命名这些区间分片，其它系统生成的名称可以由 ALTER FRAGMENT 操作更改。用户定义的分片名称不能以字符串 sys_ 开头。

## ATTACH 子句

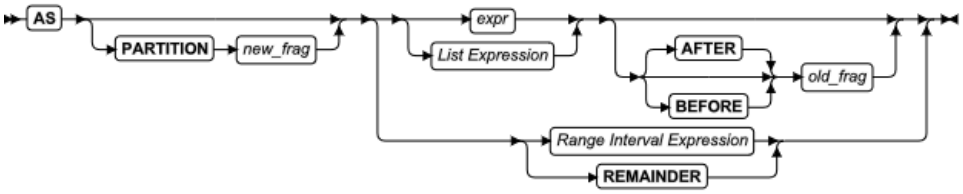
使用 ALTER FRAGMENT ON TABLE 语句的 ATTACH 子句将拥有相同结构的表合并到一个分片存储策略中。

例如，您可以使用该语法，将一个已从表分离的分片组合到一个具有相同分布存储结构的档案表中。

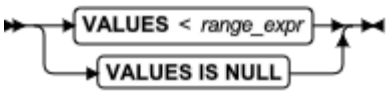
ATTACH 子句



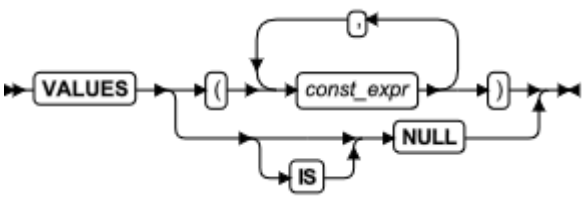

AS 子句




范围间隔表达式




列表表达式



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *const_expr* | 定义分片存储列表的常量表达式 | 必须是带引号的字符串或文字值。在同一对象的分片中，表中的值必须唯一。 | 常量表达式 |
| *consumed_table* | 要同 *surviving_table* 合并而失去身份的表 | 结构必须与 *surviving_table* 相匹配。不能保护连续列或唯一约束、引用约束或主键约束。另见 ATTACH 子句的一般限制 | 标识符 |
| *expr* | 定义通过表达式分片的表的分片中存储哪些行的表达式 | 仅包含当前表中的列以及单一行中的数据值。另见 ATTACH 子句的一般限制. | 条件；表达式 |
| *new_frag* | 此处声明的 *consumed_table* 分片的名称。缺省值为 dbspace 名称 | 在 *surviving_table* 分片的名称中必须唯一。 | 标识符 |
| *old_frag* | 含有 *surviving_table* 分片的分区或 dbspace | 必须存在。不能是区间或间隔分片。 | 标识符 |
| *range _expr* | 定义存储在分片中分片密钥的上限的常量表达式 | 必须是数字的常量文字表达式 、DATETIME 或与分片密钥表达式兼容的 DATE 数据类型。 | 常量表达式 |
| *surviving_table* | 要修改分布或存储位置的表 | 必须存在。没有任何约束。另见对 ALTER FRAGMENT 语句的限制。 | 标识符 |

当新的表达式分布被连接到由列表或区间间隔分片的表，死表上的数据和活表上的受影响的分片会被扫描并移动到合适的分区，因为这些策略没有重叠。

如果启用了自动更改分布策略模式，并且表连接到分片分布策略，数据库服务器会计算新分片的分布策略。现有分片的旧的分布策略在此时也将重新计算。分片统计的重运算在后台执行。在数据库服务器结束分片统计计算后，它从表分布策略中合并这些分片，并将结果村春在系统目录中。

要使用此子句，您必须具有 DBA 特权或您是指定表的所有者。ATTACH 子句支持以下任务：

- 通过合并两个或两个以上的相同结构的未分片的表创建单独一个分片表
  （请参阅 将多个未分片表合并以创建一个分片表）
- 将一个或多个表连接到一个分片表
  （请参阅 将一个表连接到一个分片表）

## ATTACH 子句的一般限制

此子句在 ALTER FRAGMENT ON INDEX 语句中无效。

您连接的任何表必须先前已在独立的分区中创建。您不能将同一个表连接多次。

ATTACH 子句中所列的所有死表（consumed table）必须具有同活表（surviving table）相同的结构。列的数目、名称、数据类型和相对位置必须相同。

***expression*** 不能包含聚集、子查询、或变体函数。

### *ATTACH 子句的其他限制*

对 ROW 类型列字段的用户定义的例程和引用是无效的。您不能将一个分片表连接到另一个分片表。

所有存储分片的 dbspace 必须拥有相同的页面大小。

对两个分片表的 ATTACH 操作无法产生按区间或列表分片的活表（GBase_8t surviving table）。（如果您要连接两个非分片的表，使用 ALTER FRAGMENT 的 INIT 操作为其中一个非分片表定义其区间或列表分片结构，然后使用 ATTACH 选项连接第二个表。）

对于按区间分片的活表（surviving tables），有以下限制：

- 由于数据库服务器决定区间分片的初始位置，所以 BEFORE 和 AFTER 指定无效。
- 您无法连接表达式符合现有区间分片表达式的分片。
- 当连接的分片超过事务值，要连接的分片的上限必须位于区间分片的界限。就是说，分片的上限值必须等于事务值乘以区间值的整数倍。

对于受同一安全策略保护的分片表，如果以下任一条件不满足，那么连接分片到表就会是失败：

- 源表和目标表都受同一安全策略的保护；
- 两个表都具有相同的保护粒度（是行级别或列级别或都具有行级别和列级别）；
- 在两个表中，受保护的列的相同设置是由相同的安全标签所保护。如果有多余的受保护列，每个表就会有多于一个安全标签，但是该相同的标签必须保护两个表中的相同的列。

如果由于不符合以上任一条件而使 ATTACH 操作失败的话，您可以使用 ALTER TABLE 语句让两个表的模式相同， 然后对其重复 ALTER FRAGMENT ATTACH 语句。

只有持有 DBSECADM 角色的用户才能引用 ALTER FRAGMENT 语句中受保护的表。

## 使用 BEFORE 、AFTER 和 REMAINDER 选项

BEFORE 和 AFTER 选项允许您在现有分片之前或之后放入新的分片。当分布方案为循环或区间间隔时，您不能使用 BEFORE 和 AFTER 选项。

当您连接新的列表或表达式的分片而没有显式地使用 BEFORE 或 AFTER 关键字选项时，数据库服务器会将所添加的分片置于分片存储列表的末尾，除非存在一个余项分片。如果存在一个余项分片，则新分片会刚好置于该余项分片前。您不能在余项分片之后连接一个新分片。

当分布方案是循环或区间间隔时，您无法定义余项分片。

如果您省略了 AS PARTITION *分片*规范，该分片的名称就是存储它的 dbspace 的名称。如果同一表的另一个分片已经具有其 dbspace 的名称，那么数据库服务器会声明异常，并且 ALTER FRAGMENT ATTACH 操作失败。

## 将多个未分片表合并以创建一个分片表

当您具有相同表结构的表转换为单独一个表中的分片时，您是允许数据库服务器管理分片存储，而不是允许应用程序管理分片存储。分布方案可以是循环的或急于表达式的。

要从两个或两个以上相同结构的未分片表创建单独一个分片表，ATTACH 子句必须包含***连接列表***中的活表。连接列表是 ATTACH 子句中表的列表。

要在新创建的单独一个分片表中包含 **rowid** ，请首先连接所有表，然后使用 ALTER TABLE 语句添加 **rowid**。

## 将一个表连接到一个分片表

要将一个未分片表连接到一个已分片的表，必须已在独立的 dbspace 中创建该未分片表，并且必须具有与该分片表相同的表结构。在以下示例中，循环分布方案将表 cur_acct 分片，而且表 old_acct 是驻留在独立 dbspace 中的未分片表。以下示例说明了如何将 old_acct（作为 consumed table ）连接到 cur_acct （作为 surviving table）：

```
ALTER FRAGMENT ON TABLE cur_acct ATTACH old_acct;
```

当您将一个或多个连接到一个分片表时，consumed_table 必须是未分片的。

## 在 ATTACH 操作中使用 ONLINE 关键字

如果没有错误，ONLINE 关键字指示数据库服务器内部提交 ALTER FRAGMENT ATTACH 工作，并在活表上放置意图互斥锁而不是互斥锁。互斥锁只能应用在未分片的死表上。

**ONLINE ATTACH 操作的要求**

只有通过间隔分片结构分片的 ***surviving table*** 才能使用 ALTER FRAGMENT ONLINE ON TABLE 语句 ATTACH 选项。该死表必须是未分片的。

所有在活表上的索引必须具有与表相同的分片结构。（也就是说，任何索引都要连接。）出于这一原因，如果表中有主键约束或其它参考约束，那么，建议您首先为该约束创建连接索引，然后使用 ALTER TABLE 语句添加该约束。（缺省情况下，系统创建的主键约束和其它约束是拆离的。）

对于活表上的每个约束，死表上必须有相同的相符合的索引。死表上匹配的索引在 ATTACH 操作里会作为活表上的再生的索引分片。死表上的其它索引将在 ATTACH 操作中被删除。死表上的每个将会重复利用的索引必须分离于单独的 dbspace 中，并且存储该再生索引的 dbspace 必须是存储该死表的 dbspace 。

如果活表上的索引是唯一的，那么在死表上与其对应的索引也必须唯一。

死表必须具有满足以下条件的检查约束：

- 它必须严格符合要连接的分片的表达式。
- 它只能跨越一个区间。

最后一个要求，死表中的行在活表的区间间隔分片结构中只能跨越单个区间，这对于保护数据移动十分重要。在包含 ONLINE 关键字的 ALTER FRAGMENT ATTACH 操作中不允许数据移动。

ONLINE ATTACH 操作中只能指定一个死表。

所有其它 ATTACH 选项的限制也适用于 ONLINE ATTACH 操作。有关这些限制，请参阅 ATTACH 子句的一般限制和 ATTACH 子句的其他限制。

**ALTER FRAGMENT ONLINE ATTACH 示例**

以下 SQL 语句定义了分片表 **employee** ，它使用区间间隔存储分布方案，在 **emp_id** 列上使用唯一索引 **employee_id_idx**（也是分片密钥）并在 **dept_id** 列上使用另一个索引 **employee_dept_idx**。

```
CREATE TABLE employee
        (emp_id INTEGER, name CHAR(32),
        dept_id CHAR(2), mgr_id INTEGER, ssn CHAR(12))
        FRAGMENT BY RANGE (emp_id)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
        PARTITION p0 VALUES < 200 IN dbs1,
        PARTITION p1 VALUES < 400 IN dbs2;
CREATE UNIQUE INDEX employee_id_idx ON employee(emp_id);
CREATE INDEX employee_dept_idx ON employee(dept_id);
```

最后两条语句使用高于该事务分片的上限的分片键值插入行，这导致数据库服务器产生了两个新的区间分片，由此产生的分片列表包含四个分片：

```
Fragments in surviving table before ALTER FRAGMENT ONLINE:
p0      VALUES < 200            - range fragment
p1      VALUES < 400            - range fragment (transition fragment)
sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

下一条 SQL 语句定义了未分片表 **employee2** ，它与 **employee** 表具有相同的列结构，并在两个与 **employee** 表中索引对应的列（**emp_id** 和 **dept_id**）上放置单独列索引。该语句在 **emp_ssn** 列上定义了唯一索引 **employee2_ssn_idx** 在列 **name** 上定义了 **employee_dept_idx** 索引。这四个索引都存储在 dbspace **dbs4** 中。CREATE TABLE 也语句指定检查约束（`(emp_id >=500 AND emp_id <600)` ），该约束符合要连接死表的分片表达式并跨越了 **employee** 表结构区间间隔分片的单个分区。

```
CREATE TABLE employee2
        (emp_id INTEGER, name CHAR(32),
        dept_id   CHAR(2), mgr_id INTEGER, ssn CHAR(12),
CHECK (emp_id >=500 AND emp_id <600)) in dbs4;
CREATE UNIQUE INDEX employee2_id_idx ON employee2(emp_id) in dbs4;
CREATE INDEX employee2_dept_idx ON employee2(dept_id) in dbs4;
CREATE UNIQUE INDEX employee2_ssn_idx ON employee2(ssn) in dbs4;
CREATE INDEX employee2_name_idx ON employee2(name) in dbs4;
```

以下语句因为要连接的分片是区间分片（存储了分片键值低于 employee 表的事务值 400 的分片）而返回了错误。只有区间分片才能联机连接。

```
ALTER FRAGMENT ONLINE ON TABLE employee
ATTACH employee2 AS PARTITION p3 VALUES < 300;
```

以下语句成功运行并创建了新的间隔分片 **p3** ：

```
ALTER FRAGMENT ONLINE ON TABLE employee
ATTACH employee2 AS PARTITION p3 VALUES < 600;
Fragments in surviving table after ALTER FRAGMENT ONLINE:
 p0       VALUES < 200                        - range fragment
 p1       VALUES < 400                        - range fragment
 sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
 sys_p3 VALUES >= 500 AND VALUES < 600 - interval fragment
 sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

注意到成功的 ALTER FRAGMENT ONLINE . . . ATTACH 操作多个必要规范符合 DDL 语句中的规范，它定义了活表和死表，包括列、索引、索引存储位置和活表的分片策略：

- 死表上的检查约束只能跨越单个区间。活表的区间值为 100 ，检查约束为 >= 500 and < 600 。
- 要连接的条件表达式（< 600）会在内部被转换为符合检查约束的区间分片表达式格式（>= 500 and < 600）。
- 活表上的索引可以连接（也就是说，它们由具有表相同分片结构分片），因为 CREATE INDEX 语句中没有显示地指定分片策略。
- 死表上的索引会被分离到单独的 dbspace（**dbs4**），也是存储该死表的 dbspace 。
- 对于活表上的每个索引，死表上都有与其对应的索引。
- 死表上的与 **employee** 活表不对应的其它索引（**employee2_ssn_idx** 和 **employee2_name_idx**）将在 ONLINE ATTACH 操作中删除。

## ATTACH 子句的影响

在进行 ATTACH 操作之后，所有死表不再存在。对死表的任何 CHECK 约束或 NOT NULL 约束也不再存在。您必须通过活表引用原来死表中的记录。

### *对索引有何影响？*

活表上的拆离索引保留其同一分片存储策略。即，拆离索引不自动调整以适应活表的新分片存储。关于对于索引有何影响的更多信息，请参阅 *GBase 8s 性能指南* 中有关更改表分片的讨论。

在一个日志记录数据库中，ATTACH 操作根据活表的新的分片存储策略，扩展了活表中任何连接的索引。死表中的所有行都服从这些自动调整的索引。关于数据库服务器是完全重建活表的索引还是重新使用原来死表上的索引的信息，请参阅 *GBase 8s 性能指南* 。

在 GBase 8s 的非日志记录数据库中，ATTACH 操作并不会根据活表新的分片存储策略扩展活表的索引。要根据活表的新的分片存储策略扩展所连接的索引的分片存储策略，您必须删除该索引，并在活表上重新创建它。

一些连接分片的 ALTER FRAGMENT ... ATTACH 操作可能导致数据库服务器更新索引的结构。当在这种情况下重建索引时，数据库服务器也将重新计算相关联列的分布方案，并且当其为连接分片的表设置查询计划时这些统计信息可用于查询优化器：

- 对于在 ALTER FRAGMENT ... ATTACH 自动重建 B-tree 索引的索引的列（或列的集合），重新计算的列分布统计信息相当于在 HIGH 模式下 UPDATE STATISTICS 语句创建的分布。
- 如果重建索引不是 B-tree 索引，对应自动重新计算的分布统计信息由 UPDATE STATISTIC 语句在 LOW 模式下创建。

有关在现有表上创建索引或约束时自动产生统计分布的其他信息，请参阅 CREATE INDEX 语句中自动计算分布统计信息一节的描述。

### *对 BYTE 和 TEXT 列有何影响？*

当 ATTACH 发生时，死表的 BYTE 和 TEXT 分片会成为活表的一部分，并继续与 ATTACH 操作之前所关联的相同行和数据分片相关联。

ATTACH 子句中指定的每个表中的每个 BYTE 和 TEXT 列必须具有相同的存储类型：blobspace 或 tblspace 。如果 BYTE 或 TEXT 列存储在 blobspace 中，则所有表中的同一列必须在同一 blobspace 中。如果 BYTE 或 TEXT 列存储在 tblspace 中，则所有的表中的同一列必须存储在一个 tblspace 中。

### *对触发器和视图有何影响？*

当您连接表时，活表中的触发器将在 ATTACH 后保留下来，但死表中的触发器会被自动删除。ATTACH 子句不激活任何触发器，但随后对新行的数据处理操作会激活触发器。

活表中的视图在 ATTACH 操作后保留下来，但死表中的视图会被自动删除。

### *对分布方案有何影响？*

您可以将未分片表连接到一个具有任何类型的受支持分布方案的表。通常，生成的表具有同 surviving table 先前的分片存储策略相同的分片存储策略。

但是，当您连接两个或两个以上的未分片表时，分布方案可以基于表达式也可以基于循环。

通过在 ATTACH 子句中结合表的分布方案，只能生成以下分布方案：

| 活表先前的分布方案 | 死表先前的分布方案 | 生成的分布方案 |
|---|---|---|
| 无 | 无 | 循环或表达式 |
| 循环 | 无 | 循环 |
| 表达式 | 无 | 表达式 |

### *循环分布方案*

以下示例将未分片表 pen_types 和 pen_makers 合并为单独一个分片表 pen_types。表 pen_types 驻留在 dbspace dbsp1 中，表 pen_makers 驻留在 dbspace dbsp2 中。每个表中的表结构是相同的。

ALTER FRAGMENT ON TABLE pen_types ATTACH pen_types, pen_makers;

执行 ATTACH 子句之后，数据库服务器使用循环分布方案将表 **pen_types** 分片为两个 dbspaces：包含 **pen_types** 的 dbspace 和包含 **pen_makers** 的 dbspace 。表 **pen_makers** 已死，并且不再存在；原来在表 **pen_makers** 中的所有行现在都在表 **pen_types** 中。

### *表达式分布方案*

考虑以下将表 cur_acct 和 new_acct 合并以及使用基于表达式的分布方案的示例。表 cur_acct 最初创建为一个分片表，并且在 dbspace dbsp1 和 dbsp2 中有分片。该示例的第一个语句显示表 cur_acct 是以基于表达式的分布方案创建的。该示例的第二个语句在 dbsp3 中创建表 new_acct ，而没有分片存储策略。第三个语句合并了表 cur_acct 和 new_acct 。每个表中的表结构（列）都是相同的。

```
CREATE TABLE cur_acct (a int) FRAGMENT BY EXPRESSION
        a < 5 in dbsp1, a >= 5 and a < 10 in dbsp2;
CREATE TABLE new_acct (a int) IN dbsp3;
ALTER FRAGMENT ON TABLE cur_acct ATTACH new_acct AS a>=10;
```

当您更改分片后检查 **sysfragments** 系统目录表时，您可以看到表 **cur_acct** 按表达式分片为三个 dbspace。关于 **sysfragments** 系统目录表的其它信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

除了简单的范围规则，您也可以使用 ATTACH 子句，通过哈希或仲裁规则按表达式分片。关于您可以在基于表达式的分布方案中用到的所有表达式类型的讨论，请参阅 通过 EXPRESSION 分片。

**警告：** 当您指定一个日期值作为参数的缺省值时，请确保对年份指定 4 位数字，而非 2 位数字。当指定 2 位数字的年份时，环境变量 DBCENTURY 可能不使用希望的缺省值。有关更多信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## DETACH 子句

使用 ALTER FRAGMENT ON TABLE 语句的 DETACH 子句以将表分片从分布方案拆离，并将这些内容放入新的未分片表中。

此子句在 ALTER FRAGMENT ON INDEX 语句中无效。

有关分布方案的说明，请参阅 FRAGMENT BY 子句 。

DETACH 子句


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *fragment* | 包含要拆离的表分片的分片或 | 执行时必须存在。 对于 | 标识 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | dbspace 的名称。 | 列表或范围区间分片，PARTITION 关键字必须产生 *fragment*。 | 符 |
| *new_table* | 执行 ALTER FRAGMENT 语句后产生的未分片表的名称。 | 执行前必须存在 | 标识符 |

**用法**

执行 DETACH 子句而生成的新表不会从最初的表继承任何索引或约束。只保留数据值。

类似地，新表不从最初的表继承任何特权。而是具有任何新表都有的缺省特权。关于缺省表级别特权的进一步信息，请参阅 表级权限 中的 GRANT 语句。

DETACH 子句无法应用到以下具有任一属性的表中：

- 定义了 ROWID 列
- 定义了一列或队列为参考约束的主键
- 表中定义了 Enterprise Replication 复制
- 具有拆离索引（即，存储分布方案的索引与表的分片策略不同）

如果省略 PARTITION 关键字，那么分片的名称就是存储分片的 dbspace 的名称。

在以下示例中，系统生成的范围区间分片 sys_pt1 从表 T1 拆离并放置到新的未分片表 detacht1 中：

```
ALTER FRAGMENT ON TABLE T1 DETACH PARTITION sys_pt1 detacht1;
```

下一示例为从表 T2 拆离了列表分片 part2 并将其数据放置到新的未分片表 detacht2 中：

```
ALTER FRAGMENT ON TABLE T2 DETACH PARTITION part2 detacht2;
```

**DETACH 操作后的分布统计信息**

某些 ALTER FRAGMENT . . . DETACH 操作可能导致数据库服务器更新初始表的索引结构。当在这种情况下重建索引时，数据库服务器也将重新计算相关联列的分布方案，并且当其为拆离分片的表设置查询计划时这些统计信息可用于查询优化器：

- 对于在 ALTER FRAGMENT ... DETACH 自动重建 B-tree 索引的索引的列（或列的集合），重新计算的列分布统计信息相当于在 HIGH 模式下 UPDATE STATISTICS 语句创建的分布。
- 如果重建索引不是 B-tree 索引，对应自动重新计算的分布统计信息由 UPDATE STATISTIC 语句在 LOW 模式下创建。

如果启用更新列分布统计信息的自动模式，而且来自正在拆离分片的表具有分片级别分布统计信息，那么数据库服务器使用拆离的分片的统计信息作为新表的分布统计信息。数据库服务器也合并驻留分片数据分布统计信息以计算初始表的新表分布统计信息，并会将结果存储在 **sysdistrib** 系统目录表中。新表的分布统计信息的注册和旧表的表分布统计的重新计算都在后台运行。

有关在现有表中创建索引或约束时自动生成统计分布的语句的其他信息，请参阅自动计算分布统计信息中的 CREATE INDEX 语句的描述。

## 在 DETACH 操作中使用 ONLINE 关键字

ONLINE 关键字指示数据库服务器内部提交 ALTER FRAGMENT ... DETACH 工作，如果没有错误那么在要拆离分片的表上放置意向互斥锁而不是互斥锁。互斥锁可应用在从拆离分片创建的表上。

您只能对使用范围区间分片结构的表使用 ALTER FRAGMENT ONLINE ON TABLE 语句的 DETACH 选项。

使用范围区间存储分布方案的表具有两种类型的分片：

- ***range*** 分片，由 CREATE TABLE 或 ALTER TABLE 语句的 FRAGMENT BY 或 PARTITION BY 子句的用户定义
- ***interval*** 分片，如果具有超出过渡分片值上限（最后一个范围分片）的分片键值的行，那么数据库服务器在 INSERT 和 UPDATE 操作中自动生成该分片。

ONLINE DETACH 操作中只能拆离一个区间分片。

如果已拆离的区间分片不是最后一个分片，那么数据库服务器修改系统生成的遵循分片列表以匹配活表中它们的新的 **sysfragments.evalpos** 值的拆离分片的名称。在重命名此分片的操作中，当 **sysfragments** 系统目录正在用新的 **partition** 名称更新时，在分片上放置互斥锁（并且在 ALTER FRAGMENT DETACH 操作过程中，为在分片列表中变更初始位置的任何分片使用新 **evalpos** 值）。

活表上所有的索引必须与该表具有相同的分片结构。（即，任何索引必须可连接）由于这个原因，如果此表有主键约束或其它参考约束，那么建议您首先创建为该约束连接索引，然后使用 ALTER TABLE 语句添加该约束。（缺省情况下，主键约束的系统创建索引和其它已拆离的参考约束。）

如果有会话正在访问要拆离的同一分区，建议您声明 SET LOCK MODE TO WAIT 语句来获得保护非互斥存取错误足够的时间。

其它应用于 DETACH 选项的限制同样适用于 ONLINE DETACH 操作。有关这些限制，请参阅 对 ALTER FRAGMENT 语句的限制 和 DROP 子句。

**ALTER FRAGMENT ONLINE ... DETACH 的示例**

以下 SQL 语句定义了一个分片表 **employee** ，它使用范围区间存储分布方案，在列 **emp_id**（也是分片键）上有一个唯一索引 **employee_id_idx** 在列 **dept_id** 上有另一个索引。

```
CREATE TABLE employee (emp_id INTEGER, name CHAR(32), dept_id CHAR(2),
        mgr_id INTEGER, ssn CHAR(12))
        FRAGMENT BY RANGE (emp_id)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
        PARTITION p0 VALUES < 200 IN dbs1,
        PARTITION p1 VALUES < 400 IN dbs2;
CREATE UNIQUE INDEX employee_id_idx ON employee(emp_id);
CREATE INDEX employee_dept_idx ON employee(dept_id);

INSERT INTO employee VALUES (401, "Susan", "DV", 101, "123-45-6789");
```

```
INSERT INTO employee VALUES (601, "David", "QA", 104, "987-65-4321");
```

最后两条语句使用超出过渡分片上限的分片键值插入了行，这导致数据库服务器生成了两个新区间分片，以致于生成包含四个分片的分片列表：

```
Fragments in surviving table before ALTER FRAGMENT ONLINE:
p0       VALUES < 200            - range fragment
p1       VALUES < 400            - range fragment (transition fragment)
sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

以下语句返回了错误，因为指定的要拆离的分片是范围分片（分片存储的行的分片键值低于过渡值 400）。只有区间分片才能联机拆离。

```
ALTER FRAGMENT ONLINE ON TABLE employee
DETACH PARTITION p0 employee3;
```

以下语句成功运行，并创建了新表 **employee3** 以存储已拆离的分片中的数据。

```
ALTER FRAGMENT ONLINE ON TABLE employee
DETACH PARTITION sys_p2 employee3;
```

如果有并行的会话访问 **sys_p2** ，请将锁定模式设置为 WAIT （提交的 ONLINE DETACH 操作要满足的秒数）以保护非互斥访问错误：

```
SET LOCK MODE TO WAIT 300;
ALTER FRAGMENT ONLINE ON TABLE employee DETACH PARTITION sys_p2 employee3;

Fragments in surviving table after ALTER FRAGMENT ONLINE:
p0       VALUES < 200                    - range fragment
p1       VALUES < 400                    - range fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment.
```

## 使用 BYTE 和 TEXT 列拆离

如果 DETACH 子句指定包含 BYTE 或 TEXT 数据类型的简单大对象的表的第一个分片，那么数据库服务器会锁定该表中每个分片的 blobspace 。要拆离该表的其它分片，那么请只锁定指定分片的 blobspaces ，而不是所有分片的 blobspaces ，如果此分片不是第一个那么需要较少的锁。

## 从受保护的表拆离

如果 DETACH 子句指定安全策略保护的表执行成功的话，数据库服务器会创建受相同安全策略保护的表，并具有相同行安全标签的 IDSSECURITYLABEL 列，和相同受保护的列集合作为初始表。IDSSECURITYLABEL 列有 NOT NULL 约束。只有持有 DBSECADM 角色的用户可以引用 ALTER FRAGMENT 语句中受保护的表。

## 生成未分片表的拆离

以下示例使用了已分片为两个 dbspace dbsp1 和 dbsp2 的表 cur_acct：

ALTER FRAGMENT ON TABLE cur_acct DETACH dbsp2 accounts;

此示例将 dbsp2 从 cur_acct 的分布方案拆离，并将这些行放入一个新表 accounts 中。表 accounts 现在具有与 cur_acct 相同的结构（列名、列数、数据类型等），但表 accounts 不包含表 cur_acct 中的任何索引和约束。这两个表现在都未分片的。以下示例显示了一个包含三个分片的表：

ALTER FRAGMENT ON TABLE bus_acct DETACH dbsp3 cli_acct;

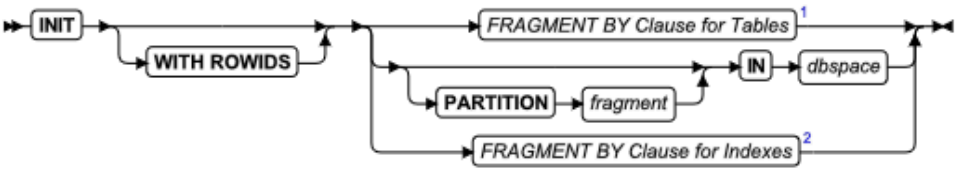
此语句将 **dbsp3** 从 **bus_acct** 的分布方案拆离，并将这些行放入一个新表 **cli_acct** 中。表 **cli_acct** 现在具有与 **bus_acct** 相同的结构（列名、列数、数据类型等），但表 **cli_acct** 不包含表 **bus_acct** 的任何索引和约束。表 **cli_acct** 是一个未分片表，但表 **bus_acct** 仍是一个分片表。

## INIT 子句

ALTER FRAGMENT 语句的 INIT 子句可以定义或修改现有表或现有索引的分片策略或存储位置。

### 语法

INIT 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbspace* | 存储已分片数据的 Dbspace | 在执行时必须存在 | 标识符 |
| *fragment* | 分片的名称 | 对同一个表不超过 2048 个 | 标识符 |

INIT 子句可以完成的任务包括：

- 将未分片表从一个 dbspace 移动到一个命名的分片或另一个 dbspace 。
- 将分片表转换为未分片表。
- 将现有的未分片表分片，而不用重新定义它。
- 将一个分片粗出策略转换为另一个分片存储策略。
- 将未分片的现有索引分片，而不重新定义该索引。
- 将分片索引转换为未分片所以。
- 向表定义添加一个新的 **rowid** 列。

当您使用 INIT 子句修改表时，系统目录表中的 **tabid** 值会为受影响的表而更改。该表所有唯一和引用约束的 **constrid** 值也会更改。

有关您可以存储表的存储空间的更多信息，请参阅使用 IN 子句。

**注意：** 当您带此子句执行 ALTER FRAGMENT 语句时，如果该表包含任何数据，则会产生数据移动。如果数据值移动，则可能存在：大量日志记录、正在作为长事务放弃的事务、正在受影响的表上存储的比较长的互斥锁。请在此语句不会妨碍日常操作时使用它。

## WITH ROWIDS 选项

为分片表包含一个称为 **rowid** 的隐藏列。缺省情况下，分片表不包含此列。它的整型值定义了此行的物理位置。

要在分片表中包含 **rowid** 列，必须在 CREATE TABLE 中使用 WITH ROWIDS（或在 ALTER TABLE 中使用 ADD ROWIDS 或在 ALTER FRAGMENT INIT 中使用 WITH ROWIDS）显式地创建。分片表的行中的 **rowid** 无法使用未分片表中 **rowid** 的方法辨别该行的物理位置。

当您使用 WITH ROWIDS 选项为一个分片表添加新的 **rowid** 列时，数据库服务器为每一列分片唯一的 **rowid** 编号并创建索引以查找该行的物理位置。使用此存取方法的性能媲美使用 SERIAL 、BIGSERIAL 或 SERIAL 列。一个行的 **rowid** 值无法更新，但在该行存在期间保持稳定。您指定 WITH ROWIDS 选项之后，每行都要求额外四个字节存储 **rowid** 列。

建议： 当创建新应用时，使用主键而不是 **rowid** 值作为存取方法。

## 将分片表转换为未分片表

您可能决定不再使一个表分片。您可以使用 INIT 子句将一个分片表转换为一个未分片表。以下示例显示了最初的分片存储定义，以及如何使用 ALTER FRAGMENT 语句的 INIT 子句转换该表：

```
CREATE TABLE checks (col1 INT, col2 INT)
        FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2, dbsp3;
ALTER FRAGMENT ON TABLE checks INIT IN dbsp1;
```

您必须使用 IN ***dbspace*** 子句将该表显式地放入一个 dbspace 。

当您使用 INIT 子句将分片表更改为未分片表时，所有连接的索引都成为未分片索引。此外，不使用现有用户定义的索引（拆离索引）的约束都成为未分片索引。所有最新未分片的索引存在于与新未分片表相同的 dbspace 中。

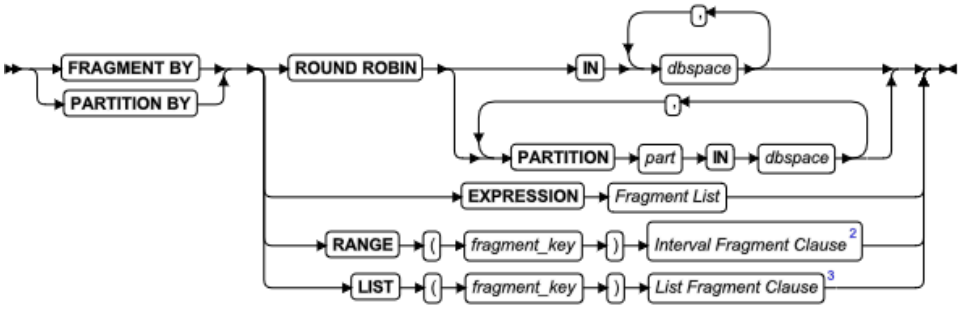
使用 INIT 子句将分片表更改为未分片表既不会对拆离索引的分片存储策略产生影响，也不会对使用拆离索引的约束产生影响。

## 表的 FRAGMENT BY 子句

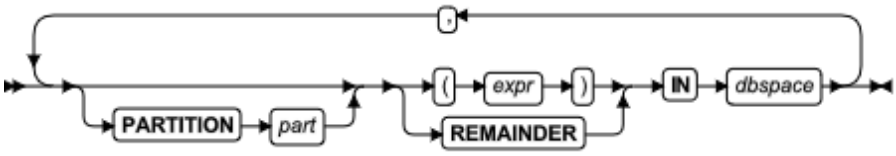
使用 ALTER FRAGMENT 语句的 INIT 子句的 FRAGMENT BY 选项来分片现有未分片表，或将一个表分片策略转换为另一个。

本文中 PARTITION BY 关键字类似于 FRAGMENT BY 关键字。

表的 FRAGMENT BY 子句




分片列表




| 约束 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 该策略适用的列 | 必须存在于表中 | 标识符 |
| *dbspace* | 包含表分片的 Dbspace | 必须指定至少 2 个但不超过 2,048 个 dbspaces | 标识符 |
| *expr* | 定义表分片的表达式 | 必须求出一个 Boolean 值（t 或 f） | 表达式 |
| *part* | 分片的名称 | 同一 dbspace 中分片的名称必须作为同一表中另一分片的名称。同一表中的分片的名称必须唯一。 | 标识符 |

它可作为 CREATE TABLE 子句 FRAGMENT BY （或 PARTITION BY ）子句的语法。有关表可用的分片策略的信息，请参阅 CREATE TABLE 中的 FRAGMENT BY 子句 。

**范围区间分片的示例**

这些示例的定义语句定义了一个现有表的范围区间分片策略。随后的 ALTER FRAGMENT 语句定义了范围区间策略的三个分片，包括非 NULL 分片，数值列 **c1** 是分片键：

```
ALTER FRAGMENT ON TABLE T1 INIT
        FRAGMENT BY RANGE(c1)
        INTERVAL (100+100) STORE IN (dbs3, dbs4, dbs5, dbs6, dbs7, dbs8)
        PARTITION part0 VALUES < 0       IN dbs0,
        PARTITION part1 VALUES < 1000 IN dbs1,
```

```
        PARTITION part2 VALUES < 2000 IN dbs2;
```

(100+100）的区间值表达式定义了列 **c1** 范围内区间分片的大小为 200 。如果当插入一个 **c1** 等于或大于 2000 行时，这仍是其存储分布，那么数据库服务器会自动创建新的分片以存储行，超出现有分片范围。区间分区以轮循机制的方式存储在 dbs2 、dbs3 、dbs4 、 dbs5 、dbs6 、dbs7 和 dbs8 dbspace 中。

以下语句类似地定义了一个范围区间分片策略（包括非 NULL 分片和 DATE 或 DATETIME 列 **c2** 是分片键 i）的三个分片：

```
ALTER FRAGMENT ON TABLE T1 INIT
        FRAGMENT BY RANGE(c2)
        INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
        PARTITION part0 VALUES <   DATE('01/01/2009') IN dbs0,
        PARTITION part1 VALUES <   DATE('07/01/2009') IN dbs1,
        PARTITION part2 VALUES <   DATE('01/01/2010') IN dbs2;
```

此处 NUMTOYMINTERVAL(1,'MONTH') 区间值表达式定义了在 **c2** 列的范围内的单个月作为区间分片大小。PARTITION 列表定义了三个分片：2008 年 12 月的 part0 、2008 年七月的 part1 和 2009 年 12 月的 part2 f。如果要插入行的 **c2** 值不是这三个其中的月，数据库服务器会为这些行创建新的分片。因为 STORE IN 子句没有指定，数据库服务器将在 dbs0 、dbs1 和 dbs2 dbspace 中以轮循机制的方式存储这些范围区间分片，在三个 PARTITION 指定的 IN 关键字之后。

## *更改一个表上的现有分片存储策略*

如果您确定一个表上的初始策略不能满足您的需求，则您可以重新定义该表的分片存储策略。当您更改分片存储策略时，数据库服务器会废弃现有的分片存储策略，并按新的分片存储策略中的定义将记录移到分片中。

以下示例显示了在 account 表上最初定义的分片存储策略，然后显示了重新定义分片存储策略的 ALTER FRAGMENT 语句：

```
CREATE TABLE account (col1 INT, col2 INT)
        FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2;
        ALTER FRAGMENT ON TABLE account
        INIT FRAGMENT BY EXPRESSION
        col1 < 0 IN dbsp1,
        col2 >= 0 IN dbsp2;
```

当您重新定义一个分片存储策略时，一个现有 dbspace 已满，则您不得在新的分片存储策略中使用它。

## *在未分片表上定义分片存储策略*

INIT 子句可以在未分片表上定义分片存储策略，无论该表是否是使用存储选项创建的。

```
CREATE TABLE balances (col1 INT, col2 INT) IN dbsp1;
ALTER FRAGMENT ON TABLE balances INIT
        FRAGMENT BY EXPRESSION col1 <= 500 IN dbsp1,
```

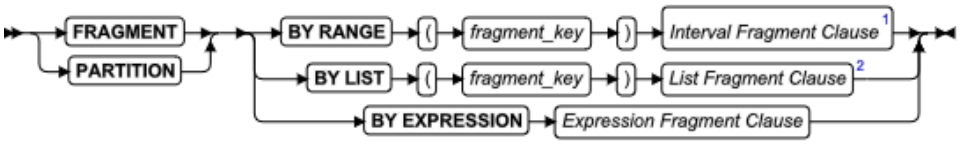
```
col1 > 500 AND col1 <=1000 IN dbsp2, REMAINDER IN dbsp3;
```

当使用 INIT 子句分片现有未分片表时，该表上的所有索引将以与表相同的方式分片。

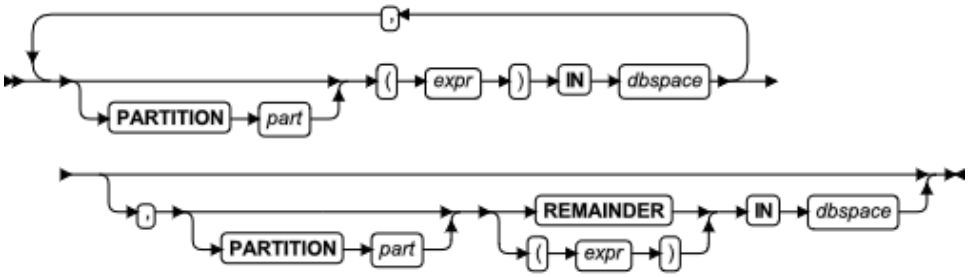
## 索引的 FRAGMENT BY 子句

可以使用 FRAGMENT BY 子句重新定义索引的存储分布策略，而不重新定义该索引。在本文中 FRAGMENT BY 和 PARTITION BY 的关键字类似。

索引的 FRAGMENT BY 子句




表达式分片子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbspace* | 包含分片信息的 Dbspace | 必须指定至少两个但不超过 2,048 个同一页大小的 dbspaces | 标识符 |
| *expr* | 定义索引分片的表达式 | 对于同一索引其分片表达式必须唯一，必须返回 Boolean 值 | 条件；表达式 |
| *fragment_key* | 基于列值的常量表达式。该索引根据此表达式分片。 | 任何列必须在此当前表中 | 表达式 |
| *part* | 您在此处为一个指定的分片的名称。缺省值为 *dbspace* 的名称。 | 对于与同一索引的另一个分片相同的 dbspace 中的任何分片都是必须的。在同一索引的分片中必须是唯一的。 | 标识符 |

ALTER FRAGMENT 语句的中索引的 INIT FRAGMENT BY 子句可以在现有索引的存储分布方案上完成以下任一操作，而不需重新定义索引：

- 将现有的已分片的索引更改为未分片的索引。
- 将现有分片索引的分布方案更改为另一种类别的分布方案，或另一个具有相同表达式、表或范围区间类型的分布方案。
- 更改现有索引的范围区间分布方案的区间值或区间分片键（或两者都更改。）

要更改现有的被范围区间策略分片的索引区间值表达式或分片键表达式，您必须使用 ALTER FRAGMENT 语句的 INIT FRAGMENT BY RANGE 选项（而不是 MODIFY 子句）。当您更改其中之一或所有的表达式时，ALTER FRAGMENT ON INDEX 语句中的 Interval Fragment 子句必须定义至少一个范围分片。

当您使用 FRAGMENT BY 或 PARTITION BY 子句将现有存储分片策略转换为另一个分配策略时，GBase 8s 会废弃现有的分片策略并将数据记录移动到新分片策略中您定义的分片中去。当您将一个未分片索引转换为分片索引和将分片索引转换为未分片索引时，数据移动同样发生。

将一个现有的已分片的索引转换为未分片的索引时，您可以使用 INIT 子句指定 IN ***dbspace***（或 PARTITION ***partition*** IN ***dbspace***）作为前一个分片索引的唯一存储规范。

正如 CREATE INDEX 语句定义的基于表达式索引分片方案，您在 ALTER FRAGMENT ON INDEX . . . INIT FRAGMENT BY EXPRESSION 语句中指定的每一个表达式都要应用以下限制：

- 任一表达式所引用的列必须来自当前表。
- 这些列必须是被索引的列或此被索引的列子集。
- 表达式无法引用 ROW 类型列的字段。
- 该表达式中的数据值必须来单个行。
- 不允许任何子查询、聚合和 **CURRVAL** 或 **NEXTVAL** 顺序对象表达式 。
- 内置的 **CURRENT** 、**DATE** 、**DBINFO** 、**DBSERVERNAME** 、**ROWID** 、**SITENAME** 、**SYSDATE** 、**TODAY** 、 **CURRENT_USER** 和 **USER** 表达式在此表达式中不可用。

以上限制同样适用于列表和范围区间索引分片结构的分片键表达式，包括 CREATE INDEX 语句的 FRAGMENT BY 子句定义的分片策略。

## *将索引从表分片存储策略中拆离*

您可以使用 ALTER FRAGMENT ON INDEXY 语句的 INIT 子句将索引从表分片存储策略中拆离，这将导致连接的索引成为拆离的索引。这打破了该索引与表分片存储策略的任何相关性。如果 INIT 子句对先前的索引仅指定 IN ***dbspace*** 或 PARTITION ***fragment*** IN ***dbspace*** ，或指定一个和该表的存储选项不同的索引分片存储策略没那么该索引将成为已拆离的索引。

## *分片唯一索引和系统索引*

您可以是使用或基于表达式或轮循分布方案分片表中的唯一索引，但是分片表达式中引用的所有列必须是被索引列。如果您的索引分片策略未能符合这些限制，那么 ALTER FRAGMENT INIT 语句会失败，并且工作会回滚。

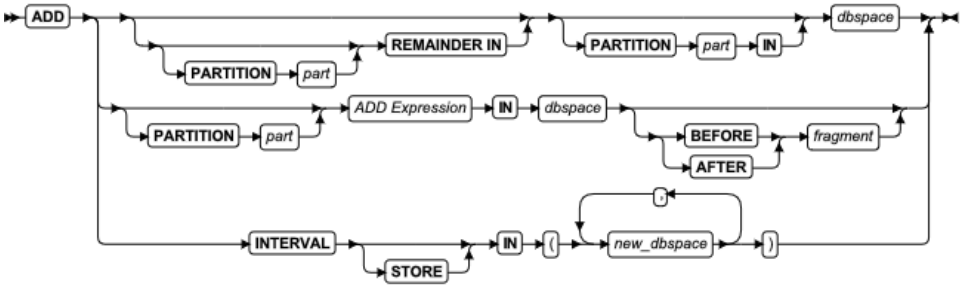
您可能在以 Column A 分片的表上具有一个连接的唯一索引。如果您使用 ALTER FRAGMENT INIT 将该表的分片存储更改为 Column B ，则该语句失败。因为唯一索引定义在 Column A 上，要解决此问题，请对该索引使用 INIT 子句以将其从表分片存储策略拆离并将其单独分片。

系统索引（例如引用约束和唯一约束中使用的那些索引）使用用户索引（如果这些索引存在）。如果没有用户索引可以使用，系统索引保留未分片状态，并移到创建该数据库服务器的 dbspace 中。要分片系统索引，请在约束列宏创建一个分片索引，然后使用 ALTER TABLE 语句添加该约束。

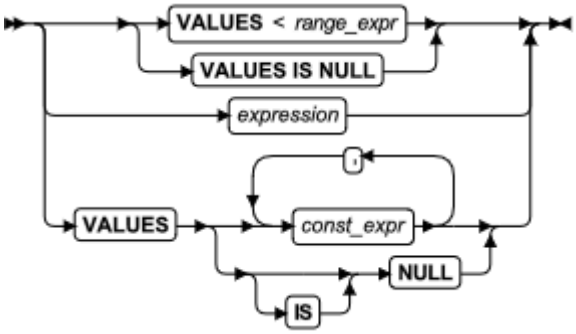
## ADD 子句

使用 ADD 子句将另一个分片添加到表或索引的现有分片列表中。

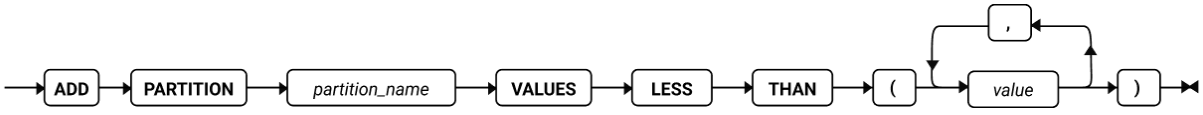
ADD 子句




ADD 表达式




oracle 模式范围分区添加分区子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *dbspace* | 存储新的分片的 dbspace 的名称 | 必须存在 | 标识符 |
| *expression* | 定义要添加的新分片的表达式 | 必须返回 Boolean 值（t 或 f） | 条件；表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *fragment* | 现有分片的名称 | 必须存在 | 标识符 |
| *new_dbspace* | 要添加到分片存储方案的 dbspace 的名称 | 必须存在 | 标识符 |
| *part* | 您在此处为分片声明的名称。缺省名称是 dbspace 的名称 | 对于与同一索引的另一个分区相同的 dbspace 中的任何分区都是必需的。 | 标识符 |
| *partition_name* | 分区名称 | 在表中分区名称唯一 | 标识符 |
| *value* | 分区值 | 数值或者日期时间 | 标识符 |

***expression*** 仅包含当前表中的 ***column*** 名称，以及单独一行中的数据值。不允许任何子查询或聚集。此外，在此处，内置的 **CURRENT** 、**DATE** 、**DBINFO** 、**SYSDATE** 和 **TODAY** 表达式无效。

**向循环分布方案添加新的 Dbspace**

您可以向循环分布方案添加更多的 dbspace 。以下示例显示了最初的循环定义：

```
CREATE TABLE book (col1 INT, col2 INT)
    FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp4;
```

要添加另一个 dbspace ，请如下例使用 ADD 子句：

```
ALTER FRAGMENT ON TABLE book ADD dbsp3;
```

## oracle 模式范围分区添加新分区

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

使用如下语法对范围分区添加新的分区

```
alter fragment on table 表名 add partition 分区名 values less than (值);
```

**用法**

- 新添加的分区不能小于原分区表分区最大值。
- 新增的分区名不能使用表中已经存在的分区名。

示例如下：

先创建范围分区表

```
create table tab11
 (cust_id integer,name char(128))
  partition by range(cust_id)
```

```
(
  partition p0 values less than (100),
  partition p1 values less than (200)
);
```

根据示例语法添加分区

```
alter fragment on table tab11 add partition p2 values less than (300);
```

## 向循环分布方案添加新的 Dbspace

您可以向循环分布方案添加更多的 dbspace 。以下示例显示了最初的循环定义：

```
CREATE TABLE book (col1 INT, col2 INT)
    FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp4;
```

要添加另一个 dbspace ，请如下例使用 ADD 子句：

```
ALTER FRAGMENT ON TABLE book ADD dbsp3;
```

## 向循环分布方案添加新的分片

在 GBase 8s 中，您可以向现有的循环分布方案添加的分片。它的名称必须在同一 dbspace 多个分片的分布中必须是唯一的。以下示例如在前一节中那样指定相同的最初循环分片存储定义：

```
CREATE TABLE book (col1 INT, col2 INT)
            FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp4;
```

要添加新的已命名的分片，请按以下示例使用 ADD 子句：

```
ALTER FRAGMENT ON TABLE book
            ADD PARTITION chapter3 IN dbsp1;
```

新的分布使用 **dbsp1** 、**dbsp4** 和 **chapter3** 作为 3 部分循环分片存储方案的存储位置。分片 **chapter3** 中的记录存储在与第一个分片中的记录相互独立的 **dbsp1** dbspace 中。

## 添加分片表达式

向一个基于表达式的分布方案的分片列表中添加分表达式会将现有分片中的记录重新分配到新的分片中。当您将新的分片添加到分片列表中时，数据库服务器将对位于新分片之后的分片中的所有数值重新求值。（**sysfragments** 系统目录表中的任一 **evalpos** 列值标识了此分片在分片列表中的初始位置。）

下一示例显示最初的表达式定义：

```
FRAGMENT BY EXPRESSION
            c1 < 100 IN dbsp1, c1 >= 100 AND c1 < 200 IN dbsp2,
            REMAINDER IN dbsp3
```

要在 dbspace dbsp2 的一个新分片中添加另一个分片以存储 200 到 299 之间的 c1 值的列，请使用以下 ALTER FRAGMENT 语句：

```
ALTER FRAGMENT ON TABLE news
```

```
ADD PARTITION century3 (c1 >= 200 AND c1 < 300) IN dbsp2;
```

除了满足标准（c1 >= 200 AND c1 < 300）的行之外，所有原来在余项分片中的行都移动到 dbspace **dbsp2** 中新的 **century3** 分区中。

如果当启用自动更新分布统计信息方式，ALTER FRAGMENT ADD 操作会重新分布数据行，数据库服务器会删除已受影响分片的分布统计信息，但是不删除该表的统计信息。该表的下一个查询将会导致数据服务器为重新计算同一分片的统计信息。

## 使用 BEFORE 和 AFTER 选项

BEFORE 和 AFTER 选项可以将新的分片放置于分片列表中现有分片之前或之后。分片的名称是 dbspace 的名称或 PARTITION 子句中声明的名称。如果分布方案是循环或范围区间的，您就不能使用 BEFORE 和 AFTER 选项。

当您连接新的分片而未使用 BEFORE 或 AFTER 选项时，数据库服务器会将所添加的分片置于分片列表的末尾，除非存在一个余项分片。如果存在一个余项分片，则新的分片会刚好置于该余项分片前。您不能在余项分片之后连接一个新分片。

## 使用 REMAINDER 选项

如果一个余项分片已存在，您就不能添加它。如果您在一个余项分片存在时添加一个新的分片，数据库服务器将检索并重新计算余项分片中的所有记录；一些记录将可能移动到新的分片中。余项分片总是分片列表中的最后一项。

您不能向范围区间分片结构的分片列表中添加一个余项分片。

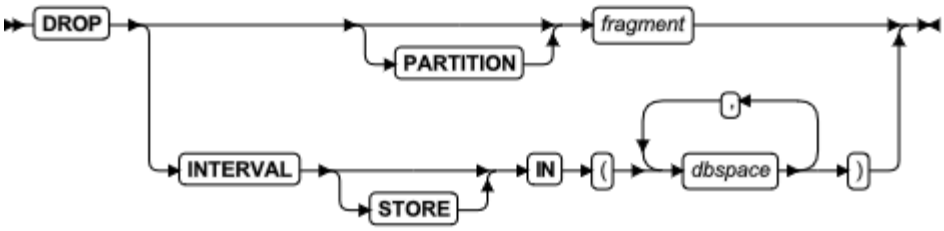
## DROP 子句

使用 DROP 子句从按循环分片的分片表或索引的分片列表中除去现有分片。在 gbase 模式下对于按范围区间分片的表或索引，您可以使用此子句删除一个或多个存储系统生成的区间分片的 dbspace 列表中的 dbspace 。

Oracle 模式下的范围分区 drop 子句只支持 DROP PARTITION *partition_name* 语法

使用 DETACH 子句而不是 DROP 子句移除使用范围区间分片的表中的现有分片，例如滚动窗口表。

DROP 子句




对于 oracle 模式下范围分区 drop 子句语法图


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***dbspace*** | 存储系统生成的分片的 Dbspace | 当执行该语句时必须存在。 | 标识符 |
| ***fragment*** | 分片的名称 | 当执行该语句时必须存在。 对于列表或范围区间分片， PARTITION 关键字必须在此名称之前。 | 标识符 |
| ***partition_name*** | 分区的名称 | 需要删除的分区必须存在，PARTITION 关键字必须在此名称之前。 | 标识符 |

如果表是按表达式分片的，则不可以删除包含无法移动到其它分片的数据的分片。如果分布方案有 REMAINDER 选项或者如果表达式重叠，则可以删除包含数据的分片。如果表只含两个分片，则不能删除其中一个分片。

当您想要取消一个分片表的分片时，请使用 ALTER FRAGMENT 语句的 INIT 子句或 DETACH 子句。而不是 DROP 子句。

如果 ***fragment*** 在被创建或添加时未命名，那么 sbspace 的名称同时也是该分片的名称。如果该分片是系统生成的表或索引的范围区间分片，那么它的名称是 **sys_p*evalpos***，***evalpos*** 是系统目录中分片的 **sysfragments.evalpos** 项。如果一个表或索引使用相同的范围区间分片策略，那么每个系统生成的索引分片具有和该表系统生成分片的一样的标识。

当您删除一个分片时，数据库服务器尝试将被删除分片中的所有记录移到到另一个分片中。在此情况下，目标分片可能没有足够的空间容纳这些添加的记录。如果发生这种情况，请遵循 ALTER FRAGMENT 和事务日志记录 中描述的过程，增加您的可用空间，并重试 ALTER FRAGMENT 操作。

当 DROP 子句指定一个或多个 dbspaces 从范围区间分片策略中移除时，这些 dbspace 不受影响，但数据库服务器会将存储在该 dbspace 中表或索引的分片中的数据移动到其它可用 dbspace 中。（范围区间策略也将会受影响，因为它不再包含在新的系统生成分片的存储位置中的指定的 dbspace 。）

您无法使用 DROP 子句删除一个包含数据的范围区间分片。

您可以使用此子句删除包含数据的分片列表，只要余项分片接收这些数据。

如果此分片表由分片级别统计信息，那么 ALTER FRAGMENT DROP 操作也删除要被移除分片的分片级别统计分布。然而，不会重新计算表级别的统计信息。表的下一个显式或自动的 UPDATE STATISTICS 操作将会重建旧的分片级分布并从表级别分布合并，再将结果存储在系统目录中。

**ALTER FRAGMENT 语句的 DROP 子句示例**

以下示例显示了如何从循环分片列表删除一个分片。第一行显示如何删除一个索引分片，第二行显示如何删除一个表分片。

```
ALTER FRAGMENT ON INDEX cust_indx DROP dbsp2;
ALTER FRAGMENT ON TABLE customer DROP dbsp1;
```

以下每个示例删除除了有列表分片策略定义的分片。第一行显示如果删除表分片，第二行显示如何删除索引分片。

```
ALTER FRAGMENT ON TABLE T2 DROP PARTITION part4;
ALTER FRAGMENT ON INDEX idx2 DROP PARTITION part4;
```

在以上所有的示例中，PARTITION 关键字是必需的，已删除的分片的名称为 part4 。如果索引 idx2 在表 T2 中定义，且要其具有与表 T2 相同的存储分布策略，那么第二条语句不是必需的。因为当表分片列表被修改时，数据库服务器会自动修改要连接的索引的分片策略。如果这些分片不为空，那么数据库服务器将把它们的数据移动到余项分片中（如果没有余项分片存在，则会返回错误）。

以下每个示例都删除了存储系统定义区间范围分片的 dbspace （由范围区间分片策略定义）。第一条语句删除来自表分片的存储空间的 dbspaces dbs7 和 dbs8 ，第二条语句删除了来自索引分片的相同存储空间：

```
ALTER FRAGMENT ON TABLE T1 DROP INTERVAL STORE IN (dbs7, dbs8);
ALTER FRAGMENT ON INDEX idx1 DROP INTERVAL STORE IN (dbs7, dbs8);
```

如果 idx1 是表 T1 中连接的索引，那么 PARTITION 关键字是必需的而第二条语句非必要：当修改表分片列表时，数据库服务器会自动修改任一已连接的索引的分片策略以匹配该表已变更的策略。如果这些分片不为空，那么数据库服务器会将来自指定 dbspace dbs7 和 dbs8 的分片移动到其它可用 dbspace 中。

**oracle 模式下范围分区表可以根据分区名称删除分区：**

例如如下示例语法：

```
alter fragment on table tab10 drop partition p0;
```

## MODIFY 子句

使用 MODIFY 子句更改表或索引的分片列表中现有分区上的现有分片表达式，或定义、修改或禁用范围区间或滚动窗口分片结构。

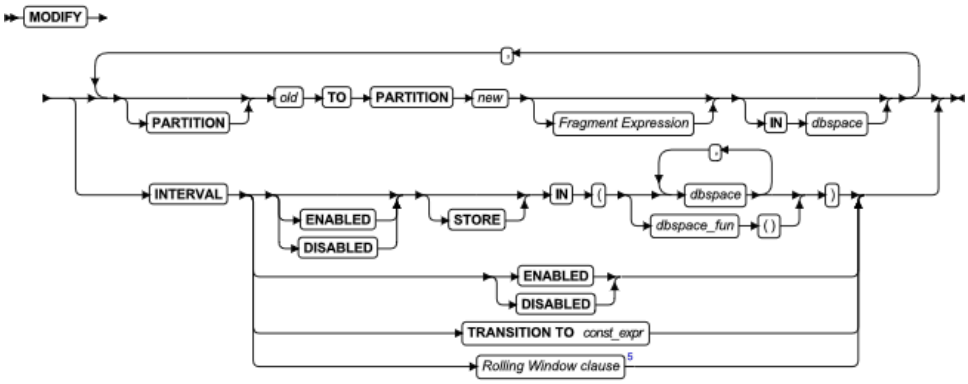
使用 MODIFY 子句更改表或索引的现有分片列表。您可以使用此子句完成以下一个或多个任务：

- 将现有的分片从一个 dbspace 移动到另一个不同的 dbspace 中
- 更改与现有基于列表或基于表达式分片相关的表达式

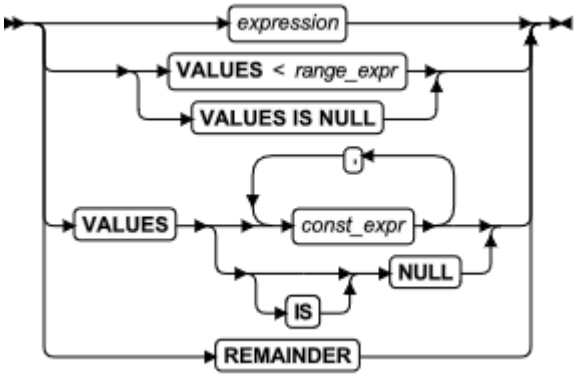
- 更改在范围区间分片列表中定义过渡分片的表达式
- 重命名一个或多个现有的分片
- 启用或禁用区间分片的自动创建方式
- 替换 dbspace 的列表或指定存储新区间分片的函数
- 将一个滚动窗口列表更改为按区间分片的表（没有清除策略）
- 将按范围区间分片的表更改为滚动窗口列表 C
- 按一下一个或多个操作来更改滚动窗口列表的清除策略：
    - 重新设置区间分片的数量的限制
    - 更改对表的已分配的存储大小的限制
    - 替换 ATTACH or DISCARD 关键字选项
    - 替换 ANY 或 INTERVAL FIRST 或 INTERVAL ONLY 关键字选项

ALTER FRAGMENT 语句的 MODIFY 子句具有以下语法：

MODIFY 子句

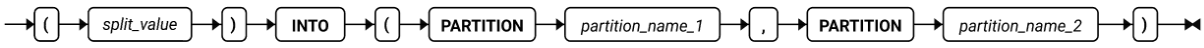



分片表达式

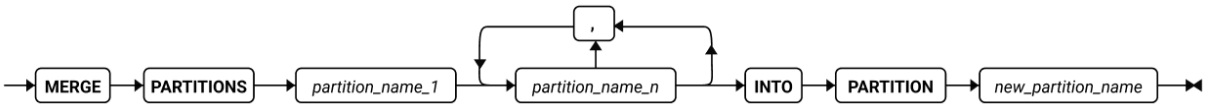



oracle 模式下范围分区表语法有如下修改

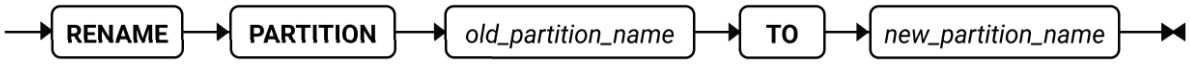
1.表分区拆分语法图:




2.表分区合并语法图:




3.表分区重命名语法图:




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***const_expr*** | 定义要分片要存储的列值或此范围区间分片的新的上限值的常量表达式 | 必须是带引号的字符串或文本值。对于按列表分片，对于同一对象的分片的表达式列表，每个值必须唯一。 | 常量表达式 |
| ***dbspace*** | 存储 ***new*** 分片的 Dbspace | 在执行时必须存在。所有的 dbspace 必须有相同的页大小。 | 标识符 |
| ***dbspace _fun*** | 返回 dbspace 名称的 UDF 名称 | 当数据库服务器调用 URD 为新的分片分配存储时，用户定义函数和返回的 dbspace 必须存在。 | CREATE FUNCTION 语句 |
| ***expression*** | 已修改的表达式 | 仅可以指定当前表中的列，以及单独一行中的数据。 | 条件；表达式 |
| ***new_dbspace*** | 存储系统生成的范围区间分片的 Dbspace | 在执行时必须存在。所有的 dbspace 必须有相同的页大小。 | 标识符 |
| ***new*** | 您在此处为已修改的分片声 | 在分片列表中分片名称必须唯一。如果表和它的索 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 明的 名称 | 引使用相同的范围区间或表分片存储策略，那么每个索引分片必须具有同对应表分片相同的名称。 | |
| ***old*** | 现有分片的名称 | 在分片表中必须存在。对于列表或范围区间分片，PARTITION 关键字必须在此名称之前。 | 标识符 |
| ***range _expr*** | 范围表达式。定义存储在分片中的分片键的上限。 | 必须是恒定文字表达式，其与分片键表达式的数据类型兼容的数字、DATETIME 或 DATE 数据类型。另见范围区间分片的 MODIFY 子句的限制。 | 常量表达式 |
| ***partition_name*** | 分区名称 | 在表中分区名称唯一 | 标识符 |
| ***split_value*** | 拆分值 | 拆分值必须在被拆分范围分区取值范围内 | 标识符 |
| ***partition_name_1*** | 分区名 1 | 分区名在表中唯一 | 标识符 |
| ***partition_name_2*** | 分区名 2 | 分区名在表中唯一 | 标识符 |
| ***partition_name_n*** | 分区名 n | 分区名在表中唯一 | 标识符 |
| ***old_partition_name*** | 老分区名 | 分区名在表中唯一 | 标识符 |
| ***new_partition_name*** | 新分区名 | 分区名在表中唯一 | 标识符 |

## 用法

如果不更改存储位置，那么此处的 ***dbspace*** 和 ***old***（或 ***old*** 和 ***new*** ）可以是相同的。对于按范围区间分片的表或索引，dbspace 指定的列表遵循 STORE IN 关键字替换在声明 ALTER FRAGMENT . . . MODIFY 语句之前的 dbspace 列表。之前 dbspace 列表中的分片不会被此选项重置。

STORE IN 子句可以选择指定返回现有 dbspace 名称的用户定义函数，而不是文字 dbspace 标识列表。此 UDF 声明的标识为随意的。有关此 UDF 及如何创建它的示例的更多信息，请参阅在 CREATE TABLE 主题 Interval fragment 子句 中 STORE IN 子句的讨论。

要使用 MODIFY 子句更改 ***expression*** 和移动它的对应分片到新的存储位置时，您必须更改 ***expression*** 并且必须指定不同于 ***dbspace*** 或分区的名称。

如果同一表或索引的多个分片与 ***dbspace*** 名称相同，那么就必须声明 ***new*** 分片的名称。在范围区间分片的 ***new*** 分片之前必须使用 PARTITION 关键字（但是它对循环分片和基于表达式的分片是可选的）。

***expression*** 必须求出一个 Boolean 值（true 或 false）。

***expression*** 中不允许任何子查询或聚集。此外内置的 **CURRENT** 、**DATE** 、**DBINFO** 、**SYSDATE** 和 **TODAY** 表达式是无效的。

当您使用 MODIFY 子句更改表达式而不变更此表达式的存储位置时，您必须使用与 *old* 分片的和 *new* 的分片的名称相同。然而，如果 dbspace 只由一个单独的分区组成，您可指定参考以下示例为 *old* 和 *dbspace* 指定名称：

```
ALTER FRAGMENT ON TABLE cust_acct
MODIFY dbsp1 TO acct_num < 65 IN dbsp1;
```

对于分片列表策略，如果新的列表表达式与同一表中或索引中的其他分片的现有列表表达式重叠，那么 ALTER FRAGMENT MODIFY 失败并发送错误。

当您使用 MODIFY 子句将一个 dbspace 移动到另一个 dbspace 中时，*old* 是分片以前位置的 dbspace 的名称，*dbspace* 是该分片的新位置。如下所示：

```
ALTER FRAGMENT ON TABLE cust_acct
MODIFY PARTITION part1 TO PARTITION part2 (acct_num < 35) IN dbsp2;
```

以上修改了 **cust_acct** 表的分布方案的 ALTER FRAGMENT 语句 ，这样 **acct_num** 列中小于 35 的所有行项 (先前被分配到分片 **part1** 的存储在 dbspace **dbsp1** 中）将会分配到存储在 dbspace **dbsp2** 中的分片 **part2** 。

当您使用 MODIFY 子句时，底层的 dbspace 不会受到影响。仅会影响到这些分片或 dbspace 中的数据。

如果已经存在非余项分片，除非此分片策略是范围区间策略，您才能重新定义一个非余项分片为余项分片（其他分片的不符合分片键值的行）。然而，如果 REMAINDER 分片中的记录不满足新的 ***expression***，则您无法将该 REMAINDER 分片更改为一个非余项分片。

***attached*** 索引与它的表具有相同的存储分布。如果表上所有的索引是连接的索引，并且您使用 MODIFY 子句修改此表分片，那么数据库服务器会自动修改此索引的存储分布策略以适应新的表分片策略。

***old*** 规范无引用按范围分区存储分布方案分片的表的过渡分片（最后一个范围分片）。该分片唯一有效的修改是使用 TRANSITION TO ***const_expr*** 子句增加此过渡值。有关其它直接尝试重新定义此过

渡分片范围表达式的语法，数据库服务器返回错误。 有关更多信息，请参阅主题使用 MODIFY INTERVAL TRANSITION 选项。

## 范围区间分片的 MODIFY 子句的限制

ALTER FRAGMENT 语句的 MODIFY 子句无法更改此区间值或分片键。要变更此范围区间存储分布方案的任意元素，您必须使用 ALTER FRAGMENT 语句的 INIT 选项。

如果以下任一条件为真，那么 MODIFY 子句不能更改一个分片的范围表达式的值：

- 此分片是最后一个分片，新的值比旧的值小。
- 新的值与现有分片的临值重叠。
- 此分片是系统生成的区间分片。

您可以修改用户定义范围分片的值，但是新的值不能超过相邻分片的界值，而且数据库服务器必须要满足新范围表达式所指示的任何数据移动。

MODIFY 子句能更改存储现有分片的存储空间的列表，并且可以更改将要存储新的系统生成的区间分片的存储空间的列表，但是同样的 MODIFY 子句不能都完成这两个任务。要将两个列表都更改，必须声明两个单独的 ALTER FRAGMENT . . . MODIFY 语句。

类似地，启用或禁用当前范围区间分片方案的 MODIFY 子句不能将现有范围区间分片移动到一个不同的 dbspace 或者创建新的用户定义分片。这些任务都需要单独的 ALTER FRAGMENT . . . MODIFY 语句。

对于表的范围分片和按区间分片的索引，您可以修改首个和中间分片的分片表达式。重叠的分片可通过移动数据来解决，从而使存储在重新定义范围分片中的行的分片键值不会重叠。然而，对于最后一个范围分片，只有此新的范围表达式满足以下条件使，您才可以修改其分片表达式的过渡值：

- 它不部分或完全符合任何现有区间分片表达式。
- 它不会符合任何之后系统自动生成的区间分片表达式。
- 新过渡遗留在分片间的间隙必须是 ***intvl_expr*** 区间值的整数倍。

您不能定义一个按范围区间分片的表的余项分片。

如果您使用 MODIFY 子句重命名现有分片，那么新的名称不能以字符 sys_p 开头。

### 范围、区间和过渡分片

对于使用范围区间存储分片策略的对象，可用于区分三种类型的分片：

- ***range*** 分片是它的的名称、分片键表达式和存储位置在表或索引定义中被定义的分片。范围区间分片需要定义至少一个范围分片。
- ***interval*** 分片是它的的名称、分片键表达式和存储位置是由数据库服务器插入或加载操作试图存储分片键值对现有的分片的分片键值表达式为 false 的行时自动定义的分片。
- 范围分片在 VALUES 子句的上限值比任一其他名为 ***transition*** 分片的范围分片的分片键表达式都大。 过渡分片的 VALUES 子句中指定的上限被称为表的 ***transition value***。如果没有为此对象创建区间分片，那么要添加比过渡值大的分片键值时则需要数据库服务器创建新的区间分片。

在过渡分片上执行的 ALTER FRAGMENT 语句的 MODIFY 子句的操作比其它范围和区间分区的 MODIFY 操作更受限制。

ALTER FRAGMENT MODIFY 语句不能更改定义一个过渡分片的范围表达式，除非它包含 MODIFY TRANSITION 关键字。

数据库服务器不能创建区间分片除非在表或索引定义中的 Interval Fragment 子句定义了范围区间分片键，并且此该分片结构没有被 ALTER FRAGMENT . . . MODIFY INTERVAL DISABLE 语句禁用。

**修改滚动窗口表的限制**

ALTER FRAGMENT MODIFY INTERVAL 语句的 Rolling Window 子句不能在具有以下属性的表上定义清除协议：

- 该表有一个 ROWID 隐藏列。
- 另一个表有引用该表中的 PRIMARY KEY 的外键约束。

## Rolling Window 子句

可以使用 ALTER FRAGMENT ON TABLE . . . MODIFY INTERVAL 语句的 Rolling Window 子句来修改或删除一个滚动参考表的现有清除策略。该子句也可以将不具有分布存储策略或不具有范围区间策略或具有范围区间策略（没有清除策略）的表更改为滚动窗口表。

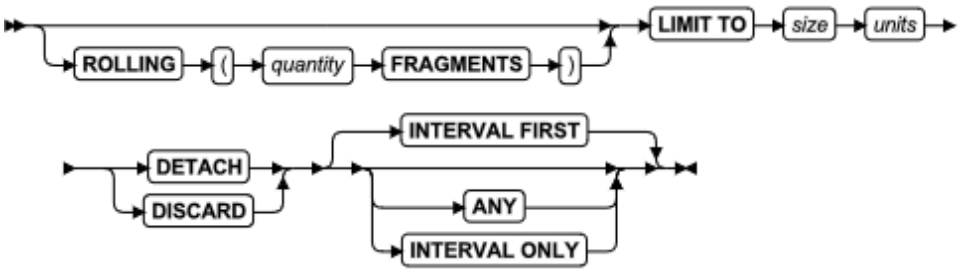
**语法**

ALTER FRAGMENT 语句的 Rolling Window 子句支持以下语法。

**ALTER FRAGMENT 的 Rolling Window 子句**

1：删除所有滚动间隔分片


2：限制最大分配存储大小




3：只限制间隔分片的数量


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *quantity* | 滚动的区间分片的最大数 | 必须是比零打的整数。用户定义的分片不在此限制内。 | 整型字符 |
| *size* | 分配给表和它的索引的存储的上限 | 必须比零大 | 整型字符 |
| *units* | 表的总存储量的单位的简写 | 必须是 K 、KB 、KiB 、M 、MB 、MiB 、G 、GB 、GiB 、T 、TB 、TiB（或这些字符的小写）。任何尾随字符会导致语法错误。 | 以字母 K 、M 、G 或 T 开头的不带引号的字符串 |

**用法**

ALTER FRAGMENT MODIFY INTERVAL 语句中的 Rolling Window 子句的语法支持之前 CREATE TABLE FRAGMENT BY INTERVAL 语句中的 Rolling Window 子句的语法。

**修改滚动窗口表**

ALTER FRAGMENT MODIFY INTERVAL 语句的 Rolling Window 子句类似于它的语法，但是它不等同于 CREATE TABLE 语句的 Rolling Window 子句。ALTER FRAGMENT 的 Rolling Window 子句支持以下功能：

- 您可以为使用范围区间分片的表定义清除策略。
- 您可以按下列变更修改现有清除策略：
  - 更改 ***quantity*** 的 ROLLING FRAGMENTS 值
  - 更改 ***size*** 的 LIMIT TO 值
  - 用 DETACH 或 DISCARD 关键字替换 DETACH 或 DISCARD 关键字。
  - 替换 ANY 或 INTERVAL FIRST 或 INTERVAL ONLY 关键字选项。

  如果您删除 ANY 或 INTERVAL FIRST 或 INTERVAL ONLY 关键字规范而没有替换，缺省的清除策略操作是 INTERVAL FIRST。（有关更多信息，请参阅 Interval fragment 子句。）
- 您可以指定 INTERVAL DISABLED 关键字禁用滚动窗口表的区间分片，从而中止它的清除策略。
- 您可以指定 INTERVAL ENABLED 关键字恢复一个表的区间分片（并重启此清除策略）。该表的区间分片和创建和滚动分片的归档和重建已被禁用。
- 您可以指定 DROP ALL ROLLING 关键字来去除现有的清除策略。其结果是将滚动参考表更改为按区间分片的表。

如果您打算暂时中止当前的清除策略，再随后恢复该相同的清除策略，则应该使用 INTERVAL DISABLED 关键字而不是 DROP ALL ROLLING 关键字。

**强制执行清除策略**

当已分配的总存储大小或区间分片的总量超出 Rolling Window 子句指定的限制时，滚动窗口表的清除策略不会立即强制执行。

清除策略被设计为在滚动窗口表的分片上的所需 DETACH 和 ATTACH 操作不可能与并发用户的访问尝试冲突时作为 Scheduler 任务每天强制执行。缺省情况下，清除策略会每天的本地时间 00:45 时强制执行。有关更多信息，请参阅 *GBase 8s 管理员指南* 中的 Scheduler 的内置的 purge_tables 任务。

清除策略也可以通过执行 syspurge() 系统函数而手动强制执行。在 DBA 调用 syspurge() 函数之后，数据库服务器会检查系统目录，并标识任何清除策略已超出的滚动窗口表。然后数据库服务器会按照清除策略指定丢弃或拆离合格的滚动分片直到满足此清除策略，或直到没有可移除的滚动分片。syspurge() 函数不需要参数，但是接受可启用联机日志诊断的可选参数。

只有具有 DBA 存取权限的用户才能调用实行 DETACH 或 DISCARD 选项以拆离滚动分片的例程。具有 RESOURCE 存取权限的用户可以执行 syspurge() 函数，但是这只能对它们所拥有的表的强制执行清除策略。

数据库服务器会默默地忽略 syspurge() 函数关于高可用数据复制集群（HDR）环境中的辅助服务器的调用。类似地，在 grid 环境下，不会强制执行已复制的表的清除策略。这是因为 grid 环境和集群环境不会复制 DETACH 和 DISCARD 选项触发的 ALTER FRAGMENT 更改，这是滚动窗口清除策略的核心。

Rolling Window 子句提供两个关键字选项以处理拆离的滚动区间分片：

- 使用 DETACH 将该分片连接到数据库服务器自动创建的非独立的表中，并且它们的表标识具有以下格式：

  < original_table_name >_< lower value >_< higher value >

  此处 ***lower_value*** 和 ***higher_value*** 是该分片在被拆离前，其范围区间的最小和最大值。

  如果表的名称已存在，那么在 higher value 后附加一个数字计数器，以 _1 开始表示第一个附加表：

  < original_table_name >_< lower value >_< higher value >_1

  以此类推，将 _2 附加在下一个表的名称后（或附加一个更大的整数，如果附加 _2 没有产生唯一的表名称。）

- 使用 DISCARD 销毁已拆离的分片。

  DISCARD 关键字指定删除已拆离的分片，因此当强制执行清除策略时，会及时地删除非必须的数据记录。通过这种方法，滚动分片的数量或滚动窗口表的存储空间总量会约束到规定值。

**滚动窗口的限制**

ALTER FRAGMENT MODIFY 语句无法使用 Rolling Window 子句将具有以下属性的表更改为滚动窗口表：

- ROWID 列
- 一列或多列定义为一个参考约束的主键

- 已拆离的索引（即，一个索引存储分布方案与该表的分片存储策略不同）

类似地，ALTER TABLE 语句不能为滚动窗口表添加 ROWID 列或主键约束。

- Rolling Window 子句为滚动窗口定义的清除策略要求数据库服务器在分片上执行的 ALTER FRAGMENT DETACH 操作必须满足 DETACH 或 DISCARD 标准。然而，对于包含由一个启动的外键约束引用的主键的表，或包含 ROWID 的表，不允许 ALTER FRAGMENT DETACH 语句。因此 CREATE TABLE and ALTER FRAGMENT ON TABLE . . . MODIFY INTERVAL 语句无法定义或修改具有主键约束或 ROWID 隐藏列表的清除策略。
- 滚动窗口表上定义的任何索引必须具有与该滚动窗口相同的范围区间存储分布。

## 使用 MODIFY INTERVAL TRANSITION 选项

您可以使用该选项添加有范围区间分片结构的表的最后一个范围分片的过渡值。此过渡值不会被使用 ALTER FRAGMENT 语句的 MODIFY INTERVAL TRANSITION 选项减少。

您不能使用 MODIFY 选项的 PARTITION ***partition*** VALUES 语法修改一个使用范围区间存储分布方案的表的最后一个范围分片（也称为 ***transition fragment***）的范围表达式。然而过渡值（此范围表达式的上限）会在使用 MODIFY INTERVAL TRANSITION TO 关键字指定新的上限时增加。当过渡值更改后不会有数据移动。

要减少过渡值（通过重置过渡分片范围的上限），您必须执行 ALTER FRAGMENT INIT 操作以重新定义该表的范围区间分布存储方案。

**过渡值增加时自动重命名分片**

指定 MODIFY INTERVAL TRANSITION 的 ALTER FRAGMENT 语句会导致重命名现有的分片：

- 如果在新的和旧的过渡值之间没有区间分片，但是区间分片早已超出新的过渡值，系统生成的区间分片名称的最终的数字将按区间分片边界值除以新过渡值和旧过渡值之间的差额的值减少。

  例如，如果区间值表达式定义了一个等于 20 的区间大小，且旧过渡值和新过渡值之间相差 60，那么名为 sys_p7 的区间分片将会被重命名为 sys_p4，因为它的商为 (60/20) = 3 。
- 如果在新的和旧的过渡值之间存在区间分片，那么字符 rg 会附加在它们的名称上以标识它们成为了范围分片，因为它们分片表达式的上限不再大于该表的过渡值。

  例如 ，如果一个表插入的过渡值符合其区间分片 **sys_p5** VALUES 值的上限，那么此分片将更改为范围分片，并重命名为 **sys_p5rg**（它也是过渡分片）。如果另一个称为 **sys_p4** 的区间分片也有一个小于 VALUES 上限值在它的分片表达式中，那么此分片也会变为范围分片，并重命名为 **sys_p4rg**。

在分片重命名操作过程中，当更新 sysfragments 系统目录表时，会在此分片上放置一个互斥锁。

- **partition** 列中带有新分片标识符值，
- 对于任何区间分片或滚动分片（在当前 ALTER FRAGMENT MODIFY 操作过程中，滚动区间分片的在分片列表中的初始位置已变更的滚动区间分片）的 **evalpos** 列包含新的整型值。

在以上列出的情况中，一些分片被重命名以确保分片列表中的每个分片是唯一的，并保持为区间分片系统生成的名称和在系统目录中这些分片对应的 **sysfragments.evalpos** 值之间的相关性。（另见自动重命名区间分片标识符。）

以下的一些 ALTER FRAGMENT 示例会阐述该分片重命名行为。

**ALTER FRAGMENT MODIFY INTERVAL TRANSITION 的示例**

以下语句定义了一个使用范围区间春初分布方案的已分片的表 **tabtrans** ，该表包含整型列 **i** 作为主键，和区间值 100 。过渡分片 **p2** 有过渡值 300 ，意味着数据库服务器将在对表的任何操作中定义一个新的区间分片以存储分片键值大于等于 300 的新行。

```
CREATE TABLE tabtrans (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;   -- last range fragment (also
        -- called transition fragment)
```

以下示例基于此 **tabtrans** 表。

下列 ALTER FRAGMENT 语句试图将过渡分片值从 300 增加到 250 ：

```
ALTER FRAGMENT ON TABLE tabtrans
            MODIFY INTERVAL TRANSITION TO 250;
```

此语句失败，因为它试图减少该过渡值。如果目标是保持当前的区间值 100 ，但是对于新过渡值则变为 250 ，那么需要 ALTER FRAGMENT INIT 操作重新定义该范围分片。为了保持范围分片的边界对齐，该范紧接的过渡分片的范围分片的新上限值必须为 150。在新的分布存储方案中，如果要插入分片键值大于 250 的行，那么数据库服务器会生成一个新的范围为 100 的区间分片，之前的整数值 50 （模 100）作为上限.

如果新的过渡值和旧过渡值之间没有区间分片，则数据库服务器更新最后一个范围分片的表达式为 VALUES < *new*（*new* 是新过渡值）：

```
INSERT INTO tabtrans VALUES (601, "BB");   -- creates interval fragment sys_p6
-- with fragment expression >= 600 AND < 700
```

tabtrans 表的分片列表和分片表达式变为以下：

```
  p0          VALUES < 100                    - range fragment
  p1          VALUES < 200                    - range fragment
  p2          VALUES < 300                    - last range (or transition) fragment
sys_p6      VALUES >= 600 AND VALUES < 700 - interval fragment
```

此处，系统生成的新区间分片的名称是 **sys_p6** ，因为 6 是系统目录中新分片的 **sysfragments.evalpos** 值。**evalpos** 值 7 和 5 会被保留（还未创建）以便区间分片存储分片键符合分片表达式 VALUES >= 300 AND VALUES < 400 and VALUES >= 400 AND VALUES < 500 的行，根据表的当前过渡值和 FRAGMENT BY 子句中 INTERVAL (100) 规范定义了该表的分片方案。

在变更此过渡值的过程中，此分片被更改为不产生数据移动。以下语句成功地将过渡值更改为 500 。

```
ALTER FRAGMENT ON TABLE tabtrans
        MODIFY INTERVAL TRANSITION TO 500;
```

旧过渡值是 300 新过渡值是 500，之间没有区间分片。第一个区间分片以 600 开始。也就是说没有在 300 和 500 之间的数据。因此最后一个范围分片（过渡分片）的表达式可变更为 VALUES < 500 而不需数据移动。因为在新过渡值后有区间分片，所以该新过渡值必须与区间分片边界对齐。在以上列子中，新过渡值 500 与区间分片边界对齐（不论此分片现在是否存在），作为修改的结果，区间分片系统目录中的 evalpos 值改变，且会重命名区间分片以符合 sys_p*evalpos* 名称的格式。

已修改的表具有以下分片：

```
  p0       VALUES < 100   -- range fragment
  p1       VALUES < 200   -- range fragment
  p2       VALUES < 500   -- last range fragment (= transition fragment
        -- with its expression modified)
  sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment (renamed
        -- to sys_p4 as evalpos changes from 6 to 4
        -- after the transition fragment change)
```

以下修改失败并产生了错误，因为有超出新过渡值的区间分片，新过渡值不与区间分片的临界值相等：

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL TRANSITION TO 550;
```

区间分片的可能值为 300 到 400 、400 到 500 、500 到 600 、600 到 700 等等。新的过渡值 550 不在区间分片界限上，因此产生了错误。

如果在新和旧过渡值之间有区间分片，那么新过渡值必须对齐区间分片边界（该区间分片不须存在），除非新过渡值超出了最后一个区间分片的范围。在新和旧过渡值之间的所有区间分片都会转换为范围分片，并且它们的表达式会修改为符合范围分片表达式的格式。最后一个区间分片的表达式将转化为一个 VALUES < ***new*** 的范围分片（ ***new*** 是新过渡值）。

此处是在新的区间分片中另一 INSERT 操作产生的示例：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0; -- last range fragment
        -- or transition fragment

        INSERT INTO tab
        VALUES (301, "AA"); -- creates interval fragment sys_p3 with
        -- fragment expression >= 300 AND < 400
        INSERT INTO tab
        VALUES (601, "BB"); -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
```

该表的分片表包含这些分片：

```
  p0          VALUES < 100        -- range fragment
  p1          VALUES < 200        -- range fragment
  p2          VALUES < 300        -- range fragment
   sys_p3      VALUES >= 300 AND VALUES < 400 -- interval fragment
   sys_p6      VALUES >= 600 AND VALUES < 700 -- interval fragment
```

随后的 ALTER FRAGMENT 示例都基于以上语句。

以下语句将过渡值从 300 增加到 500：

```
    ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL TRANSITION TO 500;
```

因为在就的和新的过渡值之间有一个区分片（**sys_p3**），此分片被转换为一个范围分片（表达式变为 < 400）。因为还有一个超出新过渡值的区间分片（**sys_p6**），所以新过渡值必须对齐区间分片边界，是 INTERVAL(100) 规范的整数倍。即，此处区间分片可能为 300 到 400 、400 到 500 、500 到 600 、600 到 700 等等。新过渡值 500 在区间分片的临界（该区间分片不必存在）。我们也不用在变更过渡值或创建任一分片的过程中移动数据。这可以通过以下操作完成：将分片 **sys_p3** 转换为新过渡值分片，更新它的表达式为 < 500 （因为它现在是范围分片）并重命名。

生成的表的分片表包含这些分片：

```
  p0            VALUES < 100   -- range fragment
  p1            VALUES < 200   -- range fragment
  p2            VALUES < 300   -- range fragment (was the old transition fragment)
sys_p3rg       VALUES < 500   -- range fragment (was previously interval
                -- fragment sys_p3. Its expression was modified to a
                -- range expression. Its name was changed to a
                -- system-generated name in format sys_p<evalpos>rq )
                -- becomes the new transition fragment
 sys_p5        VALUES >= 600 AND VALUES < 700
                -- interval fragment (renamed to sys_5 brcause the
                -- evalpos value changes from 6 to 5 after the
                -- transition fragment change.)
```

以下尝试修改过渡值失败，并返回错误：

```
ALTER FRAGMENT ON TABLE tab
            MODIFY INTERVAL TRANSITION TO 550;
```

以上语句失败的原因为：有一个超出新过渡值的区间分片。并且该新过渡值不须对齐区间分片边界。

下一示例将过渡值从 500 增加到 700：

```
ALTER FRAGMENT ON TABLE tab
            MODIFY INTERVAL TRANSITION TO 700;
```

生成的表的分片表包含以下分片：

```
p0            VALUES < 100 -- range fragment
p1            VALUES < 200 -- range fragment
```

```
p2          VALUES < 300 -- range fragment (was the old transition fragment)
sys_p3rg    VALUES < 400 -- range fragment (was previously interval fragment
            -- sys_p3, and its expression changed to a range expression.
            -- The fragment has been renamed to system-generated name
            -- in the format sys_p<evalpos>rg ).
sys_p6rg    VALUES < 700 -- range fragment (was previously the interval
            -- fragment sys_p6. Its expression was modified to a
            -- range expression and its name replaced by a system-
            -- generated name in the format sys_p<evalpos>rg )
            -- becomes the new transition fragment.
```

下一示例将过渡值从 700 增加到 750：

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL TRANSITION TO 750;
```

因为没有超出新过渡值的区间分片存在，所以它不须对齐区间分片边界。

生成的表的分片表包含以下分片：

```
p0          VALUES < 100   -- range fragment
p1          VALUES < 200   -- range fragment
p2          VALUES < 300   -- range fragment (was the old transition fragment)
sys_p3rg    VALUES < 400   -- range fragment (was previously interval
            -- fragment sys_p3. expression modified to a
            -- range expression. Fragment was renamed to a system
            -- generated name in the format sys_p<evalpos>rg)
sys_p6rg    VALUES < 750   -- range fragment (was previously the interval
            -- fragment sys_p6. Its expression was modified to a
            -- range expression, and the fragment was renamed to a
            -- system-generated name in format sys_p<evalpos>rg)
            -- becomes the new transition fragment
```

如果您希望在 MODIFY INTERVAL TRANSITION 操作过程中避免现有分片自动重命名，那么您可以首先使用 ALTER FRAGMENT MODIFY 语句用用户定义的名称重命名可能被 LTER FRAGMENT MODIFY INTERVAL TRANSITION 语句更改的系统生成的名称的区间分片。数据库服务器仅会重命名系统生成的区间分片名称（当创建新区间分片时避免产生不唯一的分片名称）。

## 在 MODIFY 操作中使用 ONLINE 关键字

ONLINE 关键字指示数据库服务器内部提交 ALTER FRAGMENT . . . MODIFY 工作，如果没有错误，在该表上应用意向互斥锁而不是互斥锁。

### ONLINE MODIFY 操作的要求

您只能对按区间分片方案分片的表使用 TER FRAGMENT ONLINE ON TABLE 语句的 MODIFY 选项。

只有过渡值（区间分片的起始值）才能 ONLINE 修改。其它应用在 MODIFY 选项上的限制同样适用于 ONLINE MODIFY 操作。有关这些限制，请参阅 ATTACH 子句的一般限制 和 范围区间分片的 MODIFY 子句的限制 。

**ALTER FRAGMENT ONLINE . . . MODIFY 的示例**

以下 SQL 语句定义了一个已分片的 **employee** 表，它使用范围区间存储分布方案，在列 **emp_id**（也是分片键）上有一个唯一索引 **employee_id_idx** 在列 **dept_id** 上有另一个索引 **employee_dept_idx**。

```
CREATE TABLE employee
        (emp_id INTEGER, name CHAR(32),
        dept_id CHAR(2), mgr_id INTEGER, ssn CHAR(12))
        FRAGMENT BY RANGE (emp_id)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
        PARTITION p0 VALUES < 200 IN dbs1,
        PARTITION p1 VALUES < 400 IN dbs2;
        CREATE UNIQUE INDEX employee_id_idx ON employee(emp_id);
        CREATE INDEX employee_dept_idx ON employee(dept_id);

INSERT INTO employee VALUES (401, "Susan", "DV", 101, "123-45-6789");
INSERT INTO employee VALUES (601, "David", "QA", 104, "987-65-4321");
```

最后两条语句使用超出过渡分片上限的分片键值插入了行，这导致数据库服务器生成了两个新区间分片，以致于生成包含四个分片的分片列表：

```
Fragments in surviving table before ALTER FRAGMENT ONLINE:
  p0       VALUES < 200                  - range fragment
  p1       VALUES < 400                  - range fragment (transition fragment)
  sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
  sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

以下语句返回了错误，因为过渡值只能被增加。这也是脱机 ALTER FRAGMENT . . . MODIFY 操作的一个限制。

```
ALTER FRAGMENT ONLINE ON TABLE employee
        MODIFY INTERVAL TRANSITION TO 300;
```

以下语句成功运行：

```
ALTER FRAGMENT ONLINE ON TABLE employee MODIFY INTERVAL TRANSITION TO 600;

Fragments in surviving table after ALTER FRAGMENT ONLINE:
p0          VALUES < 200                  - range fragment
p1          VALUES < 400                  - range fragment
sys_p2rg    VALUES < 600                  - range fragment (new transition fragment)
sys_p3      VALUES >= 600 AND VALUES < 700 - interval fragment
```

以下语句同样有效：

```
ALTER FRAGMENT ONLINE ON TABLE employee MODIFY INTERVAL TRANSITION TO 700;
ALTER FRAGMENT ONLINE ON TABLE employee MODIFY INTERVAL TRANSITION TO 900;
```

# 带有区间分片的 MODIFY 子句的示例

本节阐述了对使用范围和区间分片作为其分布策略的表使用 ALTER FRAGMENT 语句的 MODIFY 子句的语法的功能和 MODIFY 子句可更改内容的限制。

有关按列表分片的表使用 MODIFY 子句的类似的示例，请参阅列表分片的 MODIFY 子句的示例 。

**启用或禁用范围区间分片**

此语句禁用范围区间分片的创建：

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL DISABLED;
```

以下语句恢复范围区间分片的创建，撤销了前一个例子的作用：

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL ENABLED;
```

以下语句禁用范围区间分片创建，并修改了 dbspace 列表（在 STORE IN 语句中表明了存储新分片的 dbspace ）。如果随后的 ALTER FRAGMENT MODIFY 语句启用了 **tab** 表的范围区间分片创建功能。

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL DISABLED
        STORE IN (dbs4, dbs5);
```

**在范围区间分片中重命名分片**

此语句重命名两个范围区间分片。没有指定新存储位置的 IN 子句，因此两个分片新的名称替代了现有的名称：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p1 TO PARTITION newp1,
        PARTITION sys_p6 TO PARTITION newsys_p6;
```

范围区间分片需要 PARTITION 关键字。如果您使用 MODIFY 子句重命名现有的分片，那么在 MODIFY 子句中声明的新的名称不能以字符串 `sys` 开头（该字符串用于系统定义的分片），以上示例成功地重命名了系统定义的分片 **sys_p6** 。

**重新定位范围或区间分片**

假设下表有范围区间分片并接收了来自插入操作的两行：

```
CREATE TABLE tab2 (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1;

        INSERT INTO tab2 VALUES (201, "AA");
        -- creates a system-generated interval fragment sys_p2
        -- with fragment expression >= 200 AND < 300
        -- assume that this fragment is created in dbs1

        INSERT INTO tab2 VALUES (601, "BB");
```

```
-- creates a system-generated interval fragment sys_p6
-- with fragment expression >= 600 AND < 700
---assume that this fragment is created in dbs2
```

以下语句标示了数据库服务器将范围分片 **p1** 从 dbs1 移动到 dbs2：

```
ALTER FRAGMENT ON TABLE tab2 MODIFY
        PARTITION p1 TO PARTITION p1 IN dbs2;
```

下一示例将范围分片 **p1** 从 dbs1 移动到 dbs2 并将区间分片 **sys_p6** 从 dbs2 移动到 dbs3：

```
ALTER FRAGMENT ON TABLE tab2 MODIFY
        PARTITION p1 TO PARTITION p1 IN dbs2,
        PARTITION sys_p6 TO PARTITION sys_p6 IN dbs3;
```

**替换存储新区间分片的 dbspace 列表**

以下 CREATE TABLE 语句定义了一个范围区间分片策略，其中：

- 列 **i** 是分片键，
- 100 是范围区间的大小，
- 新分片将存储在 dbspace dbs1 、dbs2 和 dbs3 ，
- 初始分片 **p0**（在 dbspace dbs0 中），**p1** （在 dbspace dbs1 中）的过渡值分别为 100 和 200。

```
CREATE TABLE tab (i INT, c CHAR(2))
                FRAGMENT BY RANGE (i)
                INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
                PARTITION p0 VALUES < 100 IN dbs0,
                PARTITION p1 VALUES < 200 IN dbs1;
```

下列 ALTER FRAGMENT 语句在 STORE IN 中用新的列表 （dbs4 、dbs5）替换了列表（dbs1 、dbs2 、dbs3）。

```
ALTER FRAGMENT ON TABLE tab
        MODIFY INTERVAL STORE IN (dbs4, dbs5);
```

上个示例中，MODIFY 子句指定了新的分片将会轮流创建于 dbs4 和 dbs5 中。任何创建在最初 STORE IN 列的 dbspace （dbs1 、dbs2 、dbs3） 中的系统定义的分片（和分片 **p1**） 仍保留在这些 dbspace 中。现有的和随插入行之后的分片键在这些分片范围区间内的分片仍将继续存储在这些分片中，但是将会创建新的区间分片，轮流地存储于 dbs4 和 dbs5 dbspace 中。

考虑以下分片表：

```
CREATE TABLE mytab (col1 int)
                FRAGMENT BY RANGE (c1) INTERVAL (100)
                STORE IN (dbs1, dbs2, dbs3, dbs4, dbs5)
                PARTITION p1 VALUES < 300 in dbs0;
```

此 ALTER FRAGMENT 语句替换了存储新区间分片的 dbspace 列表：

```
ALTER FRAGMENT ON TABLE mytab MODIFY
        STORE IN (dbs1, dbs6, dbs3, dbs4, dbs8);
```

新的列表用 dbs6 代替了 dbs2 ，dbs8 替换了 dbs5。如果您希望来自当前 STORE IN 列表的任何 dbspace 可用于新的分片，那么 MODIFY 子句必须在新的列表（在已修改的分片存储方案中替换了旧的列表）中包含它们。在上述示例中，新区间分片将在 STORE IN 关键字之后列出的五个 dbspace 中创建 ，但是任何创建于 dbs2 和 dbs5 中的现有的分片将继续存储数据值符合分片键值范围的分片的行。

您可以在 STORE IN 子句中修改 dbspace 的列表。旧的列表会被您指定的新的列表所替换。不会移动旧的 dbspace 中的分片。考虑下表：

您可以通过更改该分片的 IN ***dbspace*** 规范而将现有分片移动到另一个 dbspace ：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1;

        INSERT INTO tab VALUES (201, "AA");
        -- creates interval fragment sys_p2
        -- with fragment expression >= 200 AND < 300
        -- (assume that this fragment is created in dbs1)
        INSERT INTO tab VALUES (601, "BB");
        -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
        -- (assume that this fragment is created in dbs2)
```

下一个语句指示数据库服务器将分片 p1 从 dbs1 移动到 dbs2 ：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p1 TO PARTITION p1 IN dbs2;
```

以下示例将范围分片 p1 从 dbs1 移动到 dbs2 ，并将区间分片 sys_p6 从 dbs2 移动到 dbs3 ：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p1 TO PARTITION p1 IN dbs2,
        PARTITION sys_p6 TO PARTITION sys_p6 IN dbs3;
```

然而，当系统生成分片后，您不能修改该区间分片的表达式。考虑此表：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1;

        INSERT INTO tab VALUES (201, "AA");
        -- creates interval fragment sys_p2
        -- with fragment expression >= 200 AND < 300
        INSERT INTO tab VALUES (601, "BB");
        -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
```

现在您不能修改 sys_p2 或 sys_p6 的分片表达式。如果您尝试修改，则会返回错误。

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION sys_p6 TO PARTITION sys_p6
        VALUES < 900 IN dbs2;
```

上述语句失败并发生了错误。

**修改定义范围分片的表达式**

在某些情况下，您可以使用 MODIFY 子句更改定义范围分片的表达式。以下示例说明了您可以对该表达式做出更改的种种的限制。然而，在系统生成此区间分片后，您就不能修改此范围分片的表达式。考虑下表：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1;

        INSERT INTO tab VALUES (201, "AA");
        -- creates interval fragment sys_p2
        -- with fragment expression >= 200 AND < 300
        INSERT INTO tab VALUES (601, "BB");
        -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
```

现在您不能修改区间分片 sys_p2 或 sys_p6 的表达式。如果您尝试修改，则会返回错误。

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION sys_p6 TO PARTITION sys_p6
        VALUES < 900 IN dbs2;
```

以上语句因产生错误而失败。

您可以修改第一个中间范围分片，但是替换的表达式不能跨越相邻分片的边界。此操作会导致数据移动。示例如下：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;

        INSERT INTO tab VALUES (301, "AA");
        -- creates interval fragment sys_p3
        -- with fragment expression >= 300 AND < 400
        INSERT INTO tab VALUES (601, "BB");
        -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
```

以下所有的 ALTER 示例都基于上述 CREATE 语句中定义的表的分片。下列 ALTER FRAGMENT 语句修改了范围分片 p0 的表达式：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0
        VALUES < -50 IN dbs0;
```

以下语句修改了分片 p0 的表达式并将此分片从 dbs0 移动到 dbs5 ：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0
        VALUES < -50 IN dbs5;
```

下列语句成功地完成了对分片 p0 的三次更改：

- 修改了 **p0** 的分片表达式，
- 修改了 **newp0** 的分片名称，
- 并将已命名的分片从 dbs0 移动到 dbs5。

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION newp0
        VALUES < -50 IN dbs5;
```

然而，下一示例因产生错误而失败，因为分片 p0 的新表达式超越了下一个相邻分片 p1 的范围的边界：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0
        VALUES < 250 IN dbs0;
```

以下 ALTER FRAGMENT 示例成功地修改了范围分片 p1 的表达式：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p1 TO PARTITION p1
        VALUES < 150 IN dbs1;
```

以下修改因产生错误而失败，因为分片 p1 的新表达式超越了前一个相邻分片 p0 的范围的边界：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0
          VALUES < 50 IN dbs0;
```

出于某种原因，ALTER FRAGMENT MODIFY 操作失败，行无法移动到新的分片中，并返回了错误。示例如下：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES IS NULL IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;

        ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0
```

```
        VALUES < 100 IN dbs0;
```

由于进行了修改，生成的表具有以下分片：

```
PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0
```

如果之前的 NULL 分片存储了行（意味着表中的列 i 的有 NULL 值行），那么这些行不适合此新分片结构中的任何分片。当移动行时，上述的 ALTER FRAGMENT 操作将因此失败。

注意该 NULL 分片也是表中的第一个分片。即使在 CREATE TABLE 或 ALTER TABLE 操作中用户指定此 NULL 分片作为最后一个分片，它会重新分配为该表中的首个分片，并有分片列表中的最小的 evalpos 值。当修改第一个和中间范围分片时，数据库服务器会施加新表达式不能超过相邻分片边界的限制。因此当修改 NULL 分片时，您指定的任何表达式都不能超过下一个范围或区间分片的边界。示例如下：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES IS NULL IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;
```

假设该表在分片 p0 中没有行。在这种情况下，p0 能被更改为非 NULL 分片。

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0
        VALUES < 250 IN dbs0;
```

然而，因为 p0（VALUES < 250）的新的表达式超越了 p1（VALUES < 200）的临界，以上示例返回了错误。

以下 ALTER FRAGMENT 语句可能是：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p0 TO PARTITION p0 VALUES < 150 IN dbs0;
```

您可以修改最后一个范围分片的表达式（过渡分片）但是只能增加过渡值。在此操作中没有数据移动。

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;
        -- last range fragment or transition fragment
```

以下修改返回错误。因为它尝试减少过渡值（从 300 变更为 250）：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p2 TO PARTITION p2
```

```
        VALUES < 250 IN dbs0;
```

以下语句修改了 p2（过渡分片）的分片表达式。因为还没有系统生成的区间分片，新的过渡值不需要与区间分片标记对齐。

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p2 TO PARTITION p2
        VALUES < 350 IN dbs0;
```

如果在新的和旧的过渡值之间没有区间分片，您可以将最后一个范围分片的表达式修改为 VALUES < *new transition value*。示例如下：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;
        -- last range fragment is the "transition fragment"

        INSERT INTO tab VALUES (601, "BB");
        -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
        -- (assume that this fragment is created in dbs3)
```

已修改的表现在具有这些分片：

**分片**

| **分片** | **表达式和分片类型** |
| --- | --- |
| p0 | VALUES < 100 – range fragment |
| p1 | VALUES < 200 – range fragment |
| p2 | VALUES < 300 - last range fragment (or transition fragment) |
| sys_p6 | VALUES >= 600 AND VALUES < 700 - interval fragment |

在更改过渡值的过程中，分片以不会产生数据移动的方式修改。

下列语句修改了 p2 （过渡分片或最后一个范围分片）的分片表达式。

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p2 TO PARTITION p2 VALUES < 500 IN dbs0;
```

旧的过渡值是 300 ，新的过渡值是 500 。在这些范围分片中没有系统生成的区间分片，第一个区间分片的起始值为 600 。这也意味着在 300 和 500 之间没有数据行，因此此过渡分片（最后一个范围分片）的表达式可以修改为 VALUES < 500 而不用数据移动。因为在新的过渡值后没有区间分片，所以新过渡值必须与区间分片边界对齐。在这种情况下，新的过渡值 500 与区间分片的边界对齐（此区间分片不须存在）。

此修改的结果是，随后的区间分片的 **evalpos** 值改变，区间分片重命名为符合系统生成的分片的名称的格式。此 ALTER TABLE MODIFY 操作后，产生的表具有这些分片：

**分片　　表达式和分片类型**

p0　　VALUES < 100 – range fragment

p1　　VALUES < 200 – range fragment

p2　　VALUES < 500 – modified expression for transition fragment

sys_p4　VALUES >= 600 AND VALUES <700 – interval fragment

这是分片 **p2** （最后一个范围分片）已修改的表达式。（也就是 ***transition fragment*** ，因为任何存储了大于分片键范围的值的分片将会是系统生成的区间分片。）系统生成的区间分片重命名为 **sys_p4** ，因为过渡分片的表达式更改后，**evalpos** 值从 6 变为 4 。

以下修改由于产生错误而失败，因为有超过新过渡值的区间分片存在，并且新过渡值未与区间分片边界对齐：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p2 TO PARTITION p2 VALUES YY 550 IN dbs0;
```

区间分片的范围可以是 300 到 400 、400 到 500 、500 到 600 、600 到 700 等等，但是新的过渡值 550 不在区间分片的边界，因此数据库服务器会声明错误。

如果在新的和旧的过渡值之间有区间分片，那么新过渡值必须与区间分片的边界对齐（该区间分片不须存在），除非新过渡值超过了最后一个区间分片。所有在新的和旧的过渡值之间的分片会被转换为范围分片，它们的表达式会被修改以符合范围分片表达式。最后一个区间分片的表达式转换为范围分片，更改为 VALUES < ***new transition value*** 。

下例示例可证明此行为：

```
CREATE TABLE tab (i INT, c CHAR(2))
        FRAGMENT BY RANGE (i)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < 100 IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 300 IN dbs0;
        -- last range fragment or transition fragment

        INSERT INTO tab VALUES (301, "AA");
        -- creates interval fragment sys_p3
        -- with fragment expression >= 300 AND < 400
        -- (assume this fragment is created in dbs1)
        INSERT INTO tab VALUES (601, "BB");
        -- creates interval fragment sys_p6
        -- with fragment expression >= 600 AND < 700
        -- (assume this fragment is created in dbs3)
```

在两个 INSERT 操作之后，该表将具有这些范围和区间分片：

| 分片 | 表达式和分片类型 |
|---|---|
| p0 | VALUES < 100 – range fragment |
| p1 | VALUES < 200 – range fragment |
| p2 | VALUES < 300 – range fragment |
| sys_p3 | VALUES >= 300 AND VALUES <400 – interval fragment |
| sys_p4 | VALUES >= 600 AND VALUES <700 – interval fragment |

以下的 ALTER FRAGMENT 示例基于此表。

以下示例修改了分片 **p2**（过渡分片）的表达式：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p2 TO PARTITION p2 VALUES < 500 IN dbs0;
```

因为在旧过渡值和新过渡值之间有一个区间分片（**sys_p3**），所以此分片转换为范围分片（其表达式变为 VALUES < 400）。

而且由于有超过新过渡值的区间分片（例如：分片 **sys_p6**），新过渡值必须与区间分片边界对齐，可能的区间分片必须是范围区间大小的整数倍（包括 400 到 500 、500 到 600 、700 到 800 等待）。新过渡值是 500 ，是一个区间分片的临界。它也可以在更改过渡分片期间有效避免移动数据和避免创建分片。这可能通过以下操作实现，将分片 **sys_p3** 转换为新的过渡分片，将它的表达式变更为 < 500 ，并重命名旧过渡分片的名称。

生成的表具有以下分片：

| 分片 | 表达式和分片类型 |
|---|---|
| p0 | VALUES < 100 – range fragment |
| p1 | VALUES < 200 – range fragment |
| sys_p2rg | VALUES < 300 – range fragment （这是旧的过渡分片，现在重命名为 **sys_p2rg** 。系统生成的格式 **sys_p*evalpos*rg**.) |
| p2 | VALUES <500 - range fragment （这是之前的区间分片 **sys_p3** 。它的表达式被修改为范围表达式。现在定义了新过渡分片） |
| sys_p5 | VALUES >= 600 AND VALUES <700 – interval fragment（ 重命名为 **sys_5** ，它的 **evalpos** 值在过渡分片该表后从 6 改为 5 ） |

以下对过渡分片 **p2** 的修改返回了错误：

```
ALTER FRAGMENT ON TABLE tab MODIFY
    PARTITION p2 TO PARTITION p2 VALUES < 550 IN dbs0;
```

出现此错误的原因是有一个超过过渡值的区间分片 **sys_p6**，并且新过渡值不与区间分片边界对齐。

下一示例修改了分片 **p2** （过渡分片）的表达式：

```
ALTER FRAGMENT ON TABLE tab MODIFY
    PARTITION p2 TO PARTITION p2 VALUES < 750 IN dbs0;
```

因为没有区间分片超过新过渡值，它不须与区间分片边界对齐。

产生的表具有以下分片：

| 分片 | 表达式和分片类型 |
|---|---|
| p0 | VALUES < 100 – range fragment |
| p1 | VALUES < 200 – range fragment |
| sys_p2rg | VALUES < 300 – range fragment （这是旧的过渡分片，现在按照系统生成的格式 **sys_p*evalpos*rg** 重命名为 **sys_p2rg**。） |
| sys_p3rg | < 400 – range fragment （这是之前的区间分片 **sys_p3** ，在它的表达式修改为一个范围表达式之前。） |
| p2 | VALUES <750 - range fragment （之前的分片 **sys_p6**，在它的表达式修改为一个范围表达式之前。成了新的过渡分片。） |

## 列表分片的 MODIFY 子句的示例

您可以使用 MODIFY 子句对按列表分片的表或索引的分片进行更改，包括以下更改：

- 更改现有列表分片的名称
- 将现有列表分片的存储位置移动到另一个 dbspace
- 更改一个或多个列表分片的表达式

以下 ALTER FRAGMENT ON TABLE 语句更改了按列表分区的表的分片的名称、分片表达式列表和其存储位置：

```
ALTER FRAGMENT ON TABLE T2 MODIFY
        PARTITION part1 TO PARTITION part11
        VALUES ('CA', 'OR', 'TX') IN dbs1;
```

这里将分片名称 part1 更改为 part11，将值 'TX' 添加到此分片的表达式的列表中，并且将已命名的分片移动到 dbspace dbs1 中。

以下示例说明了带有列表分片方案的 MODIFYE 子句的这些和其它用途，也说明了由于列表分片的逻辑限制会使 MODIFY 操作失败。

假设该 CREATE TABLE 语句定义了以下结构的表 **myTable** ，且此表有列表分片策略：

```
CREATE TABLE myTable (i int, c char(2))
        FRAGMENT BY LIST (c)
        PARTITION p1 VALUES ("AB", "CD") IN dbs1,
        PARTITION p2 VALUES ("PQ", "RS") IN dbs2,
        PARTITION p3 REMAINDER IN dbs3;
```

下一 ALTER FRAGMENT 语句修改了 **p2** 分片的存储分片策略：

```
ALTER FRAGMENT ON TABLE myTable MODIFY
      PARTITION p2 TO PARTITION newp2
       VALUES (NULL) IN dbs5;
```

上述语句对该分片及其存储分布具有以下影响：

- 重新定义了 **p2** 分片的分片表达式，将其变为 NULL 分片，
- 将此分片的名称更改为 **newp2**，
- 将此分片的存储位置从 dbs2 移动到 dbs5 ，
- 将存储在 **p2** 分片中的现有数据行移动到余项分片 **p3** 中，因为这些行的 **c** 列中的分片键值（"PQ" 和 "RS"）不符合新的 NULL 表达式。

如果启用自动更新分布统计信息，实现数据重分布 ALTER FRAGMENT . . . MODIFY 的语句会导致受影响的分片的分片级别统计信息将会删除。然而，表级别的统计信息不会被删除。因为该影响的分片没有分片级别的统计信息，下一个在此表中显式或自动 UPDATE STATISTICS 操作将重建分片级别分布，并将结果存储到系统目录中。

ALTER FRAGMENT 语句指定的修改都基于 **tab** 表分片，该 CREATE TABLE 语句定义了列表分布方案：

```
CREATE TABLE tab (i int, c char(2))
        FRAGMENT BY LIST (c)
        PARTITION p1 VALUES ("AB", "CD") IN dbs1,
        PARTITION p2 VALUES ("PQ", "RS") IN dbs2,
        PARTITION p3 VALUES (NULL) IN dbs3,
        PARTITION p4 REMAINDER IN dbs4;
```

下列语句修改了分片 **p1** 的分片表达式：

```
ALTER FRAGMENT ON TABLE tab MODIFY
          PARTITION p1 TO PARTITION p1
          VALUES ("AB", "CD", "EF") IN dbs1;
```

下列语句修改了分片 **p3** 的分片表达式：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p3 TO PARTITION p3
        VALUES ("XX", "YY", "ZZ") IN dbs3;
```

出于各种原因，ALTER FRAGMENT ON TABLE MODIFY 操作的结果：没有行可以移动到新分片中，并返回了错误，如下所示：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p3 TO PARTITION p3
        VALUES ("XX", "YY", "ZZ") IN dbs3;
```

修改后，**tab** 表生成的存储分布方案会有以下分片：

```
    PARTITION p1 VALUES ("AB", "CD")          IN dbs1,
    PARTITION p2 VALUES ("PQ", "RS")          IN dbs2,
    PARTITION p3 VALUES ("XX", "YY", "ZZ") IN dbs2
```

如果之前的余项分片 p3 在列 c 中有一值为 "AA" 的行，那么那一行不适合新分片策略中的任何分片。当尝试从余项分片移动行时，以上的 ALTER FRAGMENT 语句会由于错误而失败。

以下三个示例说明了同一表分片策略的更改会因为重叠而失败。

```
ALTER FRAGMENT ON TABLE tab MODIFY
```

```
PARTITION p2 TO PARTITION p2 VALUES (NULL) IN dbs2;
```

因为以上的 ALTER FRAGMENT 语句尝试将分片 **p2** 更改为一个重复的 NULL 分片，该语句由于产生错误而失败，因为 NULL 分片 **p3** 已经存在。

以下是对同一表的修改，它尝试将分片 **p2** 更改为一个重复的余项分片：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p2 TO PARTITION p2 REMAINDER IN dbs2;
```

以上语句由于产生错误而失败，因为现有的分片 **p4** 已经定义为余项分片。

以下修改在两个分片中创建了一个重复的表达式列表值 ″RS″：

```
ALTER FRAGMENT ON TABLE tab MODIFY
        PARTITION p1 TO PARTITION p1
        VALUES ("AB", "CD", "RS") IN dbs1;
```

由于在 **p2** 分片的表达式列表中已经定义了列表值 "RS" ，上述语句由于此错误而失败。

有关使用 ALTER FRAGMENT ON INDEX 语句的 MODIFY 选项的示例，请参阅 ALTER FRAGMENT ON INDEX 语句的示例。

## 范围分区表分区拆分（oracle 模式）

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

使用分区拆分语句根据语句中的拆分值将一个范围分区拆分为两个范围分区。

拆分出的新分区名不能使用表中已经存在的分区名。

参考语法:

alter fragment on table 表名 split partition 分区名 at (拆分值) into (partition 拆分后分区名 1, partition 拆分后分区名 2);

例如，分区拆分示例如下:

先创建范围分区表

```
create table tab12
        (cust_id integer,name char(128))
         partition by range(cust_id)
        (
         partition p0 values less than (100),
         partition p1 values less than (200)
        );
```

--根据示例语法拆分分区 p1

```
alter fragment on table tab12 split partition p1 at (150) into (partition p2, partition p3);
```

## 范围分区表分区合并（oracle 模式）

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

使用分区合并语句将范围分区表的多个分区合并为一个分区。

参考语法:

alter fragment on table 表名 merge partitions 分区 1, … 分区 n into partition 新的分区名;

**用法**

- 将多个分区合并成一个分区。
- 分区合并可以将需要合并的分区合并到一个新的分区中。
- 新的分区名可以是表中不存在的分区名或者被合并的分区名。
- 新的分区名不可以是表中不在被合并的分区范围的分区名。

例如,分区合并示例如下:

先创建范围分区表

```
create table tab13
    (cust_id integer,name char(128))
    partition by range(cust_id)
    (
    partition p0 values less than (100),
    partition p1 values less than (200)
   );
```

根据示例语法合并分区 p0,p1

```
alter fragment on table tab13 merge partitions p0,p1 into partition p8;
```

## 范围分区表分区重命名(oracle 模式)

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

使用分区重命名语句将指定分区重命名。

参考语法:

alter fragment on table 表名 rename partition 原分区名 to 新分区名;

例如,分区重命名示例如下:

先创建范围分区表

```
create table tab14
    (cust_id integer,name char(128))
    partition by range(cust_id)
```

```
(
partition p0 values less than (100),
partition p1 values less than (200)
);
```

根据示例语法重命名分区

```
alter fragment on table tab14 rename partition p1 to p7;
```

## ALTER FRAGMENT ON INDEX 语句的示例

以下一系列的示例阐述了 ALTER FRAGMENT ON INDEX 的 INIT 、ADD 、DROP 和 MODIFY 选项。

第一个示例创建了一个存储于 dbsp1 中的索引：

```
CREATE INDEX item_idx ON items (stock_num) IN dbsp1;
```

以下语句修改此索引以添加分片。值超过 50 的存储在 dbsp1、值在 51 和 80 之间的存储于 dbsp2 ，剩余的存储于 dbsp3：

```
ALTER FRAGMENT ON INDEX item_idx INIT
        FRAGMENT BY EXPRESSION
        stock_num <= 50 IN dbsp1,
        stock_num > 50 AND stock_num <= 80 IN dbsp2,
        REMAINDER IN dbsp3;
```

以下语句向该索引中添加了一个新的分片：

```
ALTER FRAGMENT ON INDEX item_idx
        ADD stock_num > 80 AND stock_num <= 120 IN dbsp4;
```

以下语句更改了此索引的第一个分片：

```
ALTER FRAGMENT ON INDEX item_idx
        MODIFY dbsp1 TO stock_num <= 40 IN dbsp1;
```

以下语句删除了该索引 dbsp4 中的分片：

```
ALTER FRAGMENT ON INDEX item_idx
        DROP dbsp4;
```

以下语句定义了一个按表达式分片的索引，其分片存储于 dbspaces dbsp1 和 dbsp2 的命名的分片中：

```
ALTER FRAGMENT ON INDEX item_idx INIT
        PARTITION BY EXPRESSION
        PARTITION part1 stock_num <= 10 IN dbsp1,
        PARTITION part2 stock_num > 20 AND stock_num <= 30 IN dbsp1,
        PARTITION part3 REMAINDER IN dbsp2;
```

以下语句添加了一个新的命名的分片：

```
ALTER FRAGMENT ON INDEX item_idx ADD
            PARTITION part4 stock_num > 30 AND stock_num <= 40 IN dbsp2
            BEFORE part3;
```

以下语句在索引 **idx1** 上定义了范围区间存储分布方案：

```
ALTER FRAGMENT ON INDEX idx2 INIT
        FRAGMENT BY RANGE(c2)
        INTERVAL (NUMTOYMINTERVAL(1,'MONTH')
        PARTITION part0 VALUES <   DATE('01/01/2007') IN dbs0,
        PARTITION part1 VALUES <   DATE('07/01/2007') IN dbs1,
        PARTITION part2 VALUES <   DATE('01/01/2008') IN dbs2
```

上述示例中，

- 分片键是列 **c2** 的值，
- 间隔值是一个月，
- 因为不包含 STORE IN 子句，新的系统生成的区间分区将轮流存储在 `dbs0` 、`dbs1` 和 `dbs2` 中；
- 区间分片过渡值是 01/01/2008 。（这是超过最后一个用户定义分片的范围的最小值）

以下语句在索引 **idx2** 上定义了一个列表存储分布方案：

```
ALTER FRAGMENT ON INDEX idx2 INIT
    FRAGMENT BY LIST(state)
    PARTITION part0 VALUES ('KS','IL') IN dbs0,
    PARTITION part1 VALUES ('CA','OR') IN dbs0,
    PARTITION part2 VALUES (NULL) IN dbs1,
    PARTITION part3 REMAINDER IN dbs2;
```

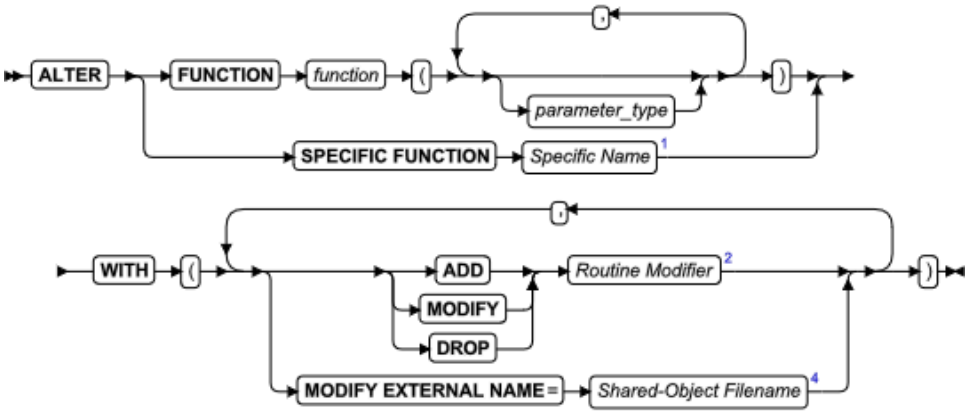
在以上列表分片示例中，

- 分片键是列 **state** 的值，
- 头两个分片的表达式列表是分别是两个 state 的邮政缩写字符串，
- 定义的行的 NULL 分片（**part2**）和余项分片（**part3**）的分片键值都不符合头两个表达式列表。

## 2.6 ALTER FUNCTION 语句

使用 ALTER FUNCTION 语句更改用户定义函数的例程修饰符或路径名。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 要修改的用户定义的函数 | 必须在数据库中注册。如果该名称没有唯一地标识函数，您必须为 *parameter_type* 输入一个或多个相应值 | 标识符 |
| *parameter_type* | 参数的数据类型 | 必须与 *function* 的定义中的数据类型相同（并以相同顺序指定） | 数据类型 |

**用法**

ALTER FUNCTION 语句可以修改用户定义的函数，以通过修改控制该函数如何运行的特征来调整其性能。您也可以添加或替换相关的用户定义的例程（UDR），这些例程为可以提供性能的查询优化器提供了另一选择。

所有修改在该函数下一次调用时生效。

只有 UDR 所有者或 DBA 可以使用 ALTER FUNCTION 语句。

## 引入修改的关键字

使用以下关键字引入您在 UDR 中所做的修改。

| 关键字 | 对指定例程修饰符的影响 |
|---|---|
| ADD | 向 UDR 添加一个新的例程修饰符 |
| MODIFY | 更改例程修饰符的属性 |
| DROP | 从 UDR 中删除该例程修饰符 |
| MODIFY EXTERNAL NAME （仅限于外部函数） | 替换可执行文件的文件规范。当 IFX_EXTEND_ROLE 配置参数为 ON 时，此选项只对被 DBSA 授予 EXTEND 角色的用户有效。当 IFX_EXTEND_ROLE 为 OFF （或未设置）时，UDR 所有者或 DBA 可以使用此选项 |
| WITH | 引入所有修改 |

如果例程修饰符为 BOOLEAN 值，则 MODIFY 将该值设置为 t （相当于使用关键字 ADD 添加该例程修饰符）。例如，下面这两个语句都更改了 func1 函数，这样它可以在可并行数据查询的环境中并行执行：

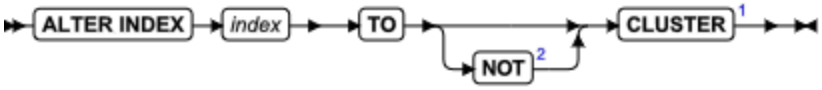
```
ALTER FUNCTION func1 WITH (MODIFY PARALLELIZABLE);
ALTER FUNCTION func1 WITH (ADD PARALLELIZABLE);
```

另见 更改例程修饰符示例 。

# 2.7 ALTER INDEX 语句

使用 ALTER INDEX 语句更改一个现有索引的集群属性。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 限制 | 限制 | 语法 |
|---|---|---|---|
| *index* | 要更改的索引的名称 | 必须存在 | 标识符 |

**用法**

ALTER INDEX 仅对 CREATE INDEX 语句显式创建的索引有效。ALTER INDEX 不可以修改临时表上的索引，也不可以修改数据库服务器默示地创建以支持约束的索引。

您不能更改现有索引的排列顺序。如果您在 SQL 的 SET COLLATIONY 语句已指定非缺省排列顺序之后，使用 ALTER INDEX 修改索引，则 SET COLLATION 语句对该索引没有任何影响。

ALTER INDEX 语句不能引用树型索引。有关树型索引的信息，请参阅 HASH ON 子句。

## TO CLUSTER 选项

TO CLUSTER 选项使得数据库服务器按索引键值的顺序将物理表的行重新排序。

集群一个索引会在同一 dbspace 的不同位置重建该表。当您运行带有 TO CLUSTER 关键字的 ALTER INDEX 语句时，与前一个版本的表相关的 extent 都会被释放。产生的新建版本的表没有空的 extent 。

对于升序索引， TO CLUSTER 以从最低到最高的顺序排列行。对于降序索引，这些行以从最高到最低的顺序重新排序。

当您重新排序时，整个文件会被重写。此进程可能需要很长的时间，并且它需要足够的磁盘空间以保留该表的两个副本。

当一个表正在集群时，它以 IN EXCLUSIVE MODE 方式锁定。当另一个进程正在使用该索引所属的表时，数据库服务器不能带有 TO CLUSTER 选项执行 ALTER INDEX 语句；除非锁定方式设置为 WAIT。（当锁定方式设置为 WAIT 时，数据库服务器会重试 ALTER INDEX 语句。）

在一段时间内，如果您修改该表，较早的集群的好处会消失，因为行是以可用空间的顺序添加，而不是按行本来的顺序添加。您可以通过对集群索引发出另一个 ALTER INDEX TO CLUSTER 语句，而重新集群该表以重新获得性能。在您对一个现有的集群索引发出另一个 ALTER INDEX TO CLUSTER 语句之前，您无需删除一个集群索引。

**集群一个索引的示例**

以下示例显示了如何使用 ALTER INDEX TO CLUSTER 语句以物理方式将 orders 表中的行排序。CREATE INDEX 语句在该表的 customer_num 列上创建了一个索引。然后 ALTER INDEX 语句使这些行以物理方式排序。

```
CREATE INDEX ix_cust ON orders (customer_num);
ALTER INDEX ix_cust TO CLUSTER;
```

### TO NOT CLUSTER 选项

TO NOT CLUSTER 选项删除指定的索引名称上的集群属性，而不影响表中行的物理顺序。

因为一个给定表仅可存在一个集群索引，在将集群属性分配给同一个表的另一个索引之前，您必须使用 TO NOT CLUSTER 选项从一个索引释放集群属性。

**删除集群属性的示例**

以下语句举例说明了如何从一个索引中除去集群，以及另一个索引如何以物理形式重新集群该表：

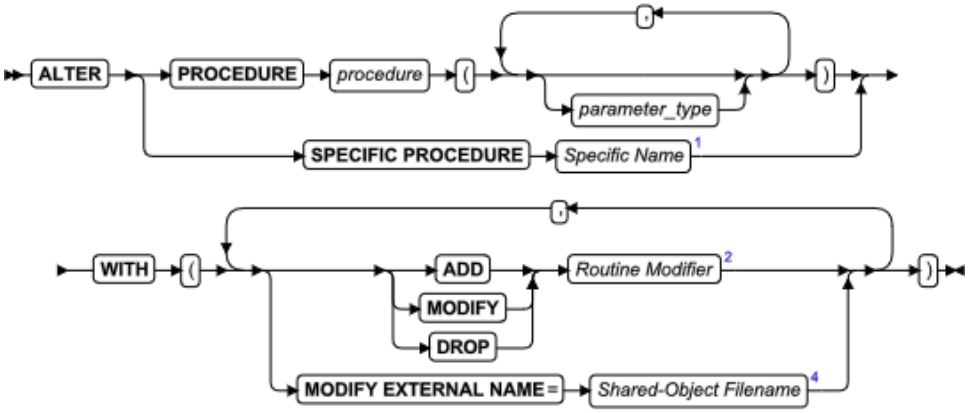
```
CREATE UNIQUE INDEX ix_ord ON orders (order_num);
CREATE CLUSTER INDEX ix_cust ON orders (customer_num);
            . . .
ALTER INDEX ix_cust TO NOT CLUSTER;
ALTER INDEX ix_ord TO CLUSTER;
```

前两个语句为 **orders** 表创建了索引，并以 **customer_num** 列的升序顺序集群了该物理表。后两个语句以 **order_num** 列的升序顺序集群了该物理表。

## 2.8 ALTER PROCEDURE 语句

使用 ALTER PROCEDURE 语句更改先前定义的外部过程的例程修饰符或路径名。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *procedure* | 要修改的用户定义的过程 | 必须在数据库中注册。如果该名称没有唯一地标识函数，您 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 必须为 *parameter_type* 输入一个或多个相应值 | |
| *parameter_type* | 参数的数据类型 | 必须与 *procedure* 定义中的数据类型相同（并以相同的顺序指定） | 数据类型 |

**用法**

ALTER PROCEDURE 语句使您可以修改外部过程以调整其性能，方法是修改控制外部过程执行的特征。您一个可以 添加或替换相关 UDR，它们为优化程序（可以提高性能）提供了其它选择。所有的修改在该过程下一次调用时生效。

只有 UDR 所有者或 DBA 可以使用 ALTER PROCEDURE 语句。

如果该过程名称在数据库中注册的例程中不是唯一的，则必须为 *parameter_type* 输入一个或多个相应值。

以下关键字介绍您在 *procedure* 中希望修改的内容：

| 关键字 | 作用 |
| --- | --- |
| ADD | 向 UDR 添加一个新的例程修饰符 |
| MODIFY | 更改例程修饰符的属性 |
| DROP | 从 UDR 中删除例程修饰符 |
| MODIFY EXTERNAL NAME (for external procedures only) | 替换可执行文件的文件规范。当 IFX_EXTEND_ROLE 配置参数为 ON 时，此选项只对被 DBSA 授予 EXTENT 角色的用户有效。当 IFX_EXTEND_ROLE 配置参数为 OFF 时，UDR 所有者或 DBA 可以使用此选项。 |
| MODIFY EXTERNAL NAME (for external procedures only) | 替换可执行文件的文件规范（仅对拥有 EXTEND 角色的用户有效） |
| WITH | 引入所有修改 |

如果例程修饰符为 BOOLEAN 值，MODIFY 将该值设置为 T （相当于使用关键字 ADD 添加该例程修饰符）。例如：下面这两个语句都更改了 proc1 过程，这样它可以在可并行数据查询的上下文中并行执行：

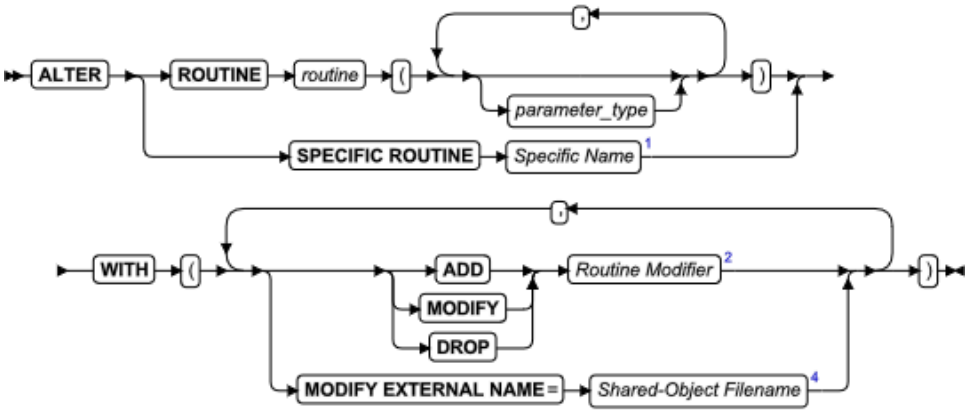
```
ALTER PROCEDURE proc1 WITH (MODIFY PARALLELIZABLE);
ALTER PROCEDURE proc1 WITH (ADD PARALLELIZABLE);
```

另见 更改例程修饰符示例 。

## 2.9 ALTER ROUTINE 语句

使用 ALTER ROUTINE 语句更改先前定义的用户定义的例程（UDR）的例程修饰符或路径名。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *routine* | 要修改的用户定义的例程 | 必须在数据库中注册。如果该名称没有唯一地标识例程，您必须为 *parameter_type* 输入一个或多个相应值 | 标识符 |
| *parameter_type* | 参数的数据类型 | 必须与 *routine* 定义中的数据类型相同（并以相同的顺序指定） | 数据类型 |

**用法**

ALTER ROUTINE 语句使您可以修改先前定义的 UDR 以调整其性能，方法是修改该 UDR 如何执行的特征。您也可以添加或替换相关的 UDR ，它们为优化程序（可以提高性能）提供了其它选择。

此语句在您不知道 UDR 是用户定义的函数还是用户定义的过程时很有用。当您使用词语时，数据库服务器会更改相应的用户定义的过程或用户定义的函数。

所有的修改在该 UDR 下一次调用时生效。

只有 UDR 所有者或 DBA 可以使用 ALTER ROUTINE 语句。

## 限制

如果该名称没有唯一地标识 UDR ，您必须为 ***parameter_type*** 输入一个或多个相应值。

当您使用词语时，UDR 的类型必须是确定的。您指定的 UDR 必须指定用户定义的函数或用户定义的过程。如果存在以下任何一种情况，则数据库服务器会返回一条错误：

- 您指定的名称（和参数）同时适用于用户定义的过程和用户定义的函数。
- 您指定的特定名称同时适用于用户定义的函数和用户定义的过程。

### 引用修改的关键字

使用这些关键字在 UDR 中引入您想要修改的项：

| 关键字 | 作用 |
| --- | --- |
| ADD | 向 UDR 添加一个例程修饰符 |
| DROP | 从 UDR 删除例程修饰符 |
| MODIFY | 更改例程修饰符的属性 |
| MODIFY EXTERNAL NAME (for external routines only) | 替换可执行文件的文件规范。当 IFX_EXTEND_ROLE 配置参数为 ON 时，此选项只对被 DBSA 授予 EXTENT 角色的用户有效。当 IFX_EXTEND_ROLE 配置参数为 OFF 时，UDR 所有者或 DBA 可以使用此选项。 |
| WITH | 引入所有修改 |

如果例程修饰符为 BOOLEAN 值，MODIFY 将该值设置为 T （相当于使用关键字 ADD 添加该例程修饰符）。

例如：下面这两个语句都更改了 func1 UDR ，这样它可以在可并行数据查询的环境中执行：

```
ALTER ROUTINE func1 WITH (MODIFY PARALLELIZABLE);
ALTER ROUTINE func1 WITH (ADD PARALLELIZABLE);
```

### 更改例程修饰符示例

假设您有一个外部函数 func1 ，它设置为处理 NULL 值，并且每次调用的成本设置为 40 。以下 ALTER ROUTINE 语句通过删除处理 NULL 值的能力而调整该函数的设置，通过将每次调用的成本更改为 20 而调整 func1 ，并指示该函数可以并行执行：

```
ALTER ROUTINE func1(CHAR, INT, BOOLEAN)
        WITH (
        DROP HANDLESNULLS,
        MODIFY PERCALL_COST = 20,
        ADD PARALLELIZABLE
        );
```

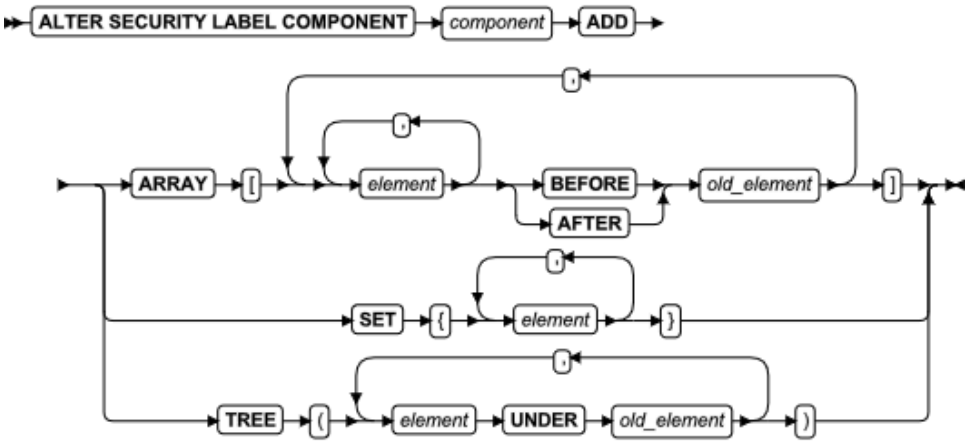
因为名称 func1 对数据库来说不是唯一的，所以指定了数据类型参数，这样例程特征符就是唯一的。如果此函数在创建时指定了一个特定名称（例如，raise_sal），您可以使用下面的第一行标识该函数：

```
ALTER SPECIFIC ROUTINE raise_sal;
```

## 2.10 ALTER SECURITY LABEL COMPONENT 语句

使用 ALTER SECURITY LABEL COMPONENT 语句向当前数据库中的一个现有的安全标签构件中添加一个或多个元件。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *component* | *element* 要添加的构件 | 必须在数据库中已存在 | 标识符 |
| *element* | *component* 的新的元素 | 在必须 *component* 的元素中必须唯一，且小于 32 字节。左括号（ ( )和右括号( ) )、逗号(,)、和冒号（: ）是无效的字符 | 引用字符串 |
| *old_element* | *component* 的现有的元素 | 必须是 *component* 的元素 | 引用字符串 |

**用法**

只有 DBSECADM 可以声明 ALTER SECURITY LABEL COMPONENT 语句，此语句定义现有安全标签构件的新元件。新的元件称为定义在引用指定构件的 CREATE SECURITY POLICY 语句中的安全策略的一部分。

一个安全标签构件由不多于 64 个 ***elements*** 的集合组成，它们由 CREATE SECURITY LABEL COMPONENT 语句定义为字符串常量。每个字符串常量不多于 32 字节，而且必须在该构件的元件中是唯一的。每个元件的声明（是构件具有的有效值）定义数据的敏感度的类别。通过向现有构件中添加新的元件，ALTER SECURITY LABEL COMPONENT 语句扩展了构件在包含该成分的安全策略内或在支持安全策略的安全标签内的可具有的值的设定。

当 ALTER SECURITY LABEL COMPONENT 语句运行成功后， GBase 8s 更新以下当前数据库中系统目录的表：

- **sysseclabelcomponentelements** 表，向该构件的新元件中添加新行，
- **sysseclabelcomponents** 表，显示构成该修改的安全构件的安全元件的新基数。

此语句可以定义一个安全标签构件的新元件，但是不能修改或伤处现有的元件。如果安全设计变更，以致于需要不同的元件，那么 DBSECADM 可以添加新的元件（如果元件的总数量保持在大小和基数的限制内，且不使用包含在该构件中的定义的表中的任何过时的元件）。

或者， DBSECADM 可以使用 DROP SECURITY LABEL COMPONENT 元件删除此构件，然后使用 CREATE SECURITY LABEL COMPONENT 元件重新定义新的构件，新的构件只包含必须的元件。然而，如果此构件是现有安全策略的一部分，则您不能删除此安全构件。有关删除安全标签构件和其它 GBase 8s 安全对象的限制，请参阅 DROP SECURITY 语句 。

要添加新元件的安全标签构件必须是三种构件类型之一。跟随在 *component* 名称之后的 ARRAY 、SET 或 TREE 关键字指定的构件类型必须与最初定义构件时 CREATE SECURITY LABEL COMPONENT 语句指定的构件类型相同。指定元件的新列表依赖于该指定的构件是否为 ARRAY 、SET 或 TREE 类型，这三种类型是 GBase 8s 支持的三种安全构件的类型。

## ADD ARRAY 子句

ARRAY 类型的安全标签组件是超过 64 个元素的有序集合。声明数组元素的顺序很重要，因为它定义了数据敏感性的降序，每个连续的元素在数据敏感性上都低于前面的元素。数组的标签元素集及其逗号（ , ）分隔符必须放在一对括号（ [ . . . ]）内。相同的新元素不能在同一 ADD ARRAY 子句中声明多次。

在 ADD ARRAY 子句中，BEFORE 或 AFTER 关键字必须跟随在新元件（或以逗号分隔的新元件的列表）之后按数据敏感度降序指定新元件的位置。在对元素大小和数量的限制中，此语法使 DBSECADM 能够在数组中任何位置（包括最高位置和最低位置）或在连续的现有元素之间插入新元素。然而，如果 ADD ARRAY 子句的 BEFORE 或 AFTER 关键字指定了之前未定义的数组元素（无论是在创建数组组件时，还是在先前的 ALTER SECURITY LABEL COMPONENT 语句中），ALTER SECURITY LABEL COMPONENT 语句都会失败并显示错误。

如果在同一个 ARRAY 类型的构件中执行了多个 ALTER SECURITY LABEL COMPONENT 操作以添加新元件， 由于数组元件的编码方式，DBSECADM 可能无法到达 64 组元件的最大值。有关安全元件是如何编码的信息，请参阅 GBase 8s 安全指南 。

以下示例定义了一个 ARRAY 类型的安全标签构件 aquilae ，它是五个元件的顺序集， imperator 在数据敏感度中是最高的，最低的是 asinus 。随后的 ALTER SECURITY LABEL COMPONENT 语句添加了两个新元件：

- 称为 **legatus** 的新元件排列在 **imperator** 和 **tribunus** 之间
- 称为 **cunctator** 的新元件排列在 **asinus** 之后，作为数据敏感度的新的最低一级。

```
CREATE SECURITY LABEL COMPONENT aquilae
        ARRAY [ "imperator", "tribunus", "centurio", "miles", "asinus" ];

ALTER SECURITY LABEL COMPONENT aquilae
        ADD ARRAY [ "legatus" BEFORE "tribunus","cunctator" AFTER "asinus" ];
```

该 ALTER SECURITY LABEL COMPONENT ... ADD ARRAY 语句的成功运行修改了 aquilae 安全标签构件组，因此构件元件新的降序顺序为：imperator 、legatus 、tribunus 、centurio 、miles 、asinus 、cunctator。

## ADD SET 子句

SET 类型的安全标签构件是不多于 64 个元件的无序集合。SET 构件中的元件的声明的顺序是没有意义的。数组元件的集合以及它们的逗号分隔符必须用一对大括号（{ ... }）括起。在同一 ADD SET 子句中，同一新 ***element*** 只能声明一次。

以下示例定义了一个 SET 类型的安全标签组件 departments ，它是三个元件的无序集合，这三个元件为 Marketing 、HR 和 Finance 。随后被 ALTER SECURITY LABEL COMPONENT 语句以添加三个新元件 Training 、QA 和 Security 的方式修改：

```
CREATE SECURITY LABEL COMPONENT departments
        SET { 'Marketing', 'HR', 'Finance' };

ALTER SECURITY LABEL COMPONENT departments
        ADD SET { 'Training', 'QA', 'Security' };
```

不像 ADD ARRAY 或 ADD TREE 规范，因为 SET 类型的构件的元件没有隐式的数据敏感度的顺序，所以 ALTER SECURITY LABEL COMPONENT 的 ADD SET 操作在重定义的构件的新的和现有元件中的不创建“多于”或“少于”数据敏感度关系。

## ADD TREE 子句

TREE 类型的安全标签构件具有没有循环的简单图的逻辑拓扑。每个 TREE 构件都有单独的根节点和不多于 63 个的附加节点。ALTER SECURITY LABEL COMPONENT 语句添加到此等级的新元件必须在根节点之下插入。每个新节点的字符串常量必须跟随在 UNDER 关键字和之前声明的字符串常量之后。TREE 构件的元件集合。包括它们的 UNDER 关键字和逗号分隔符必须用一对括号（（ ... ））括起。

在 UNDER 关键字之后指定的标签元素称为同一 UNDER 关键字（称为该父元素的 ***child***）之前的标签元素的 ***parent***。新元件称为父元件的 ***child*** 。然而，如果 ADD TREE 子句为该构件指定了在数据库中未定义的父元件，则 ALTER SECURITY LABEL COMPONENT 产生错误并失败。UNDER 关键字不能跟随在一个已添加到同一 ADD TREE 子句中的构件元件之后。

指定为 TREE 组件的根节点的字符串常量具有 TREE 分层结构内所有节点的最高的数据敏感性。在 TREE 中连续的父节点和子节点的任何子集中，每个非根元素具有比其父元素或其父元素的任何祖先低的数据敏感性，但是具有比它的任何子元素或其子元素的后代高的数据敏感性。

当没有豁免的用户尝试访问由包括 TREE 组件的标签保护的数据时，如果此用户的安全标签不包含符合数据行标签的同一组件的 TREE 组件中的一个元素，或者不包含符合这些元素其中之一的祖先的元素，则读取操作失败。除非该标签的安全策略包含 OVERRIDE 子句，否则在相同情景下写入操作也会失败。如果此数据行标签有多个 TREE 构件，则用户安全标签必须包含与该成分的 TREE 构件的元件值匹配（或祖）元件值。

在以下示例中，ALTER SECURITY LABEL COMPONENT 元件修改了一个 TREE 构件 Oakland ，修改方式为添加两个新节点到用它 CREATE SECURITY LABEL COMPONENT 语句定义的六个节点的树结构中：

```
CREATE SECURITY LABEL COMPONENT Oakland
        TREE ( 'Port' ROOT,
        'Downtown' UNDER 'Port',
        'Airport' UNDER 'Port',
        'Estuary' UNDER 'Airport',
        'Avenues' UNDER 'Downtown',
        'Hills' UNDER 'Avenues');
ALTER SECURITY LABEL COMPONENT Oakland
        ADD TREE ( 'Uptown' UNDER 'Port',
        'Bay' UNDER 'Estuary');
```

这里新的 Uptown 节点是 Port 的子节点，Port 有最高的数据敏感度，因为它是根节点。新的 Bay 节点是 Estuary 的子节点，Estuary 是 Airport 的子节点，Airport 是 Port 的子节点，由此表明 Bay 在此等级的三个节点中具有较低的数据敏感度。实际上，它不像任何被标记为 Port 数据，而是归类到较低的级别。Port 值可用作标签分配给允许存取所有关于此 Port 数据的用户。

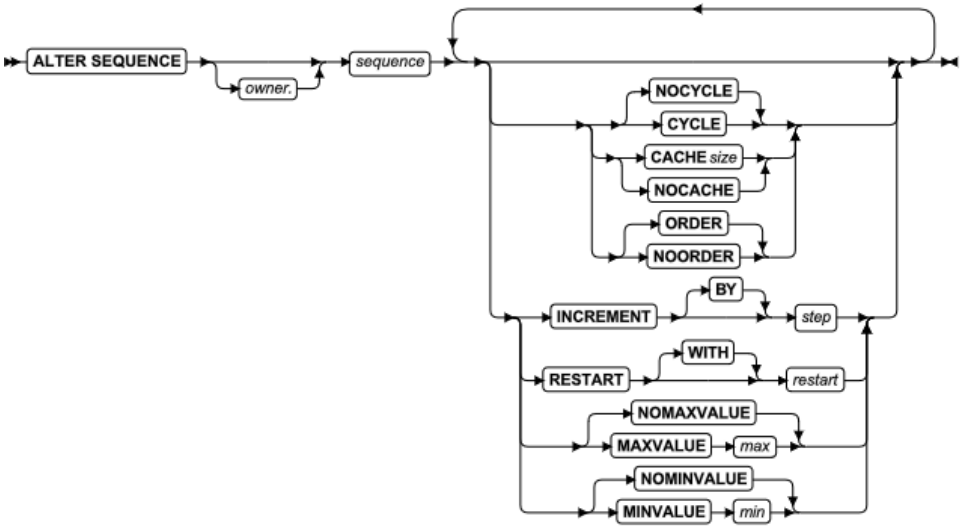
如果此示例中的 ALTER SECURITY LABEL COMPONENT 语句成功，且随后定义的数据行标签指定 **Bay** 作为 **Oakland** 构件的值，对安全策略不具有豁免权的用户在尝试在一个查询中读取受保护的表时，它需要 **Port** 、**Airport** 、**Estuary** 或 **Bay** 其中之一作为安全, 没有此安全策略的豁免的用户尝试读取查询中受保护的行时则需要将 **Port** 、**Airport** 、**Estuary** 或 **Bay** 作为用户标签值以满足此数据行标签的组件。因为它们不符合 **Bay** 且不是 **Bay** 的祖，所以此用户标签中的该构件的 **Uptown** 或 **Downtown** 值不满足。对于读取受保护的行的查询，该用户的安全标签包含满足此行安全标签的任何其它构件的值，且该用户也持有对此表的 Select 存取权限，以及至少对包含该保护的行的数据库的 Connect 存取权限。

ADD TREE 子句无法在现有子节点和其父节点之间插入一个新的节点。

## 2.11 ALTER SEQUENCE 语句

使用 ALTER SEQUENCE 语 句修改序列对象的定义。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *max* | 值的新上限 | 必须为 > CURRVAL 和 *restart* 的整数 | 精确数值 |
| *min* | 值的新下限 | 必须为 < CURRVAL 和 *restart* 的整数 | 精确数值 |
| *owner* | *sequence* 的所有者 | 不能被此语句更改 | 所有者名称 |
| *restart* | 序列中新的第一个值 | 必须是在 INT8 范围内的整数 | 精确数值 |
| *sequence* | 现有序列的名称 | 必须存在。不能是同义词 | 标识符 |
| *size* | 内存中要预分配的值的新数目 | 大于 > 2 但 < 一个循环中的基数（= \|(*max* - *min*)/*step*\|） | 精确数值 |
| *step* | 连续值之间的新的间隔 | 必须是非零整数 | 精确数值 |

**用法**

ALTER SEQUENCE 语句可以修改 **syssequences** 系统目录表中指定的序列对象的定义。

ALTER SEQUENCE 重新定义了现有的序列对象。它只影响随后生成的值（和序列高速缓存中任何未使用的值）。您不能使用 ALTER SEQUENCE 语句重命名序列，也不能更改序列的所有者。

要修改一个序列的定义，您必须是其所有者或 DBA 或具有对该序列的 Alter 权限。只会修改您在 ALTER SEQUENCE 语句中显式指定的序列定义的元素。如果您做出矛盾的更改（例如同时指定 MAXVALUE 和 NOMAXVALUE，或同时指定 CYCLE 和 NOCYCLE 选项），则会发生错误。

**示例**

以下示例基于随后的序列对象和表：

```
CREATE SEQUENCE seq_2
    INCREMENT BY 1 START WITH 1
    MAXVALUE 30 MINVALUE 0
    NOCYCLE CACHE 10 ORDER;

CREATE TABLE tab1 (col1 int, col2 int);
INSERT INTO tab1 VALUES (0, 0);

INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)


SELECT * FROM tab1;

        col1          col2
         0             0
         1             1


ALTER SEQUENCE seq_2
    RESTART WITH 5
    INCREMENT by 2
    MAXVALUE 300;

INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)
INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)
SELECT * FROM tab1;

        col1          col2
          0             0
          1             1
          5             5
          7             7
```

## INCREMENT BY 选项

使用 INCREMENT BY 选项指定一个序列中连续数字之间的新间隔。间隔（或 ***step*** 值）可以 INT8 范围内的一个正整数（对递增顺序）或负整数（对递减顺序） 。BY 关键字是可选的。

## RESTART WITH 选项

使用 RESTART WITH 选项指定该序列的新的第一个数字。如果使用 ALTER SEQUENCE 语句指定了 ***min*** 或 ***max*** 值，则 ***restart*** 值必须是 INT8 范围内的整数，它大于等于 ***min*** 值（对于递增顺序），或小于等于 ***max*** 值（对于递减顺序）。WITH 关键字是可选的。

当您使用 RESTART 选项修改序列时，***restart*** 值存储在 **syssequences** 系统目录表中，直到 **NEXTVAL** 操作中第一次使用该序列对象。在此以后，该值就在系统目录中复位。使用 **dbschema** 实用程序可以递增数据库中的序列对象，从而在生成的数字中产生间隔，您可能不希望这些数字在需要序列化整数的应用程序中出现。

### MAXVALUE 或 NOMAXVALUE 选项

使用 MAXVALUE 选项指定序列中值的新的上限。最大值（或 ***max*** ）必须是 INT8 范围内的整数，它大于 ***sequence*.CURRVAL** 和 ***restart*** （或者，如果未指定 ***restart*** ，则大于原 CREATE SEQUENCE 语句中的 ***origin*** ）。

使用 NOMAXVALUE 选项，以用新的缺省最大值（对递增顺序为 2e64；对递减顺序为 -1）替换当前的限制值。

### MINVALUE 或 NOMINVALUE 选项

使用 MINVALUE 选项指定序列中值的新下限。最小值（或 ***min***）必须是 INT8 范围内的整数，它小于 ***sequence*.CURRVAL** 和 ***restart***（或者，如果未指定 ***restart*** ，则小于原 CREATE SEQUENCE 语句中的 ***origin*** ）。

使用 NOMINVALUE 选项以用缺省值（对于递增顺序为 1；对于递减顺序为 -(2e64)）替换当前下限。

### CYCLE 或 NOCYCLE 选项

使用 CYCLE 选项替换 NOCYCLE 属性，在序列到达最大（升序）或最小（降序）限制值后继续生产序列值。在递增顺序到达 ***max*** 后，它为下一个值生产 ***min*** 值。在递减顺序到达 ***min*** 后，它为下一个值生产 ***max*** 值。

使用 NOCYCLE 选项可防止序列在到达声明的限制值之后生成更多的值。一旦序列到达该限制值，对 ***sequence*.NEXTVAL** 的下一次引用将返回一条错误消息。

### CACHE 或 NOCACHE 选项

使用 CACHE 选项指定在内存中预分配以快速访问的序列值的新数目。高速缓存大小必须是在 INT 范围内的整数，它小于循环中的个数（或小于 (|***max* - *min*)/*step***|）。最小大小为 2 个预分配的值。

使用 NOCACHE 在内存中不预分配值。（另见 CREATE SEQUENCE 语句 中有关 SEQ_CACHE_SIZE 的描述。)

### ORDER 或 NOORDER 选项

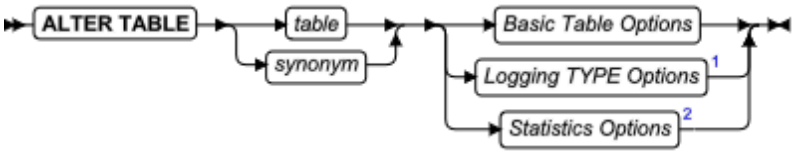
这些关键字对序列的行为没有影响。序列始终以用户请求的顺序向用户发出值，就像 ORDER 关键字是始终指定的一样。但是，ALTER SEQUENCE 语句接受 ORDER 和 NOORDER 关键字，以同在 SQL 的其它方言中序列对象的实现相兼容。

## 2.12 ALTER TABLE 语句

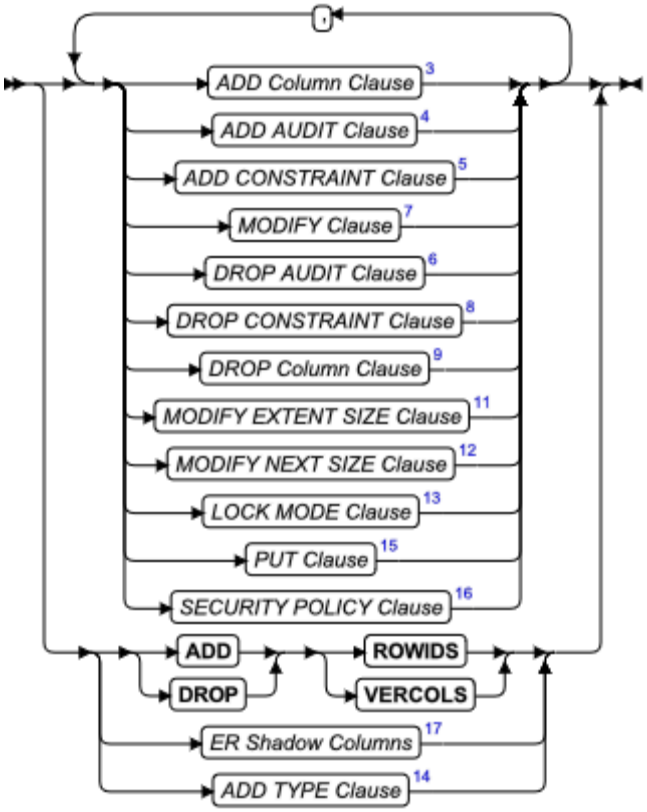
使用 ALTER TABLE 语句修改现有表的结构。

（要修改表的存储分布策略，您必须使用 ALTER FRAGMENT 语句而不是 ALTER TABLE 语句。）

**语法**




基本表选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *synonym* | 要更改的表的同义词 | 同义词及其表必须存在；USETABLENAME 必须未设置 | 标识符 |
| *table* | 要更改的表的名称 | 在当前数据库中必须存在 | 标识符 |

**语法**

GBase 8s 数据库服务器按您在 ALTER TABLE 语句中指定的顺序执行操作。如果任一操作失败，那么整个操作将被取消。

ALTER TABLE 语句不能向一个未分片表中添加分片存储策略，也不能修改分片表的存储分片策略。有关添加、修改或删除表的分片存储策略的信息，请参阅 ALTER FRAGMENT 语句 。

更改视图依赖的表可能会使得视图无效。

**警告：** 此语句可用的子句对性能的影响各不相同。采取更改操作之前，请检查 *GBase 8s 性能指南* 中 更改表定义 中相应章节来检查影响和策略。

您可以使用 Basic Table Options 段修改表的结构，方法是添加、修改或删除列和约束，或更改 extent 大小或锁定表的粒度。数据库服务器按您指定的顺序执行更改。如果任一动作失败，那么整个操作将会取消。

在 GBase 8s ，您可以将表与已命名的 ROW 类型关联，或指定新的存储空间以存储大对象数据。您也可以添加或删除 **rowid** 列或隐藏列以支持辅助服务器更新 USELASTCOMMITTED 功能的操作。然而，您不能与其他任何更改一起指定这些选项。

## Logging TYPE 选项

使用 Logging TYPE 选项指定该表具有特殊特性，以提高对它的各种批量操作。

Logging TYPE 选项


这里 STANDARD ，CREATE TABLE 语句的缺省选项 ，指定日志记录表，RAW 指定非日志记录表。

表可具有以下日志记录特性。

| 选项 | 作用 |
| --- | --- |
| STANDARD | 日志记录表允许回滚、还原和从归档恢复。这是缺省值。OLTP 数据库使用这种类型的还原和限制功能。 |
| RAW | 非日志记录表不支持主键约束、唯一约束或参考约束。RAW 表可拥有 NOT NULL 约束和 NULL 约束（但不能将这两个约束设置在同一列）。它们可以被索引和更新。使用这种类型以快速加载数据。 |

**Warning：** 使用 RAW 表以快速加载数据。建议您在事务中使用此表前或更改表中的数据前，将日志记录类型更改为 STANDARD 并执行零级备份。如果您必须在事务中使用 RAW 表，则将其设置为 Repeatable Read 隔离级别或者以互斥方式锁定该表以防止并发问题。

该 Logging TYPE 选项可以将非日志记录表（例如： RAW 表）转换为 STANDARD 表，以支持事务日志记录。如果使用此功能，您必须注意数据库服务器没有检查该表上是否在执行一个零级归档。

RAW 表上的操作不会被日志记录及恢复，因此 RAW 表一直处于风险中。当数据库服务器将一个非日志记录的表转换为一个 STANDARD 表类型时，您有责在事务使用此表前或更新表中数据时，执行零级备份。否则，执行失败可能引起在服务器崩溃的事件中的还原问题。

有关表的日志记录类型的更多信息，请参阅 *GBase 8s 管理员指南*。

该 Logging TYPE 选项具有以下限制：

- 在此日志记录表从其它日志记录类别变更为 STANDARD 之前，您必须执行零级归档。
- 该表不能是 TEMP 表，并且您不能将表的任一这些类型的变更为一个 TEMP 表。

以下示例将一个非日志记录表更改为一个使用事务日志记录的表：

```
ALTER TABLE tabnolog TYPE (STANDARD);
```

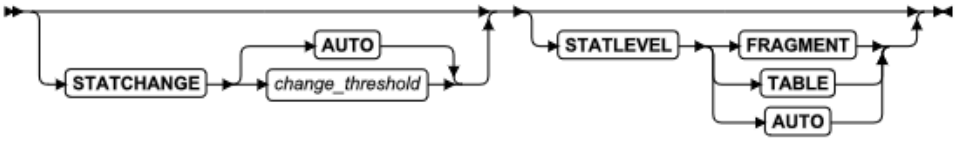
以下示例将一个日志记录表更改为非日志记录表：

```
ALTER TABLE tablog TYPE (RAW);
```

## ALTER TABLE 语句的 Statistics 选项

使用 ALTER TABLE 语句的 Statistics Options 子句更改分片表或未分片表的 STATCHANGE 属性的值，和分片表的 STATLEVEL 属性的值。这些表属性控制重新计算的阈值和数据分布统计信息的粒度。

**语法**




该子句支持与 CREATE TABLE 语句的 Statistics 选项相同的语法。

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *change_threshold* | 定义过时的分布统计信息的已更改数据的百分比 | 必须是在 0 - 100 内的整数 | 精确数值 |

**用法**

Statistics 选项子句可修改表的统计属性以允许用户控制 UPDATE STATISTICS 操作（当此 SQL 语句在一个分片表上以 LOW 、MEDIUM 或 HIGH 的方式运行时）。 ALTER TABLE 语句可以修改这些属性的指定的或缺省值（这些值在表创建时设置或者它们是由之前的 ALTER TABLE 语句设置的）。

Statistics 选项子句可将这两个表的属性设置为 STATCHANGE 和 STATLEVEL：

STATCHANGE 表指定需要考虑统计过时的更改的最小百分比（从该表中行的 UPDATE 、DELETE 和 INSERT 操作或从上次计算分布统计信息的分片）。您可以指定 0 - 100 内的整数作为更改的百分比，或使用 AUTO 关键字应用在 ONCOFIG 文件或会话环境中的当前的 STATCHANGE 配置参数作为缺省的更改的阈值。

UPDATE STATISTICS 语句的 AUTO 关键字也能启用比较用 STATCHANGE 已更改的设置值确定系统目录中的`统计信息是否过时的行的比例。包含 AUTO 关键字的 UPDATE STATISTICS 语句

只在当前的 UPDATE STATISTICS 操作期间启动旧的统计的检查（并只选择性地更新过时或丢失统计信息的表或分片）。

当 AUTO_STAT_MODE 配置参数或 AUTO_STAT_MODE 会话环境变量启用了自动方式，UPDATE STATISTICS 语句使用显式或缺省的 STATCHANGE 值辨别表、索引或统计信息丢失或过时的分片存储策略，并只更改丢失的或过时的统计信息。有关 UPDATE STATISTICS 操作的自动方式的信息，请参阅 *GBase 8s 管理员参考* 中有关 AUTO_STAT_MODE 的信息。

STATLEVEL 属性可决定数据分布粒度的级别和分片表的索引统计信息。它可以采用其中以下三个值之一（如果在创建的时候它没有值，则可使用 AUTO 作为缺省值）：

- TABLE 指定的该表的所有分布存储以表级别创建。
- FRAGMENT 指定的分布是创建和维护每个分片。
- AUTO 指定数据库服务器在运行时决定分片级别分布是否重要的标准。这些标准需要需要以下条件为真：
  - SYSSBSPACENAME 配置参数设置指定了一个现有的 sbspace
  - 该表按 EXPRESSION 、INTERVAL 、Rolling Window 、或 LIST 策略分片
  - 该表有超过 100 万行

  如果任何一种标准没有满足，那么数据库服务器创建表级别分布而不是分片级别分布。

会经常应用这些属性。如果该 STATLEVEL 设置为 AUTO ，则此设置会重写缺省值。

**注：** 当初始化数据库服务器时，必须设置 SYSSBSPACENAME 配置参数，指定数据库服务器存储分片级别数据分布统计信息的 sbspace 。它们作为存储在 **syfragsdist** 系统目录表的 **encdist** 列中 BLOB 对象。为了使数据库服务器支持分配级别统计信息，SYSSBSPACENAME 配置参数设置必须指定一个现有的 sbspace 。

如果您使用 Statistics 选项子句将 STATLEVEL 属性设置为 FRAGMENT ，且以下条件之一为真时，数据库服务器返回错误 -9814（"Invalid default sbspace name"）：

- 未设置 SYSSBSPACENAME 配置参数
- 按 gspaces -c -S 命令分配给 SYSSBSPACENAME 指定的 sbspace 不合适

**更改 STATLEVEL 的示例**

假设表 tabFrag 使用分片分布存储策略而不是 ROUND ROBIN ，且它包含一个名为 smartblob 的 BLOB 或 CLOB 列。决定保持该存储分布策略，但是使用 TABLE 而不是 FRAGMENT 作为 STATLEVEL 粒度。以下 SQL 语句引用了 tabFrag 表并拥有下列成功的作用：

- 将 STATLEVEL 更改为 TABLE ，通过使用 ALTER TABLE 的 Statistics 选项子句。
- 通过使用 UPDATE STATISTICS LOW 丢弃在 **sysfragdist** 系统目录表中 **tabFrag.smartblob** 的当前分片级别分布。
- 通过使用 UPDATE STATISTICS HIGH 为 **sysdistrib** 系统目标表中的 **tabFrag** 创建新的表级别统计信息。

```
ALTER TABLE tabFrag STATLEVEL TABLE;
```

```
UPDATE STATISTICS LOW
        FOR TABLE tabFrag (smartblob) DROP DISTRIBUTIONS

UPDATE STATISTICS HIGH
         FOR TABLE tabFrag (smartblob);
```

以上最后一条语句，缺省的 0.5 HIGH 决议意味着 tabFrag.smartblob 分布统计信息基于近似 200 bins 。

## 表的限制

跟随在 ALTER TABLE 关键字之后的表名或同义词的表必须是当前数据库中的常驻表。它具有以下限制：

- 不能是临时表。
- 不能是不是当前数据库中的表。
- 不能是 CREATE EXTERNAL TABLE 语句定义的表对象。
- 不能是违例表或诊断表。

此外，您不能用 ALTER TABLE 语句进行以下操作：

- 添加、删除或修改与违规表或诊断表关联的表中列。
- 在 RAW 表上定义引用约束或唯一约束。
- 在列或列的集合上定义 将列作为索引键的限制 冲突的索引。

如果已设置了 USETABLENAME 环境变量，那么您就不能在 ALTER TABLE 语句中指定表的 *synonym* 。

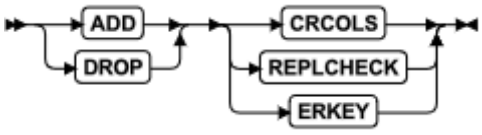
要使用 ALTER TABLE ，您的存取权限必须至少满足以下条件之一：

- 必须在包含该表的数据库上拥有 DBA 权限。
- 必须拥有该表。
- 必须拥有指定表上的 Alter 权限并在表驻留的数据库上拥有 Resource 权限。
- 要添加引用约束，必须在引用列或引用表上拥有 DBA 权限或 References 权限。
- 要删除约束，必须拥有 DBA 权限或是约束的所有者。如果是约束的所有者但不是表的所有者，必须在指定的表上拥有 Alter 权限。您不需要 References 权限就能删除约束。

## Enterprise Replication 阴影列

当您更改表时，可以添加或删除 Enterprise Replication 阴影列。

添加或删除 Enterprise Replication 阴影列



**语法**

如果当您正在使用 ADD CRCOLS 、ADD REPLCHECK 或 ADD ERKEY 关键字更改表时 Enterprise Replication 是活动的，则您必须用 cdr alter 命令将该表处于更改模式。

Enterprise Replication 使用 ADD CRCOLS 关键字创建阴影列(**cdrserver** 和 **cdrtime**)以解决冲突。如果该表的任一列的数据类型需要缓慢更改，那么更改此表以添加 CRCOLS 阴影列会是一种缓慢的更改操作。缓慢更改操作需要的磁盘空间至少是初始表加日志空间的两倍。有关应用 ALTER TABLE 语句性能的信息，请参阅更改表的定义。

使用 DROP CRCOLS 关键字删除 **cdrserver** 和 **cdrtime** 阴影列。您必须在删除 **cdrserver** 和 **cdrtime** 阴影列之前停止复制。

ADD REPLCHECK 关键字创建此阴影列（**ifx_replcheck**），您可以在此列创建一个索引与主键 ，以加速 Enterprise Replication 的一致性检查的过程。向表中添加 **ifx_replcheck** 阴影列是一个缓慢的更改操作，它所需要的磁盘空间至少是初始表加日志空间的两倍。

使用 DROP REPLCHECK 关键字删除 **ifx_replcheck** 阴影列。

Enterprise Replication 使用 ADD ERKEY 关键字创建阴影列 **ifx_erkey_1** 、**ifx_erkey_2** 和 **ifx_erkey_3**（代理主键）。 更改表以添加 ERKEY 阴影列是一个缓慢的更改操作。

使用 DROP ERKEY 关键字删除 **ifx_erkey_1** 、**ifx_erkey_2** 和 **ifx_erkey_3** 阴影列。

有关更多信息，请参阅使用 WITH CRCOLS 选项、使用 WITH REPLCHECK 关键字、使用 WITH ERKEY 关键字和 *GBase 8s Enterprise Replication 指南* 。

**示例**

以下示例中，往 **customer** 表中添加了 **cdrserver** 和 **cdrtime** 阴影列：

```
ALTER TABLE customer ADD CRCOLS;
```

以下示例中，往 **customer** 表中添加了阴影列 **ifx_replcheck**：

```
ALTER TABLE customer ADD REPLCHECK;
```

以下示例删除了 **customer** 表的 **ifx_replcheck** 列：

```
ALTER TABLE customer DROP REPLCHECK;
```

以下示例往 customer 表中添加了 i**fx_erkey_1** 、**ifx_erkey_2** 和 **ifx_erkey_3** 列：

```
ALTER TABLE customer ADD ERKEY;
```

## 使用 ADD ROWIDS 关键字

使用 ADD ROWIDS 关键字将名为 **rowid** 的新列添加至已分片的表。（缺省情况下，已分片的表不包含隐藏的 **rowid** 列。）当添加 **rowid** 列时，数据库服务器为每个在存续期间保持稳定的行分配一个唯一号码。数据库服务器创建一个它用来查找行的物理位置的索引。添加 **rowid** 列后，表的每一行包含了另外 4 个 字节来存储 **rowid** 值。

以下示例使用 ADD ROWIDS 选项向分片表 frag1 中添加了一个 INTEGER 类型的新 rowid 列：

```
ALTER TABLE frag1 ADD ROWIDS;
```

**提示：** 仅对分片表使用 ADD ROWIDS 子句。在非分片表中，rowid 列保持不变。建议将主键用作存取方法而不是利用 rowid 列。

有关 **rowid** 列的其它信息，请参阅 *GBase 8s 管理员参考手册* 。

## 使用 DROP ROWIDS 关键字

DROP ROWIDS 关键字可删除您添加（用 ALTER TABLE 或 ALTER FRAGMENT 语句）到分片表上的 **rowid** 列。

以下示例删除了 frag1 表的 rowid 列：

```
ALTER TABLE frag1 DROP RWIDS;
```

您无法删除未分片表的 rowid 列。

## 使用 ADD VERCOLS 关键字

ADD VERCOLS 关键字创建用于支持复制服务器更新的阴影列 **ifx_insert_checksum** 和 **ifx_row_version**。

在以下示例中，向 customer 表中添加了 ifx_insert_checksum 和 ifx_row_version 阴影列：

```
ALTER TABLE customer ADD VERCOLS;
```

修改表以添加新的版本控制列是一种快速更改操作。

有关更多信息，请参阅使用 WITH VERCOLS 选项和 *GBase 8s 管理员指南* 。

## 使用 DROP VERCOLS 关键字

使用 DROP VERCOLS 关键字删除 **ifx_insert_checksum** 和 **ifx_row_version** 阴影列。

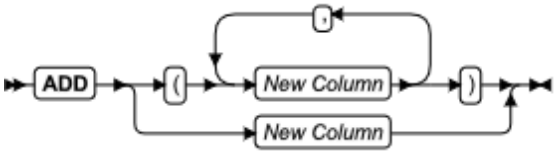
以下示例删除了 customer 表的这些阴影列：

```
ALTER TABLE customer DROP VERCOLS;
```

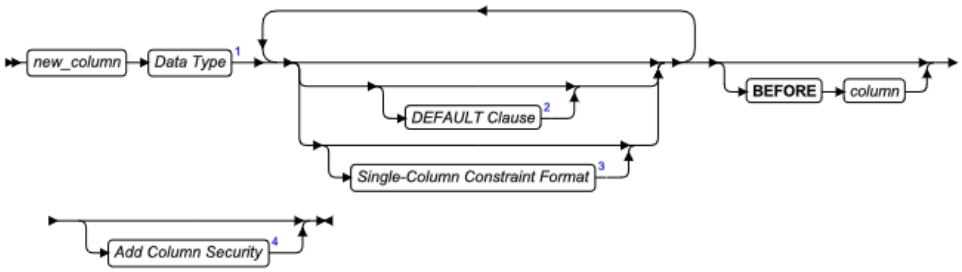
## ADD Column 子句

使用 ADD Column 子句向表添加列或安全策略。

ADD Column 子句




New Column




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 要在前面放置 *new_column* 的列的名称 | 必须已经在表中存在 | 标识符 |
| *new_column* | 正在添加的列的名称 | 如果表包含数据，则无法添加连续列 | 标识符 |

以下限制应用于 ADD 子句：

- 不能向包含数据的表添加顺序列。
- 不能添加超过最大行大小 40 M 字节的列。

以下限制影响了 ADD Column 子句添加 IDSSECURITYLABEL 数据类型的列以支持基于标签的存取控制的使用：

- 如果该表没有安全策略，持有 DBSECADM 角色的用户必须也包含 ADD SECURITY POLICY 语句指定一个现有的安全策略。
- 只有持有 DBSECADM 角色的用户才能添加 IDSSECURITYLABEL 类型的列。
- 一个表最多可拥有一个 IDSSECURITYLABEL 类型的列。
- IDSSECURITYLABEL 列不能持有列保护。
- IDSSECURITYLABEL 列缺省情况下具有隐式 NOT NULL 约束。如果 DEFAULT 子句中没有指定缺省的安全标签 ***label*** 的名称，那么该列的缺省值是该用户持有的写入存取的安全标签。
- IDSSECURITY LABEL 列不能拥有任何显式单独列约束，而且它不能是多列引用或检查约束的一部分。

## 字符列中支持的逻辑字符

对于您声明作为内置字符数据类型的新列，显式或缺省大小规格以字节为单位阐述，除非 SQL_LOGICAL_CHAR 配置参数已经为当前数据库启用逻辑字符语义。设计此功能的目的是降低在本地语言环境中调整数据字符串的风险，并支持多字符代码设置，例如：UTF-8。启用此功能会导致 SQL 语法分析器以逻辑字符而不是字节作为单位来解释声明的列大小，此功能声明的存储大小由一个正整数值，基于 SQL_LOGICAL_CHAR 设置分配新的字符列的倍数。

- 如果此设置的值为 OFF 或 1 ，那么 SQL_LOGICAL_CHAR 配置参数没有影响。
- 如果该值设置为 ON ，而不是一个数字，该扩充因素是存储数据库代码集中最大逻辑字符所需的字节数。(设置 ON 相当于 4 ，是最大的有效数。）

该扩充因素的值是数据库的属性，其基于创建数据库时的 QL_LOGICAL_CHAR 设置而不是由 ALTER TABLE 语句声明（如果这两个设置不同）。

当启用此功能时，声明其为 VARCHAR 或 NCHAR 数据类型，只有最大大小规格由此功能扩展。保留的大小是由数据类型声明中的显示或缺省的 ***reserved*** 值指定的字节数，因为逻辑字符的最小大小为 1 字节。

用户定义的字符列类型的大小规范（UDTs）通常解释为字节并不会被此功能影响。存储字符串作为大对象的列（例如：CLOB 和 TEXT），同样不受影响。

有关 SQL_LOGICAL_CHAR 配置参数的更多信息，请参阅 GBase 8s 管理员参考手册。有关多字节语言环境和逻辑符号的其它信息，请参阅 GBase 8s GLS 用*户指南* 。

## BEFORE 子句

该可选的 BEFORE 子句决定在表结构中新的列的最初位置，通过在 ALTER TABLE ADD 语句插入新列之前 ，指定现有列的名称。

在以下示例中，BEFORE 选项引导数据库服务器在 **total_price** 列之前添加 **item_weight** 列：

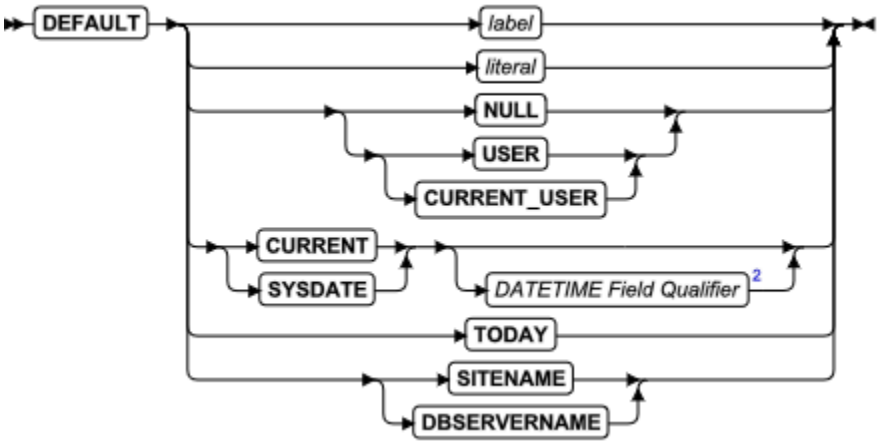
```
ALTER TABLE items
        ADD (item_weight DECIMAL(6,2) NOT NULL BEFORE total_price);
```

如果没有包含 BEFORE 子句，则缺省情况下数据库服务器将新的列添加到当前表结构的最后一列的末尾，以 ADD 子句中的词法顺序。

## DEFAULT 子句

使用 DEFAULT 子句指定当列的显式值没有指定时，数据库服务器应当在该列中插入的值。

DEFAULT 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 安全标签的名称 | 必须存在且必须属于保护此表的安全策略。该列必须是 IDSSECURITYLABEL 类型。 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *literal* | 列的文字缺省值 | 必须适合列的数据类型。请参阅将文字值作为缺省值 | 标识符 |

如果当您添加包含缺省值的列时，您正在更改的表已经有行，则此缺省值只应用在 ALTER TABLE MODIFY 语句之后插入的新列的行中。任何新列插入之前已存在的行，会在新列中拥有一个 NULL 值，除非您更改这些行以插入非 NULL 值。现在您插入的行将拥有 MODIFY 子句指定的缺省值，除非您向新列中插入了其它值。

你不能指定顺序列的缺省值。 对于 DISTINCT 或 OPAQUE 数据类型的列，您不能指定作为缺省值像变量函数的常数表达式（例如：CURRENT 、SYSDATE DBSERVERNAME 、SITENAME 、TODAY 、USER 或 CURRENT_USER）。

以下示例将数据类型为 DECIMAL(6,2) 的列添加到 items 表。在 items 中，新的列 item_weight 有文字缺省值：

```
ALTER TABLE items
        ADD item_weight DECIMAL (6, 2) DEFAULT 2.00
        BEFORE total_price;
```

在此示例中，**items** 表中的每个现有行的 **item_weight** 列都有缺省值 2.00 。

有关 DEFAULT 子句的选项的更多信息，请参阅 CREATE TABLE 语句中 DEFAULT 子句 章节。

### *DEFAULT 标签*

当 DBSECADM 向由安全策略保护的表添加 IDSSECURITYLABEL 列时，需要 DEFAULT ***label*** 规范，除非该表为空。如果表不为空，那么指定的 ***label*** 会插入到表的现有行中。

如果 DEFAULT 子句我数据类型不是 IDSSECURITYLABEL 的列指定安全标签，或者如果表没有安全策略，或者如果该标签的安全策略不是该表的安全策略，那么会发出错误。

要定义指定的 ***label*** 为 IDSSECURITYLABEL 列的缺省值，指定不带 ***policy*** 限定符的 ***label*** 名称，而不是 ***policy.label***，因为该表的安全策略只对表中保护数据的安全标签是有效策略。

以下示例中的语句向表 T1 中添加安全策略标签 MegaCorp，并通过声明新列 D 为 IDSSECURITYLABEL 类型（其缺省值是名为 mylabel 的安全标签）来为表指定列级别保护：

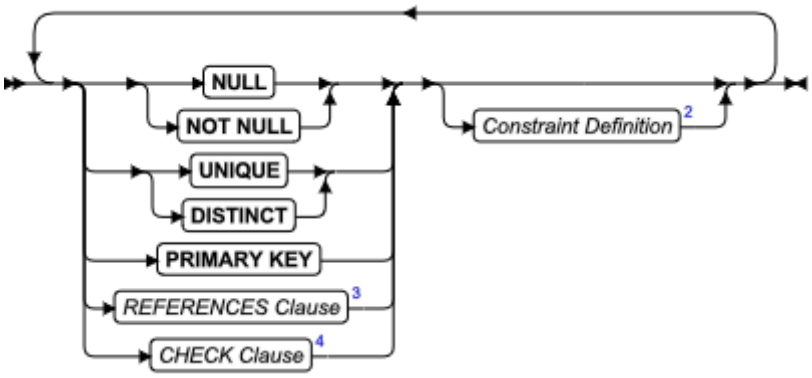
```
ALTER TABLE T1
        ADD (D IDSSECURITYLABEL DEFAULT mylabel1)
        ADD SECURITY POLICY MegaCorp;
```

因为不包含 BEFORE 子句，列 D 是表 T1 结构中的最后一列。如果它引用的数据库对象在当前数据库中不存在（除了新列 D），则该语句失败。

## 单列约束格式

使用单列约束格式将一个或多个约束与单列关联。

*单列约束格式*




如果表包含数据，则您不能在新列上指定主键或唯一约束。然而在唯一约束的情况下，表可以包含单行的数据。如果希望添加带有主键约束的列，则发出 ALTER TABLE 语句时表必须为空。

*当在现有列上放置主键或者唯一约束时，应用以下规则:*

- 当您在一列或一组列上放置主键或唯一约束时，数据库服务器在受约束的列或一组列上创建内部 B-tree 索引，并且自动计算列统计信息，相当于 UPDATE STATISTICS 语句在 HIGH 模式下创建分布。除非在该列或该组列上定义了用户创建的索引。
- 如果在一列或一组列上放置主键或唯一约束，而且该列或该组列上已存在唯一索引，则约束共享该索引。然而如果现有索引允许复制，则数据库服务器返回错误。那么您必须在添加约束之前删除现有索引。
- 如果在一列或一组列上放置主键或唯一约束，并且该列或该组列上已存在引用约束，则重复索引升级为 UNIQUE （如果可能的话），并且该索引被共享。

不能在 BYTE 或 TEXT 列上放置唯一约束，也不能在这些数据类型的列上放置引用约束。 BYTE 或 TEXT 列上的检查约束可只能用于检查 IS NULL 、IS NOT NULL 或 LENGTH。

如果在同一列上指定 NOT NULL 约束和 NULL 约束，则该语句失败。不能在数据类型是 LIST 、MULTISET 、SET 或 IDSSECURITYLABEL 的列上定义 NULL 约束。

IDSSECURITYLABEL 列具有隐式 NOT NULL 约束，但是它不能具有显式单列约束也不能是多列引用约束或检查约束的一部分。如果约束在存储加密数据的列上，那么 GBase 8s 不能强制执行此约束。

***重要：*** 不能使用单列约束格式在 ENABLED NOVALIDATE 或 FILTERING WITH ERROR NOVALIDATE 或 FILTERING WITHOUT ERROR NOVALIDATE 约束模式中添加外键约束的新列。对于 ALTER TABLE 语句在 ALTER TABLE 操作期间使用 NOVALIDATE 关键字绕过违规检查来创建新的外键约束，必须使用具有多列约束格式的 ALTER TABLE ADD CONSTRAINT 语法。

## *ADD COLUMN 使用 NOT NULL 约束*

当您添加一个 NOT NULL 约束的列时，如果表中含有数据，则您必须包含 DEFAULT 子句。

然而，如果表为空，您可以添加列并只应用 NOT NULL 约束。无论该表是否包含数据，以下示例语句都有效：

```
ALTER TABLE items
        ADD (item_weight DECIMAL(6,2)
        DEFAULT 2.0 NOT NULL
        BEFORE total_price);
```

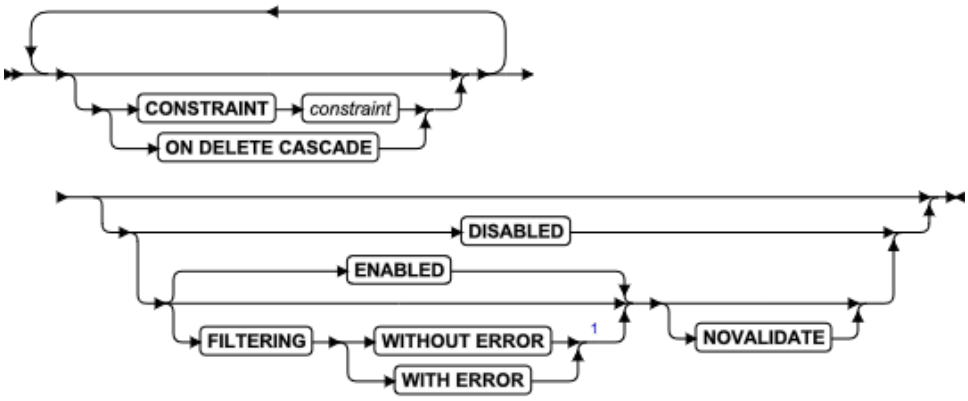
## *约束定义*

使用 ALTER TABLE 语句中的约束定义部分来声明约束的名称并将约束方式设置为 DISABLED 或 ENABLED ，或者对于包含隔离表的表，设置为两个 FILTERING 方式。

对于启用或过滤 ALTER TABLE ADD CONSTRAINT 语句可以创建的外键约束，NOVALIDATE 方式可以在数据库服务器在创建此约束时，防止其检查该表的每一行是否符合启用约束。

单列约束格式和多列约束格式都支持以下定义约束的语法：

约束定义




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *constraint* | 在此为约束声明的名称 | 在数据库中的索引和约束名称中必须唯一 | 标识符 |

### 用法

如果此 ALTER TABLE 语句包含单个列约束格式或多列约束格式，但是约束定义为空，则数据库服务器创建并启用指定任意类型的单列约束格式或多列约束格式，给此约束分配一个系统生成的标识符和一个缺省对象状态，并将这些属性注册到 **sysconstraints** 和 **sysobjstate** 系统目录表中。

如果您未为此约束指定方式，那么约束使用缺省的方式。

可选的 ON DELETE CASCADE 关键字可以在该约束名称之前或之后。对于引用约束，ON DELETE CASCADE 关键字指示数据库服务器在删除父表中具有关联的主键的行时，删除子表中具有外键的行。有关 DELETE 操作上这些关键字的影响的更多信息，请参阅使用 ON DELETE CASCADE 选项 。

当创建 ALTER TABLE ADD CONSTRAINT 语句定义的外键约束时，NOVALIDATE 关键字防止数据库服务器在运行 ALTER TABLE 语句时检查该表的每一行是否符合启用约束。有关此关键字的限制和影响的更多信息，请参阅在 NOVALIDATE 方式下创建外键约束。

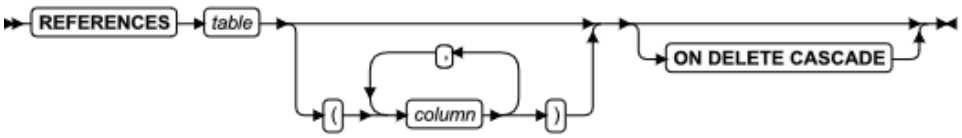
在 CREATE TABLE 语句中，您不能在 BYTE 或 TEXT 列上定义唯一约束、主键约束或引用约束。此外，该表不能是 RAW 表。

有关约束方式选项的更多信息，请参阅选择约束方式选项。

## *REFERENCES 子句*

REFERENCES 子句具有以下语法。

REFERENCES 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 被引用表中被引用的列 | 请参阅 对引用约束的限制. | 标识符 |
| *table* | 被引用表 | 被引用表和引用表必须驻留在同一数据库中 | 标识符 |

REFERENCES 子句允许您在一个或多个列上放置外键引用。被引用的列和引用列可以在同一表中，或在同一数据库的不同表中。

如果被引用表与引用表不同，则缺省的 ***column*** 是主键列。如果被引用表与引用表相同，则没有缺省主键列。

可选的 ON DELETE CASCADE 关键字可在 REFERENCES 子句中被指定为最后的关键字，或者它们可跟随在声明的约束定义中约束名称之后。有关在 DELETE 操作中的这些关键字的作用的信息，请参阅使用 ON DELETE CASCADE 选项 。

## *对引用约束的限制*

必须拥有 REFERENCES 权限才能创建引用约束。

以下限制应用到在 REFERENCES 子句中指定（被引用列）的 *column*：

- 被引用表和引用表必须在同一数据库中。
- 被引用表（或列组）必须拥有唯一约束或主键约束。
- 被引用表和引用表是相同的数据类型。

  唯一的例外在于，如果被引用列是顺序数据类型，则引用列必须为整数数据类型：

  - 对 BIGSERIAL 被引用列，使用 BIGINT 引用列。
  - 对 SERIAL 被引用列，使用 INT 引用列。
  - 对 SERIAL8 被引用列，使用 INT8 引用列。
- 不能在 BYTE 或 TEXT 列上放置引用约束。

- 不能在 RAW 表的任何列上放置约束。
- 约束使用创建时生效的排序规则。
- 列级 REFERENCES 子句只能包含单独一个列名称。
- 表级的 REFERENCES 子句中最大列数为 16 。
- 表级的 REFERENCES 子句列的总长度不能超过 390 字节。

### *引用子句的缺省列*

如果被引用表与引用表不同，则无须指定被引用列；缺省列是被引用表的主键列（或列组）。如果被引用表与引用表相同，则你必须指定被引用列。

以下示例创建在 cust_calls 表中创建了新列 ref_order 。ref_order 列是引用 orders 表中 order_num 列的外键。

```
ALTER TABLE cust_calls
        ADD ref_order INTEGER
        REFERENCES orders (order_num)
        BEFORE user_id;
```

如果您在一列或一组列上放置引用约束，并且重复索引和唯一索引已存在于该列或该组列上，则索引被共享。

### *使用 ON DELETE CASCADE 选项*

如果您希望当 DELETE 或 MERGE 语句删除父表中的行时，子表中相应的行也被删除，则使用 ON DELETE CASCADE 选项。

此处，***parent table*** 是定义启用外键约束的 REFERENCING 子句指定的表 ，***child table*** 是定义启用外键约束的表。如果您不指定级联删除，则在某个表被其它表用主键外键关系引用的情况下，数据库服务器的缺省行为防止 DELETE 和 MERGE 语句删除该表中的数据。

如果您指定此选项，则当删除父表中的行时，数据库服务器还删除任何与子表的该行（外键）关联的行。ON DELETE CASCADE 选项的运行您减少执行删除操作所需的 SQL 语句的数量。

例如：在 stores_demo 数据库中，stock 表将 stock_num 列包含为主键。catalog 表引用 stock_num 列作为外键。以下 ALTER TABLE 语句删除现有外键约束（没有级联删除），并添加指定级联删除的新约束：

```
ALTER TABLE catalog DROP CONSTRAINT aa;
ALTER TABLE catalog ADD CONSTRAINT
        (FOREIGN KEY (stock_num, manu_code) REFERENCES stock
        ON DELETE CASCADE CONSTRAINT ab);
```

如果在子表上指定了级联删除，则除了从 **stock** 表中删除 stock 项以外，还删除级联到与 **stock_num** 外键相关联的目录表。此级联删除仅当 **stock_num** 没有订购时有效；否则，**items** 表中的约束将不允许级联删除。有关更多信息，请参阅 表有级联删除时对 DELETE 的限制 。

如果表带有 DELETE 触发事件的触发器，则您不能在该表上定义级联删除引用约束。当您试图将指定 ON DELETE CASCADE 的引用约束添加到有删除触发器的表时，接收到一条错误。

TRUNCATE 语句无法从子表进行级联删除。TRUNCATE 语句的目标表不能被在另一个定义启用外键约束表引用（除非子表没有行）。

有关从具有级联删除的表中删除行时的语法限制和锁定音响的信息，请参阅级联删除表时的注意事项。

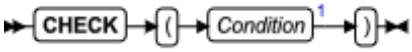
### *引用约束创建期间持有的锁定*

当您创建引用约束时，数据库服务器在被引用的表上放置互斥锁。锁定在您的 ALTER TABLE 语句结束之后，或者在事务末尾时是释放（如果正在更新数据库中使用事务日志记录的表）。

## *CHECK 子句*

检查约束指明了在数据可插入列 ***before*** 必须满足的条件。

CHECK 子句




插入或更新时，如果行对表上定义的任何检查约束返回 ***false*** ，则数据库服务器返回一条错误。然而，如果行对检查约束返回 NULL ，则不返回错误。在某些情况下，您可能希望同时使用检查约束和 NOT NULL 约束。

检查约束是使用***搜索条件***定义的。搜索条件不能包含用户定义的例程、子查询、聚集、主变量或行标识。另外，条件不能包含变量内置函数 CURRENT 、SYSDATE 、USER 、CURRENT_USER 、SITENAME 、DBSERVERNAME 或 TODAY 。

检查约束不能包含不同表中的列。当正在使用 ADD 或 MODIFY 子句时，检查约束不能依赖于相同表的其它列中的值。

下一示例添加一个新的 unit_price 列到 items 表，并包含一个检查约束来确保输入的值大于 0：

```
ALTER TABLE items
        ADD (unit_price MONEY (6,2) CHECK (unit_price > 0));
```

要创建检查多个列中值的约束，请使用 ADD CONSTRAINT 子句。以下示例在先前示例中添加的列上构建了一个约束。检查约束现在跨越了表中的两列。

```
ALTER TABLE items ADD CONSTRAINT CHECK (unit_price < total_price);
```

## *Add Column SECURED WITH Label 子句*

此 Add Column Security Label 子句使新列与一个安全标签相关联。

Add Column SECURED WITH Label 子句


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *label* | 安全标签的名称 | 必须存在且必须属于保护该表的同一个安全策略 | 标识符 |

**用法**

只有持有 DBSECADM 角色的用户才能在 ALTER TABLE 语句中使用此子句。

ALTER TABLE 语句不能向以下类别的表对象中添加安全标签：

- 虚拟表接口（VTI）表或虚拟索引接口（VII）表
- 用户定义或系统定义的临时表（TEMP）
- 已命名或未命名的 ROW 类型表
- 类型表层次结构中的父表或子表
- 没有基于标签的安全策略的表

必须指定不带 ***policy*** 限定符的 ***label*** 名称，而不是 ***policy.label*** ，因为该表当前的安全策略是表中仅有的保护数据的安全标签有效策略。

该列不能是 IDSSECURITYLABEL 类型。

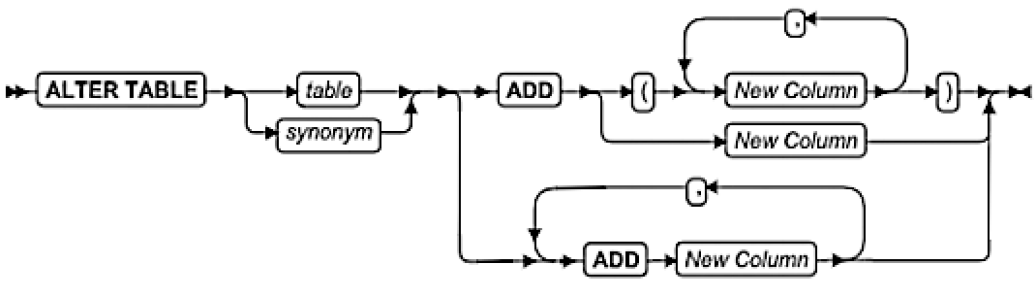
当持有合适基于标签存取权限的用户试图存取受保护列中的数据，数据库服务器会将此标签与该用户的安全凭据比较，从而在比较的基础上允许或拒绝存取。

有关添加和删除该表基于标签的安全策略的语法，请参阅 Add Column SECURED WITH Label 子句 。

## ORACLE 模式下添加多列语法

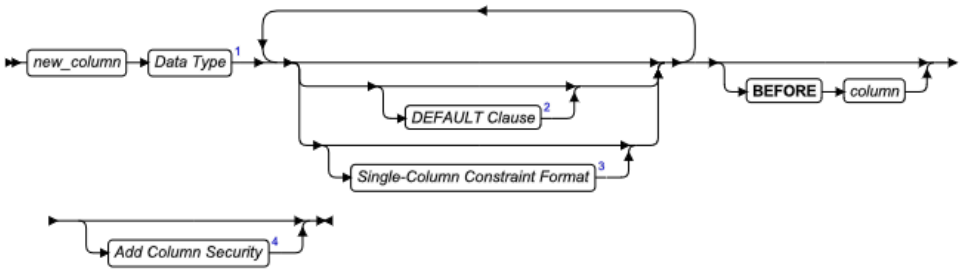
ALTER TABLE 语句的添加多列语法，支持 Oracle 添加多列的语法，即 ADD 新列, ADD 新列，该功能仅在 GBase 8s 的 ORACLE 模式下支持。

ADD Column 子句



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *new_column* | 所添加的列的名称 | 不能与表内已经存在的列重名 | 标识符 |
| *synonym* | 序列对象的同义词 | 必须指向序列 | 标识符 |
| *table* | 所需添加列的表的名称 | 必须已注册在当前数据库中 | 标识符 |

New Column




说明和限制

新添加的列不能和表中已有的列名重复。

添加多个列时，添加操作从左到右顺序执行。

- 添加多列语法支持使用原生 ADD() 语法，添加列使用逗号来分隔。例如，alter table tab1 add (c1 int,c2 int),add (c3 int,c4 int); 。

添加列大小不能超过行的最大大小 40M 字节。

## ADD AUDIT 子句

使用带有 ALTER TABLE 命令的 ADD AUDIT 子句使表包含选择性行级审计。

当您使用 ADD AUDIT 子句更改表，则当选择性行级别审计开始时，会记录表中行级审计事件。表自己应用 ADD AUDIT 属性并不会启用选择性行级审计。当 adtcfg 文件的 ADTROWS 参数设置为 1 或 2（通过使用 gaudit -R 命令）时，启用该类型的审计。

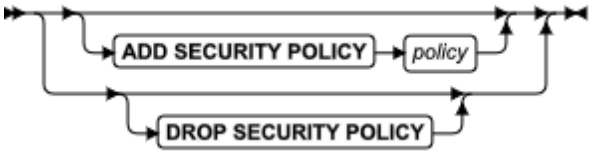
如果未启用选择性行级审计，则表中的 ADD AUDIT 属性没有影响。

必须拥有 RESOURCE 或 DBA 权限才能用 ADD AUDIT 子句运行 ALTER TABLE 命令。

## SECURITY POLICY 子句

该可选的 SECURITY POLICY 子句可用以下语法来删除当前与该表相关联的安全策略或给没有安全策略的表关联一个安全策略。

SECURITY POLICY 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *policy* | 安全策略的名称 | 必须是保护该表的安全策略 | 标识符 |

只有 DBSECADM 可以使用此语句向现有表添加安全策略，或移除当前保护该表的安全策略。

ALTER TABLE 语句不能向由 CREATE EXTERNAL TABLE 语句定义的表中添加安全策略。

以下准则适用于执行 ALTER TABLE 语句 ADD SECURITY POLICY 子句进行保护的表：

- 除非表具有与其相关联的安全策略或表具有任一被保护的行，否则该表未受到保护。前一种情况说明该表是拥有行级别粒度的受保护的表，后一种情况说明该表是拥有列级别粒度的受保护的表。
- 如果该表没有与它关联的安全策略，那么使用 ALTER TABLE ... ADD 语句向现有表中添加 IDSSECURITYLABEL 列保护行的操作失败。
- 如果该表没有与它关联的安全策略，则保护列的 ALTER TABLE ... MODIFY ... COLUMN SECURED WITH 子句失败。
- 一个表最多可以拥有一个安全策略。如果该表已有安全策略，则 ALTER TABLE ... ADD SECURITY POLICY 语句失败。
- 表可用于多个受保护的列。每个受保护的列库具有不同的安全标签或某些受保护的列共享同一安全标签。
- 不能使用此子句向临时表或在当前数据库之外的表中添加安全策略。
- 一个表只能拥有最多一个 IDSSECURITYLABEL 类型的列。
- IDSSECURITYLABEL 列不能拥有列保护。
- IDSSECURITY LABEL 列不能拥有单列约束或不能是多列引用约束或检查约束的一部分。
- IDSSECURITYLABEL 列不能被加密。
- IDSSECURITYLABEL 列有隐式 DEFAULT NOT NULL 约束。列缺省值是用户写入存取安全标签的值。
- DBSECADM 不能删除 IDSSECURITYLABEL 列。必须持有一般 CONNECT 、RESOURCE 和 ALTER 存取权限才能删除该列。
- ALTER TABLE 语句不能修改 IDSSECURITYLABEL 列。

- 如果以下任何条件为真，则向受保护分片中添加分片的操作失败：
    - 如果源表和目标表没有使用相同的安全策略进行保护；
    - 如果表不具有相同的保护粒度；
    - 如果表受保护的列的设置不同，由同一安全标签保护。

  有关使用 ALTER FRAGMENT 语句向受保护的表中连接分片的更多信息，请参阅 ATTACH 子句的其他限制 。
- 拆离受保护表的分片以创建一个新的表，该表的拥有相同的安全策略，相同的行安全标签列且受保护列的设置也相同。

如果 DROP SECURITY POLICY 子句执行成功，则它会产生以下影响：

- 除非表具有与其相关联的安全策略或表具有任一被保护的行，否则该表未受到保护。前一种情况说明该表是拥有行级别粒度的受保护的表，后一种情况说明该表是拥有列级别粒度的受保护的表。
- 如果该表没有与它关联的安全策略，那么使用 ALTER TABLE ... ADD 语句向现有表中添加 IDSSECURITYLABEL 列保护行的操作失败。
- 如果该表没有与它关联的安全策略，则保护列的 ALTER TABLE ... MODIFY ... COLUMN SECURED WITH 子句失败。
- 当表的安全策略被删除时（通过 ALTER TABLE DROP SECURITY POLICY 语句），IDSSECURITYLABEL 列也会自动被删除。如果该表拥有一个或多个保护的，则这些列变为不受保护。

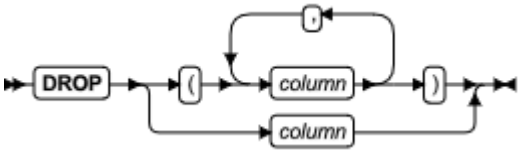
不要将 ALTER table 语句的 DROP SECURITY POLICY 子句与 DROP SECURITY POLICY 语句混淆。

- 当 ALTER TABLE 语句的 DROP SECURITY POLICY 子句执行成功时，它终止了表与安全策略的关联，删除 IDSSECURITYLABEL 列，并将表中受保护的数据的 LBAC 保护删除。然而，对于安全策略或被该策略保护的其它表，它没有影响。
- 当 DROP SECURITY POLICY 语句执行成功是，它的影响依赖于该策略是以 RESTRICT 方式或 CASCADE 方式删除，但是在任一种方式中，它终止了指定的策略。更多有关 SQL 的 DROP SECURITY POLICY 语句的信息和其限制信息，请参阅 DROP SECURITY 语句 的描述。

## DROP Column 子句

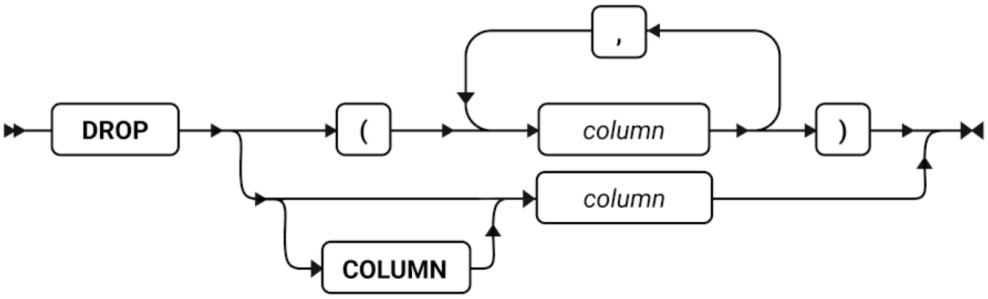
使用 DROP Column 子句移除表结构中的一列或多列。

DROP Column Clause




在 Oracle 模式下，单列删除时，被删除列前支持添加 COLUMN 关键字。

DROP COLUMN column Clause




例如，删除 test_tab 表中 col 列：

alter table test_tab drop column col;

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要删除的列的名称 | 必须存在于表中。任何分段表达式都不能引用该列，同时它不是表的最后一列。 | 标识符 |

您不能发出可删除表中每一列的 ALTER TABLE DROP 语句。至少一列必须保留于表中。

您不能删除是分片存储策略的分片键一部分的列。

由安全标签保护的列可以通过使用 ALTER TABLE DROP 语句删除，但是该用户必须是 DBSECADM 且还必须持有修改该表的结构的一般 CONNECT 、RESOURCE 和 ALTER 存取权限。

## 删除一列如何影响约束

当您删除列时，该列上的所有约束也被删除：

- 所有单列约束被删除。
- 所有引用该列的引用约束被删除。
- 所有引用该列的检查约束被删除。
- 如果列是多列主键约束或唯一约束的一部分，则多列上放置的约束也被删除。此操作接下来触发引用多列的所有引用约束的删除。

由于当删除列时任何与列相关联的约束被删除，因此当使用此子句时其它表的结构也可能改变。例如，如果被删除的列是在其它表中被引用的唯一键或者主键，则那些引用约束也被删除。因此那些其它表的结构也发生改变。

## 删除一列如何影响触发器

通常，从表中删除列时，基于该表的触发器不变。然而如果您删除的列出现在触发器操作子句中，删除该列可使得触发器无效。以下语句说明了可能对触发器的影响：

```
CREATE TABLE tab1 (i1 int, i2 int, i3 int);
CREATE TABLE tab2 (i4 int, i5 int);
CREATE TRIGGER col1trig UPDATE OF i2 ON tab1
BEFORE(INSERT INTO tab2 VALUES(1,1));
ALTER TABLE tab2 DROP i4;
```

ALTER TABLE 语句之后，**tab2** 仅有一列。**col1trig** 触发器已失效，因为当前用两列的值定义的操作子句不能发生。

如果您删除了在 UPDATE 触发器的触发列列表中出现的列，则数据库服务器从触发列列表删除该列。如果列是触发列列表的唯一成员，则数据库服务器从表删除该触发器。有关 UPDATE 触发器中触发列的更多信息，请参阅 CREATE TRIGGER 语句 。

如果更改底层的表使触发器失效，则删除然后重新创建触发器。

## 删除一列如何影响视图

当您从表删除列时，基于该表的视图保持不变。也就是，数据库服务器不自动从关联的视图删除对应的列。

视图没有自动删除，因为 ALTER TABLE 可通过删除一列然后用相同的名称添加一个新的列来更改表中列的顺序。这种情况下，基于已改变的表的视图继续有效，但是保留它们原始的列顺序。

如果更改底层的表使视图失效，您必须使用 DROP VIEW 和 CREATE VIEW 语句重新构建该视图。

## DROP AUDIT 子句

当表启用了选定行级审计时，使用带 ALTER TABLE 命令的 DROP AUDIT 子句从一组表中删除该表。

DROP AUDIT 子句只影响已标记列入选定行级审计中的表。如果您还未用 WITH AUDIT 子句或 ADD AUDIT 子句创建或更改该表，那么不必要使用 DROP AUDIT 从按行级审计的一组表中移除它。

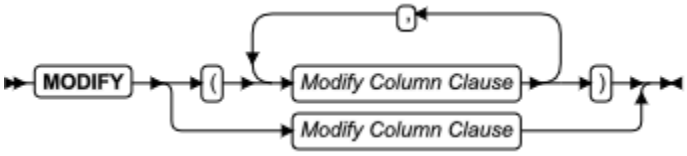
从表中移除 AUDIT 属性不会禁用或改变数据库中其它表选定行级审计。

您必须是 DBSSO 才能运行 ALTER TABLE 命令的 DROP AUDIT 子句。

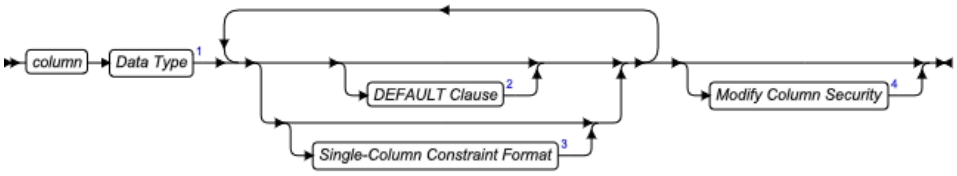
## MODIFY 子句

使用 MODIFY 子句更改列的数据类型、长度或缺省值，添加或删除列的安全标签，允许或不允许列中有 NULL 值或者重置 SERIAL 、SERIAL8 或 BIGSERIAL 列的序列号。

MODIFY Clause




Modify Column Clause




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要修改的列 | 必须在表中存在。不能是集合或 IDSSECURITYLABEL 数据类型。 | 标识符 |

您不能将列的数据类型更改为 COLLECTION 或 ROW 类型。

受保护表的 IDSSECURITYLABEL 列不能被不同的数据类型修改，现有的列也不能被更改为 IDSSECURITYLABEL 类型。

当修改列时，***所有*** 先前与该列相关联的属性（即，缺省值、单列检查约束或引用约束）被删除。当希望保留列的某些属性，例如 PRIMARY KEY ，您必须在同一 MODIFY 子句中重新指定那些属性。

例如，如果您将现有列 quantity 的数据类型更改为 SMALLINT ，但希望保留其缺省值（在这种情况中，为 1）和 NOT NULL 列属性，可以发出以下语句：

```
ALTER TABLE items MODIFY (quantity SMALLINT DEFAULT 1 NOT NULL);
```

**注：** 在 MODIFY 子句中两个属性都***再次***被指定。

在 oracle 模式下，使用 ALTER TABLE…MODIFY 语句修改列数据类型，被修改列上的默认约束保留。

例如，表 test_tab1 中 c1 列为 int 类型且存在 default 约束为 1201，修改 c1 数据类型为 varchar 类型后，默认约束保持不变：

```
create table test_tab1(
c1 int default 1201,
c2 integer
);
alter table test_tab1 modify (c1 varchar(20));

dbschema -d test2023 -t test_tab1
DBSCHEMA Schema Utility          GBASE-SQL Version 12.10.FC4G1TL
set environment sqlmode 'oracle';
{ TABLE test_tab1 row size = 26 number of columns = 2 index size = 0 ccolnum = 0
  commcol = c1,c2 }
```

```
create table test_tab1
(
      c1 varchar(20)
      default (1201 ),
      c2 integer
);
```

当在 MODIFY 子句中指定 PRIMARY KEY 约束，数据库服务器也会默示地在同一列创建 NOT NULL 约束或将同一组列更改为主键。

当更改列的数据类型时，数据库服务器不会在适当的位置执行修改。下个示例将 VARCHAR(15) 列更改为 LVARCHAR(3072) 列：

ALTER TABLE stock MODIFY (description LVARCHAR(3072));

当修改有列约束与之相关联的列时，以下约束被删除：

- 所有单列约束被删除。
- 所有引用该列的引用约束被删除。
- 如果被修改列是多列主键或者唯一约束的一部分，则所有引用多列的引用约束也被删除。

例如，如果修改具有唯一约束的列，则该唯一约束被删除。如果其它表中的列引用该列，则这些引用约束也被删除。另外，如果该列是多列主键或唯一约束的一部分，则不删除多列约束，但其它表放置在该列上的任何引用约束要被删除。

另一个示例，假设列是多列主键约束的一部分。其它两个表中的外键引用该主键。如果修改此列，则不删除多列主键约束，但是其它表放置在它上面的引用约束要被删除。

考虑以下语句定义的表：

CREATE TABLE tab1(c1 INT, c2 INT);

要添加 NOT NULL 约束，则需要 ALTER TABLE MODIFY 语句：

ALTER TABLE tab1 MODIFY (c1 INT NOT NULL);

不能使用 ADD CONSTRAINT 子句添加 NULL 或 NOT NULL 约束。

### 使用 MODIFY 子句

您试图修改对象的特征可影响您如何处理您的修改。

## 更改 BYTE 和 TEXT 列

您可使用 MODIFY 子句将 BYTE 列更改为 TEXT 列，反之亦然。还可使用 MODIFY 子句将 BYTE 列更改为 BLOB 列，将 TEXT 列更改为 CLOB 列。

然后，除去这些操作，您不能使用 MODIFY 子句将 BYTE 或 TEXT 列更改为其它任一一种类型的列，也不能将其它类型的列更改为 BYTE 或 TEXT 列。

当您使用此语句将 BYTE 列更改为 BLOB 列，或将 TEXT 列更改为 CLOB 列时，也可以使用 ALTER TABLE 语句的 PUT 子句 指定 sbspace 并定义它的特征以存储 BLOB 或 CLOB 对象。

## 更改下一个顺序值

您可以使用 MODIFY 子句来复位 SERIAL 或 BIGSERIAL 或 SERIAL8 列的下一个值。不能将下一个值设置为低于列中的当前最大值，因为该操作可导致数据库服务器生成重复数值。然而可将下一个值设置为任何高于当前最大值的值，这将在一系列值中创建间隔。

如果您指定的新顺序值小于顺序列中当前的最大值，则该最大值将保持不变。如果最大值小于您指定的值，那么下一个顺序数值将是您指定的值。在以下两种情况中，下一个顺序值不会大于该列中的最大顺序值：

- 当创建表（或是通过先前的 ALTER TABLE 语句创建）时，表中没有任何行，而且指定了一个初始的顺序值。
- 表中有行，但是前一个 ALTER TABLE 语句修改了下一个顺序值。

以下示例将下一个顺序值设置为 1000 ：

```
ALTER TABLE my_table MODIFY (serial_num SERIAL (1000));
```

作为备选方法，您可使用 INSERT 语句，在列中一系列顺序值中创建间隔。有关更多信息，请参阅将值插入到串行列之内 。

### *在类型表中更改下一个顺序值*

您可以使用 ALTER TABLE 语句的 MODIFY 子句对 ROW 类型字段设置初始顺序数值或修改下一个顺序数值。（当创建 ROW 数据类型时，您不能为顺序字段设置初始值。）

假设您有 ROW 类型 parent 、 child1 、 child2 和 child3 。

```
CREATE ROW TYPE parent (a int);
CREATE ROW TYPE child1 (s serial) UNDER parent;
CREATE ROW TYPE child2 (b float, s8 serial8) UNDER child1;
CREATE ROW TYPE child3 (d int) UNDER child2;
```

然后创建对应的类型表：

```
CREATE TABLE OF TYPE parent;
CREATE TABLE OF TYPE child1 UNDER parent;
CREATE TABLE OF TYPE child2 UNDER child1;
CREATE TABLE OF TYPE child3 UNDER child2;
```

要将下一个 SERIAL 和 SERIAL8 编号更改为 75 ，您可以输入以下语句：

```
ALTER TABLE child3 MODIFY (s serial(75), s8 serial8(75));
```

当执行 ALTER TABLE 语句时，数据库服务器更改 **child1** 、**child2** 和 **child3** 表中的对应的顺序列。

## 更改字符列

您可使用 MODIFY 子句更改现有 CHAR 、LVARCHAR 、NCHAR 、NVARCHAR 或 VARCHAR 列发布的长度。

类似地，MODIFY 子句可以将字符型列的数据类型更改为非字符数据类型。

对于修改为内置字符数据类型的列，显式或缺省的大小规范以字节为单位进行解释，除非创建数据库服务器时 SQL_LOGICAL_CHAR 配置参数的设置为在字符类型声明中启用逻辑字符语义。有关 ALTER TABLE 语句声明字符列的大小规范时逻辑字符语义的更多信息，请参阅 字符列中支持的逻辑字符。有关 SQL_LOGICAL_CHAR 配置参数的更多信息，请参阅 *GBase 8s 管理员参考手册* 。有关多字节本地语言环境和逻辑字符的其它信息，请参阅 *GBase 8s GLS 用户指南* 。

在创建为 NLSCASE INSENSITIVE 的数据库中，将 NCHAR 或 NVARCHAR 类型的字符列更改为 CHAR 、 LVARCHAR 或 VARCHAR 类型导致数据库服务器处理修改后的列值以区分大小写。

相反，在同一区分大小写的数据库中，将 CHAR 、LVARCHAR 或 VARCHAR 类型的字符列更改为 NCHAR 或 NVARCHAR 类型时，导致在已修改的列中值要进行区分大小写的处理。（数据值没有改变，但是在对这些值比较和排序操作中会忽略其字母大小写的变化。）

## 更改表结构

当使用 MODIFY 子句时，还可更改其它表的结构。如果其它表引用已修改的列，则那些引用约束被删除。必须使用 ALTER TABLE 语句，再次将那些约束添加到引用表。

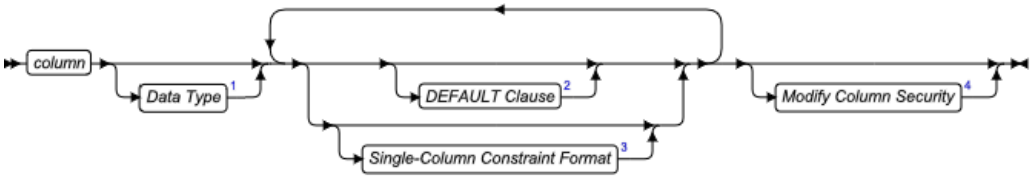
当更改现有列的数据类型时，所有数据均转换到新数据类型，包括数字转换到字符以及字符转换到数字（如果这些字符代表数值的话）。以下语句更改 `quantity` 列的数据类型：

```
ALTER TABLE items MODIFY (quantity CHAR(6));
```

但是，当存在主键或唯一约束时，则仅当不违反约束时发生转换。如果数据类型转换导致出现重复值（例如，通过将 FLOAT 更改为 SMALLFLOAT，或通过截短 CHAR 值），那么 ALTER TABLE 语句失败。

## 省略 Data Type 添加约束

Modify Column Clause




此功能仅在 Oracle 模式下支持，可以发出以下语句设置 Oracle 模式：

```
set environment sqlmode ‘oracle’;
```

当修改列时，仅修改该列相关联的属性，包括 DEFAULT 缺省值、单列 NULL 或 NOT NULL 约束，可省略定义数据类型，该列数据类型与修改前数据类型保持一致。其余约束不可省略数据类型。

例如，如果您将现有数据类型 SMALLINT 的列 **quantity** 缺省值更改为 1 并添加 NOT NULL 列属性，可以发出以下语句：

```
ALTER TABLE items MODIFY quantity DEFAULT 1 NOT NULL;
```

*Data Type*、*DEFAULT Clause*、*Single-Column Constraint Format* 必须在 MODIFY 子句中定义其中之一，不能全部省略。例如，如果您试图仅定义 *column* 执行更改，MODIFY 语句将失败：

```
ALTER TABLE items MODIFY quantity;  ----语句失败
```

## 为 NULL 值修改表

在列未包含 NULL 值的前提下，您可以将以前允许 NULL 的现有值修改为不允许 NULL 。要做到这点，使用相同的列名和数据类型以及 NOT NULL 关键字来指定 MODIFY 。那些关键字在列上创建 NOT NULL 约束。

您可将不允许 NULL 的现有列修改为允许 NULL 。要做到这点，使用列名和现有的数据类型来指定 MODIFY ，并省略 NOT NULL 关键字。NOT NULL 关键字的省略删除了列上的 NOT NULL 约束。如果列上存在唯一索引，您可使用 DROP INDEX 语句将之除去。

在不允许 NULL 值的现有列中允许 NULL 值的备选方法是，使用 DROP CONSTRAINT 子句删除列上的 NOT NULL 约束。

当定义 PRIMARY KEY 约束时，数据库服务器也默示地在同一列上创建 NOT NULL 约束，或将同一列组变为主键。

## 向非 Opaque 列添加约束

ALTER TABLE ... MODIFY 操作使用单列约束隐式地在非 opaque 列创建索引，并自动计算指定的列的分布存储。当定义下定义约束时，分布存储统计信息为该表设置查询计划时，这些统计信息可用于查询优化器：

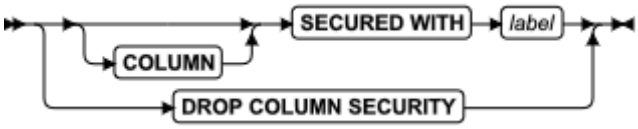
- 对于作为 B-tree 索引的新的约束，该重新计算的列分布统计信息相当于在 HIGH 模式下 UPDATE STATISTICS 语句创建的分布。
- 如果新的约束不是 B-tree 索引，对应自动重新计算的分布统计信息由 UPDATE STATISTIC 语句在 LOW 模式下创建。

有关在现有表上创建索引或约束时自动产生统计分布的其它信息，另见 CREATE INDEX 语句中 自动计算分布统计信息 章节的描述。

## Modify Column Security

Modify Column Security 子句只对按安全策略保护的表有效。使用此子句给列添加或删除行级安全。

Modify Column Security 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 安全标签的名称 | 必须存在且必须属于保护该表的同一安全策略。 | 标识符 |

此子句可以添加或删除行级保护。

- 要删除列的行级保护，请指定 DROP COLUMN SECURITY 关键字。
- 要为列提供的行级保护，请指定 SECURED WITH ***label*** （或者 COLUMN SECURED WITH ***label***）。

该安全标签可以是保护该表的其它行或列的同一标签，或者是同一安全策略的不同标签。以下限制应用于 SECURED WITH 标签选项：

- 该列不能是 IDSSECURITYLABEL 类型。
- 指定不带 policy 限定符的 ***label*** 而不是 ***policy.label*** 。
- ***label*** 必须是保护此表的安全策略的标签。

## 当现有行违法约束时添加约束

如果您使用 MODIFY 子句添加处于启用方式的约束，但因为现有行违反约束而接收到错误消息，则采取以下步骤来成功添加约束：

1. 添加处于禁用方式的约束。

   再次发出 ALTER TABLE 语句，但这次在 MODIFY 子句中指定 DISABLED 关键字。
2. 使用 START VIOLATIONS TABLE 语句启动违例和诊断表。
3. 发出 SET CONSTRAINTS 语句将约束的数据库对象方式切换到启用方式。

   当您发出该语句，将在违例表中复制目标表中违反约束的现有行；但是，您会接收到完整性违例错误消息，且约束保持为禁用。
4. 在违例表上发出 SELECT 语句来检索复制自目标表的不一致行。

   您可能需要连接违例表和诊断表来获得所有必要的信息。
5. 在目标表中违反约束的行上采取正确的操作。
6. 在您修正目标表中所有不一致行之后，再次发出 SET 语句以启用被禁用的约束。

   此时约束被禁用，并且没有返回任何完整性违例错误消息，因为此时目标表中的所有行均满足新约束。

## 修改列如何影响触发器

如果您修改了在 UPDATE 触发器的触发列列表中出现的列，则触发器保持不变。

当您修改表中的列，基于该表的触发器保持不变，但列修改可能会使触发器无效。

以下语句说明了可能对触发器的影响：

```
CREATE TABLE tab1 (i1 INT, i2 INT, i3 INT);
CREATE TABLE tab2 (i4 INT, i5 INT);
CREATE TRIGGER col1trig UPDATE OF i2 ON tab1
    BEFORE(INSERT INTO tab2 VALUES(1,1));
ALTER TABLE tab2 MODIFY i4 CHAR;
```

在 ALTER TABLE 语句之后，列 **i4** 仅接受字符值。因为字符列仅接受包含在引号中的值，所以 **col1trig** 触发器的操作子句失效。

如果修改基础表使得触发器失效，则删除再重新创建触发器。

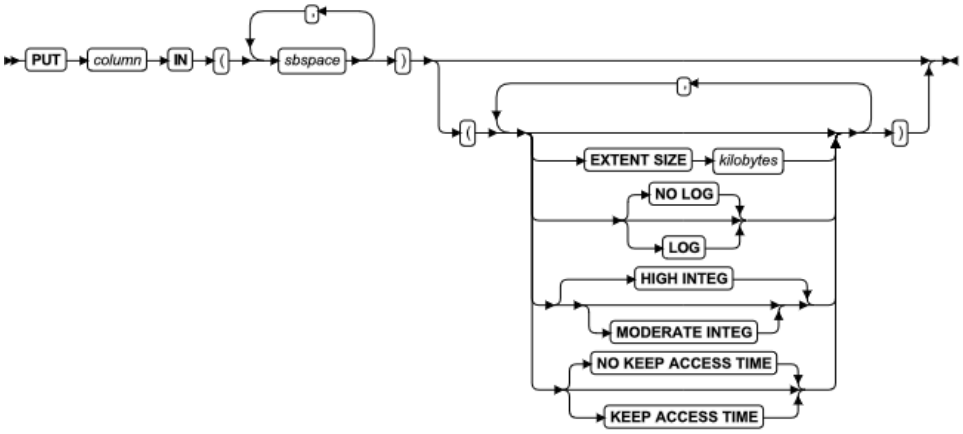
## 修改列如何影响视图

当您修改表中的列时，基于该表的视图保持不变。如果更改基础表使得视图失效，则必须重新构建该视图。

## PUT 子句

使用 PUT 子句为包含智能大对象的列指定存储空间（sbspace）。该子句可以指定新列的存储特征或替换现有列的存储特征。它的语法类似于 CREATE TABLE 语句的 PUT 子句，但是它只指定单个列而不是列的列表。

PUT 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要存储在指定 sbspace 中的列 | 必须是 UDT 、或 complex 、BLOB 或 CLOB 数据类型 | 标识符 |
| *kilobytes* | 要为 extent 大小分配的千字节数 | 必须是整数值 | 精确数值 |
| *sbspace* | 只能大对象的存储区域的名称 | sbspace 必须存在 | 标识符 |

当您修改列的存储特征时，所有先前与该列的存储空间相关联的属性均被删除。当您希望保留某些属性时，必须重新指定那些属性。例如，要保留日志记录，您必须重新指定 log 关键字。

格式 ***column.field*** 在此处无效。即，您所存储的智能大对象不能是 row 类型的一个字段。

当您修改存放智能大对象的列的存储特征时，数据库服务器不会更改已经存在的智能大对象，而是将新存储特征仅应用于那些 ALTER TABLE 语句生效后插入的智能大对象。

以下示例修改了表 sbtab ，将 BLOB 列 c1 存储于 sbspace sbs1 ，更改 extent 大小为 32 千字节，并打开了事务日志记录：

```
ALTER TABLE sbtab PUT c1 IN (sbs1) (EXTENT SIZE 32, LOG);
```

以下示例将日志记录状态更改为 NO LOG ，并不保留 BLOB 列最后一次的存取时间：

```
ALTER TABLE sbtab PUT c1 IN (sbs1) (NO LOG, NO KEEP ACCESS TIME);
```

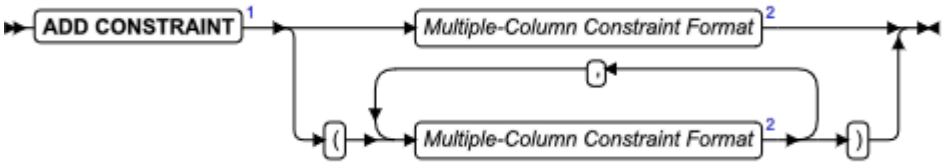
以下示例修改表，将 BLOB 列 c1 存储于 sbspaces sbs1 和 sbs2 ，更改 extent 大小为 100 千字节，并打开了事务日志记录，且保留了最后一次存取时间：

```
ALTER TABLE sbtab PUT c1 IN (sbs1, sbs2)
        (EXTENT SIZE 100, LOG, KEEP ACCESS TIME);
```

有关 PUT 子句的关键字描述和可用的存储特征的信息，请参阅 CREATE TABLE 语句 PUT 子句中与本节相对应的部分。有关大对象特征的讨论，请参阅 大对象数据类型 。

## ADD CONSTRAINT 子句

使用 ADD CONSTRAINT 子句指定新列或现有列或列组上的主键约束、外键约束、引用约束、唯一约束或检查约束。




例如，要将唯一约束添加至 customer 表的 fname 和 lname 列，请使用以下语句：

```
ALTER TABLE customer ADD CONSTRAINT UNIQUE (lname, fname);
```

要声明约束的名称，请更改上述语句（给该约束添加 CONSTRAINT 关键字和标识符）：

```
ALTER TABLE customer
ADD CONSTRAINT UNIQUE (lname, fname) CONSTRAINT u_cust;
```

在同一表上定义的约束的标识符的名称必须唯一。如果没有定义此约束的名称，则数据库服务器会为其指定系统定义的标识符，并将此名称存储至系统目录表的 sysconstraints.constrid 列。

缺省情况下，新约束是启用的。要添加未启用的约束，您可以在该约束的名称之后包含 DISABLED 关键字：

```
ALTER TABLE customer
ADD CONSTRAINT UNIQUE (lname, fname) CONSTRAINT u_cust DISABLED;
```

在您希望强制执行此约束 的 DML 操作之前，可以使用 SET Database Object Mode 语句启用该禁用的约束。

当您未指定新约束的名称时，则数据库服务器会提供一个名称。您可以在 **sysconstraints** 系统目录表中找到约束的名称。有关 **sysconstraints** 系统目录表的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

**由 ALTER TABLE 定义约束的限制**

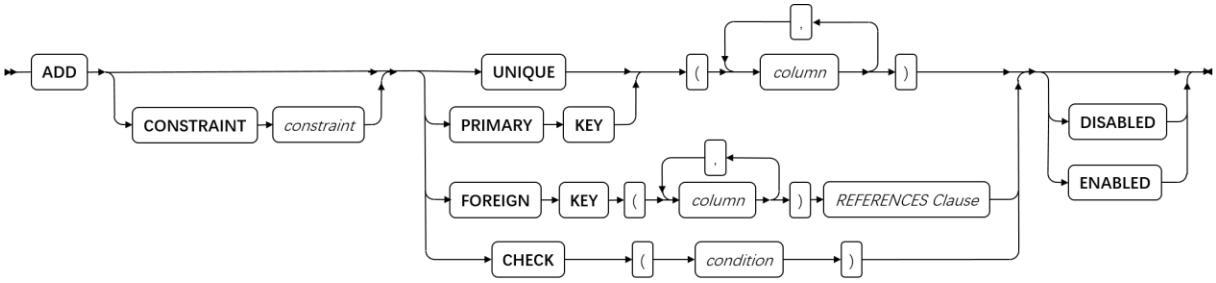
以下有关 ADD CONSTRAINT 子句（和 MODIFY 子句）的限制会影响 ALTER TABLE 语句定义的约束：

- 当您添加约束时，它的排列顺序必须与创建表时顺序一致。
- ADD CONSTRAINT 子句不能在任何数据类型的列上定义 NULL 或 NOT NULL 约束。只有 MODIFY 子句才能在现有表的列上定义 NULL 或 NOT NULL 约束。
- 您不能在 RAW 表上定义主键约束、外键约束或唯一约束。然而您可以使用 ALTER TABLE 语句的 MODIFY 子句在 RAW 表上的列中定义 NOT NULL 约束或 NULL 约束（但是不能都定义）。有关在现有表的列上添加 NULL 或 NOT NULL 约束的语法，请参阅 MODIFY 子句。
- 您不能在 BYTE 或 TEXTY 列上放置唯一约束或引用约束。
- BYTE 或 TEXTA 列的检查约束只检查 IS NULL 、IS NOT NULL 或 LENGTH 。
- 缺省情况下，每个 IDSSECURITYLABEL 列具有隐式 NOT NULL 约束。然而您不能使用 ADD CONSTRAINT 子句引用在单列约束的定义中的 IDSSECURITYLABEL 列，也不能引用作为多列引用约束或检查约束的一部分的 IDSSECURITYLABEL 列。

# ORACLE 模式

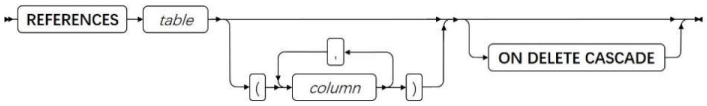
oracle 模式下，使用 alter table add constraint 命令向指定表的现有列或列组上增加主键约束、外键约束、唯一约束或检查约束；

ADD CONSTRAINT 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***constraint*** | 新增约束的名称 | 在数据库中的索引和约束名称中必须唯一 | 标识符 |
| ***column*** | 指定定义约束的列的名称 | 在当前表中必须存在 | 标识符 |

*REFERENCES Clause*




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***table*** | 被引用表的名称 | 被引用表与引用表必须在同一数据库中 | 标识符 |
| ***column*** | 被引用表中被引用的列的名称 | 在被引用表中必须存在 | 标识符 |

例如，要将唯一约束添加至 customer 表的 fname 和 lname 列，并声明约束的名称，请使用以下语句：

```
ALTER TABLE customer ADD CONSTRAINT u_cust UNIQUE (lname, fname);
```

由 ALTER TABLE 定义约束的限制：

- 定义的约束名称必须在索引和约束名称中是唯一的，如果没有定义此约束的名称，则数据库服务器会为其指定系统定义的标识符，并将此名称存储至系统表的 sysconstraints.constrid 列
- 缺省情况下，新约束是启用的。要添加未启用的约束，您可以在该约束的名称之后包含 DISABLED 关键字

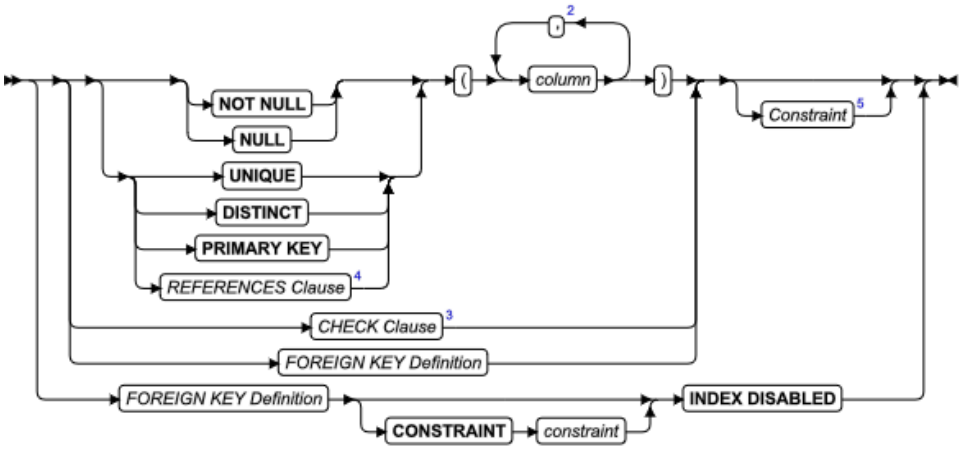
- 不能在 RAW 表上定义主键约束、外键约束或唯一约束。
- 不能在全局临时表上定义外键约束

## 多列约束格式

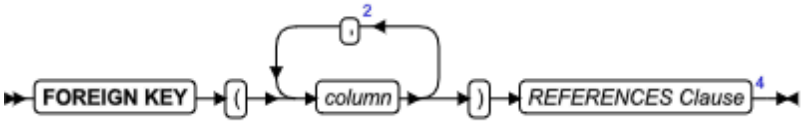
使用此选项将一个或多个约束指定到现有表中的一个列或一组列。

它与 CREATE TABLE 语句的多列约束格式极其类似，但是可选的 INDEX DISABLED 关键字在 CREATE TABLE 语句定义的外键约束中无效（返回错误）。

多列约束格式




FOREIGN KEY 定义




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 放置约束的列 | 不超过 16 列 | 标识符 |
| *constraint* | 禁用的外键约束的名称 | 在数据库中的索引和约束的名称中它必须唯一 | 标识符 |

在 CREATE TABLE 语句中，ALTER TABLE 的多列约束格式与单列约束格式不同之处在于，当您指定外键约束时，在 REFERENCES 子句之前需要 FOREIGN KEY 关键字。此外，如它的名称所指，多列约束格式可指定列列表作为新约束的范围，此语法同样对单列有效。

有关 INDEX DISABLED 关键字选项的更多信息，请参阅 在外键的定义中使用 INDEX DISABLED 关键字 。

多列约束具有以下基数和大小限制：

- 可包含的列名不多于 16 个。
- 最该列列表的总长度的最大值依赖于页大小，计算公式为：

MAXLength = (((PageSize - 93)/3) -1)

- 对于 2K 的页大小，总长度不能超过 650 字节。
- 对于 16K 的页大小，总长度不能超过 5429 字节。

此处的反斜杠（ / ）符号代表整除。

如果您在同一列上定义了 NOT NULL 约束和 NULL 约束或在缺省值为 NULL 的列上定义了 NOT NULL 约束，则该语句失败并返回错误。

您不能在 LIST 、MULTISET 、SET 或 IDSSECURITYLABEL 数据类型的列上定义 NULL 约束。

如果一组列中有一列存储了加密数据，则 GBase 8s 无法强制执行此组列的约束。您可以发布此约束的名称并将用 约束定义 设置它的方式。

如果 ALTER TABLE ADD CONSTRAINT 语句在同一表中了定义多个引用约束，则每个约束需要它自己所有的 REFERENCES 子句，因此每个单独的约束可指定（或忽略）类似 ON DELETE CASCADE 的选项，而不是将其应用至所有的约束。

如果数据库服务器在同一非 opaque 列或列组上隐式地创建了索引作为引用约束，则会自动计算指定列的分别存储统计或多列约束的主列。

这些分布存储统计相当于 STATISTICS 语句在 HIGH 方式中创建的分布，且当其作为创建了新约束的表查询计划时，可用于查询优化器。有关在现有表上创建索引或约束时计算分布存储的其它信息，请参阅 CREATE INDEX 语句中自动计算分布统计信息章节中的描述。

## 当被引用表上存在索引时创建外键约束

缺省情况下，当 ALTER TABLE 语句的 ADD CONSTRAINT 或 MODIFY 选项包含 REFERENCES 关键字以定义一个外键约束时，数据库服务器自动验证启用的引用约束。如果被引用表的列（列组）上已经含有与引用约束的键对应的唯一索引或主键约束，则在外键约束验证期间，您可能节省一些时间。

数据库服务器根据如何验证外键约束而做出成本基础决策。在许多情况下索引键算法可能较快，因为它在验证约束时，只扫描索引值而不是扫描表中所有的行。

数据库服务器可以考虑使用索引键算法验证外键约束，但是仅在以下所有条件都满足的情况下，才能使用该算法：

- ALTER TABLE 语句正在创建只有一个外键约束。

  如果在这种情况下，数据库服务器只需要检查要创建外键约束的列的单独值。而在同一时刻验证两个外键约束需要在同一扫描中使用两个指标，该情况不支持此行为。

- 该语句不只创建或启用一个 CHECK 约束。

  如果该 ALTER TABLE 语句正在创建多个约束，则验证 CHECK 约束需要检查每一行，而不是单个值。在这种情况下，索引键算法不能用于验证外键约束。

- 创建外键约束的语句不须更改同一表中任何现有列的数据类型。

如果创建外键约束的 ALTER TABLE 语包含更改任何列的数据库类型的 MODIFY 子句，则数据库服务器不会考虑使用索引扫描执行路径来验证约束。

- 此外键列不包括用户定义数据类型（UDTs）或内置 opaque 数据类型。

  要使快速索引键算法尽可能有效，它排除所有与用户定义或 opaque 数据类型相关联的低效的执行例程，例如：内置与 opaque 类型中的 BOOLEAN 和 LVARCHAR。

- 新外键约束的方式不能是 DISABLED 。

  如果它是禁用的，则不需要约束检查算法。因为不会发生验证参照完整性的检查。

- 该表与活动的违例表没有关联。

  违例表需要在同一时间检查，不满足新约束的每一行必须插入此违例表。验证时扫描每一行可防止数据库服务器使用快速索引键算法忽略的重复的行。

在一个行或多行违例的情况下，ALTER TABLE ADD CONSTRAINT 或 ALTER TABLE MODIFY 语句可以创建并验证某些不满足上述要求的外键约束。扫描整个表的额外验证成本一般与表的大小成比例。对于很大的表这些成本也将是巨大的。

当您创建了自我引用外键约束（其 REFERENCING 子句指定了定义该约束的同一表）时，数据库服务器可以考虑索引键算法用于验证参照完整性（如果以上条件都满足）。

## 在外键的定义中使用 INDEX DISABLED 关键字

当您定义外键约束时包含可选的 INDEX DISABLED 关键字，可以放置表上的 DML 操作使用与数据库服务器关联的外键的索引。如果您包含 INDEX DISABLED 关键字，则它们必须是 ALTER TABLE 语句的最后规范。

### 在 ALTER TABLE 语句中定义外键约束

要添加外键约束，您必须在被引用列或子表上拥有 References 权限。如果您拥有父表或有对父表的 Alter 权限，则您可以在该表上创建外键约束并指定您子句作为此约束的所有者。当您持有 DBA 权限时，您可以为其他用户创建外键约束。

当 ALTER TABLE ADD CONSTRAINT 语句在引用子表的一列或一组列上放置外键约束，且该列或该组列没有引用约束或用户定义索引存在时，数据库服务器在指定的列或列组上创建一个内部 B-tree 索引。如果用户定义索引已经存在，则约束共享现有的索引。

如果 ALTER TABLE ADD CONSTRAINT 语句在同一表上定义多个外键，则每个约束需要它自己的 REFERENCES 子句，并为每个约束指定（或忽略）INDEX DISABLED 关键字。

INDEX DISABLED 选项在从集群环境中更新辅助服务器语句中发出的 ALTER TABLE ADD CONSTRAINT 语句中有效。

### 外键索引可能降低性能的情况

尽管引用约束保护数据完整性，在某些情况下用户定义或系统生成的与数据库服务器关联的外键约束的 B-tree 索引可以降低表（表很大）的操纵数据操作的效率。如果未从父表删除，则该索引不用在子表中锁定行以级联删除。如果子表不需要使用此索引进行查询。在此情境中，并不需要索引，

但是这并不意味在子表中修改、删除和插入行的操作中该索引是不重要的。如果禁用对应于外键约束的索引，则在子表上存取数百万行数据的应用可能需要较少的资源。

在这些情况下，ALTER TABLE ADD CONSTRAINT 语句的 INDEX DISABLED 关键字选项提供一个机制，即定义外键时避免大量关联 b-tree 索引的开销。

当您在约束定义的末尾包含 INDEX DISABLED 关键字时，如果没有合适的用户定义索引存在，数据库服务器禁用系统生成的索引。如果在子表上的外键列或（列组）已存在用户定义索引，则数据库服务器禁用该索引。随后的子表上的 DML 存在会在没有索引的情况下实现，而且只需要最小的系统资源以维护和存储该禁用的索引。

**Effects of the INDEX DISABLED 关键字的影响**

当您在约束定义的末尾包含 INDEX DISABLED 关键字时，如果没有合适的用户定义索引存在，则数据库服务器禁用系统生成的索引。如果在子表上的外键列或（列组）已存在用户定义索引，则数据库服务器禁用该索引。随后的子表上的 DML 存在会在没有索引的情况下实现，而且只需要最小的系统资源以维护和存储该禁用的索引。

这些是您成功使用 INDEX DISABLED 选项添加外键约束时数据库服务器发生的操作：

- 与外键约束关联的索引被标识。
- 该索引被禁用，且在系统目录的 **sysobjstate** 表中标记为禁用。
- 该物理索引已从数据库中删除。
- **sysfragments** 系统目录表变更为显示没有存储分配给此索引。

INDEX DISABLED 关键字对于您定义的外键约束没有影响。数据库服务器强制执行此约束，如果随后对子表或父表的 DML 操作违例了指定外键约束则发出错误。

以下限制应用于约束定义中的 INDEX DISABLED 关键字：

- INDEX DISABLED 选项只在外键定义中可用。
- 只有 ALTER TABLE ADD CONSTRAINT 语句支持此语法。如果 CREATE TABLE 或者 ALTER TABLE MODIFY COLUMN 语句的外键定义包含 INDEX DISABLED 关键字，则 CREATE TABLE 或者 ALTER TABLE MODIFY COLUMN 语句返回异常。
- 如果被外键使用的索引正在被另一个约束使用，则数据库服务器返回错误。
- 如果您在约束定义中包含禁用该外键约束的 DISABLED 关键字，且您还指定了 INDEX DISABLED 关键字则数据库服务器返回错误。如下所示：

```
ALTER TABLE child ADD
        CONSTRAINT(FOREIGN KEY(x1) REFERENCES parent(c1)
        CONSTRAINT cons_child_x1 DISABLED INDEX DISABLED);
```

要纠正以上 ALTER TABLE ADD CONSTRAINT 示例中的错误，您必须删除第一个 DISABLED 关键字，或者删除 INDEX DISABLED 关键字。

**使用 INDEX DISABLED 创建外键约束的示例**

假设在以下示例中的 parent 表和 child 表具有一个主键约束和外键约束，且存储在这些表中您的数据满足下列条件：

- **parent** 表只有少量行。
- **child** 表由=有百万条行。
- **child** 表中外键只有一些基于 **parent** 表的主键的不同的可能值。

该示例显示了如何使用 ALTER TABLE ADD CONSTRAINT 语句的 INDEX DISABLED 选项。

```
CREATE TABLE parent(c1 INT, c2 INT, c3 INT);
        CREATE UNIQUE INDEX idx_parent_c1 ON parent(c1);
        ALTER TABLE parent ADD
        CONSTRAINT PRIMARY KEY(c1)
        CONSTRAINT cons_parent_c1;
        CREATE TABLE child(x1 INT, x2 INT, x3 VARCHAR(32));
        CREATE INDEX idx_child_x1 ON child(x1);

ALTER TABLE child ADD
        CONSTRAINT(FOREIGN KEY(x1) REFERENCES parent(c1)
        CONSTRAINT cons_child_x1 INDEX DISABLED);
```

在以上示例中，

- **cons_parent_c1** 是 **parent** 表的一个主键约束，
- **cons_child_x1** 是 **child** 表的一个外键约束，
- **idx_parent_c1** 是唯一索引，且被 **cons_parent_c1** 约束共享，
- 并且 **idx_child_x1** 是被 **cons_child_x1** 约束共享的索引。

子表上的数据操纵语言操作（例如：UPDATE 、DELETE 、INSERT 和 MERGE ）不能使用被外键约束共享的 idx_child_x1 索引，因为该索引现在是禁用的。

然而，对于某些含有主键和外键依赖的表，查询优化器可能会在执行计划中选择基于 WHERE 子句子表上的其它索引。

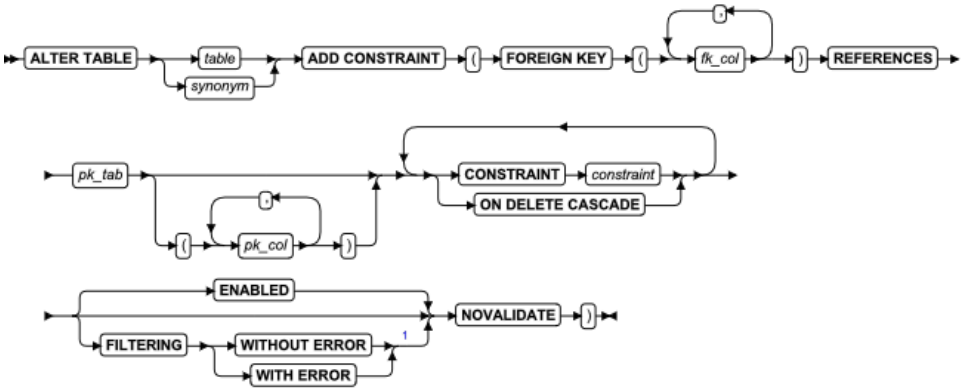
如上所述，在外键定义中使用 INDEX DISABLED 选项只有在子表非常大时才可能提高性能，通常在数据仓库应用的情况下。不建议在小的表上使用该语法选项操作。

## 在 NOVALIDATE 方式下创建外键约束

ALTER TABLE ADD CONSTRAINT 语句可以创建或过滤 NOVALIDATE 方式的外键约束。NOVALIDATE 约束方式防止数据库服务器在创建引用约束时，验证每一行的外键值是否符合被引用表中的主键值。

使用此语法创建启用或过滤 NOVALIDATE 方式的外键约束：

ALTER TABLE ADD CONSTRAINT




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constraint* | 发布此处约束的名称 | 在数据库中的索引和约束的名称中必须唯一 | 标识符 |
| *fk_col* | *constraint* 的外键列 | 必须存在于子表 | 标识符 |
| *pk_col* | 被引用表中的外键列 | 必须存在于被引用表 | 标识符 |
| *pk_tab* | 被引用表的名称 | 必须存在于当前数据库 | 标识符 |
| *table*, *synonym* | 放置 *constraint* 的表 | 必须存在于当前数据库 | 标识符 |

## 用法

该语法忽略 DISABLED 关键字。因为禁用的约束不会进行违例检查，在这种情况下，NOVALIDATE 关键字不重要。

如果没有列或列表立即跟随在 REFERENCES 关键字之后，则缺省列（或列组）是 ***pk_tab*** 表的主键。如果 ***pk_tab*** 和 ***table*** 或 ***synonym*** 指定同一表，则约束自我引用，并没有缺省的主键列。

如果您没有声明约束的名称，则数据库服务器为此新约束生成一个标识符，它注册在 **sysconstraints** 和 **sysobjstate** 系统目录表中。

ALTER TABLE ADD CONSTRAINT 语句支持引用约束的 NOVALIDATE 方式作为创建或过滤引用约束时绕过数据完整性检查的机制。

### NOVALIDATE 方式可以提高性能的情况

尽管引用约束保护数据完整性，在某些情况下您正要移动到新数据库服务器实例的数据库表是已知的自由参照完整性违规。对于大表上的外键约束，验证约束所需的时间要十分充分。如果有数百万行的表正从 OLTP 环境移动至数据仓库环境，则验证目标环境中的外键可能增加数量级迁移所需的时间。

例如，您可以删除大表上的外键约束，然后在该表迁移到目标表环境之前，立即重新创建 ENABLED NOVALIDATE 方式或 FILTERING NOVALIDATE 方式的约束。重建外键约束的 ALTER TABLE ADD CONSTRAINT 操作的花销几乎很小，因为它绕过了对每一行引用约束的验证。因为 NOVALIDATE 方式不会持续超过创建该约束的 ALTER TABLE 操作，抵达数据仓库环境中的表带有 ENABLED 或 FILTERING 方式的约束，保护了随后 DML 操作中数据的参照完整性。

**使用 NOVALIDATE 关键字的限制**

创建外键约束时， NOVALIDATE 关键字仅在 ALTER TABLE ADD CONSTRAINT 语句的 DDL 上下文中有效。 例如，您不能在以下任何 SQL 语句中创建处于 NOVALIDATE 方式的外键约束：

- CREATE TABLE 语句
- CREATE TEMP TABLE 语句
- SELECT INTO TABLE 语句

如果满足下列条件，则您可以使用 ALTER TABLE ADD CONSTRAINT 语句在现有表上创建处于 NOVALIDATE 方式的启用约束：

- 您正在添加的约束是外键约束。如果您在同一 ALTER TABLE 语句中创建多个约束，那么所有的约束必须都是外键约束。
- 在 ALTER TABLE 语句中，NOVALIDATE 关键字只在 ADD CONSTRAINT FOREIGN KEY 选项中有效。
- ALTER TABLE 在 DISABLED 方式下创建的约束是无效的。

如果在以下约束定义的语法上下文中包含 NOVALIDATE 关键字，则 ALTER TABLE 语句发生错误而失败：

- ALTER TABLE ADD COLUMN 语句
- ALTER TABLE INIT 语句
- ALTER TABLE MODIFY 语句

使 NOVALIDATE 关键字有效的其它 DDL 语句只有 SET Database Object Mode 语句的 SET CONSTRAINTS 选项。当运行 SET CONSTRAINTS 语句时，它可以将现有外键约束的方式更改为这些 NOVALIDATE 约束方式：

- ENABLED NOVALIDATE 方式
- FILTERING WITH ERROR NOVALIDATE 方式
- FILTERING WITHOUT ERROR NOVALIDATE 方式.

有关更多信息，请参阅 SET CONSTRAINTS 语句。

**建立 NOVALIDATE 方式作为缺省方式**

如果约束方式规范忽略 NOVALIDATE 关键字，则 SQL 的 SET ENVIRONMENT NOVALIDATE ON 语句和加载数据库的 dbimport -nv 命令都可以重写任何由 ALTER TABLE ADD CONSTRAINT 或 SET CONSTRAINTS 语句指定的外键约束方式（除了 DISABLED）。

- SET ENVIRONMENT NOVALIDATE ON 语句的范围是同一用户会话中 ALTER TABLE ADD CONSTRAINT 和 SET CONSTRAINTS 语句的后续。
- dbimport -nv 命令的范围是导出数据库的 .sql 文件中的 ALTER TABLE ADD CONSTRAINT 和 SET CONSTRAINTS 语句，其路径名在同一 dbimport 命令中指定。

**创建 NOVALIDATE 方式的约束的示例**

以下 DDL 语句创建了名为 parent 的表且在该表的 c1 列上定义了唯一索引和主键约束：

```
CREATE TABLE parent(c1 INT, c2 INT, c3 INT);
        CREATE UNIQUE INDEX idx_parent_c1 ON parent(c1);
        ALTER TABLE parent ADD CONSTRAINT
        PRIMARY KEY(c1) CONSTRAINT cons_parent_c1;
```

以下语句创建了另一个表，名为 **child**，它的第一列与 **parent** 表的主键列的数据类型相同，并在 **child** 表定义主键约束 **cons_child_x**：

```
CREATE TABLE child(x1 INT, x2 INT, x3 VARCHAR(32));
        ALTER TABLE child
        ADD CONSTRAINT (FOREIGN KEY(x1)
        REFERENCES parent(c1) CONSTRAINT cons_child_x1);
```

假设之后的 DML 操作（未显示）向此 **parent** 表和 **child** 表填充数据行。在一些点，此工作流程要求数据从它的 OLTP 产品环境移动到另一个处理业务分析应用程序的数据库中。

如果在这一点，child 表中的数据集包含大量行，则验证 cons_child_x1 引用约束将会是新数据库导入 child 表要花费的显著成本。以下语句删除此约束：

```
ALTER TABLE child DROP CONSTRAINT cons_child_x1;
```

child 表导入到新的环境后，下列语句可以在 child 表上重建一个相同名称的约束，而不用检查每一行的参照完整性违规：

```
ALTER TABLE child
        ADD CONSTRAINT (FOREIGN KEY(x1)
        REFERENCES parent(c1)
        CONSTRAINT cons_child_x1 NOVALIDATE);
```

ALTER TABLE 语句执行完毕之后，新的 **cons_child_x1** 引用约束在缺省情况下处于 ENABLED 方式。

## 添加主键或唯一约束

当您在一列或一组列上放置主键或唯一约束时，那些列必须包含唯一值。数据库服务器检查是否存在现有约束和索引：

- 如果该列或该组列上已经存在用户创建的唯一索引，则约束将共享该索引。
- 如果该列或该组列上已经存在用户创建的允许重复的索引，则数据库服务器返回错误。
- 在这种情况下，您必须在添加主键或唯一约束之前删除现有索引。

- 如果在该列或该组列上已经存在引用约束，则重复索引将升级到唯一索引（如果可能）并且共享该索引。
- 如果该列或该组列上不存在引用约束或用户创建的索引，则数据库服务器在指定列上创建内部 B-tree 索引。

如果您在一列或一组列上放置引用约束，而该列或该组列上已经存在索引，则将共享该索引。

如果您拥有该表，或拥有对该表的 Alter 权限，则可以在该表上创建检查、主键或唯一约束并指定您自己作为该约束的所有者。要添加引用约束，您必须对被引用的列或被引用的表拥有 References 权限。当您拥有 DBA 权限时，可以为其他用户创建约束。

## 从约束违例中恢复

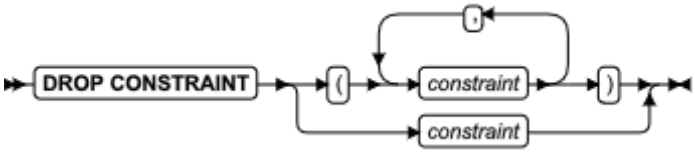
如果您使用 ADD CONSTRAINT 子句添加处于启用方式的约束，就会接收到错误消息，因为现有行会违反约束。有关成功添加约束的过程，请参阅当现有行违法约束时添加约束。

## DROP CONSTRAINT 子句

使用 DROP CONSTRAINT 子句删除指定的约束。

ALTER TABLE 语句的 DROP CONSTRAINT 子句具有此语法：

DROP CONSTRAINT 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constraint* | 要删除的约束 | 在数据库中必须存在 | 标识符 |

**用法**

要删除现有约束，请指定 DROP CONSTRAINT 关键字和此约束的名称。要删除同一表上的多个约束，则约束名称列表必须逗号分隔并由括号分隔。

您要删除的约束可具有 ENABLED 、DISABLED 或 FILTERING 方式。

这里是一个删除约束的示例：

```
ALTER TABLE manufact DROP CONSTRAINT con_name;
```

以下示例删除了定义在 **orders** 表中的引用约束和检查约束：

```
ALTER TABLE orders DROP CONSTRAINT (con_ref, con_check);
```

GBase 8s 的 SQL 执行包含非 DROP CONSTRAINT 语句。然而，如果该语句存在，则 ALTER TABLE 语句的此子句提供一个 DROP CONSTRAINT 语句除外的功能。

当 DROP TABLE 语句删除该表时，会默示地删除该指定表上所有的约束。

**获取约束名称**

DROP CONSTRAINT 子句需要约束的名称。如果创建约束时没有声明名称，则数据库服务器生成新的约束的名称。您可以查询 **sysconstraints** 系统目录表以获得约束的名称的所有者。例如，要查找位于 **items** 表上约束的名称，您可以发出以下语句：

```
SELECT constrname FROM    sysconstraints
        WHERE tabid = (SELECT tabid FROM systables
        WHERE tabname = 'items');
```

**约束之间的依赖**

当您删除拥有对应外键的主键约束或唯一约束时，任何相关联的引用约束也会被删除。

例如，在 **stores_demo** 数据库中，**orders** 表的 **order_num** 列上具有主键约束。对应的外键约束也定义在 **items** 表中的 **order_num** 列。这些约束定义在这两个表中的 **order_num** 列定义了引用关系。

假设您运行 ALTER TABLE orders DROP CONSTRAINT 语句删除 orders 表 order_num 列上的主键约束。因为在这两个表之间的参照完整性关系不能没有主键约束，如果此示例中的 ALTER TABLE 语句成功，则数据库服务器将会采取这些操作：

- 删除 **orders** 表 **order_num** 列上指定的主键约束。
- 删除 **items** 表 **order_num** 列上对应的引用约束。
- 从系统目录中删除所有引用 **orders** 表的主键或 **items** 表上引用约束。

**删除约束对系统目录的影响**

数据库服务器维护这些系统目录表中的现有约束的信息：

- **sysconstraints**（所有的约束）
- **sysobjstate**（所有的约束）
- **syschecks** （检查约束）
- **syscoldepend**（检查约束和 NOT NULL 约束）
- **syscheckudrdep**（UDR 引用的检查约束）
- **sysreferences**（引用约束）
- **sysindices**（ 在 **sysindices** 中没有对应所有项的引用、主键或唯一约束）

当 DROP CONSTRAINT 子句成功删除约束之后，数据库服务器至少删除或更新以上一个或多个表的一行。

**数据类型的注意事项**

缺省情况下，每一 IDSSECURITYLABEL 列拥有一个隐式的 NOT NULL 约束，但是 DROP CONSTRAINT 子句不能引用 IDSSECURITYLABEL 类型的列。

您可以在 ROW 数据类型的已分类表的 ALTER TABLE 操作中包含 DROP CONSTRAINT 子句。

**还原引用约束**

对于某些操作，例如将表重新定位到另一个数据库中，您可能要求引用约束暂时不会对它的表产生影响。然而，当您不带此约束而完成操作后，数据库的参照完整性通常要求还原此约束的功能。可能的选项是：

- 使用 DROP CONSTRAINT 子句删除此约束。
- 完成此任务需要避免该约束的影响。
- 使用 ALTER TABLE ADD CONSTRAINT 重建此约束。

对于符合已删除外键约束的大表，在 ALTER TABLE ADD CONSTRAINT 已经中使用 NOVALIDATE 选项可以避免创建约束时使用全表扫描验证约束的显著成本。此 NOVALIDATE 选项要求使 Multi-Column Constraint Format 语法定义此约束。

类似地，比起删除大表上的引用约束，您可以禁用它，然后完成任务，使用禁用的约束需要较少的资源。要恢复约束的强制性，您可以使用 SET Database Object Mode 语句的 SET CONSTRAINTS 选项将此对象的方式重置为 ENABLED NOVALIDATE 或 FILTERING WITH ERROR NOVALIDATE 或 FILTERING WITHOUT ERROR NOVALIDATE。在以上每一种约束方式中，NOVALIDATE 关键字避免了重置方式时验证约束的开销。

## DROP PRIMARY KEY

oracle 模式下，使用 alter table tabname drop primary key 命令可以删除指定表的主键约束。

DROP PRIMARY KEY 子句


删除主键约束时，与之关联的外键约束和唯一索引也被删除。

例如，将 products 表上的主键删除，可以使用以下语句：

```
ALTER TABLE products DROP PRIMARY KEY;
```

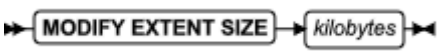
## MODIFY EXTENT SIZE 子句

使用 ALTER TABLE 语句的 MODIFY EXTENT SIZE 子句更改数据库中表的第一个 extent 的大小。

您不能使用该 MODIFY EXTENT SIZE 子句更改这些 extent 的大小：

- 在 blobspace 中的表中的第一个 extent
- 外部表、虚拟表或系统目录表中的第一个 extent
- tblspace **tblspace** 中的第一个 extent

MODIFY EXTENT SIZE 子句



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *kilobytes* | 在此为该表的第一个 extent 分配的长度（以千字节） | 规范不可以是变量，而且（4（页大小））≤ *千字节* ≤（chunk 大小） | 表达式 |

最小大小是磁盘页大小的 4 倍。例如，在一个 2 千字节页的系统上，最小长度是 8 千字节。最大长度等于 chunk 大小。

以下示例指定了一个 32 千字节大小的 extent ：

```
ALTER TABLE customer MODIFY EXTENT SIZE 32;
```

当您更改第一个 extent 的大小时，数据库服务器记录系统目录和分区页的更改，但是在重建表或创建分区和分片时，只记录实际的更改。

例如，如果表拥有一个 8 千字节大小的首个 extent，且您使用 ALTER TABLE 语句将其大小修改为 16 千字节，则数据库不会删除当前首个 extent 并按新的大小重建它。而是，只有在服务器在该表上重建聚集索引或从该表拆离分片操作之后重建此表时，第一个 16 千字节大小的 extent 才会生效。

如果一个不带 REUSE 选项的 TRUNCATE TABLE 子句在带有 MODIFY EXTENT SIZE 子句的 ALTER TABLE 语句之前执行，则第一个 extent 的大小不能发生改变。

如果在 dbspace 中现有表含有数据，并已为该表分配了第一个和第二个 extent ，则您将不能更改第一个和第二个 extent 的大小。如果您想要更改现有 extent 的大小，则必须删除此表，用包含希望的值的强有力的子句重建该表，并再次加载数据。

您可以同时更改第一个和第二个 extent 的大小。以下示例指定更改第一个和第二个 extent 的大小：

```
ALTER TABLE customer MODIFY EXTENT SIZE 32 NEXT SIZE 32
```

第一个和下一个 extent 的大小记录在 PNSIZES 逻辑日志记录中。

## MODIFY NEXT SIZE 子句

使用 MODIFY NEXT SIZE 子句更改下一个 extent 的大小。

MODIFY NEXT SIZE 子句


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *kilobytes* | 在此为该表的下一个 extent 分配的长度（以千字节） | 规范不可以是变量，而且（4（页大小））≤ *千字节* ≤（chunk 大小） | 表达式 |

最小大小是磁盘页大小的 4 倍。例如，在一个 2 千字节页的系统上，最小长度是 8 千字节。最大长度等于 chunk 大小。以下示例指定了一个 32 千字节大小的 extent ：

```
ALTER TABLE customer MODIFY NEXT SIZE 32;
```

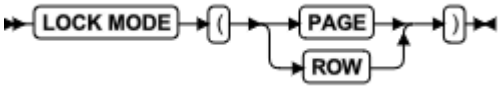
此子句不能更改现有 extent 的大小。如果不卸装所有数据，就不能更改现有 extent 的大小。

要更改现有 extent 的大小，您必须卸装所有数据，删除表明仔数据库模式的 CREATE TABLE 定义中修改 ***first-extent*** 和 ***next-extent*** 的大小，重新创建数据库并重新加载数据。关于如果优化 extent 大小的信息，请参阅 *GBase 8s 性能指南* 。

## LOCK MODE 子句

使用 LOCK MODE 关键字更改表的锁定粒度。

LOCK MODE 子句




下表描述了可用的锁定粒度选项。

| 粒度 | 作用 |
| --- | --- |
| PAGE | 在一整页的行上获取并释放一个锁<br>这是缺省锁定粒度。当您知道行分组到各页所依照的顺序与您正在用来处理所有行的顺序相同时，页级别锁定就特别有用。例如，如果您依照与表的集群索引相同的顺序来处理表的内容，页锁定就特别合适。 |
| ROW | 在每一行上获取并释放一个锁<br>行级别锁定提供最高级别的并发性。只有具有行级别锁定的表才支持 LAST COMMITTED 功能，当有另一个会话在您尝试读取的行上持有互斥锁时，它会提高 Committed Read 和 Dirty Read 隔离级别的性能。然而如果您正在一次使用许多行，则锁管理开销可能变得很重要。根据您的数据库服务器的配置，也可以超出可用锁的最大数目。 |

以下语句更改将 customer 表的锁定方式更改为页级别锁定：

```
ALTER TABLE customer LOCK MODE(page);
```

下一示例将 customer 表的锁定方式更改为行级别说点：

```
ALTER TABLE customer LOCK MODE(row);
```

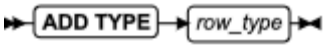
## 优先顺序和缺省行为

ALTER TABLE 语句中的 LOCK MODE 设置优先于 **IFX_DEF_TABLE_LOCKMODE** 环境变量和 DEF_TABLE_LOCKMODE 配置参数的设置。有关 **IFX_DEF_TABLE_LOCKMODE** 环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》* 。有关 DEF_TABLE_LOCKMODE 配置参数的信息，请参阅 *GBase 8s 管理员参考* 。

## ADD TYPE 子句

使用 ADD TYPE 子句将未基于指定的 ROW 数据类型的表转换为类型表。此子句是 SQL ANSI/ISO 标准的扩展。

ADD TYPE 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *row_type* | 添加到表的 ROW 数据类型的名称 | 该 *row_type* 字段必须与列数据类型的顺序和数量相匹配 | 标识符 |

当您使用 ADD TYPE 子句时，就将指定的已命名的 ROW 数据类型分配到所包含的列与该数据类型的字段相匹配的表。

除了所有的 ALTER TABLE 操作的共同要求 （即数据库的 DBA 权限、表的 Alter 权限、表的所有权），当您使用 ADD TYPE 子句将一个未分类的表转换为指定的 ROW 数据类型时，以下所有的条件必须成立：

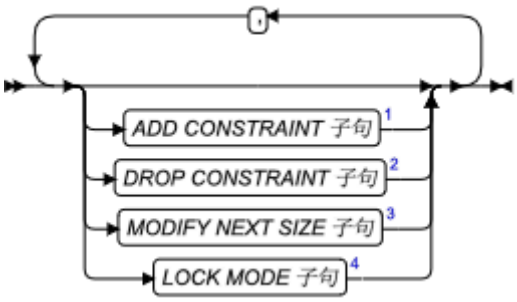
- 在数据库中该指定的 ROW 数据类型已存在。
- 您对指定的 ROW 数据类型持有 Usage 权限。
- 未分类表列的数据类型的顺序与指定 ROW 数据类型字段数据的顺序必须 1 对 1 相对应。
- 该表不能是具有 **rowid** 值的分片表。

您不能将 ADD TYPE 子句与任何更改表结构的子句相结合。在具有 ADD TYPE 子句的同一 ALTER TABLE 语句中，任何其它 ADD 、DROP 或 MODIFY 子句都是无效的。ADD TYPE 子句不允许您更改列数据类型。（要更改列的数据类型，请使用 MODIFY 子句。）

## 类型表上有效的选项

对于 ROW 数据类型的表 ALTER TABLE 只支持以下选项。

Typed-Table 选项



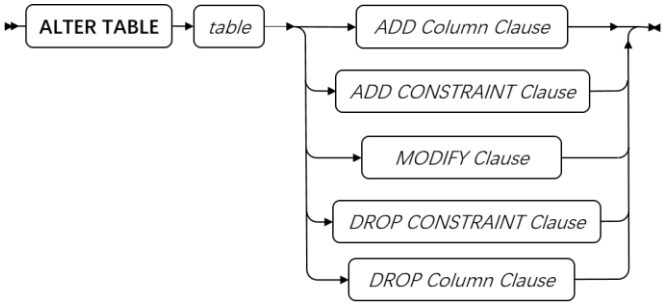

两个注意事项应用于为继承层次结构的一部分的类型表：

- 对于子表，ADD CONSTRAINT 和 DROP CONSTRAINT 对于继承的约束是无效的。

- 对于超表，ADD CONSTRAINT 和 DROP CONSTRAINT 传播到所有子表。

## 全局临时表上有效的选项

全局临时表的修改要在 oracle 模式下进行，对于全局临时表的 ALTER TABLE 支持以下选项。




使用 alter table 语句可以对表的列进行如下修改：

- 增加一列及该列上的列级约束；
- 修改一列的数据类型、精度、刻度，设置列上的 DEFAULT、NOT NULL、NULL；
- 删除一列；

使用 alter table 语句可以对全局临时表上的约束进行如下修改：

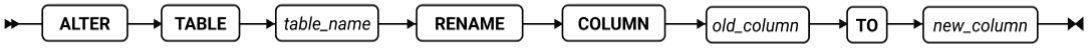
- 增加、删除表上的约束；

只有在没有会话绑定到全局临时表时，才能对全局临时表执行 ALTER TABLE 操作。通过发出 TRUNCATE 语句或终止会话来解除与全局临时表的绑定，或者对于事务临时表，通过发出 COMMIT 或 ROLLBACK 语句解除与全局临时表的绑定。

## Oracle 模式下 RENAME COLUMN 的用法

使用如下语句来更改列的名称。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *old_column* | 需要重命名的列 | 在表内必须存在 | 标识符 |
| *new_column* | 在此声明来代替 *old_column* 的名称 | 在该 *table* 中的列名称之中必须唯一 | 标识符 |
| *table_name* | 包含 *old_column* 的表 | 必须注册在当前数据库中 | 标识符 |

说明和限制：

该重命名列的方式仅在 GBase 8s 的 ORACLE 模式下支持。

使用该方式重命名一个列，必须具有 ALTER 该列所在表的权限。

新列名不能与表中原有列名同名；

- 列重命名后，视图、索引、同义词和约束仍然有效，触发器、存储过程的逻辑文本部分需要单独处理；

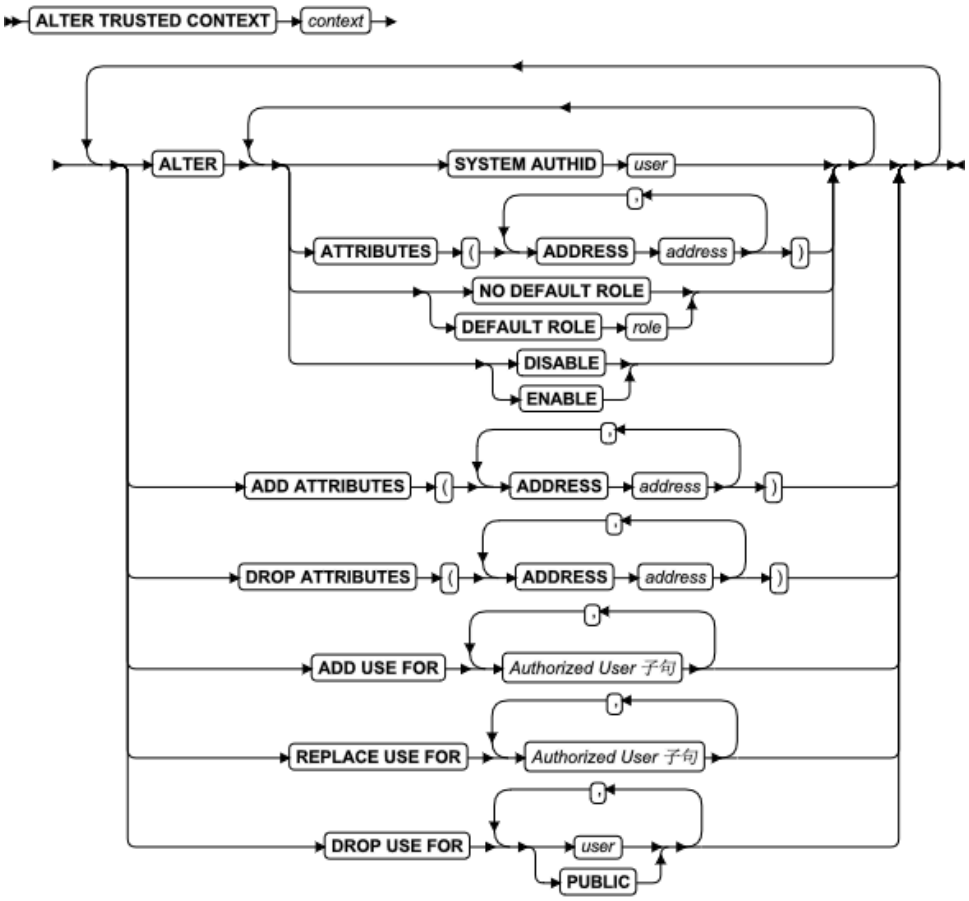
存在虚拟列的表，虚拟列及其引用的列不允许重命名列操作。

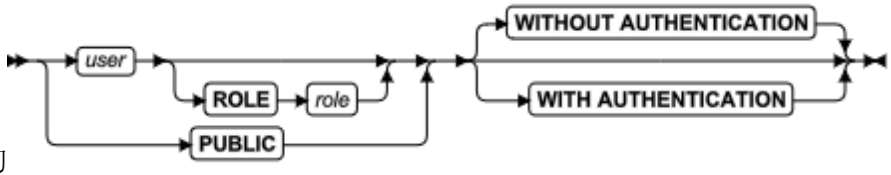
## 2.13 ALTER TRUSTED CONTEXT 语句

使用 ALTER TRUSTED CONTEXT 语句修改上下文受信任的对象的当前选项和属性（包括 ENABLED 或 DISABLED 方式）。

此语句是 SQL 语言 ANSI/ISO 标准的扩展。

**语法**






Authorized User 子句

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *address* | 客户机与数据库服务器之间的通信地址 | 对于此可信上下文对象的客户端通信地址必须唯一。有关其它的 *address* 限制，请参阅下文中的 **ADDRESS 属性** | 引用字符串 |
| *context* | 可信上下文对象的名称 | 必须存在于数据库服务器实例的可信上下文中且不能以字符 SYS 开头 | 标识符 |
| *role* | 现有的用户定义或内置角色 | 在数据库中必须存在，且在可信上下文对象的属性中必须唯一 | 所有者名称 |
| *user* | 用户的授权标识符 | 必须是有效的授权标识符。不能超过 32 字节。不能是发出此语句的用户的授权 ID 。 REPLACE USE FOR 子句不能指定该用户多次。 | 所有者名称 |

**用法**

您必须持有数据库安全管理员（DBSECADM）角色才能运行此语句。如果该语句被嵌入一个应用程序中，则权限是这些程序的所有者。如果此语句以可信上下文的角色运行，则该组权限是这些自由访问特权的结合：

- 该角色持有的权限组与主身份验证 ID 相关联，
- 此语句引用的每个角色都持有该权限组。

当 ALTER TRUSTED CONTEXT 语句执行成功后，该可信上下文对象的任何更改（它的属性、它的授权用户列表）会注册于 GBase 8s 数据库服务器实例的 sysuser 数据库的这些表中：

- systrustedcontext
- systcxattributes
- systcxusers

**ADDRESS 属性**

在定义可信上下文对象时，ALTER ATTRIBUTES 、ADD ATTRIBUTES 和 DROP ATTRIBUTES 选项可以指定一个或多个通信地址列表以连接数据库服务器，它们的状态可作为连接可信属性。下列限制适用于 ALTER TRUSTED CONTEXT 或 CREATE TRUSTED CONTEXT 语句引用的通信地址：

- 在该可信上下文对象的客户机通信地址中，每个地址必须唯一。

- 每个地址必须符合 TCP/IP 协议。
- 每个地址必须是 IPv4 地址、 IPv6 地址或安全域名称。
- IPv4 地址或 IPv6 地址必须是真实的主机地址（不是本地主机），且不能包含空格键。
- 此外，IPv6 地址不能是 IPv4 映射的 IPv6 地址。
- 安全域额名称不能是动态主机配置协议（DHCP）地址。

如果新的 *address* 值是安全域的名称，则该名称被域名服务器（决定产生的地址为 IPv4 或 IPv6 地址）转换为 IP 地址。当域名转换为 IP 地址时，该转换的结果可能是一个或多个 IP 地址。在这种情况下，如果从连接源发起的 IP 地址与域名转换的 IP 地址匹配，则数据库服务器解释传入的连接请求作为作为符合可信上下文对象的 ADDRESS 属性。

ALTER ATTRIBUTES 子句将指定的属性的现有值替换为新的值。如果属性不是当前可选上下文对象的一部分，则返回一个错误。未指定的属性仍保留先前的值。

可信上下文对象指定的 ADDRESS 值可通过 ALTER ATTRIBUTES 子句和 DROP ATTRIBUTES 子句删除。 ADDRESS 属性可被指定多次，但是每个 *address* 值在属性组中必须唯一。

ADD ATTRIBUTES 子句为该定义的可信上下文对象指定一个或多个可信属性列表。

DROP ATTRIBUTES 子句指定从可信上下文对象的定义中删除一个或多个属性。如果该属性不是当前可信上下文对象定义的一部分，则返回错误。

**注意：**

如果您现有的应用程序使用 ALTER TRUSTED CONTEXT 语句，且在 ATTRIBUTES 子句中包含 `ENCRYPTION` 或 `WITH ENCRYPTION` 选项，则数据库服务器不会发出 SQL 错误。然而，除此之外，对于 `WITH ENCRYPTION 'NONE'` 和 `ENCRYPTION 'NONE'` 关键字选项，ALTER TRUSTED CONTEXT 语句的加密选项在 GBase 8s 数据库服务器中不支持。

**DEFAULT ROLE 属性**

ALTER 子句的 `DEFAULT ROLE` ***role*** 选项标识在现有数据库服务器中已经存在的角色。该角色可以被不具有定义为可信上下文对象定义的一部分的用户指定的角色的用户使用。

`NO DEFAULT ROLE` 关键字指定可信上下文对象没有缺省角色。

如果此可信上下文的可信连接是活动的，则 DEFAULT ROLE 属性的变更在下一个请求新连接时或下一个更换用户请求时生效。

**ALTER 子句的 ENABLE 和 DISABLE 选项**

ENABLE 属性指示可信上下文对象处于启用状态。

DISABLE 属性指示可信上下文对象处于禁用状态，且对于新建立的可信连接禁用。

您不能使用 SET Database Object Mode 语句更改可信上下文的 ENABLE 或 DISABLE 属性。

**ADD USE FOR 子句**

ADD USE FOR 子句指定其它可建立基于此可信上下文对象的可信连接的用户。 PUBLIC 属性指示基于此可信上下文对象的可信连接可被任何用户使用。

PUBLIC 属性必须不能被指定为可信上下文对象的属性，且在 ADD USE FOR 子句中只能指定一次 PUBLIC 。如果可信上下文对象的定义允许被 PUBLIC 存取且还可被一个或多个用户存取，则用户规范重写此 PUBLIC 规范。

**REPLACE USE FOR 子句**

REPLACE USE FOR 子句指定更改使用此可信上下文对象的指定用户或 PUBLIC 组。当您使用 REPLACE USE FOR 子句 PUBLIC 时，该可信上下文对象必须已经被定义为允许 PUBLIC 使用，且 PUBLIC 在 REPLACE USE FOR 子句中只能指定一次。

REPLACE USE FOR 可以指定不同的角色名称，它必须是数据库服务器定义的角色。如果缺省角色与当前的可信上下文相关联，则可为用户明确指定角色以代替缺省的角色。

REPLACE USE FOR 子句还能更改当前的要求授权的用户或 PUBLIC 组。

**AUTHENTICATION 属性**

REPLACE USE FOR 和 ADD USE FOR 子句可以指定基于此可信上下文对象可信连接的授权要求。缺省的是 WITHOUT AUTHENTICATION 。

WITH AUTHENTICATION 属性指定将当前基于此可信上下文对象的连接的用户变更为该用户需要授权。

WITHOUT AUTHENTICATION 属性指定更改当前用户不需要授权。

**DROP USE FOR 子句**

DROP USE FOR 子句指定不能再使用该可信上下文对象的用户。这些从可信上下文的定义中删除的用户是当前允许使用该可信上下文对象的用户。如果一个或多个，不是全部用户可从可信上下文定义中删除，则指定的用户被删除且返回警告。如果没有指定的用户可以从可信上下文的定义中删除，则返回错误。

如果您对 DROP USE FOR 子句使用 PUBLIC，它移除所有用户（除了 SYSTEM AUTHID 用户 ID 和其他标识符已经显式启用的用户）使用此可信上下文对象的能力。

**修改可信上下文的示例**

在以下示例中，假设该可信上下文对象 appserver 存在并启用。以下的 ALTER TRUSTED CONTEXT 语句将 appserver 可信上下文对象的对象方式重置为 DISABLE。当其处于该方式时，appserver 可信上下文仍然存在，但是它不能用于存取数据库服务器。

```
ALTER TRUSTED CONTEXT appserver
        DISABLE;
```

下列示例中，假设该可信上下文对象 secure_role 存在。发出 ALTER TRUSTED CONTEXT 语句更改现有用户 joe 需要授权才能使用此可信上下文对象并给其它用户添加不用授权即可适应此可信上下文对象的权限。

```
ALTER TRUSTED CONTEXT securerole
        REPLACE USE FOR joe WITH AUTHENTICATION
        ADD USE FOR PUBLIC WITHOUT AUTHENTICATION;
```

以下示例修改了该可信上下文对象 securerole 使用 IPv4 地址，此地址与最初定义使用的地址不同。

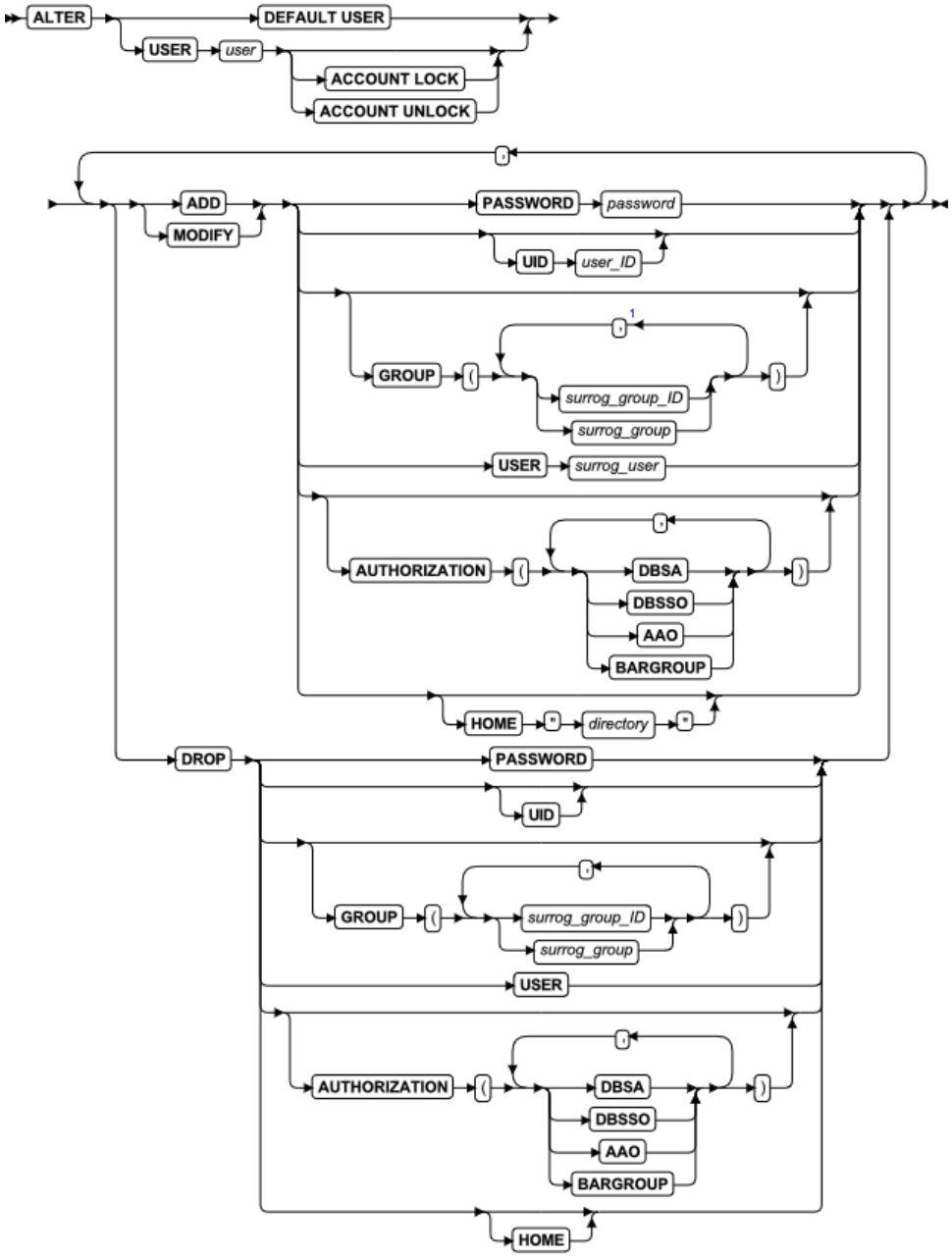
```
ALTER TRUSTED CONTEXT securerole
ALTER ATTRIBUTES (ADDRESS '9.12.155.200');
```

## 2.14 ALTER USER 语句 (UNIX™ 、Linux™)

使用 ALTER USER 语句更改用户的一个或多个属性，包括密码、用户 ID 、代理组、管理权限和主目录、启用或禁用内部已经授权用户的账户或缺省的内部已授权的用户。

该语句是 SQL 语言的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *directory* | 存储用户文件目录的路径名 | 必须少于或等于 255 字节，且必须符合您的操作系统的规范。*directory* 还必须：<br>• 属于映射的 *user_ID* 和 *surrog_group_ID*<br>• 所有者拥有读、写和执行权限 | 引用字符串 |
| *password* | 内部身份验证的用户的密码 | 必须在 6 到 32 字节 | 引用字符串 |
| *surrog_group* | 拥有您想要映射 *user* | 必须少于或等于 32 字节 | 所有者名称 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 的权限的现有操作系统组（代理组）的名称。*surrog_group* 值列表必须用括号括起。 | | |
| *surrog_group_ID* | 您要映射 *user* 的组标识符编号。*surrog_group_id* 值列表必须用括号括起。 | 该 *surrog_group_ID* 不能是：<br>• 具有服务器管理权限的组 ID（DBSA 、DBSSO 、AAO 和 BARGROUP）<br>• 组 0 (**root**，某些时候引用为 **wheel** 或 **system**）<br>• 与 **bin** 组或 **sys** 组相关联的 组 ID | 精确数值 |
| *surrog_user* | 在 GBase 8s 主键上吸纳有的 OS 用户(代理用户）账户名称，它拥有您要映射 *user* 的权限。 | 必须符合您的操作系统的规则 | 所有者名称 |
| *user* | 您要映射的属性的指定用户的身份验证标识符 | 必须是经过身份验证的身份验证标识符 | 所有者名称 |
| *user_ID* | 您要映射的 *user* 的用户标识符编号 | *user_ID* 不能是用户 root 或用户 gbasedbt | 精确数值 |

**用法**

只有 DBSA 才能运行 ALTER USER 语句。在非 root 安装中，安装服务器的用户等同于 DBSA ，除非该用户将 DBSA 权限委托给另一个用户。

必须在 CREATE USER 语句创建用户之前将 USERMAPPING 配置参数值设置为一个启用支持映射用户的值（ADMIN 或 BASIC），如此才能连接该数据库服务器。

必须将 USERMAPPING 配置参数设置为 ADMIN 才能启用 AUTHORIZATION 子句。有关此不推荐使用语法的更多信息，请参阅 AUTHORIZATION 子句中 CREATE USER 语句（UNIX、Linux）的描述。

您还必须在 **sysusers** 数据库的 SYSUSERMAP 表中输入值以用合适的属性映射用户，以致于该映射用户的 SQL 语句可以正确工作。

如果用户使用可插入式身份验证 （PAM）或单点登录（SSO）对用户进行身份验证，则用户可以使用代理用户属性连接 GBase 8s 。

如果用户使用可插入身份验证模块对用户进行身份验证，则映射用户可以使用代理用户属性连接 GBase 8s 。

最好的练习是将 ***user*** 映射到指定 ***surrog_user***，这样作为代理用户身份被保留。您可以使用 GROUP 关键字添加与代理用户身份相关联的组，使用 HMOE 关键字更改主目录，

ALTER USER 语句不会影响任何活动的具有相同的代理用户或用户 ID 的操作。只会影响需要身份验证的子操作。

如果用户没有密码，ALTER USER 语句可以使用 ADD 关键字为用户添加密码。要更改现有的密码，请在 ALTER USER 语句中使用 MODIFY 选项。

ALTER USER 操作之后的组的总数不能超过 16 个，是允许的组的最大数量。

如果主目录不存在，ALTER USER 语句仅能使用 ADD 关键字添加主目录。要更改现有主目录，请使用 MODIFY 关键字。

在一个单独的 ALTER USER 语句中，属性只能指定一次。例如，您不能在同一语句中删除 GROUP 属性又添加 GROUP 属性。

ALTER USER 语句之后，该用户必须拥有 USER 属性或 UID 属性。

ALTER USER 语句的执行可以用 ALUR 审计代码审计。

**示例**

Example 1: 用 UID 属性替换 USER 属性

以下示例将 **bill** 用户的 USER 属性替换为 UID 属性：

```
ALTER USER bill DROP USER, ADD UID 1360;
```

Example 2: 修改和添加属性

下列语句修改了用户 **bill** 的 UID 属性，将其添加到 DBSA 组，并添加主目录：

```
ALTER USER bill MODIFY UID 1361, ADD GROUP (dbsa), ADD HOME "/u/user1";
```

Example 3: 解锁账户并删除验证属性

以下语句解锁用户 **bill** 的账户并删除其 DBSSO 身份验证：

```
ALTER USER bill ACCOUNT UNLOCK DROP AUTHORIZATION (dbsso);
```

Example 4: 删除主目录

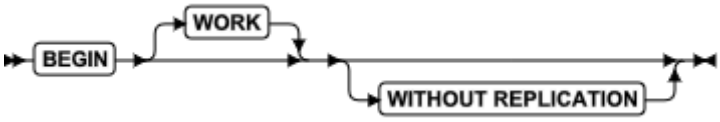
以下语句删除用户 **bill** 的主目录：

```
ALTER USER bill DROP HOME;
```

## 2.15 BEGIN WORK 语句

使用 BEGIN WORK 语句来启动*事务*（COMMIT WORK 或 ROLLBACK WORK 语句终止的一系列数据库操作，数据库服务器将其作为单个工作单元）。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




用法

BEGIN WORK 语句只在支持事务日志记录的数据库中有效。该语句在 ANSI 兼容的数据库中无效。

在事务锁定期间 UPDATE 、DELETE 、INSERT 或 MERGE 语句影响的每行将在整个事务中保持锁定。包含许多这样的语句的事务或包含影响许多行的语句的事务，可超过您的操作系统或数据库服务器对同步锁定数目施加的限制。

如果没有其他用户访问该表，您可以在开始事务后通过使用 LOCK TABLE 语句来避免锁定限制并减少锁定开销。如同其它锁定一样，此表锁定在事务终止时释放。在 BEGIN WORK 的示例 的示例上的事务包含一条 LOCK TABLE 语句。

**重要：** 仅当事务不在进行时发出 BEGIN WORK 语句。如果当在事务中时发出一个 BEGIN WORK 语句，数据库服务器返回一条错误。

WORK 关键字是可选的。以下两条语句等价：

```
BEGIN;
BEGIN WORK;
```

在读取 SQL 源代码时忽略 WORK 关键字，不要混淆 SQL 的 BEGIN 语句和 SPL 关键字 BEGIN ，它们和 END 关键字一起可用于 SPL 例程内定义语句块的分隔符。

在 GBase 8s ESQL/C 中，如果在 WHENEVER 语句调用的 UDR 中使用 BEGIN WORK 语句，则应在 ROLLBACK WORK 语句之前指定 WHENEVER SQLERROR CONTINUE 和 WHENEVER SQLWARNING CONTINUE 。如果 ROLLBACK WORK 语句碰到错误或者警告，则这些语句可防止程序发生无限循环。

## BEGIN WORK 和兼容 ANSI 的数据库

在兼容 ANSI 的数据库中，不需要 BEGIN WORK 语句，因为事务是隐式的；每个 SQL 语句均在事务内部发生。当您在以下任一语句之后立即使用 BEGIN WORK 语句时，数据库服务器生成一条警告：

- DATABASE
- COMMIT WORK
- CREATE DATABASE
- ROLLBACK WORK

在兼容 ANSI 的数据库中，如果您在任何其它语句之后使用 BEGIN WORK 语句，数据库服务器会返回一条错误。

## BEGIN WORK WITHOUT REPLICATION (ESQL/C)

当您使用 Enterprise Replication 进行数据复制时，可使用 BEGIN WORK WITHOUT REPLICATION 语句来启动不复制到其它数据库服务器的事务。

您无法将 BEGIN WORK WITHOUT REPLICATION 作为 GBase 8s ESQL/C 应用程序中的独立植入语句执行。而只能间接执行此语句。可以使用以下两种方法之一：

- 可以使用 PREPARE 和 EXECUTE 语句的组合来准备并执行 BEGIN WORK WITHOUT REPLICATION 语句。
- 可以使用 EXECUTE IMMEDIATE 语句，只用单独一个步骤就准备并执行 BEGIN WORK WITHOUT REPLICATION 。

不能将 DECLARE ***游标*** CURSOR WITH HOLD 与 BEGIN WORK WITHOUT REPLICATION 语句一起使用。

有关数据复制的更多信息，请参阅 *GBase 8s Enterprise Replication 指南*。

## BEGIN WORK 的示例

当连续的 SQL 语句执行的是逻辑上的一个工作单元，您可以通过在 BEGIN WORK 和 COMMIT WORK 语句之间给它们分组来定义此事务。 如果此任务需求要求要么所有语句都执行成功，要么它们都没被执行，则您可以在启动事务 BEGIN 和成功完成此事务 COMMIT WORK （或者 ROLLBACK WORK ，如果程序检测到错误，则取消该事务）之间包含该事务的语句。

以下代码分段中，该事务锁定了 **stock** 表（LOCK TABLE），更改了 **stock** 表中的行（UPDATE），从 **stock** 表删除行（DELETE）并将行插入 **manufact** 表（INSERT）。在此示例中（没有错误处理），数据库服务器按顺序执行这些 SQL 语句：

```
BEGIN WORK;
        LOCK TABLE stock;
        UPDATE stock SET unit_price = unit_price * 1.10
        WHERE manu_code = 'KAR';
        DELETE FROM stock WHERE description = 'baseball bat';
        INSERT INTO manufact (manu_code, manu_name, lead_time)
        VALUES ('LYM', 'LYMAN', 14);
        COMMIT WORK;
```

每个语句本身就是原子；它成功完成或者数据库从未更改。如果任一语句失败，其它语句仍会继续执行，其结果好像是失败语句从未尝试执行。当 COMMIT WORK 语句执行后，这些成功的变更将变成永久性的。

然而，一般情况下，事务被定义为带有错误处理，因此数据库服务器必须完整执行这一系列操作，或者完全不执行。在这种情况下，当您在单个事务中包含所有这些操作时，数据库服务器保证所有的语句完整地一个不漏地提交到磁盘，或者恢复到与事务开始之前完全相同的状态。

通过提交错误处理属性（例如，在 DB-Access 中设置 **DBACCNOIGN** 环境变量或者在 ESQL/C 中添加 EXEC SQL WHENEVER ERROR STOP），该事务可隐式地回滚，因为此程序由于一个错误而

停止且没有执行 COMMIT WORK。更多细致的条件编码例如（ESQL/C）允许程序员在继续执行更大的程序时显式回滚该事务。

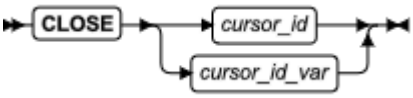
应用程序和 UDRs 中的错误处理和业务逻辑还可通过包含 SAVEPOINT 和 ROLLBACK TO SAVEPOINT 语句将事务分隔为一个或多个分区。如果在遇到错误后，或在事务的部分结构指示与业务规范或其它标准冲突之后发出 ROLLBACK TO SAVEPOINT 语句，只有在 ROLLBACK 语句和其指定或缺省的保存点之间的数据库的更改会被取消，而不是取消整个事务。ROLLBACK 之后当前事务继续该语句，包括数据未提交的变更或数据库超过保存点仍暂挂的操作的结构都将继续执行。直到整个事务被提交或回滚。任何已回滚语句持有的锁定将会保留直到此事务完全结束。

## 2.16 CLOSE 语句

当您不再需要引用 Select 或 Function 游标检索的行时，请使用 CLOSE 子句关闭游标。

在 ESQL/C 中，该语句还可以刷新并关闭 Insert 游标。可在 GBase 8s ESQL/C 或 SPL 中使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 要关闭的游标的名称 | 必须已声明 | 标识符 |
| *cursor_id_var* | 包含 *cursor_id* 的值的主变量 | 必须是字符数据类型 | 必须符合特定于语言的名称规则 |

**用法**

关闭游标使得游标对于除 OPEN 或 FREE （或 OPEN FOR）之外的任何语句无用，并释放数据库服务器已经分配到游标的资源。

在不兼容 ANSI 的数据库中，您可以关闭尚未打开的游标或已经关闭的游标。在这些情况下没有采取任何操作。

在兼容 ANSI 的数据库中，如果您关闭尚未打开的游标，那么数据库服务器返回错误。

**示例**

以下示例关闭了游标 *democursor* 。

```
EXEC SQL close democursor;
```

以下是来自 demo1.ec 的 ESQL/C Source 代码示例：

```
#include <stdio.h>

EXEC SQL define FNAME_LEN          15;
```

```
EXEC SQL define LNAME_LEN           15;

main()
{

EXEC SQL BEGIN DECLARE SECTION;
      char fname[ FNAME_LEN + 1 ];
      char lname[ LNAME_LEN + 1 ];
EXEC SQL END DECLARE SECTION;

      printf( "DEMO1 Sample ESQL Program running.\n\n");

      EXEC SQL WHENEVER ERROR STOP;

      EXEC SQL connect to 'stores7';

      EXEC SQL declare democursor cursor for
            select fname, lname
                into :fname, :lname
                from customer
                where lname < "C";

      EXEC SQL open democursor;
      for (;;)
            {
            EXEC SQL fetch democursor;
            if (strncmp(SQLSTATE, "00", 2) != 0)
                  break;
            printf("%s %s\n",fname, lname);
            }
      if (strncmp(SQLSTATE, "02", 2) != 0)
            printf("SQLSTATE after fetch is %s\n", SQLSTATE);

      EXEC SQL close democursor;
      EXEC SQL free democursor;
      EXEC SQL create routine from 'del_ord.sql';
      EXEC SQL disconnect current;
      printf("\nDEMO1 Sample Program over.\n\n");
    exit(0);
}
```

## 关闭 Select 或 Function 游标

当游标与 SQL 的 SELECT 、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句相关时，关闭游标将终止此相关联的 SQL 语句。

数据库服务器释放它可能已经分配到活动行集的所有资源。例如，它用来保存有序集的临时表。数据库服务器还释放它在在通过游标选择的行上可能持有的任何锁定。然而，如果事务包含 CLOSE 语句，则在您执行 COMMIT WORK 或 ROLLBACK WORK 语句之前数据库服务器不释放锁定。

关闭 Select 游标或 Function 游标之后，您无法执行游标的 FETCH 语句，直到您重新打开它。

在 SPL 例程中，内置的 **SQLCODE** 函数可以显示 Select 游标或 Function 游标的 CLOSE 语句的结果。该函数返回的值相当于 **sqlca** 结构的 **SQLCODE** 字段。然而，如果您在调用 SPL 例程的上下文之外调用内置的 **SQLCODE** 函数，则 GBase 8s 发出错误。

## 关闭 Insert 游标

由于 GBase 8s 在 SPL 例程中不支持 Insert 游标，本节有关 Insert 游标的讨论仅适用于 GBase 8s ESQL/C。在 SPL 例程中，只能执行 DECLARE 语句定义的 Select 或 Function 游标的 CLOSE 语句。（SPL 的 FOREACH 语句在其语句块中包含 INSERT 语句可以声明功能类似 Insert 游标的 ***direct cursor*** ，但是不能执行 FOREACH 声明的直接定位游标的 CLOSE 语句。 GBase 8s 在程序控制从定义直接定位游标的 FOREACH 循环退出时，会自动关闭该直接定位游标。）

在 GBase 8s ESQL/C 中，CLOSE 语句对待与 INSERT 语句关联的游标和与 SELECT 、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句关联的游标不同。当游标标识与 INSERT 语句关联时，CLOSE 语句将任何剩下的已缓冲行写入数据库。在 **sqlca** 结构中 **sqlerrd** 数组的第三个元素 **sqlca.sqlerrd[2]** 中返回成功插入数据库的行数。有关如何使用 **SQLERRD** 对插入的总行数计数的信息，请参阅 错误检查 。

**sqlca** 结构的 **SQLCODE** 字段，指示了 Insert 游标 CLOSE 语句的结果。如果所有已缓冲的行成功插入，则 **SQLCODE** 被置零。如果遇到错误，则 **SQLCODE** 字段被设为负的错误消息数。

当 SQLCODE 为零时，释放行缓冲区空间，且关闭游标；也就是，您无法执行指定游标的 PUT 或 FLUSH 语句，直到您重新打开它。

**提示：** 如果遇到 sqlca.SQLCODE 错误，则还会存在对应的 SQLSTATE 错误值。关于如果获得消息文本的信息，请检查 GET DIAGNOSTICS 语句。

如果插入不成功，则成功插入的行数存储在 **sqlerrd** 中。在最后成功插入的行之后的任何已缓冲行被废弃。由于插入失败，CLOSE 语句也失败，并且游标没有关闭。例如，如果磁盘空间不足而使得某些行无法插入，则 CLOSE 语句会失败。在这种情况中，第二个 CLOSE 语句可能成功，因为不存在已缓冲的行。OPEN 语句也会成功，因为 OPEN 语句执行了一个隐式关闭。

## 关闭集合游标

可以在集合变量上同时声明 Select 和 Insert 游标。这样的游标被称为集合游标。使用 CLOSE 语句来收回已分配给集合游标的资源。SPL 例程中的 CLOSE 语句不能引用 SPL 声明的 FOREACH 语句的直接定位集合游标。

有关如何使用集合游标的更多信息，请参阅 从集合游标访存 和 插入到 Collection 游标内 。

### 使用事务结束来关闭游标

COMMIT WORK 和 ROLLBACK WORK 语句关闭所有的游标（除了那些声明为保留的）。不过，最好显示关闭所有游标。对于 Select 或 Function 游标，此操作仅使得程序意图明显。如果随后向游标声明添加 WITH HOLD 子句，则也有助于避免逻辑错误。

对于 ESQL/C 例程中的 Insert 游标，显式使用 CLOSE 语句以便可以测试错误代码，这一点很重要。在 COMMIT WORK 语句之后，**SQLCODE** 反映 COMMIT 语句的结果，而不是正在关闭的游标的结果。如果使用 COMMIT WORK 语句而没有首先使用 CLOSE 语句，并且如果最后一个已缓冲的行写入数据库时出现错误，则事务仍被提交。

有关如何使用 Insert 游标和 WITH HOLD 子句的信息，请参阅 DECLARE 语句 。

在 ANSI 兼容的数据库中，游标无法隐式关闭。您必须发出 CLOSE 语句。

## 2.17 CLOSE DATABASE 语句

使用 CLOSE DATABASE 语句关闭当前数据库的隐式连接。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


**用法**

当发出 CLOSE DATABASE 语句时，紧接其后您仅能发出以下 SQL 语句：

- CONNECT
- CREATE DATABASE
- DATABASE
- DROP DATABASE
- DISCONNECT

  （这里，只有在执行 CLOSE DATABASE 之前存在显式连接的情况下，DISCONNECT 语句才有效。）

在删除当前数据库之前发出 CLOSE DATABASE 语句。

如果当前数据库支持事务日志记录，并且如果已启动了事务，则必须发出 COMMIT WORK 或 ROLLBACK WORK 语句，然后才能使用 CLOSE DATABASE 语句。

以下示例显示了如何使用 CLOSE DATABASE 语句来删除当前数据库，它的会话已建立一个隐式连接：

```
DATABASE stores_demo;
. . .
CLOSE DATABASE;
DROP DATABASE stores_demo;
```

在 GBase 8s ESQL/C 中，CLOSE DATABASE 语句不能出现在多语句的 PREPARE 操作中。

如果先前的 CONNECT 语句已经和数据库建立了显式连接，而且该连接仍然是当前连接，那么就不能使用 CLOSE DATABASE 语句关闭该显式连接。（可以使用 DISCONNECT 语句关闭该显式连接。）

如果在 WHENEVER 语句调用的 UDR 中使用 CLOSE DATABASE 语句，则应在 ROLLBACK WORK 语句之前指定 WHENEVER SQLERROR CONTINUE 和 WHENEVER SQLWARNING CONTINUE 。如果 ROLLBACK WORK 语句遇到错误或警告，则此操作可防止程序循环。

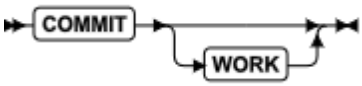
当发出 CLOSE DATABASE 语句时，任何已声明的游标不再有效。您必须重新声明任何想要使用的游标。

在 ANSI 兼容的数据库中，如果在没有发出 CLOSE DATABASE 、COMMIT WORK 或 DISCONNECT 语句的情况下，以非交互式方式从 DB-Access 退出时没有遇到错误，则数据库服务器自动提交任何打开的事务。

## 2.18 COMMIT WORK 语句

使用 COMMIT WORK 语句提交从事务开始时对数据库所作的全部修改。

**语法**




**用法**

COMMIT WORK 语句通知数据库服务器您到达了必须作为单个单元完成的一系列语句的末尾。数据库服务器采取必需的步骤来确保事务做出的所有修改正确完成且保存到磁盘。

当确定希望保留所有从事务开始起对其数据库所做的所有改变时，仅在带有事务日志记录的数据库的多语句操作结束时使用 COMMIT WORK。

COMMIT WORK 语句释放所有行锁定和表锁定。

WORK 关键字在 COMMIT WORK 语句中是可选的。以下两条语句等价：

```
COMMIT;
COMMIT WORK;
```

以下示例显示了 BEGIN WORK 和 COMMIT WORK 语句所限制的事务。

```
BEGIN WORK;
DELETE FROM call_type WHERE call_code = 'O';
INSERT INTO call_type VALUES ('S', 'order status');
COMMIT WORK;
```

在此示例中，用户首先从 **call_type** 表中删除行，其中表的 **call_code** 列的值为 O 。用户然后在其中 **call_code** 列的值为 S 的 **call_type** 表中插入一个新的行。数据库服务器保证两个操作同时成功或不成功。

在 GBase 8s ESQL/C 中，COMMIT WORK 语句关闭所有打开的游标（除了那些使用 WITH HOLD 选项声明的游标。）

### 在不兼容 ANSI 的数据库中发出 COMMIT WORK

在不兼容 ANSI 但支持事务日志记录的的数据库中，如果您用 BEGIN WORK 语句启动事务，就必须在事务结束时发出 COMMIT WORK 语句。在这种情况下，如果未能发出一个 COMMIT WORK 语句，则数据库回滚事务对数据库所做出的任何修改。

然而，如果没有发出 BEGIN WORK 语句，每个语句会在自己的事务中执行。这些单语句事务不需要 BEGIN WORK 语句或 COMMIT WORK 语句。

### 显式的 DB-Access 事务

当在不兼容 ANSI 但是支持事务日志记录的数据库中以交互方式使用 DB-Access 时，如果选择 Commit 菜单但是在 BEGIN WORK 语句启动事务以后没有发出 COMMIT WORK 语句，则 DB-Access 会自动地提交数据，但是会发出以下警告：

Warning: Data commit is a result of unhandled exception in TXN PROC/FUNC

此警告是为了提醒您显式地发出 COMMIT WORK 以结束 BEGIN WORK 启动的事务。

然而，在非交互方式中，如果没有发出 COMMIT WORK 语句就结束会话，则 DB-Access 将会回滚当前的事务。

### 在兼容 ANSI 的数据库中发出 COMMIT WORK

在兼容 ANSI 的数据库中，您不需要 BEGIN WORK 来标记事务的开始。您仅需要标记每个事务的结束，因为事务总是有效的。新的事务在每个 COMMIT WORK 或 ROLLBACK WORK 语句后自动启动。

然而，您必须发出显式 COMMIT WORK 语句来标记每个事务的结束。如果无法这样做，数据库服务器回滚事务对数据库所做的任何修改。

然而在兼容 ANSI 的数据库中，如果在不发出 CLOSE DATABASE 、COMMIT WORK 或 DISCONNECT 语句的情况下，便在非交互式方式下退出 DB-Access 而没有遇到错误，则数据库服务器自动提交任何打开的事务。

## 2.19 COMMENT 语句

使用 COMMENT 语句对表、视图或列添加注释。

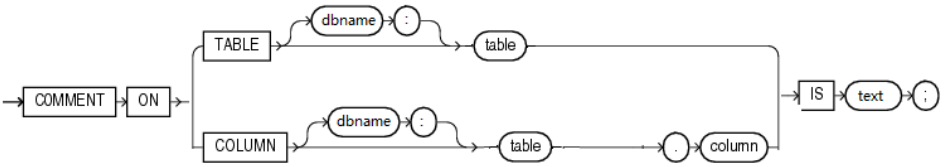
**语法**

该语句的语法格式如下所示：

```
COMMENT ON
  { TABLE [ dbname:]{ table }
```

```
    | COLUMN [ dbname:]
      { table. } column
     }
  IS 'text' ;
```

语法图为：




**用法**

要向表、视图或列添加注释，您必须具有 COMMENT ANY TABLE 权限，或有对该数据库的 DBA 权限。

您可以通过查询视图 SYSCOMMENTS、SYSCOLCOMMENTS 来查看特定表、视图或列上的注释。

要从数据库中删除注释，请将其设置为空字符串‘’。

**TABLE 子句**

指定要添加注释的表或视图的模式和名称，包括所有用户表，不包括系统表。若连接时指定数据库名，则数据库名可省略。

**重要：**当您想要向视图添加注释时，也请使用关键字 TABLE ，后跟视图名称，即可向此视图添加注释。

例如，以下示例向视图 View1 添加注释 ’视图 1’：

```
COMMENT ON TABLE View1 IS '视图 1';
```

**COLUMN 子句**

指定要注释的表或视图的列的名称。若连接时指定数据库名，则数据库名可省略。

## IS 'text'

指定注释的文本。

## COMMENT ON 的示例

以下示例给 EMPLOYEE 表添加注释：

```
COMMENT ON TABLE employee IS '员工表';
```

要查看此表的注释，请执行以下语句：

```
select * from  syscomments  where  TABNAME='EMPLOYEE';
```

查询结果为:

```
TANAME                    COMMENTS
------------------------------ ------------
EMPLOYEE                   员工表
```

以下示例给 employee 表的 salary 列添加注释：

```
COMMENT ON COLUMN employees.salary IS '薪水';
```

要查看此表的列的注释，请执行以下语句：

```
select * from syscolcomments where TABNAME='EMPLOYEE';
```

查询结果为：

```
TABNAME        COLNAME         COMMENTS
---------------      ---------------         -------------
EMPLOYEE       EMP_ID
EMPLOYEE       NAME
EMPLOYEE       ADDRESS
EMPLOYEE       SALARY            薪水
```

要从数据库删除此列的注释，请执行以下语句：

```
COMMENT ON COLUMN employee.salary IS ' ';
```

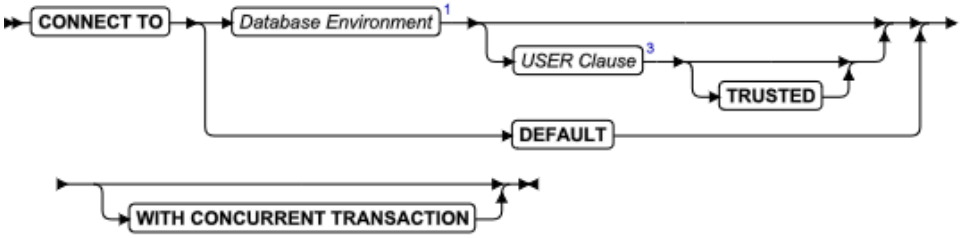
以下示例授予用户注释权限：

```
GRANT COMMENT ANY TABLE TO user_name;
```

## 2.20 CONNECT 语句

使用 CONNECT 语句连接数据库环境。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




**用法**

CONNECT 语句将应用程序连接到*数据库环境*，数据库环境可以是数据库、数据库服务器或数据库连同数据库服务器。如果应用程序成功连接到指定的数据库环境，则连接成为应用程序的*当前连接*。如果应用程序没有到数据库服务器的当前连接，则 SQL 语句失败。如果指定数据库名称，则数据库服务器打开该数据库，您不能在 PREPARE 语句中包含 CONNECT 。

应用程序可同时连接到数个数据库环境，并且它可以建立到同一个数据库环境的多个连接，条件是每个连接有唯一的连接名称。

在 UNIX™ 上，建立到同一个数据库环境的多个连接的唯一限制是，一个应用程序到每个使用共享内存连接机制的本地服务器只能建立一个连接。要找出本地服务器是使用共享内存连接机制，还是使用本地回送连接机制，请检查 $GBASEDBTDIR/etc/sqlhosts 文件。有关 sqlhosts 文件的更多信息，请参阅 *GBase 8s 管理员指南*。

在 Windows™ 上，本地连接机制是命名管道。从一个客户机到本地服务器可以存在多个连接。

任何时候都只有一个连接处于当前状态；其它连接均处于休眠状态。应用程序无法通过休眠的连接来与数据库进行交互。当应用程序建立新连接时，该连接就成为当前连接，而上一个当前连接变成休眠连接。可以使用 SET CONNECTION 语句使休眠的连接成为当前连接。另见 SET CONNECTION 语句 。

对于不同 GBase 8s 实例的数据库之间的连接，您不能使用不同的服务器别名在相同的两台数据库服务器之间建立多个活动的连接。如果使用 CONNECT TO *dbserveralias* 语句指定不同服务器别名来连接同一个远程服务器（*dbserveralias* 标识在 DBSERVERALIASES 配置参数的设置中声明），则不会发出错误，但是初始连接是重复使用的。

## 执行 CONNECT 语句的权限

当前用户，或者 PUBLIC ，必须在 CONNECT 语句指定的数据库上拥有 Connect 权限。执行 CONNECT 语句的用户不能和数据库中现有的角色拥有相同的用户名。

有关当 CONNECT 语句连接到远程主机上的数据库服务器时如何使用 USER Authentication 子句指定备用用户名的信息，请参阅 USER Authentication 子句 。

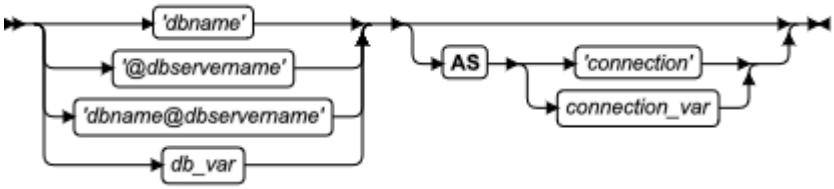
## 连接上下文

每个连接包含一组称为 ***连接上下文***的信息。连接上下文包含当前用户的名称。数据库环境与此名称相关联的信息以及连接状态的信息（例如活动的是否与连接相关联）。当应用程序进入休眠状态时保存连接上下文，而当应用程序再次成为当前应用程序时恢复此上下文。（有关更多信息，请参阅 使休眠连接成为当前的连接。）

## 数据库环境

此 CONNECT 以及类似于 SET CONNECTION 语句，可使用数据库环境语法段指定应用程序尝试建立连接的数据库或数据库服务器。不同于 SET CONNECTION 语句，该 CONNECT 语句还可以为此指定的数据库环境的连接声明名称。

数据库环境




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *connection* | 此处声明的连接的名称，该名称可选的并区分大小写 | 在连接名称中必须唯一 | 标识符 |
| *connection_var* | 存储 *connection* 名称的主变量 | 必须为固定长度的字符数据类型 | Language specific |
| *db_var* | 包含有效数据库环境（以语法图中的格式）的主变量 | 必须为固定长度的字符数据类型，其内容以语法图中的格式显示 | Language specific |
| *dbname* | 要连接的数据库 | 必须已经存在 | 标识符 |
| *dbservername* | 进行连接的数据库服务器的名称 | 必须已经存在；@ 符号和 *dbservername* 之间的空格无效。另见对 dbservername 的限制 。 | 标识符 |

如果设置了 **DELIMIDENT** 环境变量，数据库环境中的任何引号（ ' ）必须是单引号。如果没有设置 **DELIMIDENT** ，那么单引号（ ' ）或者双引号（ " ）在此都有效。

## 对 dbservername 的限制

如果指定 *dbservername* ，则它必须服满足以下限制。

- 如果您指定的数据库服务器没有联机，则您会接收到一条错误。
- 您在 ***dbservername*** 中指定的数据库服务器必须匹配 **sqlhosts** 文件中的数据库服务器的名称。

注： 如果数据库服务器的名称是一个分隔标识符或它包含大写字母，则数据库服务器不能参与跨数据库分布的 DML 操作。（如果该服务器包含大写字母，它还不能参与由 SQL 语句指定的服务器名称作为到数据库名称的限定符的跨数据库分布的 DML 操作。这些语句发生 -908 错误并失败，因为该 SQL 语法分析器将服务器名称中的所有的大写字母降档为小写字母。）要避免此限制，当声明要参与分布查询的数据库服务器的别名或名称使，仅指定没有大写字母的未分隔的名称。

## 指定数据库环境

可以指定数据库服务器和数据库，或仅指定数据库服务器，或仅指定数据库。数据库如何定位和打开取决于您是否在数据库环境表达式中指定数据库服务器名称。

### *仅指定数据库服务器*

***@dbservername*** 选项仅建立一个到数据库服务器的连接；它不打开数据库。使用此选项时，必须随后使用 DATABASE 或 CREATE DATABASE 语句(或它们的 PREPARE 语句以及 EXECUTE 语句）来打开数据库。

### *指定数据库服务器和数据库*

如果同时指定数据库服务器和数据库，则应用程序连接到数据库服务器，该数据库服务器找到并打开数据库。

### *仅指定数据库*

***dbname*** 选项建立到缺省数据库服务器或 **DBPATH** 环境变量中的另一个数据库服务器的连接。它还找到并打开指定的数据库。（如果这仅指定数据库名称，则 ***db_var*** 选项也是如此。）

如果仅指定 ***dbname*** ，则其数据库服务器从 **DBPATH** 环境变量读取。在 **GBASEDBTSERVER** 环境变量中的数据库服务器总是在 **DBPATH** 值之前。

在 UNIX™ 上，如以下示例所示设置 GBASEDBTSERVER 和 DBPATH 环境变量（对于 C shell ）：

```
setenv GBASEDBTSERVER srvA
setenv DBPATH //srvB://srvC
```

在 Windows™ 上，从任务栏选择**开始** > **程序** > GBase 8s > setnet32 并设置 GBASEDBTSERVER 和 DBPATH 环境变量：

```
set GBASEDBTSERVER = srvA
set DBPATH = //srvA://srvB://srvC
```

下一个示例显示应用程序使用的生成的 DBPATH ：

```
//srvA://srvB://srvC
```

应用程序首先建立到 **GBASEDBTSERVER** 指定的数据库服务器的连接。数据库服务器使用配置文件中的参数来找到数据库。如果数据库不驻留在缺省数据库服务器上，或缺省数据库服务器脱机，则应用程序连接到 **DBPATH** 中的下一个数据库服务器。在先前的示例中，该数据库服务器是 **srvB** 。

## 声明连接名称

在 ESQL/C 应用程序中，可以通过包含 AS 关键字为此数据库环境的连接声明标识符，其跟随在引号字符或存储标识符的主变量之后。该主变量必须是固定长度的字符数据类型。

## 连接标识

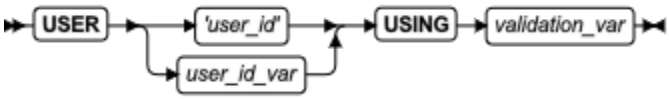
可选的 ***connection*** 名称是应用程序可以用来引用随后的 SET CONNECTION 和 DISCONNECT 语句中的连接的唯一标识。如果应用程序不提供连接名称（或连接主机变量），则它可以引用使用数据库服务器的连接。然而，如果应用程序对同一个数据库环境进行了多个连接，则每个连接必须拥有唯一的名称。

只有 CONNECT 语句可以使用 AS 关键字声明连接名称。然而，CONNECT 语句不能引用之前声明的连接名称以指定数据库环境的连接。

## USER Authentication 子句

USER Authentication 子句指定用来确定应用程序是否可以访问远程主机上目标计算机的信息。

USER Authentication Clause




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *user_id* | 有效的登录名 | 请参阅 对用户标识参数的限制 | 引用字符串 |
| *user_id_var* | 包含 *user_id* 的主变量 | 必须为固定长度字符数据类型；与 *user_id* 相同的限制 | 特定于语言 |
| *validation_var* | 包含 *user_id* 或 *user_id_var* 中登录名称的有效密码的主变量 | 必须为固定长度字符数据类型。请参阅对验证参数变量的限制 | 特定于语言 |

当 CONNECT 语句连接到远程主机上的数据库服务器时需要 USER Authentication 子句。CONNECT 语句后，远程主机上的所有数据库操作使用指定的用户名。

在 DB-Access 中，USING 子句在从 DB-Access 执行的文件中是有效的。在交互方式中，DB-Access 提示输入密码，所以就不使用 USING 关键字和 ***validation_var*** 。

## 对验证参数变量的限制

在 UNIX™ 上，***validation_var*** 中存储的密码必须是有效的密码，并且必须存在于 **/etc/passwd** 文件中。如果应用程序连接到远程数据库服务器，则密码必须同时存在于本地远程数据库服务器的此文件中。

在 Windows™ 上，***validation_var*** 中存储的密码必须是有效的密码，并且必须是**用户管理器**中输入的密码。如果应用程序连接到远程数据库服务器，则密码必须同时存在于客户机和服务器的域中。

## 对用户标识参数的限制

如果发生以下任意一种情况，则连接会被拒绝：

- 指定的用户缺少访问数据库环境中指定数据库的权限。
- 指定的用户缺少连接到远程主机所必需的许可权。
- 您提供了 USER Authentication 子句但是遗漏了 USING ***validation_var*** 指定。

为了与 CONNECT 语句的 X/Open 标准一致，GBase 8s ESQL/C 预处理器允许具有 USER Authentication 子句的 CONNECT 语句不带 USING ***validation_var*** 指定。然而如果没有提供 ***validation_var*** ，则数据库服务器在运行时拒绝连接。

在 UNIX™ 上，您指定的 ***user_id*** 必须是有效的登录名且必须存在于 **/etc/passwd** 文件中。如果该应用程序连接到远程服务器，则登录名必须同时存在于本地和远程数据库服务器的此文件中。

在 Windows™ 上，您指定的 ***user_id*** 必须是有效的登录名且必须存在于**用户管理器**中。如果应用程序连接到远程数据库服务器，则登录名必须同时存在于客户机和服务器的域中。

## 缺省用户 ID 的使用

如果未提供 USER Authentication 子句，则缺省的用户 ID 用于尝试此连接。

缺省用户 ID 是运行此应用程序的用户的登录名。在这种情况下，您通过标准授权过程获得网络许可权。例如，在 UNIX™ 上，缺省用户 ID 必须匹配可信主机文件（/etc/hosts.equiv 文件或由 REMOTE_SERVER_CFG 配置参数指定的文件）。在 Windows™ 上，您必须是域的成员，或者如果数据库服务器是本地安装的，则您必须是安装它的计算机上的有效用户。

### 缺省连接规范

可以使用 DEFAULT 关键字请求到缺省数据库服务器的***缺省连接***，而不用指定显式数据库环境。缺省数据库可以是本地的或远程的。要指定缺省数据库服务器，请在 **GBASEDBTSERVER** 中设置其名称。这种形式的 CONNECT 选项打不开数据库。

如果该 CONNECT TO DEFAULT 语句成功，则必须使用 DATABASE 语句或 CREATE DATABASE 语句来在缺省数据库环境中打开或创建数据库。

## 使用 DATABASE 语句的隐式连接

如果不在应用程序中执行 CONNECT 语句，则第一条 SQL 语句必须是以下***数据库语句***（或者是以下语句的单个 PREPARE 语句）：

- DATABASE
- CREATE DATABASE
- DROP DATABASE

如果这些数据库语句是应用程序中的第一条 SQL 语句，则该语句建立到数据库服务器的连接，这被称为***隐式***连接。如果数据库服务器仅指定数据库名称，则从 **DBPATH** 环境变量获取数据库服务器名称。这种情况在 指定数据库环境 中有描述。

进行隐式连接的应用程序可以显式建立其它的连接（使用 CONNECT 语句），但是不能建立另一个隐式连接，除非最初的隐式连接已关闭。应用程序可使用 DISCONNECT 语句终止隐式连接。在建立隐式连接之后，在关闭显式连接之前，不能使用任何数据库语句创建隐式连接。

进行*任何*隐式连接之后，该连接被认为是缺省连接，不管数据库服务器是否是 **GBASEDBTSERVER** 环境变量指定的缺省值。如果进行了其它显式连接，则此功能允许应用程序引用隐式连接，因为隐式连接没有标识。

例如，如果您在隐式连接后建立一个显式连接，则您可以通过发出 SET CONNECTION DEFAULT 语句使隐式连接变为当前连接。然而，这意味着一旦建立了隐式连接，则不能使用 CONNECT DEFAULT 语句，因为隐式连接现在是当前连接。

数据库语句总可用来打开数据库或者在当前数据库服务器上创建新的数据库。

## WITH CONCURRENT TRANSACTION 选项

WITH CONCURRENT TRANSACTION 子句使您能够在当前连接中存在活动事务时，切换到另一个连接。如果当前连接*不是*使用 WITH CONCURRENT TRANSACTION 子句建立的，则您不能在事务是活动的情况下切换到另一个连接；CONNECT 或 SET CONNECTION 语句失败，返回一条错误，并且当前连接中的事务继续保持活动。

这种情况中，应用程序必须在当前连接切换到另一个连接之前提交或者回滚当前连接中活动的事务。

WITH CONCURRENT TRANSACTION 子句支持多个并发事务的概念，其中每个连接可有其自己的事务并且 COMMIT WORK 和 ROLLBACK WORK 语句仅影响当前的连接。WITH CONCURRENT TRANSACTION 子句不支持单个事务横跨多个连接上的数据库的全局事务。COMMIT WORK 和 ROLLBACK WORK 语句对跨越多个连接的数据库不起作用。

以下示例说明如何使用 WITH CONCURRENT TRANSACTION 子句：

```
main()
{
EXEC SQL connect to 'a@srv1' as 'A';
EXEC SQL connect to 'b@srv2' as 'B' with concurrent transaction;
EXEC SQL connect to 'c@srv3' as 'C' with concurrent transaction;

/*
    Execute SQL statements in connection 'C' , starting a transaction
*/
EXEC SQL set connection 'B'; -- switch to connection 'B'

/*
    Execute SQL statements starting a transaction in 'B'.
    Now there are two active transactions, one each in 'B' and 'C'.
*/

EXEC SQL set connection 'A'; -- switch to connection 'A'

/*
    Execute SQL statements starting a transaction in 'A'.
    Now there are three active transactions, one each in 'A', 'B' and 'C'.
```

```
*/

EXEC SQL set connection 'C'; -- ERROR, transaction active in 'A'

/*
    SET CONNECTION 'C' fails (current connection is still 'A')
    The transaction in 'A' must be committed or rolled back because
    connection 'A' was started without the CONCURRENT TRANSACTION
    clause.
*/
EXEC SQL commit work;     -- commit tx in current connection ('A')

/*
    Now, there are two active transactions, in 'B' and in 'C',
    which must be committed or rolled back separately
*/

EXEC SQL set connection 'B'; -- switch to connection 'B'
EXEC SQL commit work;              -- commit tx in current connection ('B')

EXEC SQL set connection 'C'; -- go back to connection 'C'
EXEC SQL commit work;               -- commit tx in current connection ('C')

EXEC SQL disconnect all;
}
```

**警告：** 当应用程序使用 WITH CONCURRENT TRANSACTION 子句建立到同一个数据库环境的多个连接时，可能发生死锁现象。

### TRUSTED 子句

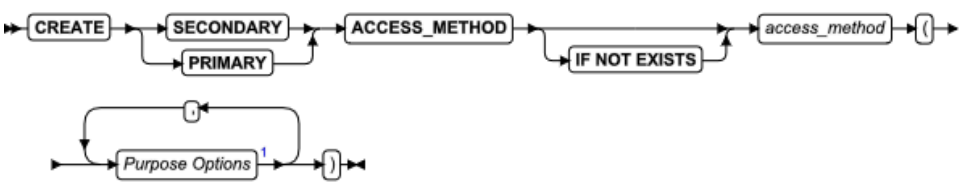
使用 TRUSTED 子句指定应用程序连接到数据库环境的连接是可信连接。

## 2.21 CREATE ACCESS_METHOD 语句

使用 CREATE ACCESS_METHOD 语句在 **sysams** 系统目录表中注册新的主或辅助存取方法。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *access method* | 此处为新的存取方法声明的名称 | 在 sysams 系统目录表的存取方法名称中必须是唯一的 | 标识符 |

**用法**

CREATE ACCESS_METHOD 语句将用户定义的存取方法添加到数据库。要创建存取方法，必须指定*目的函数*（或*目的方法*）、*目的标志*或*目的值*作为存取方法的属性，并将关键字（基于 **sysams** 系统目录表中列名）与 UDR 相关联。您必须具有 DBA 或 Resource 权限才能创建存取方法。

有关设置目标选项的信息，包含所有函数关键字的列表，请参阅用途选项。

PRIMARY 关键字为虚拟表指定用户定义的主存取方法。SECONDARY 关键字为虚拟索引指定创建用户定义的辅助存取方法。SECONDARY 关键字（和创建虚拟索引）在 Java™ Virtual-Table Interface 中不受支持。

以下语句创建了名为 T_tree 的辅助存取方法：

```
CREATE SECONDARY ACCESS_METHOD T_tree
(
am_getnext = ttree_getnext,

. . .
am_unique,
am_cluster,
am_sptype = 'S'
);
```

在前面的示例中，在目标选项列表中的 **am_getnext** 关键字与 **ttree_getnext( )** UDR 关联作为满足查询而扫描下一项的方法的名称。该示例指示了 **T_tree** 辅助存取方法支持唯一键和集群，并驻留在 sbspace 中。

任何目标函数任务中与 CREATE ACCESS_METHOD 语句相关联的 UDR（例如，之前示例中 **ttree_getnext( )** 和 **am_getnext** 的关联），必须已经由 CREATE FUNCTION 语句（或者具有等同功能的语句，例如：CREATE PROCEDURE FROM)）在数据库中注册过。

以下语句创建驻留在外部空间的名为 am_tabprops 的主存取方法。

```
CREATE PRIMARY ACCESS_METHOD am_tabprops
(
am_open = FS_open,
am_close = FS_close,
am_beginscan = FS_beginScan,
am_create = FS_create,
am_scancost = FS_scanCost,
am_endscan = FS_endScan,
am_getnext = FS_getNext,
am_getbyid = FS_getById,
```

```
am_drop = FS_drop,
am_truncate = FS_truncate,
am_rowids,
am_sptype = 'x'
);
```

如果包含可选的 IF NOT EXISTS 关键字，且该指定名称的存取方法已将存在于当前数据库中，则数据库服务器不采取操作（而不是向此应用程序发送异常）。

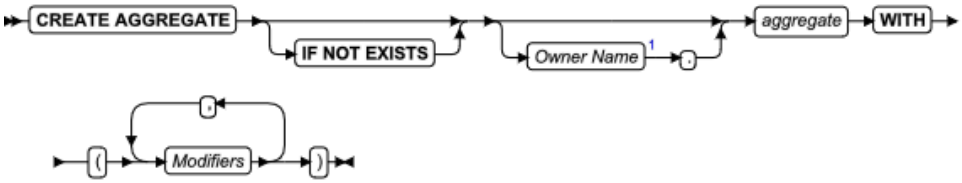
## 2.22 CREATE AGGREGATE 语句

使用 CREATE AGGREGATE 语句创建新的聚集函数，并在 **sysaggregates** 系统目录表中注册它。

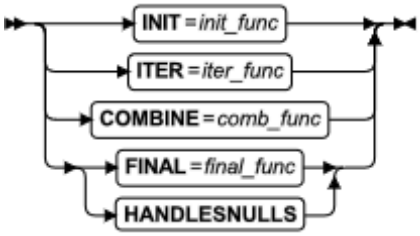
用户定义聚集（UDA）通过执行用户实施的聚集计算来扩展数据库服务器的功能。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




Modifiers




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *aggregate* | 新聚集的名称 | 在内置聚集和 UDR 的名称中必须唯一 | 标识符 |
| *comb_func* | 用来将部分结果合并到其它部分结果中并返回更新的部分结果的函数 | 必须同时为并行查询和顺序查询指定组合函数 | 标识符 |
| *final_func* | 将部分结果转换到结果类型的函数 | 如果省略此项，则返回值是 *iter_func* 的最终结果 | 标识符 |
| *init_func* | 初始化聚集计算所需 | 必须能够处理 NULL 参数 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 的数据结构的函数 | | |
| *iter_func* | 将单个值与部分结果合并并返回更新的部分结果的函数 | 必须指定迭代符函数。如果省略 *init_func* ，则 *iter_func* 必须能够处理 NULL 参数 | 标识符 |

**用法**

您可以按任何顺序指定 INIT 、ITER 、COMBINE 、FINAL 和 HANDLESNULLS 修饰符。

**重要：** 必须在 CREATE AGGREGATE 语句中指定 ITER 和 COMBINE 修饰符。不需要在 CREATE AGGREGATE 语句中指定 INIT 、FINAL 和 HANDLESNULLS 修饰符。

ITER 、COMBINE 、FINAL 和 INIT 修饰符指定用户定义的聚集的支持函数。这些支持函数在创建用户定义的聚集时无需存在。

如果省略 HANDLESNULLS 修饰符，拥有 NULL 聚集参数值的行将不会对聚集计算起作用。如果您将 HANDLESNULLS 修饰符包含在内，则还必须定义所有支持函数以处理 NULL 值。

**重要：** SELECT 语句只能包含一个 UDA 表达式，它的第一个参数是 DISTINCT 或 UNIQUE 关键字（而不是 ALL 关键字，或者没有关键字）。然而在包含子查询的查询中，您可在查询的每一级指定零个或一个 DISTINCT 或 UNIQUE 用户定义聚集表达式。内置的聚集不服从该限制。

如果包含可选的 IF NOT EXISTS 关键字，且指定名称的聚集已经在当前数据库中注册过，则数据库服务器不会采取任何操作（而不是向该应用程序发送异常）。

## 扩展聚集的功能

GBase 8s 通过两种方法扩展聚集的功能。仅将 CREATE AGGREGATE 语句用于两种情况的第二种。

- 内置聚集的扩展

  内置聚集是数据库服务器提供的聚集，如 **COUNT** 、**SUM** 或 **AVG** 。这些仅支持内置数据类型。要扩展内置聚集以使其支持用户定义的数据类型（UDT），就必须创建用户定义的例程，这些例程为该聚集重载二目运算符。关于扩展内置聚集的进一步信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

- 用户定义的聚集的创建

  用户定义的聚集是您定义来执行数据库服务器不提供的聚集计算的聚集。可使用内置数据类型和/或扩展数据类型的用户定义的聚集。要创建用户定义的聚集，请使用 CREATE AGGREGATE 语句。在此语句中，您命名新的聚集并指定计算聚集结果的支持函数。这些支持函数执行初始化、顺序聚集、结果组合和类型转换。

## 创建用户定义聚集的示例

以下示例定义了名为 **average** 的用户定义聚集：

```
CREATE AGGREGATE average
        WITH (
        INIT = average_init,
        ITER = average_iter,
        COMBINE = average_combine,
        FINAL = average_final
        );
```

在查询中使用 **average** 聚集之前，必须使用 CREATE FUNCTION 语句来创建 CREATE AGGREGATE 语句中指定的支持函数。

下表给出了每个支持函数可能为 **average** 执行的任务的示例。

| 关键字 | 支持函数 | 作用 |
|---|---|---|
| INIT | average_init | 分配并初始化存储当前总和和当前行计数的扩展数据类型 |
| ITER | average_iter | 对于每行，将表达式的值添加到当前总和并将当前行计数加一 |
| COMBINE | average_combine | 将部分结果的当前总和和当前行计数添加到其它结果，并返回更新的结果 |
| FINAL | average_final | 返回当前总和与当前行计数的比率，并将此比率转换到结果类型 |

## 并行执行

数据库服务器可将聚集计算拆分为几部分并并行计算它们。

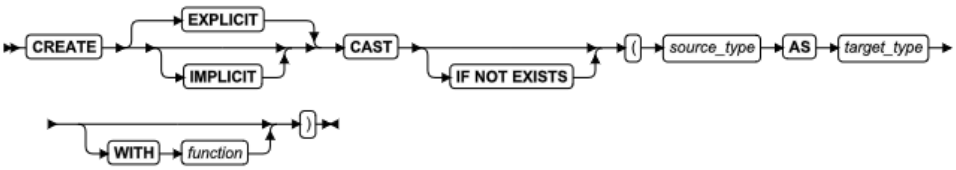
数据库服务器使用 INIT 和 ITER 支持函数来顺序计算每个部分。然后数据库服务器使用 COMBINE 函数将来自所有部分的部分结果组合为单个结果值。聚集是否并行是对用户透明的优化决策。

# 2.23 CREATE CAST 语句

使用 CREATE CAST 语句注册强制转型，强制转型可将数据从一种数据类型转换到另一种数据类型。

该语法是 SQL ANSI/ISO 标准的扩展。

**语法**



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 注册用来实施强制转型的 UDR | 请参阅 WITH 子句 | 标识符 |
| *source_type* | 要转换的数据类型 | 在注册强制转型时必须存在于数据库中。另见 源和目标数据类型 | 数据类型 |
| *target_type* | 从转换得到的数据类型 | 应用到 *source_type* （如以上所列）的相同的限制页应用到 *target_type* | 数据类型 |

**用法**

强制转型是数据库服务器用来将一种数据类型转换到另一种数据类型的机制。数据库服务器使用强制转型来执行以下任务：

- 比较 SELECT 、UPDATE 或 DELETET 语句中 WHERE 子句的两个值
- 将值作为参数传递到用户定义的例程
- 从用户定义的例程返回值

要创建强制转型，您必须在***源***数据类型和***目标***数据类型上均拥有必须的权限。所有用户拥有使用内置数据类型的许可权。然而，要在 OPAQUE 、DISTINCT 或指定的 ROW 数据类型之间创建强制转型，需要对数据类型上具有 Usage 权限。

如果包含可选的 IF NOT EXISTS 关键字，且在指定的数据类型之间的强制转型已经在当前数据库中注册过，则数据库服务器不会采取任何操作（而不是向该应用程序发送异常）。

CREATE CAST 语句在 **syscasts** 系统目录表中注册强制转型。有关 **syscasts** 的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 中的关于系统目录表的章节。

## 源和目标数据类型

CREATE CAST 语句定义将***源***类型转换为***目标***类型的强制转型。***源***和***目标***数据类型在执行 CREATE CAST 语句来注册强制转型时必须同时存在于数据库中。

***源***和***目标***数据类型具有以下限制:

- ***源***和***目标***类型，可以是（但不能同时是）内置数据类型。
- ***源***和***目标***类型都不能是对方的 DISTINCT 类型。
- ***源***和***目标***类型都不能是 COLLECTION 数据类型。

## 显式和隐式强制转型

处理多个数据类型的查询常常需要将数据从一种数据类型转换为另一种数据类型的强制转型。

可使用 CREATE CAST 语句创建以下类型的强制转型：

- 使用 CREATE EXPLICIT CAST 语句定义***显式***强制转型。

- 使用 CREATE IMPLICIT CAST 语句定义*隐式*强制转型。

数据库服务器调用内置强制转型将一种内置数据类型转换为另一种内置类型而不是直接替代。例如，数据库服务器通过内置强制转型将 CHAR 字符类型转换为 INTEGER 数值类型。

## 显式强制转型

显式强制转型是您必须用 CAST AS 关键字或用强制转型运算符（::）特别调用的强制转型。数据库服务器*不*自动调用显式强制转型来解决数据类型转换。EXPLICIT 关键字是可选的；缺省情况下，CREATE CAST 语句创建一个显式强制转型。

以下 CREATE CAST 语句定义了从 rate_of_return 不透明数据类型到 percent distinct 数据类型的显式强制转型：

```
CREATE EXPLICIT CAST (rate_of_return AS percent
        WITH rate_to_prcnt);
```

以下 SELECT 语句在其 WHERE 子句中调用此显式强制转型，来将 bond_rate 列（rate_of_return 类型的）与 initial_APR 列（percent 类型的）进行比较：

```
SELECT bond_rate FROM bond
        WHERE bond_rate::percent > initial_APR;
```

## 隐式强制转型

隐式强制转型是当数据库服务器遇到无法与内置强制转型相比较的数据类型时可自动调用的强制转型。此类型的强制转型使得数据库服务器能够自动处理其它数据类型之间的转换。

要定义隐式强制转型，在 CREATE CAST 语句中指定 IMPLICIT 关键字。例如，以下 CREATE CAST 语句指定数据库服务器应当自动使用 prcnt_to_char( ) 函数来从 CHAR 数据类型转换到 distinct 数据类型 percent ：

```
CREATE IMPLICIT CAST (CHAR AS percent WITH char_to_prcnt);
```

此强制转型只支持*从* CHAR 数据类型*到* percent 的自动转换。要使数据库服务器*从* percent *转换到* CHAR，您也需要定义另一个隐式强制转型如下：

```
CREATE IMPLICIT CAST (percent AS CHAR WITH prcnt_to_char);
```

数据库服务器自动调用 char_to_prcnt( ) 函数来评估以下 SELECT 语句的 WHERE 子句：

```
SELECT commission FROM sales_rep WHERE commission > "25
```

用户也可以显式调用隐式强制转型。有关如何显式地调用强制转型函数的更多信息，请参阅 显式强制转型。

当数据类型之间的转换不存在内置强制转型时，您可创建用户定义的强制转型来进行必要的转换。

## WITH 子句

CREATE CAST 语句的 WITH 子句指定要调用的用户定义函数的名称，来执行强制转型。该函数称为强制转型函数。

除非***源数据类型***和***目标数据类型***有相同的表示法，否则您必须指定***函数名称***。当满足以下条件时两种数据类型有相同的表示法：

- 两种数据类型都有相同的长度和对齐方式。
- 两种数据类型都通过引用或都通过值传递。

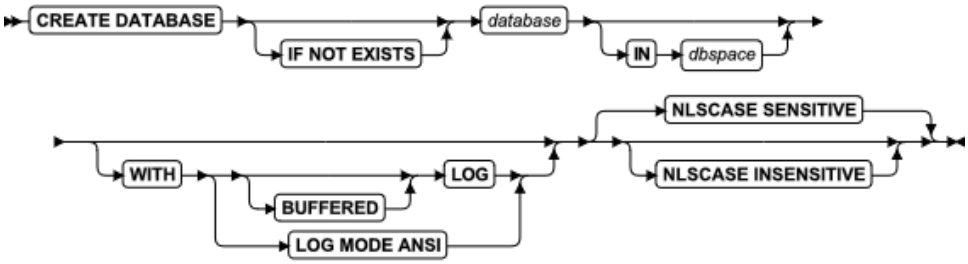
强制转型函数注册的数据库必须与调用强制转型时强制转型所在的数据库是同一数据库，但在创建强制转型时不需要存在。CREATE CAST 语句不检查特定***函数名称***上的许可，甚至不检查强制转型的存在。每次用户显式或隐式调用强制转型，数据库服务器验证用户在强制转型函数上是否有 Execute 权限。

## 2.24 CREATE DATABASE 语句

使用 CREATE DATABASE 语句创建新数据库。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *database* | 在此您为正在创建的新数据库声明的名称 | 在数据库服务器的数据库名称中必须唯一 | 数据库名 |
| *dbspace* | 为此数据库存储数据的 dbspace；缺省值为 root dbspace | 必须存在 | 标识符 |

**用法**

该语句是 ANSI 标准语法的扩展。（SQL 语句的 ANSI/ISO 标准不为数据库的构造，即数据库的形成过程指定任何语法。）

CREATE DATABASE 指定的*数据库*成为当前的数据库。

如果 DBCREATE_PERMISSION 配置参数没有设置，则任何用户都可创建数据库。然而，如果该配置文件包含一个或多个 DBCREATE_PERMISSION 指定，则只有指定的用户才能创建数据库。无论 DBCREATE_PERMISSION 是否设置，用户 **gbasedbt** 都可以使用 CREATE DATABASE 语句。

您声明的*数据库*名称在您正在工作的数据库服务器环境中必须是唯一的。数据库服务器创建描述新数据库的结构的系统目录表。

如果您包含了可选的 IF NOT EXISTS 关键字，则当指定的数据库名称已经存在于您要连接的由数据库服务器实例管理的数据库中时，数据库不会采取任何操作（而不是向应用程序发送异常）。

当创建数据库时，只有您自己可以存取它。它不会让其他用户访问，除非您作为 DBA 通过运行 GRANT 语句授权。

如果先前的 CONNECT 语句建立了到数据库的显式连接，而且该连接仍是当前连接，那么在使用 DISCONNECT 语句关闭该显式连接之前，你不能使用 CREATE DATABASE 语句（或任何创建隐式连接的 SQL 语句）。

在 GBase 8s ESQL/C 中，CREATE DATABASE 语句不能出现在多语句 PREPARE 操作中。

您要连接的数据库服务器实例的 SQL_LOGICAL_CHAR 配置参数记录在该新数据库的系统目录中。SQL_LOGICAL_CHAR 配置参数扩展内置字符数据类型声明中的大小设定的扩展。即使 GBase 8s 实例管理的数据库被新的 SQL_LOGICAL_CHAR 值停止并重启，也不能更改此设置，它会持续到数据库被删除。**systables** 系统目录表的 **flags** 列为此数据库编码 SQL_LOGICAL_CHAR 设置。

如果您未指定 *dbspace* ，则缺省情况下，数据库服务器在 **root** dbspace 中创建系统目录表。然而，如果您启用数据库自动定位，则缺省情况下，数据库创建于被此服务器选择的 dbspace 中。要启用数据库的自动定位，请将 AUTOLOCATE 配置参数或会话环境变量设置为一个正整数。

以下语句在 root dbspace 或由服务器选定的 dbspace 中（这取决于是否要启用自动定位）创建 vehicles 数据库：

```
CREATE DATABASE vehicles;
```

由于以上示例不包含日志记录设定和 NLSCASE 设定，因此缺省情况下

- **vehicles** 数据库不支持事务日志记录，
- 并且如果它的语言环境使用区分大小写字母的代码集，则数据库对所有的内置的字符数据类型都区分大小写。

以下语句在 research dbspace 中创建 vehicles 数据库：

```
CREATE DATABASE vehicles IN research;
```

但是如果没有 DROP DATABASE 语句删除现有的第一个示例创建的 **vehicles** 数据库，则第二个示例发生错误并失败，并且没有数据库被创建，因为 **vehicles** 数据库的标识符在数据库服务器实例中不是唯一的。

## 日志记录选项

CREATE DATABASE 语句的日志记录选项决定了为数据库所作的日志记录类型。失败的情况下，数据库服务器使用日志来重新创建您的数据库中的所有已提交的事务。

以下示例使用 WITH LOG 选项创建带有未缓冲日志记录的数据库：

```
CREATE DATABASE unbufDatabase WITH LOG;
```

如果不指定 WITH LOG 关键字，则 GBase 8s 将创建不能使用支持事务日志记录的事务或语句（BEGIN WORK 、COMMIT WORK 、ROLLBACK WORK 、RELEASE SAVEPOINT 、ROLLBACK

TO SAVEPOINT 、SET IMPLICIT TRANSACTION 、SET LOG 和 SET ISOLATION） 的没有日志记录的数据库。

当您在高可用集群的辅助服务器上创建数据库时，必须使用 WITH LOG 选项。

## 指定已缓冲的日志记录

以下示例创建了使用已缓冲的日志的数据库：

```
CREATE DATABASE vehicles WITH BUFFERED LOG;
```

如果使用已缓冲的日志，则会略微提高日志记录的性能，但要冒失败之后无法重新创建最后几个事务的危险。

## 兼容 ANSI 的数据库

当在 CREATE DATABASE 语句中使用 LOG MODE ANSI 选项时，您创建的数据库就是兼容 ANSI 的数据库，且符合 SQL 语言的 ANSI/ISO 标准。

以下示例创建兼容 ANSI 的数据库：

```
CREATE DATABASE employees WITH LOG MODE ANSI;
```

兼容 ANSI 的数据库与不兼容 ANSI 的数据库存在几个方面的不同。包括以下特征的不同：

- 所有 SQL 语句自动包含在事务中。
- 所有数据库使用未缓冲的日志记录。
- 实施所有者命名。

  除非您是所有者，否则查看任何表、视图、同义词、索引或约束时必须使用所有者名称。除非您将所有者名称包含在引号中，否则所有者名称中的字母缺省采用大写字符。（要防止在未分隔所有者名称中小写字母升档，您可以将 **ANSIOWNER** 环境变量设置为 1。）

  此外，UDR 的例行签名包含所有者的名称；在不兼容 ANSI 的数据库中，它只对 **sysdbopen( )** 和 **sysdbclose( )** 程序为真。
- 对于会话，缺省隔离级别为 REPEATABLE READ 。
- 对象上的缺省权限与那些不兼容 ANSI 的数据库不同。当您创建表或同义词时，缺省情况下其它用户不接收对其存取权限（如果是 PUBLIC 组的成员）。
- 所有的 DECIMAL 数据类型时定点值。如果您声明列为 DECIMAL(***p***)，则缺省大小是零，表明只能存储整型值。（在不兼容 ANSI 的数据库中，DECIMAL(***p***) 是浮点数据类型，它的规模大到足以存储一个值的指数符号。）

兼容与不兼容 ANSI 的数据库之间存在其它小差别。这些差别在此手册中与其它相关 SQL 语句一起记录。

创建兼容 ANSI 的数据库不意味着当您运行数据库时，自动收到 SQL 语法 ANSI/ISO 标准的 GBase 8s 警告。还必须使用 **-ansi** 标记或 **DBANSIWARN** 环境变量来接收这类警告。

有关 **-ansi** 和 **DBANSIWARN** 的其它信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

## 指定 NLSCASE 区分大小写

您可以显式地创建区分大小写或不区分大小写的数据库。

缺省情况下，在数据库中的区域设置将代码集的分离子集分类为***大写字母***和***小写字母***，GBase 8s 数据库创建为区分大小写。数据库语言环境通过设置 DB_LOCALE 环境变量而定义。语言环境的示例，若其代码集在缺省 US English 语言环境中识别字母大小写，则在升序排列中，小写字母会超过大写字母。在缺省的语言环境中，以下语句创建区分大小写的数据库：

```
CREATE DATABASE employees IN dbspaceYee WITH BUFFERED LOG;
```

要显式地创建区分大小写的数据库，请将 NLSCASE SENSITIVE 关键字包含在 CREATE DATABASE 语句中，并作为其最后的指示，如下所示：

```
CREATE DATABASE stores IN dbsp1 WITH LOG NLSCASE SENSITIVE;
```

因为缺省启用区分大小写，以下语句具有相同的作用：

```
CREATE DATABASE stores IN dbsp1 WITH LOG;
```

在区分大小写的数据库中，例如 Boolean 条件 'M' MATCHES 'm' 计算为假。

所有的 GBase 8s 数据库对于内置 CHAR 、LVARCHAR 和 VARCHAR 数据类型的字符串字符操作都区分大小写。如果您创建区分大小写数据库，则不论缺省或显式使用 NLSCASE SENSITIVE 关键字，如果数据库语言环境支持字母大小写，数据库仍将区域语言支持的数据类型 NCHAR 和 NVARCHAR 看作区分大小写。

### 创建不区分大小写的数据库

在某些应用程序中，字符串的大小写会被忽略。 例如，数据项处理，可能接受字符串 'M' 和 'm' 在一条记录中是逻辑等价的。对于大数据集，应用条件逻辑将两种情况变量转换为单个值可能导致性能低于将记录存储在不区分大小写的数据库的 NCHAR 或 NVARCHAR 列中，其中 'M' 和 'm' 字符串编码都是不区分大小写的值此处的条件 'M' MATCHES 'm' 对 NCHAR 或 NVARCHAR 列计算为真。

每个创建有 NLSCASE INSENSITIVE 属性的数据库存储大小写的 NCHAR 和 NVARCHAR 字母，正如它们被加载到它们的表中一样；查询返回的任何未更改的记录都具有原始字母。然而，在所有的对 NCHAR 和 NVARCHAR 值的操作(例如:排序、分组或标识重复行）中，数据库服务器会忽略字母大小写的变化，例如：字符串 'Mi' 和 'mI' 的值是一样的。有关字母大小写的信息没有被丢弃，但是当数据库服务器处理 NLS 数据类型时仍不会使用这些信息。

当在 CREATE DATABASE 语句中包含 NLSCASE INSENSITIVE 关键字作为其最后的指定时，数据库服务创建处理以下字符串类型时不考虑字母大小写的数据库：

- 存储在 NLS 数据类型的 NCHAR 和 NVARCHAR 列中字符串
- 存储为基于 NCHAR 或 NVARCHAR 数据类型的 DISTINCT 数据类型的字符串
- 存储为具有集合数据类型的那些数据类型的元素的字符串
- 存储在指定或未指定的 ROW 数据类型中的以上数据类型的字段中字符串
- 存储为 SPL 变量的那些数据类型的字符串

- 隐式或显示强制转型为那些数据类型的字符串
- 作为被函数返回的那些数据类型的输出参数的字符串

此处的 "***这些数据类型***" 引用了在同一列表中标识的字符数据类型。

以下语句创建了具有 NLSCASE INSENSITIVE 属性的数据库：

```
CREATE DATABASE stores IN dbsp2 WITH BUFFERED LOG NLSCASE INSENSITIVE;
```

**重要：**创建为 NLSCASE INSENSITIVE 的数据库将所有其它内置字符数据类型（CHAR 、LVARCHAR 和 VARCHAR）都对待为区分大小写。也就是说，如果它们的数据类型不在上表的 NLS 字符数据类型中，则区分大小写的数据库仍可以执行区分大小写字符串值的处理。

要在区分大小写的数据库的 NCHAR 或 NVARCHAR 数据类型的字符串上执行区分大小写操作，您必须显式地将字符串强制转型成 CHAR 、LVARCHAR 或 VARCHAR 数据类型，然后再执行区分大小写操作。

**NLSCASE INSENSITIVE 查询的示例**

在区分大小写的数据库中，当一个查询调用聚集函数或者对 NCHAR 或 NVARCHAR 列包含 GROUP BY 子句时，数据库服务器将数据库中的大小写字母作为重复的值，程序片段示例如下。

```
CREATE DATABASE casedb WITH LOG NLSCASE INSENSITIVE;
    CREATE TABLE foo (cc CHAR(5), nc NCHAR(5));
    INSERT INTO foo VALUES ('GBASE', 'gBASE');
    INSERT INTO foo VALUES ('gbase', 'gbaSE');
    INSERT INTO foo VALUES ('gbase', 'gbaSE');
    INSERT INTO foo VALUES ('GBase', 'GBase');

    SELECT COUNT(nc) FROM foo
    GROUP BY nc;
    SELECT COUNT(nc) FROM foo
    WHERE nc = 'gbase' GROUP BY nc;
```

在以上两条查询中，**COUNT** 聚集函数都返回 4 ，是 INSERT 语句加载到 **foo** 中的总行数。因为 **nc** 列是 NLS 数据类型，所有的行都满足 WHERE 子句中的 `nc = 'gbase'` 条件，尽管在 **nc** 值中有字母大小写的变化。

在同一表上执行以下查询，

```
SELECT nc FROM foo GROUP BY nc;
```

输出可能是来自 INSERT 语句的任何字符串值（亦即 'GBASE'、'gBASE'、'gbase'、'gbaSE' 或 'GBase'），这取决于服务器处理或扫描行的顺序。

在同一表上的下一查询，通过在投影子句中包含 DISTINCT 关键字从结果集排除了重复的行：

```
SELECT DISTINCT nc FROM foo;
```

此处返回一行，因为从 NLSCASE INSENSITIVE 角度来看，所有的行具有相同的值，尽管在字母大小写上有变化。正如先前的示例，从插入的行中检索到的第一行将由查询返回。

以下示例包含 DISTINCT 关键字，将其作为 COUNT 聚集函数其中的一个参数：

```
SELECT COUNT(DISTINCT nc) FROM foo;
```

此处还是返回计数 1，因为在此不区分大小写的数据库中，foo 表中所有的行都评估为重复行。

**NLSCASE INSENSITIVE 数据库的限制**

以下限制应用于创建 NLSCASE INSENSITIVE 属性的数据库：

- 它们支持只与拥有 NLSCASE INSENSITIVE 属性的数据库的分布跨数据库和跨服务器查询。
- 区分大小写的数据库无法连接到 NLSCASE INSENSITIVE 数据库。若要尝试此操作，则会产生以下错误：

  -26801 Cannot reference an external database that is not case sensitive.

- NLSCASE INSENSITIVE 数据库无法连接到区分大小写的数据库。若要尝试此操作，则会产生以下错误：

  -26802 Cannot reference an external database that is case sensitive.

  唯一的例外是，NLSCASE 设置不会阻止同一 GBase 8s 数据库服务器实例到区分大小写系统数据库（例如 **sysmaster** 、**sysadmin** 、**sysutils** 、**sysusers** 和 **syscdr**）的连接。存取系统数据库操作的结果依赖于其它数据库中的 NLSCASE 设置。
- **gload** 和 **gunload** 实用程序不支持具有 NLSCASE INSENSITIVE 属性的数据库。
- 在 Enterprise Replication 集群中，当您在不同于它们 NLSCASE 属性的数据库中指定了复制对时，不会发出错误或警告。要降低不一致性的风险，复制只与区分大小写的数据库区分大小写的数据库，复制只与 NLSCASE INSENSITIVE 的数据库 NLSCASE INSENSITIVE 数据库。

# 2.25 CREATE DEFAULT USER 语句 (UNIX™、Linux™)

使用 CREATE DEFAULT USER 语句定义缺省的内部已经通过身份验证的用户的属性集。该语句是 SQL 语言 ANSI/ISO 标准的扩展。

**语法**


**用法**

CREATE DEFAULT USER 是 CREATE USER 语句的特例。在使用 CREATE DEFAULT USER 语句定义缺省用户属性之后，您可使用 CREATE USER 语句（但是省略 PROPERTIES 子句）创建具有缺省用户属性的新用户。

只有 DBSA 能发出 CREATE DEFAULT USER 语句。在非 root 安装中，安装服务器的用户等价于 DBSA，除非用户将 DBSA 权限委托给另一个用户。

在缺省用户（CREATE DEFAULT USER 语句创建）可以连接到数据库服务器之前，USERMAPPING 配置参数必须设置为启用支持映射用户的值（ADMIN 或 BASIC）。DBSA 可以发出 CREATE

DEFAULT USER 语句给缺省用户映射关联的合适身份级别的属性。USERMAPPING 配置参数必须设置为 ADMIN 来启用缺省用户拥有服务器管理权限，通过 AUTHORIZATION 关键字，AAO 、BARGROUP 、DBSA 和 DBSSO 是指定的管理权限的关键字选项

您必须在 **sysusers** 数据库中的 SYSUSERMAP 表中输入值以映射用户具有适当的用户属性，以致于 SQL 语句映射的用户可以正确地工作。

您不能在 CREATE DEFAULT USER 语句中指定密码或锁定账户或解锁账户信息。该语句等价于 GRANT ACCESS TO PUBLIC PROPERTIES 语句。等价的语法

REVOKE ACCESS TO PUBLIC;

是：

DROP DEFAULT USER;

要内部更改缺省的已经通过身份验证的用户的属性，您可发出 ALTER DEFAULT USER WITH PROPERTIES 语句。

CREATE DEFAULT USER 语句的执行可带有 CRUR 审计码审计，也是 CREATE USER 语句相同的助记符。

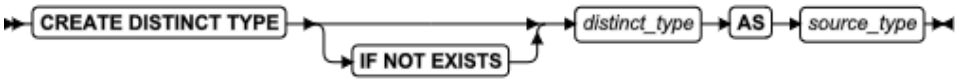
有关 CREATE DEFAULT USER 语句的 PROPERTIES 选项的更多信息，请参阅 CREATE USER 语句（UNIX、Linux）。

## 2.26 CREATE DISTINCT TYPE 语句

使用 CREATE DISTINCT TYPE 语句创建新的 distinct 数据类型。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *distinct_type* | 在这里为新的 distinct 数据类型声明的名称 | 在兼容 ANSI 的数据库中，所有者和数据类型的组合在数据库中必须是唯一的。在不兼容 ANSI 的数据库中，名称在数据库中的数据类型名称中必须是唯一的 | 数据类型 |
| *source_type* | 新类型所基于的现有类型名称 | 必须为内置数据类型或用 CREATE DISTINCT TYPE 、 CREATE OPAQUE TYPE 或 CREATE ROW TYPE 语句创建的类型 | 数据类型 |

**用法**

***Distinct 类型***是基于内置数据类型或现有不透明数据类型，指定的 ROW 数据类型或者其它 Distinct 数据类型的数据类型。Distinct 数据类型是强归类的。虽然 Distinct 类型与其源类型对数据有相同的物理表示法，但两种类型的值在没有从一种类型到另一种类型的显式强制转型的情况下无法进行比较。

要创建 Distinct 数据类型，您必须拥有数据库上的 Resource 权限。任何拥有 Resource 权限的用户均可从内置数据类型之一创建 Distinct 类型，而该用户是用户 gbasedbt 所拥有的。

**重要：** 不能在 SERIAL 、BIGSERIAL 或 SERIAL8 数据类型上创建 Distinct 类型。

要从不透明类型、指定的 ROW 类型或另一 Distinct 类型创建 Distinct 类型，您必须是该数据类型的所有者或在该数据类型上拥有 Usage 权限。

缺省情况下，一旦定义了 Distinct 类型，之一该 Distinct 类型的所有者和 DBA 可以使用它。然而，Distinct 类型的所有者可向其他用户授权对该 Distinct 类型的 Usage 权限。

Distinct 类型与其源类型有相同的存储结构。以下语句创建了基于内置 DATE 数据类型的 Distinct 类型 birthday：

```
CREATE DISTINCT TYPE birthday AS DATE;
```

虽然 GBase 8s 使用对 Distinct 类型以及它的源类型使用相同的存储格式，但 Distinct 类型与其源类型不能在一个操作中进行比较，除非一个类型显式强制转型到另一个类型。

如果您包含了 IF NOT EXISTS 关键字，当指定名称的 Distinct 数据类型已经在当前数据库中注册过时，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

## 对 Distinct 类型的权限

要创建 Distinct 类型，您必须拥有该数据库上的 Resource 权限。当创建 Distinct 类型时，只有您，即所有者对此类型拥有 Usage 权限。使用 GRANT 或 REVOKE 语句向其它数据库用户授权或调用 Usage 权限。

要找出特定类型上存在哪些特权，请在 **sysxtdtypes** 系统目录表中检查所有者名称，并在 **sysxtdtypeauth** 系统目录表中检查可能已经授予的其它数据类型特权。有关系统目录表的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

DB-Access 实用程序也可显示对 Distinct 类型的特权。

## 支持函数和强制转型

当您创建 Distinct 类型时， GBase 8s 自动定义两种显式强制转型：

- 从 Distinct 类型到其源类型的强制转型
- 从源类型到 Distinct 类型的强制转型

因为这两个数据类型具有相同的表示法（相同的长度和对齐方式），所以实现这些强制转型不需要支持函数。

您可在 Distinct 类型与其源类型之间创建隐式强制转型。要创建隐式强制转型，请使用 Table Options 子句来指定外部数据的格式。然而，您必须首先删除 Distinct 类型与其源类型之间的缺省显式强制转型。

在源类型上定义的所有支持函数的强制转型可用于 Distinct 类型。然而，对 Distinct 类型定义的强制转型和支持函数对源类型不可用。请使用 Table Options 子句指定外部数据的格式。

## 操纵 Distinct 类型

当将 Distinct 类型与其源类型进行比较或操纵它们的数据时，在以下情况中您必须显式地将一种类型强制转型为其它类型：

- 使用其它类型的值插入或更改一种类型的列
- 使用关系运算符来加、减、乘、除、比较或以其它方式操纵两个值，一个源类型的值和一个 Distinct 类型的值

例如，假设您创建了 Distinct 类型 `dist_type` ，它基于 NUMERIC 数据类型。然后您创建了带有两列的表，一个属于 `dist_type` 类型，一个属于 NUMERIC 类型。

```
CREATE DISTINCT TYPE dist_type AS NUMERIC;
CREATE TABLE t(col1 dist_type, col2 NUMERIC);
```

要直接将 Distinct 类型与其源类型进行比较或者将源类型的值分配到 Distinct 类型的列上，您必须将一种类型强制转型到其它类型，如以下示例所示：

```
INSERT INTO tab (col1) VALUES (3.5::dist_type);

    SELECT col1, col2
    FROM t WHERE (col1::NUMERIC) > col2;

    SELECT col1, col2, (col1 + col2::dist_type) sum_col
    FROM tab;
```

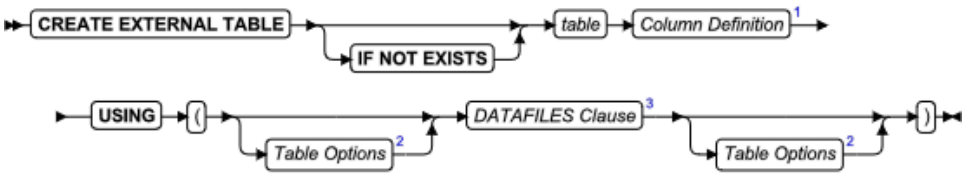
有关在本地数据库外的表内存取 DISTINCT 数据类型的查询和其它分布 DML 操作的信息，请参阅分布式操作中的 DISTINCT 类型 。

# 2.27 CREATE EXTERNAL TABLE 语句

使用 CREATE EXTERNAL TABLE 语句定义不属于您的数据库的外部源以加载和卸载您数据库的数据。

CREATE EXTERNAL TABLE 语句是实施是 SQL ANSI/ISO 标准的扩展。

**语法**



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *table* | 存储外部数据的表的名称 | 在当前数据库中的表、视图和同义词的名称中必须是唯一的 | 标识符 |

**用法**

使用外部表从您的数据库加载和卸载数据或者加载和卸载数据到数据库中。还可以使用外部表查询不在 GBase 8s 数据库文本文件中的数据。

语法图的第一部分声明了表的名称并定义了它的列。

The portion that 随 USING 关键字之后的部分标识了当您使用外部表时数据库服务器打开的外部文件，并指定该外部表特征的其它选项。

在执行 CREATE EXTERNAL TABLE 语句后，可以使用 INSERT INTO ... SELECT 语句从外部源移动数据，或者移动数据到外边源。有关将查询结果加载到外部表中的更多信息，请参阅 INTO EXTERNAL 子句 章节。

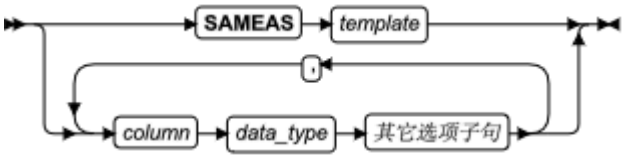
在高可用集群的辅助服务器上不支持 CREATE EXTERNAL TABLE 语句。

如果您包含 IF NOT EXISTS 关键字，而指定名称的外部表已经在当前数据库的 **systables** 系统目录表中注册过，或者指定的名称是当前数据库中数据库表、视图、顺序对象、或者同义词的标识符时，数据库服务器不采取任何操作（而不是向应用程序发送异常）。

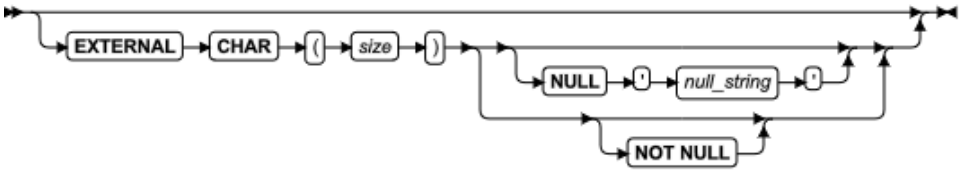
## 列定义

使用 CREATE EXTERNAL TABLE 已经的列定义段声明新外部表的单个列的名称和数据类型。

列定义




其它选项子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 外部表的每一列的列名 | 对于每一列，必须指定内置数据类型 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *data_type* | 列的数据类型 | *data_type* 可以是任何 GBase 8s 支持的数据类型 | 数据类型 |
| *template* | 具有与外部表相同结构的现有表 | 不能是列的子集且与任一列数据类型不同 | 数据库对象名 |
| *size* | 列大小（以字节表示）。缺省为 1 | 整数； 1 ≤ *size* ≤ 32,767 | 精确数值 |
| *null_string* | 代表 NULL 的值 | 请参阅定义 NULL 值 | 引用字符串 |

## 使用 SAMEAS 子句

SAMEAS ***template*** 子句使用在新表的定义中 ***template*** 表的所有的列名称及其数据类型。

您不能对固定格式的文件使用 SAMEAS 子句。

**示例**

考虑将定界的 ASCII 文本文件加载到具有以下结构的表中：

```
TABLE employee (
        name CHAR(18) NOT NULL,
        hiredate DATE DEFAULT TODAY,
        address VARCHAR(40),
        empno INTEGER);
```

SQL 语句可使用以下方法将数据加载到 employee 表：

```
CREATE EXTERNAL TABLE emp_ext
        SAMEAS employee
        USING (
        DATAFILES ("DISK:/work2/mydir/emp.dat"),
        REJECTFILE "/work2/mydir/emp.rej"
        );
        INSERT INTO employee SELECT * FROM emp_ext;
```

外部表都具有与源表每一列相同的名称、类型和缺省值，因为 CREATE 语句包含 SAMEAS 关键字。缺省的格式是定界的，因此不需要任何格式化关键字。

定义在数据库表中列上的检查约束不会为外部表继承。然而，NOT NULL 约束会被外部表继承。

定界文件缺省为 ASCII 。缺省的行定界符是行结束字符，除非您在创建外部表时使用 RECORDEND 关键字定义不同的定界符。（RECORDEND 关键字只对定界格式有效。）

## 使用 EXTERNAL 关键字

使用 EXTERNAL 关键字给您的拥有一个不同于内部表数据类型的外部表的每一列指定 CHAR 数据类型。

例如，内部表中有一 VARCHAR 列，您想要将其映射到外部表中的 CHAR 列。

必须以固定格式为每一列指定外部类型。除了 BYTE 和 TEXT 列（它们的指定是可选的），您不能为分隔的列指定外部类型。

### *定义 NULL 值*

当从外部源加载或卸载数据时，您可以定义一个解释为 NULL 的值。

对固定格式的外部表，数据库服务器使用 NULL 表示法作为数据加载到数据库中时的解释值，也使用 NULL 表示法作为数据卸载到外部表时将 NULL 值格式为合适的数据类型。

NULL 表示法必须适合外部字段的长度。

## 操纵固定格式文件中数据

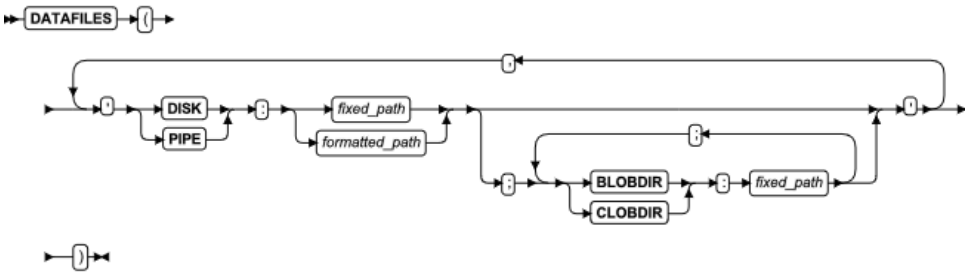
固定格式文件是所有行都具有一样长度的文件。

对于 FIXED 格式的文件，您必须为每一列声明列名和 EXTERNAL 项来设置属性的名称和数目。对于固定格式文件，只允许的数据类型是 CHAR。您可以使用 NULL 关键字指定哪个字符串被诠释为 NULL 值。

## DATAFILES 子句

DATAFILES 子句指定当您使用外部表时打开的操作系统文件或管道。

DATAFILES 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *fixed_path* | 在外部表的定义中的输入或输出文件的路径名 | 请参阅随该表之后的注意事项 | 必须遵循操作系统规则 |
| *formatted_path* | 使用模式匹配字符的格式化的路径名 | 请参阅随该表之后的注意事项 | 必须遵循操作系统规则 |

T 数据库服务器不会验证存在于指定的 *fixed_path* 或 *formatted_path* 上的任何文件或管道，此指定的管道是打开的，且用户具有存取该文件系统的权限。然而，如果当数据库服务器尝试读或写到外部表时，指定的管道正在被使用，即是打开的，随后外部表上的操作会失败，除非该路径是有效的。

有关 DATAFILES 子句的示例，请参阅外部表示例。

| 关键字 | 描述 |
| --- | --- |
| CLOBDIR | 指定存储 CLOB 文件的服务器的目录。 |
| BLOBDIR | 指定存储 BLOB 文件的服务器的目录。当创建查询时，在 CLOBDIR 之后的 BLOBDIR 之后，指定 DISK 。如果省略了 BLOBDIR ，则 BLOB 文件存储于 DISK 子句指定的相同的目录。如果 BLOBDIR 和 CLOBDIR 都被省略，则会为 BLOB 或 CLOB 列创建一个新的文件，并将其存储于 DISK 子句指定的目录下。 |

以下示例中，存储在 /work1/exttab1.dat 中的行拥有位于 /work1/blobdir1 的 BLOBs 和位于 /work1/clobdir1 目录下的 CLOBs 。

存储在 /work1/exttab2.dat 中的行拥有位于 /work1 目录中的 BLOBs 和位于 /work1/clobdir2 目录中的 CLOBs。因为省略了 BLOBDIR 子句，所以 BLOBs 存储在存储 exttab2.dat 的目录中。

存储在 /work1/exttab3.dat 中的行拥有位于 /work1 目录中的 BLOBs 和 CLOBs ，因为 BLOBDIR 和 CLOBDIR 都被省略了。

```
CREATE EXTERNAL TABLE exttab (
                        id      SERIAL,
                        lobc    CLOB,
                        lobb    BLOB)
                        USING (DATAFILES(

"DISK:/work1/exttab1.dat;BLOBDIR:/work1/blobdir1;CLOBDIR:/work1/clobdir1",
                        "DISK:/work1/exttab2.dat;CLOBDIR:/work1/clobdir2",
                        "DISK:/work1/exttab3.dat"),
                        DELIMITER '|');
```

## 在外部表中使用格式化字符

可以使用格式化的路径名称来指定文件名，通过使用替代字符 %r（***first ..last***）。

| 格式化字符串 | 作用 |
| --- | --- |
| %r(***first ..last***) | 在 GBase 8s 服务器上对外部表指定多个文件。 |

first 和 last 参数代表语句运行时代替的表达式中的值的范围。例如：指定 my_file.%r(1..3) 可展开为：

my_file.1

my_file.2

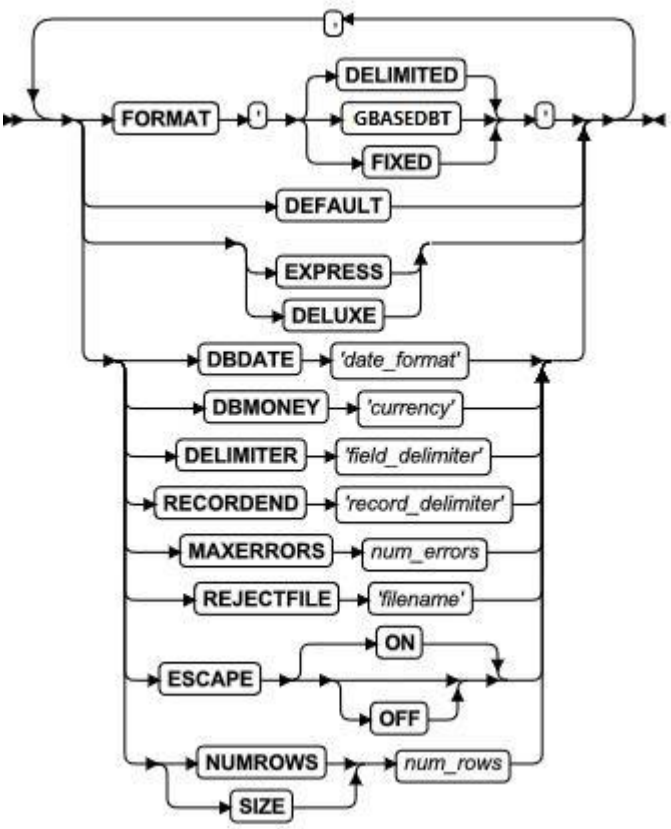
my_file.3

GBase 8s 仅支持的格式字符是 %r 。

## Table 选项

这些选项指定定义外部表的其它特征，并定义该表上加载或卸载操作的属性。

Table 选项




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *field_delimiter* | 用来分隔字段的字符。缺省值为管道（ \| ）字符 | 对于非打印字符，使用八进制符号 | 引用字符串 |
| *filename* | 用来转换错误消息的完整路径名 | 请参阅 拒绝文件 | 必须遵循操作系统规则 |
| *num_errors* | 终止加载操作之前的错误数目 | 除非设置 REJECTFILE 值，否则该值被忽略 | 精确数值 |
| *num_rows* | 外部表中包含的行的近似数目 | 必须为正数 | 精确数值 |
| *record_delimiter* | 用来分隔记录的字符。缺省为换行符 | 对于非打印字符，使用八进制 | 引用字符串 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | （ \n ） | 符号 | |

***num_errors*** 规范在 unload 任务期间被忽略。

如果未指定 RECORDEND 值，***record_delimiter*** 缺省值为换行符（\n）。要将非打印字符指定为记录定界符或字段定界符，您必须将它编码为 ASCII 字符的八进制表示法。例如 \006 可表示 CTRL-F 。

在 Windows™ 系统上，如果您使用 DB-Access 实用程序或者 dbexport 实用程序加载数据库表到文件中，然后计划使用该文件作为外部表数据文件，则您应当在 CREATE EXTERNAL TABLE 语句中将 RECORDEND 定义为 '\012' 。

按照下表描述使用 table 选项关键字。除非指定了两种方式其中一种，否则只要计划 load 或 unload 数据，就可以使用每个关键字。

**关键字**

**描述**

DBDATE

当读取或写入外部表时指定数据格式。在从外部表加载和卸装操作期间，使用 DBDATE 子句转换数据。在以下示例中，DBDATE 设置为 DMY2-。如果数据库表中的日期值存储为 06/24/2009 ，则该值写入外部表后为 24-06-09 。

```
CREATE EXTERNAL TABLE ext_date (dob date)
        USING (      DATAFILES ("DISK:/tmp/datedisk"),
        REJECTFILE "/tmp/datereject",
        DBDATE "DMY2-",
        FORMAT "delimited");

INSERT INTO ext_date SELECT * FROM basetab;
```

DBDATE 子句还在从外部表表插入数据到数据库表时使用。在以下示例中，外部表中的数据被转换为基于由 CREATE EXTERNAL TABLE 语句设置的 DBDATE 值的内部二进制格式。

```
INSERT INTO basetab SELECT * FROM ext_date;
```

如果在 CREATE EXTERNAL TABLE 的 USING 子句中未指定 DBDATE 关键字，则日期格式由 DBDATE 环境变量的设置决定。如果未指定 DBDATE 环境变量，则日期格式由 GL_DATE 环境变量的设置决定。DBDATE 子句指定的值优先于 DBDATE 环境变量指定的值。DBDATE 变量设置的值优先于 GL_DATE 环境变量。有关 DBDATE 和 GL_DATE 值的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

DBMONEY

当正在读取或写入外部表时，指定货币格式。在从外部表加载和卸装操作期间，使用 DBMONEY 子句转换数据。在以下示例中，DBMONEY 设置为 DM, 。货币格式化为 DM（德国马克）单位，使

用货币符号 DM 和逗号（,）。如果在数据库表中的存储的货币值为 100.50，则写入外部表的值为 100,50 。

```
CREATE EXTERNAL TABLE ext_money (sales money)
        USING ( DATAFILES ( "DISK:/tmp/moneydisk" ),
        REJECTFILE "/tmp/moneyreject",
        DBMONEY "DM,",
        FORMAT "delimited");

INSERT INTO ext_money SELECT * FROM basetab;
```

当将外部表中的数据读到数据库表中时，外部表不需要货币符号。例如，如果外部表包含值 1000,78 且 DBMONEY 设置为 DM, ，则该数据不会被拒绝且该行会正确存储。

如果十进制符号在外部表中且设置的 DBMONEY 值不符合，那么该行被拒绝。例如：如果外部表包含值 1000,78 （使用逗号代替小数点）且将 DBMONEY 子句设置为 DM.，则该行被拒绝。如果数据文件包含一个货币符号并且该货币符号不符合 DBMONEY 货币符号，则该行被拒绝。

当从数据库表写入数据到外部表时，货币符号不会写入到外部表中。

如果没有指定 DBMONEY 子句，则通过设置 DBMONEY 环境变量决定数据格式。DBMONEY 子句指定的值优先于 DBMONEY 环境变量指定的值。如果没有指定 DBMONEY 子句且也没有设置 DBMONEY 环境变量，则使用数据库语言环境指定的十进制分隔符。有关 DBMONEY 值的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

DEFAULT (load only)

指定用列缺省值（如果已定义）而不是 NULL 值来替换定界输入文件中缺少的值，这样就可以稀疏地填充输入文件。在缺省值是要装入的值的文件中，文件不需要每一列都有一个条目。

DELIMITED

指定数据文件是一个定界文本文件。可以使用可选的 DELIMITER 表选项指定定界符。

DELIMITER

指定在定界文本文件分隔字段的字符。如果表选项不包含 DELIMITER 规范，则管道符（|）是缺省的字符分隔符。

DELUXE (load only)

请求使用 DELUXE 方式加载数据（而不是 EXPRESS 方式）时，数据库服务器会忽略该关键字，并内部选择 DELUXE 或 EXPRESS 方式。如果您指定 DELUXE 关键字，但数据库服务器内部选择 EXPRESS 方式，则向联机日志中写入一条警告："Switching load on target table ***<owner>.<table>*** to EXPRESS".

DELUXE 方式将更改索引、执行约束检查以及评估触发器的数据插入到该表。DELUXE 方式加载不如 EXPRESS 方式加载快，但是它更灵活。在 DELUXE 方式，您可以存取并更改正在加载的表。

数据库服务器在加载数据到使用事务日志记录的数据库的 STANDARD 表和任何定义索引的表中时，总是选择 DELUXE 方式。

ESCAPE

在 DELIMITER 指定任何 ***field_delimiter*** 分隔符的实例之前，立即插入缺省转义字符，该字符是数据的字面值而不是分隔符。无论您包含或者省略该 ESCAPE 关键字，缺省启用该功能，或者您可以指定 ESCAPE ON 关键字使它清晰地认识到人类读者的您启用了此功能的 SQL 代码。要阻止数据中的 ***field_delimiter*** 字面分隔符被转义，您必须指定 ESCAPE OFF 关键字。

缺省情况下，在 ***field_delimiter*** 字符之前 ESCAPE 关键字插入的转义字符是反斜杠（\）。但是如果 DEFAULTESCCHAR 配置参数设置为单符号值，则当指定了 ESCAPE 或 ESCAPE ON 后，用此符号代理反斜杠（\）作为定界符。

**注：**

在 GBase 8s 中 ESCAPE 缺省的设置是 OFF 。

EXPRESS (load only)

请求使用 EXPRESS 方式加载数据（而不是 DELUXE 方式）时，数据库服务器会忽略该关键字，并内部选择 DELUXE 或 EXPRESS 方式。如果您指定 EXPRESS 关键字，但数据库服务器内部选择 DELUXE 方式，则向联机日志中写入一条警告："Switching load on target table ***<owner>.<table>*** to DELUXE" 。

数据库服务器仅在以下情景中内部地选择 EXPRESS 方式：

- 数据库是非日志记录的且目标表（任一表类型）没有索引。
- 数据库是日志记录的且目标表是 RAW 并没有索引。

对于其它情况，数据库服务器内部选择 DELUXE 方式。

EXPRESS 方式加载（使用轻量级追加）比 DELUXE 方式加载明显快很多，但是不太灵活。在 EXPRESS 方式，直到加载完毕，您才能更新该表或读取新数据项。

如果指定 EXPRESS 方式，而表 BLOB 、BYTE 、CLOB 或 TEXT 类型的对象，则加载停止，产生错误消息。

当使用 EXPRESS 方式加载数据时，目标表不能位于 Enterprise Replication (ER) 复制中。此外，目标数据库服务器必须没有启用高可用数据复制（HDR）。

FIXED

指定数据文件是固定宽度的。当在外部表中使用 EXTERNAL 数据类型时，必须使用 FIXED 格式。

FORMAT

指定数据文件中数据的格式。

GBASEDBT

指定数据文件的格式是内部 GBase 8s 格式。从以 GBase 8s 格式保存的外部表加载数据比从固定或者定界外部文件中加载数据的速度快。当从一个 GBase 8s 数据库移动到另一个数据库时，使用 GBase 8s 格式。

MAXERRORS

设置在数据库服务器停止加载数据之前所允许的错误数目。

MAXERRORS 的最小值为 1 。将 MAXERRORS 值设置为小于 1 的值会产生错误。MAXERRORS 的最大值为 2,147,483,647 。

RECORDEND

指定已定界的文本文件中分隔记录的字符。

REJECTFILE

设置数据库服务器写入数据转换错误的完整路径名。如果未指定或者文件不能打开，则任何错误均会异常结束装入作业。另见拒绝文件 。

NUMROWS *或* SIZE

外部表中的大约行数。

在连接查询中使用外部表时，指定 NUMROWS（或者它的同义词 SIZE）可以提高性能。该值不能为 NULL 。

## 拒绝文件

在加载期间发生转换错误的行或违反外部表上检查约束的行将写入到拒绝文件中。REJECTFILE 子句声明了该拒绝文件的路径和文件名称。

如果您在同一会话中对同一个表执行另一个装入，则任何先前相同名称的拒绝文件将被覆盖。

拒绝文件条目具有以下格式：

***filename, record, reason-code,***

***field-name: bad-line***

下表描述拒绝文件中的这些元素：

| **元素** | **描述** |
| --- | --- |
| ***filename*** | 输入文件的名称 |
| ***record*** | 输入文件中记录号，在该文件中检测到错误 |
| ***reason-code*** | 错误的描述 |
| ***field-name*** | 外部字段名称，其中行中发生第一个错误；如果拒绝不是特定于某个列，则为 <none> |

***bad-line*** 造成错误的行（仅限于定界或固定位置字符文件）

拒绝文件以 ASCII 格式写 *filename*, *record* 、*field-name* 和 *reason-code*。*bad-line* 信息随输入文件类型不同而变化。

- 对于定界文件或固定位置字符文件，错误行有多达 80 个字符被直接复制到拒绝文件中。
- 对于 GBase 8s 内部数据文件，错误行选项不放在拒绝文件中，因为您不能在文件中编辑二进制表示法；但 ***filename*** 、***record*** 、***reason-code*** 和 ***field-name*** 仍在拒绝文件中报告，这样您可以隔离该问题。请使用 Table Options 子句指定外部数据的格式。

以下错误可导致行被拒绝。

| 错误文本 | 说明 |
|---|---|
| CONSTRAINT ***constraint name*** | 违反了该约束。 |
| CONVERT_ERR | 有字段遇到转换错误。 |
| MISSING_DELIMITER | 找不到定界符。 |
| MISSING_RECORDEND | 找不到记录结尾。 |
| NOT NULL | 在 ***field-name*** 中找到 NULL 值。 |
| ROW_TOO_LONG | 输入记录超过 32 千字节。 |

**虚拟处理器**

GBase 8s 外部表使用 FIFO 虚拟处理器。当初始化服务器时，创建一个 FIFO 虚拟处理器。可使用 gadmin -p 命令添加其它 FIFO 虚拟处理器。例如，使用以下语句添加三个 FIFO 虚拟处理器：

```
gadmin -p +3 fifo
```

无法删除 FIFO 虚拟处理器。

FIFO 虚拟处理器用来处理与使用 PIPE 子句定义的管道相关的 I/O 。

有关使用 FIFO 虚拟处理器的更多信息，请参阅 *GBase 8s 管理员指南*。

## 外部表示例

本节中的示例显示使用外部表加载和卸装数据的不同方法。

以下是 CREATE EXTERNAL TABLE 语法的示例。在本示例中，创建带有两列且名为 empdata 的外部表。DATAFILES 子句指示数据文件的位置，指定该文件是定界的，指示拒绝文件的位置并指定该拒绝文件包含的错误不能多于 100 个。

```
CREATE EXTERNAL TABLE empdata
        (
        empname     char(40),
        empdoj   date
        )
        USING
        (DATAFILES
        (
```

```
        "DISK:/work/empdata.unl"
        ),
        FORMAT "DELIMITED",
        REJECTFILE "/work/errlog/empdata.rej",
        MAXERRORS 100);
```

**使用 SAMEAS 子句创建外部表**

SAMEAS ***template*** 子句在新表的定义中使用来自 ***template*** 表所有的列名称和数据类型。以下示例使用 empdata 表的列名及其数据类型来创建外部表：

```
CREATE EXTERNAL TABLE emp_ext SAMEAS empdata
        USING
        (DATAFILES
        (
        "DISK:/work/empdata2.unl"
        ),
        REJECTFILE "/work/errlog/empdata2.rej",
        DELUXE
        );
```

**向外部表中卸装数据**

以下示例显示将数据库表的数据加载到外部表所使用的语句。

```
CREATE EXTERNAL TABLE ext1( col1 int )
        USING
        (DATAFILES
        (
        "DISK:/tmp/ext1.unl"
        )
        );

        CREATE TABLE base (col1 int);
        INSERT INTO ext1 SELECT * FROM base;
```

还可以按照以下示例使用 SELECT...INTO EXTERNAL 语法卸装数据：

```
SELECT * FROM base
        INTO EXTERNAL emp_target
        USING
        (DATAFILES
        (
        "DISK:/tmp/ext1.unl"
        )
        );
```

**从外部表查询数据将其加载到数据库表**

以下示例从外部表查询，且显示了将查询到的数据加载到数据库表的各种方法。

```
CREATE EXTERNAL TABLE ext1( col1 int )
        USING
        (DATAFILES
        (
        "DISK:/tmp/ext1.unl"
        )
        );

        CREATE TABLE target1 (col1 int);
        CREATE TABLE target2 (col1 serial8, col2 int);

        SELECT * FROM ext1;
        SELECT col1,COUNT(*) FROM ext1 GROUP BY 1;
        SELECT MAX(col1) FROM ext1;
        SELECT col1 FROM ext1 a, systables b WHERE a.col1=b.tabid;

        INSERT INTO target1 SELECT * FROM ext1;
        INSERT INTO target2 SELECT 0,* FROM ext1;
```

**从数据库表中的将数据卸装到一个使用 FIXED 格式的文本文件中**

以下示例创建了一个名为 emp_ext 的外部表，定义了列名及其数据类型，并从使用固定格式的数据库卸装数据。

```
CREATE EXTERNAL TABLE emp_ext
        ( name CHAR(18) EXTERNAL CHAR(20),
        address VARCHAR(40) EXTERNAL CHAR(40),
        empno INTEGER EXTERNAL CHAR(6)
        )
        USING (
        FORMAT 'FIXED',
        DATAFILES
        (
        "DISK:/work2/mydir/emp.fix"
        )
        );

INSERT INTO emp_ext SELECT * FROM employee;
```

**从数据文件将数据加载到使用 FIXED 格式的数据库表中**

下一示例创建了名为 emp_ext 的外部表，并从固定格式文件将数据加载到数据库中。

```
CREATE EXTERNAL TABLE emp_ext
        ( name CHAR(18) EXTERNAL CHAR(18),
        address VARCHAR(40) EXTERNAL CHAR(40),
        empno INTEGER EXTERNAL CHAR(6)
        )
        USING (
```

```
FORMAT 'FIXED',
DATAFILES
(
"DISK:/work2/mydir/emp.fix"
)
);

INSERT INTO employee SELECT * FROM emp_ext;
```

**在 DATAFILES 子句中使用格式化字符**

要处理三个文件，请按照以下示例创建 DATAFILES 子句。

```
DATAFILES
        (
        "DISK:/work2/extern.dir/mytbl.%r(1..3)"
        )
```

下列显示了当运行语句时列表是如何展开的：

```
DATAFILES
        (
        "DISK:/work2/extern.dir/mytbl.1",
        "DISK:/work2/extern.dir/mytbl.2",
        "DISK:/work2/extern.dir/mytbl.3"
        )
```

## 将外部表中的加载到 GBase 8s

要加载数据，请定义该外部数据为外部表然后将数据插入到数据库中。

数据库服务器执行 express 方式加载和 deluxe 方式加载。您只能当表类型是 RAW 且不拥有活动的索引时，才能执行 express 方式加载。数据库服务器对这两种加载方式都允许约束检查。

Express 方式在加载期间提供最高的性能。

Deluxe 方式将快速并行加载和索引及唯一约束的评估结合，并且在以下情况中，更有效率：

- 对于您正在加载的数据量，重建索引的开销太高。
- 您想要使用从您正在加载表中的已删除行的空的空间。

如果接收外部表的行的表是 STANDARD 表（即，不是由 CREATE TEMP TABLE 或 CREATE RAW TABLE 语句创建的数据库表），则 EXPRESS 关键字没有作用，且该表以 DELUXE 方式加载。当数据库服务器忽略加载操作中的 EXPRESS 关键字时（此处正在接收数据的表不是 RAW 表），它不会发出异常。

### *以 Express 方式加载数据*

Express® 方式支持数据快递加载到没有索引的表中。在日志记录的数据库中，只有 RAW 表能使用此方式。

**警告：** 支持事务日志记录的数据库中的 STANDARD 表不允许 Express 方式加载。

**注：**

Express 方式加载使用轻量级追加，它绕过缓冲池。轻量级追加消除与缓冲管理器相关的开销但不日志记录数据。在 express 方式，数据库服务器会自动排他锁定该表。其它用户不能存取该表。

无论您是否使用 DELUXE 关键字，数据库服务器使用 express 方式，除非目标表有索引或者目标表是 STANDARD 表。

如果您定义了 RAW 类型表并在加载数据后没有定义任何索引，则可以对此没有数据的新建的表使用 express 方式。如果您不希望在支持事务日志记录的数据库中使用，请选择 RAW 表。

要为现有的表准备 express 方式加载，请删除所有的索引，并确保表类型是 RAW 。

从外部表加载数据到 raw 表不会被日志记录；因此，您必须在删除数据库之前执行零级备份。如果您在执行零级备份之前尝试删除数据库，则数据库服务器发出 ISAM 错误 error -197，如下所示：

Partition recently appended to; can't open for write or logging

考虑具有以下结构的表：

```
TABLE employee (
        name CHAR(18),
        hiredate DATE,
        address CHAR(40),
        empno INTEGER);
```

在现有表上使用 express 方式加载

1. 更改表类型从而允许快速加载。

   ```
   ALTER TABLE employee TYPE (RAW);
   ```

2. 创建外部表描述。

   ```
   CREATE EXTERNAL TABLE emp_ext
           SAMEAS employee
           USING (
           FORMAT 'DELIMITED',
           DATAFILES
           ("DISK:/work2/mydir/emp.dat"),
           REJECTFILE "/work2/mydir/emp.rej",
           EXPRESS
           );
   ```

3. 加载该表。

   ```
   INSERT INTO employee SELECT * FROM emp_ext;
   ```

   如果数据库服务器选择 express 方式，当目标表包含索引、约束或者任意其它问题条件时，加载停止并发出错误消息。

4. 创建零级备份。

   因为数据是未日志记录的，您必须执行零级备份以允许数据复原。如果磁盘失败，则您无法自动恢复数据。需要使用最近的零级备份文件。

如果表类型是 RAW （未日志记录），则忽略 BEGIN WORK 和 COMMIT WORK 语句。

**注：** 如果您删除表中的很多行，然后以 EXPRESS 方式向表加载很多新行，则该表大小增加，因为轻量级追加在表的末尾插入行，并且它不拒绝表内的空白。（不管您是否指定 EXPRESS 方式，如果表由很多已删除的行，则该加载可能选择 DELUXE 方式填充到该空间。）

## *以 DELUXE 方式加载数据*

DELUXE 方式将快速并行加载与索引评估和唯一约束评估结合起来。数据库服务器对所有数据库中的被索引的目标表和在日志记录的数据库中 STANDARD 目标表的选择这种方式。

**注：**

DELUXE 方式通常使用单行插入，可以向包含索引的表添加行。在加载期间，该插入修改每行的每个索引。该插入还会检查每一行的约束。DELUXE 方式加载允许您在加载期间将表保持为未锁定的状态，因此其它用户可以继续使用它。

您还可以在不包含索引的表上使用 DELUXE 方式；例如，您希望在加载期间可以完成恢复或持续存取表。

要准备表以 DELUXE 方式加载，请创建内部表的类型为 STANDARD ，并用 DELUXE 关键字创建内部表。

在表上使用 DELUXE 方式加载：

1. 如果您希望行锁定，请在 CREATE TABLE 语句中指定行锁定。（缺省情况下时页面锁定。）如果您想要其它用户在加载期间可以读取该表，那么将锁定方式设置为 `share` 。否则，设置为 `exclusive` 。

   ```
   BEGIN WORK;
   LOCK TABLE employee IN SHARE MODE;
   ```

2. 定义外部表。

   ```
   CREATE EXTERNAL TABLE emp_ext
               SAMEAS employee
               USING (
               DATAFILES ("DISK:/work2/mydir/emp.dat"),
               REJECTFILE "/work2/mydir/emp.rej",
               DELUXE
               );
   ```

3. 加载该表。

   ```
   INSERT INTO employee SELECT * FROM emp_ext;
   ```

4. 提交该加载，释放行或页面锁定。

   ```
   COMMIT WORK;
   ```

**重要：** 配置逻辑日志以允许最大并发 DELUXE 加载事务完成。

## *使用相同的结构将定界文件中的数据加载到数据库表中*

如果外部表与数据库表具有相同的结构，则可以不用定义外部表的结构。

考虑将定界 ASCII 文本文件加载到具有以下结构的表中：

```
TABLE employee (
        name CHAR(18) NOT NULL,
        hiredate DATE DEFAULT TODAY,
        address VARCHAR(40),
        empno INTEGER);
```

以下 SQL 语句可以用来将数据加载到 employee 表中：

```
CREATE EXTERNAL TABLE emp_ext
            SAMEAS employee
            USING (
            DATAFILES ("DISK:/work2/mydir/emp.dat"),
            REJECTFILE "/work2/mydir/emp.rej"
            );
            INSERT INTO employee SELECT * FROM emp_ext;
```

外部表的每一列具有与源表列相同的名称、类型和缺省值，因为 CREATE 语句包含了 SAMEAS 关键字。缺省格式为定界，因此不需要格式关键字。

定界文件缺省为 ASCII 。除非您在创建外部表时使用 RECORDEND 关键字定义不同的定界符，否则缺省的行定界符是行结束符。（RECORDEND 关键字仅对定界格式有效。）

## *从固定文本文件加载*

固定文本文件是一种数据驻留在文件中固定位置的文件。

以下 SQL 语句将 emp_exp 外部表中的数据加载到固定位置表（employee）：

```
CREATE EXTERNAL TABLE emp_ext
        ( name CHAR(18) EXTERNAL CHAR(18),
        hiredate DATE EXTERNAL CHAR(10),
        address VARCHAR(40) EXTERNAL CHAR(40),
        empno INTEGER EXTERNAL CHAR(6) )
        USING (
        FORMAT 'FIXED',
        DATAFILES ("DISK:/work2/mydir/emp.fix")
        );
        INSERT INTO employee SELECT * FROM emp_ext;
```

枚举的列使用 EXTERNAL 关键字描述在外部文件中存储数据的格式。

## *在拥有相同结构的表之间加载数据*

如果表具有相同的结构，则您能轻松地将外部表的数据移动到数据库表。

您可以使用简单的 INSERT 语句将一个表的数据加载到另一个具有相同结构的表中（例如：worldemp）。

INSERT INTO worldemp SELECT * FROM emp_ext;

### *将值装入 Serial 列*

您可以在在 Serial 列中插入连续数字或显式值。

数据库服务器将使用原始数据文件中的值或数据库服务器自动生成的值装入到 Serial 列。

如果您希望该 serial 列值为来自数据文件的值，那么 INSERT 语句不需要进行 special 处理。如果您希望数据库服务器自动生成该值，那么忽略来自 INSERT 语句的 serial 列。例如：在表 col1 中第一列是 serial 列，使用以下语句，缺省机制提供 serial 值：

INSERT INTO ***mytable*** (***col2***, ...) SELECT ...

如果该表被装入到多个分区，则 serial 值的以与表分片相同的顺序递增。

### *加载数据仓库表*

对数据仓库应用程序，您可以使用外部表加载每一个大表。

本节讨论加载非常大的表的各种场景：

- 最初加载
- 定期刷新
- 从数据库服务而不是 GBase 8s 加载 OLTP 数据

**最初加载**

以下场景使用外部表创建并加载一个数据仓库表。

最初加载表

1. 创建 RAW 类型的表以便从轻量级追加中获利并避免在加载期间日志记录的开销。

   CREATE RAW TABLE tab1 ...

2. 对数据库服务器使用 CREATE EXTERNAL TABLE 语句描述外部数据文件，在 USING 子句中指定 EXPRESS 语句。

3. 加载该表。

   INSERT INTO tab1 SELECT * FROM ext_tab

   该表加载很快，并且该操作用了非常小的日志空间。

4. 验证数据的完整性。

5. 在该表上创建索引以致于查询运行更快。

6. 如果必要，执行零级备份以便您之后可以恢复该表。如果在有问题的情况下，从原始源重新加载表也很容易的话，则不需要执行此零级备份。

**定期刷新**

该场景定期从其它源加载新数据到数据仓库表。

该场景假设在正常操作期间该表的类型是 STANDARD ，之前执行过 CREATE EXTERNAL TABLE 语句，并且在 USING 子句中指定了 EXPRESS 关键字。

要定期刷新表

1. 删除表上所有的索引。
2. 更改表类型为 RAW。

   ALTER TABLE tab1 TYPE(RAW);
3. 向表加载新数据。

   INSERT INTO tab1 SELECT * FROM ext_tab
   此 insert 语句很快在该表的末尾追加了新数据，且该操作用了很小的日志空间。
4. 验证数据的完整性。
5. 更改表类型为 STANDARD 。

   ALTER TABLE tab1 TYPE(STANDARD);
6. 重新创建表上的索引以致于查询运行更快。
7. 如果必要，执行零级备份以便您之后可以恢复该表。如果在有问题的情况下，从原始源重新加载表也很容易的话，则不需要执行此零级备份。

**从其它数据库服务器初始加载 OLTP 数据**

该场景是首次加载数据到 GBase 8s 中，当您从另一个数据库服务迁移是，您可能会执行此操作。

在此场景。要加载的表将被用于 OLTP，因此您需要日志记录事务、回滚和恢复能力。

使用 CREATE EXTERNAL TABLE 语句从不同的数据服务器使用最初加载 OLTP 数据：

1. 创建 RAW 类型的表以便从轻量级追加中获利并避免在加载期间日志记录的开销。

   CREATE RAW TABLE tab1 ...
2. 对数据库服务器使用 CREATE EXTERNAL TABLE 语句描述外部数据文件，在 USING 子句中指定 EXPRESS 语句。
3. 加载该表。

   INSERT INTO tab1 SELECT * FROM ext_tab
   该表加载很快，并且该操作用了非常小的日志空间。
4. 验证数据的完整性。
5. 执行零级备份，提供一个恢复点。

6. 更改表的类型为 STANDARD。

   ALTER TABLE tab1 TYPE(STANDARD);

7. 在表上创建的索引以便查询运行更快。

8. 启用表上的约束以保留数据的完整性。

## 将数据从 GBase 8s 卸装到外部表

通过创建外部表并将数据插入到该表中，或者从内部表查询数据并将数据插入到外部文件，来卸装数据。

要并行地卸装数据，启动并行运行的查询，并将其输出结果写到多个文件中。该卸装任务使用循环技术均衡输出文件汇总的行数。

### *卸载到定界文件*

您可以将表中的数据卸载到定界的 ASCII 文本文件中，如下所示：

```
CREATE EXTERNAL TABLE emp_ext
        SAMEAS employee
        USING (
        DATAFILES ("DISK:/work2/mydir/emp.dat")
        );
        INSERT INTO emp_ext SELECT * FROM employee;
```

定界文件缺省为 ASCII 。

### *卸载到 GBase 8s 数据文件*

要将 employee 表卸载到 GBase 8s 内部格式的表，请使用类似的下列语句：

```
SELECT * FROM employee
        WHERE hiredate > "1/1/1996"
        INTO EXTERNAL emp_ext
        USING (
        FORMAT 'GBASEDBT',
        DATAFILES ("DISK:/work2/mydir/emp.dat")
        );
```

因为输出文件使用 GBase 8s 内部表示法，所以您需要在 USING 子句中指定 FORMAT 'GBASEDBT' 选项。（缺省为定界 ASCII 格式。）

### *卸载到固定文本文件*

可将数据库的数据卸载到固定格式文件。

以下 SQL 语句将以固定文本格式的 employee 表卸载到 emp_ext 外部表中：

```
CREATE EXTERNAL TABLE emp_ext
        ( name CHAR(18) EXTERNAL CHAR(20),
```

```
hiredate DATE EXTERNAL CHAR(10),
address VARCHAR(40) EXTERNAL CHAR(40),
empno INTEGER EXTERNAL CHAR(6) )
USING (
FORMAT 'FIXED',
DATAFILES ("DISK:/work2/mydir/emp.fix")
);
INSERT INTO emp_ext SELECT * FROM employee;
```

这些语句创建了有 20 个字符位置的第一个字段，10 个字符位置的第二字段等等的固定文本文件。因为通过 SELECT 语句选择行，所以您可以任何您希望的方式格式化该 SELECT 列表。

## *向固定文本文件添加行结束符*

您可以向固定文本文件的每行添加行结束符，以将文件用于其它应用程序。

如果您以固定文本格式写文本，一行写一个记录是有帮助的。行结束符使得数据更清晰易读。如果您使用缺省的定界格式，则会自动添加行结束符。然而，对于固定格式卸载，需要在您的记录中添加行结束符。例如：考虑具有以下结构的表：

```
TABLE sample (
        lastname CHAR(10),
        firstname CHAR(10),
        dateofbirth DATE);
```

该表包含以下值：

```
Adams       Sam         10-02-1957
Smith       John        01-01-1920
```

接下来，考虑以下结构的外部表：

```
CREATE EXTERNAL TABLE sample_ext (
                lastname CHAR(10) EXTERNAL CHAR(10),
                firstname CHAR(10) EXTERNAL CHAR(10),
                dateofbirth DATE EXTERNAL CHAR(12));
```

不带行结束符卸载 sample_ext 会产生以下输出：

```
Adams       Sam         10-02-1957  Smith       John        01-01-1920
```

您可以通过使用程序或脚本添加行结束符，或者在 SELECT 语句中添加换行字段。

### 使用程序或脚本

要添加行结束符，您可以将固定长度的记录写到数据文件中，然后用程序或脚本修改该数据文件。

例如：您可以用 C 程序查找每一记录的长度，定位每一行的末尾，然后添加行结束符。

### 在 SELECT 语句中添加换行字段

您可以使用外部表卸载在您内部表中的换行符。

要添加行结束符，请从含有换行符的表中查询最终值，如下所示：

1. 创建一个只包含换行字符的文件。

   echo "" > /tmp/cr.fixed

2. 创建一个存储该换行值的内部表，在卸载数据时使用该表。

   CREATE TABLE dummyCr (cr CHAR(1));

3. 创建外部表以卸载换行值。

   CREATE EXTERNAL TABLE x_cr (cr CHAR(1) EXTERNAL CHAR(1))

   USING (DATAFILES ("DISK:/tmp/cr.fixed"), FORMAT 'FIXED');

4. 加载该外部表到内部表 **dummyCr** 中。

   INSERT INTO dummyCr SELECT * FROM x_cr;

该内部表，**dummyCr** ，现在包含一个行结束符，您可以在 SELECT 语句中既使用它进行卸载。

1. 要将内部表的数据卸载到外部表，请创建带有行结束符 EXTERNAL CHAR 的外部表。

```
CREATE EXTERNAL TABLE sample_ext
        (
        lastname CHAR(10) EXTERNAL CHAR(10),
        firstname CHAR(10) EXTERNAL CHAR(10),
        dateofbirth DATE EXTERNAL CHAR(12),
        eol CHAR(1) EXTERNAL CHAR(1))
        USING (DATAFILES ....), FORMAT 'FIXED');
```

2. 将从内部表和 **dummyCr** 表的查询用于创建拥有被行结束符分隔的行的输出文件。

```
INSERT INTO sample_ext(lastname, firstname, dateofbirth, eol)
            SELECT a.lastname, a.firstname, a.dateofbirth, b.cr
            FROM mytable a, dummyCr b;
```

## 外部表的限制

确定外部表上不支持或具有限制范围的操作。

表 1 比较了支持数据库表和外部表的表操作。

| 表操作 | 数据库表 | 外部表 |
| --- | --- | --- |
| 支持索引和：<br>• 主键<br>• 外键<br>• 唯一和非唯一索引<br>• 索引扫描<br>• 在执行查询时自动索引（autoindex）<br>• 索引连接 | 是 | 否，使用顺序扫描 |

| 表操作 | 数据库表 | 外部表 |
| --- | --- | --- |
| 支持触发器 | 是 | 否 |
| MERGE 语句中表可以是目标 | 是 | 否。不允许作为目标允许作为源。请参阅 MERGE 示例 |
| 支持表分片 | 是 | 否 |
| 在 FROM 子句中允许多个数据库表 | 是 | 否。请参阅 查询示例 |
| 支持 DB-Access LOAD FROM ... INSERT INTO 语句 | 是 | 否 |
| 清除表的 TRUNCATE TABLE 语句 | 是 | 否。使用 TRUNCATE 语句不会清除外部表中的数据。将数据库表中的数据卸载到外部表时会自动删除此外部表。 |
| 复制表数据 | 是 | 否 |
| 支持 UPDATE STATISTICS 语句 | 是 | 否 |
| 支持 UPDATE 和 DELETE 语句 | 是 | 否 |
| 支持 ALTER TABLE 语句 | 是 | 否 |
| 支持 LBAC | 是 | 否 |
| 支持压缩 | 是 | 否 |
| 支持 START 和 STOP VIOLATIONS 语句 | 是 | 否 |
| 支持 TEMP 表 | 是 | 否 |
| 表列支持 EXTERNAL 数据类型 | 否 | 是 |
| 支持 DEFAULT 子句 | 是 | 否 |
| 支持 BLOB 和 CLOB 类型的 PUT 子句 | 是 | 否。可以使用 DATAFILES 指定 BLOBDIR 和 CLOBDIR。 |
| SERIAL 、SERIAL8 和 BIGSERIAL 数据类型生成的 | 是 | 否。这些数据类型被转换为等价的整型且不会生成 |

| 表操作 | 数据库表 | 外部表 |
| --- | --- | --- |
| serial 数 | | serial 值 |
| 使用 Enterprise replication （ER）时可以复制表 | 是 | 否 |
| 表的更改可被日志记录且可以复制 | 是 | 否。外部表是非日志记录的且不能被复制；然而系统目录表可以被复制。 |
| 支持 ACID（原子性、一致性、隔离性、持久性）属性 | 是 | 否 |
| 支持 ETL （提取、转换、加载） | SQL 接口 不支持 ETL 操作的；然而，例如 dbload 、gload 、gunload 实用程序和 LOAD 、UNLOAD 语句支持。 | 支持使用简单的 SQL 接口，对高性能的数据加载和卸载，使用 INSERT ... SELECT 语句。 |

确定不支持高可用集群操作（请参阅 *GBase 8s 管理员指南* 中的***高可用集群环境中的外部表***）。

要从外部表加载 BLOB 或者 CLOB 对象，您必须重建临时 sbspace ，并在那个空间中创建临时智能大对象来存储来自外部表 BLOB 或 CLOB 数据。不支持从只读的辅助服务器加载 BLOB 或 CLOB 数据，因为您不能在只读的辅助服务器上创建临时智能大对象。

**MERGE 示例**

外部表不能是 MERGE 语句中的目标。例如：如果 **ext** 是一个外部表，以下将 **ext** 作为源表的 MERGE 语句是有效的：

```
MERGE INTO t1
USING ext ON t1.c1 = ext.c1
      WHEN MATCHED THEN UPDATE
        SET t1.c2 = ext.c2
WHEN NOT MATCHED THEN INSERT VALUES (99, '999');
```

然而，以下语句因为把 **ext** 作为目标表而失败：

```
 MERGE INTO    ext
 USING t1 ON ext.c1 = t1.c1
   WHEN MATCHED THEN UPDATE
      SET ext.c2 = t1.c2
   WHEN NOT MATCHED THEN INSERT VALUES (99, '999');
```

**查询示例**

只有最外层的查询才能有外部表引用。在任何查询中只允许一个外部表。例如：以下语句是允许的：

```
SELECT * FROM ext, t2 WHERE ext.c1 = t2.c1;
```

然而，以下语句是不允许的：

- 一个查询中不能指定多个外部表：

  ```
  SELECT * FROM ext, ext3 WHERE ext.c1 = ext3.c1;
  ```

- 在子查询中不能使用外部表：

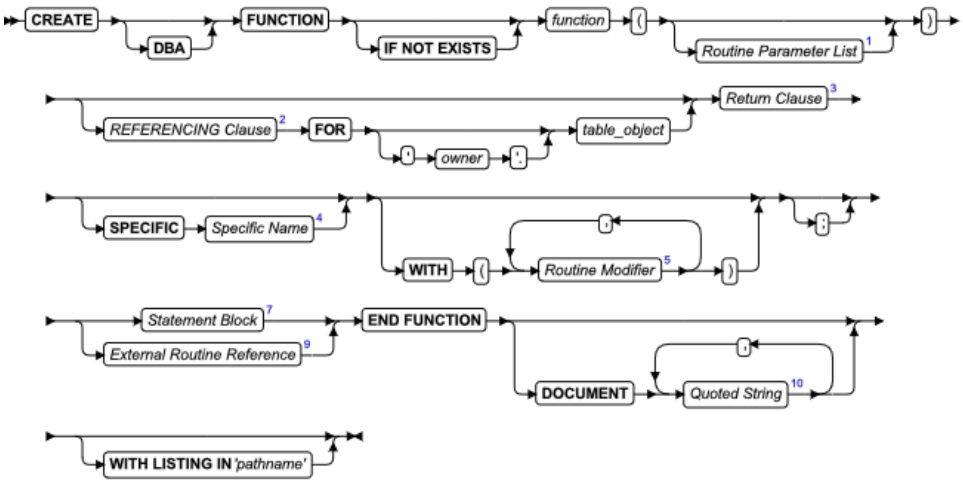
  ```
  SELECT * FROM t1 WHERE t1.c1   IN (SELECT c1 FROM ext);
  ```

## 2.28 CREATE FUNCTION 语句

使用 CREATE FUNCTION 语句创建用户定义函数，注册外部函数，编写并注册 SPL 函数。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 在此定义的函数的名称 | 您必须拥有相应的语言特权，请参阅 GRANT 语句 和重载函数名 | 标识符 |
| *owner* | *table_object* 的所有者 | 必须拥有 *table_object* | 所有者名称 |
| *pathname* | 存储编译事件警告的文件的路径名 | 指定的路径名必须存在于数据库所驻留的计算机之上 | 路径和文件名必须符合您的操作系统规则 |
| *table_object* | 具有可以调用 *function* 的触发器的表或视图的名称或同 | 在本地数据库中必须存在 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 义词 | | |

**提示：**如果您尝试从独立文件中的源代码文本创建函数，则使用 CREATE FUNCTION FROM 语句。

**用法**

GBase 8s 支持这些语言编写的用户定义的函数：

- GBase 8s 存储过程语言（SPL）。
- GBase 8s 支持的外部语言之一（C 或 Java™）（*外部函数*）。

当 IFX_EXTEND_ROLE 配置参数设置成 ON 时，只有 DBSA 授予内置 EXTEND 角色的用户才可以创建外部函数。使用 CREATE FUNCTION 语句的其它要求在 使用 CREATE FUNCTION 时必需的特权 中标识。

一个函数可以返回多少值取决于语言。用 SPL 写的函数可以返回一个或多个值。用 C 或 Java 语言写的外部函数必须只返回一个值。但是 C 函数可以返回一个集合类型，而查询的外部函数可以从 OUT 参数（对于 SPL 和 Java 语言，从 INOUT 参数）， GBase 8s 可以将这些参数作为语句-局部变量（SLV）处理。

SPL 函数的 OUT 和 INOUT 参数的返回值可以作为 SLV 处理。您还可以使用 SPL 例程的局部变量或参数从拥有 OUT 或 INOUT 参数的 SPL 或 C 例程检索值。

有关该手册如何使用术语 UDR 、函数和过程以及建议用法的信息，请分别参阅 例程、函数和过程之间的关系 和 使用 CREATE PROCEDURE 与 CREATE FUNCTION 的对比 。

在 ESQL/C 中，只能在 PREPARE 语句中使用 CREATE FUNCTION 语句。如果您想要创建用户定义函数(在编译时按该函数识别文本)，则必须将文本放在文件中并用 CREATE FUNCTION FROM 语句指定该文件。

如果您包含了 IF NOT EXISTS 关键字，且指定名称的函数已经在当前数据库中注册，那么数据库服务器不采取任何操作（而不是向应用程序发送异常）。（因为函数的标识符可以被重载，所以如果数据库服务器可以解析新函数的参数列表与当前数据库中任何其它同名函数的参数列表不同，则可能有必要包含这些关键字。）

函数使用在其创建时生效的排列顺序。请参阅 SET COLLATION 语句 获取关于非缺省对照的信息。

## 使用 CREATE FUNCTION 时必需的特权

必需拥有数据库上的 Resource 特权或 DBA 特权，才能在该数据库中创建函数。

在创建函数之前，您还必须对要编写的函数的程序语言拥有 Usage 特权。当 GRANT USAGE ON LANGUAGE 语授予一个用户或角色语言级别的特权时，它可以指定 SPL 、C 或 Java™ 语言 。有关更多信息，请参阅 语言级权限 。

缺省情况下，SPL 上的 Usage 特权授予 PUBLIC 。

要以 C 或 Java 外部程序语言中注册函数，除非 IFX_EXTEND_ROLE 配置参数设置成 0 或 Off，否则您还必须持有内置 EXTEND 角色。

## 已创建函数上的 DBA 关键字和 Execute 特权

如果您用 DBA 关键字创建 UDR，则它将被称为 ***DBA 特权 UDR***。您需要 DBA 特权创建 DBA 特权 UDR。

在不拥有 DBA 特权的用户中，只有 DBA 授予 Execute 特权的用户才能调用 DBA 特权 UDR。然而，如果 DBA 授予 PUBLIC Execute 特权，那么所有的用户都可以使用 DBA 特权 UDR。有关 DBA 特权 UDR 的其它信息，请参阅创建数据库对象的所有权。

如果您省略 DBA 关键字，则 UDR 将被称为***拥有者特权***的 UDR。

当您在兼容 ANSI 的数据库中创建拥有者特权 UDR 时，只有您自己能执行该 UDR。它的拥有者必须将 Execute 权限授予个别用户或角色或 PUBLIC 后，其它用户才能执行拥有者特权 UDR。

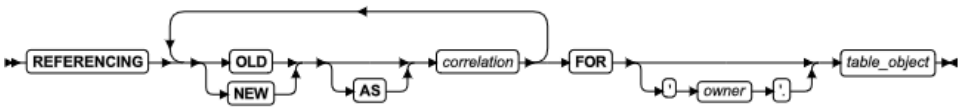
如果您在不兼容 ANSI 的数据库中创建拥有者特权 UDR，则任何人都可以执行该 UDR，因为缺省地将 Execute 权限授予 PUBLIC。要让特定用户才能存取拥有者特权 UDR，则所有者必须撤销 PUBLIC 的对此 UDR Execute 权限，然后将该权限授予指定的用户或角色。将 **NODEFDAC** 环境变量设置为 `yes` 可以防止当以 Onwer 方式创建 UDR 后缺省地将 URD 的权限授予 PUBLIC。如果该环境变量设置成 `yes`，除非，所有者将该 UDR 的 Execute 权限授予其它用户，否则除了该 UDR 所有者其它用户都不能调用此 UDR。

如果外部 C 或 Java™ 语言函数有否定函数，则必须授予对该外部函数及其否定函数的 Execute 权限，然后用户才能执行外部函数。

## REFERENCING 和 FOR 子句

REFERENCING 子句可以声明原始值的相关名称和 FOR 子句指定的 ***table_object*** 列中的已更改的值相关名称。

REFERENCING 和 FOR 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 在此定义的触发器例程中限定旧的或新的列值（*correlation.column*） | 不能是 *table_object* | 标识符 |
| *owner* | *table_object* 的所有者 | 必须拥有 *table_object* | 所有者名称 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *table_object* | 具有可以调用 *function* 的触发器的表或视图的名称或同义词 | 必须存在于本地数据库中 | 标识符 |

如果您在 CREATE FUNCTION 语句列表之后立即包含 REFERENCING 和 FOR ***table_object*** 子句，则您创建的函数被称为***触发器函数***（或 ***触发器 UDR*** 或 ***触发器例程***）。 FOR 子句指定触发器可以从它们的 Triggered Action 列表的 FOR EACH ROW 部分调用函数的表或视图。

在 REFERENCING 子句中，OLD *correlation* 指定一个前缀，通过该前缀触发器例程可以引用 *table_object* 列在触发器例程修改列值之前所具有的值。 NEW correlation 指定用于引用触发器例程分配给该列的新值的前缀。无论该触发器例程是否能使用 correlation 名引用 OLD 列值，NEW 列值或者这两种类型的值都取决于正在触发事件的类型：

- 由 Insert 触发器调用的触发器例程仅能引用 NEW correlation 名称。
- 由 Delete 触发器或 Select 触发器调用的触发器例程仅能引用 OLD correlation 名称。
- 由 Update 触发器调用的触发器例程能引用 OLD 和 NEW correlation 名称。

有关如何在触发动作中使用 *correlation.column* 符号的信息，请参阅 REFERENCING 子句 。

除了任何以 SPL 语言编写 GBase 8s UDR 的一般需求，触发器例程支持某些附加语法特性，并且受到一定的限制，对于不是触发器例程的一般 UDR ，它们不支持此功能（或者不受此限制）：

- 触发器必须包含 FOR ***table_object*** 子句已指定本地数据库中表或视图的名称，该触发器可调用此例程。
- 触发器例程还可以包含 REFERENCING 子句以声明该相关名称为 OLD 和 NEW 值，它们可被 UDR 中的 SPL 语句引用。
- 触发器例程只能在触发器定义中的 Triggered Action 列表的 FOR EACH ROW 节被调用。
- OLD 或 NEW 值的相关变量可以出现在 SPL 的 IF 语句 和 CASE 表达式中。
- OLD 值的相关变量不能在一个 LET 表达式的左边。
- 如果 FOR 子句指定一个视图，它的 INSTEAD OF 触发动作列表调用该触发器例程，则 NEW 值的相关变量不能在一个 LET 表达式的左边。
- 只有 NEW 值的相关变量可以在引用相关变量的 LET 表达式的左边从而。然而，在这种情况下，FOR 子句必须指定表而非视图。
- OLD 和 NEW 值都可以在一个 LET 表达式的右边。
- Boolean 运算符 SELECTING 、INSERTING 、DELETING 和 UPDATING 在具有有效 Boolean 表达式的上下文内的触发器例程中（和只在触发动作语句中调用的触发例程和其它 UDR 中）是可用的。如果正在触发的事件符合由此运算符名称引用的 DML 操作，则这些运算符返回 TRUE （'t'），否则返回 FALSE（'f'）。

- 如果一个正在触发的事件激活了同一表或视图上的多个触发器，则所有的 BEFORE 动作在任何 FOR EACH ROW 动作之前发生，并且所有的 AFTER 操作在 FOR EACH ROW 动作之后发生。同一事件上不同触发器的执行顺序不被保证。
- 触发器例程必须用 SPL 语言编写。它们不能用外部语言编写，如 C 或 Java™ 语言，但是它们可以包含到外部语言例程的调用，如：应用程序编程编程接口 **mi_trigger** ，用于触发器内省。
- 触发器函数不能引用保存点。由触发动作引起的数据值或数据库结构变更必须被全部提交或全部回滚。 GBase 8s 不支持触发动作的部分回滚。

有关 **mi_trigger** API 的更多信息，请参阅 *GBase 8s DataBlade API 程序员指南* 和 *GBase 8s DataBlade API 函数参考* 。

如果您包含 REFERENCING 子句但省略了 FOR 子句，或者您包含 FOR 子句但省略了 REFERENCING 子句，则 CREATE FUNCTION 语句发生错误而失败。

如果您都省略了 REFERENCING 和 FOR 子句，则 UDR 不能使用 SELECTING 、INSERTING 、DELETING 和 UPDATING 运算符，并且不能声明可以表示和操纵触发器定义指定的表或视图上触发动作的列值。

有关 Delete 、Insert 、Select 和 Update 触发器的 REFERENCING 子句的语法的描述，请参阅 CREATE TRIGGER 语句中有关 REFERENCING 子句部分。

## 重载函数名

因为 GBase 8s 支持***例程重载***，所有您可以用同一名称定义多个函数，但有不同的参数列表。您可能希望在以下情况中重载函数：

- 您使用与内置函数相同的名称创建用户定义函数（如 **equal( )**），以处理新拥有定义的数据类型。
- 您创建***类型层次结构***，其中子类型从超类型继承数据表示法和函数。
- 您创建 ***distinct 类型***，它是用于与现有数据类型具有相同的内部存储表示的数据类型，但是名称不同。并且在没有强制转型的情况下无法与源类型相比较。Distinct 类型从其源类型继承支持函数。

有关唯一标识每个用户定义函数的例程特征符的简短描述，请参阅例程重载以及例程签名 。

**示例**

重载函数通过名称和输入参数列表来唯一标识。除了提供一个长的唯一标识符，还可以通过特定名称从而在之后使用它。以下示例显示了一个重载函数，它的标识符是 getArea，具有特定名称 getSquareArea 和 getRectangleArea ：

```
CREATE FUNCTION getArea
        (i INT DEFAULT 0)
        RETURNING INT SPECIFIC getSquareArea;
        DEFINE j INT;
        LET j = i * i;
```

```
RETURN j;
END FUNCTION;

CREATE FUNCTION getArea
(i INT DEFAULT 0, j INT DEFAULT 0)
RETURNING INT SPECIFIC getRectangleArea;
DEFINE k INT;
LET k = i * j;
RETURN k;
END FUNCTION;
```

现在您可以使用特定名称，如下所示：

```
GRANT EXECUTE ON SPECIFIC FUNCTION getSquareArea TO gbasedbt;
GRANT EXECUTE ON SPECIFIC FUNCTION getRectangleArea TO gbasedbt;
```

若没有特定名称，则您可能需要发出下列语句：

```
GRANT   EXECUTE ON FUNCTION getArea (INTEGER) TO gbasedbt;
GRANT   EXECUTE ON FUNCTION getArea (INTEGER,INTEGER) TO gbasedbt;
```

## 使用 SPECIFIC 子句指定特定名称

可以为一个用户定义函数声明一个在数据库中独一无二的特定名称。当您重载函数时，特定名称就很有用。

## DOCUMENT 子句

DOCUMENT 子句中带引号的字符串提供对 UDR 的摘要和描述。该字符串存储在 **sysprocbody** 系统目录表中，适用于 UDR 的用户。拥有对数据库访问特权的任何人均可查询 **sysprocbody** 系统目录表，以获取对存储在数据库中的一个或全部 UDR 的描述。

例如，以下查询获取对 SPL 函数所显示的 SPL 函数 update_by_pct 的描述：

```
SELECT data FROM sysprocbody b, sysprocedures p
        WHERE b.procid = p.procid
        --join between the two catalog tables
        AND p.procname = 'update_by_pct'
        -- look for procedure named update_by_pct
        AND b.datakey   = 'D'-- want user document;
```

先前的查询返回以下文本：

```
USAGE: Update a price by a percentage
      Enter an integer percentage from 1 - 100
      and a part id number
```

UDR 或者应用程序可查询系统目录表来取出 DOCUMENT 子句并为用户显示它。

对于 C 和 Java™ 语言函数，无论您指定是否使用 END FUNCTION 关键字，均可以在 CREATE FUNCTION 语句末尾包含 DOCUMENT 子句。

## WITH LISTING IN 子句

WITH LISTING IN 子句指定编译时发送警告的文件名。编译 UDR 之后，此文件包含一条或多条警告消息。

如果您不使用 WITH LISTING IN 子句，则编译器不生成警告列表。

在 UNIX™ 平台上，如果您指定文件名而非目录，将在数据库驻留的计算机上的主目录中创建此列表文件。如果您在此计算机上没有主目录，则在根目录 （名为 “/”的目录）创建此文件。

在 Windows™ 系统中，如果您指定文件名而非目录，则在数据库位于本地计算机的情况下，在当前工作目录中创建此列表文件。否则，缺省目录为 **%GBASEDBTDIR%\bin**。

## SPL 函数

SPL 函数是用 SPL 编写的 UDR ，可返回一个或多个值。要编写并注册 SPL 函数，请使用 CREATE FUNCTION 语句。在 CREATE FUNCTION 和 END FUNCTION 关键字之间嵌入适当的 SQL 和 SPL 语句。还可以将 DOCUMENT 和 WITH FILE IN 选项放在该函数后面。

分析 SPL 函数，（尽可能）优化函数 f，并以可执行文件的形式存储在系统目录表中。SPL 函数的主体存储在 **sysprocbody** 系统目录表中。关于该函数的其它信息存储在其它系统目录表中，包括 **sysprocedures** 、**sysprocplan** 和 **sysprocauth**。有关这些系统目录表的更多信息，请参阅 *《GBase 8s SQL 指南：参考》*。

END FUNCTION 关键字在每个 SPL 函数中是必需的，并且在紧邻语句块之前的子句后面要加上分号（;）。以下代码示例创建了 SPL 函数：

```
CREATE FUNCTION update_by_pct ( pct INT, pid CHAR(10))
        RETURNING INT;
        UPDATE inventory SET price = price + price * (pct/100)
        WHERE part_id = pid;
        return (select price from inventory where part_id = pid);
        END FUNCTION
        DOCUMENT "USAGE: Update a price by a percentage",
        "Enter an integer percentage from 1 - 100",
        "and a part id number"
        WITH LISTING IN '/tmp/warn_file';
```

有关如何编写 SPL 函数的更多信息，请参阅 *GBase 8s SQL 教程指南* 中关于 SPL 的章节。

另见 SPL 例程中的事务 一节。

您可以在 SPL 函数中包含有效的 SQL 或 SPL 语言语句。但是，请参阅 其它语法段 中的以下各节，这些节描述了 SPL 例程中 SQL 和 SPL 语句上的限制：SPL 语句的子集在语句块中有效 、SQL 语句在 SPL 语句块中有效 和 在数据操纵语句中 SPL 例程的限制 。

## 外部程序

***外部函数***是用 GBase 8s 支持的外部语言（即，SPL 以外的编程语言）编写的函数。

创建一个 C 用户定义函数

1. 编写 C 函数。
2. 编译该函数并将编译代码存储在共享库中（ C 的共享对象文件）。
3. 使用 CREATE FUNCTION 语句在数据库服务器中注册该函数。

创建以 Java™ 语言编写的用户定义函数

1. 编写 Java 静态方法，该方法可使用 JDBC 函数与数据库服务通信。
2. 编译该 Java 源文件并创建 **.jar** 文件（ Java 的共享对象文件）。
3. 使用 EXECUTE PROCEDURE 语句执行 **install_jar( )** 程序以在当数据库中安装该 JAR 文件。
4. 如果此 UDR 使用用户定义类型，那么请在 SQL 数据类型和 Java 类之间创建一个映射。使用 EXECUTE PROCEDURE 语句 中说明的 **setUDTextName( )** 程序。
5. 使用 CREATE FUNCTION 语句注册该 UDR 。

数据库服务器只将包含已编译的例程的共享对象文件的路径名存储于数据库，而不是外部例程的内容。当它执行该外部例程时，数据库服务器调用此外部对象代码。

数据库服务器存储有关系统目标表中外部函数的信息，包括 **sysprocbody** 和 **sysprocauth** 。有关系统目录的更多信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 注册 C 用户定义函数的示例

以下示例将一个名为 equal( ) 的外部 C 用户定义函数注册到数据库中。该函数采用了两个 basetype1 类型的参数，并返回一个 Boolean 值。外部例程引用名称指定了到 C 共享库（实际存储函数对象代码）的路径。该库包含 C 函数 basetype1_equal( ) ，可在 equal( ) 函数执行期间调用此函数。

```
CREATE FUNCTION equal ( arg1 basetype1, arg2 basetype1)
        RETURNING BOOLEAN;
        EXTERNAL NAME
        "/usr/lib/basetype1/lib/libbtype1.so(basetype1_equal)"
        LANGUAGE C
        END FUNCTION;
```

## 注册用 Java 语言编写的 UDR 的示例

以下 CREATE FUNCTION 语句注册用户定义函数 sql_explosive_reaction( ) 。该函数曾在 sqlj.install_jar 中讨论。

```
CREATE FUNCTION sql_explosive_reaction(INT) RETURNS INT WITH (class="jvp") EXTERNAL NAME "course_jar:Chemistry.explosiveReaction" LANGUAGE JAVA;
```

该函数返回一个 INTEGER 值。EXTERNAL NAME 子句指定 **sql_explosive_reaction( )** 函数的 Java™ 实现是称为 **explosiveReaction( )** 的方法，它驻留于驻留在 **course_jar** JAR 文件中的 **Chemistry** Java 类中。

## 已创建数据库对象的所有权

当执行此 UDR 时，除非为建立的对象指定了其它所有者，否则创建所有者特权 UDR 的用户，而非执行此 UDR 的用户拥有此 UDR 创建的任何数据库对象。

例如，假设用户 **mike** 创建了该用户定义函数：

```
CREATE FUNCTION func1 () RETURNING INT;
        CREATE TABLE tab1 (colx INT);
        RETURN 1;
        END FUNCTION;
```

如果用户 **joan** 现在执行 **func1** 函数，用户 **mike** 而非用户 **joan** 是新建的表 **tab1** 的所有者。

然而，在 DBA 特权 UDR 的情况中，除非在此 UDR 中的数据库对象被指定给其它所有者，否则执行 UDR 的用户（而非 UDR 的所有者）拥有此 UDR 创建的任何数据库对象。

例如，假设用户 **mike** 创建了该用户定义函数：：

```
CREATE DBA FUNCTION func2 () RETURNING INT;
        CREATE TABLE tab2 (coly INT);
        RETURN 1;
        END FUNCTION;
```

如果用户 **joan** 现在执行 **func2** 函数，用户 **joan** 而非用户 **mike** 是新建的表 **tab2** 的所有者。

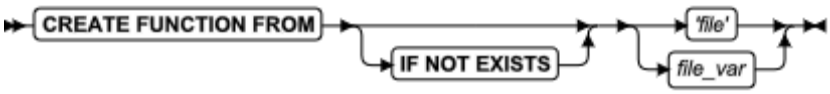
另见 对用户身份.和角色的支持 一节。

# 2.29 CREATE FUNCTION FROM 语句

使用 CREATE FUNCTION FROM 语句访问 CREATE FUNCTION 语句驻留在独立文件中的用户定义的函数。

该语句是 SQL ANSI/ISO 标准的扩展。请在 ESQL/C 中 使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *file* | 包含完整 CREATE FUNCTION 语句文本的文件的路径和文件名。缺省路径名为当前目录。 | 必须存在，且仅包含一个 CREATE FUNCTION 语句 | 必须遵循操作系统规则 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *file_var* | 存储 *file* 值的变量 | 与 *file* 的限制相同 | 特定于语言 |

**用法**

使用 C 或 Java™ 语句编写的函数称为***外部***函数。当 IFX_EXTEND_ROLE 配置参数设置成 ON 时，只有被授予内置 EXTEND 角色的用户才可以创建外部函数。

GBase 8s ESQL/C 程序不能直接创建用户定义的函数。即，它不能包含 CREATE FUNCTION 语句。

在 GBase 8s ESQL/C 程序内创建这些函数：

1. 用 CREATE FUNCTION 语句创建源文件。
2. 使用 CREATE FUNCTION FROM 语句将该源文件的内容发送到数据库服务器以执行。

   您在 ***file*** 参数中指定的文件只能包含一个 CREATE FUNCTION 语句。

例如：假设以下 CREATE FUNCTION 语句位于名为 del_ord.sql 的独立文件中：

```
CREATE FUNCTION delete_order( p_order_num INT) RETURNING INT, INT;
                    DEFINE item_count INT;
                    SELECT count(*) INTO item_count FROM items
                    WHERE order_num = p_order_num;
                    DELETE FROM orders WHERE order_num = p_order_num;
                    RETURN p_order_num, item_count;
                    END FUNCTION;
```

在 GBase 8s ESQL/C 程序中，您可以使用以下 CREATE FUNCTION FROM 语句访问 delete_order( ) SPL 函数：

```
EXEC SQL create function from 'del_ord.sql';
```

如果您不确定文件中的 UDR 是用户定义的函数还是用户定义的过程，请使用 CREATE ROUTINE FROM 语句。

您提供的文件名是相对的。如果您提供不带路径名的简单文件名（如上述示例所示），则客户端应用程序在当前目录中查找该文件。

**重要：** GBase 8s ESQL/C 预处理器不处理您指定的文件的内容。它只将内容发送到数据库服务器以执行。因此，对于您在 CREATE FUNCTION FROM 中指定的文件是否实际包含 CREATE FUNCTION 语句没有进行语法检查。然而，要提高代码的可读性，建议匹配这两个语句。

## 2.30 CREATE GLOBAL TEMPORARY TABLE

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

全局临时表与永久表的区别是表中的数据不会永远存在，与 8s 原有临时表的区别是表的结构会被保存且各个会话可见。

全局临时表分为会话全局临时表和事务全局临时表。会话全局临时表的数据在会话期间存在，会话的数据对于当前会话私有，每个会话只能看到并修改自己的数据；事务全局临时表的数据只是在事务期间存在，当事务结束，表中数据自动清除。

使用 CREATE GLOBAL TEMPORARY TABLE 语句在当前数据库中创建全局临时表。

**语法**

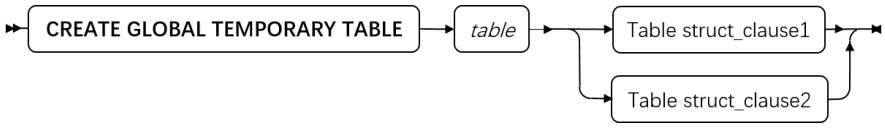
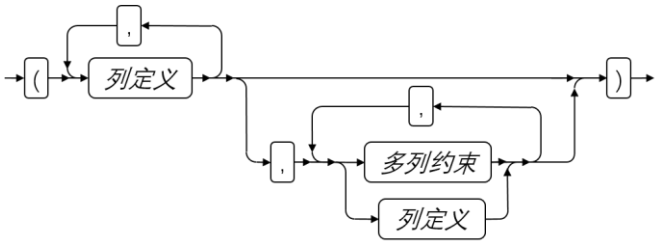



Table struct_clause1



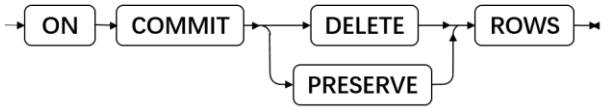
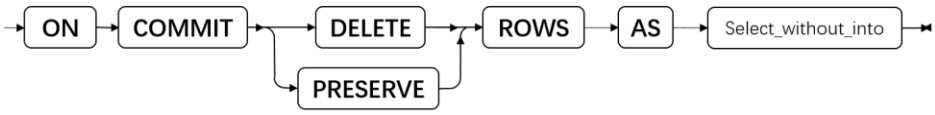




Table struct_clause2




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***table*** | 为新的全局临时表声明的名称 | 在数据库中的表、视图、和序列的名称中必须是唯一的，表名最大长度为 128 个字节 | 标识符 |

由 on commit <delete | preserve> rows 指定全局临时表为会话全局临时表还是事务全局临时表。

| 子句 | 表类型 | 数据存续期 | 清除数据操作 |
|---|---|---|---|
| on commit delete rows | 事务临时表 | 事务级 | commit<br>rollback<br>结束当前会话 |
| on commit preserve rows | 会话临时表 | 会话级 | 结束当前会话 |
| 缺省 | 事务临时表 | 事务级 | commit |

| | | | |
|---|---|---|---|
| | | | rollback<br>结束当前会话 |

通过 Table struct_clause2 创建会话临时表，会复制源表的结构和数据；通过 Table struct_clause2 创建事务临时表，只复制源表的结构。源表的索引和约束不参与复制。

**用法**

您必须具有数据库上的 Connect 特权才能创建临时表。该临时表的结构对所有连接的用户可见，但数据只对当前用户的当前连接可见。

支持在全局临时表上定义索引和约束。

不支持在全局临时表上创建视图。

全局临时表在无日志记录的数据库中禁用。

全局临时表的表约束不能有引用完整性约束。支持CHECK、NOT NULL、NULL、PRIMARY KEY、UNIQUE约束。

例如，在以下示例中，创建了一个会话级临时表：

```
CREATE GLOBAL TEMPORARY TABLE employee(
   id            SERIAL            PRIMARY KEY,
   name          VARCHAR(40)       NOT NULL,
   age           INT               NOT NULL   UNIQUE,
   address       CHAR(50)          DEFAULT null,
   salary        DECIMAL(10,2)   CHECK(salary>2000),
   join_date     DATE              DEFAULT today
)
ON COMMIT PRESERVE ROWS;
```

在以下示例中，创建了一个事务级临时表：

```
CREATE GLOBAL TEMPORARY TABLE employee(
   id            SERIAL            PRIMARY KEY,
   name          VARCHAR(40)       NOT NULL,
   age           INT               NOT NULL   UNIQUE,
   address       CHAR(50)          DEFAULT null,
   salary        DECIMAL(10,2)   CHECK(salary>2000),
   join_date     DATE              DEFAULT today
);
```

在以下示例中，创建了一个事务级临时表，该事务临时表的结构与表 employee 一致：

```
CREATE GLOBAL TEMPORARY TABLE emp_copy
ON COMMIT DELETE ROWS
AS SELECT * FROM employee;
```

## 全局临时表的 DDL 操作

支持对全局临时表执行 DDL 操作（如 ALTER TABLE、DROP TABLE、CREATE INDEX），但只有在没有会话绑定到全局临时表时才被允许。

通过对全局临时表执行插入操作，将会话绑定到全局临时表。通过发出 TRUNCATE 语句或终止会话来解除与全局临时表的绑定，或者对于事务临时表，通过发出 COMMIT 或 ROLLBACK 语句解除与全局临时表的绑定。

## 全局临时表的存储

创建全局临时表时，只创建表的结构并在系统表中记录，不初始化内存空间，当某一会话使用临时表时，从空间分配一块内存。也就是说只有向全局临时表中插入数据时，才会给全局临时表分配存储空间。

全局临时表的元数据信息存在系统表中（如 systables、syscolumns、sysconstraints、sysindexes 等）。支持使用 info tables 命令返回临时表的表名。通过查询表 systables 的 flags 列来查看表的类型。

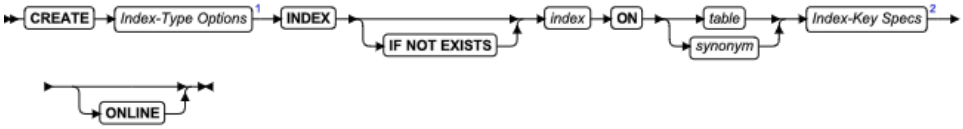
全局临时表的数据以及全局临时表上建立的索引，约束存储在 DBSPACETEMP 指定的临时表空间内，如果未指定，则存储在 rootdbs 中。

# 2.31 CREATE INDEX 语句

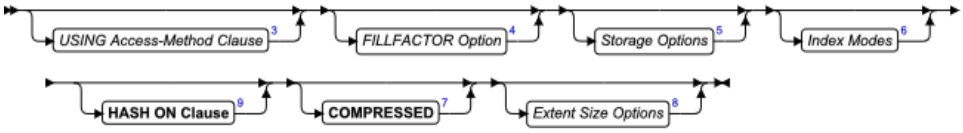
使用 CREATE INDEX 语句为表中的一列或多列，或者使用列作为参数的 UDR 返回的值创建索引。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




索引选项




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *index* | 在此为新的索引声明名称 | 在数据库中的索引名称中必须唯一 | 标识符 |
| *synonym, table* | 要建立索引的标准或临时 *table* 的名称或者同义词 | 同义词以及其表必须存在于当前数据库中 | 标识符 |

**用法**

当发出 CREATE INDEX 语句时，表在互斥方式下锁定。如果另一个过程正在使用表，CREATE INDEX 返回一个错误。（然而，关于异常，请参阅 CREATE INDEX 的 ONLINE 关键字。）

如果索引在存储加密数据的列上，则数据库服务器不能使用该索引。

如果包含了可选的 IF NOT EXISTS 关键字，当指定名称的索引已在当前数据库内的指定的表中定义时，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

索引使用当 CREATE INDEX 执行时生效的对照。

***辅助存取方法***（有时称为 ***索引存取方法***）是一组构建、存取和操作索引结构（如 B-tree 、R-tree 或 DataBlade 模块提供的索引结构）以加速数据检索的数据库服务器函数。

***synonym*** 或 ***table*** 都不可以参考虚拟表或 CREATE EXTERNAL TABLE 语句定义的表对象。

您不能直接在内置函数中预定函数型索引，但是可以创建一个 SPL 包装器以调用并返回内置函数的值。此用户定义函数的参数定义了值不能是来自结合数据类型的列的函数型索引。

以下统计信息由带或不带 ONLINE 关键字的 CREATE INDEX 语句自动生成：

- 索引级别统计信息，等价于 B-tree 索引以 LOW 方式在 UPDATE STATISTICS 操作中生成的统计信息。
- 列分布存储统计信息，等价于一般的 B-tree 索引的非透明主索引列以 HIGH 方式在 UPDATE STATISTICS 操作中生成的分布存储。

## 索引类型选项

使用 CREATE INDEX 语句的 DISTINCT 或 UNIQUE 和 CLUSTER 选项指定索引的特征。

索引类型选项

DISTINCT　　指定索引所基于的列仅接受唯一数据。

UNIQUE　　指定索引所基于的列仅接受唯一数据。

CLUSTER　　按索引指定的顺序对表的行重新排序。

**UNIQUE 或 DISTINCT 选项用法**

如果您未指定 UNIQUE 或 DISTINCT 关键字，则该索引在被索引的列或者被索引的列集上允许重复的值。

带有唯一索引的列至多可有一个 NULL 值。

您不能为 UNIQUE 索引键指定 R-tree 辅助存取方法。

以下示例创建了一个唯一索引，防止 customer_num 列出现重复值：

```
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num);
```

DISTINCT 和 UNIQUE 关键字是同义词，所以以下语句与前面的示例具有相同的作用：

```
CREATE DISTINCT INDEX c_num_ix ON customer (customer_num);
```

两个示例中的索引维持在升序中，这是缺省顺序。下一个示例在同一列上定义了一个名为 **c_num_desc_ix** 的唯一降序索引：

```
CREATE UNIQUE INDEX c_num_desc_ix ON customer (customer_num DESC);
```

您也可以通过创建带有 CREATE TABLE 的唯一约束或 ALTER TABLE 语句 的 ADD CONSTRAINT 子句来阻止列或列组中的复制。

在一个 NLSCASE INSENSITIVE 数据库中，NCHAR 和 NVARCHAR 数据类型列上的索引忽视字母大小写差异，以致于数据库服务器将由相同字母序列组成的不同大小写的字符串视为重复值。如果新行中列值与同一表中现有行同一列的值的不同之处仅在于大小写，则您不能向具有 NCHAR 或 NVARCHAR 列并在其列定义了唯一约束或唯一索引的表中插入行或更改该表的行。有关带有 NLSCASE INSENSITIVE 属性的数据库的更多信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式 。

**CLUSTER 选项用法**

您不能在同一语句中同时指定 CLUSTER 选项和 ONLINE 关键字。此外，一些辅助存取方法（如 R-tree ）不支持集群。在您为索引指定 CLUSTER 之前，请确保它使用支持集群的存取方法。

如果同一表上已存在 CLUSTER 索引，则 CREATE CLUSTER INDEX 语句失败。

```
CREATE CLUSTER INDEX c_clust_ix ON customer (zipcode);
```

该语句在 **customer** 表上创建了一个索引，并按行的邮政编码对行做物理的升序（缺省的）排序。

如果指定了 CLUSTER 选项并且数据中存在分片，则仅在每个分片内集群数据值，而非在整个表全局集群数据值。

如果 CREATE CLUSTER INDEX 语句还包含 COMPRESSED 关键字作为存储选项，则数据服务器发出错误 -26950 。要创建支持压缩的聚簇索引，需要两步：

- 使用 CREATE CLUSTER INDEX 语句定义不带索引压缩的集群索引。
- 调用带有 `'index compress'` 参数的 SQL 管理 API **task( )** 或 **admin( )** 函数压缩现有的聚簇索引。

您不能在树索引的集群中使用 CLUSTER 选项。

## 索引是如何影响主键、唯一和引用约束的

数据库服务器为主键、唯一和引用约束创建内部 B-tree 索引。如果创建表之后添加了主键、唯一或引用约束，则在受约束的列使用用户创建的索引（如果合适）。

适当的索引是索引与在主键、引用或唯一约束中使用的相同的列的索引。

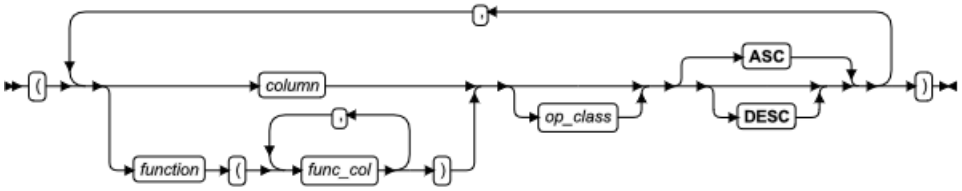
如果适当的用户创建的索引不可用，则数据库服务器在约束列或列组上创建未分片的内部索引。

## 索引键规范

使用 CREATE INDEX 语句的索引键规范定义索引的键值。它也可以指定升序或降序顺序和运算符类。

这是索引键规范的语法：

索引键规范




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 对此索引用作键的列 | 请参阅将列作为索引键的限制。 | 标识符 |
| *function* | 对此索引用作键的用户定义函数 | 必须为不返回大对象数据类型的非变量函数。不能是内置代数、指数、日志或十六进制函数。 | 标识符 |
| *func_col* | 值作为***函数***参数的列 | 不能是集合数据类型。请参阅使用函数的返回值作为索引键。 | 标识符 |
| *op_class* | 与此索引键的***列***或***函数***相关联的运算符类 | 如果 USING 子句中的辅助存取方法没有缺省运算符类，则您必须在此指定一个。（请参阅 使用运算符类。） | 标识符 |

索引键值可为包含内置数据类型的一个或多个列。如果指定多个列，列集合中的值并置将被作为索引的单个组合列。

索引键值也可以是下面几种列之一：

- LVARCHAR（***size***）类型的列，如果 ***size*** 小于 387 字节
- 包含用户定义的数据类型的一个或多个列
- 用户定义的函数返回的一个或多个 值（称为***函数型索引***），UDF 的参数列表是同一行中一个或多个列值
- 一列或多列的值和来自一个或多个用户定义函数返回值的组合

387 子字节的 LVARCHAR 大小限制是 dbspaces 的缺省页大小（2 千字节），但是大的页大小的 dbspace 不支持大索引键大小，如下表所示。

| 页大小 | 最大索引键大小 |
|---|---|
| 2 千字节 | 387 字节 |

| 页大小 | 最大索引键大小 |
|---|---|
| 4 千字节 | 796 字节 |
| 8 千字节 | 1,615 字节 |
| 12 千字节 | 2,435 字节 |
| 16 千字节 | 3,245 字节 |

**指定排列顺序**

缺省情况下，索引按升序顺序排序，从最小值到最大值，根据区域设置的排列顺序，或者如果 SET COLLATION 语句指定了非缺省排列顺序，根据创建索引时生效的顺序。您可以使用 DESC 关键字颠倒此排列顺序，以致于索引按最大值到最小值排序。

如果您在索引键规范中显式地指定 ASC 关键字，则该索引按照升序顺序排序。

**指定运算符类**

如果在 USING 子句中的辅助存取方法没有缺省的运算符类，则索引键规范可以为此索引指定运算符类。

如果在 USING 子句中的辅助存取方法有缺省的运算符类，则索引键规范可以为此索引指定覆盖此缺省运算符类的运算符类。

## 将列作为索引键的限制

以下限制应用到 CREATE INDEX 语句参考的索引键规范的任何列或列列表：

- 所有列必须存在并必须位于创建该索引的表中。
- 表必须存在于当前数据库中，且不能是 CREATE EXTERNAL TABLE 语句定义的对象。
- 该列的数据类型不能是集合数据类型。
- 列的最大值和所有列的总宽度的最大值取决于数据库服务器的页大小。请参阅 创建复合索引。
- 您无法对在其上已有唯一约束的列或者列列表添加升序索引。请参阅 使用 ASC 和 DESC 排序顺序选项 。
- 您无法对有主键约束的列或者列列表添加唯一索引。原因是：将列或列列表定义为主键将使数据库服务器在列或列列表上创建唯一的内部索引；你不能使用 CREATE INDEX 语句在此列或列列表上定义另一个唯一索引。
- 您可在同一列或同一列组上创建的索引数是受限制的。请参阅 列组上索引数目的限制。

有关应用到指定为函数型索引的参数的列的其它索引键的限制，请参阅 使用函数的返回值作为索引键 。

## 使用函数的返回值作为索引键

***函数型索引***是对指定函数返回的值建立的索引，而非对列的值建立索引。例如，以下语句在将函数 Area( ) 返回的值作为键使用的表 zones 上创建函数型索引：

```
CREATE INDEX zone_func_ind ON zones (Area(length,width));
```

可在 SPL 例程中创建函数型索引。也可在不返回大对象的非变量用户定义的函数上创建索引。

函数型索引可以是 B-tree 索引、R-tree 索引或者用户定义的辅助存取方法。

函数返回的值可以是索引键，如上例所示，或者可以是其它键部分是列值、部分列值或者是其它函数索引的返回值的复合索引（有关更多信息，请参阅 创建复合索引）。

**重要：** 数据库服务器在定义函数型索引的用户定义例程（UDR）上施加以下限制：

- 参数不能是集合数据类型（LIST 、MULTISET 或 SET）列的名称。
- 此函数不能返回 BLOB 、BYTE 、CLOB 和 TEXT 数据类型的大对象。
- 此函数不能是 VARIANT 函数。
- 此函数不能包含任何 SQL 的 DML 语句。
- 当您创建函数型索引时，ONLINE 关键字无效；请参阅 CREATE INDEX 的 ONLINE 关键字 。
- 此函数必须是用户定义函数。您不能在任何 SQL 的内置函数上创建函数型索引。

然而，除了以上最后一条限制，您可以在调用非变量内置 SQL 函数的用户定义函数上创建函数型索引以，以致于内置函数返回的值是函数型索引的索引键。（即，创建调用并返回 SQL 内置函数的值的 SPL 包装器，然后在此用户定义的 SPL 函数上定义函数型索引。）

## 创建复合索引

***简单***索引在其索引键规范中仅列出一个***列***（或者仅一个***函数***，参数列表必须是一列或多列列表）。其它索引是***复合***索引。您应当按照从最常用到最少用的顺序在符合索引中列出这些列。

如果您使用 SET COLLATION 指定非缺省的语言环境的排列顺序，则可以使用不同的对照在同一列集合上创建多个索引。（类似索引仅在 NCHAR 或 NVARCHAR 列上使用。）

以下示例使用 stock 表的 stock_num 和 manu_code 列创建复合索引：

```
CREATE UNIQUE INDEX st_man_ix ON stock (stock_num, manu_code);
```

UNIQUE 关键字阻止 **stock_num** 和 **manu_code** 的给出组合的任何复制。缺省情况下索引是升序的。

您可在一个复合索引中最多包含 16 列。单个组合索引中的所有已建立索引的列的总宽度不能超过 380 字节。

***索引键部件***是表中的一列或者一个或多个列上用户定义的函数的结果。复合索引可最多有 16 键部分（为列时），或者最多 341 键部分（为 UDR 返回的值时）。此限制是语言相关的，并应用到 SPL 或 Java™ 所写的 UDR ；基于 C 语言 UDR 的函数型索引可最多有 102 个键部分。复合索引可将任一以下项作为索引键：

- 一列或多列
- 用户定义的函数返回的一个或多个值（称为***函数型索引***）。

复合索引的索引键部分可以是列和用户定义函数的组合。

对于缺省 2 千字节页大小的 dbspace ，除了 GBase 8s 的函数型索引（它依赖的语言的限制已在本节的前面描述过），单个 CREATE INDEX 语句中所有索引的列的总宽度不能超过 387 字节。 对于 dbspace 中最大大小大于 2 千字节，请参阅 索引键规范 。

无论该索引是否直接基于表中的列值或者将列值作为参数的函数上，索引键的最大大小仅取决于页大小。Dbspace 中函数型索引的最大索引键大小与列索引一样都大于 2 千字节，列索引和函数型索引的唯一不同是键部件的数量。基于列的索引可以有多于 16 个键部件，但是函数型索引具有不同的依赖于语言限制的键部件。对于给出的页大小，基于列的索引和函数索引最大索引键大小都一样。

## 使用 ASC 和 DESC 排序顺序选项

ASC 选项指定了索引维持在升序中； 这是缺省顺序。DESC 选项可以指定以降序顺序保存的索引。这些 ASC 和 DESC 选项仅限于 B-trees 有效。

## 唯一约束在排序顺序选项上的影响

当列或列列表在 CREATE TABLE 或 ALTER TABLE 语句中定义为唯一时，数据库服务器通过创建唯一升序索引实施 UNIQUE CONSTRAINT 。因此，您无法使用 CREATE INDEX 语句来将升序索引添加到已经定义为唯一的列或列列表上。

然而，您可以在这样的列上创建降序索引，并且可以将这些列包含在不同组合的组合升序索引中。例如，以下序列的语句是有效的：

```
CREATE TABLE customer (
        customer_num    SERIAL(101) UNIQUE,
        fname                   CHAR(15),
        lname                   CHAR(15),
        company                 CHAR(20),
        address1                CHAR(20),
        address2                CHAR(20),
        city                    CHAR(15),
        state                   CHAR(2),
        zipcode                 CHAR(5),
        phone                   CHAR(18)
        );

        CREATE INDEX c_temp1 ON customer (customer_num DESC);
        CREATE INDEX c_temp2 ON customer (customer_num, zipcode);
```

在此示例中，在 **customer_num** 列上放置了一个唯一约束。第一个 CREATE INDEX 语句在 **customer_num** 列上放置一个按降序顺序排序的索引。第二个 CREATE INDEX 将 **customer_num** 列作为复合索引的一部分包含。关于复合索引的更多信息，请参阅 创建复合索引 。

## 索引的双向遍历

如果对一列创建索引时不指定 ASC 或 DESC 关键字，则缺省情况下值以升序顺序存储；但是数据库服务器的双向遍历能力让您对一列仅创建一个索引并将该索引用于查询；这些查询指定结果以排序列的升序还是降序来排序。

由于此能力，是否将单列索引创建为升序或者降序索引无关紧要。不管您为索引选择哪种存储顺序，数据库服务器在处理查询时可按照升序或降序顺序来遍历该索引。

然而，如果您在表上创建一个复合索引，可能需要 ASC 和 DESC 关键字。例如，如果您希望输入其 ORDER BY 子句按照多个列排序并且按照不同的顺序对每个列排序的 SELECT 语句，并且您希望对此查询使用一个索引，则需要创建一个与 ORDER BY 列对应的复合索引。例如，假设您希望输入以下查询：

```
SELECT stock_num, manu_code, description, unit_price
        FROM stock ORDER BY manu_code ASC, unit_price DESC;
```

此查询通过 manu_code 列的值升序排序，然后按照 unit_price 列的值降序排序。要对此查询使用索引，您需要发出一个与 ORDER BY 子句的需求相对应的 CREATE INDEX 语句。例如，您可输入以下两个语句之一来创建索引：

```
CREATE INDEX stock_idx1 ON stock
        (manu_code ASC, unit_price DESC);
        CREATE INDEX stock_idx2 ON stock
        (manu_code DESC, unit_price ASC);
```

为此查询使用的复合索引（stock_idx1 或 stock_idx2）不能用于您使用 ORDER BY 子句为两列指定相同的排序方向的查询。例如，假设您希望输入以下查询：

```
SELECT stock_num, manu_code, description, unit_price
        FROM stock ORDER BY manu_code ASC, unit_price ASC;
        SELECT stock_num, manu_code, description, unit_price
        FROM stock ORDER BY manu_code DESC, unit_price DESC;
```

如果您希望使用复合索引来提高这些查询的性能，则需要输入以下 CREATE INDEX 语句之一。可使用两种已创建的索引之一（stock_idx3 或 stock_idx4 ）来提高先前的查询的性能。

```
CREATE INDEX stock_idx3 ON stock
        (manu_code ASC, unit_price ASC);
        CREATE INDEX stock_idx4 ON stock
        (manu_code DESC, unit_price DESC);
```

可在一列上创建不超过一个的升序索引和不超过一个的降序索引。由于数据库服务器的双向遍历能力，您仅需创建这些索引之一。同时创建与在 **stock_num** 列上的升序或者降序排序完全达到相同的结果。

在索引的表上执行 INSERT 或 DELETE 操作之后，索引项的数量在页里变化，表需要的索引页的数量可以取决于该索引是否指定升序或降序顺序。对于一些加载和 DML 操作，按降序排序的单个列或多列索引可能导致数据库服务器分配的索引页多于按升序排序的索引。

## 列组上索引数目的限制

你可以在列组上创建多个索引，提供到每个索引有一个唯一升序和降序列的组合。例如，要在 stock 表的 stock_num 和 manu_code 列上创建所有可能的索引，您能创建四个索引：

- 两个列上升序的 **ix1** 索引
- 两个列上降序的 **ix2** 索引

- **ix3** 索引在 **stock_num** 上升序以及在 **manu_code** 上降序
- **ix4** 索引在 **stock_num** 上降序以及在 **manu_code** 上升序

由于数据库服务器的双向遍历能力，你不需要创建这四个索引。您仅需创建两个索引：

- **ix1** 和 **ix2** 索引对于用户为两个列指定的相同的排序方向（升序或降序）的排序达到相同的结果，因此您仅需这对索引之一。
- **ix3** 和 **ix4** 索引对于用户为两个列（第一列上的升序和第二列上的降序或反之）指定的不同的排序方向的排序达到相同的结果。因此，您仅需创建这对索引之一。（另见 索引的双向遍历。）

如果每个索引都有不同的排列顺序，则 GBase 8s 还可以支持对同一升序和降序列组合的多个索引；请参阅 SET COLLATION 语句 。

## 使用运算符类

***运算符类***是用于查询优化和构建索引的与辅助存取方法相关的运算符集合。如果以下两个之一成立，则当创建索引时您必须指定运算符类：

- 对辅助存取方法不存在缺省运算符类。（用户定义的存取方法可提供非缺省运算符类。）
- 您希望使用与辅助存取方法提供的缺省运算符类不同的运算符类。

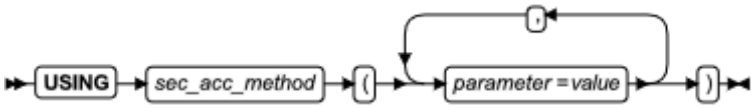
如果使用代替的存取方法，并且如果存取方法有一个缺省运算符类，则您能在此省略运算符类；但是如果您不指定运算符类并且辅助存取方法没有缺省运算符类，数据库服务器返回一个错误。有关更多信息，请参阅缺省运算符类。以下 CREATE INDEX 语句在将 abs_btree_ops 运算符类用于 cust_num 键的 cust_tab 表上创建一个 B-tree 索引：

```
CREATE INDEX c_num1_ix ON cust_tab (cust_num abs_btree_ops);
```

## 使用 access-method 子句

USING 子句为新的索引指定辅助存取方法。

使用 access-method 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *parameter* | 此索引的辅助存取方法参数 | 请参阅您的用户定义存取方法的用户文档 | 引用字符串 |
| *sec_acc_method* | 此索引的辅助存取方法 | 方法可为 B-tree 、R-tree 、BTS 或用户定义的存取方法，如 DataBlade 模块定义的方法 | 标识符 |
| *value* | 指定 *parameter* | 必须是辅助存取方法中 | 引用字符串 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 的值 | *parameter* 的有效的文字值 | 或 精确数值 |

*辅助存取方法*是执行索引所需的所有操作的一组例程，如创建、删除（drop）、插入、删除（delete）、更新和扫描。

数据库服务器提供以下辅助存取方法：

- 一般 B-tree 索引是内置辅助存取方法。

  B-tree 索引有助于检索某一范围数据值的查询。数据库服务器实现此辅助存取方法并在系统目录表中将其注册为 **btree** 。

- R-tree 方法是注册的辅助存取方法。

  R-tree 索引有助于搜索多维数据。数据库服务器在数据库的系统目录表中将此辅助存取方法注册为 **rtree**。R-tree 辅助存取方法对 UNIQUE 索引键无效。R-tree 索引不能都被集群。R-tree 索引可存储在非缺省页大小的 dbspace 中。有关 R-tree 索引的更多信息，请参阅 *GBase 8s R-Tree 索引用户指南* 。

- **bts** 方法是注册的辅助存取方法。

  使用 **bts** 存取方法在存储于表的某个列中的文档存储库中执行词和语句的基本文本查询。要执行基本文本查询，请使用 **bts** 存取方法在文本列上创建索引，然后使用 bts_contains() 查询谓词函数和其它管理函数。有关 **bts** 存取方法的更多信息，请参阅 Create the index by specifying the bts access method 。

您指定的存取方法必须在 **sysams** 系统目录表中注册。缺省辅助存取方法是 B-tree 。

如果存取方法是 B-tree ，则您仅能为每个升序或降序列的组合或使用运算符类的功能键创建一个索引。（此限制不适用于其它辅助存取方法。）缺省情况下，CREATE INDEX 创建一个一般 B-tree 索引。如果希望使用非 B-tree 的辅助存取方法来创建索引，则您必须在 USING 子句中指定辅助存取方法的名称。

一些用户定义的存取方法作为 DataBlade 模块打包。一些 DataBlade 模块提供在创建它们时需要特定参数的索引。关于用户定义的存取方法的更多信息，请参阅您的辅助存取方法的文档或 DataBlade 模块。

以下（实现 R-tree 索引的数据库的）示例在包含不透明数据类型 point 的 location 列上创建 R-tree 索引，并在 location 上执行带有过滤器的查询。

```
CREATE INDEX loc_ix ON TABLE emp (location) USING rtree;
SELECT name FROM emp WHERE location N_equator_equals point('500, 0');
```

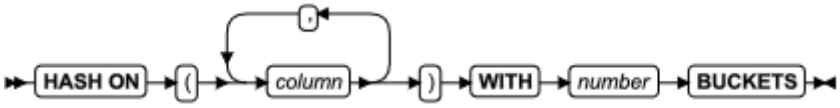
以下 CREATE INDEX 语句创建使用 fulltext 辅助存取方法的索引，它接受两个参数：WORD_SUPPORT 和 PHRASE_SUPPORT 。它为表 t 建立索引，它有两列：i，一个整数列，以及 data ， 一个 TEXT 列。

```
CREATE INDEX tx ON t(data)
        USING fulltext (WORD_SUPPORT='PATTERN',
        PHRASE_SUPPORT='MAXIMUM');
```

## HASH ON 子句

使用 CREATE INDEX 语句的 HASH ON 子句指定森林树索引的子树（存储区）的数目及其列。

HASH ON 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 您使用 HASH ON 子句创建森林树索引的列或列组的名称 | 该列表必须是在 CREATE INDEX 语句中使用的索引列的前缀列表。 | 标识符 |
| *number* | 要创建森林树索引所需的子树（存储区）数 | 森林树索引的存储区数必须在 2 到每个 dbspace 可用索引页数之间。 | 整数文字 |

用法

森林树索引是拆离的索引。它们不能是连接的索引。

您可以在基本数据类型的列上创建森林树索引。

您不能：

- 在具有复杂数据类型的列、UDT 、或函数列上创建森林树索引。
- 当创建森林树索引时，使用 CREATE INDEX 语句的 FILLFACTOR 选项。因为索引是从顶向底建立。
- 创建集群的森林树索引。
- 在森林树索引上运行 ALTER INDEX 语句。
- 在使用聚合（包括最小和最大范围值）的查询中使用森林树索引。
- 直接在森林树索引的 HASH ON 列上执行范围扫描。

  然而，您可以在不列在 HASH ON 列列表的列上执行范围扫描。对于在 HASH ON 列列表列出的列的范围扫描，您必须创建包含此范围扫描的适当列表的附加 B-tree 索引。该附加 B-tree 索引可能具有与森林树索引相同的列列表，加或减一列。
- 可对 OR 索引路径使用森林树索引。数据库服务器不会在索引列上具有 OR 为此的查询中使用森林树索引。

当您创建森林树索引时，选择足够的列以创建唯一值。

**提示**: 一般情况下，要选择的列取决于每一列的副本数。例如，如果第一列包含很少量的副本，如果前两列不包含大量副本,则前两列满足散列。如果前两列包含大量的副本，则您还需要选择第三个列。

子树的数目取决于您创建的索引的目的。如果您的目的是：

- 为减少争用，最初创建每个 CPU VP 2 个子树的森林树索引。您可能需要更多子树，这取决于表中的行数和存在多少副本。
- 要减少 B-tree 中层级数：
  1. 运行 gcheck -pT 命令。
  2. 在此输出中，找到每一层级的节点数。
  3. 决定需要多少个子树来实现索引中每个数的期望深度。

例如，假设一个索引每页均有 100 个键，该索引有 1M 个键，则该树则看起来像这样：

- Level 1 (root) 100 keys
- Level 2 10K keys
- Level 3 1M keys

要将 3-level tree 减少到 100 2-level tree， 该索引大概需要 100 个子树。要将 3-level tree 减少到 10K 1-level tree ，该索引大概需要 10K 个子树。

如果使用了太多的或太少的子树，则森林树页面可以比传统的 B-tree 页面更稀疏。当页面稀疏时，更多页占据缓冲池，这将导致其它表的缓存变得更少。

**示例**

以下命令创建了名为 idx1 的森林树索引，它在 c1 列上有 100 个子树：

```
CREATE INDEX idx1 ON tab1(c1) HASH ON (c1) with 100 buckets;
```

以下命令创建了名为 idx2 的森林树索引。在此命令中，该语句的 HASH ON 部分的前缀列表是 c1 和 c2，它是在该语句的 CREATE INDEX 部分中使用的 c1 、c2 和 c3 列的前缀列表：

```
CREATE INDEX idx2 on tab2(c1, c2, c3) HASH ON (c1, c2) with 10 buckets;
```

以下命令在列 c1 和列 c2 上创建了一个等式查找的森林树索引：

```
CREATE INDEX idx3 on tab3(c1, c2) HASH ON (c1, c2) with 100 buckets;
```

以下命令创建了类似于先前森林树索引的 B-tree 索引。该索引用于列 c1 和列 c2 的范围扫描：

```
CREATE INDEX idx4 on tab4(c1, c2, c3);
```

## FILLFACTOR 选项

当您想要创建压缩索引或为以后索引的扩展提供信息时，可使用 FILLFACTOR 选项指定索引页的充满程度。

FILLFACTOR 选项只在以下情况中有效：

- 当在一个含有超过 5,000 行并使用超过 100 个表页的表上构建索引时
- 挡在分片表上创建索引时
- 当在非分片表上创建分片索引时

不能在森林树索引上使用 FILLFACTOR 选项。

FILLFACTOR Option


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *percent* | 当创建索引时每个被索引数据填充的索引页的百分比。缺省值为 90 。 | 1 ≤ *percent* ≤100 | 精确数值 |

当创建索引时，数据库服务器最初仅填充 FILLFACTOR 值指定的节点百分比。

FILLFACTOR 还能在 ONCONFIG 文件中作为参数来设置。CREATE INDEX 语句上的 FILLFACTOR 子句覆盖 ONCONFIG 文件中的设定。有关 ONCONFIG 文件和您能使用的参数的更多信息，请参阅 *GBase 8s 管理员指南*。

## 提供低百分比值

如果您提供一个低百分比值，类似 50 ，您允许索引中的增长。索引的节点最初填充到一定的百分比并为插入包含空间。可用空间的数量取决于每页中键的数量以及百分值。

例如，FILLFACTOR 值为 50%，页面将半满并能适应大小的加倍。低百分比值能导致更快的插入并能用于期望增长的索引。

## 提高百分比值

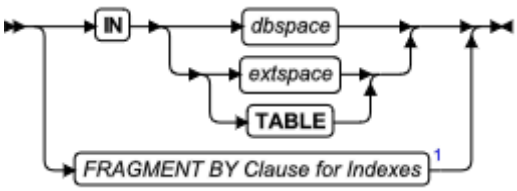
如果您提高百分比值，例如 99 ，则索引将是压缩的，并且任何新索引的插入将呆滞分割节点。密度的最大值是 100% 。具有 100% 的 FILLFACTOR 值时，索引没有空间可用于增长；任何索引的添加都将导致分割节点。

99% 的 FILLFACTOR 值允许每个节点至少一个插入的空间。高百分比值可能产生更快的查询，并适合与您不希望增长的索引或大部分只读索引。

### 存储选项

存储选项指定索引的分布方案。您可使用 IN 子句来为整个索引指定存储空间，或者可使用 FRAGMENT BY 子句来在多个存储空间上分片索引。

存储选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *dbspace* | 存储索引的 dbspace | 必须存在 | 标识符 |
| *extspace* | 由 gspaces 命令分配到数据库服务器外的存储区域的名称 | 必须存在 | 请参阅您的存取方法的文档。 |

如果您指定任何存储选项（除了 IN TABLE），您将创建一个*拆离的索引*。拆离索引是使用特定的分布模式创建的索引。即使为索引指定的分布模式与为表指定的相同，索引仍被认为要拆离。如果表的分布方案更改了，所有拆离的索引还将继续使用 Storage Option 子句指定的分布方案。

如果您不包含 Storage Option 子句，则缺省情况下在同一 dbspace 建立连接索引，并将其作为对应表的分片。然而，如果启用自动定位，则缺省情况下单个分片中循环表上建立的索引被拆离，并将其放置在服务器选择的 dbspace 中。通过设置 AUTOLOCATE 配置参数或者将会话环境变量选项设置为正整数来启用自动定位。

## 索引的 COMPRESSED 选项

如果索引拥有 2000 或更多的键，则可使用 CREATE INDEX 语句的 COMPRESSED 关键字压缩 B-tree 索引。

您可以在分片表或未分片表上创建压缩的索引。

您不能创建的压缩的索引是集群索引。然而，您可以通过运行带 index compress 参数的 SQL 管理 API task() 或 admin() 函数压缩现有的集群索引。

要被压缩，索引中的分片或索引必须具有至少 2000 个键。如果您在创建不具有足够多的键的索引时使用 COMPRESSED 选项，则数据库服务器在创建该索引时不会压缩该索引或分片。即使向该索引添加键，其仍然保持未压缩状态。如果您想要压缩该索引，则运行带 index compress 参数的 SQL 管理 API task() 或 admin() 函数。

如果表没有足够的数据来提供足够大的索引键样本，则数据库服务器不会压缩该索引或分片。即使将最小数量的新键添加到现有未压缩的索引中，数据库服务器仍不会压缩该索引。 然而，压缩索引后，数据库服务器压缩并插入任何新的键到该索引中。

以下示例在 customer 表的 address 列创建了一个名为 cust3_ix 的压缩的索引：

```
CREATE INDEX cust3_ix ON customer (address) COMPRESSED
        EXTENT SIZE 32 NEXT SIZE 32;
```

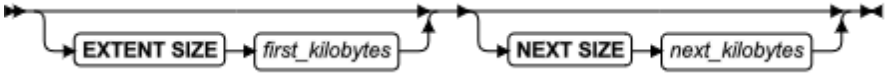
以下示例创建了一个唯一的压缩的索引：

```
CREATE UNIQUE INDEX cust3_ix ON customer (address) COMPRESSED ;
```

## Extent Size 选项

Extent Size 选项可定义分配到索引的存储 extent 的大小。

Extent Size 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *first_kilobytes* | 该索引的第一个 extent 的长度(以千字节为单位) | 必须返回正整数；最大值是 chunk 大小，以千字节为单位 | 表达式 |
| *next_kilobytes* | 每个后续的 extent 的长度（以千字为单位） | 同 *first_kilobytes* 一样 | 表达式 |

***first_kilobytes*** （***next_kilobytes***）的最小长度是您系统上磁盘页大小的四倍。例如，如果您有 2 千字节的页系统，则最小长度是 8 千字节。

如果需要修改索引的 extent 大小,则您可以修改生成的卸载表的模式文件中 extent 和下一个 extent 大小。例如，要使数据库更有效率，您可以删除该索引，修改模式文件中的 extent 大小，然后创建新的索引。有关如何优化 extent 的更多信息，请参阅 *GBase 8s 管理员指南* 。

只有您为该索引显式分配作为 extent 大小的 extent 大小的值存储在系统目录中。您在 CREATE INDEX 语句的 EXTENT SIZE 选项中指定的值存储于 **sysindices** 系统目录表的 **fextsize** 列，NEXT SIZE 选项中指定的值存储于同一表的 **nextsize** 列中。然而，如果您省略这些选项，数据库服务器在这些系统目录列中存储零值（ 0 ），而非它为该索引的第一个 extent 和第二个 extent 计算并分配的缺省值。

**定义带显式 extent 大小的索引的示例**

以下程序段创建了新表并在该表上定义了两个未分片的索引。

```
CREATE TABLE IF NOT EXISTS t (a INT, b INT);
CREATE INDEX   IF NOT EXISTS idx1 ON t(a) EXTENT SIZE 32 NEXT SIZE 32;
CREATE INDEX   IF NOT EXISTS idx2 ON t(b);
```

此处 idx1 的定义指定 32 千字节作为显式 extent 大小。第二个索引 idx2 ，具有系统计算的缺省 extent 大小。这两个 CREATE INDEX 语句包含这些 extent 大小条目的索引的系统目录描述：

- **sysindices.fextent** 和 **sysindices.nextent** 列值对 **idx1** 分别为 32 。
- **sysindices.fextent** 和 **sysindices.nextent** 列值对 **idx2** 分别为 0 。

此处 **idx2** 的 0 值指示没有指定显式 extent 大小（并非指示未分配存储空间）。

## IN 子句

使用 IN 子句来指定持有整个索引的存储空间。您指定的存储空间必须已经存在。

## 在 dbspace 中存储索引

使用 IN ***dbspace*** 子句指定您希望索引驻留的 dbspac。当和任何选项（除了 TABLE 关键字）一起使用此子句时，将创建拆离的索引。

IN *dbspace* 子句允许您隔离索引。例如， 如果 customer 表在 custdata dbspace 中创建，但您希望在称为 custind 的单独的 dbspace 中创建索引，请使用以下语句：

```
CREATE TABLE customer
    . . .
    IN custdata EXTENT SIZE 16;

CREATE INDEX idx_cust ON customer (customer_num) IN custind;
```

## 在命名的分区中存取索引分片

除了在 dbspace 中存储索引分片的选项， GBase 8s 还支持在一个或多个 dbspaces 的命名子集中存储索引的分片。除非显式地在 PARTITION BY 或 FRAGMENT BY 子集中声明分片的名称，缺省情况下，每个分片都与它所驻留的 dbspace 拥有相同的名称。这包括了所有分片表和由 GBase 8s 较早期发行版中迁移过来的索引。

## 在 extspace 中存储数据

通常，***extspace*** 存储选项与使用 access-method 子句一起使用。有关更多信息，请参阅您所使用的定制存取方法的用户文档。

## 使用 IN TABLE 关键字创建索引

指定 IN TABLE 作为存储选项会创建一个存储行为等同于 GBase 8s 较早版本的缺省的索引。该表的索引和数据页都存储在同一 extent 中，且该索引的 dbspace 分布方案与创建该表时的分布方案相同

使用 IN TABLE 作为存储选项，对未分片 B-tree 索引指定与启用 **DEFAULT_ATTACH** 环境变量相同的存储方案，但是 **DEFAULT_ATTACH** 和 IN TABLE 关键字都是不建议使用的功能。

**DEFAULT_ATTACH** 环境变量的名称保留术语***连接索引***的过时定义。在当前 GBase 8s 术语中，该术语现在指定一个索引，其数据页存储在单独的 tablespace 中，并且是表的数据页单独 extent，但是索引和其表共享同一 dbspace 分布方案有关更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 中 **DEFAULT_ATTACH** 的描述。

以下限制应用于作为索引存储选项的 IN TABLE 关键字：

- 如果您定义索引的表是未分片表，则当指定 IN TABLE 选项时， GBase 8s 发出错误 -212 和 -130 。
- 你不能在森林树索引上应用 IN TABLE 存储选项。
- 该选项不支持与可扩展性相关的索引，如 R-tree 索引、函数型索引或 DataBlade 模块提供的索引。

- 不能使用不同于该表或不同于 **DB_LOCALE** 所指定的一个排列顺序创建附加索引。有关 **DB_LOCALE** 环境变量的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

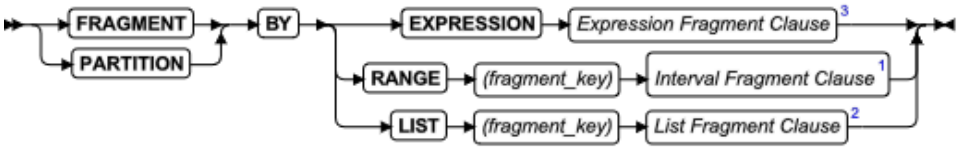
GBase 不建议新的应用程序中使用 IN TABLE 存储选项 **DEFAULT_ATTACH** 环境变量。这样的索引是一个不推荐使用的功能，以后发行的 GBase 8s 可能不支持此功能。

## 索引的 FRAGMENT BY 子句

使用 FRAGMENT BY 子句创建拆离索引，并在多个 sbspace 或分区上定义存储策略。

这类似表的 FRAGMENT BY 子句的语法，但索引分片不支持 ROUND ROBIN 关键字不。PARTITION BY 关键字是此上下文中 FRAGMENT BY 关键字的同义词。

索引的 FRAGMENT BY 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbspace* | 存储索引分片的 *dbspace* | 您能指定至多 2,048 个 *dbspaces* 。所有存储分片的 *dbspaces* 必须具有相同的页大小。 | 标识符 |
| *fragment_key* | 索引列上的强制转型、列或函数表达式。索引根据此表达式的值分片。 | 列只能来自当前的表。 | 表达式 |

此处 IN 关键字引入了用来存储索引分片的存储空间的名称。如果您在 IN 关键字时候列出多个 ***dbspace*** 名称，则使用圆括号来为 ***dbspace*** 列表定界。所有存储分片的 ***dbspaces*** 必须具有相同的页大小。将位于 EXPRESSION 关键字后面的分段定义列表括起的圆括号是可选的。

对于使用同一 RANGE 区间或者 LIST 分片策略作为它们表策略的索引，您在 PARTITION 关键字之后声明的每个分片的名称必须与对应表分片的标识符相同。

对于由 RANGE 区间分片策略分片的连接索引，如果在新插入的行的范围内不存在表分片，则数据库服务器创建一个新的表分片以存储新行，并为此新表分片声明系统生成的名称。如果该表被索引，且索引被与表相同的 RANGE 区间策略分片，则数据库服务器还会创建新的索引分片。在这种情况中，该索引分片具有与对应表分片相同的系统生成的标识符。有关系统生成的 RANGE 区间分片的信息，请参阅 Interval fragment 子句和通过 RANGE INTERVAL 分片 。

## 分片存储表达式上的限制

以下限制适用于此表达式：

- 每个分片表达式仅可包含来自当前表的列，其数据值仅来自单行。
- 该表达式必须返回一个 BOOLEAN 值（true 或 false)。
- 子查询、聚集、用户定义的例程或对 ROW 类型列字段或顺序对象的参考都是无效的。
- 内置 CURRENT 、SYSDATE 、TODAY 、SITENAME 、DBSERVERNAME 、CURRENT_USER 或 USER 函数都无效。
- DEFAULT_ROLE 和 CURRENT_ROLE 运算符是无效的。

以上列出的限制也适用于使用 LIST 分片存储策略的索引。

## 系统索引的分片存储

如果存在用户定义的索引，系统索引（如那些实现引用约束和唯一约束的索引）将利用用户定义的索引。如果没有用户定义的索引可以利用，系统索引保留未分片，并移到创建该数据库的 dbspace 中。

要分片系统索引，请在约束列中创建分片索引，然后使用 ALTER TABLE 语句添加约束。

## 唯一索引的分片存储

您可以在使用循环或基于表达式的分布方案的表上分片唯一索引，但是分片表达式中的列必须是被索引列的一部分。如果您的索引分片存储策略违反了此限制，则 CREATE INDEX 语句失败并且工作会回滚。

## 临时表上索引的分片存储

您可在临时表上分片唯一索引（仅当下面的表使用基于表达式的分布方案时）。就是说，定义临时表的 CREATE TEMP TABLE 语句必须指定显式的基于表达式的分布方案。（不支持按 ROUND ROBIN 索引的分片，且自动按照 LIST 或 INTERVAL 分片，对于表的唯一索引使用列表或区间存储分区策略。）

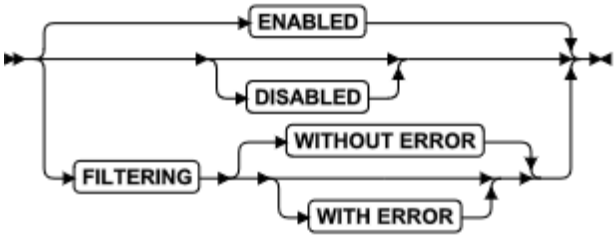
如果尝试在当您创建表时没有为其指定分片存储策略的临时表上创建分片的唯一索引，则数据库服务器在 **DBSPACETEMP** 环境变量指定的第一个 dbspace 内创建索引。有关 **DBSPACETEMP** 环境变量的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

有关临时表的缺省存储特征的更多信息，请参阅临时表的存储位置。

## 索引模式

使用 CREATE INDEX 语句的索引方式在 INSERT 、DELETE 、MERGE 和 UPDATE 操作期间指定索引的行为。

索引模式




DISABLED

数据库服务器不在修改基本表的 insert 、delete 和 update 操作之后更新索引。优化程序在查询执行期间不使用索引。

ENABLED

数据库服务器在修改基本表的 insert 、delete 和 update 操作之后更新索引。优化程序在查询执行期间使用索引。 如果 insert 或 update 操作导致复制键值添加到唯一索引，则语句失败。

FILTERING

数据库服务器在修改基本表的 insert 、delete 和 update 操作之后更新唯一索引。（此选项对复制索引不可用。）

优化程序在查询执行期间使用索引。如果在过滤方式中，插入或更新操作导致复制键值添加到唯一索引，则语句继续处理，但是坏行被写入与基本表相关联的违例表中。关于唯一索引违例的诊断信息被写入与基本表相关联的诊断表。

如果为唯一索引指定过滤，则您也可指定以下错误选项之一。

WITHOUT ERROR

插入或更新操作期间的唯一索引违例没有完整性违例错误返回给用户。

WITH ERROR

插入或更新操作期间的唯一索引违例完整性违例错误返回给用户。

有关变更唯一索引的数据库对象方式的信息，请参阅 约束和唯一索引的模式 。

## 为唯一索引指定方式

为 CREATE INDEX 语句中的唯一索引指定方式时必须遵守以下规则：

- 您可将唯一索引的方式设置为 enabled 、disabled 或 filtering 。
- 如果不指定方式，那么缺省情况下其它索引。
- 对于设置为过滤方式的索引，如果您不指定错误选项，则缺省为 WITHOUT ERROR 。
- 当您向现有的基本表添加新的唯一索引并为索引指定禁用方式时，CREATE INDEX 语句成功，即使索引的列中的复制值将导致唯一索引违例。
- 当向现有的基本表添加新的唯一索引并为索引指定启用或过滤方式时，CREATE INDEX 语句成功（倘若将导致唯一索引违例的索引的列中不存在复制值）。然而，如果索引的列中存在任何复制值，CREATE INDEX 语句失败并返回一个错误。

- 当向启用或过滤方式中现有的基本表添加新的唯一索引时，并且复制值存在于索引的列中，基本表中的错误行没有过滤到违例表中。这样，您无法使用违例表来监测基本表中的错误行。

### *当复制值存在于列中时添加唯一索引*

如果您试图在启用方式中添加唯一索引但由于复制值在索引的列中而接收到错误信息，则采取以下步骤来成功添加索引：

1. 在禁用方式中添加索引，再次发出 CREATE INDEX 语句，但是这次指定 DISABLED 关键字。
2. 使用 START VIOLATIONS TABLE 语句为目标表启动违例和诊断表。
3. 发出 SET Database Object Mode 语句来将索引方式更改为启用。当发出此语句时，违反唯一索引需要的目标表中的现有行在违例表中复制。然而，您接收到一个完整性违例的错误消息，并且索引保持禁用。
4. 在违例表上发出 SELECT 语句来检索从目标表复制的不一致性。您可能需要连接违例表和诊断表来获取所有必要的信息。
5. 在违反唯一索引需要的目标表中的行上采取更正操作。
6. 修复目标表中所有不一致行之后，再次发出 SET Database Object Mode 语句俩奖金用的索引切换为启用方式。这次索引被启用，并且没有返回完整性违例错误。因为目标表中的所有行满足新的唯一索引需要。

## 为复制索引指定方式

当您在 CREATE INDEX 语句中为复制索引指定方式时必须遵守以下规则：

- 可将复制索引设置为启用或禁用方式。过滤方式仅限于唯一索引可用。
- 如果不指定复制索引方式，则缺省情况下索引是启用的。

### 数据库服务器如何对待禁用的索引

无论禁用的索引是唯一索引还是重复索引，实际上数据库服务器在数据操纵（DML）操作期间会忽略该索引。

当禁用索引时，数据库服务器停止更新索引并停止在查询期间使用索引，但是保留关于禁用索引的目录信息。如果在该列或列组上已经存在禁用索引，则您不能在列或列组上创建新的索引。类似地，如果活动的约束依赖的索引被禁用，则您不能在列或列组上创建活动的（启用的）唯一约束、外键约束或主键约束。

### CREATE INDEX 的 ONLINE 关键字

DBA 将 ONLINE 关键字包含在 CREATE INDEX 语句末尾的规范中，从而降低非互斥存取错误的风险，提高被索引表的可用性。ONLINE 关键字指示数据库服务器在最小化排它锁持续时间时创建该索引，以致于并行用户存取该表时能创建索引。

缺省情况下，CREATE INDEX 试图在被索引的表上放置互斥锁以防止创建该索引时其它用户存取该表。如果另一个用户已经锁定该表或者当前存取的表处于 Dirty Read 隔离级别，则 CREATE INDEX 语句失败。

即使其它用户正在该被索引的表的上执行 Dirty Read 和 DML 操作，数据库服务器仍建立索引。一发出 CREATE INDEX ONLINE 语句，该新索引将对用于查询计划或成本评估的查询优化器不可见，并且数据库服务器也不支持任何在该已建立索引的表上的 DDL 操作，直到正确地构建了指定的索引。此时，数据库服务器在用新索引的信息更新系统目录的同时暂时地锁定该表。

CREATE INDEX ONLINE 语句中已建立索引的表可以是永久表也可以是临时表，可以是日志记录的也可以是没有日志记录的，可以是分片表也可以是非分片表。然而，在创建具有以下任一属性的索引时，不能指定 ONLINE 关键字：

- 函数型索引
- 集群索引
- 虚拟索引
- R-tree 索引
- 按区间分片策略分区的索引
- 在按区间分片策略分区的表上的索引

此外，如果在该表上定义了主键，且一个或多个并发的会话正在拥有引用该主键的外键约束的子表上执行 DML 操作，则 CREATE INDEX ONLINE 操作产生错误 -710 。在此索引上创建 ONLINE 之前，您必须等待直到所有的具有子表的用户会话已完成。

以下语句指示数据库服务器在 customer 表的 lname 列上很创建一个名为 idx_1 的唯一联机索引：

```
CREATE UNIQUE INDEX IF NOT EXISTS idx_1 ON customer(lname) ONLINE;
```

如果正在创建此索引时，有其它用户在 **customer** 表中插入了新行，其中 **lname** 不唯一，则数据库服务器将在它完成创建该新的 **idx_1** 索引并在系统目录表中注册此索引后发出一个错误。

术语***联机索引***指的是数据库在使用 ONLINE 关键字创建或删除索引时所遵循的锁定策略，而不是索引在完成创建（或销毁）后它继续拥有的属性。然而此术语也出现在一些错误消息中，在恢复或复原操作中，数据库服务器将把任何您创建为联机索引的索引重新创建为联机索引。

只有 CREATE INDEX ONLINE 或 DROP INDEX ONLINE 语句可以在同一表或具有相同标识符的联机索引上并发地引用联机索引。

## 自动计算分布统计信息

当 CREATE INDEX 语句执行成功，有或没有 ONLINE 关键字， GBase 8s 自动为新建的索引生成统计信息，并用等同的在根据索引类型方式中 UPDATE STATISTICS 操作的值更新 sysdistrib 系统目录表：

- 索引级别统计信息，等同于 UPDATE STATISTICS 在 LOW 方式中生成的统计信息，它们可用于大多数类型的索引计算，包括 B-tree 、虚拟索引和函数型索引。
- 列分布存储统计信息，等同于普通 B- tree 索引的前导索引列在 HIGH 方式中的生成的分布。如果表由少于百万的行，则解析百分比为 1.0 ，对于较大的表则为 0.5 。

当优化程序为创建新索引的表设置查询计划时，这些分布统计信息对优化程序有效。

对于复合键索引，只有前导列的分布存储能由 CREATE INDEX 语句隐式地创建。

以下类型的索引不支持隐式创建分布统计信息：

- 在用户定义的数据类型的列上的索引
- 在内置透明数据类型（包括 BOOLEAN 和 LVARCHAR）的列上的索引
- R-tree 索引
- 连接索引

如果分布统计的计算在 CREATE INDEX 操作期间失败，则数据库服务器在错误日志中报告该失败，但仍会继续创建索引。

当通过显式或隐式 CREATE INDEX 操作成功创建了分布存储时，如果 SET EXPLAIN 设置为 ON ，则会生成以下解释信息（类似于 UPDATE STATISTICS 生成的信息）。

```
Index:                  idx_01 on nita.foo
STATISTICS CREATED AUTOMATICALLY:
Column Distribution for:                nita.foo.a
Mode:                   MEDIUM
Number of Bins:         101 Bin size:   100.0
Sort data:              0.3 MB
Completed building distribution in:     0 minutes 33 seconds
```

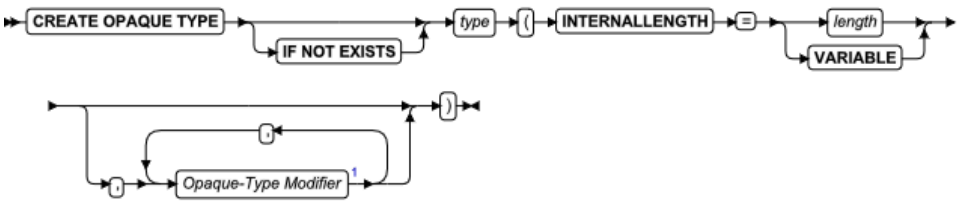
有关分布存储统计信息和分布在 LOW 方式和 MEDIUM 方式之间的不同的信息，请参阅 UPDATE STATISTICS 语句中的描述。

## 2.32 CREATE OPAQUE TYPE 语句

使用 CREATE OPAQUE TYPE 语句创建不透明的数据类型。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *length* | 存储此数据类型值需要的字节数 | 当 sizeof( ) 伪指令应用于类型结构时返回的正整数 | 精确数值 |
| *type* | 在这里为新的不透明数据类型声明的名称 | 在数据库的数据类型名称中必须是唯一 | 标识符 |

用法

CREATE OPAQUE TYPE 语句在 **sysxtdtypes** 系统目录中注册新的不透明数据类型。

如果您包含 IF NOT EXISTS 关键字，如果指定名称的 OPAQUE 数据类型已经在当前数据库中注册过，则数据库服务器不采取任何操作（而非向应用程序发送异常）。

要创建不透明数据类型，必须拥有数据库上的 Resource 权限。当创建不透明数据类型时，只有您，即所有者拥有此新的不透明数据类型上的 Usage 特权。可以使用 GRANT 或 REVOKE 语句向此数据库的其他用户授予或撤销 Usage 权限。

要查看数据类型上的特权，请在 **sysxtdtypes** 系统目录表中检查所有者的名称，并在 **sysxtdtypeauth** 系统目录表中检查可能已经授权的附加类型特权。

有关系统目录表的详细信息，请参阅 *《GBase 8s SQL 指南：参考》*。

DB-Access 实用程序也可显示不透明数据类型上的特权。

## 为不透明类型声明名称

您为不透明数据类型声明的名称是 SQL 标识符。当在不符合 ANSI 的数据库中创建不透明数据类型时，该名称在数据库中的数据类型名称中必须是唯一的。

当在兼容 ANSI 的数据库中创建不透明数据类型时，***owner.type*** 组合必须在数据库内是唯一的。所有者名称是区分大小写的。如果不在所有者名称上加引号，则不透明类型 ***owner*** 的名称将以大写字母存储。

## INTERNALLENGTH 修饰符

INTERNALLENGTH 修饰符指定作为固定长度或者变化长度的不透明数据类型所需的存储大小。

## 固定长度不透明类型

固定长度不透明类型有固定大小的内部结构。要创建 固定长度不透明类型，请为 INTERNALLENGTH 修饰符指定内部结构的大小（按字节）。下一示例创建称为 fixlen_typ 的固定长度不透明类型，并分配 8 个字节存储此数据类型。

```
CREATE OPAQUE TYPE fixlen_typ(INTERNALLENGTH=8, CANNOTHASH)
```

## 可变长度的不透明类型

可变长度的不透明类型有一个内部结构，其大小可能从一个值变化成另一个值。例如，不透明数据类型的此内部结构可能最多保存某确定大小的字符串的实际值，但是超出此大小时它肯使用 CLOB 的 LO 指针保存该值。

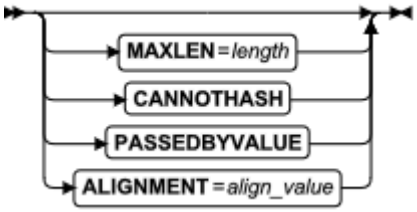
要创建可变长度的不透明数据类型，将 VARIABLE 关键字与 INTERNALLENGTH 修饰符一起使用。以下语句创建称为 varlen_typ 的可变长度不透明数据类型：

```
CREATE OPAQUE TYPE varlen_typ
```

(INTERNALLENGTH=VARIABLE, MAXLEN=1024)

## 不透明类型修饰符

Opaque-Type 修饰符




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *align_value* | 依照其对齐传递给用户定义的例程的不透明数据类型的字节边界。缺省值为 4 字节。 | 必须为 1 、2 、4 或 8，者取决于不透明数据类型的 C 定义以及用以构建该数据类型的对象文件的硬件和编译器 | 精确数值 |
| *length* | 为可变长度不透明类型的示例分配的最大长度。缺省为 2 千字节。 | 必须为正整数 ≤ 32 KB。请勿为固定长度数据类型指定。超过此长度的值会返回错误 | 精确数值 |

修饰符可以为不透明数据类型指定以下可选信息：

- MAXLEN 为可变长度类型指定最大长度。
- CANNOTHASH 指定数据库服务器不能对不透明类型使用内置散列函数。
- ALIGNMENT 指定数据库服务器对齐不透明类型所依据的字节边界。
- PASSEDBYVALUE 指定需要 4 字节或更少字节存储的不透明类型由值传递。

缺省情况下，不透明类型按引用传递至用户定义的例程。

## 定义不透明类型

要定义数据库服务器新的不透明数据类型，必须通过以下 C 或 Java™ 语言中的信息：

- 作为不透明数据类型的内部存储器的数据结构 A

  该类型的内部存储器详细信息已隐藏或不透明。一旦定义了新的不透明数据类型。数据库服务器就可以操纵该类型，而无需了解存储它的 C 或 Java 结构的知识。
- 允许数据库服务器与此内部结构相互作用的支持函数。

  支持函数告诉数据库服务器如何与该数据类型的内部结构相互作用。这些支持函数必须以 C 或 Java 变成语言编写。
- 其它支持函数或最终用户可以调用从而在不透明类型上运行的附加用户定义的函数（可选）

可能的支持函数包括操作员函数和强制转型函数。在 SQL 语句中能够使用这些函数之前，必须使用适当的 CREATE CAST 、CREATE PROCEDURE 或 CREATE FUNCTION 语句注册这些函数。

下表总结不透明数据类型的支持函数。

| 函数 | 描述 | 调用 |
|---|---|---|
| input( ) | 将不透明类型从其外部 LVARCHAR 表示转换为内部表示 | 当客户端应用程序将不透明类型的字符表示发送至 INSERT 、UPDATE 或 LOAD 语句中时 |
| output( ) | 将不透明类型从其内部表示转换为其外部 LVARCHAR 表示 | 当数据库服务器将不透明类型的字符表示作为 SELECT 或 FETCH 语句的结果发送时 |
| receive( ) | 将不透明类型从其在客户端计算机上的内部表示转换为其在服务器计算机上的内部表示。不管客户端和服务器计算机类型之间的区别而提供独立于平台的结果 | 当客户端应用程序将不透明类型的内部表示发送至 INSERT 、UPDATE 或 LOAD 语句中时 |
| send( ) | 将不透明类型从其在服务器计算机上的内部表示转换为其在客户端计算机上的内部表示。不管客户端和服务器计算机类型之间的区别而提供独立于平台的结果 | 当数据库服务器将不透明类型的内部表示作为 SELECT 或 FETCH 语句的结果发送时 |
| db_receive( ) | 将不透明类型从其在本地数据库上的内部表示转换为传送到本地服务器上外部数据库的 dbsendrecv 类型 | 当本地数据库从本地数据库服务器上的外部数据库接收到 dbsendrecv 类型时 |
| db_send( ) | 将不透明类型从其在本地数据库上的内部表示转换为传送到本地服务器上外部数据库的 dbsendrecv 类型 | 当本地数据库将 dbsendrecv 类型发送至本地数据库服务器上的外部数据库时 |
| server_receive( ) | 将不透明类型从其在本地服务器计算机上的内部表示转换为传送到远程数据库服务器的 srvsendrecv 类型。为此函数使用任意名称 | 当本地数据库服务器从远程数据库服务器接收到 srvsendrecv 类型时 |

| 函数 | 描述 | 调用 |
| --- | --- | --- |
| server_send( ) | 将不透明类型从其在本地服务器计算机上的内部表示转换为传送到远程数据库服务器的 srvsendrecv 类型。为此函数使用任意名称 | 当本地数据库服务器将 srvsendrecv 类型发送到远程数据库服务器时 |
| import( ) | 执行从不透明类型的外部（字符）表示转换为批量复制的内部格式所需的所有任务 | 当 DB-Access（LOAD）启动从文本文件到数据库的批量复制时 |
| export ( ) | 执行从不透明类型的内部表示转换为批量复制的外部（字符）格式所需的所有任务 | 当 DB-Access（UNLOAD）或 High Performance Loader 启动从文本文件到数据库的批量复制时 |
| importbinary( ) | 执行从客户端服务器上不透明类型的内部表示转换为批量复制在服务器计算机上的内部表示所需的所有任务 | 当 DB-Access（LOAD）或 High Performance Loader 启动从二进制文件到数据库的批量复制时 |
| exportbinary( ) | 执行从服务器计算机上不透明类型的内部表示转换为批量复制在客户端计算机上的内部表示所需的所有任务 | 当 DB-Access（LOAD）或 High Performance Loader 启动从数据库到二进制文件的批量复制时 |
| assign( ) | 在将不透明类型存储到磁盘前执行所有必需的处理。此支持函数必须命名为 assign( ) | 当数据库服务器将不透明存储到磁盘前执行 INSERT 、UPDATE 或 LOAD 时 |
| destroy( ) | 在除去包含不透明类型的行之前执行所有必需的处理。此支持函数必须命名为 destroy( ) | 当数据库服务器从磁盘除去不透明类型前执行 DELETE 或 DROP TABLE 时 |
| lohandles( ) | 返回不透明类型的 LO 指针结构（智能大对象的指针） 的列表 | 当数据库服务器必须在不透明类型中搜索智能大对象的引用时；当 gcheck 运行时或执行归档时 |
| compare( ) | 比较两个不透明类型的值并返回整数值以指示第一个值是小于、等于还是大于第二个值 | 当数据库服务器在 SELECT 语句中遇到 ORDER BY 、UNIQUE 、DISTINCT 或 UNION 子句时，或当 CREATE INDEX 创 |

| 函数 | 描述 | 调用 |
| --- | --- | --- |
| | | 建 B-tree 索引时 |

在为不透明类型编写了必要的支持函数后，使用 CREATE FUNCTION 语句在与不透明类型相同的数据库中注册这些支持函数。某些支持函数将其它数据类型转换为新的不透明类型或从新的不透明类型转换为其它数据类型。在创建并注册这些支持函数后，使用 CREATE CAST 语句将每个函数与特殊的强制转型相关联。强制转型必须注册在与支持函数相同的数据库中。

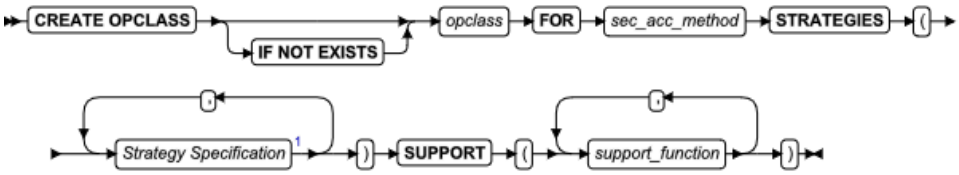
在已经编写了必要的 C 语言或 Java 语言源代码以定义不透明数据类型后，可随后使用 CREATE OPAQUE TYPE 语句在数据库中注册不透明数据类型。

## 2.33 CREATE OPCLASS 语句

使用 CREATE OPCLASS 语句为***辅助存取方法***创建***运算符类***。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *opclass* | 在此处声明的新的运算符类的名称 | 在数据库内的运算符类中必须是唯一的 | 标识符 |
| *sec_acc_method* | 新的运算符类与其关联的辅助存取方法 | 必须已经存在且必须 sysams 表中注册 | 标识符 |
| *support_function* | 辅助存取方法需要的支持函数 | 必须以存取方法所期望的顺序列出 | 标识符 |

**用法**

***运算符类***是支持用于查询优化和构建索引的与辅助存取方法的运算符集合。***辅助存取方法***（有时称为 ***索引存取方法***）是构建、存取和操作索引结构（如 B-tree 、R-tree 或 DataBlade 模块提供的索引结构）的一组数据库服务器函数。

数据库服务器提供 B-tree 和 R-tree 辅助存取方法。更多关于 **btree** 辅助存取方法的信息，请参阅缺省运算符类 。

当需要下列之一时定义一个新的运算符类：

- 索引为数据使用与缺省运算符类提供的顺序不同的顺序

- 与任何现有运算符类不同的一个运算符集合，这些运算符类与特殊的辅助存取方法相关联

如果您包含了可选的 IF NOT EXISTS 关键字，且指定名称的运算符类已经当前数据库中注册过，则数据库服务器不采取任何行动（而非向应用程序发送异常）。

必须具有 Resource 权限或者必须是 DBA 才可创建运算符类。运算符类的实际名称是 SQL 标识符。当创建运算符类时，***opclass*** 名称必须在数据库内是唯一的。

在兼容 ANSI 的数据库中创建运算符类时，***owner.opclass*** 组合必须在数据库中是唯一的。所有者名称是区分大小写的。如果不在 ***owner*** 名称旁边添加引号（或者设置 **ANSIOWNER** 环境变量），则运算符类索引者名称以大写字母存储。

以下 CREATE OPCLASS 语句为 btree 辅助存取方法创建称为 abs_btree_ops 的新的运算符类：

```
CREATE OPCLASS abs_btree_ops FOR btree
STRATEGIES (abs_lt, abs_lte, abs_eq, abs_gte, abs_gt)
 SUPPORT (abs_cmp);
```

一个运算符类有两种运算符类函数：

- 策略函数

在 CREATE OPCLASSS 语句的 STRATEGY 子句中指定运算符类的策略函数。在先前的 CREATE OPCLASS 代码示例中，**abs_btree_ops** 运算符类有 5 个策略函数。

- 支持函数

在 SUPPORT 子句中指定运算符类的支持函数。在先前的 CREATE OPCLASS 代码示例中，**abs_btree_ops** 运算符类有 1 个支持函数。

## STRATEGIES 子句

***策略函数***是用户可以在 DML 语句内调用以对指定数据类型起作用的函数。查询优化器使用策略函数确定给定的索引是否能够用以处理查询。

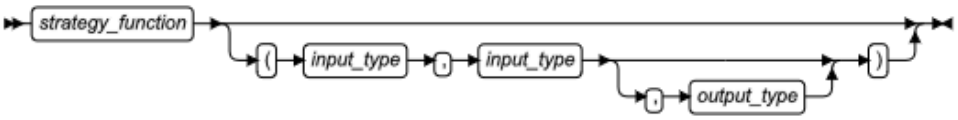
如果查询包含 UDF 或某个有索引的列，并且如果查询中的限制操作符与 STRATEGIES 子句中的任何函数都匹配，则查询优化器将考虑使用此索引进行查询。更多查询计划的信息，请参阅 *GBase 8s 性能指南* 。

当创建新的运算符类时，STRATEGIES 子句将为辅助存取方法标识策略函数。每个策略规范都列出策略函数的名称（以及可选的，它的参数的数据类型）。必须以辅助存取方法所期望的顺序列出这些函数。关于 B-tree 索引的和 R-tree 索引的缺省运算符类的策略运算符特顺序，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## 策略规范

STRATEGIES 关键字引入一个逗号分隔的，新运算符类的函数名或函数特征符列表。此列表中的每个元素被称为***策略规范***并有如下的语法：

策略规范




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *input_type* | 策略函数的输入参数的数据类型，您要为该策略函数使用特定的辅助存取方法 | ***策略函数***接受两个输入参数，并且可以有一个可选的输入参数 | 数据类型 |
| *output_type* | 策略函数的可选输出参数的数据类型 | 副作用索引的可选输出参数 | 数据类型 |
| *strategy_function* | 与指定的运算符类相关联的策略函数 | 必须以指定的辅助存取方法所期望的顺序列出 | 标识符 |

每个***策略函数***都是一个外部函数。CREATE OPCLASS 语句不验证您指定名称的用户定义的函数是否存在。但是，对于要使用策略函数的辅助存取方法，外部函数必须：

- 在共享库中编译
- 用 CREATE FUNCTION 语句在数据库中注册
- （可选）除了策略函数名称外还可以指定其特征。策略函数需要两个输入参数和一个可选的输出参数。要指定函数特征符，请指定：
- 为策略函数的两个输入参数的每一个都指定一个***输入数据类型***，从而该策略函数可使用它们
- （可选）为策略函数的输出参数指定一个***输出数据类型***

可以指定 UDT 和内置数据类型。如果不指定函数特征符，则数据库服务器假设每个策略函数采用两个相同数据类型的参数并返回一个 BOOLEAN 值。

## 对边界效果数据的索引

***边界效果***数据是策略函数在包含该策略函数的查询后返回的附加值。例如，图像 DataBlade 模块可能使用***模糊***索引搜索图像数据。索引根据图像与搜索条件的接近程序对它们进行排序。数据库服务器返回等级值作为对于合格图像的边界效果数据 。

## SUPPORT 子句

***支持函数***是辅助存取方法在内部使用以构建和搜索索引的函数。在 CREATE OPCLASS 语句的 SUPPORT 子句中为辅助存取方法指定这些函数。

必须以辅助存取方法所期望的顺序列出支持函数的名称。关于 B-tree 索引和 R-tree 索引的缺省运算符类的支持运算符特定顺序，请参阅 缺省运算符类。

支持函数是外部函数。CREATE OPCLASS 不验证指定的支持函数是否存在。但是，对于要使用支持函数的辅助存取方法，支持函数必须满足这些条件：

- 在共享库中编译
- 用 CREATE FUNCTION 语句在数据库中注册

### 缺省运算符类

每个辅助存取方法都有一个与之相关联的缺省运算符。缺省情况下，CREATE INDEX 语句将缺省运算符与索引相关联。

例如，以下 CREATE INDEX 语句在 **zipcode** 列创建 B-tree 索引并自动将缺省 B-tree 运算符类与此列相关联：

```
CREATE INDEX zip_ix ON customer(zipcode)
```

对于 GBase 8s 提供的每个辅助存取方法，它提供缺省运算符类，如下：

- 缺省 B-tree 运算符类为内置运算符类。

  数据库服务器为此运算符类执行运算符类函数并在数据库的系统目录中将其注册为 **btree_ops** 。
- 缺省 R-tree 运算符类是已注册的运算符类

  数据库服务器在系统目录表中将此运算符类注册为 **rtree_ops** 。数据库服务器不为缺省 R-tree 运算符类执行运算符类函数。

重要： 要使用 R-tree 索引，必须安装空间 DataBlade 模块，如测量 DataBlade 模块或执行 R-tree 索引的任何第三方 DataBlade 模块。这些模块执行 R-tree 运算符类函数。

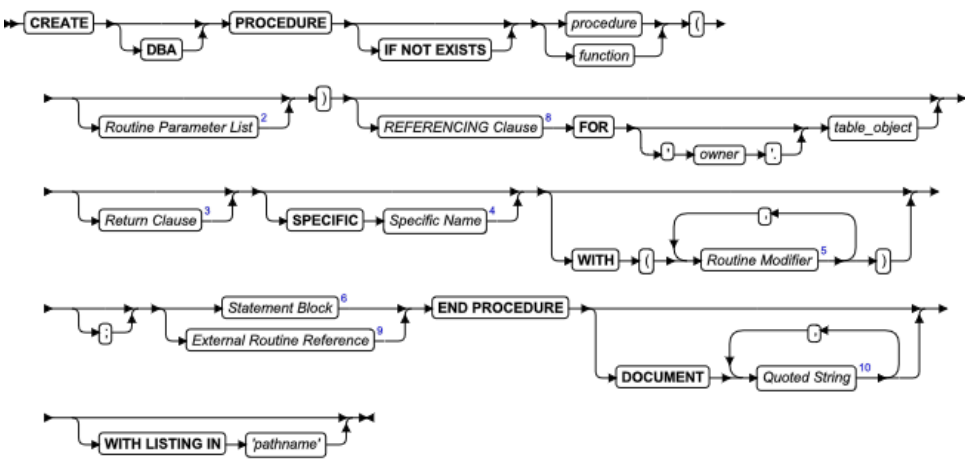
DataBlade 模块可提供其它类型的辅助存取方法。如果 DataBlade 模块提供一个辅助存取方法，则它可能还提供一个缺省运算符类。对于更多信息，请参阅 DataBlade 模块用户指南。

## 2.34 CREATE PROCEDURE 语句

使用 CREATE PROCEDURE 语句创建用户定义过程。（要从单独文件中的源代码文本创建过程，请使用 CREATE PROCEDURE FROM 语句。）

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *function, procedure* | 在此为新的 SPL 例程函数或过程声明的名称 | 请参阅 GBase 8s 上的过程名称 | 标识符 |
| *owner* | *table_object* 的所有者 | 必须拥有 *table_object* | 所有者名称 |
| *pathname* | 存储编译时警告的文件 | 必须存在于数据库驻留的计算机上 | 特定于操作系统 |
| *table_object* | 触发器可以调用 UDR 的表或视图的名称或同义词 | 必须存在于本地数据库中 | 标识符 |

**用法**

在 GBase 8s ESQL/C 中，您可以将 CREATE PROCEDURE 仅作为 PREPARE 语句中的文本使用。如果您希望创建编译时文本已知的过程，则必须使用 CREATE PROCEDURE FROM 语句。

如果您包含可选的 IF NOT EXISTS 关键字，且指定名称的过程已经注册于当前数据库中，则数据库服务器不采取任何操作（而非向应用程序发送异常）（因为过程的标识符可以被重载，所以它可以不必包含这些关键字，如果数据库服务器可以将新过程的参数列表解析为与当前数据库中相同名称的任何其他过程的参数列表不同。）

例程使用创建时有效的对照顺序，请参阅 GBase 8s 的 SET COLLATION 语句 语句，获取关于使用非缺省对照的信息。

**示例**

对于此示例，假设您具有两个如下定义的重载的过程：

```
CREATE PROCEDURE raise_prices ( per_cent INT)
UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) );
END PROCEDURE

CREATE PROCEDURE raise_prices ( per_cent INT, selected_unit CHAR )
UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) )
        where unit=selected_unit;
END PROCEDURE
```

为了引用上述过程，您需要在参数列表后提供过程名称，如下所示：

```
DROP PROCEDURE raise_prices(INT);
DROP PROCEDURE raise_prices(INT, CHAR);
```

更简便的方法是，使用指定的名称来标识它们。以下示例将会使用指定名称创建过程：

```
CREATE PROCEDURE raise_prices ( per_cent INT ) SPECIFIC        raise_prices_all
UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) );
END PROCEDURE
```

```
DROP SPECIFIC PROCEDURE raise_prices_all;
CREATE PROCEDURE raise_prices ( per_cent INT, selected_unit CHAR )
        SPECIFIC raise_prices_by_unit
UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) )
        where unit=selected_unit;
END PROCEDURE
```

我们可以简单地使用它们指定的名称来删除它们：

```
DROP SPECIFIC PROCEDURE raise_prices_by_all;
DROP SPECIFIC PROCEDURE raise_prices_by_unit;
```

## 使用 CREATE PROCEDURE 与 CREATE FUNCTION 的对比

在 GBase 8s 中，尽管可以使用 CREATE PROCEDURE 来写入并在注册返回一个或多个值的 SPL 例程（即 SPL 函数），但建议您改为使用 CREATE FUNCTION 。要注册外部函数，必须使用 CREATE FUNCTION 。

使用 CREATE PROCEDURE 语句来写入并注册 SPL 过程或注册外部过程。

有关类似用户定义的过程和用户定义的函数的术语如何在此手册中使用的信息，请参阅 例程、函数和过程之间的关系 。

## 例程、函数和过程之间的关系

***过程***是可接受参数但不返回任何值的例程。***函数***是可接受参数并返回一个或多个值的例程。 ***用户定义例程***（UDR）是包括用户定义的过程和用户定义的函数的一般术语。关于指定的和未指定的已返回值的信息，请参阅返回子句 。

可以将 UDR 写入数据库服务器为支持的（ SPL ***例程***）或者外部语言（***外部例程***）。其中术语 UDR 出现在此手册中，它同时可以指定 SPL 例程和外部例程。

***用户定义的过程***指 SPL 过程和外部过程。***用户定义函数***指 SPL 函数和外部函数。

在较早发行版的文档中，术语***存储过程***同时用于 SPL 过程和 SPL 函数。在此手册中，术语 ***SPL 例程***替换术语存储过程。在有必要区分 SPL 函数和 SPL 过程函数时，本手册将区分两者。

术语***外部例程***应用于外部过程或外部函数，这两者都构造指定 UDR ，这些 UDR 由 SPL 以外的编程语言编写。在有必要区分外部函数和外部过程时，本手册将区分这两者。

## 使用 CREATE PROCEDURE 的必要特权

必须拥有数据库上的 Resource 特权来在该数据库中创建用户定义的过程。

在能创建 SPL 过程之前，您还必须拥有要编写的过程中的 SPL 、C 或 Java™语言的 Usage 特权。有关更多信息，请参阅语言级权限。

缺省情况下，SPL 上的 Usage 特权授权为 PUBLIC 。您还必须至少拥有数据库上的 Resource 特权来在该数据库中创建 SPL 过程。

## DBA 关键字和过程上的特权

如果创建带有 DBA 关键字的 UDR ，则其称为 ***DBA 特权 UDR*** 。您需要 DBA 特权来创建或者执行 DBA 特权的 UDR 。

在不具有 DBA 特权的用户中，只有 DBA 授予 Execute 权限的用户才能调用 DBA 特权 UDR.。然而，如果 DBA 将 Execute 特权授予 PUBLIC ，则所有的用户都可以使用 DBA 特权 UDR 。有关 DBA 特权 UDR 的其它信息，请参阅 创建数据库对象的所有权 。

如果您省略 DBA 关键字，则此 UDR 称为***所有者特权*** UDR 。

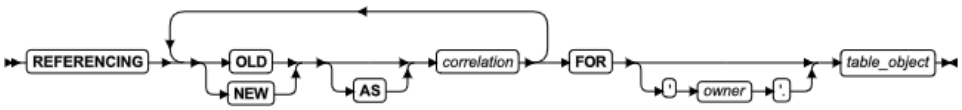
当您在兼容 ANSI 的数据库中创建了所有者特权 UDR 时，只有您自己能执行此 UDR。在其它用户可以执行所有者特权 UDR 之前，它的所有者必须将 Execute 特权授权给个别用户或角色或者 PUBLIC 。

如果您在不兼容 ANSI 的数据库中创建了所有者特权 UDR ，则任何人都可以执行此 UDR ，因为缺省情况下 PUBLIC 被授予 Execute 特权。要限制指定用户对所有者特权 UDR 的存取，则所有者必须从 PUBLIC 撤销其在 UDR 上的 Execute 特权，然后将它授权给指定的用户或角色。将 **NODEFDAC** 环境变量设置成 `yes` 可以阻止该 UDR 上的权限被缺省授予给 PUBLIC 。如果该环境变量设置成 `yes` ，则除了此 UDR 的所有者，其它任何人都不能调用此 UDR ，除非所有者将此 UDR 的 Execute 特权授权给其他用户。

## REFERENCING 和 FOR 子句

REFERENCING 子句可以声明初始值的相关名，以及 FOR 子句指定的 ***table_object*** 列中更新的值。

REFERENCING 和 FOR 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 在此声明的名称，用于在触发器例程中限定的旧的或新的列值（如 *correlation.column*） | 不能是 *table_object* | 标识符 |
| *owner* | *table_object* 的所有者 | 必须拥有 *table_object* | 所有者名称 |
| *table_object* | 其触发器可以调用此过程的表或视图的名称或同义词 | 必须在当前数据库中存在 | 标识符 |

如果在 REATE PROCEDURE 语句的参数列表后面包含 REFERENCING and FOR ***table_object*** 子句，那么您创建的例程为***触发器过程***（或***触发器 UDR*** 或***触发器例程***）。FOR 子句指定表或视图的触发器可以从 Triggered Action 列表的 FOR EACH ROW 部分调用例程。

在 REFERENCING 子句中，OLD *correlation* 指定了一个前缀，通过该前缀，触发器例程可以引用在触发器例程修改该列之前 *table_object* 的列所具有的值。NEW *correlation* 指定一个前缀，用于引用触发器例程分配给列的新值。触发器例程是否可以使用相关名称来引用 OLD 列值，NEW 列值或这两个列值取决于触发事件的类型：

- 由 Insert 触发器调用的触发器例程只能引用 NEW 相关名称。
- 由 Delete 触发器或 Select 触发器调用的触发器例程只能引用 OLD 相关名称。
- 由 Update 触发器调用的触发器例程可引用 OLD 和 NEW 相关名称。

有关在触发器操作中如何使用 *correlation.column* 表示法的更多信息，请参阅 REFERENCING 子句。

除了以 SPL 语言编写的 GBase 8s UDR 的一般要求外，触发例程还可以支持某些附加的语法特性，并且受某些限制的约束，这些特性不是对于普通的（或不是限制）触发程序：

- 触发器例程必须包含 FOR ***table_object*** 子句，该子句指定本地数据库中的表或视图的名称，其触发器可以调用此例程。
- 触发器例程还可以包含 REFERENCING 子句以声明 UDR 中的 SPL 语句可以引用的 OLD 和 NEW 值。
- 触发器例程只能在触发器定义中触发器动作列表的 FOR EACH ROW 部分调用。
- OLD 或 NEW 值的相关变量可以出现在 SPL 和 CASE 表达式的 IF 语句中。
- OLD 值的相关变量不能位于 LET 表达式的左侧。
- 如果 FOR 子句指定了其 INSTEAD OF 触发器操作列表调用触发器例程的视图，那么 NEW 值的相关变量不能位于 LET 表达式的左侧。
- 只有 NEW 值的相关变量可以位于引用相关变量的 LET 表达式的左侧。在这种情况下，FOR 子句必须指定一个表而不是一个视图，并且其操作调用 SPL 例程的触发器不能是 INSTEAD OF 触发器。
- OLD 和 NEW 值都可以在 LET 表达式的右侧。
- Boolean 运算符 SELECTING 、INSERTING 、DELETING 和 UPDATING 在 Boolean 表达式有效的上下文中的触发例程中（并且仅在触发例程和触发动作语句中调用其它 UDR）有效。如果触发事件匹配由操作符的名称引用的 DML 操作，则这些运算符返回 TRUE('t')，否则返回 FALSE（'f'）。
- 如果单个触发事件在同一个表或视图上激活多个触发器，则所有 BEFORE 操作都将在任何 FOR EACH ROW 操作之前发生，并且所有 AFTER 操作在 FOR EACH ROW 操作后发生。不保证同一事件上不同触发器的执行顺序。
- 触发器例程必须用 SPL 语言编写。它们不能用外部语言编写（如 C 或 Java™ 语言），但是它们可包含对外部例程的调用。例如用于触发器内省的 **mi_trigger** 应用程序编程接口。
- 触发器例程不能引用保存点。触发操作对数据值或数据库模式的任何更改必须完全落实或回滚。不支持部分回滚触发的操作。

有关 **mi_trigger** API 的更多信息，请参阅 *GBase 8s DataBlade API 程序员指南*和 *GBase 8s DataBlade API 函数参考*。

如果包含 REFERENCING 子句但省略 FOR 子句，或者包含 FOR 子句但省略 REFERENCING 子句，那么 CREATE PROCEDURE 语句发生错误并失败。.

如果省略 REFERENCING and FOR 子句，则 UDR 不能使用 SELECTING 、INSERTING 、DELETING 和 UPDATING 运算符，并且不能在触发器定义指定的表或视图上声明可表示和操作触发动作中的列值的变量。

有关表上的 Delete 、Insert 、Select 和 Update 触发器的 REFERENCING 子句的语法以及 Delete 、Insert 、Select 和 Update 视图上的 INSTEAD OF 触发器的语法，请参阅 CREATE TRIGGER 语句说明中的 REFERENCING 子句 部分。

## GBase 8s 上的过程名称

由于 GBase 8s 提供***例程重载***，您可以使用相同的名称但不同的参数列表来定义多个用户定义的例程（UDR）。在以下情况中您可能希望重载 UDR ：

- 使用与内置例程相同的名称创建 UDR （如 **equal( )**）来处理新的用户定义的数据类型。
- 您创建在其中子类型从超类型继承数据表示和 UDR 的 ***type hierarchies***。
- 您创建 ***distinct 类型***，它是拥有与现有数据类型相同的内部存储表示的数据类型，但是名称不同，并且没有强制转型就无法与源类型相比较。Distinct 类型从它们的源类型继承 UDR 。

关于唯一标识每个 UDR 的例程特征符的简述，请参阅例程重载以及例程签名 。

## 使用 SPECIFIC 子句来指定特定名称

可以为用户定义的过程声明一个在数据库中唯一的***特定名称***。特定名称当您重载过程时有用。

## DOCUMENT 子句

DOCUMENT 子句中带引号的字符串提供对 UDR 的摘要和描述。该字符串存储在 **sysprocbody** 系统目录表中，适用于 UDR 的用户。

拥有对数据库访问权限的任何人均可查询 **sysprocbody** 系统目录表，以获取对存储在数据库中的一个或全部 UDR 描述。

例如，以下查询获取对 SPL 函数 所显示的 SPL 过程 `raise_prices` 的描述：

```
SELECT data FROM sysprocbody b, sysprocedures p
        WHERE b.procid = p.procid
        --join between the two catalog tables
        AND p.procname = 'raise_prices'
        -- look for procedure named raise_prices
        AND b.datakey  = 'D';-- want user document
```

先前的查询返回以下文本：

```
USAGE: EXECUTE PROCEDURE raise_prices( xxx )
        xxx = percentage from 1 - 100
```

对于外部过程，无论您是否使用 END PROCEDURE 关键字，均可以在 CREATE PROCEDURE 语句末尾包含 DOCUMENT 子句。

## 使用 WITH LISTING IN 选项

WITH LISTING IN 子句指示编译时发送警告的文件名。编译 UDR 之后，此文件包含一条或多条警告消息。该列表文件创建于数据库驻留的计算机上。

如果您不使用 WITH LISTING IN 子句，则编译器不生成警告列表。

在 UNIX™ 上，如果您指定文件名而非目录，将在数据库驻留的计算机上的主目录中创建此列表文件。如果您在此计算机上没有主目录，则在根目录中（名为 "/" 的目录）创建此文件。

在 Windows™ 上，如果果您指定文件名而非目录，则在数据库位于本地计算机的情况下，在当前工作目录中创建此列表文件。否则，缺省目录为 **%GBASEDBTDIR%\bin** 。

## SPL 函数

SPL 函数是用存储过程语言（SPL）编写的 UDR ，且不会返回一个值。要编写并注册 SPL 例程，请使用 CREATE PROCEDURE 语句。在 CREATE PROCEDURE 和 END PROCEDURE 关键字之间嵌入适当的 SQL 和 SPL 语句。还可以将 DOCUMENT 和 WITH FILE IN 选项放在该函数后面。

分析 SPL 例程，（尽可能）优化例程，并以可执行文件的形式存储在系统目录表中。SPL 函数的主题存储在 **sysprocbody** 系统目录表中。关于函数的其他信息存储在其它系统目录表中，包括 **sysprocedures** 、**sysprocplan** 和 **sysprocauth** 。

如果 CREATE PROCEDURE 语句的 Statement Block 部分为空，则当调用函数时，没有操作发生。当您试图建立未编码的过程，可能在开发阶段使用如“虚拟”过程。

如果在参数列表后面指定了可选子句，则您必须在此子句之后紧邻 Statement Block 之前加上分号。

以下示例创建了 SPL 函数：

```
CREATE PROCEDURE raise_prices ( per_cent INT )
        UPDATE stock SET unit_price =
        unit_price + (unit_price * (per_cent/100));
        END PROCEDURE
        DOCUMENT "USAGE: EXECUTE PROCEDURE raise_prices( xxx )",
        "xxx = percentage from 1 - 100 "
        WITH LISTING IN '/tmp/warn_file';
```

## 外部过程

*外部过程*是用数据库服务器支持的外部语言写的过程。（使用 SPL 语言编写的过程不是外部过程。）

要创建 C 的用户定义的过程：

1. 编写不返回值的 C 函数。
2. 编译 C 函数并将编译过的代码存储在共享库中（C 的共享对象文件）。
3. 在数据库服务器中使用 CREATE PROCEDURE 语句注册 C 函数。

要创建以 Java™ 语言编写的用户定义的过程：

1. 写一个 Java 静态方法，它能使用 JDBC 函数与数据库服务器交互。
2. 编译 Java 源并创建一个 JAR 文件（共享对象文件）。
3. 用 EXECUTE PROCEDURE 语句执行 **install_jar( )** 过程，在当前数据库中安装 JAR 文件。
4. 如果 UDR 使用用户定义类型，则使用在 EXECUTE PROCEDURE 语句 中说明的 **setUDTextName( )** 过程在 SQL 数据类型和 Java 类别之间创建映射。
5. 使用 CREATE PROCEDURE 语句注册 UDR 。（如果外部例程返回一个值，则您必须使用 CREATE FUNCTION 语句而非 CREATE PROCEDURE 来注册它。）

并非将外部例程的主体直接存储在数据库中，数据库服务器仅存储包含已编译版本例程的共享对象文件的路径名。数据库服务器通过调用外部目标代码执行外部例程。

您还必须持有要注册的外部过程所在数据库的 Resource 特权或 DBA 特权，和对编写的例程所有的程序语言的 Usage 特权。（有关在 C 语言或 Java 语言上将 Usage 特权授予用户或角色或 PUBLIC 组的语法的信息，请参阅 语言级权限。）

当 IFX_EXTEND_ROLE 配置参数设置成 1 或 ON 时，只有拥有内置 EXTEND 角色的用户才可以创建外部过程。

## 注册用户定义的过程

此示例注册了一个取得类型 LVARCHAR 的一个自变量的名为 check_owner( ) 的 C 用户定义的过程。外部例程参考指定了到存储目标代码的 C 共享库的路径。此库包含一个 C 函数 unix_owner( ) ，它在 check_owner( ) 过程执行期间被调用。

```
CREATE PROCEDURE check_owner ( owner lvarchar )
        EXTERNAL NAME "/usr/lib/ext_lib/genlib.so(unix_owner)"
 LANGUAGE C
 END PROCEDURE;
```

此示例注册了一个以 Java™ 语言编写的名为 showusers( ) 的用户定义过程：

```
CREATE PROCEDURE showusers()
   WITH (CLASS = "jvp") EXTERNAL NAME 'admin_jar:admin.showusers'    LANGUAGE JAVA;
```

EXTERNAL NAME 子句指定了 **showusers( )** 过程的 Java 实现是名为 **showusers( )** 的方法，它驻留在驻留于 **admin_jar** JAR 文件的 **admin** Java 类中。

## 创建数据库对象的所有权

创建所有者特权 UDR 的用户拥有 UDR 执行时它所创建的任何数据库对象，除非已经为该对象指定了某个其他的*所有者*。换句话说，UDR 所有者，而非执行拥有所有者特权的 UDR 用户，是 UDR 创建的任何数据库对象的所有者，除非在创建数据库对象的 DDL 语句中指定了另一个所有者。

然而在拥有 DBA 特权的 UDR 情况下，执行 UDR 的用户，而非 UDR 所有者，拥有 UDR 创建的任何数据库对象，除非在 UDR 中为数据库对象指定某个其他的所有者。

有关示例，请参阅已创建数据库对象的所有权的 CREATE FUNCTION 语句的描述。

## 2.35 CREATE PROCEDURE FROM 语句

使用 CREATE PROCEDURE FROM 语句存取用户定义的过程。CREATE PROCEDURE 语句的实际文本驻留在单独的文件中。

该语句是 SQL ANSI/ISO 标准的扩展。您可以在 GBase 8s ESQL/C 中使用此语句。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *file* | 包含 CREATE PROCEDURE 语句的全文本的文件的路径名和文件名。缺省路径名为当前目录。 | 必须存在，并且仅可包含一个 CREATE PROCEDURE 语句。另见 持有文件的缺省目录 | 特定于操作系统 |
| *file_var* | 包含*文件*规范的程序变量的名称 | 必须是字符数据类型；其内容有与 *file* 相同的限制 | 特定于语言 |

**用法**

不能直接在 GBase 8s ESQL/C 程序中创建用户定义的过程。这就是说，程序不能包含 CREATE PROCEDURE 语句。

要在 ESQL/C 程序中使用用户定义的过程：

1. 用 CREATE PROCEDURE 语句创建源文件。
2. 使用 CREATE PROCEDURE FROM 语句来将此源文件的内容发送到数据库服务器用于执行。

文件仅能包含一个 CREATE PROCEDURE 语句。

例如，假设以下 CREATE PROCEDURE 语句是在称为 raise_pr.sql 的单独文件中：

```
CREATE PROCEDURE raise_prices( per_cent INT )
                UPDATE stock -- increase by percentage;
                SET unit_price = unit_price +
                ( unit_price * (per_cent / 100) );
                END PROCEDURE;
```

在 GBase 8s ESQL/C 程序中，您可以使用以下 CREATE PROCEDURE FROM 语句存取 raise_prices( ) SPL 过程：

```
EXEC SQL create procedure from 'raise_pr.sql';
```

如果您不确定文件中的 UDR 是否返回值，请使用 CREATE ROUTINE FROM 语句。

当 IFX_EXTEND_ROLE 配置参数设置成 ON 时，只有拥有内置 EXTEND 角色的用户才可以创建外部例程。

当 IFX_EXTEND_ROLE 配置参数设置成 1 或 ON 时，只有由数据库服务器管理员（DBSA）授予内置 EXTEND 角色的用户才能创建外部例程。此外，您对要注册例程所在的数据库必须至少持有的 Resource 存取权限。还必须对编写例程所使用的程序语言拥有 Usage 权限。（有关使用 C 语言将 Usage 特权授予用户或角色的语法，请参阅 语言级权限 。）

用户定义过程，类似用户定义函数，使用创建时有效的对照顺序。请参阅 SET COLLATION 语句 获取关于使用非缺省对照的信息。

### 持有文件的缺省目录

数据库服务器将此指定的文件名（以及任何路径名）看作相关的。

在 UNIX™ 上，如果您指定一个简单文件名而不是完整的路径名作为 ***file*** 参数，则客户机应用程序在数据库驻留的计算机上的主目录中寻找文件，如果您在此计算机上没有主目录，则缺省目录为根目录。

在 Windows™ 上，如果您指定文件名而非目录名为 ***file*** 参数，则客户端应用程序在您的当前工作目录中寻找文件（如果数据库在本地计算机上）。否则，缺省目录为 **%GBASEDBTDIR%\bin** 。

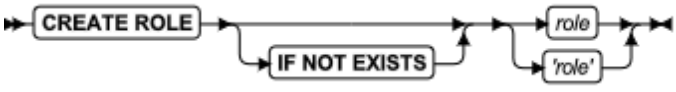
**重要：** GBase 8s ESQL/C 预处理器不处理您指定的文件的内容。它只将内容发送到数据库服务器用于执行。因此对您在 CREATE PROCEDURE FROM 中指定的文件是否实际包含 CREATE PROCEDURE 语句没有语法检查。然而，要提高代码的可读性，建议匹配这两个语句。

## 2.36 CREATE ROLE 语句

使用 CREATE ROLE 语句声明并注册新的角色。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *role* | 此处为 DBA 创建的角色声明的名称 | 在数据库的 *role* 和用户名称中必须是唯一的。最大字节数为 32 | 所有者名称 |

**用法**

CREATE ROLE 创建一个新角色并在系统目录中注册。角色可以用来将数据库对象上的一组权限标识符和一组访问特权相关联。系统目录维护被授权给用户或其它角色的角色（以及他们相应的特权）的信息。

只有数据库管理员（DBA）可以使用 CREATE ROLE 创建一个新角色。DBA 可以将完成某些工作任务所需的特权（如 **engineer** ）指定给某个角色，然后可以使用 GRANT 语句将该角色指定给某个特定的用户，而不是将相同的特权集授权给每个用户。

***角色***名称是***权限标识符***。它不能是对数据库服务器或对数据库服务器的操作系统已知的用户名。***角色***名称不能已经列在 **sysusers** 系统目录表的 **username** 列，也不能已经列在 **systabauth** 、**syscolauth** 、**sysfragauth** 、**sysprocauth** 或 **sysroleauth** 系统目录表的 **grantor** 或 **grantee** 列中。

***角色***名称不能与已经列在 **sysxtdtypeauth** 系统目录表的 **grantor** 或 **grantee** 列中的任何用户或角色名称匹配，也不能与任何内置角色（ EXTEND 或 DBSECADM ）匹配。

如果包含了 IF NOT EXISTS 关键字，则当指定名称的角色已经在当前数据库中注册时，数据库服务器不采取操作（而非向应用程序发生异常）。

创建角色之后，DBA 可以使用 GRANT 语句将该角色指定给 PUBLIC 、用户、或其它角色，并授予该角色特定的特权。（然而，角色不能持有数据库级别特权。） 角色被成功授权到用户或 PUBLIC 后，用户必须使用 SET ROLE 语句来启用该角色。只有那样用户才能使用角色的特权。

例如，要创建角色 engineer ，请输入以下语句：

```
CREATE ROLE engineer;
```

要将访问特权授予角色 engineer ，DBA 可以发出在被授予者列表中包括 engineer 的 GRANT 语句：

```
GRANT USAGE ON LANGUAGE SPL TO engineer;
```

要将角色 engineer 指定给用户 kaycee ，DBA 可以发出这样的语句：

```
GRANT engineer TO kaycee;
```

要激活角色 engineer ，用户 kaycee 必须发出以下的语句：

```
SET ROLE engineer;
```

如果该 SET ROLE 语句成功，则用户 **kaycee** 将获得除了 **kaycee** 作为个人或作为 PUBLIC 已经持有的特权意外的授权给角色 **engineer** 的所有特权。

一个用户可以被授予多个角色，但是在某一时刻对任一用户只能启用一个非缺省的角色，此角色由 SET ROLE 指定。

需要 SET ROLE 来显式地启用角色的一个例外是 DBA 在 GRANT DEFAULT ROLE ***role*** TO ***user*** 语句中指定的任何缺省角色。如果此语句执行成功，则当 ***user*** 连接到数据库时，缺省角色自动被启用。任何角色都可以作为缺省角色。（类似的， GBase 8s DBSA 授予 EXTEND 角色的用户不需要执行 SET ROLE 就能创建和删除外部例程和共享库。）

CREATE ROLE 当与 GRANT 和 SET ROLE 语句一起使用时，允许 DBA 为角色创建一组特权，然后将角色授权给许多用户，而不是将同一特权集合逐一授权给许多用户。

使用 GRANT DEFAULT ROLE 和 SET ROLE DEFAULT 语句，***default roles*** 使 DBA 可以为角色创建特权，该角色在任何持有那个角色的用户连接到数据库时将自动激活。当应用程序执行需要特定访问权限的操作时，此功能很有用，但是应用程序并不包括 SET ROLE 语句。

REVOKE 语句可以取消角色的访问特权，从角色中除去用户，或为一个或多个用户取消角色的缺省状态。角色保持存在直到 DBA 或者使用 WITH GRANT OPTION 关键字授权其角色的用户使用 DROP ROLE 语句来删除角色。

**重要：** 用户定义角色（GRANT 语句分配给该角色的自由裁量存取权限）的范围是当前数据库。当 GRANT DEFAULT ROLE 或者 SET ROLE 语句激活角色时，该角色和它的特权只在当前数据库中生效。作为安全防范措施，用户从角色接收自由访问特权不能通过视图或触发器操作提供当前数据库之外的访问。

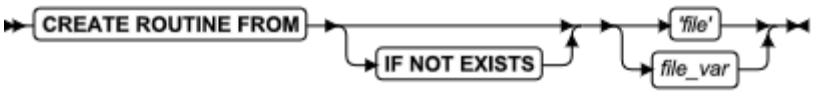
## 2.37 CREATE ROUTINE FROM 语句

使用 CREATE ROUTINE FROM 语句通过引用驻留在单个文件中的 CREATE FUNCTION 语句或 CREATE PROCEDURE 语句的文本来注册 UDR。

该语句是 SQL ANSI/ISO 标准的扩展。

你可以在 ESQL/C 中使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *file* | 包含 CREATE PROCEDURE 或 CREATE FUNCTION 语句的全文本的文件的路径名和文件名。缺省路径名为当前目录。 | 必须存在并且仅能包含一个 CREATE FUNCTION 或 CREATE PROCEDURE 语句 | 特定于操作系统 |
| *file_var* | 包含*文件*规范的程序变量的名称 | 必须是字符数据类型；内容必须满足 *file* 限制 | 特定于语言 |

**用法**

ESQL/C 程序不能使用 CREATE FUNCTION 或 CREATE PROCEDURE 语句直接定义 UDR 。您可以这样做：

1. 使用 CREATE FUNCTION 或 CREATE PROCEDURE 语句创建源文件。
2. 在 ESQL/C 程序中执行 CREATE ROUTINE FROM 语句将此源文件的内容发送到数据库服务器用于执行。您指定的文件只能包含一个 CREATE FUNCTION 或 CREATE PROCEDURE 语句。

您提供的文件规范是相对的。如果您不提供路径名，客户端应用程序在当前目录中寻找文件。

如果您不知道编译时文件中的 UDR 是否是一个函数或过程，请使用 GBase 8s ESQL/C 程序中的 CREATE ROUTINE FROM 语句。如果您确实知道 UDR 是否是一个函数或过程，建议您使用匹配语句来存取源文件：

- 要存取用户定义的函数，请使用 CREATE FUNCTION FROM 。
- 要存取用户定义的过程，请使用 CREATE PROCEDURE FROM 。

当 IFX_EXTEND_ROLE 配置参数设置成 1 或 ON 时，只有被数据库服务器管理员（DBSA）授权内置 EXTEND 角色的用户才能可以创建外部例程。此外，您对要注册例程所在的数据库必须至少持有的 Resource 存取权限。还必须对编写例程所使用的程序语言拥有 Usage 权限。（有关使用 C 语言将 Usage 特权授予用户或角色的语法，请参阅语言级权限。）

例程使用创建时有效的对照顺序。请参阅 SET COLLATION 语句 获取有关使用非缺省对照的信息。

示例

以下语句通过参考 del_ord.sql 文件中的文本在 UDR 注册。

```
EXEC SQL CREATE ROUTINE FROM 'del_ord.sql';
```

ESQL/C 源代码示例：

```
#include <stdio.h>

main()
{

      printf( "CREATE ROUTINE FROM ESQL Program running.\n\n");
      EXEC SQL WHENEVER ERROR STOP;
      EXEC SQL connect to 'stores_demo';

      EXEC SQL CREATE ROUTINE FROM 'del_ord.sql';

      EXEC SQL disconnect current;
      printf("\nCREATE ROUTINE Sample Program over.\n\n");

    exit(0);
}
```

del_ord.sql

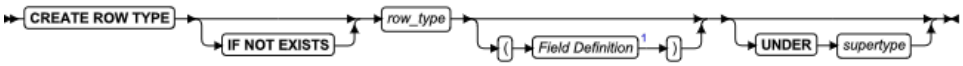
```
CREATE FUNCTION delete_order( p_order_num int) RETURNING int, int;
    DEFINE item_count int;
    SELECT count(*) INTO item_count FROM items
        WHERE order_num = p_order_num;
    DELETE FROM orders WHERE order_num = p_order_num;
    RETURN p_order_num, item_count;
END FUNCTION;
```

# 2.38 CREATE ROW TYPE 语句

使用 CREATE ROW TYPE 语句创建命名的 ROW 类型。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *row_type* | 此处为已命名的新的 ROW 数据类型 | 请参阅 创建子类型的过程 | 标识符 |
| *supertype* | 类型继承层次结构中超类型的名称 | 必须已经作为已命名的 ROW 类型在数据库中存在 | 数据类型 |

**用法**

CREATE ROW TYPE 语句声明已命名的 ROW 数据类型并在当前系统目录中注册。可以将已命名的 ROW 数据类型分配给表或视图，以创建***类型表***或***类型视图***。您也可以将列定义为命名 ROW 类型。尽管您能分配一个 ROW 类型到表来定义表模式，但是 ROW 数据类型与表行不同。表行由一个或多个***列***构成；ROW 数据类型由一个或多个***字段***构成，使用 Field Definition 语法定义。

已命名的 ROW 数据类型在大多数您可指定数据类型的上下文中有效。已命名的 ROW 类型是***强类型***。没有两个已命名的 ROW 类型是相等的，即使它们结构上相等。

没有标识符的 ROW 类型称为 ***unnamed ROW 类型***。任何两个未命名 ROW 类型被认为是相等的（如果它们结构上相等）。有关更多信息，请参阅 ROW 数据类型 。

在命名 ROW 类型列上的特权与任何其它列上的特权相同。有关更多信息，请参阅 表级权限。（要查看列上拥有的特权，请检查 **syscolauth** 系统目录表，该系统目录表在 *《GBase 8s SQL 指南：参考》* 中有描述。）

如果您包含可选的 IF NOT EXISTS 关键字，则当指定名称的已命名的 ROW 已经在当前数据库中注册时，数据库服务器不采取操作（而非向应用程序发送异常）。

## 在命名 ROW 数据类型上的特权

在类型表上操作所需的自由访问特权（被指定已命名的 ROW 数据类型的表）与任何表上的特权相等。有关更多信息，请参阅 表级权限。下表显示要创建命名的 ROW 类型必须具有哪些特权。

| 任务 | 需要的特权 |
|---|---|
| 创建命名 ROW 类型 | 数据库上的资源特权 |
| 按照超类型下子类型创建命名 ROW 类 | 在超类型的特权下，以及 Resource 特 |

| 任务 | 需要的特权 |
| --- | --- |
| 型 | 权 |

有关 Resource 和 Under 特权以及特权上下文中 ALL 关键字的信息，请参阅 GRANT 语句 。

要找出 ROW 类型上存在什么特权，在 **sysxtdtypes** 系统目录表中检查*所有者*名称，并在 **sysxtdtypeauth** 系统目录表中检查可能已经授权给用户或角色的 ROW 类型上的特权。

要找出在给定表上有什么特权，检查 systabauth 系统目录表。有关系统目录表的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 。

## 继承和命名 ROW 类型

已命名的 ROW 类型属于继承层次结构，作为子类型或超类型。使用 CREATE ROW TYPE 语句中的 UNDER 子句将已命名的 ROW 数据类型作为现有的 ROW 数据类型的子类型创建。

超类型必须也是已命名的 ROW 数据类型。如果在现有的超类型下创建已命名的 ROW 数据类型，那么新的类型名称 ***row_type*** 称为子类型的名称。

当将命名 ROW 类型创建为子类型时，子类型继承超类型的所有字段。另外，当创建此子类型时，您可以向其添加新的字段。新的字段单独特定于子类型。

不能将继承层次结构中的 ROW 类型替换其超类型或子类型。例如，考虑 person_t 是超类型且 employee_t 是子类型的类型层次结构。如果列是类型 person_t ，则该列仅能包含 person_t 数据。它不能包含 employee_t 数据。同样地，如果列是类型 employee_t ，则列仅能包含 employee_t 数据。它不能包含 person_t 数据。

## 创建子类型

大多数情况下，当命名 ROW 类型作为另一种命名 ROW 类型（其超类型）的子类型创建时候您添加新的字段。要创建命名的 ROW 类型的字段，请使用 字段定义 中描述的字段定义子句。当创建子类型时，必须使用 UNDER 关键字来将超类型与希望创建的已命名 ROW 类型相关联。下一个示例将在 person_t 类型下创建 employee_t 类型：

```
CREATE ROW TYPE employee_t (salary NUMERIC(10,2),
        bonus NUMERIC(10,2)) UNDER person_t;
```

**employee_t** 类型继承 **person_t** 的所有字段并有两个另外的字段：**salary** 和 **bonus** ；但是 **person_t** 类型没有改变。

## 类型层次结构

当创建子类型时，会创建*类型层次结构*。在类型层次结构中每个您创建的子类型从单个超类型继承其属性。如果在 **person_t** 下创建命名 ROW 类型 **customer_t** ，**customer_t** 继承 **person_t** 的所有字段。如果您在 **customer_t** 下创建另一个命名 ROW 类型 **salesrep_t** ，则 **salesrep_t** 继承 **customer_t** 的所有字段。

**因此，salesrep_t 继承所有 customer_t 从 person_t 继承的字段，以及所有特别为 customer_t 定义的字段。有关类型继承的讨论，请参阅 *GBase 8s SQL 教程指南* 。**

## 创建子类型的过程

在您将命名 ROW 类型作为继承层次结构中子类型创建之前，请检查以下消息：

- 验证您已授权创建新的数据类型。必须在数据库上拥有 Resource 特权。可以在 **sysusers** 系统目录表中找到此信息。
- 验证超类型存在。可以在 **sysxtdtypes** 系统目录表中找到此信息。
- 验证您已授权创建该超类型的子类型。必须拥有超类型上的 Under 特权。可以在 **sysusers** 系统目录表中找到此信息。
- 验证您已命名 ROW 类型声明的名称是唯一的。在兼容 ANSI 的数据库中，***owner.type*** 组合必须在数据库中是唯一的。要验证新的数据类型的名称是否唯一的。在不兼容 ANSI 的数据库中，名称必须在数据库中的数据类型名称中是唯一的，要验证新的数据类型的名称是否唯一，请检查 **sysxtdtypes** 系统目录表，名称必须不是现有数据类型的名称。
- 如果您正在为 ROW 类型定义字段，则检查没有复制字段名称同时存在于新的和继承的字段中。

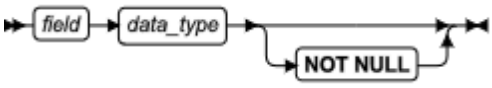
**重要：** 当创建子类型时，不能重新定义它为其超类型继承的字段。如果您试图重新定义这些字段，则数据库服务器返回一个错误。

不能将约束应用到已命名 ROW 数据类型，但是可以在创建或更改使用已命名 ROW 类型的表时指定约束。您也可以在某个单独的 ROW 类型的字段上指定 NOT NULL 约束。

## 字段定义

使用 Field Definition 子句在已命名的 ROW 类型中定义新的字段。

字段定义




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | 字段的数据类型 | 请参阅 对序列和简单大对象数据类型的限制 。 | 标识符 |
| *field* | *row_type* 中字段的名称 | 在此 ROW 类型及其超类型的字段名称中必须是唯一的 | 标识符 |

当已命名的 EOW 类型的类型表被创建时，已命名的 ROW 类型字段上的 NOT NULL 约束适用于相应的列。

## 对序列和简单大对象数据类型的限制

序列和简单大对象数据类型不能嵌套在表中。因此，如果 ROW 类型包含 BYTE 、TEXT 、SERIAL 、BIGSERIAL 或 SERIAL8 字段，则不能使用 ROW 类型在并非基于 ROW 类型的表中定义列。例如，以下代码示例产生一个错误：

```
CREATE ROW TYPE serialtype (s serial, s8 serial8);
        CREATE TABLE tab1 (col1 serialtype); --INVALID CODE
```

不能创建有存储在单独存储空间中的 BYTE 或 TEXT 值的 ROW 类型。即，不能使用 IN 子句指定存储位置。例如，以下示例产生一个错误：

```
CREATE ROW TYPE row1 (field1 byte IN blobspace1); --INVALID CODE
```

表层次结构只能包含一个 SERIAL 、BIGSERIAL 或 SERIAL8 列。如果超级表包含 SERIAL 列，则没有子表能够包含 SERIAL 列。（但是如果没有其它子表包含 BIGSERIAL 或 SERIAL8 列，则子表可以拥有 BIGSERIAL 或 SERIAL8 列。）从而，当创建表层次结构所依据的已命名的 ROW 类型时，它们最多可以包含这些类型中的一个 SERIAL 和一个 BIGSERIAL 或 SERIAL8 字段。

不能用 CREATE ROW TYPE 语句设置起始 SERIAL 、BIGSERIAL 或 SERIAL8 值。要修改序列字段的值，必须使用 ALTER TABLE 语句的 MODIFY 子句或使用 INSERT 语句插入大于当前最大（或缺省）序列值的一个值。 ROW 类型中的序列字段在表层次结构上具有性能影响。要将数据插入超级表（或其超级表）包含序列计数器的子表中，数据库服务器还必须打开超级表，更新序列值并关闭该超级表，从而添加额外开销。

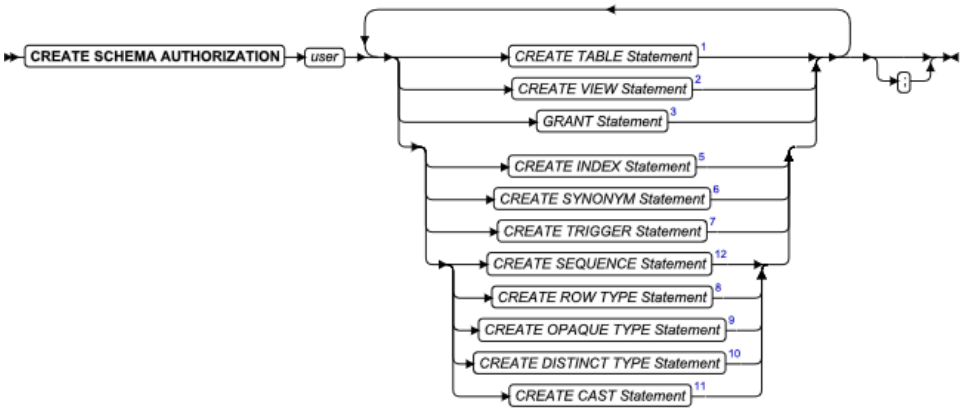
在对 SERIAL 、BIGSERIAL 或 SERIAL8 数据类型发出的限制或性能与您的设计目标冲突的上下文中，可能要考虑使用序列对象来模拟序列字段或序列列的功能。

# 2.39 CREATE SCHEMA 语句

使用 CREATE SCHEMA 语句将数据定义语言（DDL）和 GRANT 语句块作为一个单位发出。

在 DB-Access 中使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *user* | 拥有语句创建的数据库对象的用户 | 如果有 DBA 特权，则可以指定任意用户的名称。否则，必须具有 Resource 特权，而且必须指定您子句的用户名。 | 所有者名称 |

**用法**

CREATE SCHEMA 语句允许 DBA 为 CREATE SCHEMA 语句创建的所有数据库指定一个所有者。在创建存储该对象的数据库之前，不能发出 CREATE SCHEMA 。

带有 Resource 特权的用户可为他们自己创建模式。在这情况下，*用户*名称必须是正在运行 CREATE SCHEMA 语句的带有 Resource 特权的人的名称。带有 DBA 特权的任何人也可以为其他人创建模式。在这种情况下，***user*** 可以指定正在运行 CREATE SCHEMA 语句的人以为的用户。

可以任意逻辑顺序放置 CREATE 和 GRANT 语句，如下例所示，直到遇到分号 (;)或文件结束符时，才认为语句是 CREATE SCHEMA 语句的一部分。

```
CREATE SCHEMA AUTHORIZATION sarah
        CREATE TABLE mytable (mytime DATE, mytext TEXT)
        GRANT SELECT, UPDATE, DELETE ON mytable TO rick
        CREATE VIEW myview AS
        SELECT * FROM mytable WHERE mytime > '12/31/2004'
        CREATE INDEX idxtime ON mytable (mytime);
```

### 在 CREATE SCHEMA 中创建数据库对象

即使您没有明确地命名每个数据库对象，CREATE SCHEMA 语句创建的所有数据库对象都归*用户*所有。如果您是 DBA ，则可以为另一个用户创建数据库对象。如果您不是 DBA ，则指定您自己以外的所有者会导致错误消息。

只能用 CREATE SCHEMA 语句授予特权；不能使用 CREATE SCHEMA 撤销或删除特权。

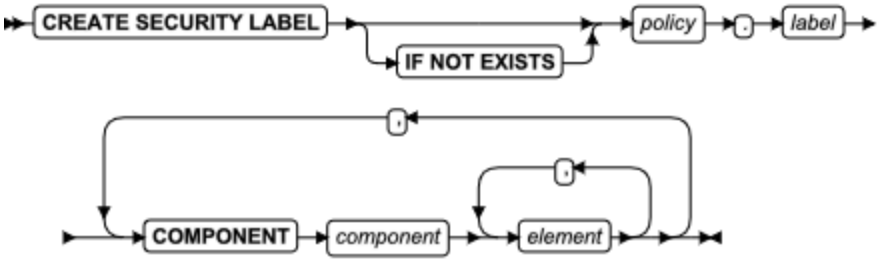
如果在 CREATE SCHEMA 语句之外创建数据库对象或使用 GRANT 语句，则在使用 **-ansi** 标志或设置 **DBANSIWARN** 时会接收到警告。

## 2.40 CREATE SECURITY LABEL 语句

使用 CREATE SECURITY LABEL 语句在当前数据库中为指定的安全策略定义新的安全标签，并标识其组件和组件名称。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *component* | 安全标签组件 | 必须作为指定 *policy* 的组件存在于当前数据库中，且在此 *label* 的组件名称中必须唯一 | 标识符 |
| *label* | 为此标签声明的名称 | 在此安全 *policy* 的安全标签名称中必须唯一 | 标识符 |
| *element* | 指定的 *component* 的一个元素 | 当其 *component* 定义完成或最后的更改完成时，必须已经定义。如果 *component* 是一个数组，则只能指定一个 *element* | 引用字符串 |
| *policy* | 此 *label* 的安全策略 | 必须存在于数据库中 | 标识符 |

**用法**

***安全标签*** 是支持指定安全策略的已命名的数据库对象。安全标签可应用于用户，或当前数据库中表的行或列（或者行和列）。当持有安全标签的用户尝试存取有安全标签的数据时，数据库服务器将列或行的安全标签和用户的安全标签作为决定是否允许用户存取数据的考虑条件。

每个安全标签存储以下种类信息：

- 它标识标签支持的现有安全策略。
- 它标识至少一个，但不超过 16 个标签支持的安全策略的现有的组件。
- 它标识一个或多个安全标签的每个组件的现有的元素。（只有 SET 或 TREE 类型的安全标签组件能在同一安全标签中包含多个元素。）

只有 DBSECADM 能发出此语句。当 CREATE SECURITY LABEL 语句执行成功，它会在 **sysseclabels** 系统目录表中注册指定的 ***label*** 名称，与安全 ***policy*** 关联的数字标识符，和它的安全标签 ***components*** 的基数。

如果您包含可选的 IF NOT EXISTS 关键字，则当指定名称的安全标签已经在当前数据库中注册时，数据库服务器不采取操作（而非向应用程序发送异常）。

## 安全标签的组件和元素

类似安全策略，一个安全标签必须拥有至少一个，但不超过 16 个的组件。CREATE SECURITY LABEL 语句无法不是指定安全策略的组件的安全标签组件。同一 ***component*** 名称在 CREATE SECURITY LABEL 语句中只能指定一次。这些组件必须已经存在于数据库中，由此 DBSECADM 可使用 CREATE SECURITY LABEL COMPONENT 语句注册它们。

安全标签组件可以是 ARRAY 、SET 或 TREE 类型，可在 CREATE SECURITY LABEL COMPONENT 语句中描述。对于 ARRAY 类型的 ***component*** ，***element*** 列表只能标识一个元素。对于 SET 或 TREE 类型的组件，***element*** 列表可以标识创建组件完成时（或其最后修改完毕后）已定义的多个组件元素。有关安全标签组件的结构和语义的更多信息，请参阅 CREATE SECURITY LABEL COMPONENT 语句。

以下示例为同一安全策略 MegaCorp 创建名为 label1 的安全标签 。该标签使用两个安全标签组件，称为 levels 和 compartments 。每个组件有一个元素，分别称为 VP 和 Marketing ：

```
CREATE SECURITY LABEL MegaCorp.label1
        COMPONENT levels 'VP',
        COMPONENT compartments 'Marketing';
```

要使该示例有效，则 **levels** 和 **compartments** 组件，以及它们的安全标签组件，**VP** 和 **Marketing** 元素，必须在先前执行的 CREATE SECURITY LABEL COMPONENT 语句中定义。

在下个示例中，DBSECADM 对同一 MegaCorp 安全策略创建了名为 label2 的安全标签 f。此标签使用了三个标签组件，分别为 levels 、compartments 和 groups ，其中两个组件有一个元素，另一个组件有两个元素：

```
CREATE SECURITY LABEL MegaCorp.label2
        COMPONENT level 'Director',
        COMPONENT compartments 'HR', 'Finance',
        COMPONENT groups 'EntireRegion';
```

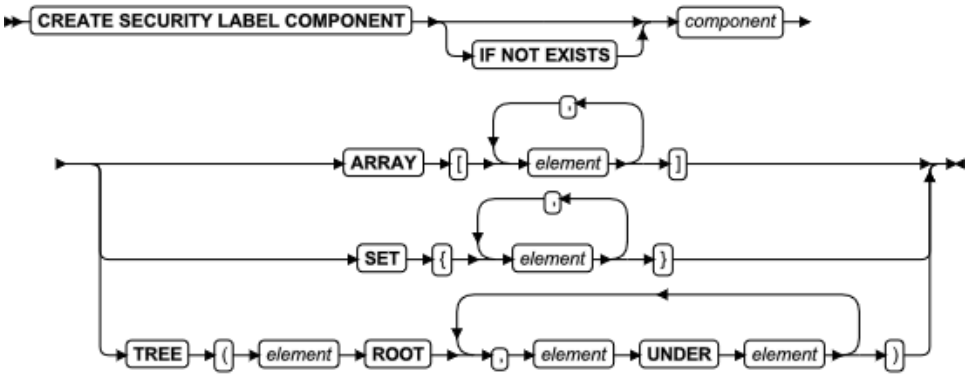
这些示例说明了安全标签的组件可以是其标签支持的安全策略的组件的子集，且多个安全标签可支持同一安全策略。

# 2.41 CREATE SECURITY LABEL COMPONENT 语句

使用 CREATE SECURITY LABEL COMPONENT 语句在当前数据库中定义新的安全标签组件和其元素。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *component* | 此处为此组件声明的名称 | 在当前数据库中的安全标签组件中必须唯一 | 标识符 |
| *element* | 此处定义的组件元素 | 在该 *component* 的元素中必须唯一，且不能多于 32 字节。左括号（ ( ）和右括号（ ) ）、逗号（ , ）和冒号（ : ）是无效字符 | 引用字符串 |

**用法**

只有 DBSECADM 能发出 CREATE SECURITY LABEL COMPONENT 语句，来定义***安全标签组件***。这是定义一个或多多个逻辑类别的数据库对象，其值可以在安全策略中使用以确定用户的读取或写入数据的请求是接受还是拒绝。安全组件可具有的所有有效单个值的集合由该语句为组件指定的安全标签组件的集合来定义。

安全标签组件实施的逻辑类别由 DBSECADM 在设计***安全策略***的过程中标识，它是基于标签访问控制（LBAC）的核心构造。然而，要在数据库中实施此安全功能，DBSECADM 必须以下列顺序创建安全对象：

1. 一个或多个***安全标签***集合，每个集合可由 CREATE SECURITY LABEL COMPONENT 语句定义。该语句指定安全做件的名称、其值范围的结构和此组件可被分配给将安全策略应用与数据或用户的安全标签的可能值。
2. 一个或多个***安全策略***，每个策略可通过 CREATE SECURITY POLICY 语句定义，它可指定应用于数据和尝试对数据库中的安全策略包含的数据执行读取或写入操作的用户，一个或多个组件列表和角色集合。安全策略总是包含 CREATE SECURITY POLICY 值的组件的所有的元素。
3. 可使用 CREATE SECURITY LABEL 定义***安全标签***，它为每个标签支持的安全策略的一个或多个组件指定一个或多个值。此安全标签可以应用于数据和用户。一个安全标签的所有组件必须是同一安全策略的组件，但是多个安全策略和多个安全标签可以共享同一组件。

CREATE SECURITY LABEL COMPONENT 定义的一个安全标签一般只包含一个安全组件元素的子集。

如果您包含可选的 IF NOT EXISTS 关键字，则当指定名称的安全标签组件已经在当前数据库中注册时，数据库服务器不采取操作（而非向应用程序发生异常）。

请参阅 GRANT Security 和 REVOKE Security 语句以获取安全标签和豁免安全策略的规则是如何定义用户或角色的 LABC 凭证的信息。

请参阅 CREATE TABLE 和 ALTER TABLE 语句获取安全标签如何与数据库表或表中的一个数据行关联的信息。

## 安全标签组件的类型和元素

安全标签组件它自己包含一个或多个 CREATE SECURITY LABEL COMPONENT 语句声明为字符串变量的***元素***。这些元素定义对此组件有效的值的集合。

当 CREATE SECURITY LABEL 语句成功执行，GBase 8s 更新了具有以下新条目的数据库的系统目录：

- 它在 **sysseclabelcomponents** 表中创建新行以注册新的组件。
- 对于新组件的每个***元素***，它在 **sysseclabelcomponentelements** 表中创建新行。

安全标签组件必须定义为三种组件类型之一。紧跟在***组件***名称声明之后的 ARRAY 、SET 或 TREE 关键字指定组件类型，它必须跟随在此安全组件的***元素***列表之后。 这些元素定义组件在安全策略内可具有的值的集合。对于所有安全标签策略的三种类型，元素集具有以下限制：

- 安全组件不能拥有超过 64 个的元素。
- 安全组件的每个元素都是不超过 32 字节的带引号的字符串变量。
- 在带引号字符串变量中的字符不能包含左括号（ ( ）或右括号（ ）），逗号（ , ）或冒号（ : ），但是 **DB_LOCALE** 设置支持的其它符号是有效的，包括空格符（ASCII 32）。
- 每个元素在同一安全策略组件的元素中必须唯一，但是同一引号字符串变量还可以是其它安全标签组件的元素。

组件中每个元素的定义隐含与数据库表或单个数据行相关联的安全标签的数据敏感性的级别，还影响了拥有读取或写入由同一标签或指定该组件的一个或多个元素的不同标签保护的数据的安全标签的用户的安全凭证。

就像其它可定义数据库对象的数据库 SQL 数据定义语言 ，CREATE SECURITY LABEL COMPONENT 必须为每个组件元素指定一个字符值，而非占位符。要更改现有安全标签组件的定义，DBSECADM 可使用 ALTER SECURITY LABEL COMPONENT 向 ARRAY 、SET 或 TREE 组件中插入新元素。然而，要删除或重命名一个或多个组件的单个元素， DBSECADM 必须使用 DROP SECURITY LABEL COMPONENT 语句销毁现有的组件，然后重新发出 CREATE SECURITY LABEL COMPONENT 语句来创建新的组件，定义所需的组件结构内的元素值集。

## ARRAY 组件

ARRAY 类型的安全标签组件是不超过 64 个元素的有序集合。每个元素定义了一个对安全策略内的组件有效的值。声明的元素的顺序是十分重要的，因为它定义数据敏感性的降序顺序，其中每个

连续的元素在数据敏感性上排名低于前面的元素。ARRAY 的标签元素集和其逗号分隔符必须包含在一对方括号（[ ... ] ）之间。

当在安全标签的定义中指定 ARRAY 组件时，该标签只能指定一个此组件的元素作为组件的值。

以下示例定义名为 aquilae ARRAY 类型的安全标签组件，其是五个元素的顺序集合，这五个元素分别为 imperator 、tribunus 、centurio 、miles 和 asinus ：

```
CREATE SECURITY LABEL COMPONENT aquilae
        ARRAY [ "imperator", "tribunus", "centurio", "miles", "asinus" ];
```

此处具有最高数据敏感性的组件元素是 imperator ，asinus 具有最低的数据敏感性，具有 tribunus 数据敏感性的数据排在有 centurio 数据敏感性的数据之前，有 imperator 数据敏感性的数据之后。

在多维度安全策略的一些维度可以被映射到单调递减的单个标度的上下文中，适用 ARRAY 类型的组件。

## SET 组件

SET 类型的安全标签组件是指不超过 64 个元素的无序集合。SET 的每个元素都是不超过 32 字节的字符串常量，且在此组件中必须唯一，但是同一值可在其它组件中使用。SET 组件的元素声明的顺序对这些元素识别的类别的数据敏感性并不重要。这些元素和其逗号分隔符必须包含在一对大括号（{ ... } ）之间。

当在安全标签的定义中指定 SET 组件时，此标签可以指定该组件一个或多个元素作为其的有效值。

在以下示例中，DBSECADM 定义了一个称为 departments 的安全标签组件，它是三个元素的无序集合，这三个元素分别为 Marketing 、HR 和 finance ：

```
CREATE SECURITY LABEL COMPONENT departments
        SET { 'Marketing', 'HR', 'Finance' };
```

就像所有 SET 类型的组件，这些元素声明的顺序意味着在数据敏感性上没有相对的排名。

SET 类型的组件适用于多维度安全策略的一些维度可表示为名义类别的上下文中，没有任何逻辑基础用于单调的比例对它们排序，也不用将它们排列在层次结构中。

## TREE 组件

TREE 类型的安全标签组件具有层次结构的逻辑拓扑（即，没有循环的简单图），其具有单个根节点和不超过 63 个附加节点。必须首先列出根节点的字符串变量，并且后面必须跟 ROOT 关键字。每个后续声明的节点的字符串常量必须后跟关键字 UNDER 和一些先前声明的节点的字符串常量。TREE 组件的元素集（包括它们的 ROOT 和 UNDER 关键字及逗号分隔符）必须包含在一对（( ... )）括号之间。

在 UNDER 关键字之后指定的标签元素称为同一 UNDER 关键字（称为该父元素的 ***child***）之前的标签元素的 ***parent***。如果没有在同一语句中声明 UNDER 关键字之后的节点名称，则 CREATE SECURITY LABEL COMPONENT 语句发生错误并失败。

指定为 TREE 组件的根节点的字符串常量具有最高的数据敏感性。对于用户读取或写入受保护的数据，用户安全标签的每个 TREE 组件必须包括数据行安全标签的 TREE 组件中的至少一个元素，或者一个这样元素的祖先。例如，如果 "**Beta**" 声明为 UNDER "**Alpha**" ，"**Gamma**" 声明为 UNDER "**Beta**" 则 "**Gamma**" 的数据敏感度低于 "**Alpha**" 。只有在同一父子关系链中的元素才能在其数据敏感性中进行比较。

下一示例定义了一个 TREE 结构并具有六个节点的名为 Oakland 的安全标签组件：

```
CREATE SECURITY LABEL COMPONENT Oakland
        TREE ( 'Port' ROOT,
        'Downtown' UNDER 'Port',
        'Airport' UNDER 'Port',
        'Estuary' UNDER 'Airport',
        'Avenues' UNDER 'Downtown',
        'Hills' UNDER 'Avenues');
```

此处的根节点是 Port，它具有最高数据敏感性。在此层次结构中，Downtown 、Avenues 和 Hills 元素代表数据敏感性的降序级别，Airport 元素的数据敏感性比 Estuary 元素高。在此示例中，UNDER 关键字指定为父节点的四个组件元素在被包括在 UNDER 规范之前被声明。如果 Avenues 节点声明在 Airport 节点之前，则此示例的修改版本也是有效的，但是如果 Hills 节点声明先于 Avenues 节点，则会导致错误。

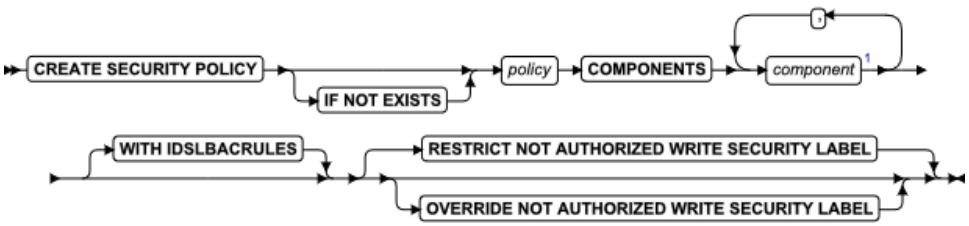
TREE 类型组件可以适用于多维安全策略的一些维度映射到单个逻辑层次结构或者共享公共根的层次结构的上下文中。

## 2.42 CREATE SECURITY POLICY 语句

使用 CREATE SECURITY POLICY 语句在当前数据库中定义新的安全策略，标识它的安全标签组件和存取规则。只有 GBase 8s 支持此语句。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *component* | 安全标签组件 | 必须已经存在于数据库中，且在此 *policy* 的组件名称中必须唯一 | 标识符 |
| *policy* | 此处声明的安全策略名 | 在数据库中安全策略的名称中必须唯一 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 称 | | |

**用法**

***安全策略***是存储以下信息的已命名的数据库对象：

- 它定义包括安全标签的安全标签组件集。
- 它将存取规则集合与安全标签关联。

对于由安全策略保护的表，该存取规则使 GBase 8s 比较具有一列或行的安全标签的用户的安全凭证。安全策略应用于确定拥有给定安全标签的用户是否可以向被安全标签标号的行或列读取或写入数据。安全策略对不具有安全标签的数据没有作用。

同一时刻一个表只能连接一个安全策略，并且一个安全策略只能包含不超过 16 个安全标签组件。

如果您包含可选的 IF NOT EXISTS 关键字，则当指定名称的安全策略已经在当前数据库中注册时，数据库服务器不采取操作（而非向应用程序发发送异常）。

只有 DBSECADM 能发出此语句。当 CREATE SECURITY POLICY 已经执行成功， GBase 8s 会对当前数据库的系统目录做出以下更改：

- 在 **syssecpolicies** 表中注册指定的 ***policy*** 名称和其安全标签组件的敏感性；
- 为每个 ***component*** 在 **syssecpolicycomponentrules** 表中的创建新行。

## 安全策略的安全标签组件

CREATE SECURITY POLICY 语句必须指定至少一个（但不多于 16 个）安全标签组件。这些组件必须已经在当前数据库中存在，DBSECADM 可使用 CREATE SECURITY LABEL COMPONENT 语句注册它们。相同的***组件***名称在同一 CREATE SECURITY POLICY 语句中不能指定两次。

请参阅 CREATE SECURITY LABEL COMPONENT 章节以获取有关安全标签组件的结构和语义的更多信息。

## 与安全策略相关的规则

WITH IDSLBACRULES 关键字指定新安全策略执行的读访问权规则和写访问权规则。如果您没有指定它们，则这些关键字缺省生效，因为 **IDSLBACRULES** 存取规则安全策略唯一支持的存取规则。

以下是 IDSLBACRULES 读访问权规则，称为 IDSLBACREAD ，当在 SELECT 、 UPDATE 或 DELETE 操作中从被标签的列或 列中读取数据值时应用此规则：

- **IDSLBACREADARRAY**：用户安全标签的每个 array 组件必须大于或等于数据行安全标签的 array 组件。也就是说，只有等于或低于用户级别的数据能被读取。
- **IDSLBACREADTREE**：用户安全标签的每个 tree 组件在数据行安全标签（或一个类似此元素的祖先）的 tree 组件中必须包含至少一个元素。
- **IDSLBACREADSET**:：用户安全标签的每个 SET 组件必须包含数据行安全标签的 SET 组件。

以下是 IDSLBACRULES 写访问权规则，称为 IDSLBACWRITE ，当在 INSERT 、UPDATE 或 DELETE 操作中向被标签的列或行中写入数据值时应用此规则：

- **IDSLBACWRITEARRAY**：用户安全标签的每个 array 组件必须等于数据行安全标签的 array 组件。即，只能写入与用户相同级别的数据。
- **IDSLBACWRITETREE**：用户安全标签的每个 tree 组件在数据行安全标签（或一个类似此元素的祖先）的 tree 组件中必须包含至少一个元素。
- **IDSLBACWRITESET**：用户安全标签的每个 SET 组件必须包含数据行安全标签的 SET 组件。

如果 DBSECADM 省略 WITH IDSLBACRULES 关键字，则这些规则缺省生效。然而，如果 WITH 关键字之后跟随除 IDSLBACRULES 值之外的任何规范，则 CREATE SECURITY POLICY 语句发生错误而失败，且不会创建安全策略。

除了显式或缺省 WITH IDSLBACRULES 关键字，CREATE SECURITY POLICY 语句还必须指定当用户未授权写入由安全策略保护的表 DELETE 、INSERT 或 UPDATE 语句中提供的显式指定的安全标签时，执行此写访问规则。用户的安全标签和用户持有的豁免凭证可确定用户是否对显式提供的安全标签具有写访问权。

- 如果 CREATE SECURITY POLICY 语句指定 OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL ，则 GBase 8s 使用此用户安全标签的值，而非 DELETE 、INSERT 或 UPDATE 语句中显式指定的安全标签，来确定用户是否具有对 DELETE 、 INSERT 或 UPDATE 操作中安全标签保护的数据值的写访问权。
- 缺省为 RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL 。如果您显式地指定这些关键字，或者如果它们缺省生效，当用户没有被授权在有此显式指定的安全标签中行或列中写入数据时，DELETE 、INSERT 或 UPDATE 语句失败。

以下示例创建了名为 MegaCorp 的安全策略，它使用三个安全标签，没有为用户安全标签设置 OVERRIDE 。以便在 DELETE 、INSERT 或 UPDATE 操作中对明确指定的标签不授权该用户的写访问权的数据提供写访问。

```
CREATE SECURITY POLICY MegaCorp
        COMPONENTS levels, compartments, groups
        WITH IDSLBACRULES;
```

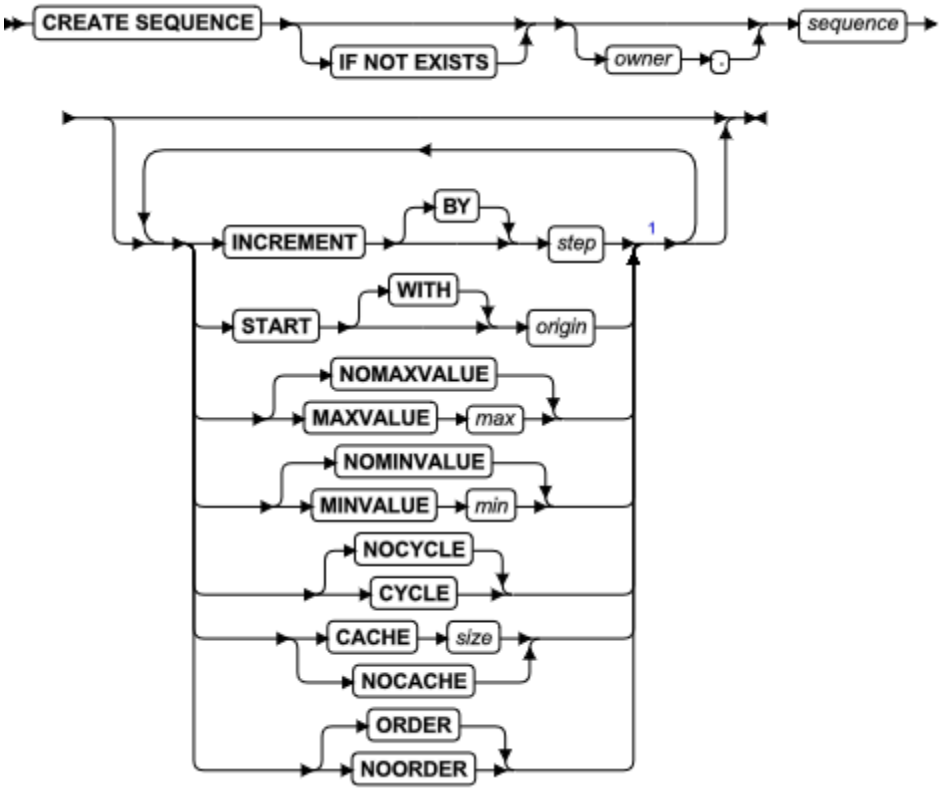
要使此示例有效，levels 、compartments 和 groups 安全标签组件（或者已被重命名为这些标识符的组件）必须在先前的 CREATE SECURITY LABEL COMPONENT 语句定义。

## 2.43 CREATE SEQUENCE 语句

使用 CREATE SEQUENCE 语句创建从多个用户生成唯一整数的序列数据库对象。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *max* | 值的上限 | 必须为整数 > *origin* | 精确数值 |
| *min* | 值的下限 | 必须是小于 *origin* 的整数 | 精确数值 |
| *origin* | 序列中的第一个数字 | 必须是在 INT8 或 BIGINT 范围内的整数 | 精确数值 |
| *owner* | *sequence* 的所有者 | 必须是权限标识符 | 所有者名称 |
| *sequence* | 在这里为新的序列声明的名称 | 必须在序列、顺序、视图和同义词名称中是唯一的 | 标识符 |
| *size* | 内存中预分配的值的数目 | 整数 > 1，但 < 周期 (=\|(*max* - *min*)/*step*\|) 的基数 | 精确数值 |
| *step* | 连续值间的时间间隔 | INT 范围内的非零整数 | 精确数值 |

## 用法

序列（有时称为***系列生成器*** 或 ***序列对象***）返回一系列单调升序或单调降序的唯一整数，一次返回一个。 CREATE SEQUENCE 语句定义新的序列对象，声明其标识符并在 **syssequences** 系统目录表中注册此对象。

序列已授权的用户可以通过在 DML 语句中包含 ***sequence*.NEXTVAL** 表达式来请求新的值。***sequence*.CURRVAL** 表达式返回指定 ***sequence*** 的当前值。**NEXTVAL** 和 **CURRVAL** 表达式只在 SELECT 、DELETE 、INSERT 和 UPDATE 语句中有效；如果尝试在其它上下文中调用内置的 **NEXTVAL** 或 **CURRVAL** 函数， GBase 8s 将返回一个错误。

生成的值在逻辑上类似 BIGSERIAL 或 SERIAL8 数据类型，但在该序列内是唯一的。因为数据库服务器生成这些值，所以多个序列能比一个序列列支持更高级别的并发性。这些值独立于事务；即使生成值的事务失败，生成的值也不能回滚。

可以使用序列自动生成主键值（为许多表使用一个序列），或者每个表都可以有子句的序列。

CREATE SEQUENCE 可以指定序列的以下特征：

- 初始值
- 值间增量的大小和符号
- 最大和最小值
- 序列在达到其限制后是否回收值
- 在内存中预先分配了多少值用于快速存取

数据库可以同时支持多个序列，但是在表、临时表、视图、同义词和序列的名称中当前数据库内序列的名称（或在兼容 ANSI 的数据库中，***owner.sequence*** 组合）必须是唯一的。

如果包含对立选项（如同时指定 MINVALUE 和 NOMINVALUE 选项或同时指定 CACHE 和 NOCACHE），则会发生错误。

如果您包含可选 IF NOT EXISTS 关键字，则当指定名称的序列对象已经在当前数据库中注册时，或者指定的名称是当前数据库中的表、视图或同义词的标识符时，数据库服务器不采取操作（而非向应用程序发送异常）。

**示例**

以下示例创建了序列，将序列中的值插入到表，并从该表查询了这些行和列。

```
CREATE SEQUENCE seq_2
        INCREMENT BY 1 START WITH 1
        MAXVALUE 30 MINVALUE 0
        NOCYCLE CACHE 10 ORDER;

        CREATE TABLE tab1 (col1 int, col2 int);
        INSERT INTO tab1 VALUES (0, 0);

        INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)

        SELECT * FROM tab1;

        col1          col2

        0             0
```

1 1

## INCREMENT BY 选项

使用 INCREMENT BY 选项指定序列中连续数字间的间隔。BY 关键字是可选的。间隔或 ***step*** 值可以是 INT8 范围内的正整数（对于*升序*序列 ）或者负整数（对于*降序*序列）。如果您不指定任何 ***step*** 值，则连续生成值之间的缺省间隔为 1 ，且序列为升序序列。

## START WITH 选项

使用 START WITH 选项指定序列的第一个数字，如果 CREATE SEQUENCE 语句中指定了 ***min*** 或 ***max*** ，则此 ***origin*** 值必须是 INT8 范围内大于或等于 ***min*** 值（对于升序序列）或者小于等于 ***max*** 值（对于降序序列）的整数。WITH 关键字是可选的。

如果您未指定 origin 值，则缺省初始值为 min （对于升序序列）或者 max （对于降序序列）。（以下 MAXVALUE 或 NOMAXVALUE 选项 和 MINVALUE 或 NOMINVALUE 选项 两节分别描述 max 和 min 规范。）

## MAXVALUE 或 NOMAXVALUE 选项

使用 MAXVALUE 选项指定序列中值的上限。最大值或 ***max*** ，必须是 INT8 范围内大于 ***origin*** 的值的整数。

如果未指定 ***max*** 值，则缺省值为 NOMAXVALUE 。此缺省设置支持小于或等于 2e64 的值（对于升序序列）或者小于或等于 -1 （对于降序序列）。

## MINVALUE 或 NOMINVALUE 选项

使用 MINVALUE 选项定序列中值的下限。最小值或 ***min*** 必须是 INT8 范围内小 ***origin*** 的值的整数。

如果未指定 ***min*** 值，则缺省值为 NOMINVALUE 。此缺省设置支持大于或等于 1 （对于升序序列）或者大于或等于 -(2e64) （对于降序序列）。

## CYCLE 或 NOCYCLE 选项

使用 CYCLE 选项在序列达到最大值（升序）或最小值（降序）限制后继续生成序列。在升序序列达到 ***max*** 值，它为下一个序列值生成 ***min*** 值。在降序序列达到 ***min*** 值后，它为下一个序列生成 ***max*** 值。

缺省值为 NOCYCLE 。在此缺省设置，序列无法在达到声明的限制后生成更多的值。一旦序列达到该限制，***sequence*.NEXTVAL** 的下一引用返回一个错误。

## CACHE 或 NOCACHE 选项

使用 CACHE 选项指定预先分配在内存中用于快速存取的序列值数。此功能可增强大量使用的序列的性能。

高速缓存 ***size*** 必须是 INT 范围内的正整数。如果指定 CYCLE 选项，则 ***size*** 必须小于周期（或小 |(***max*** - ***min***)/***step***| ）中的值数。最小值为 2 个预先分配的值，缺省为 20 个预先分配的值。

NOCACHE 关键字指定没有为此序列对象在内存中预先分配生成的值（即，零）。

配置参数 SEQ_CACHE_SIZE 指定可以在序列高速缓存中有预先分配的值的序列对象的最大数。如果没有设置此配置参数，则缺省情况下可用 CACHE 选项定义 10 个以下的不同序列对象。

### ORDER 或 NOORDER 选项

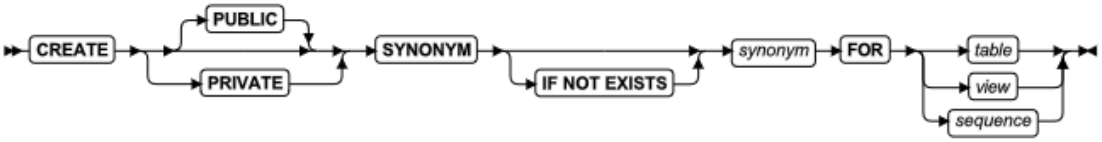
这些关键字对序列的行为没有影响。序列始终以用户请求的顺序向用户发出值，似乎 ORDER 关键字是始终指定的。ORDER 和 NOORDER 关键字由 CREATE SEQUENCE 语句接受以与其它 SQL 方言的序列对象的实现相兼容。

## 2.44 CREATE SYNONYM 语句

使用 CREATE SYNONYM 语句为现有表、视图或序列对象声明并注册备用名。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *sequence* | 本地序列的名称 | 必须在当前数据库中存在 | 标识符 |
| *table, view* | 正为其创建 *synonym* 的表或视图的名称 | 必须注册于当前数据库中，或者注册于限定符内指定的数据库中 | 数据库对象名 |
| *synonym* | 在此处为 *table* 、*view* 或 *sequence* 的名称声明的同义词 | 在表对象名称中必须唯一的；另请参阅 Usage 说明 | 数据库对象名 |

**用法**

用户对同义词以及该同义词引用的数据库对象具有相同的特权。**syssynonyms** 、**syssyntable** 和 **systables** 系统目录表保存关于同义词的信息。

不能在同一数据库中为同义词创建同义词。

同义词的标识符必须在相同数据库中的表、临时表、外部表、视图和序列对象的名称中是唯一。（反之，请参阅带有相同名称的同义词 章节。）

如果您包含了可选 IF NOT EXISTS 关键字，则当指定名称的同义词已经在当前数据库中注册时，或指定的名称是当前数据库中表、视图、或序列对象的名称时，数据库不采取操作（而非向应用程序发送异常）。

一旦创建了同义词，则它会一直持续到所有者执行 DROP SYNONYM 语句。（此持久性将同义词与您能在 SELECT 语句的 FROM 子句中声明的别名区别开来；别名仅在 SELECT 语句的执行期间位于作用域内。）

如果同义词引用相同数据库中的表、视图或序列，则该同义词在所引用的表、视图或序列被删除时自动删除。有关其它信息，请参阅 外部数据库对象的同义词。

## 外部数据库对象的同义词

可以为您的会话当前连接的数据库服务器上任意数据库中的任意表或视图创建同义词。

此示例为当前数据库外的表在当前数据库服务器的 payables 数据库中声明同义词。

```
CREATE SYNONYM mysum FOR payables:jean.summary;
```

您也可以为 CREATE EXTERNAL TABLE 注册在当前数据库中的外部表创建同义词。（外部表要注册在创建它的数据库的系统目录中，而非存储它的任意数据库。）

您也可以为存在于某个数据库服务器的数据库中的表或视图创建同义词，该数据库服务器不是您的当前数据库服务器。当创建同义词时，这两个数据库服务器必须都处于联机状态。在网络中，远程数据库服务器验证该同义词引用的表或视图在创建同义词时是否存在。下一示例为由远程数据库服务器支持的表创建同义词：

```
CREATE SYNONYM mysum FOR payables@phoenix:jean.summary;
```

现在，标识符 **mysum** 引用表 **jean.summary** ，该表位于 **phoenix** 数据库服务器上的 **payables** 数据库中。如果从 **payables** 数据库删除了 **summary** 表，则 **mysum** 同义词保持不动。随后尝试使用 **mysum** 会返回错误：`Table not found` 。

然而， GBase 8s 并不支持这些外部对象的同义词：

- 类型表（包括作为表层次结构一部分的任何表）
- 包含任意扩展数据库类型的表或视图
- 本地数据库外的序列对象

## PUBLIC 和 PRIVATE 同义词

如果使用 PUBLIC 关键字（或不使用任何关键字），则所有可以访问数据库的人都可以使用您的同义词。如果数据库不兼容 ANSI ，则用户不必知道公共同义词的所有者名称。位于不兼容 ANSI GBase 8s 数据库服务器内创建的数据库中的任意同义词都是公共同义词。

在兼容 ANSI 的数据库中，所有的同义词都是专用的。如果使用 PUBLIC 或 PRIVATE 关键字，则数据库服务器发出语法错误。

如果使用 PRIVATE 关键字在不兼容 ANSI 的数据库中声明同义词，则非限定的同义词可由其所有者使用。其他用户必须用所有者的名称限定同义词。

## 带有相同名称的同义词

在兼容 ANSI 的数据库中，*owner.synonym* 组合在所有同义词、表、视图以及序列中必须是唯一的。在引用不是您自己的同义词时必须指定 *owner* ，如下所示：

```
CREATE SYNONYM emp FOR accting.employee
```

在不兼容 ANSI 的数据库中，两个公共同义词不能有相同的标识符，而同义词的标识符也必须在相同数据库中的表、视图和序列的名称中是唯一的。

专用同义词的 *owner.synonym* 组合必须在数据库内所有同义词中是唯一的。即，同一数据库中可以存在多个带有相同名称的同义词，但是这些同义词中的每一个都必须归不同用户所有。同一用户不能同时创建名称相同的专用和公共同义词。例如，以下代码生成一个错误：

```
CREATE SYNONYM our_custs FOR customer;
CREATE PRIVATE SYNONYM our_custs FOR cust_calls;-- ERROR!!!
```

只有在两个同义词所有者不同的情况下，才能用与公共同义词相同的名称声明专用同义词。如果拥有一个专用同义词且存在带有相同名称的公共同义词，则数据库服务器未限定的名称解析诶专用同义词。（在这种情况下，您必须指定 ***owner.synonym*** 以引用公共同义词）如果在专用同义词和另一用户的公共同义词都有相同的标识符时，将 DROP SYNONYM 与未限定的同义词标识符一起使用，则仅删除专用同义词。如果重复相同的 DROP SYNONYM 语句，则数据库服务器删除公共同义词。

## 链接同义词

如果为不是当前数据库中的表或视图创建同义词，且已删除了此表或视图，则同义词保留在注册的目录中。可以用已删除的表或视图的名称作为同义词为删除的表或视图创建新的同义词，但是该同义词值指向当前数据库（而非另一个数据库）中的的表或视图。

以此方式，可将表或视图移至新的位置并链接仍然有效的原始的同义词。以此方法最多可以链接 16 个同义词。

链接同义词以引用已重新定位的表对象对表或视图是可能的，但是对指向序列对象的同义词无效，因为 CREATE SYNONYM 只能为已注册在当前数据库中的序列定义同义词。

以下步骤为 customer 表将两个同义词链接在一起，该表将最终驻留在 zoo 数据库服务器上。这里的省略号（ . . . ）表示 CREATE TABLE 语句不完整：

1. 在称为 **training** 的数据库服务器上的 **stores_demo** 数据库中，发出以下语句：

   ```
   CREATE TABLE customer (lname CHAR(15)...);
   ```

2. 在称为 **accntg** 的数据库服务器上，发出以下语句：

   ```
   CREATE SYNONYM cust FOR stores_demo@training:customer;
   ```

3. 在称为 **zoo** 的数据库服务器上，发出以下语句：

   ```
   CREATE TABLE customer (lname CHAR(15)...);
   ```

4. 在称为 **training** 的数据库服务器上，发出以下语句：

```
DROP TABLE customer;
CREATE SYNONYM customer FOR stores_demo@zoo:customer;
```

**accntg** 数据库服务器上的同义词 **cust** 现在指向 **zoo** 数据库服务器上的 **customer** 表。

以下示例显示将两个同义词连接在一起并链接同义词指向的表的示例：

1. 在称为 **training** 的数据库服务器上，发出以下语句：

   ```
   CREATE TABLE customer (lname CHAR(15)...);
   ```

2. 在称为 **accntg** 的数据库服务器上，发出以下语句：

   ```
   CREATE SYNONYM cust FOR stores_demo@training:customer;
   ```

3. 在称为 **training** 的数据库服务器上，发出以下语句：

   ```
   DROP TABLE customer;
   CREATE TABLE customer (lastname CHAR(20)...);
   ```

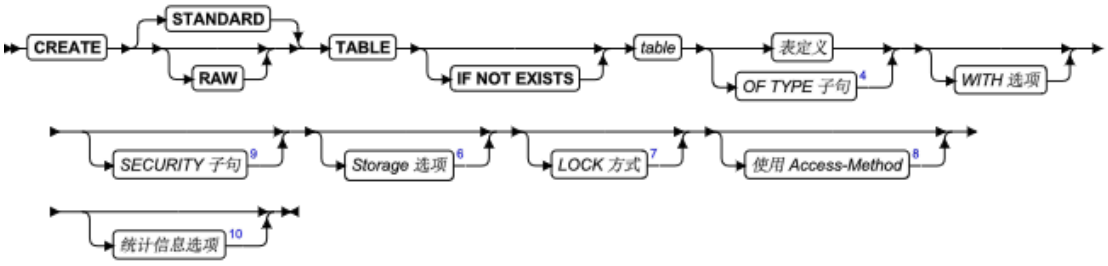
**accntg** 数据库服务器上的同义词 **cust** 现在指向 **training** 数据库服务器上的 **customer** 表。

## 2.45 CREATE TABLE 语句

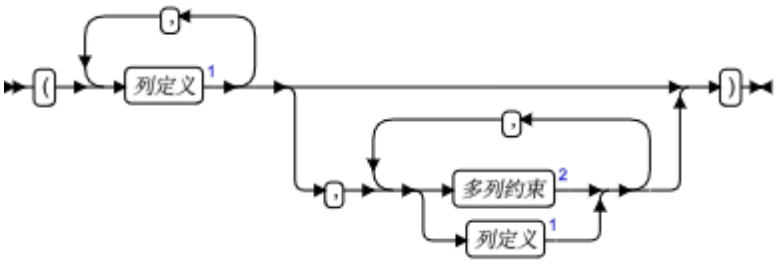
使用 CREATE TABLE 语句在当前数据库中创建新的永久表。

可以使用 CREATE TABLE 语句来创建关系数据库表或类型表（对象关系型表）。有关创建临时表的信息，请参阅 CREATE TEMP TABLE 语句 。有关如何创建不存储在数据库中的外部表对象的信息，请参阅 CREATE EXTERNAL TABLE 语句 。有关如何创建全局临时表的信息，请参阅 CREATE GLOBAL TEMPORARY TABLE 语句。

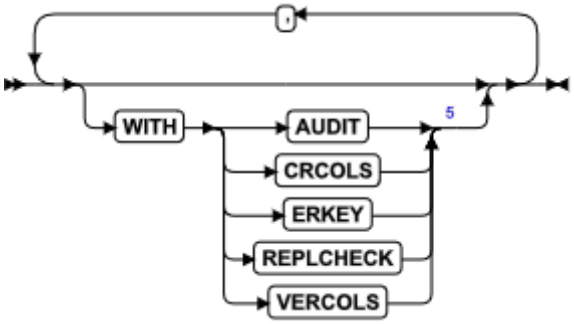
**语法**




表定义




WITH 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *table* | 在这里为新的表声明的名称 | 在数据库中的物化视图、视图、表、序列和同义词名称中必须唯一。 | 标识符 |

**用法**

当您创建表时，必须声明它的名称，定义它的结构和它的日志记录状态。可以如随后的章节所标识的那样可选地指定其他属性。该语法图显示了需要的或可选的规范的序列。CREATE TABLE 语句的语法段，和一些它们的组件都标识在一下的五个列表中。

下列关键字和子句定义新的列的属性：

表 1. 定义列的名称、数据类型、缺省值和安全标签

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| 列定义 | 列定义 | 列的名称和属性，包括数据类型、约束、缺省值 |
| DEFAULT | DEFAULT 子句 | 列的缺省值 |
| COLUMN SECURED WITH | 列定义 | 受保护的表的 LBAC 标签 |

以下关键字和子句在新表上定义约束：

表 2. 在表的一列或多列上定义约束

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| 单列约束 | 单列约束格式 | 单列上的数据库完整性约束、参照完整性约束或其它约束 |
| 约束定义 | 约束定义 | 表上约束的名称、属性和启用或禁用的状态 |
| NULL | 使用 NULL 约束 | 列允许 NULL 值 |
| NOT NULL | 使用 NOT NULL 约束 | 列不允许 NULL 值 |
| UNIQUE *or* | 使用 UNIQUE 或 | 列不允许重复的值 |

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| DISTINCT | DISTINCT 约束 | |
| CHECK | CHECK 子句 | 检查其它列的约束 |
| PRIMARY KEY | 使用 PRIMARY KEY 约束 | 表中的每一行包含一个非 NULL 的唯一值 |
| FOREIGN KEY | 使用 FOREIGN KEY 约束 | 建立表之间的依赖 |
| REFERENCES | REFERENCES 子句 | 与其它列的参照完整性约束 |
| 多列约束 | 多列约束格式 | 列集合上的数据完整性约束 |

下列关键字和子句定义了该表的影子列和行级别审计支持：

表 3. 定义影子列和行级别审计支持

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| WITH *keyword* | Options 子句 | 影子列或行级别审计支持的关键字选项 |
| WITH AUDIT | 使用 WITH AUDIT 子句 | 行级别审计支持 |
| WITH CRCOLS | 使用 WITH CRCOLS 选项 | 影子列或行级别审计支持的关键字选项 |
| WITH ERKEY | 使用 WITH ERKEY 关键字 | Enterprise Replication 定义主键的 3 个影子列 |
| WITH REPLCHECK | 使用 WITH REPLCHECK 关键字 | 在完整性检查中使用的影子列 |
| WITH ROWIDS | 使用 WITH ROWIDS 选项 | 已分片表中的隐藏列（不推荐使用） |
| WITH VERCOLS | 使用 WITH VERCOLS 选项 | 辅助服务器上用于 UPDATE 操作的 2 个影子列 |

下列关键字和子句定义新表的存储选项：

表 4. 定义新表或其智能大对象列的存储

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| Storage Options | 存储选项 | 表物理存储位置和有关表如何存储的其它信息 |
| IN dbspace, sbspace, | 使用 IN 子句 | 拥有新表（或表的一部分、或大对象）的存储对象 |

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| blobspace, *or* extspace | | |
| FRAGMENT BY *or* PARTITION BY | FRAGMENT BY 子句 | 分片表的分布存储方案 |
| BY ROUND ROBIN | 通过 ROUND ROBIN 分片 | 存储表分片的 dbspace 列表 |
| BY EXPRESSION | 表达式分片子句 | 基于表达式的分片分布 |
| BY LIST | 列表分片子句 | 基于列表的分片分布 |
| BY RANGE . . . INTERVAL | Interval fragment 子句 | 基于 RANGE INTERVAL 的分片分布 |
| PUT 子句 | PUT 子句 | BLOB 或 CLOB 列的存储位置、extent 大小以及其它 sbspace 属性 |
| EXTENT SIZE | EXTENT SIZE 选项 | 表的第一个和后续的存储 extent 的大小 |
| COMPRESSED | 表的 COMPRESSED 选项 | 大量的行数据是否启用自动压缩 |

下列关键字和子句定义日志记录方式和其它表属性，或者向新表中插入指定查询返回的符合条件的行。

表 5. 日志记录选项、锁定粒度、访问方法、类型表属性、数据分布统计选项、插入来自查询结果的数据或表的 LBAC 安全策略。

| 规范 | 主题 | 该关键字或子句定义的内容 |
|---|---|---|
| 日志记录选项 (STANDARD 或 RAW) | 日志记录选项 | 新表的日志记录特征 |
| LOCK MODE (PAGE *或* ROW) | LOCK MODE 选项 | 新表的锁定粒度 |
| USING Access-Method | USING 存取方法子句 | 如何访问新表 |
| OF TYPE | OF TYPE 子句 | 在关系对象型数据库中新类型表的已命名 ROW 类型 |
| UNDER | 使用 UNDER 子句 | 在类型表层次结构中新子表的超表 |
| SECURITY POLICY | SECURITY POLICY 子句 | 表的基于标签的访问控制策略 |

| 规范 | 主题 | 该关键字或子句定义的内容 |
| --- | --- | --- |
| STATCHANGE, STATLEVEL | CREATE TABLE 语句的 Statistics 选项 | 更改数据分布统计信息的粒度和域 |
| AS SELECT | AS SELECT 子句 | 创建并填充查询结果表 |

**表名称和列名称的唯一性规则**

当您创建新表是，每个列必须有与之相关的数据类型。列的名称在同一表的列的名称中必须是唯一的。（ OF TYPE 选项指定现有已命名 ROW 类型，其字段为您正在创建的类型表提供列名称和列的数据类型。）

如果数据库没有被创建为 MODE ANSI ，则***表***名称必须在相同的数据库中的表、视图、序列和同义词的名称中是唯一的。

在兼容 ANSI 的数据库中，组合 ***owner.table*** 必须在在相同的数据库中的表、同义词、视图、和序列对象的名称中是唯一的。具有不同***所有者***名称的表对象可以具有相同的标识符。

如果您包含了可选的 IF NOT EXISTS 关键字，则当指定名称的表已经存在于当前数据库中时，数据库不采取操作（而非向应用程序发送异常）。

**CREATE TABLE 的其它语法说明**

对于在新的永久性表中存储一个查询的结果集的 CREATE TABLE 语句的受限制的语法选项，请参阅 AS SELECT 子句 。

在 DB-Access 中，如果使用 **-ansi** 标识或设置 **DBANSIWARN** 环境变量，那么当您在 CREATE SCHEMA 语句外使用 CREATE TABLE 时将会生成警告。

## 日志记录选项

使用 Logging Type 选项指定对表进行各种批量操作时可以提高性能的日志记录特征。

除了用于 OLTP 数据库的缺省选项（STANDARD）以外，这些日志记录选项主要用于提高数据仓库数据库的性能。

永久表可以拥有以下任一日志记录特征。

| 日志记录类型 | 作用 |
| --- | --- |
| STANDARD | 允许回滚、恢复和从归档恢复的记录表。该类型是缺省值。对所有 OLTP 数据库需要的恢复和约束功能性使用该类型的表。 |
| RAW | 不支持主键约束或唯一约束的非日志记录表。但是它支持引用约束，且可以被索引和更改。使用此类型表来快速加载数据。 |

**警告：** 使用 raw 表进行数据的快速加载，但是在事务中使用表或在表中修改数据之前将日志记录类型设置为 STANDARD 并执行 0 级备份。如果必须在事务中使用原始表，则将隔离级别设置为 Repeatable Read 或将表锁定为互斥方式来阻止并行性问题。

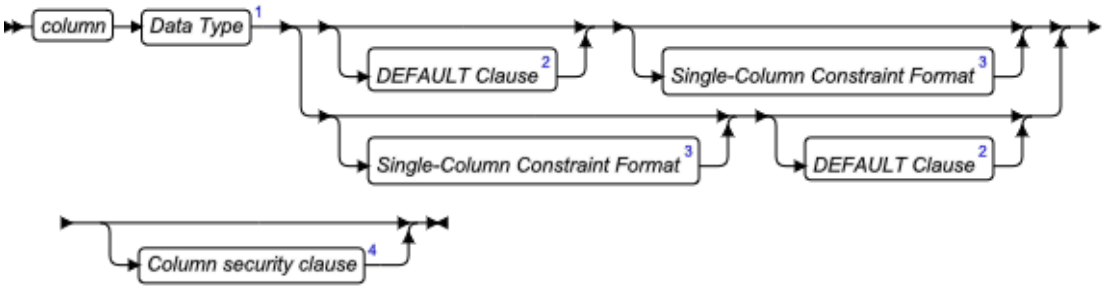
高可用集群中的辅助服务器上不支持 CREATE RAW TABLE 语句。

有关这些表的日志记录类型的更多信息，请参阅 *GBase 8s 管理员指南*。

## 列定义

使用 CREATE TABLE 语句的列定义部分列出新表的单列的名称和数据类型（可选的缺省值以及约束或安全标签）。

列定义




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 表中列的名称 | 在此表中必须是唯一的 | 标识符 |

行的最大大小是 40M 字节。表中最多只能有大约 97 列可以是 COLLECTION 数据类型（SET 、LIST 和 MULTISET）。表中最多大约有 195 列可以是数据类型 BYTE 、TEXT 、ROW 、LVARCHAR 和可变长度的 UDT。最多大约有 151 列可以是 VARCHAR 和 NVARCHAR 数据类型。 （此处的 195 列和 151 列是使用 2 KB 基本页大小的平台的最低的近似值。对于 4 KB 基本页大小的平台，如 Windows™ 和 AIX® 系统，这些数据类型的上限值大约为 450 列。）

这些数据类型的列的数目的上限还取决于描述数据库服务器存储在同一分区的表的其它数据。对于某些表，列数的最大值可能很小，如果压缩和存储在磁盘上的所有的 SQL 标识符（包括数据库名称、表名称和索引名称）的聚合长度减少了用于列的可用空间，则最大列数可能变小。

### 字符列大小语义

除非将 SQL_LOGICAL_CHAR 配置参数设置为在数据类型定义中的启用逻辑字符语义，否则内置字符类型列（如 CHAR 、LVARCHAR 、NCHAR 、NVARCHAR 或 VARCHAR）的任何显式或缺省存储大小规范以字节为单位进行解释。

将大小声明解释为逻辑字符语义可降低 INSERT 和 UPDATE 操作中列值存储不足的风险。当数据长度超出列的最大大小时，该结果取决于数据库的 ANSI 兼容的状态：

- 如果数据库不兼容 ANSI ，则 GBase 8s 删除该值。当此删除发生时不会生成警告。
- 如果数据库兼容 ANSI ，则 INSERT 或 UPDATE 操作失败并返回它们的错误。

-1279: Value exceeds string column length.

请参阅 *GBase 8s 管理员参考手册* 中有关 SQL_LOGICAL_CHAR 配置参数的描述，以获取有关设置在多字节代码集（如，**UTF-8**）的语言环境中的效果的更多信息，其中单个逻辑字符可能需要多个字节的存储空间。

**IDSSECURITYLABEL 列上的限制**

以下限制影响列定义子句指定 DSSECURITYLABEL 数据类型的列以支持基于标签访问控制（LBAC）的使用：

- 如果表没有安全策略，则持有 DBSECADM 角色的用户还必须包含 SECURITY POLICY 子句以指定安全策略。
- 只有持有 DBSECADM 角色的用户可指定 IDSSECURITYLABEL 类型列。
- 一个表只能具有一个 IDSSECURITYLABEL 类型的列。
- IDSSECURITYLABEL 列不能具有列包含。
- IDSSECURITYLABEL 列具有隐式的 NOT NULL 约束。如果在 DEFAULT 子句中没有为缺省的安全标签指定 *label* 名称，则该列的缺省值是由用户持有的写访问权的安全标签。
- IDSSECURITY LABEL 列不能有任何显式单列约束，并且它不能是多列引用或检查约束的一部分。
- IDSSECURITYLABEL 列不能被加密。

与任何 SQL 标识符一样，如果列名称是关键字，或者与表的名称相同，或者您以后与其它表一起使用的的另一个表的名称，则可能发生语义模糊（有时还会出现错误消息或意外行为）。有关 GBase 8s 的关键字的信息，请参阅 GBase 8s 的 SQL 关键字。

如果您将表的一列定义为已命名的 ROW 类型，则该表不会采用该已命名的 ROW 的任何约束。

## 列安全子句

使用列安全子句给列添加基于标签的行级别安全保护。

列安全子句


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 安全标签的名称 | 必须存在其必须属于保护此表的安全策略 | 标识符 |

列安全子句可以添加基于标签的行级别保护。该子句只对受安全策略保护的表有效。有关基于标签的安全策略和表关联的语法，请参阅 SECURITY POLICY 子句。

安全标签可以是保护该表其它行或列的相同的标签，或者它可以是同一安全策略的不同标签。以下限制应用于 SECURED WITH 子句：

- 该列不能是 IDSSECURITYLABEL 类型。

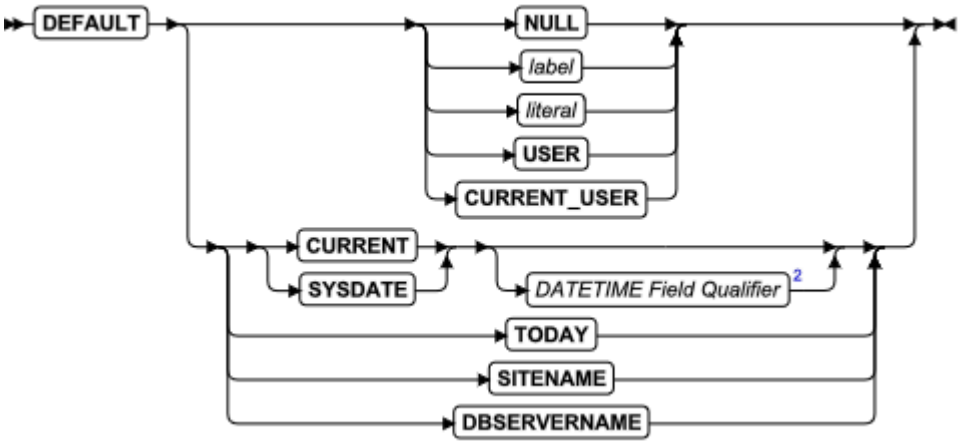
- 指定没有策略限定符的标签，不是 *policy.label* 。
- 该标签必须是保护该表的安全策略的标签。

## DEFAULT 子句

使用 DEFAULT 子句为数据库服务器指定插入列的值（当没有为列指定显式值时）。

不能为 SERIAL 、BIGSERIAL 或 SERIAL8 列指定缺省值。

DEFAULT 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 安全标签的名称 | 必须存在并且属于保护该表的安全策略。该列必须是 IDSSECURITYLABEL 类型。 | 标识符 |
| *literal* | 字母或数字字符的字符串 | 必须是适合该列的数据类型。请参阅将文字值作为缺省值。 | 表达式 |

## 将 NULL 作为缺省值

如果没有为列指定缺省值，除非您在该列上放置了 NOT NULL 约束，否则缺省值为 NULL 。在这种情况中，不存在缺省值。

如果将 NULL 指定为列的缺省值，则不能将 NOT NULL 约束指定为列定义的一部分。（有关 NOT NULL 约束的更多信息，请参阅 使用 NOT NULL 约束。）

当列是主键的一部分时，NULL 不是该列的有效缺省值。

当列是 BYTE 或 TEXT 数据类型时，NULL 是唯一有效的缺省值。

在 GBase 8s 中，如果列是 BLOB 或 CLOB 数据类型，则 NULL 是唯一有效的缺省值。

## 将文字值作为缺省值

可将*文字*值指定为缺省值。文字值是字母或数字字符组成的字符串。要将文字值用作缺省值，您必须遵循以下表中的语法限制。

| 对数据类型的列 | 缺省值的格式 |
|---|---|
| BOOLEAN | 将 't' 或 'f' （分别代表 *true* 或 *false*）用作 引用字符串 。 |
| CHAR, CHARACTER VARYING, DATE, VARCHAR, NCHAR, NVARCHAR, LVARCHAR | 引用字符串。请参阅 DATE 后的说明。 |
| DATETIME | 文字的 DATETIME |
| BIGINT, DECIMAL, FLOAT, INT8, INTEGER, MONEY, SMALLFLOAT, SMALLINT | 精确数值 |
| INTERVAL | 文字的 INTERVAL |
| Opaque data types | 引用字符串 ，以 单列约束格式 标识 |

DATE 文字必须是 **DBDATE** （或 **GL_DATE**）环境变量指定的格式。在缺省的语言环境中，如果没有设置 **DBDATE** 也没有设置 **GL_DATE** ，则日期文字必须是 ***mm/dd/yyyy*** 的格式。

## 使用常量表达式作为缺省值

可以将常量表达式作为缺省列值。

下表列出了您可指定的常量表达式，以及相应的列的数据类型要求和建议的大小（以字节为单位）。

| 常量表达式 | 数据类型要求 | 建议大小 |
|---|---|---|
| CURRENT, SYSDATE | DATETIME 列及匹配的限定符 | 足够的字节存储语言环境中最长的 DATETIME 值 |
| DBSERVERNAME, SITENAME | CHAR 、VARCHAR 、NCHAR 、NVARCHAR 或 CHARACTER VARYING 列 | 128 字节 |
| TODAY | DATE 列 | 足够的字节存储语言环境中最长的 DATE 值 |
| USER, CURRENT_USER | CHAR 、VARCHAR 、NCHAR 、NVARCHAR 或 CHARACTER VARYING 列 | 32 字节 |

这些是我们建议的列大小，因为如果在 INSERT 或 ALTER TABLE 操作期间由于列长度太小无法存储缺省值时，数据库服务器将返回一个错误。

您不能为保存 OPAQUE 或 DISTINCTY 数据类型的列指定行为类似可变函数的常量表达式（即 CURRENT 、 SYSDATE 、USER 、TODAY 、SITENAME 或 DBSERVERNAME）作为缺省值。另外，如果数据值是加密或使用 **UTF-8** 语言环境中 Unicode 字符集编码的话，则需要较大的列大小。（关于对加密数据所需的存储大小的更多信息，请参阅本章随后关于 SET ENCRYPTION 语句的描述。）

有关这些函数的描述，请参阅 常量表达式。

以下示例创建了一个名为 accounts 的数据库。在 accounts 中，acc_num 、acc_type 和 acc_descr 列中有文字缺省值。acc_id 列的缺省值是用户的登录名。

```
CREATE TABLE accounts (
        acc_num INTEGER DEFAULT 1,
        acc_type CHAR(1) DEFAULT 'A',
        acc_descr CHAR(20) DEFAULT 'New Account',
        acc_id CHAR(32) DEFAULT CURRENT_USER);
```

## 使用函数作为缺省值

可以将系统函数、自定义函数作为列的缺省值。

该功能具有以下限制：

a) 函数返回值类型与 DEFAULT 对应的字段类型保持一致。

b) 系统函数返回值长度不得超过字段定义长度。

c) 函数的参数不能是列名。

外部注册 UDR 或自定义函数允许返回值长度超过字段定义。

以下示例创建了一个名为 t1 的表，并使用字符串函数设置 VARVHAR 字段 **c1** 的缺省值。此函数包括在 default 表达式的括号中。

```
CREATE TABLE t1 (
        c1 varchar(10) DEFAULT ( concat('a','b') || 'c'|| relplace('def', 'de', 'DD') ),c2 INT);
```

使用以下语句向表 t1 插入数据，default 值的字段不设置值：

```
INSERT INTO t1(c2) VALUES(1);
```

最后，使用 select 语句查询此表。

```
SELECT * FROM t1;
```

上述查询返回结果如下，default 的字段值为函数的返回值：

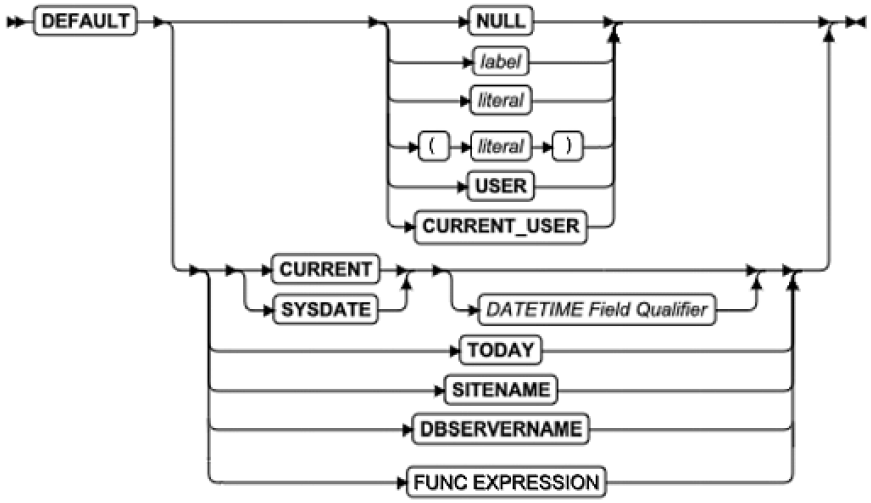
```
c1              c2
abcDDf          1
```

# ORACLE 模式下 DEFAULT 相关功能

以下功能仅在 GBase 8s 的 ORACLE 模式下支持

支持将列类型为整型和浮点型列的 DEFAULT 值设置成 0.0,实际相当于 0.0 默认转化成 0。支持的字段类型包括 int 、bigint 、smallint 、int8 、dec 、dec(m,n) 、decimal 、decimal(m,n) 、double precision 、float(m) 、money(m,n) 、numeric(m,n) 、real 、smallfloat ；

新增 DEFAULT 的使用方式，参考以下语法图：




支持语法格式 default func_expression 。例如，default length(‘a’);

支持语法格式 default (literal) ，与 default literal 功能同义。例如，default(‘123ab’);

当没有为列指定显式值时，数据库将 DEFAULT 值插入该列中。

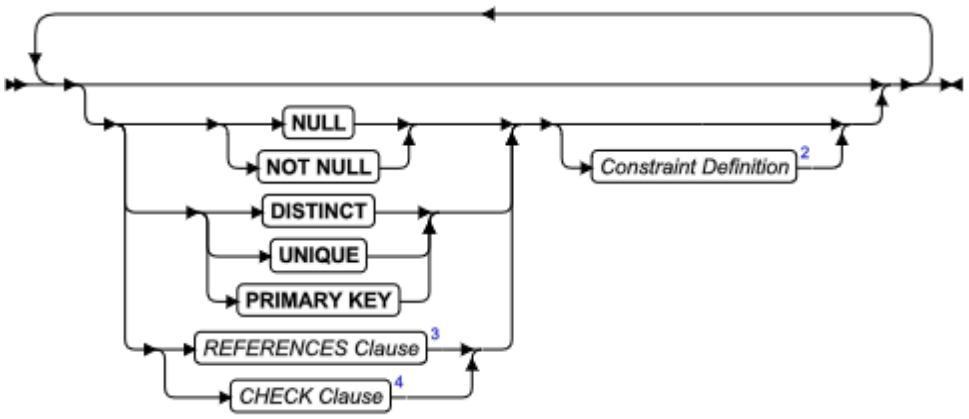
## 单列约束格式

使用单列约束格式为单列定义和声明至少一个约束的名称，并指定每个约束的方式。

使用单列约束格式为某列关联一个多多个约束。从而可以执行以下任务：

- 为列创建一个或多个数据完整性约束。
- 为约束指定一个有意义的名称。
- 指定在插入、删除和更新期间控制约束行为的约束方式。

单列约束格式




NULL 约束指定列可以存储 NULL 值。它不可用于序列列或复杂数据类型的列。如果您在同一列上指定了 NOT NULL 和 NULL 约束，则 CREATE TABLE 语句发生错误并失败。

以下示例创建了有两个约束的标准表：acc_num 列上的主键约束 num ；另一个是 acc_code 列上的唯一约束 code ：

```
CREATE TABLE accounts (
        acc_num     INTEGER PRIMARY KEY CONSTRAINT num,
        acc_code    INTEGER UNIQUE CONSTRAINT code,
        acc_descr CHAR(30));
```

本示例中使用的约束类型将在后面的章节中定义。

## 使用单列约束格式的限制

单列约束格式无法指定包含多列的约束。因此，不能使用单列约束格式来定义组合关键字。有关多列约束的信息，请参阅多列约束格式。

不能在 RAW 表的任一列上定义引用约束或唯一约束。RAW 表只支持 NOT NULL 或 NULL 约束。

不能在 BLOB 、BYTE 、CLOB 或 TEXTY 列上放置唯一、主键、或引用约束。但是可以使用检查约束来检查 BYTE 或 TEXT 列上的 NULL 或 non-NULL 值。

如果约束在存储加密数据的列上，则 GBase 8s 不执行此约束。

## 使用 NOT NULL 约束

使用 NOT NULL 关键字来要求列必须在插入或更新操作期间接收值。如果在列删放置了 NOT NULL 约束（并且没有指定缺省值），则当您插入一行或者某行中更新列是，***必须***在该列中输入一个值。如果没有输入值，则由于不存在缺省值，所以数据库服务器将返回一个错误。

以下示例创建了 newitems 表。在 newitems 中，列 manucode 没有缺省值也不允许有 NULL 值。

```
CREATE TABLE newitems (
        newitem_num INTEGER,
        manucode CHAR(3) NOT NULL,
        promotype INTEGER,
        descrip CHAR(20));
```

当您定义 PRIMARY KEY 约束时，数据库服务器还静默地在同一列或在构成主键的列集上创建了 NOT NULL 约束。

如果还指定了 NOT NULL 约束，则不能指定 NULL 作为列的显式缺省值。

如果在同一列上指定 NOT NULL 约束和 NULL 约束，则 CREATE TABLE 语句发生错误并失败。

集合数据类型 LIST 、MULTISET 和 SET 的列要求 NOT NULL 约束。在结合数据类型上不允许其它的列约束。

## 使用 NULL 约束

使用 NULL 关键字指定列可以存储其数据类型的 NULL 值。这意味着该列在插入或更改操作期间不接受任何值。NULL 约束逻辑等价于从列定义中省略 NOT NULL 约束。

以下示例创建 newitems 表。在 newitems 中，列 descrip 没有缺省值，但是它允许 NULL 值。

```
CREATE TABLE newitems (
        newitem_num INTEGER,
        manucode CHAR(3) NOT NULL,
        promotype INTEGER,
        descrip CHAR(20) NULL);
```

在上述示例中，列 **newitem_num** 和 **promotype** 显式允许 NULL 值，因为它们没有定义 NOT NULL 约束。

如果在同一列上指定 NOT NULL 约束和 NULL 约束，则 CREATE TABLE 语句发生错误并失败。

不能在同一列上同时指定 NULL 约束和 PRIMARY KEY 约束，因为当 CREATE TABLE 语句定义 PRIMARY KEY 约束时，数据库服务器还静默地创建了同一列或在构成主键的列集上创建了 NOT NULL 约束。

NULL 约束对于集合数据类型 LIST 、MULTISET 和 SET 的列无效，对 IDSSECURITYLABEL 列也无效。

## 使用 UNIQUE 或 DISTINCT 约束

使用 UNIQUE 或 DISTINCT 关键字以要求某列或一组列的集合只接收唯一数据值。如果列有唯一约束，则将不能把与其它行重复的值插入到该列中。当您创建了 UNIQUE 或 DISTINCT 约束时，数据库服务器将自动在被约束的列上创建内部索引。（在此上下文中，关键字 DISTINCT 是 UNIQUE 的同义词）。

不能在已经有主键约束的列上放置唯一约束。不能在 BYTE 或 TEXT 列上放置唯一约束。

如先前所述，不能在 GBase 8s 的 BLOB 或 CLOB 列上放置唯一约束或主键约束。

只有当辅助存取方法唯一支持不透明数据类型时，这种数据类型才支持唯一约束。缺省的辅助存取方法是一种 B-tree ，它支持 **equal( )** 操作符函数。因此，如果不透明类型的定义包含 **equal( )** 函数，则该不透明类型的列可以有唯一约束。

以下示例创建了一个简单表，该表在它的某一列上具有唯一约束：

```
CREATE TABLE accounts
        (acc_name   CHAR(12),
        acc_num     SERIAL UNIQUE CONSTRAINT acc_num);
```

有关约束名称的说明，请参阅声明约束名称 。

## 唯一约束和唯一索引的区别

尽管唯一索引和唯一约束的功能相似，除了在声明、更改或销毁它们的语法之间的各种不同，还有在这两种的数据库对象之间的其它不同：

- 在 DDL 语句中，它们注册于或删除系统目录的不同表。
- 在 DML 语句中，在日志记录的表上启用唯一约束会在语句末尾被检查，但是唯一索引是一行一行的检查，从而防止可能潜在地违反指定列（或者对于多列列约束或索引是列列表）的唯一性的行的插入或更改。

例如，当您将值 1 、2 和 3 存储在具有 INT 列的日志记录表的行中，如果列 c 上有一个唯一索引，则对该表上指定的 SET c = c + 1 UPDATE 操作将发生错误而失败，但是如果该列是具有一个唯一约束的话，该语句会成功。

## 使用 PRIMARY KEY 约束

***主键***是表中每行都具有非 NULL 唯一值的列（或如果您使用多列约束格式时列的集合）。当您定义了 PRIMARY KEY 约束时，数据库服务器将自动在组成主键的列上创建内部索引，并静默地在同一列或列集合上创建 NOT NULL 约束。

只能为每个表指定一个主键。如果将单列定义为主键，则该列是唯一的。不能显式地给相同的列指定唯一约束。

不能在 BLOB 或 CLOB 列上放置唯一约束或主键约束。

只有当辅助存取方法支持 GBase 8s 的不透明类型的唯一性时，这种类型才支持主键约束。缺省辅助存取方法是一种 B-tree ，它支持 **equal( )** 函数。因此，如果不透明类型的定义包含 **equal( )** 函数，该不透明的列可以有主键约束。

不能在 BYTE 或 TEXT 列上放置主键约束。

在前面两个示例中，在列 acc_num 上放置了唯一约束。以下示例将该列创建为 accounts 表的主键：

```
CREATE TABLE accounts
        (acc_name   CHAR(12),
        acc_num     SERIAL PRIMARY KEY CONSTRAINT acc_num);
```

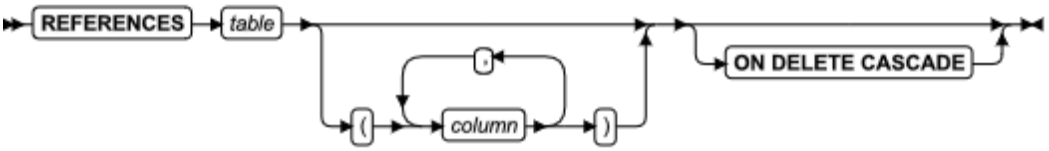
## REFERENCES 子句

使用 REFERENCES 子句建立引用关系：

- 表中（即在同一表的两列中间）

- 两个表之间（换句话说，创建一个外键）

  REFERENCES 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 被引用列 | 请参阅 引用约束的限制 | 标识符 |
| *table* | 被参考表 | 必须与引用表驻留在相同的数据库中 | 标识符 |

*引用*列（定义的列）是对被引用的列或一组列的集合进行引用的列或一组列的集合。引用列中可以包含 NULL 和重复值，但是被引用的列（或列的集合）中的值必须是唯一的。

被引用列和引用列之间的关系被称为*父 — 子*关系，其中父亲是被引用的列（主键），孩子是引用列（外键）。引用约束将建立这人父 — 子关系。

当您创建了引用约束后，数据库服务器将自动在受约束的列上创建内部索引。

## 引用约束的限制

您必须具有 References 特权来创建引用约束。

当您使用 REFERENCES 子句时，您必须注意下列限制：

- 被引用表和引用表必须在同一数据库中。
- 被引用列（当你使用多列约束格式时列的集合）必须具有唯一或主键约束
- 引用列和被引用列的数据类型必须相同。

  唯一的例外是如果被参考列是 serial 数据类型，则参考列必须为整数数据类型：

  - 对于 BIGSERIAL 被引用的列，使用 BIGINT 引用列。
  - 对于 SERIAL 被引用的列，使用 INT 引用列。
  - 对于 SERIAL8 被引用列，使用 INT8 引用列。
- 不能在 RAW B 表的任何列上放置约束。
- 不能在 BYTE 、TEXT 、BLOB 或 CLOB 列上放置引用约束。
- 如果使用单列约束格式，您只能引用一列。

- 如果使用多列约束格式，则 REFERENCES 子句中列的最大数目是 16 ，并且如果页大小为 2 千字节时，这些列的总长度不能超过 390 字节。（最大长度随着页大小增加而增加。）

## 被引用列的缺省值

如果被引用表与引用表不同，则您不需要知道被引用列；缺省列为被引用表的主键列（或列组）。如果被引用表与引用表一样，则必须指定被引用列。

## 表内的引用关系

可以在同一表的两列之间建立引用关系。在以下示例中， employee 表中的 emp_num 列通过雇员编号唯一地标识了每个雇员。该表中的 mgr_num 列包含管理该雇员的经理的编号。在该示例中，mgr_num 将引用 emp_num 。在 mgr_num 列中出现重复的值，这是因为经理可以管理多个雇员。

```
CREATE TABLE employee
        (
        emp_num INTEGER PRIMARY KEY,
        mgr_num INTEGER REFERENCES employee (emp_num)
        );
```

其中行之间存在引用关系的表可以具有没有显式外键的 PRIMARY KEY 约束。有关递归查询其中行存在逻辑层次结构的多个级别的表的语法。请参阅 层级查询子句 。

## 创建引用约束时的锁定问题

当您创建引用约束时，将在被引用的表上放置互斥锁。当 CREATE TABLE 语句完成时，该锁才被释放。如果在支持事务记录的数据库中创建表并使用事务，则直到事务结束时锁才被释放。

## 使用单列约束格式的示例

这些示例指示了单列约束格式选项来定义缺省启用的外键约束，并声明禁用的引用约束的名称。

### 缺省启用的引用约束

以下示例使用单列约束格式定义 sub_accounts 和 accounts 表之间的引用关系。（术语*外键约束*和*引用约束*是同义词）。sub_accounts 表的 ref_num 列（外键）引用了 accounts 表中的 acc_num 列（外键）。

```
CREATE TABLE accounts (
            acc_num INTEGER PRIMARY KEY,
            acc_type INTEGER,
            acc_descr CHAR(20));
            CREATE TABLE sub_accounts (
            sub_acc INTEGER PRIMARY KEY,
            ref_num INTEGER REFERENCES accounts (acc_num),
            sub_descr CHAR(20));
```

以上定义 sub_accounts 表的 CREATE TABLE 语句的单列格式约束语法没有显式指定 ref_num 列是外键，但是 REFERENCES 关键字指定 ref_num 必须具有与 accounts 表中 acc_num 列的一些行的值相同的值。这意味着在 sub_accounts 是引用表，accounts 是被引用表的引用关系中，ref_num 列是外键。

在单列约束格式中，您不能显式指定 ref_num 列为外键。当在引用表的单列（或引用同一主键的列列表）上放置引用约束时，要包含 FOREIGN KEY 关键字，您必须代替使用多列约束格式语法来定义引用约束。

缺省情况下，sub_accounts 表上的约束不用过滤而启用，因为没有指定显式约束方式。可以使用 DISABLED 或 FILTERING 关键字在此示例中指定。SET Database Object Mode 语句的 SET CONSTRAINTS 选项可以重置现有约束的对象方式。

因为以上的 sub_accounts 示例没有为引用约束声明名称，所以数据库服务器在将此约束注册到 sysconstraints 系统目录表中时隐式生成标识符，并将它的方式（ E ）注册到 sysobjstate 系统目录表中。

**禁用的引用约束**

以下 CREATE TABLE 引用创建了 xeno_counts 表，并在它的 xeno_num 列和第一个示例中 accounts 表的 acc_num 列之间定义了引用约束。此单列约束格式语法还包含了约束定义，指定 DISABLED 作为它的约束方式，且声明 xeno_constr 作为外键约束的名称。此处 xeno_accounts 是引用表，accounts 是被引用表。

```
CREATE TABLE xeno_counts (
        xeno_acc INTEGER PRIMARY KEY,
        xeno_num INTEGER REFERENCES accounts (acc_num)
        CONSTRAINT xeno_constr DISABLED,
        xeno_descr CHAR(20));
```

在 DISABLED 方式，当 DML 操作在 xeno_counts 表中产生违例行时，不会执行 xeno_constr 约束。然而，要实现参照完整性，可以使 SET Database Object Mode 语句的 SET CONSTRAINTS 选项将约束的方式更改为 ENABLED。或者，START VIOLATION 语句将违例表与 xeno_counts 表关联后， SET CONSTRAINTS 可将 xeno_constr 约束重设为 FILTERING 方式。

## 使用 ON DELETE CASCADE 选项

使用 ON DELETE CASCADE 选项来指定从父表中删除某行时，是否需要从子表中删除相应的行。如果您不指定级联删除，则数据库服务器的缺省行为将阻止您删除表中的数据（如果有其它表引用它）。

如果指定此选项，则稍后当您删除父表中的行时，数据库服务器还删除任何与子表中的行（外键）相关的行。级联删除功能最主要的好处是，需要执行删除操作时它可以减少 SQL 语句的数量。

例如，all_candy 表中包含的 candy_num 列是主键。hard_candy 表将 candy_num 列作为外键引用。以下的 CREATE TABLE 语句将创建 hard_candy 表，其外键上具有级联删除选项：

```
CREATE TABLE all_candy
```

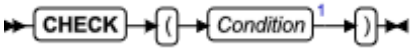
```
    (candy_num SERIAL PRIMARY KEY,
     candy_maker CHAR(25));

CREATE TABLE hard_candy
    (candy_num INT,
     candy_flavor CHAR(20),
     FOREIGN KEY (candy_num) REFERENCES all_candy
     ON DELETE CASCADE);
```

由于为子表指定了 ON DELETE CASCADE ，因此当从 **all_candy** 表中删除某行时，**hard_candy** 表中的相应行也将被删除。有关从具有级联删除的表中删除行时的语法限制和锁定影响的信息，请参阅级联删除表时的注意事项。

## CHECK 子句

使用 CHECK 子句来指定在 INSERT 或 UPDATE 语句中为某列指定数据***之前***应满足的条件。

CHECK 子句




该***条件***不能包含用户定义的例程。

在插入或更新期间，如果某行的检查约束等于 ***false*** ，则数据库服务器将返回错误。如果对某行进行检查约束时的值等于 NULL ，数据库服务器将不会返回错误。在某些情况下，您可能希望同时使用检查约束和 NOT NULL 约束。

## 使用搜索条件

定义了检查约束的***搜索条件***不能包含以下元素：用户定义的例程、子查询、聚集、主变量或行标识。此外，搜素条件还不能包含以下内部函数：CURRENT 、SYSDATE 、USER 、CURRENT_USER 、SITENAME 、 DBSERVERNAME 或 TODAY 。

当您在搜素条件中指定日期值时，确保为年指定了四位数，这样 **DBCENTURY** 环境变量就不会影响条件。当您指定了 2 位数的年份时，如果条件取决于缩写形式的年份值，则 **DBCENTURY** 环境变量将产生不可预料的结果。有关 **DBCENTURY** 的更多信息，请参阅 *《GBase 8s SQL 指南：参考》*。

更多情况下，数据库服务器从检查约束创建时就开始保留这些环境变量的设置。如果这些设置中的任意一个发生了更改，并且此更改将影响检查约束中对条件的求值，则对条件进行求值时将忽视新设置，使用最初的环境变量设置。

对于 BYTE 或 TEXT 列,可以检查是否有 NULL 或 not-NULL 值。该约束是 BYTE 或 TEXT 列上的唯一约束。

## 使用单列约束格式时的限制

当使用单列约束格式定义检查约束时，该检查约束无法依赖表中其它列的值。以下示例将创建有两列具有检查约束的 my_accounts 表，每个约束都是单列约束格式：

```
CREATE TABLE my_accounts (
        chk_id    SERIAL PRIMARY KEY,
        acct1     MONEY CHECK (acct1 BETWEEN 0 AND 99999),
        acct2     MONEY CHECK (acct2 BETWEEN 0 AND 99999));
```

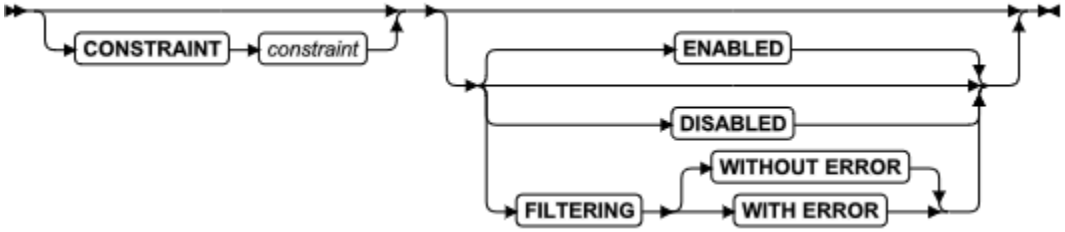
**acct1** 和 **acct2** 都是 MONEY 数据类型的列，其值必须在 0 到 99999 之间。然而，如果您想测试 **acct1** 的余额是否比 **acct2** 多，则不能使用单列约束格式。如果要创建在多列中检查值的约束，则必须使用多列约束格式。

## 约束定义

使用 CREATE TABLE 的约束定义部分，是为了：

- 为约束声明一个名称
- 将约束设置为禁用的、启用的或过滤方式。

约束定义




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constraint* | 约束的名称 | 必须在索引和约束名称中是唯一的 | 标识符 |

## 声明约束名称

数据库服务器将约束作为索引实现。每次使用单列或多列约束格式在列上放置数据约束，但没有声明***约束***名称时，数据库服务器将创建一个约束并在 **sysconstraints** 系统目录表中为该约束添加一行。

数据库服务器还生成一个标识符并在 **sysindices** 系统目录表中为不与已有约束共享索引的每个新主键、唯一或引用约束添加一行。即使为约束声明了一个名称，数据库服务器也会生成一个名称并出现 **sysindices** 表中。（该系统目录表还包含 **sysindices** 表上的视图，称为 **sysindexes** ，列出了复合索引的每个组件。）

如果愿意的话，您还可以为该约束指定一个有意义的名称。在数据库中的约束和索引的名称中，该名称必须是唯一的。

如果约束违例则约束名称将出现错误消息中。当您使用 ALTER TABLE 语句的 DROP CONSTRAINT 子句时可以使用该名称。

当您使用 SET Database Object Mode 语句或 SET Transaction Mode 语句更改约束方式时也可以指定约束名称以及在 DROP INDEX 语句中约束作为用用户定义名称的索引实现 。

在兼容 ANSI 的数据库中，当声明任何类型约束名称时，在数据库内***所有者***名称和***约束***名称的组合必须是唯一的。

### *数据库服务器生成的约束名称*

如果未指定约束名称，数据库服务器将使用以下模板生成一个约束名称：

***<constraint_type><tabid>_<constraintid>***

模板中，***constraint_type*** 是字母 **u** 时标识唯一约束或主键约束，**r** 表示引用约束，**c** 表示检查约束，**n** 表示 NOT NULL 约束。模板中，***tabid*** 和 ***constraintid*** 分别来自 **systables** 和 **sysconstraints** 系统目录表的 **tabid** 和 **constrid** 列的值。例如，唯一约束的约束名称可能看起来像" **u111_14**"前面有一个空格）。

如果生成的名称与已有的名称冲突，则数据库服务器将返回一个错误并且您必须提供一个显式的约束名称。

sysindexes（或 sysindices）中生成的索引名称具有以下格式：

[blankspace]<***tabid*>_<*constraintid***>

例如，索引名称类似于" **111_14** "（这里是有引号显示空白的位置）。

## 选择约束方式选项

使用约束方式(ENABLED 、DISABLED 和 FILTERING)选项来控制 INSERT 、DELETE 、MERGE 和 UPDATE 操作中约束的行为。

对于 CREATE TABLE 语句定义的约束，这些是可选的。

| 方式 | 作用 |
|---|---|
| DISABLED | 不要在 INSERT 、DELETE 和 UPDATE 操作期间强制约束 |
| ENABLED | 在 INSERT 、DELETE 和 UPDATE 操作时强制使用约束。如果目标行引起约束违例，则该语句失败。该方式是缺省值。 |
| FILTERING | 如果 START VIOLATIONS 语句创建了违例表和诊断表，在 INSERT 、DELETE 和 UPDATE 操作时强制使用约束。如果目标行引起约束违例，则该语句继续进行。数据库服务器将有问题的行写到与目标关联的违例表中，并将诊断信息写到关联的诊断表中。 |

如果选择过滤方式，则可以指定 WITHOUT ERROR 或 WITH ERROR 选项。以下列表将说明这些 ERROR 选项。

| 错误选项 | 作用 |
| --- | --- |
| WITH ERROR | 在 INSERT 、DELETE 和 UPDATE 操作期间，如果违反过滤方式约束将返回违反完整性的错误。 |
| WITHOUT ERROR | 在 INSERT 、DELETE 和 UPDATE 操作期间，如果违反过滤方式约束将不返回违反完整性的错误。这是缺省错误选项。 |

**注：**

要使 FILTERING WITHOUT ERROR 方式具有这些作用，您还必须使用 START VIOLATIONS TABLE 语句为定义约束的目标表启动违例表和诊断表。您可以发出这些语句

- 在您设置表的任何约束为过滤方式之前，
- 或在您将约束设置成过滤方式后，但在任何用户在对表中的行执行 INSERT、DELETE 或 UPDATE 操作之前。

约束方式注册在 sysobjstate 系统目录表中。

**外键约束的 NOVALIDATE 方式**

以上列出的方式只是 SET Database Object Mode 语句的 SET CONSTRAINTS 选项在其重置现有外键约束方式时可以指定的约束方式的一个子集。它们还是 ALTER TABLE ADD CONSTRAINT 语句在现有表上创建新的外键约束时能指定的约束方式的子集。

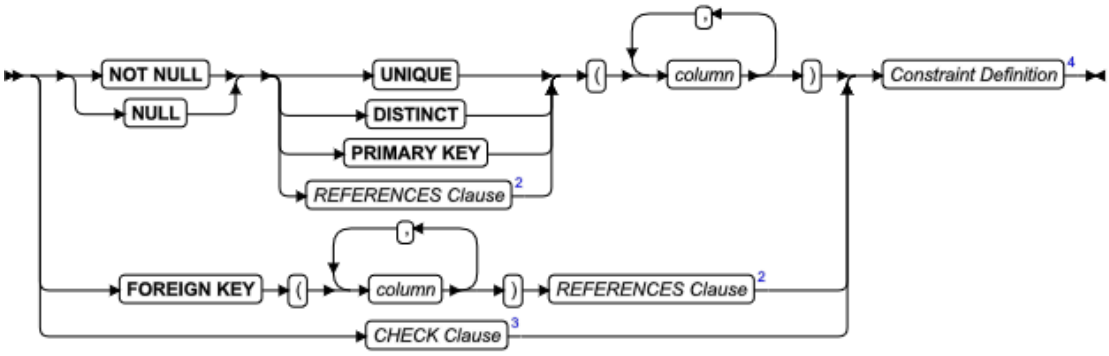
ALTER TABLE ADD CONSTRAINT 和 SET CONSTRAINTS 语句可在约束定义中包含 NOVALIDATE 关键字来指定这些额外外键约束方式中的一种方式。其效果是当创建或启用外键约束时，数据库服务器跳过对违例的现有行的检查，从而减少了处理 DDL 语句所需的时间和资源。然而，当语句执行完毕后，每个 NOVALIDATE 方式自动还原成 ENABLED 或 FILTERING 方式。因此，NOVALIDATE 关键字无法阻止随后表的 DML 操作的参照完整性的强制执行，因为 NOVALIDATE 关键字不能持续超越定义它们的 DDL 语句。

由于大多数表在它们创建后都是空表，因此对现有行的参照完整性检查一般不会在建立表时发生，而且 CREATE TABLE 语句不支持 NOVALIDATE 约束方式。然而，在有外键约束的非空表需要被移动到另一个数据库或数据仓库中的上下文中，这些方式是很有效率的。

## 多列约束格式

使用多列约束格式将单列或多列与约束关联起来。这种单列约束格式的备用方法允许您将多列与一个约束关联起来。

多列约束格式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要放置约束的列 | 不能是 BYTE 、TEXT 、BLOB 、CLOB | 标识符 |

多列约束具有这些基数和大小限制：

- 指定的列的名称不能超过 16 个。
- 在 GBase 8s 中，列列表的最大总长度依赖于页大小，其计算公式为：

  MAXLength = (((PageSize - 93)/3) -1)

  - 对于 2K 的页大小，总长度不能超过 650 字节。
  - 对于 16K 的页大小，总长度不能超过 5429 字节。

此处的反斜杠（/）符号代表整除。

当您定义唯一约束时（通过使用 UNIQUE 或 DISTINCT 关键字），列在约束列表中只能出现一次。

使用多列约束格式，您可以完成以下任务：

- 为一组一列或多列的集合创建数据完整性约束
- 为约束指定助记符名称
- 指定在插入、删除和更新操作期间控制约束行为的约束方式选项。

当您使用此格式时，可以创建主键和外键的组合，或者定义能比较不同列中数据的检查约束。

另见 唯一约束和唯一索引的区别 章节。

## 使用多列格式约束的限制

使用多列约束格式时，不能为这些列定义缺省值。此外，不能在同一张表的两列之间建立参考引用关系。

要定义列的缺省值或者建立同一张表中两列之间的引用关系，请分别参考单列约束格式和表内的引用关系。

### *在约束中使用大对象类型*

不能在 BYTE 或 TEXT 列上放置唯一、主键或引用（FOREIGN KEY）约束。但是可以使用检查约束来检查 NULL 或 non-NULL 值。

不能在 BLOB 或 CLOB 列上放置唯一或主键约束。如果约束是在包含存储了加密数据的列的列集合上，则 GBase 8s 无法强制执行此约束。

您可以在以下各节中找到对特定约束的详细讨论：

| 约束 | 有关更多信息，请参阅 | 有关示例，请参阅 |
|---|---|---|
| CHECK | CHECK 子句 | 在多个列上定义检查约束 |
| DISTINCT | 使用 UNIQUE 或 DISTINCT 约束 | 多列约束格式的示例 |
| FOREIGN KEY | 使用 FOREIGN KEY 约束 | 定义组合的主键和外键 |
| PRIMARY KEY | 使用 PRIMARY KEY 约束 | 定义组合的主键和外键 |
| UNIQUE | 使用 UNIQUE 或 DISTINCT 约束 | 多列约束格式的示例 |

## 使用 FOREIGN KEY 约束

外键***连接***并建立表之间的相关性。即，创建了一个引用约束。（有关引用约束的更多信息，请参阅 REFERENCES 子句。）

外键将引用表中唯一键或主键。对于外键列中的每个条目，如果所有的外键列都包含 non-NULL 值，则匹配的条目必须存在于唯一或主键列中。

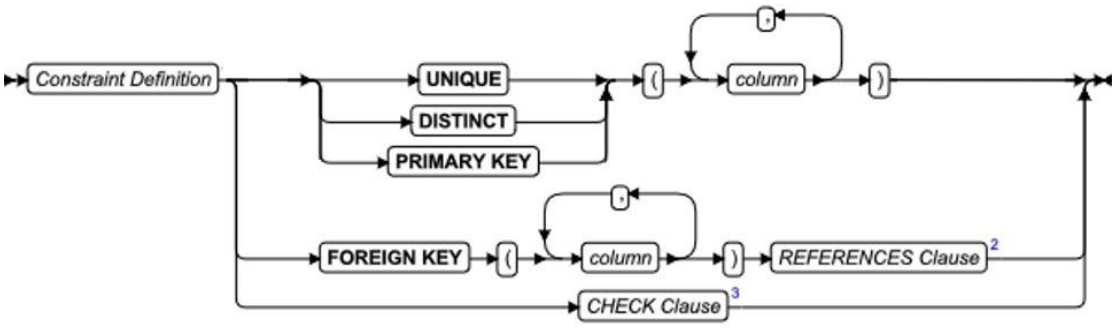
不能将 BYTE 或 TEXT 列指定为外键。

不能将 BLOB 或 CLOB 列指定为外键。

## 使用 Oracle 多列格式约束

使用多列约束格式将单列或多列与约束关联起来。支持先约束名定义后指定约束的语法顺序定义约束。

Oracle 多列约束格式




此功能仅在 Oracle 模式下支持，可以发出以下语句设置 Oracle 模式：

set environment sqlmode ‘oracle’;

此功能支持的约束包括 UNIQUE、DISTINCT、PRIMARY KEY、FOREIGN KEY、CHECK。

在 Oracle 模式下，不能使用 GBase 8s 原生多列约束格式语法。

## 多列约束格式的示例

以下示例创建了一个名为 order_items 的标准表，它有一个使用多列约束格式的名为 items_constr 的唯一约束：

```
CREATE TABLE order_items
        (
        order_id SERIAL,
        line_item_id INT not null,
        unit_price DECIMAL(6,2),
        quantity INT,
        UNIQUE (order_id,line_item_id) CONSTRAINT items_constr
        );
```

有关约束名称的信息，请参阅声明约束名称。

### *在多个列上定义检查约束*

当您使用多列约束格式定义检查约束时，检查约束可以应用于相同表的多列。（但是，不能创建其 ***condition*** 使用来自其它表中某列的值的检查约束。）

该示例比较了新表中的两列，acct1 和 acct2：

```
CREATE TABLE my_accounts
        (
        chk_id     SERIAL PRIMARY KEY,
        acct1      MONEY,
        acct2      MONEY,
        CHECK (0 < acct1 AND acct1 < 99999),
        CHECK (0 < acct2 AND acct2 < 99999),
        CHECK (acct1 > acct2)
        );
```

在此示例中，列 **acct1** 必须比大于列 **acct2** ，否则插入或更改会失败。

### *定义组合的主键和外键*

当您使用多列约束格式时，可以创建一个组合关键字。***组合关键字***指定多列的主键或外键约束。

以下示例创建两个表。第一个表的组合关键字表现为主键。第二表的组合关键字表现为外键。

```
CREATE TABLE accounts (
        acc_num INTEGER,
        acc_type INTEGER,
        acc_descr CHAR(20),
        PRIMARY KEY (acc_num, acc_type)
        );
```

```
CREATE TABLE sub_accounts (
        sub_acc INTEGER PRIMARY KEY,
        ref_num INTEGER NOT NULL,
        ref_type INTEGER NOT NULL,
        sub_descr CHAR(20),
        FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
        (acc_num, acc_type)
        );
```

在此示例中，**sub_accounts** 表的外键 **ref_num** 和 **ref_type**，引用 **accounts** 表中的组合关键字 **acc_num** 和 **acc_type**。如果在插入和更新期间，当您试图向 **sub_accounts** 表中插入一行，而其中 **ref_num** 和 **ref_type** 的值没有精确地与 **accounts** 表中已有行的 **acc_num** 和 **acc_type** 的值对应，则数据库服务器将返回一个错误。

在引用和被引用的列之间，引用约束必须具有一对一的关系。换句话说，如果组合关键字是一组列的结合（组合关键字），则外键也必须是与组合关键字对应的一组列的集合。

由于数据库服务器的缺省行为。因此当您创建外键引用时，并不需要显式地引用组合关键字（acc_num 和 acc_type）。可以如下重写前面示例的引用部分：

```
FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
```

## Oracle 多列约束格式的示例

此功能仅在 Oracle 模式下支持，可以发出以下语句设置 Oracle 模式：

```
set environment sqlmode ‘oracle’;
```

以下示例创建了一个名为 order_items 的标准表，它有一个使用多列约束格式的名为 items_constr 的唯一约束：

```
CREATE TABLE order_items
        (
        order_id SERIAL,
        line_item_id INT not null,
        unit_price DECIMAL(6,2),
        quantity INT,
        CONSTRAINT items_constr UNIQUE (order_id,line_item_id)
        );
```

有关约束名称的信息，请参阅声明约束名称。

### *在多个列上定义检查约束*

当您使用多列约束格式定义检查约束时，检查约束可以应用于相同表的多列。（但是，不能创建其 ***condition*** 使用来自其它表中某列的值的检查约束。）

该示例比较了新表中的两列，acct1 和 acct2：

```
CREATE TABLE my_accounts
        (
        chk_id    SERIAL PRIMARY KEY,
        acct1     MONEY,
        acct2     MONEY,
        CONSTRAINT items_con1 CHECK (0 < acct1 AND acct1 < 99999),
        CONSTRAINT items_con2 CHECK (0 < acct2 AND acct2 < 99999),
        CONSTRAINT items_con3 CHECK (acct1 > acct2)
        );
```

在此示例中，列 **acct1** 必须比大于列 **acct2** ，否则插入或更改会失败。

### *定义组合的主键和外键*

当您使用多列约束格式时，可以创建一个组合关键字。***组合关键字***指定多列的主键或外键约束。

以下示例创建两个表。第一个表的组合关键字表现为主键。第二表的组合关键字表现为外键。

```
CREATE TABLE accounts (
        acc_num INTEGER,
        acc_type INTEGER,
        acc_descr CHAR(20),
        CONSTRAINT items_pk PRIMARY KEY (acc_num, acc_type)
        );

CREATE TABLE sub_accounts (
        sub_acc INTEGER PRIMARY KEY,
        ref_num INTEGER NOT NULL,
        ref_type INTEGER NOT NULL,
        sub_descr CHAR(20),
        CONSTRAINT items_fk FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
        (acc_num, acc_type)
        );
```

在此示例中，**sub_accounts** 表的外键 **ref_num** 和 **ref_type**，引用 **accounts** 表中的组合关键字 **acc_num** 和 **acc_type**。如果在插入和更新期间，当您试图向 **sub_accounts** 表中插入一行，而其中 **ref_num** 和 **ref_type** 的值没有精确地与 **accounts** 表中已有行的 **acc_num** 和 **acc_type** 的值对应，则数据库服务器将返回一个错误。

在引用和被引用的列之间，引用约束必须具有一对一的关系。换句话说，如果组合关键字是一组列的结合（组合关键字），则外键也必须是与组合关键字对应的一组列的集合。

## 约束缺省的索引创建策略

当您创建带有唯一或主键约束的表时，数据库服务器为每一个约束创建一个唯一并升序的内部索引。

当创建带有引用约束的表时，数据库服务器创建一个升序的内部索引，它允许引用约束中您指定的每一列有重复的值。

内部索引占据与其表相同的存储位置。对于已分片的表，内部索引的分片占据与您为此表分片指定的相同的 dbspace 分区（或者在某些情况下，为数据库 dbspace）。

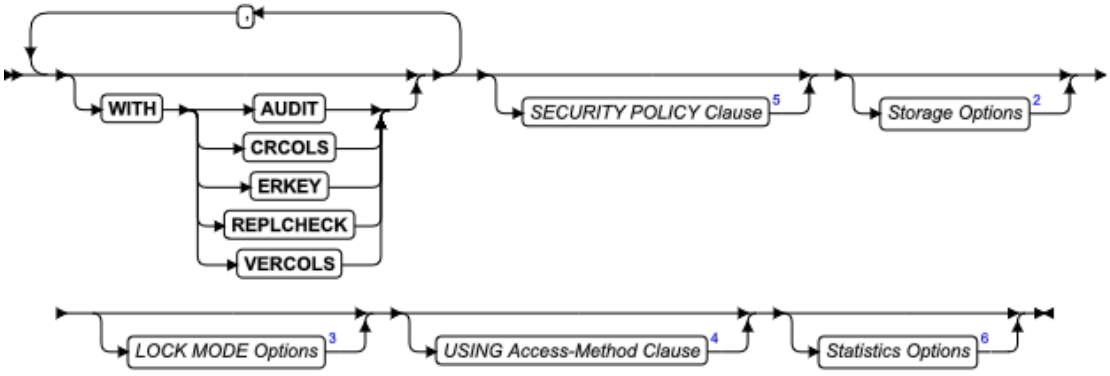
如果您需要索引分片策略独立于底层表分片，则创建该表时不要定义此约束。而使用 CREATE INDEX 语句创建具有期望分片存储策略的唯一索引。然后使用 ALTER TABLE 语句添加约束。新的约束使用先前定义的索引。

**重要：** 在非日志记录的数据库中，*detached checking* 是唯一可用的约束检查。已拆离检查意味着一行一行地进行约束检查。

## Options 子句

CREATE TABLE 语句的 Options 子句提供创建各类隐藏列的选项。支持行级别审计、基于标签的安全策略、存储位置、分布存储策略、页扩展大小、锁定粒度、用户定义的存取方法和可影响列分布统计信息集合的属性。

Options




**表选项的顺序**

此语法图显示了包含多个下列选项的 CREATE TABLE 语句中表选项的顺序：

- WITH 选项
- SECURITY POLICY 选项
- 存储选项
- LOCK MODE 选项
- USING Access-Method 子句
- 统计信息选项。

如果您包含了多个 WITH 选项，则在连续的 WITH 选项之间需要逗号分隔符（,）。

有关在分片表的行中使用 WITH ROWIDS 选项的信息，请参阅使用 WITH ROWIDS 选项。

同一 CREATE TABLE 语句中的多个 WITH 选项不要求先后顺序，但是您包含的所有的 WITH 选项必须在上述列表中的任何其它五个表选项之前。

例如，下列两个 CREATE TABLE 语句是等价的：

```
CREATE STANDARD TABLE IF NOT EXISTS myShadowy_tab(colA INT, colB CHAR)
  WITH ERRKEY, WITH CRCOLS, WITH AUDIT LOCK MODE ROW;

CREATE STANDARD TABLE IF NOT EXISTS myShadowy_tab(colA INT, colB CHAR)
  WITH AUDIT, WITH ERRKEY, WITH CRCOLS LOCK MODE ROW;
```

如果您在相同的数据库中连续发出这些语句，则第二条语句失败，因为第一条语句创建的名为 myShadowy_tab 的表已经在数据库中存在。由于 IF NOT EXISTS 关键字，冗余的第二条语句不会返回错误，但是它不会创建新表。

以下示例发生错误而失败，因为其它 Options 子句不能在 WITH 子句前面：

```
CREATE TABLE shadow_columns (colA INT, colB CHAR)
            LOCK MODE ROW WITH AUDIT, WITH ERRKEY, WITH CRCOLS; --bad options
order
```

下一个 CREATE TABLE 示例也失败，因为在同一 Options 子句中 Statistics 选项不能在 LOCK MODE 选项前面：

```
CREATE TABLE shadow_columns (colA INT, colB CHAR)

STATCHANGE 25 STATLEVEL TABLE LOCK MODE PAGE; --bad options order
```

**重要：** 您不能使用 CREATE TABLE 语句的 Options 子句在已经存在的表结构中添加新的隐藏列或做其它更改。要做更改，例如，当创建现有表时，Options 子句包含或忽略 WITH 关键字，使用 ALTER TABLE ADD 或 ALTER TABLE DROP 语句的适当的选项。有关更多信息，请参阅 ALTER TABLE 语句。

## 使用 WITH AUDIT 子句

使用 WITH AUDIT 关键字创建表，如果启用了选择性行级别，则该表将包含在行级别升级的表的集合中。

如果创建带有 WITH AUDIT 子句的表，则当选择性行级别审计启动时，表中行级别审计事件会重新排序。在本表上应用 WITH AUDIT 属性不会启用选择性行级别审计。此审计类型在使用 gaudit -R 命令将 adtcfg 文件的 ADTROWS 参数设置成 1 或 2 时启用。

您必须具有 RESOURCE 或 DBA 权限才能运行带有 WITH AUDIT 子句的 CREATE TABLE 语句。

## 使用 WITH CRCOLS 选项

使用 WITH CRCOLS 关键字创建两个影子列，Enterprise Replication 将它们用于冲突解决。第一列 **cdrserver** 中包含最近发生修改的数据库服务器的标识。第二列 **cdrtime** 中包含最近一次修改的时间戳记。必须在您可以使用时间戳记或 UDR 冲突解决之前添加这两列。这两列是隐藏的影子列，因为它们不能被索引且不能在系统目录表中查看。

对于大多数数据库操作，cdrserver 和 cdrtime 列都是隐藏的。例如，如果您在创建表时包含了 WITH CRCOLS 关键字，则 cdrserver 和 cdrtime 列将有以下的行为：

- 当查询时指定星号（*）作为投影列表时（如下列语句所示），它们不会返回：

  SELECT * FROM ***tablename***;
- 当您询问有关表中列的信息时，它们并不出现在 DB-Access 中。
- 它们并不包含在 ***tablename*** 的 **systables** 系统目录表条目的列数目（**ncols**）中。

要查看 cdrserver 和 cdrtime 的内容，请在 SELECT 语句的投影列表中显式指定这些列，如以下示例所示：

SELECT cdrserver, cdrtime FROM ***tablename***;

有关如何使用这些选项的更多信息，请参阅 *GBase 8s Enterprise Replication 指南*。

## 使用 WITH ERKEY 关键字

使用 WITH ERKEY 关键字创建 ERKEY 影子列，它可以被 Enterprise Replication 用作复制键。

*ERKEY 影子列（**ifx_erkey_1** 、**ifx_erkey_2** 和 **ifx_erkey_3**）是可见的影子列，因为它们可以被索引且能在系统目录表中查看。在创建 ERKEY 影子列之后，会在使用这些列的表上创建新的唯一索引和唯一约束。Enterprise Replication 使用此索引作为复制键。*

对于大多数数据库操作，ERKEY 列是隐藏的。例如，如果您在创建表时包含了 WITH ERKEY 关键字，则 ERKEY 列具有以下行为：

- 当查询时指定星号（*）作为投影列表时（如下列语句所示），它们不会返回：

  SELECT * FROM ***tablename***;
- 当您询问有关表中列的信息时，它们并不出现在 DB-Access 中。
- 它们并不包含在 ***tablename*** 的 **systables** 系统目录表条目的列数目（**ncols**）中。

要查看 ERKEY 列的内容，请在 SELECT 语句的投影列表中显式指定这些列，如以下示例所示：

SELECT ifx_erkey_1, ifx_erkey_2, ifx_erkey_3 FROM customer;

示例

*在以下示例中，ERKEY 影子列添加到 **customer** 表中：*

CREATE TABLE customer (id INT) WITH ERKEY;

## 使用 WITH REPLCHECK 关键字

使用 WITH REPLCHECK 关键字创建 ifx_replcheck 影子列，Enterprise Replication 将该列用于一致性检查。

**ifx_replcheck** 列是可见的影子列，因为它们可以被索引且能在系统目录表中查看。创建 **ifx_replcheck** 影子列之后，必须在主键和 **ifx_replcheck** 列上创建唯一索引。**ifx_replcheck** 影子列必须是该索引中的最后一列。Enterprise Replication 使用此索引加速一致性检查。

对于大多数数据库操作，ifx_replcheck 列是隐藏的。例如，如果创建表时使用了 WITH REPLCHECK 关键字，则 ifx_replcheck 列包含以下行为：

- 当查询时指定星号（*）作为投影列表时（如下列语句所示），它们不会返回：

  SELECT * FROM *tablename*;
- 当您询问有关表中列的信息时，它们并不出现在 DB-Access 中。
- 它们并不包含在 ***tablename*** 的 **systables** 系统目录表条目的列数目（**ncols**）中。

要查看 ifx_replcheck 列的内容，请在 SELECT 语句的投影列表中显式指定这些列，如以下示例所示：

SELECT ifx_replcheck FROM customer;

**示例**

在下列示例中，将 **ifx_replcheck** 影子列添加到 **customer** 表中：

CREATE TABLE customer (id int) WITH REPLCHECK;

## 使用 WITH VERCOLS 选项

使用 WITH VERCOLS 关键字创建两个影子列， GBase 8s 使用它们来支持辅助服务器上的更改操作。

第一列 **ifx_insert_checksum** 中包含首次创建时的行的校验和。第二列 **ifx_row_version** 中包含行的版本号。当行第一次插入时，生成 **ifx_insert_checksum** ，且 **ifx_row_version** 将设置为 1 。每当行更新时，**ifx_row_version** 加一，但 **ifx_insert_checksum** 不会更改。这两列是可见的影子列，因为它们可以被索引且能在系统目录表中查看。

对于大多数数据库操作，ifx_insert_checksum 和 ifx_row_version 列是隐藏的。例如，如果您创建表时包含了 WITH VERCOLS 关键字，则 ifx_insert_checksum 和 ifx_row_version 列具有以下行为：

- 当查询时指定星号（*）作为投影列表时（如下列语句所示），它们不会返回：
- SELECT * FROM ***tablename***;
- 当您询问有关表中列的信息时，它们并不出现在 DB-Access 中。
- 它们并不包含在 ***tablename*** 的 **systables** 系统目录表条目的列数目（**ncols**）中。

要查看 ifx_insert_checksum 和 ifx_row_version 列的内容，请在 SELECT 语句的投影列表中显式指定这些列，如以下示例所示：

SELECT ifx_insert_checksum, ifx_row_version FROM ***tablename***;

当启用了行版本化时，则 **ifx_row_version** 在行每更新一次时增加一；但是 Enterprise Replication 做出的行更改不会增加行的版本。要在使用 Enterprise Replication 的服务器上更改行的版本，必须在复制参与者定义中包含 **ifx_row_version** 列。

有关如何使用此选项的更多信息，请参阅 *GBase 8s 管理员指南* 。

## SECURITY POLICY 子句

该可选的 Security Policy 子句可使用以下语法来指定与表相关联的安全策略。

SECURITY POLICY Clause


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *policy* | 安全策略的名称 | 在数据库中必须存在 | 标识符 |

只有 DBSECADM 才能创建包含 Security Policy 子句的表，该子句为此表指定安全策略。

以下准则适用于可通过 CREATE TABLE 语句中包含有效的 SECURITY POLICY 子句来保护的表：

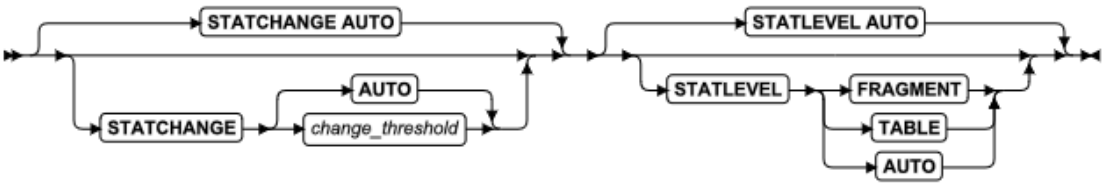
- 除非表具有与自己相关联的安全策略，且有行被保护或至少有一列被保护，那么该表是受保护的表，否则它没有被保护。前一种情况指示该表是具有行级粒度保护的表，后一种情况指示表是具有列级粒度保护的表。
- 如果该表不具备与其相关联的安全策略，则使用 IDSSECURITYLABEL 列子句保护列失败。
- 如果该表不具备与其相关联的安全策略，则使用 COLUMN SECURED WITH 列子句保护列失败。
- 一个表最多只能有一个安全策略。
- 一个表可具有任何数量的被保护列，且每个被保护列可具有不同的标签，或者一些被保护的列可以共享相同的标签。
- 安全策略不能与临时表或在表层次结构中的类型表相关联。

## CREATE TABLE 语句的 Statistics 选项

使用 CREATE TABLE 语句的 Statistics 选项子句设置已分片表或未分片表的 STATCHANGE 属性的值，和已分片表的 STATLEVEL 属性。

### 语法

这些表属性控制自动重新计算(STATCHANGE)的阈值和数据分布统计信息的粒度(STATLEVEL)。




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *change_threshold* | 定义过时分布统计信息的已更改数据的百分比 | 必须是 0 - 100 之间的整数 | 精确数值 |

### 用法

Statistics 选项子句可定义在 SQL 语句以 LOW 、MEDIUM 或 HIGH 方式运行下，允许用户控制 UPDATE STATISTICS 行动的表统计信息属性。

Statistics 选项子句可以设置的两个表属性为 STATCHANGE 和 STATLEVEL ：

STATCHANGE 表属性指定需要考虑统计信息过时的变更（来自自上次重新计算分布统计信息表中或分片中对行的 UPDATE \DELETE \MERGE 和 INSERT 操作）的最小百分比。可以指定 0 - 100 范围内的整数或者使用 AUTO 关键字将 ONCONFIG 文件中或会话环境中当前的 STATCHANGE 配置参数作为缺省更改的阈值。

选择性更新表和分片统计信息的自动方式可通过以下任意方法启用：

- 将 AUTO_STAT_MODE 配置参数设置成 1 （或者不设置）。系统缺省启用自动方式。
- 将 AUTO_STAT_MODE 会话环境变量设置成 "ON"。在当前会话期间启用自动方式。
- UPDATE STATISTICS 语句包含 AUTO 关键字。当当前数据库运行时启用自动方式。

当启用自动方式时，UPDATE STATISTICS 语句使用显式或缺省的 STATCHANGE 值来标识统计信息丢失或过时的表、索引或分布，以及只是统计信息丢失或过时的更改。有关 UPDATE STATISTICS 操作的自动方式的更多信息，请参阅 *GBase 8s 管理员参考*中关于 AUTO_STAT_MODE 配置参数的描述。另见 AUTO_STAT_MODE 环境选项 和 使用 FORCE 和 AUTO 关键字.

分片表的 STATLEVEL 属性可以决定它的数据分布和索引统计信息的粒度级别。如果在创建时没有指定任何值，则它可以是以下三个值的其中之一（AUTO 是缺省的）：

- TABLE 指定为表创建的所有的分布都是表级别的。
- FRAGMENT 指示按每一个分片创建和维护分布。
- AUTO 指定数据库服务器在运行时应用标准以确定分片级分布是否必要。这些标准要求以下所有的条件都为真：
    - SYSSBSPACENAME 配置参数设置指定了现有 sbspace 。
    - 该表按 EXPRESSION 、INTERVAL 或 LIST 策略分片。
    - 该表拥有超过百万条的行。

  如果任何这些标准都不满足，则数据库服务器创建表级别分布，而非分片级分布。

这些属性总是应用。如果 STATLEVEL 设置是 AUTO ，则该设置重写缺省值。

**注：** SYSSBSPACENAME 配置参数（在数据库例程初始化时必须设置）指定数据库服务器存储分片级数据分布统计信息的 sbspace 。这些统计信息作为 BLOB 对象存储在 **syfragsdist** 系统目录表的 **encdist** 列中。对于支持分片级统计信息的数据库服务器，SYSSBSPACENAME 配置参数设置指定现有 sbspace 。

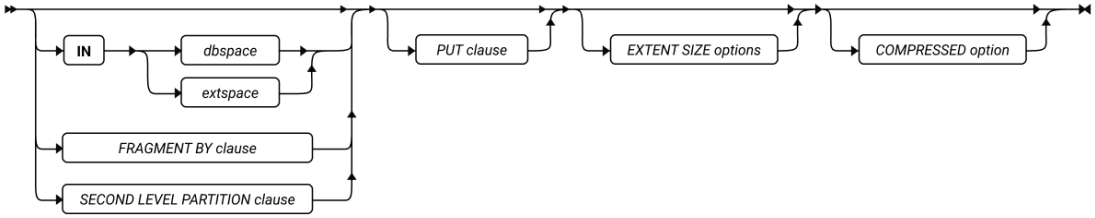
如果您使用 Statistics 选项子句将 STATLEVEL 属性设置为 FRAGMENT，则当以下任一条件为真时，数据库服务器返回错误：

- 没有设置 SYSSBSPACENAME 配置参数。
- SYSSBSPACENAME 指定的 sbspace 没有合适地按 gspaces -c -S 命令分配。

## 存储选项

使用 CREATE TABLE 语句的 FRAGMENT BY 子句、SECOND LEVEL PARTITION 子句、PUT 子句、EXTENT 大小选项和 COMPRESSED 选项指定存储位置、分布方案、表的 extent 大小以及该表的大量新的数据行是否启用自动压缩。

Storage 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbspace* | 存储表的 Dbspace | 必须存在 | 标识符 |
| *extspace* | 在 gspaces 命令中声明的名称，指的是数据库服务器之外的存储区域 | 必须存在 | 请参阅文档以了解您使用的存取方法 |

**用法**

为该表指的位置、分布方案和 extent 大小的存储选项是 SQL 语法 ANSI/ISO 标准的扩展。

如果使用 USING 存取方法子句来指定存取方法，则该方法必须支持存储空间。

您可以为表指定一个不同于数据库存储位置的 dbspace，或将表在多个 dbspace 之间分片，或在一个或多个 dbspace 的多个分区之间分片。

如果没有指定 IN 子句也没有指定分片方案，则新表将驻留在当前表所在的同一个 dbspace 中。但是，如果您启用了自动定位和分片，则表在被服务器选定的 dbspace 中创建和分片。要启用表的自动定位和分片功能，请将 AUTOLOCATE 配置参数或会话环境变量设置为正整数。该整数值代表初始分配给该表的分片数量。其它的分片随着表的增长而添加。

在 GBase 8s 中，您可以使用 PUT 子句为智能大对象指定存储选项。

**注：** 如果您的表包含简单对象（TEXT 或 BYTE），则可以为每个对象指定单独的 blobspace 。

# 使用 IN 子句

使用 IN 子句为该表指定存储空间。您指定的存储空间必须已经存在。

## *在 dbspace 中存储数据*

可以使用 IN 子句来隔离表。例如，如果 history 数据库在 dbs1 dbspace 中，但是您希望将 family 数据放在名为 famdata 的一个单独的 dbspace 中，请使用以下语句：

```
CREATE DATABASE history IN dbs1;

        CREATE TABLE family
        (
        id_num          SERIAL(101) UNIQUE,
        name             CHAR(40),
        nickname        CHAR(20),
        mother           CHAR(40),
        father          CHAR(40)
        )
        IN famdata;
```

有关如何在不同的 dbspace 中存储和管理表的信息，请参阅 *GBase 8s 管理员指南* 。

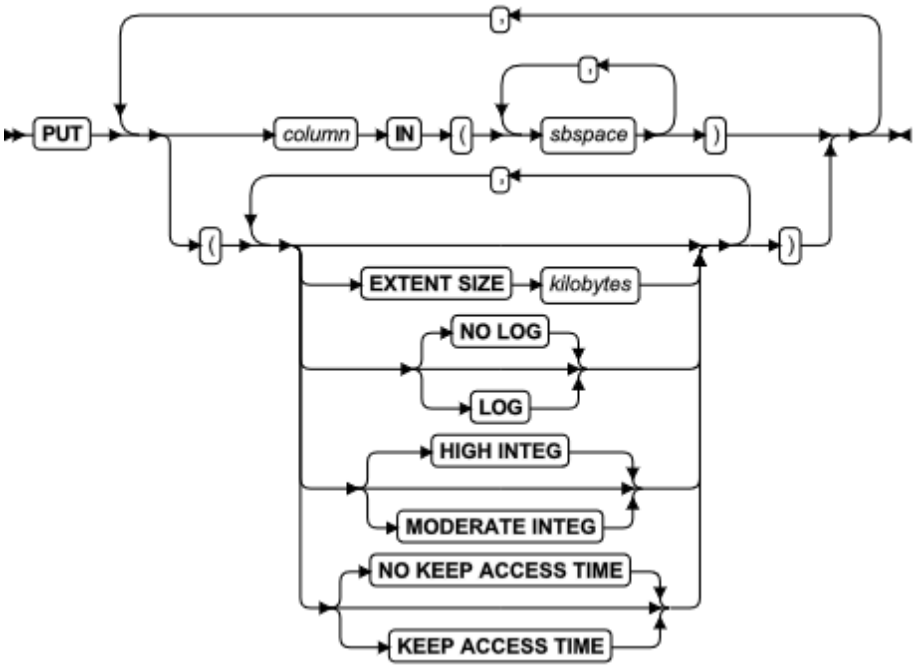
### *在 extspace 中存储数据*

通常，将 extspace 存储选项与 USING 存取方法子句 一起使用。有关更多信息，请参阅您使用的存取方法的文档。

## PUT 子句

使用 PUT 子句指定存储空间以及要包含智能大对象的每个列的特征。

PUT Clause




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 存储在 *sbspace* 中的列 | 必须包含 BLOB 、CLOB 、用户定义的或复杂数据类型 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *kilobytes* | 要为 extent 大小而分配的千字节数 | 必须是整数值 | 精确数值 |
| *sbspace* | 智能大对象的存储区域的名称 | 必须存在 | 标识符 |

指定的列不能是 ***column.field*** 格式。即，您所存储的智能大对象不能是 ROW 类型的一个字段。

**指定存储位置**

每个智能大对象都存储在单独的 sbspace 中。除非 PUT 子句 指定了另外的区域，否则 SBSPACENAME 配置参数指定在其中创建智能大对象的系统缺省值。

例如，以下示例创建只包含了 BLOB 数据类型列的 **tabwblob** 表。声明该列的名称为 **image01**，且 PUT 子句指定所有的 BLOB 对象的存储位置为 **sbspace01** ：

```
CREATE TABLE tabwblob
        (
        image01 BLOB
        ) PUT image01 IN (sbspace01);
```

要使以上示例有效，则 **sbspace01** 必须已经存在。因为 PUT 子句没有指定其它选项，所以 **sbspace01** 具有 extent 大小的缺省值和 PUT 子句能定义的其它存储特征的缺省值，包括以下定义中的 NO LOG 、HIGH INTEG 和 NO KEEP ACCESS TIME 。

PUT 子句可为 BLOB 和 CLOB 列列表指定存储位置。以下示例定义了 tabw2blobs 表，它有两个列，image02 列是 BLOB 类型，commentary03 列是 CLOB 类型。在下一个示例中，PUT 子句指定了在这两个列中的所有的智能大对象都将存储在同一智能大对象空间 sbspace01 中：

```
CREATE TABLE tabw2blobs
        (
        image02 BLOB,
        commentary03 CLOB
        ) PUT image02 IN (sbspace01),
        commentary03 IN (sbspace01);
```

可以在循环分布方案中指定多个 sbspace 来存储同一 BLOB 或 CLOB 列，这样每个 sbspace 中的智能大对象旧大致相同。单个列的 sbspace 列表（按逗号分隔）必须包含在括号内。

下一示例定义了 **tabw2sblobs** 表，它具有两列，**image04** 列是 BLOB 类型，**commentary05** 列是 CLOB 类型。PUT 子句指定 **image04** 列中的 BLOB 对象存储在两个 sbspace 中：**sbspace01** 和 **sbspace02** ，而 **image05** 列中所有的 CLOB 对象存储在 **sbspace03** sbspace 中：

```
CREATE TABLE tabw2sblobs
        (
        image04 BLOB,
        commentary05 CLOB
        ) PUT image04 IN (sbspace01,sbspace02),
```

```
commentary05 IN (sbspace03);
```

如果 INSERT 或 MERGE 操作向此示例中的表中添加六个新的行，

- 三个 **image04** BLOB 对象将存储于 **sbspace01** ，
- 其它三个 **image04** BLOB 对象将存储于 **sbspace02** ，
- 所有的六个 **commentary05** CLOB 对象将存储于 **sbspace03** 。

当您跨不同的 sbspace 传递智能大对象时，可以用较小的 sbspace 操作。如果限制了 sbspace 的大小，则备份和归档操作会加快执行，有关 PUT 子句的其它示例，请参阅完全备份的备用方式。

**指定 sbspace 特征**

以下存储选项可用于存储 BLOB 和 CLOB 数据：

| 选项 | 作用 |
|---|---|
| EXTENT SIZE | 指定智能大对象中能最少存储多少千字节。数据库服务器可能将指定的 ***kilobytes*** 值集中起来，从而使范围是 sbspace 页面大小的若干倍。 |
| HIGH INTEG | 此高级数据完整性选项生成包含页眉和页尾的用户数据页以检测不完整的写和数据损坏。这是缺省的数据完整性行为。 |
| MODERATE INTEG | 此数据完整性选项生成包含页头但不包含页尾的用户数据页面。该选项可将页头和页尾进行比较以检测不完整的写和数据损坏。 |
| KEEP ACCESS TIME | 对于智能大对象元数据中的记录，智能大对象是最近一次对其进行读或写操作的系统时间。 |
| NO KEEP ACCESS TIME | 不记录智能大对象最近一次进行读或写的系统时间。与 KEEP ACCESS TIME 选项相比，它能提供更好的性能并且是缺省的跟踪行为。 |
| LOG | 对于相应的智能大对象，与当前数据库日志一起记录日志的过程。该选项将会生成大量的日志流量并增加填满逻辑日志的风险。（另见 完全备份的备用方式。） |
| NO LOG | 关闭日志记录。该选项是缺省行为。 |

定义 sbspace 特征的关键字选项的逗号分隔的列表必须包含在括号中内，且紧随存储 BLOB 或 CLOB 列的 sbspace （或 sbspace 列表）的后面。在以下示例中， PUT 子句指定未日志记录的 **sbspace01** 和 **sbspace02** sbspaces 存储 **image04** 列的 BLOB 对象，它们与 **sbspace03** 具有不同的特征，**sbspace03** 是一个日志记录的 sbspace ，它存储 **commentary05** 列的 CLOB 对象：

```
CREATE TABLE tabw2sblobs
        (
        image04 BLOB,
        commentary05 CLOB
        ) PUT image04 IN (sbspace01,sbspace02) (KEEP ACCESS TIME, MODERATE INTEG),
        commentary05 IN (sbspace03) (EXTENT SIZE 30, LOG);
```

当您为智能大对象启动日志记录时，必须立即执行 0 级备份以便恢复并重新存储智能大对象。

syscolattribs 系统目录表包含 PUT 子句中每个 *sbspace* 和 *column* 组合的行：

- **syscolattribs.extentsize** 列存储 extent 大小，它基于 ***kilobytes*** 值。
- **syscolattribs.flags** 列存储关联到日志记录和存取时间状态的位图，和数据完整性设置。

如果用户定义的或复杂数据类型包含多个大对象，则指定的大对象存储选项将应用到该类型中的所有大对象。除非当大对象创建时重设了存储选项。

**重要：** PUT 子句不影响简单大对象数据类型（BYTE 和 TEXT）的存储。有关如何存储 BYTE 和 TEXT 数据的信息，请参阅大对象数据类型。

### *完全备份的备用方式*

您可以在最初开始加载智能大对象时关闭日志记录，一旦对象已经加载后就可以返回日志记录，从而取代完全日志记录。

使用 NO LOG 选项关闭日志记录。如果您使用 NO LOG ，可以稍后将智能大对象元数据恢复到没有结构不一致存在的状态。在大多数情况下，不存在事务不一致，但结构无法保证。

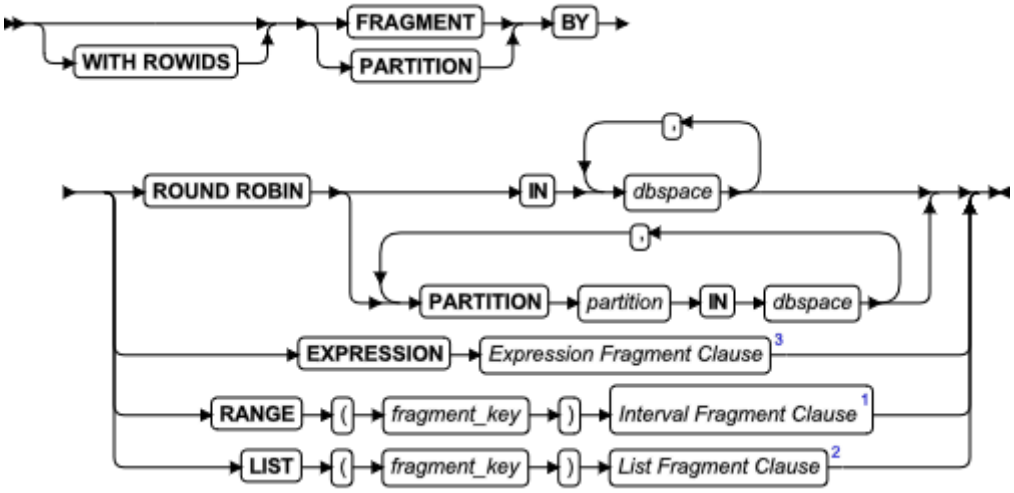
以下语句将创建 greek 表。表中的数据值被分片并保存到 dbs1 和 dbs2 dbspace 中。PUT 子句将 gamma 和 delta 列中的智能大对象数据分别指定给智能大对象 sb1 和 sb2 sbspace。eps 列的 TEXT 数据值被指定给 blb1 blobspace 。

```
CREATE TABLE greek
(alpha INTEGER,
 beta   VARCHAR(150),
 gamma CLOB,
 delta BLOB,
 eps    TEXT IN blb1)
   FRAGMENT BY EXPRESSION
   alpha <= 5 IN dbs1, alpha > 5 IN dbs2
   PUT gamma IN (sb1), delta IN (sb2);
```

## FRAGMENT BY 子句

使用 FRAGMENT BY 子句创建分片表并指定它的存储分布方案。PARTITION BY 关键字是 FRAGMENT BY 的同义词。

表的 FRAGMENT BY 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 应用分片存储策略的列 | 必须是表中的列 | 标识符 |
| *dbspace* | 存储表分片的 Dbspace | 最多可以指定 2,048 个 *dbspaces*。所有存储分片的 *dbspaces* 必须具有相同的页大小。 | 标识符 |
| *fragment_key* | 表列上的强制转型、列或函数表达式。这是在已分片的表上的表达式。 | 列只能来源于当前表。 | 表达式 |
| *partition* | 此处为分片声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |

当您分片表时，存储表分片的存储空间名跟在 IN 关键字的后面。

## *使用 WITH ROWIDS 选项*

未分片的表包含名为 rowid 的隐藏列，但是缺省情况下，分片表没有 rowid 列。可以使用 WITH ROWIDS 关键字向已分片表中添加 rowid 列。每行将自动分配一个唯一的 rowid 值，该行存在时这个值将一直保持不变，这样数据库服务器就可以使用该值来查找行的物理位置。每行需要额外的四个字节来存储 rowid 。

**重要：** 不推荐使用此功能。请使用主键作为存取方法，而不要使用 rowid 列。

无法对类型表使用 WITH ROWIDS 子句。

## *通过 ROUND ROBIN 分片*

在循环分布方案中，至少指定两个您希望放置分片的 dbspace 或分区。当记录被插入表中时，它们被放置在第一个可用的分片中。当您插入记录并按照分片总是保持大致相同的行数的方式分布行时，数据库服务器将在指定的分片中平衡负载。在该分布方案中，如果数据库服务器查找某行则它必须扫描所有的分片。

**重要：**

FRAGMENT BY ROUND ROBIN 子句重写表的自动定位和分片，当 AUTOLOCATE 配置参数或 AUTOLOCATE 环境变量设置成正整数时，启用此功能。

当启用自动定位和分片时，则数据库服务器会自动决定

- 表 extent 大小，
- 存储分片的 dbspaces ，
- 新表的 ROUND ROBIN 分布存储策略。

**大对象数据类型的循环分片存储**

对于包含 BYTE 或 TEXT 列的表中的简单大对象，您可以通过设置 PN_STAGEBLOB_THRESHOLD 配置参数来为插入 BYTE 和 TEXT 列保留空间。有关数据库服务器如何在循环分布分片中 stage 简单大对象的信息，请参阅 *GBase 8s 管理员参考手册* 中 PN_STAGEBLOB_THRESHOLD 的描述。

对于包含 BLOB 或 CLOB 列的表中的智能大对象，您可使用 PUT 子句在 sbspace 列表中指定循环分布存储。当您在 CREATE TABLE 语句（或 CREATE TEMP TABLE 语句或 ALTER TABLE 语句 ）中包含 PUT 子句时，您可以包含或不包含为同一表上其它列定义分布存储的 FRAGMENT BY 子句的选项。PUT 子句只对指定多个 sbspace 的智能大对象应用循环存储分布策略。有关更多信息和示例，请参阅 PUT 子句。

## *通过 EXPRESSION 分片*

在***基于表达式***的分布方案中，规则中的每个分片表达式都指定了一个存储空间。规则中的每个分片表达式将数据隔离起来并帮助数据库服务器查找这些行。

要通过表达式对表分片，请指定以下规则之一：

- 范围规则

范围规则指定使用范围的分片表达式来指定在分片中放置哪些行，如以下示例所示：

```
FRAGMENT BY EXPRESSION c1 < 100 IN dbsp1,
        c1 >= 100 AND c1 < 200 IN dbsp2, c1 >= 200 IN dbsp3;
```

- 仲裁规则

仲裁规则根据预先定义的 SQL 表达式来指定分片表达式，该表达式通常使用 OR 子句将数据分组，如以下示例所示：

```
FRAGMENT BY EXPRESSION
        zip_num = 95228 OR zip_num = 95443 IN dbsp2,
        zip_num = 91120 OR zip_num = 92310 IN dbsp4,
        REMAINDER IN dbsp5;
```

**警告：** 请参阅日志记录选项这一节中关于 DBCENTURY 环境变量和分片表达式中数据值的说明。

在具有 NLSCASE INSENSITIVE 属性的数据库中，对 NCHAR 和 NVARCHAR 数据的操作会忽略字符大小写，从而数据库服务将由相同序列字母组成的大小写变化的字符串视为重复的值。如果通过表达式分片的表的分片键是 NCHAR 或 NVARCHAR 列，则通过字符表达式定义的每个分片将存储符合定义分片的表达式的所有大小写不同的变量。例如，对于表达式 lname = 'Garcia'，其中 lname 是 NCHAR 或 NVARCHAR 类型的列，在该列中具有下列值的行将被存储在相同的分片中。 因为对于这些（并相似）字符串值字符大小写表达式是等价的：

'Garcia' 'garcia' 'GARCIA' 'GarCia' 'gARCia'

有关 NLSCASE INSENSITIVE 数据库的更多信息，请参阅 CREATE DATABASE 语句 、在 NLSCASE INSENSITIVE 数据库中重复的行和在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

**分片表达式中的用户定义的函数**

对于包含用户定义的数据类型的行，您可以使用比较条件或用户定义的函数来定义范围规则。在以下示例中，比较条件为包含不透明数据类型的 long1 列定义了范围规则：

```
FRAGMENT BY EXPRESSION
        long1 < '3001' IN dbsp1,
        long1 BETWEEN '3001' AND '6000' IN dbsp2,
        long1 > '6000' IN dbsp3;
```

隐式的、用户定义的强制转型将 3001 和 6000 转换为不透明类型。

此外，您还可以使用用户定义的函数为 long1 列的不透明数据类型定义范围规则：

```
FRAGMENT BY EXPRESSION
        (lessthan(long1,'3001')) IN dbsp1,
        (greaterthanorequal(long1,'3001') AND
        lessthanorequal(long1,'6000')) IN dbsp2,
        (greaterthan(long1,'6000')) IN dbsp3;
```

如前面的示例所示，显式的用户定义的函数需要在 IN 子句前的整个分片表达式周围加上括号。

可以用 SPL 或 C 或 Java™ 语言来编写分片表达式中用户定义的函数。这些函数必须满足四个要求：

- 必须能对 Boolean 值求值。
- 它们必须是不可变的。
- 它们必须驻留在相同数据库的同一表中。
- 它们不得生成 OUT 或 INOUT 参数。

有关如何为分片表达式创建 UDR 的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

**使用 REMAINDER 关键字**

使用 REMAINDER 关键字来指定存储超出指定表达式的有效值的存储空间。如果不指定 remainder ，并且在一行中插入或更新不符合任何分片定义的值，则数据库服务器将返回一个错误。

下列示例使用仲裁规则为 c1 列的特定值定义五个分片，以及另一个可以接受所有值的分片：

```
CREATE TABLE T1 (c1 INT) FRAGMENT BY EXPRESSION
        PARTITION PART_1 (c1 = 10) IN dbs1,
        PARTITION PART_2 (c1 = 20) IN dbs1,
        PARTITION PART_3 (c1 = 30) IN dbs1,
        PARTITION PART_4 (c1 = 40) IN dbs2,
        PARTITION PART_5 (c1 = 50) IN dbs2,
        PARTITION PART_6 REMAINDER IN dbs2;
```

此处，前面三个分片存储在 **dbs1** dbspace 的分区中，其它分片，包括 remainder ，都存储在 **dbs2** dbspace 的分区中。此示例需要有显式的分片名，因为每个 dbspace 都有多个分区。

### *通过 LIST 分片*

通过列表分片定义每个分片都基于分片键的离散值。

当分片键的值是在该类别集合内没有量化顺序的名义量表上的类别时，您可以使用此分片策略。当表包含分片键的有限值集合并且表上的查询具有分片键上的等式谓词时，通过列表分片很有帮助。例如，您可以按地理分片数据，它基于一个国家内省或洲的列表。每个分片中存储的行可以限制到单个分片键值，或者限制为表示分片键值的某个逻辑子集的值的列表，提供没有分片键值被两个或多个分片共享。

通过 LIST 分片还有助于逻辑上隔离数据。

通过 LIST 分片支持这三个功能：

- 表及其索引都可以通过 LIST 分片。
- 分片键可以是基于单列或多列的列表达式。
- 该列表可以可选地包含 remainder 分片。
- 该列表可以可选地包含只存储 NULL 值的 NULL 分片。

通过 LIST 分片（或在 CREATE INDEX 语句中分片索引 ）必须满足这些要求：

- 包含 NULL (或 IS NULL）的列表不能包含其它值。
- 分片键必须基于单行。
- 分片键必须是列表达式。该***常量表达式***可基于单列或多列。
- 列表不能包含重复的***常量表达式***值。每个值在 FRAGMENT BY LIST 子句中必须是唯一的。

在以下情景中在通过 LIST 分片的表上进行 Load 、INSERT 、MERGE 或 UPDATE 操作时，会失败：

- 行的分片键计算为 NULL ，但是 FRAGMENT BY LIST 子句定义的为非 NULL 值。
- 行的分片键符合对任何分片都是非***常量表达式***值的条件，但是没有定义 remainder 分片。

以下是通过 LIST 分片表的示例：

```
CREATE TABLE customer(id SERIAL, fname CHAR(32), lname CHAR(32), state CHAR(2),
phone CHAR(12))
        FRAGMENT BY LIST (state)
        PARTITION p0 VALUES ("KS", "IL") IN dbs0,
        PARTITION p1 VALUES ("CA", "OR") IN dbs1,
```

```
    PARTITION p2 VALUES ("NY", "MN") IN dbs2,
    PARTITION p3 VALUES (NULL) IN dbs3,
    PARTITION p4 REMAINDER IN dbs3;
```

在以上示例中，表在列 **state** 上分片，该列称为***分片键***或***分区键。***该分片键可以是列表达式：

```
FRAGMENT BY LIST (SUBSTR(phone, 1, 3))
```

该分片键表达式可以具有多列，如以下示例所示：

```
FRAGMENT BY LIST (fname[1,1] || lname[1,1])
```

该分片必须是不能重叠的，即在值列表中不允许重复值。例如，以下表达式列表对同一表或索引的分片无效，因为它们的 "KS" 表达式重叠：

```
PARTITION p0 VALUES ("KS", "IL") IN dbs0,
PARTITION p1 VALUES ("CA", "KS") IN dbs1,
PARTITION p0 VALUES ("KS", "OR", "NM") IN dbs0,
```

该列表值必须是常量字符。例如，标识符或 name 变量在以下列表中是不允许的：

```
PARTITION p0 VALUES (name, "KS", "IL") IN dbs0,
```

NULL 分片是分片键列包含具有 NULL 值的分片。不同于 FRAGMENT BY EXPRESSION 定义，您不能在相同的 LIST 分片定义中将 NULL 和 其它列表值组合。例如，下列 VALUES 列表无效：

```
PARTITION p0 VALUES ("KS", "IL", NULL) IN dbs0,
```

Remainder 分片是存储分片键值不符合显式定义分片的表达式列表中的任一表达式的行的分片。如果定义了 remainder 分片，它必须在定义列表分片策略的 FRAGMENT BY 或 PARTITION BY 子句中最后列出的分片。

**在 NLSCASE INSENSITIVE 数据库中的 LIST 分片**

在具有 NLSCASE INSENSITIVE 属性的数据库中，对 NCHAR 和 NVARCHAR 数据类型的列的操作会忽略字符大小写，因此数据库服务器将由相同序列字母组成的大小写变化的字符串视为重复的值。如果分片键是 NCHAR 或 NVARCHAR 列，则定义分片的字符表达式的列表还符合分片表中表达式的字符大小写变化的列值。

下列示例中，具有 'A' 和 'a' 的 **ad_state** 列值将存储在 **part0** 分片/分区中。

```
CREATE TABLE addr
    (
    ad_id NCHAR(100),
    ad_street NVARCHAR(255),
    ad_apt INT,
    ad_state NCHAR(2),
    ad_zip1 INT,
    ad_zip2 INT,
    checksum CHAR(48),
    PRIMARY KEY(ad_id)
    )
    FRAGMENT BY LIST(ad_state)
```

```
PARTITION part0 VALUES ('A', 'B', 'C', 'D') IN dbs1,
PARTITION part1 VALUES ('E', 'F', 'G', 'H') IN dbs2,
PARTITION part2 VALUES ('I', 'J', 'K', 'L') IN dbs3,
PARTITION part3 VALUES ('M', 'N', 'O', 'P') IN dbs4,
PARTITION part4 VALUES ('Q', 'R', 'S', 'T') IN dbs5,
PARTITION part5 REMAINDER IN dbs6 LOCK MODE ROW;
```

设计为返回只具有 'A' 或 'a'的行的查询可以在 **ad_state** 列上应用过滤，以致于只有第一个分片在查询执行计划中扫描：

```
SELECT * FROM addr WHERE ad_state = 'A';
```

以上区分大小写的查处排除了所有的分片，除了只按此分片扫描的 **part0** ，其包含 'A' 或 'a'的行被存储。

有关具有 NLSCASE INSENSITIVE 属性的数据库的更多信息，请参阅 CREATE DATABASE 语句、在 NLSCASE INSENSITIVE 数据库中重复的行和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## *通过 RANGE INTERVAL 分片（gbase 模式）*

可以使用此存储分布策略将分片将的量化值分片给其数值 或 DATE 或 DATETIME 范围内的非重叠间隔。

基于 RANGE INTERVAL 分片的分布存储通常将表分区为两种分片类型：

- ***范围分片*** ，它是您在 FRAGMENT BY 或 PARTITION BY 子句中显式定义的分片
- ***间隔分片***，它是数据库服务器在插入操作期间自动创建的分片。

要根据分片键（也称为***分区分片***）的范围内的间隔分片表或索引，您必须定义下列参数：

- 一个分片键表达式，它基于一个数字、DATE 或 DATETIME 列。
- 至少一个范围表达式。在特定范围内带有分片键值的行存储在那个分片中。
- 对于每一个范围表达式，至少有一个存储相应分片的 dbspace 列表。

您通过特定范围表达式的显式定义的分片称为***范围分片***。RANGE INTERVAL 分片的语法要求只有有一个基于范围表达式的分片。

对于系统生成的分片（称为***间隔分片***），它是数据库服务器自动创建地以存储分片键值超出当前分片列表上限的行，您可以指定这些额外参数：

- 每个间隔分片可以存储的在分片键值范围内的间隔大小。
- 存储间隔分片的 dbspace 列表。

如果指定了间隔大小但是 dbspace 列表为空，则间隔分片将存储在存储范围分片的同一的 dbspace 中。如果没有指定间隔大小，则禁用间隔分片的自动创建功能。在那种情况下，范围分片可以存储分片键值在指定范围内的行，但是该表无法存储有分片键值超出这些范围的行。

CREATE INDEX 语句也支持 RANGE INTERVAL 分片策略。如果一个表具有用相同的 FRAGMENT BY RANGE 语法定义的连接的索引，则当行超出现有已插入的间隔时，会类似地自动创建对应的索引分片（与新表分片具有一样的名称）。

通过范围分片分片的表或索引不支持 REMAINDER 分片，因为如果您定义所有了以上列出的参数，则数据库服务器自动创建新的间隔分片以存储那些已插入的分片键值超出现有分片范围的行。

对于不具有 NOT NULL 约束的表，您可以通过指定 VALUES IS NULL 为范围表达式来定义 NULL 分片。

当不知道增长的表中所有的可能分片键值，且 DBA 不想为还未存在的数据预分配分片时，RANGE INTERVAL 分片策略会很有用。

以下是通过范围间隔分片表的示例，它使用整数列作为分区键：

```
CREATE TABLE employee (id INTEGER, name CHAR(32), basepay DECIMAL (10,2),
        varpay DECIMAL (10,2), dept CHAR(2), hiredate DATE)
        FRAGMENT BY RANGE (id)
        INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
        PARTITION p0 VALUES IS NULL IN dbs0,
        PARTITION p1 VALUES < 200 IN dbs1,
        PARTITION p2 VALUES < 400 IN dbs2;
```

在此表中

- 间隔大小的值为 100，
- 该分片键是 **employee.id** 列的值，
- VALUES IS NULL 关键字定义 p0 作为存储没有 **id** 列值的行的表分片。

当 employee ID 超过 199 ，自动创建 100 间隔（指定分间隔大小）的分片。

如果带有 employee ID 405 的行被插入，则会创建新的间隔分片来容纳此行，新的分片拥有 **id** 列值在 >= 400 AND < 500 范围的行。

如果更改了行，且将 employee ID 更改为 821，则数据库服务器创建新的分片来容纳新的行。该分片拥有 **id** 列值在 >= 800 AND < 900 范围的行。

间隔分片创建于 STORE IN 子句定的 dbspace 中的循环分布方案中。如果忽略了该子句，则间隔分片将会在存储范围分片的 dbspace （之前例子中的 dbspace **dbs0** 、**dbs1** 和 **dbs2** ）中创建。如果为间隔分片指定的 dbspace 已满或关闭，则数据库服务器略过此 dbspace ，选择列表中下一个 dbspace 。

注意此间隔分片的范围表达式不能重叠，且不能有 remainder 分片。

范围间隔分片的分片键只能引用单列。例如，以下规范无效：

```
FRAGMENT BY RANGE (basepay + varpay)
```

分片键可以是列表达式，如下例规范所示：

```
FRAGMENT BY RANGE ((ROUND(basepay))
```

创建分片不需要互斥锁。分片键表达式必须评估为数字、DATE 或 DATETIME 数据类型。例如，您可以为每一个月创建一个分片，或为每百万客户记录创建一个分片。间隔大小规范（在 INTERVAL 关键字之后 ）必须是

- 一个数字数据类型的非零整数常量表达式（对于数字分片键），

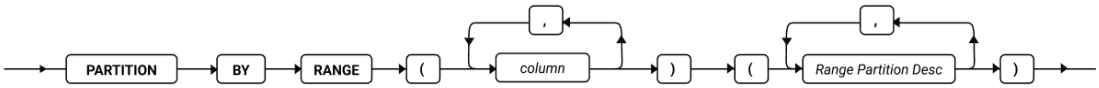
- 或 INTERVAL 数据类型（对于 DATE 或 DATETIME 分片键）。

SQL 的 ALTER FRAGMENT 语句可以在非分片表或索引上应用 RANGE INTERVAL 存储分布，如 INIT 子句 中描述的那样。该语句还会修改现有 RANGE INTERVAL 策略的功能。有关更多信息和示例，请参阅 MODIFY 子句和带有区间分片的 MODIFY 子句的示例。

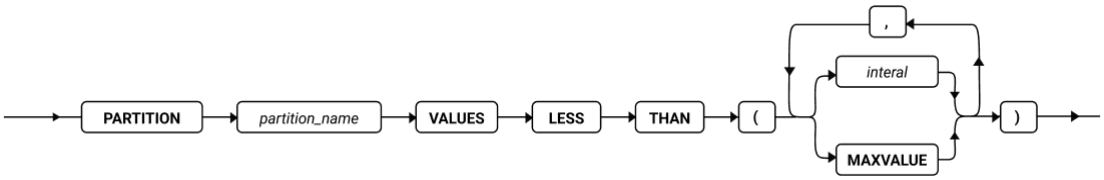
## *通过 RANGE 分片（oracle 模式）*

350 版本更新之后原 RANGE INTERVAL 语法在只能在 gbase 模式下使用了，oracle 模式使用新的 range 分区语法，目前暂不支持 INTERVAL 功能

语法图如下：




Range Partition Desc 子句:




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***column*** | 分区字段 | 表内的字段名 | 标识符 |
| ***partition_name*** | 分区名 | 分区名在表中唯一 | 标识符 |
| ***interal*** | 分区界限 | 必须是与分片键表达式的数据类型符合的值 | 标识符 |

用法:

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

目前暂不支持 INTERVAL 功能。

整个表分区范围必须按照从小到大创建。

分区字段可以是一个也可以是多个。

如果设置了多个分区字段,某个分区其中一个字段设置了 maxvalue，那么这个分区的所有字段需要同时设置为 maxvalue。

分区字段不能使用表达式。

示例如下:

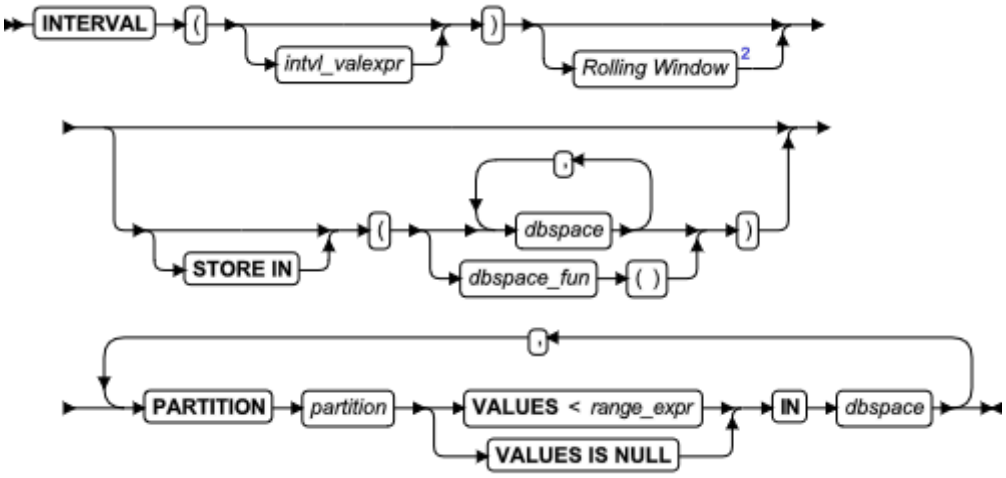
```
create table tab1
 (cust_id integer,name char(128))
  partition by range(cust_id)
   (
     partition p0 values less than (100),
      partition p1 values less than (200),
      partition p2 values less than (maxvalue)
);
```

## Interval fragment 子句

使用 Interval Fragment 子句存储通过一个或多个计算为数字或 INTERVAL 数据类型的范围表达式定义的行。在您为一个分片指定至少一个非 NULL 范围后，数据库服务器在插入分片键值超出现有分片的范围的行的 DML 操作中自动创建新的间隔分片。

CREATE TABLE 语句的 Interval Fragment 子句支持以下语法：

Interval Fragment 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *dbspace* | 存储分片的 dbspace 名称 | 最多只能指定 2,048 个 *dbspaces* 。所有存储分片的 *dbspaces* 必须具有相同的页面大小。 | 标识符 |

| | | | |
|---|---|---|---|
| *dbspace_fun* | 返回 dbspace 的名称的 UDF | 当数据库服务器调用 UDR 为新分片分配存储时，用户定义的函数以及返回的 dbspace 必须存在 | CREATE FUNCTION 语句 |
| *intvl_valexpr* | 在分片键范围捏定义间隔大小的间隔值表达式 | 必须是与分片键表达式的数据类型符合的求值为数字或 INTERVAL 值的常量字符表达式 | 标识符 |
| *partition* | 此处为范围分片声明的名称 | 在同一表的分片名称中必须是唯一的。如果表和它的索引使用同一范围间隔分片策略，则每个索引分片必须具有与对应表分片相同的名称。 | 标识符 |
| *range_expr* | 定义分片中分片键的上限的常量表达式 | 必须是与分片键表达式的数据类型符合的求值为数字或 INTERVAL 值的常量字符表达式 | 常量表达式 |

**用法**

Interval Fragment 子句定义紧跟在 FRAGMENT BY 子句的 FRAGMENT BY RANGE 关键字之后指定的分片键表达式范围内的一个或多个非重叠的间隔。PARTITION BY RANGE 关键字是 FRAGMENT BY RANGE 关键字的同义词。如果向表中插入或更改符合为此分片定义的范围的 DML 操作，那么数据库服务器将新的或更改的行存储在那个分片中。

必须定义指示一个分片以存储具有非 NULL 分片键值的行。对于分片键值为 NULL 的行不需要定义分片。但是如果没有为 NULL 值定义分片，则对表会产生带有 NULL 分片键值的 DML 操作会失败。

Interval Fragment 子句无法定义 remainder 分片。

对于由包含 Rolling Window 子句的 Interval Fragment 子句定义的分布存储表，数据库服务器在创建了超出用户定义的上限的足够多新的间隔分片时会自动拆离表分片。您可以使用 Rolling Window 子句定义以下一个或所有的限制：

- 系统生成的间隔分片的当前数量，
- 或者为表以及它的索引分配的存储空间的总大小。

有关 Rolling Window 表的行为和语法的信息，请参阅上面语法图中的链接。

**INTERVAL 大小规范**

INTERVAL 关键字之后的 ***intvl_valexpr*** 表达式定义在分片键值范围内的分片的大小。

*intvl_valexpr* 表达式的数据类型取决于 RANGE 关键字之后的分片键列的数据类型：

- 如果分片键是数字数据类型，则 ***intvl_valexpr*** 表达式必须计算为一个数字值。数字 ***intvl_valexpr*** 表达式必须是大于零没有小数部分的常量表达式。
- 如果分片键是 DATE 或 DATETIME 数据类型，则 ***intvl_valexpr*** 表达式必须计算为 INTERVAL 值。INTERVAL ***intvl_valexpr*** 表达式必须是大于零的常量表达式。

*intvl_valexpr* 表达式的最小值取决于分片键表达式的数据类型。

- 如果分片键是 DATETIME 列则最小值是一秒
- 如果分片键是 DATE 列则最小值是一天
- 如果分片键是数值列则最小值是 1 。

您可以使用字符数值或者字符 INTERVAL 值作为 ***intvl_valexpr*** 表达式。还可以使用内置的 NUMTODSINTERVAL 、NUMTOYMINTERVAL 、TO_DSINTERVAL 或 TO_YMINTERVAL 函数指定 ***intvl_valexpr*** 表达式。有关这些函数的语法和在 Interval Fragment 子句中使用它们的示例，请参阅 TO_YMINTERVAL 函数 和 TO_DSINTERVAL 函数。

如果您没有指定 ***intvl_valexpr*** 表达式，则禁用间隔分片的自动创建功能，但是 INTERVAL 关键字之后仍需要空的括号来避免语法错误。

**STORE IN 规范**

当 DML 操作存储分片键值超出现有分片范围的行时，紧随 STORE IN 关键字之后的 dbspace（或以逗号分隔的 dbspace 名称列表）为服务器自动创建的新的间隔分片标识存储空间。如果您指定多个 dbspace ，则数据库服务器在 STORE IN 子句中指定的 dbspace 的循环分布方案中创建间隔分片，并为此新的分片声明系统生成的名称。

当创建表或索引时，不需要显示 STORE IN 子句中的 dbspace 。您可以在创建表或索引后向系统中添加 dbspace。所有 Interval Fragment 子句中引用的 dbspace 必须具有相同的页大小。

如果您省略了 STORE IN 子句， 且表需要存储超出现有分片和范围分片的行时，数据库服务器自动在范围表达式分片的 PARTITION 规范列表的 dbspace 的循环分布方案中创建新的间隔分片。

STORE IN 子句可以可选地指定一个返回现有 dbspace 名称的用户定义的函数，而不是文本 dbspace 标识符列表。您为此 UDF 声明的标识符是任意的。

此函数接受四个参数：

- 表的所有者，CHAR(32) 数据类型
- 表的名称，CHAR(255) 数据类型
- 与分片键同一数据类型的分片值，或者可以隐式强制转型为那个数据类型的兼容类型
- 重新尝试标记，INT 数据类型。

重要：

然而，当您在 STORE IN 子句中引用任意 UDR 时，不要给 UDF 指定任何参数。它所需要的仅是 UDF 名称，紧随在一对空括号之后，像以上语法图中所指示的那样。数据库服务器自动在调用时自动提供参数。

此处有一个可返回 dbspace 名称的 UDF 示例，CREATE TABLE 语句定义了按范围分片分片的表，并在 STORE IN 子句中调用了该函数：

```
CREATE FUNCTION mydbname
    (
    owner CHAR(255),
    table CHAR(255),
    value DATE,          -- Data type must match or must be compatible
                         -- with the data type of the fragment key
    retry INTEGER
    )
    RETURNING CHAR(255)
    IF (retry > 0)
    THEN
        RETURN NULL;    -- This UDF does not handle retries: if the first call
                        -- fails, an invalid dbspace is returned, and the DML
                        -- statement that requires a new fragment also fails.
    END IF;
    IF (MONTH(value) < 7)
    THEN
        RETURN "dbs1";
    ELSE
        RETURN "dbs2";
    END IF;
END FUNCTION;

CREATE TABLE orders
       (
       order_num                SERIAL(1001),
       order_date               DATE,
       customer_num             INTEGER NOT NULL,
       ship_instruct            CHAR(40),
       backlog                  CHAR(1),
       po_num                   CHAR(10),
       ship_date                DATE,
       ship_weight              DECIMAL(8,2),
       ship_charge              MONEY(6),
       paid_date                DATE
       )
PARTITION BY RANGE(order_date) INTERVAL(1 UNITS MONTH)
STORE IN (mydbname())
PARTITION prv_partition VALUES < DATE("01/01/2010") IN mydbs;
```

当需要为此表创建新的间隔分片时，数据库服务器定义指定的函数。如果在返回的 dbspace 中尝试创建分片失败，则用 retry 标记设置第二次调用相同的函数，以致于会返回一个不同的现有 dbspace 名称。第二次尝试失败后，执行的 DML 语句返回错误。（如果第一次尝试失败，则以上示例中的 UDF 不会进行第二次尝试，但是返回 NULL ，一个无效的 dbspace 名称。）

### 用户定义的范围分片

在 Interval Fragment 子句中必须定义至少一个范围分片。声明每个分片需要这些元素：

- PARTITION 关键字，后面跟随您为此分片生成的名称。表的其它分片不能具有相同的名称。
- VALUES 值，后跟具有以下其中其中之一格式的 Boolean 表达式：
  - 小于（ < ）关系运算符和范围表达式，它定义可以存储在分片中分片键值的上限
  - IS NULL 运算符。如果分片键采用 NULL 值，您可以使用此运算符定义 NULL 分片，该分片只存储将 NULL 作为其分片键值的行。

  只能定义一个 IS NULL 运算符分片。NULL 分片并不是必需的，但是如果 NULL 分片不存在，而用户尝试插入分片键列为 NULL 的行时，数据库服务器返回错误。
- IN 关键字，后跟存储该分片的 dbspace 的名称。它可以还是 STORE IN 规范引用的 dbspace ，或者不在 STORE IN 列表中包含的 dbspace 。

如果范围分片没有按升序顺序定义，则数据库服务器按升序顺序存储它们，因此在第一个初始位置的分片具有最小的上限范围。

在相同的 Interval Fragment 子句中的两个分片不能具有相同的上限。 PARTITION 规范中定义的范围分片不能重叠。如果（***intvl_valexpr***）大小规范在 INTERVAL 关键字之后，那么如果定义连续范围分片的范围表达式之间的差异与 INTERVAL 大小规范不同，则数据库服务器发出错误。

NULL 分片并非必需，但如果用户尝试擦汗荣分片键值是 NULL 的行，而不存在 NULL 分片时，数据库服务器返回错误。

重要：

dbschema -ss 命令的输出显示了通过范围间隔分布方案分片的表的结构，该方案只返回用户在 CREATE TABLE 或 ALTER FRAGMENT 语句中的定义的范围分片。

此情况同样对于 dbexport -ss 命令的输出同样为真。

### 系统生成的间隔分片

当您使用 Interval Fragment 子句定义表或索引的范围分片时，不需要知道分片键值的全部范围。当插入一个不适合范围分片或间隔分片的行是，数据库服务器自动创建新的间隔分片来存储该行（基于间隔 ***intvl_valexp***，不用 DBA 干预）。

表或索引的间隔分区的系统生成的名称是 sys_*evalpos*，*evalpos* 是系统目录中的此分片 sysfragments.evalpos 条目。如果表和它的索引使用相同的范围间隔分片策略，每个系统生成的索引分片将具有与该表的系统生成的分片相同的标识符。

这些自动生成的分片对应于包括新数据值的分片键范围的部分。如果在两个连续分片之间大于 ***intvl_valexp*** 范围的一部分不包含行，则间隙可以自动生成间隔分片。但是，在您在 Interval Fragment 子句中显式定义的分片之间是不允许间隙的。

如果未指定 ***intvl_valexp*** 表达式，则在 Interval Fragment 子句中显式定义的范围分片可用于存储在其范围间隔内具有对应分片键值的行，以及在禁用 ALTER FRAGMENT 以及之前生成的任何现有间隔分片自动创建间隔分片。然而，在这两种情况中，禁用自动创建新分片。如果用户尝试插入分片键值不在现有分片范围内的行，则数据库服务器发出错误 -772，并插入失败。

如上所述，dbschema -ss 和 dbexport -ss 命令显示按范围间隔分片的表的模式的输出仅包括用户定义的范围分片。在输出显示中不会显示系统生成的间隔分片。

但是，当从 dbexport 数据文件加载数据记录时，数据库服务器自动创建额外的间隔分片，

- 基于范围分片和间隔过渡分片，
- 和插入的行中的分片键值，
- 以及 Interval Fragment 子句的其它存储规范，注册于系统目录中。

**范围间隔分片的示例**

以下示例使用 INT 列 cust_id 的值作为数字分片键，并定义了四个范围分片。数据库服务器将为 cust_id 值超过 7999999 的插入行创建间隔大小是 1000000 的间隔发分片：

```
CREATE TABLE customer (cust_id INT, name CHAR (128), street CHAR (1024),
    state CHAR (2), zipcode CHAR (5), phone CHAR (12))
    FRAGMENT BY RANGE (cust_id)
    INTERVAL (1000000) STORE IN (dbs2, dbs1)
        PARTITION p0 VALUES < 2000000 IN dbs1,
        PARTITION p1 VALUES < 4000000 IN dbs1,
        PARTITION p2 VALUES < 6000000 IN dbs2,
        PARTITION p3 VALUES < 8000000 IN dbs3;
```

在以下 DATETIME 分片键示例中，如果 dt1 列的值超过 VALUES 子句指定的范围分片的界限，则将在 25 年间隔中在 2005 年之后的 dbs1 dbspace 中创建间隔分片：

```
CREATE TABLE t1 (c1 int, d1 date, dt1 DATETIME YEAR TO FRACTION)
    FRAGMENT BY RANGE (dt1) INTERVAL (INTERVAL(25) YEAR(2) TO YEAR)
        PARTITION p1 VALUES <
        DATETIME(2006-01-01 00:00:00.00000) YEAR TO FRACTION(5) IN dbs1;
```

在下一个示例中，DATE 列 order_date 的值是分片键，并且定义了四个范围分片，包括 order_date 具有 NULL 值的行的 p4。对于 order_date 的年值晚于 2007 的插入行，将在 01/01/2008 之后的 1 个月内自动创建间隔分片，并在 dbs1 、dbs2 和 dbs3 dbspace 中创建连续分片：

```
CREATE TABLE orders (order_id INT, cust_id INT,
                        order_date DATE, order_desc CHAR (1024))
    FRAGMENT BY RANGE (order_date)
    INTERVAL (NUMTOYMINTERVAL (1,'MONTH')) STORE IN (dbs1, dbs2, dbs3)
        PARTITION p0 VALUES < DATE ('01/01/2005') IN dbs1,
        PARTITION p1 VALUES < DATE ('01/01/2006') IN dbs1,
        PARTITION p2 VALUES < DATE ('01/01/2007') IN dbs2,
        PARTITION p3 VALUES < DATE ('01/01/2008') IN dbs3,
        PARTITION p4 VALUES IS NULL in dbs3;
```

下一个 DATE 分片键的示例与先前的示例类似，但是此处的间隔大小指定为 1.5 年。在 12/31/2009 之后，对于 order_date 值，将以 18 个月（1.5 年）的间隔创建间隔分片：

```
CREATE TABLE orders1 (order_id INT, cust_id INT, order_date DATE,
                      order_desc CHAR (1024))
  FRAGMENT BY RANGE (order_date)
  INTERVAL (NUMTOYMINTERVAL (1.5,'YEAR')) 000STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < DATE ('01/01/2004') IN dbs1,
      PARTITION p1 VALUES < DATE ('01/01/2006') IN dbs1,
      PARTITION p2 VALUES < DATE ('01/01/2008') IN dbs2,
      PARTITION p3 VALUES < DATE ('01/01/2010') IN dbs3;
```

如果 order_date 值丢失，则不能将行插入 orders1 表。因为没有定义非 NULL 分片。有关向现有使用范围间隔分片的表中添加 NULL 分片的语法，请参阅 ALTER FRAGMENT 语句的 ADD 子 r 句主题。

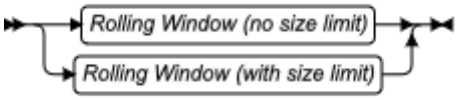
- Rolling Window 子句
- 使用 Rolling Window 子句来定义表或它的索引的范围分片分布存储策略，并定义*清除策略*以拆离过剩的分片。就像其它具有范围间隔分片的表一样，每个 *rolling window table* 的新的间隔分片是通过数据库服务器自动创建地，以用来存储超出当前分片范围的分片键值的新行。当分片的集合超出清除策略针对分片的数量或针对所分配的存储大小定义的用户定义的 "window" 之后，数据库服务器标识并拆离来自数据库中所有的滚动窗口表的过剩分片。缺省情况下，它们的清除策略作为调度程序的日常任务而强制执行。

## *Rolling Window 子句*

使用 Rolling Window 子句来定义表或它的索引的范围分片分布存储策略，并定义*清除策略*以拆离过剩的分片。就像其它具有范围间隔分片的表一样，每个 *rolling window table* 的新的间隔分片是通过数据库服务器自动创建地，以用来存储超出当前分片范围的分片键值的新行。当分片的集合超出清除策略针对分片的数量或针对所分配的存储大小定义的用户定义的 "window" 之后，数据库服务器标识并拆离来自数据库中所有的滚动窗口表的过剩分片。缺省情况下，它们的清除策略作为调度程序的日常任务而强制执行。

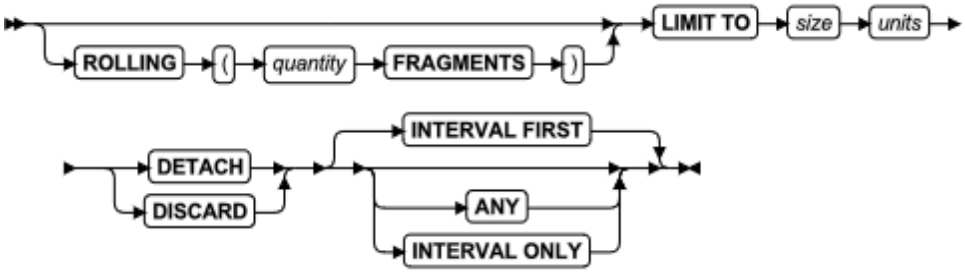
CREATE TABLE 的 Rolling Window 子句支持下列语法：

Rolling Window 子句




Rolling Window (no size limit)


Rolling Window (with size limit)




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *quantity* | 滚动间隔分片的最大数量 | 必须是大于零的整数。用户定义的范围分片不包含在此限制内。 | 字符整数 |
| *size* | 表的总存储大小的上限 | 必须大于零 | 字符整数 |
| *units* | 表的总质量存储的缩写单位 | 必须是 K 、KB 、KiB 、M 、MB 、MiB 、G 、GB 、GiB 、T 、TB 、TiB （或者这些字符的小写）。尾随任何字符都会导致语法错误。 | 不带引号的字符串 |

**用法**

INTERVAL 分片子句可启用新分片基于间隔分片表达式的自动创建功能。不像一般的范围间隔分布存储，它创建新的分片，但不提供对随着时间增长表的大小的管理，Rolling Window 选项定义间隔分片的当前数量的上限，或为表和其索引分配的总存储的大小或者这两者限制。在超过这些限制后，数据库服务器自动归档或销毁 Rolling Window 子句标识为超出的分片，并用基于 Rolling Window 子句定义的*清除策略*的新的间隔分片替换它们。

范围间隔分布存储策略包含 Rolling Window 子句的表称为 *rolling window tables*。数据库服务器为 Rolling Window 表创建的间隔分片称为 ***滚动分片*** 。

与启用 AUTOLOCATE 配置参数或会话环境设置创建的表的循环分片不同，本子句定义的滚动分片的动态 "window" 可以支持查询中的分片消除，其中 numeric 、DATE 和 DATETIME 表达式中的分片键值与查询谓词相关。滚动窗口表的另一个效率是当计划程序通过运行 purge_tables 任务强制执行清除策略时自动归档或销毁过多的碎片。

Rolling Window 子句定义自动从表中删除现有分片的以下任一条件或两个条件：

- ROLLING FRAGMENTS 关键字指定了表的间隔分片数量可以同时存在的限制。
- LIMIT TO 关键字为分配给表及其索引的总存储空间指定了限制。

当超过其中之一的限制时，超出时间间隔的分片将被数据库服务器自动销毁或拆离，如分别使用 DISCARD 或 DETACH 关键字指定。这些规范定义了表的清除策略。此策略通过定义自动删除现有分片的标准以及自动替换这些分片以插入新数据记录的新空分片来限制表可以存储多少数据。

**滚动窗口表的清除策略**

Rolling Window 子句为此表定义清除策略。 此清除策略通过定义自动拆离表分片的条件，在表达到滚动分片数量的用户指定的限制或分配的存储的总大小后，限制表可以存储多少数据。当启用清除策略时，数据库服务器自动用新的空的分片替换已拆离的分片以插入新的数据记录。

当达到清除策略的限制后，拆离分片的数量取决于定义此限制的关键字和定义此清除策略操作的关键字：

- 对于使用 ROLLING INTERVALS 关键字指定分片数量限制，只会考虑间隔分片。它们按最低分片键值的顺序拆离，如系统目录中分片的 sysfragments.evalpos 值所示。

  此选择拆离间隔分片的行为等价于 INTERVAL FIRST 关键字使用 LIMIT TO 选项指定的行为。然而，ROLLING INTERVALS 选项不支持显式 INTERVAL FIRST 关键字，和任何在 DETACH 或 DISCARD 清除规范之后随后的 LIMIT TO 选项关键字。

  ROLLING INTERVALS 选项还不提供拆离范围分片，因为 DETACH 或 DISCARD 清除之后，没有保留的范围分片。出于此原因，包含范围分片作为 ROLLING INTERVALS 选项（Rolling Window 子句语法也不支持）不会减少驻留在表中的分片数量。

- 对于使用 LIMIT TO 关键字指定已分配的存储大小限制，三个关键字选项可以指示拆离哪些分片：
  - 如果 ANY 关键字紧随在 DETACH 或 DISCARD 关键字之后，则会拆离范围或间隔分片，从 **sysfragments.evalpos** 值最低的分片开始。清除策略指定 ANY 为拆离的分片时可以减少当前已分配存储的大小，但是如上所述，拆离范围分片不会减少分片的总数。
  - 如果指定 INTERVAL ONLY 关键字，那么只拆离间隔分片，也是从 **sysfragments.evalpos** 值最低的分片开始。

    如果不存在间隔分片，则数据库服务器不会满足 LIMIT 子句的限制。如果该项发生在现有的滚动窗口表中，您必须考虑使用 ALTER FRAGMENT MODIFY INTERVAL 语句更改清除策略，以致于范围分片可以被拆离。可以通过将 INTERVAL ONLY 关键字替换为 ANY or INTERVAL FIRST 关键字来实现。反之如果您的存储策略支持较大的大小限制，则您可以使用 ALTER FRAGMENT 添加 LIMIT TO ***size*** 值。
  - 如果 INTERVAL FIRST 关键字紧跟着 DETACH 或 DISCARD 关键字，则数据库服务器首先拆离间隔分片，从最低的 **sysfragments.evalpos** 值开始，直到满足分配的存储大小需求。

  如果 Rolling Window 子句包含 LIMIT TO 关键字，但是没有上面的选项中的分片被拆离，那么缺省情况下 INTERVAL FIRST 策略决定拆离哪个分片。

  如果已经拆离了所有的间隔分片，而还没有填满存储大小限制，则作为安全措施，数据库服务器拆离范围分片，从最低值开始。在任何情况下，当范围分片被拆离或丢弃时，它们被新的空的分片代替以存储相同范围的值，因此该表的结构被保留。

**对已清除的分片中的数据的处置**

Rolling Window 子句提供两个关键字选项，DETACH 和 DISCARD ，用于自动主力滚动窗口表的已拆离的分片。这里对此关键字的选择没有缺省值。如果 Rolling Window 子句没有包含 DETACH 或 DISCARD 关键字则数据库服务器返回错误。

- 使用 DETACH 将分片与数据库服务器自动创建的独立的表建立连接，且表并标识符是这种格式：

  < original_table_name >_< lower value >_< higher value >

  此处的 ***lower_value*** 和 ***higher_value*** 是该分片被拆离之间间隔范围的最小值和最大值。

  如果表的名称已经存在，则会在 *higher_value* 之后附加一个数字计数，第一个附加表从 _1 开始：

  < original_table_name >_< lower value >_< higher value >_1

  等等等等，用 _2 附加到下一个表名称（或者如果附加 _2 不能产生一个唯一的表名称，则附加更多的整数）。

- 使用 DISCARD 销毁已拆离的分片。

  DISCARD 关键字指定删除已经成功拆离的分片，以致于当执行清除策略时，不需要的数据记录会及时移除。通过这种方法，过渡分片的数量或滚动窗口表的存储空间总量会约束到规定值。

这些配置选项旨在自动化由范围间隔分片的表的空间管理，以致于不需要的数据记录会及时移除，且存储空间被包含到规定量。丢弃数据的另一种方法是拆离分片。这提供了从不正确指定的清除策略恢复的机会，并允许将清除的分片（或者它们的数据被移动）附加到归档。

**强制执行清除策略**

在具有有限存储的数据库中，插入新行的 DML 或加载操作（包括插入超出现有分片的范围行）可能导致分配的存储大小或超过 Rolling Window 子句为一个或多个滚动窗口表指定的限制的间隔分片的数量。

但是，在它的界限超出的那一刻，不会立即执行 Rolling window 表的清除策略。

清除策略被设计为在移除和处理滚动窗口表表的分片的所需的 ALTER FRAGMENT DETACH 和 ALTER FRAGMENT ATTACH 操作不可能与并发用户的访问尝试冲突时作为调度任务每天强制执行。缺省情况下，清除策略会在每天本地时间 00:45 执行。有关更多信息，请参阅 *GBase 8s 管理员指南* 中 Scheduler 的内置 purge_tables 任务的描述。

清除策略也可以通过运行 syspurge( ) 系统函数来强制执行。在 DBA 调用此函数之后，数据库服务器检测系统目录，并标识清除策略已经超出的滚动窗口表。然后数据库服务器按照清除策略的指定丢弃或拆离、限定滚动分片直到满足清除策略，或者直到没有更多的滚动分片能移除。syspurge( ) 函数不需要参数，但是接受启用联机日志诊断的可选参数。

**更改、删除或添加清除策略**

可以使用 ALTER FRAGMENT 语句更改或删除滚动窗口表的清除策略。或将创建时具有其它存储选项的表更改为滚动窗口表。例如，简单地通过添加清除策略， ALTER FRAGMENT 语句的 Rolling Window 子句可将使用简单范围间隔分片的表更改为滚动窗口表。

CREATE TABLE 语句的 Rolling Window 子句支持 ALTER FRAGMENT ON TABLE . . . MODIFY INTERVAL 语句中的 Rolling Window 子句语法的子集。

如果您不满意现有滚动窗口表的清除策略。则可以使用 ALTER FRAGMENT 语句以多种方法更改此策略，包括：

- 更改 ROLLING FRAGMENTS 或 LIMIT TO 规范，
- 替换清除策略的 DETACH 或 DISCARD 关键字
- 使用 DISABLE 关键字选项终止清除策略
- 通过 ENABLE 关键字重新启用一个已终止的清除策略
- 移除该清除策略和滚动窗口表的滚动分片

要将滚动分布存储策略更改为简单的范围间隔分片策略，您可以为此表运行 ALTER FRAGMENT MODIFY INTERVAL DROP ALL ROLLING 语句。如果您要保留这些数据，则在做此操作之前首先归档表的非空滚动间隔分片中的行。

**滚动窗口表的限制**

使用 ROLLING INTERVALS 或 LIMIT TO 关键字定义滚动窗口分片策略的表，和其清除策略具有以下限制：

- Rolling Window 子句为滚动窗口分片定义的清除策略需要数据库服务器在满足 DETACH 或 DISCARD 标准的分片上执行 ALTER FRAGMENT DETACH 操作。 具有被启用的外键约束引用主键的列的表或具有 ROWID 的表不允许 ALTER FRAGMENT DETACH 语句。出于此原因，CREATE TABLE 和 ALTER FRAGMENT MODIFY INTERVAL 语句不能在具有主键约束或 ROWID 影子列的表上定义分片清除策略。
- 任何定义在滚动窗口表上的索引都必须具有与滚动窗口表相同的存储分布。
- 只有具有 DBA 存取权限的用户才能调用实现拆离滚动分片的 DETACH 或 DISCARD 选项的例程。 具有 RESOURCE 存取权限的用户可以执行 syspurge( ) 函数，但是只能对您自己所拥有的表执行清除策略。
- 数据库服务器会静默地忽略高可用数据复制（HDR）集群环境中辅助服务器上任何 syspurge( ) 函数的调用。这是因为集群环境不会复制滚动窗口清除策略核心中 DETACH 和 DISCARD 选项触发的 ALTER FRAGMENT 更改。
- 同样，在 grid 环境中，不会执行复制表行清除策略。

**没有存储大小限制的滚动窗口表**

以下 CREATE TABLE 语句的示例定义了一个范围间隔分片存储策略，包括把 **p4** 作为 NULL 分片以存储 **order_date** 分片键列值为 NULL 的行。因为此分片键的范围内的间隔定义为一个月，且分片过渡值是 2014 的第一天，则当具有比 2013 年晚一年内插入 **order_date** 值的记录时，会生成第一个间隔分片。连续的间隔分片将会以循环的形式存储在 dbspaces **dbs1** 、**dbs2** 和 **dbs3** 中：

CREATE TABLE orders

```
    (order_id INT, cust_id INT,
     order_date DATE, order_desc CHAR (1024))
FRAGMENT BY RANGE (order_date)
INTERVAL (NUMTOYMINTERVAL (1,'MONTH'))
    ROLLING (3 FRAGMENTS) DETACH
    STORE IN (dbs1, dbs2, dbs3)
    PARTITION p0 VALUES < DATE ('01/01/2014') IN dbs1,
    PARTITION p4 VALUES IS NULL in dbs3;
```

在以上示例中，Rolling Window 子句将滚动间隔分片的最大数量设置为 3 。如果在 2014 年前三个月的每一个中添加行，则在该年的三月将生成三个滚动分片，因为每个新的间隔分片仅存储一个月的数据。如果在 4 月创建了第 4 个时间间隔分片，这将超过滚动分片上的清除策略限制。由于未指定存储大小限制，因此默认的 INTERVAL FIRST 标准将拆离四个滚动分片中其 evalpos 值最小的间隔分片。该分片将附加到另一个表中，而不是销毁，因为清除策略指定 DETACH，而不是 DISCARD。

**具有存储大小限制的滚动窗口**

对于 **employee** 表，以下范围间隔分布式存储策略使用 INTEGER 列 **emp_id** 中的值作为主键列，1000 是滚动间隔分片的此分片键范围内的间隔。在三个范围分片中，最后一个具有 20000 的间隔跃迁值，这意味着当插入 **emp_id** 值为 20002 或更大时，将生成第一个滚动间隔分片。滚动间隔分片将再次以循环方式存储在 dbspaces **dbs1** 、**dbs2** 和 **dbs3** 中：

```
CREATE TABLE employee
            (emp_id INTEGER, emp_name CHAR(64),
             ssn CHAR(12), basepay FLOAT, varpay FLOAT,
             dept_id SMALLINT, hire_date DATE)
    FRAGMENT BY RANGE(emp_id)
    INTERVAL(1000)
        ROLLING ( 10 FRAGMENTS )
        LIMIT TO 100000MiB DETACH ANY
        STORE IN (dbs1, dbs2, dbs3)
        PARTITION p1 VALUES < 5000 IN dbs0,
        PARTITION p2 VALUES < 10000 IN dbs0,
        PARTITION p3 VALUES < 20000 IN dbs4;
```

这里的 Rolling Window 子句将滚动间隔分片的最大数量设置为 10 。除非以下任一事件发生否则此清除策略将不会强制执行：

- 数据库服务器在将记录插入到三个 3 个分片中的任何分片范围之外之后创建了第 11 个滚动分片，或者在需要第 11 个间隔分片的时候， 创建 10 个滚动间隔分片。
- 数据库服务器为 **employee** 表及其索引分配的总存储空间超过 100000 兆字节。

如果在存储大小限制之前 10 滚动间隔的限制已经超出，则数据库服务器将会在 11 个滚动分片中拆离 **evalpos** 值最小的间隔分片。

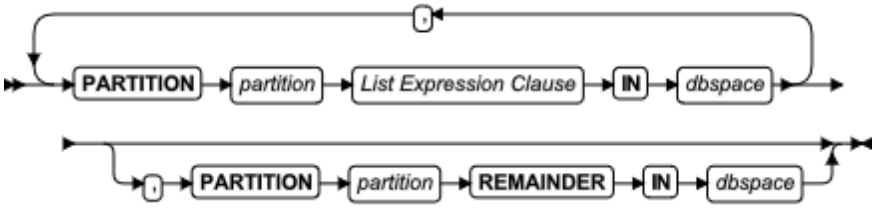
如果超过 100000 兆字节超过了数量的限制，则 DETACH ANY 选项运行数据库服务器选择任何范围分片或间隔分片以拆离。

在任一情况下，此分片将被附加到另一个表，而不是被销毁，因为清除策略指定了 DETACH ，而不是 DISCARD 。

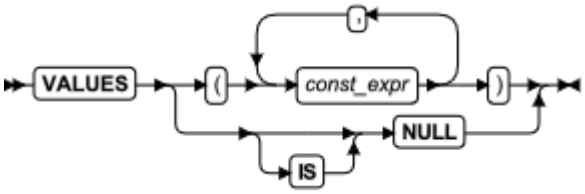
## 列表分片子句

使用 List Fragment 子句指定存储在同一分片中分片键值的列表。分配给每个分片的行必须符合定义此分片的分片键值（或以逗号分隔的分片键值列表中的一个值）。

List Fragment 子句




List Expression 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *const_expr* | 为要存储的分片定义分片键值列表的常量表达式 | 必须是带引号的字符串或字符值。列表中的每个值在同一对象的分片列表中必须是唯一的。 | 常量表达式 |
| *dbspace* | 存储分片的 dbspace | 最多只能指定 2,048 个 *dbspaces*。这些所有的 *dbspaces* 必须具有相同的页大小。 | 标识符 |
| *partition* | 此处为分片声明的名称 | 在同一对象的分片的名称中必须是唯一的。如果表和它的索引使用相同的列表分片存储策略，则每个索引分片必须具有与相应表分片一样的名称。 | 标识符 |

**在基于列表存储分布中的 REMAINDER 和 NULL 分片**

对于任何分片，可以可选地定义 REMAINDER 分片来存储不符合分片键值列表的行。

可以可选地定义 NULL 分片，以存储具有缺失分片键数据的行。做法为在该分片的列表表达式子句的 VALUES 关键字之后只指定 IS NULL 或 NULL。您不能在包含其它表达式的表达式列表中包含 NULL 或 IS NULL 。（在此上下文中，NULL 和 IS NULL 是关键字同义词。）

如果没有定义 NULL 分片，且有一个操作试图插入缺少分片键数据的行，则其结果取决于 REMAINDER 分片是否存在：

- 如果定义了 REMAINDER 分片，则行存储在 REMAINDER 分片中。
- 如果没有定义 REMAINDER 分片，则数据库服务器发出异常。

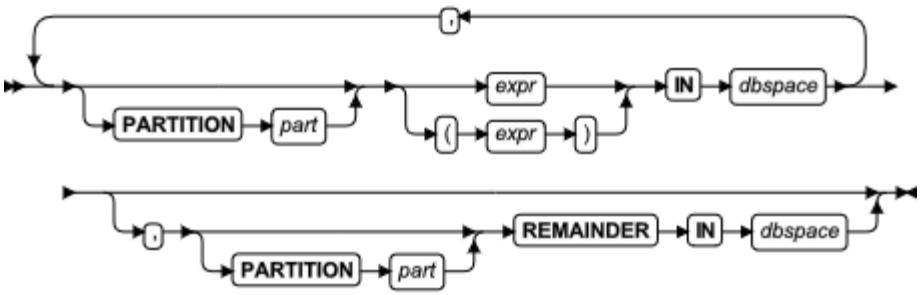
如果没有定义 REMAINDER 分片，而 INSERT 、UPDATE 、MERGE 或其它 DML 操作试图存储一个分片键与所有分片的列表值不符合的行，则数据库服务器发出异常。

当您为表或索引定义一个基于列表的分区方案时，该分片列表最多只能包含一个 NULL 分片，和一个 REMAINDER 分片。

如果 BY LIST 分区的表没有 NULL 或 REMAINDER 分片，但是您紧跟地决定需要这些分片或需要其中一个，可以通过使用 ALTER FRAGMENT 语句的 ADD 选项在分片列表中添加 NULL 分片或 REMAINDER 分片（或者两者都添加）。有关更多信息，请参阅 ADD 子句。

## 表达式分片子句

表达式分片子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *part* | 分片的名称 | 如果 *part* 作为该表的另一个分片存储在同一的 dbspace 中，则需要此元素。在相同表的分片名称中必须是唯一的。 | 标识符 |
| *dbspace* | 存储此表分片的 dbspace | 至多只能指定 2,048 个 *dbspaces*。所有存取分片的 *dbspaces* 必须具有相同的页大小。 | 标识符 |

| | | | |
|---|---|---|---|
| *expr* | 定义分片的表达式，它基于列值 | 必须返回 Boolean 值（true 或 false）。数据值必须来自该表中的一行。 | 表达式 |

## SECOND LEVEL PARTITION 子句

目前二级分区支持的分类有 8 种：

- 一级分区是范围分区的：范围-范围分区，范围-列表分区，范围-哈希分区
- 一级分区是列表分区的：列表-范围分区，列表-列表分区，列表-哈希分区
- 一级分区是哈希分区的：哈希-范围分区，哈希-列表分区

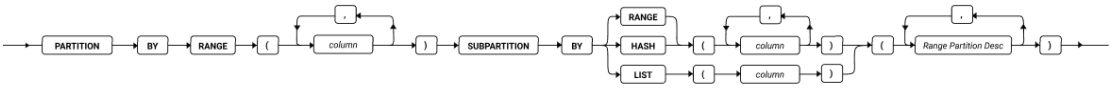
本版本二级分区表不支持 ALTER FRAGMENT 语句。

本版本二级分区表所有字段均不允许重命名列名操作。

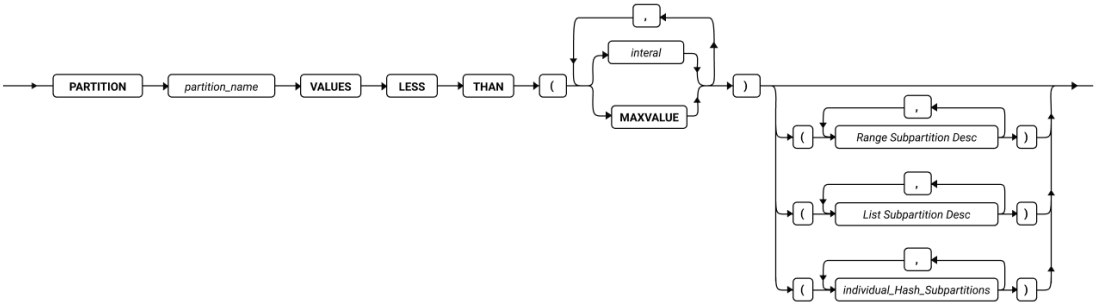
### *一级分区是范围分区的二级分区的创建*

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

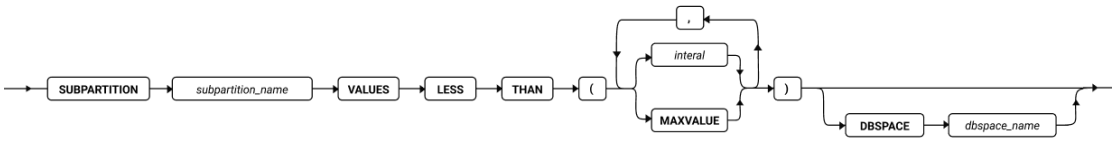
SECOND LEVEL PARTITION 子句语法图：



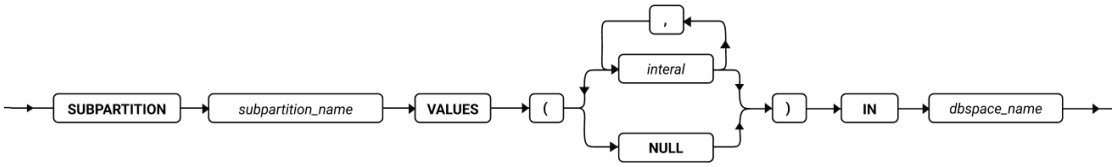

Range Partition Desc 语法图：



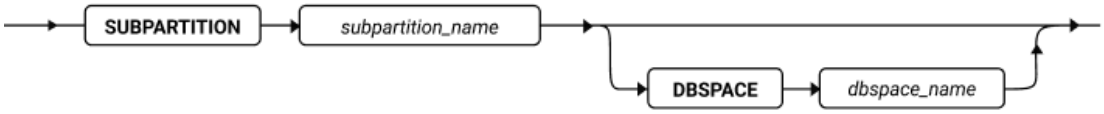

Range Subpartition Desc 语法图：




List Subpartition Desc 语法图：




individual_Hash_Subpartitions 语法图：




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***column*** | 应用分片存储策略的列 | 必须是表中的列 | 标识符 |
| ***dbspace_name*** | 存储表分片的 Dbspace | 最多可以指定 2,048 个 ***dbspaces***。所有存储分片的 ***dbspaces*** 必须具有相同的页大小。 | 标识符 |
| ***partition*** | 此处为分片声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |
| ***subpartition_name*** | 此处为二级分区声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |
| ***interal*** | 分区字段进行分区的值,如果是二级范围分区 values less than Interal 则是小于这个值<br>如果是二级列表分区则是等于这个值 | 必须是与分片键表达式的数据类型符合的值 | 标识符 |

**用法**

- 一级分区可以指定一个或多个字段做分区字段；
- 二级分区可以在范围，列表，哈希选择其中一种，不能多种二级分区混合使用；
- 二级分区为范围分区或者哈希分区可以指定一个或者多个字段做分区字段；

- 二级分区为列表分区只能指定一个字段当做分区字段；
- 一级分区不能指定 dbspace；
- 二级分区为范围分区或者哈希分区可以指定 dbspace，如果不明确指定则会使用数据库默认 dbspace，二级分区为列表分区必须明确指定使用的 dbspace；
- 一级分区和二级分区的范围分区如果定义多个字段做分区字段，如果其中一个字段设置了 maxvalue，那么这个分区的所有字段需要同时设置为 maxvalue；
- 每个一级范围分区下都可以有一个二级分区设置 maxvalue；
- 二级范围分区只能有一个子分区为 maxvalue；
- 整个表分区范围必须按照从小到大创建；
- 整个表的所有二级列表分区只能有一个 null；
- 目前布尔类型(BOOLEAN)、LVARCHAR 类型、虚拟列和 ROWNUM 不支持做分区字段；
- 表内的分区字段与非分区字段均不允许重命名列名操作(RENAME COLUMN) 。

**示例**

范围-范围分区表示例如下：

```
create table tab1(
    cust_id integer,
    name char(128),
    score integer
)
partition by range(cust_id)
subpartition by range(score)
(
    partition p0 values less than (100)
    (
        subpartition p0_1 values less than (50) dbspace dbs1_1,
        subpartition p0_max values less than (100) dbspace dbs1_2
    ) ,
    partition p1 values less than (200)
```

```
    (
        subpartition p1_1 values less than (150) dbspace dbs2_1,
        subpartition p1_max values less than (200) dbspace dbs2_1
    )
);
```

范围-列表分区表示例如下：

```
create table tab2(
    cust_id integer,
    name char(128),
    score integer
)
partition by range(cust_id)
subpartition by list(score)
(
    partition p0 values less than (100)
    (
        subpartition shangxun1 values(1,2,3) in dbs1,
        subpartition zhongxun1 values(4,5,6) in dbs2
    )
);
```

范围-哈希分区表示例如下：

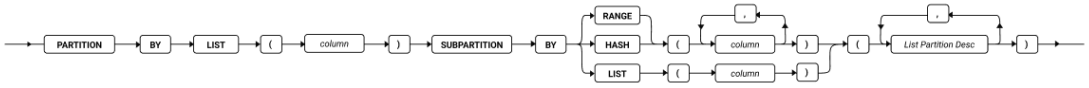
```
create table tab3(
    cust_id integer,
    name char(128),
    score integer
)
partition by range(cust_id)
subpartition by hash(score)
(
    partition p0 values less than (100)
```

```
    (
        subpartition p0_1 dbspace datadbs1,
        subpartition p0_2 dbspace datadbs2
    )
);
```

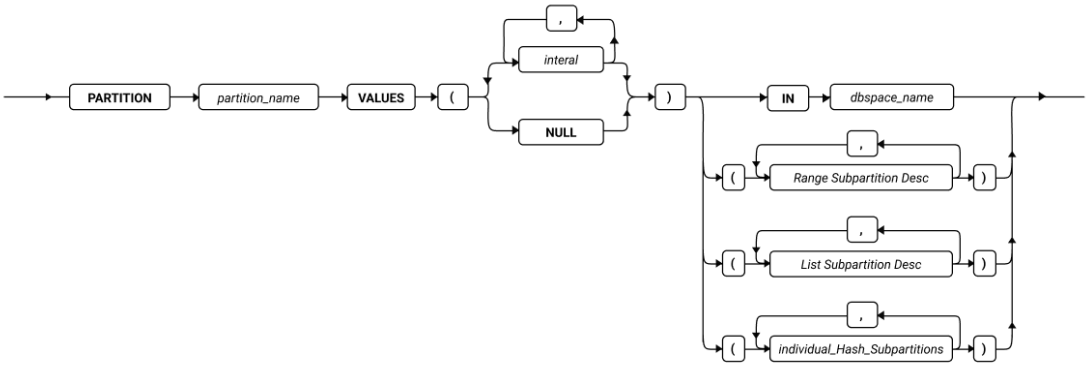
### *一级分区是列表分区的二级分区的创建*

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

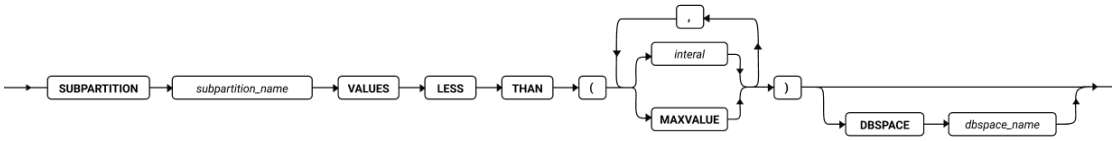
SECOND LEVEL PARTITION 子句语法图:



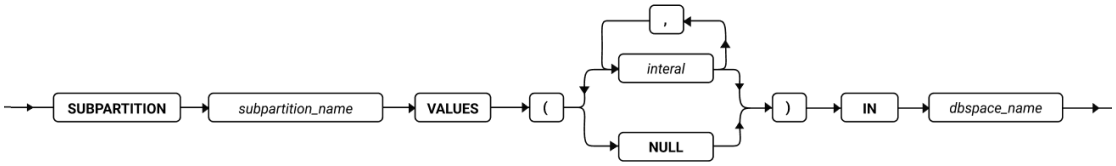

List Partition Desc 语法图：



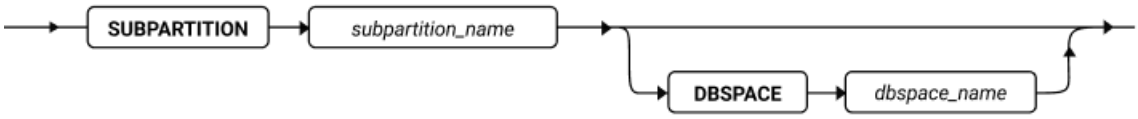

Range Subpartition Desc 语法图：




List Subpartition Desc 语法图：




individual_Hash_Subpartitions 语法图：




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***column*** | 应用分片存储策略的列 | 必须是表中的列 | 标识符 |
| ***dbspace_name*** | 存储表分片的 Dbspace | 最多可以指定 2,048 个 ***dbspaces***。所有存储分片的 ***dbspaces*** 必须具有相同的页大小。 | 标识符 |
| ***partition*** | 此处为分片声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |
| ***subpartition_name*** | 此处为二级分区声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |
| ***Interal*** | 分区字段进行分区的值,如果是二级范围分区 values less than Interal 则是小于这个值<br>如果是二级列表分区则是等于这个值 | 必须是与分片键表达式的数据类型符合的值 | 标识符 |

**用法**

- 一级分区只能指定一个字段做分区字段；
- 二级分区可以在范围，列表，哈希选择其中一种，不能多种二级分区混合使用；
- 二级分区为范围分区或者哈希分区可以指定一个或者多个字段做分区字段；
- 二级分区为列表分区只能指定一个字段当做分区字段；
- 一级分区不能指定 dbspace。当二级分区是范围分区且没有定义二级分区范围,只定义了唯一的一级分区的时候,如果是一级列表分区必须使用 in dbspace_name 语法指定 dbspace；
- 二级分区为范围分区或者哈希分区可以指定 dbspace，如果不明确指定则会使用系统默认 dbspace；

- 二级分区为列表分区必须明确指定使用的 dbspace；
- 二级分区的范围分区如果定义多个字段做分区字段，如果其中一个字段设置了 maxvalue，那么这个分区的所有字段需要同时设置为 maxvalue；
- 二级范围分区只能有一个子分区为 maxvalue；
- 整个表的所有二级列表分区只能有一个 null；
- 目前布尔类型(BOOLEAN)、LVARCHAR 类型、虚拟列和 ROWNUM 不支持做分区字段；
- 表内的分区字段与非分区字段均不允许重命名列名操作(RENAME COLUMN) 。

**示例**

**列表-范围分区示例如下：**

```
create table tab4(
    cust_id integer,
    name char(128),
    score integer
)
partition by list(cust_id)
subpartition by range(score)
 (
    partition p0 values (100)
    (
        subpartition p0_1 values less than (50) dbspace dbs1_1,
        subpartition p0_max values less than (100) dbspace dbs1_2
    )
);
```

**列表-列表分区示例如下：**

```
create table tab5(
    cust_id integer,
    name char(128),
    score integer
```

```
)
partition by list(cust_id)
subpartition by list(score)
(
    partition p0 values (100)
    (
        subpartition shangxun1 values(1,2,3) in dbs1,
        subpartition zhongxun1 values(4,5,6) in dbs2
    )
);
```

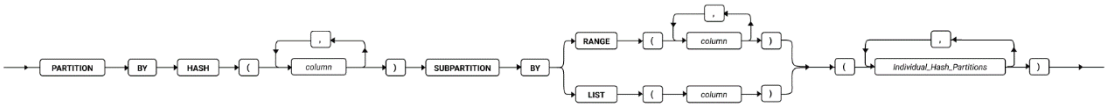
**列表-哈希分区示例如下：**

```
create table tab6(
    cust_id integer,
    name char(128),
    score integer
)
partition by list(cust_id)
subpartition by hash(score)
(
    partition p0 values (100)
    (
        subpartition p0_1 dbspace datadbs1,
        subpartition p0_2 dbspace datadbs2
    )
);
```

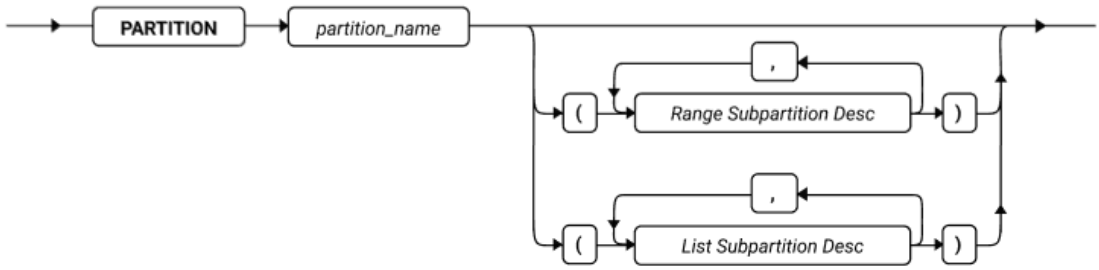
### *一级分区是哈希分区的二级分区的创建*

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

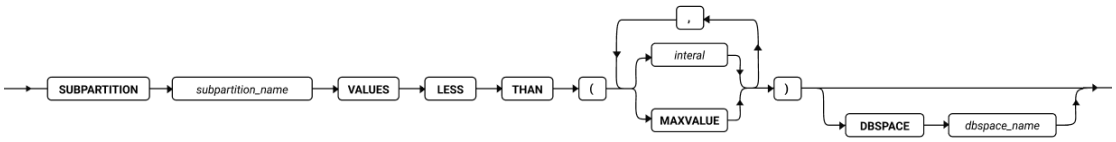
SECOND LEVEL PARTITION 子句语法图：



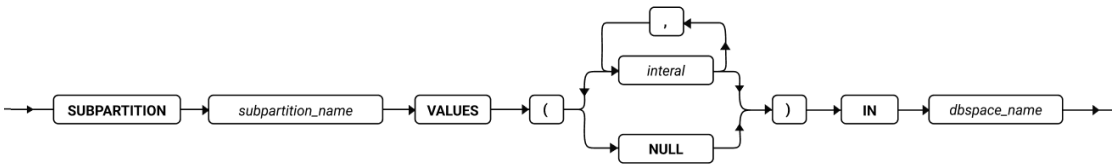
Individual_Hash_Partitions 语法图：




Range Subpartition Desc 语法图：




List Subpartition Desc 语法图：




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***column*** | 应用分片存储策略的列 | 必须是表中的列 | 标识符 |
| ***dbspace_name*** | 存储表分片的 Dbspace | 最多可以指定 2,048 个 ***dbspaces***。所有存储分片的 ***dbspaces*** 必须具有相同的页大小。 | 标识符 |
| ***partition*** | 此处为分片声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |
| ***subpartition_name*** | 此处为二级分区声明的名称 | 在该表的分片名称中必须是唯一的 | 标识符 |
| ***Interal*** | 分区字段进行分区的值，如 | 必须是与分片键表达式 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 果是二级范围分区 values less than Interal 则是小于这个值<br>如果是二级列表分区则是等于这个值 | 的数据类型符合的值 | |

**用法**

- 一级分区只能指定一个或多个字段做分区字段；
- 二级分区可以在范围，列表选择其中一种，不能多种二级分区混合使用；
- 二级分区为范围分区可以指定一个或者多个字段做分区字段；
- 二级分区为列表分区只能指定一个字段当做分区字段；
- 一级分区不能指定 dbspace；
- 二级分区为范围分区可以指定 dbspace，如果不明确指定则会使用系统默认 dbspace；
- 二级分区为列表分区必须明确指定使用的 dbspace；
- 二级分区的范围分区如果定义多个字段做分区字段，如果其中一个字段设置了 maxvalue，那么这个分区的所有字段需要同时设置为 maxvalue；
- 哈希分区只能使用指定分区名写法；
- 二级范围分区只能有一个子分区为 maxvalue；
- 整个表的所有二级列表分区只能有一个 null；
- 目前布尔类型(BOOLEAN)、LVARCHAR 类型、虚拟列和 ROWNUM 不支持做分区字段；
- 表内的分区字段与非分区字段均不允许重命名列名操作(RENAME COLUMN) 。

**示例**

哈希-范围分区表示例如下：

```
create table tab7(
    pid number(10),
    pname varchar(30),
    sex varchar(20),
    create_date date
)
```

```
partition by hash(sex)
subpartition by range(pname)
(
    partition p1
    (
        subpartition p1_1 values less than('aa'),
        subpartition p1_2 values less than('dd')
    ),
    partition p2
    (
        subpartition p2_1 values less than('ff'),
        subpartition p2_2 values less than('kk')
    )
 );
```

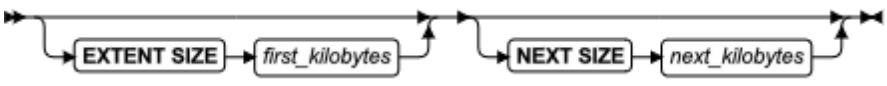
哈希-列表分区表示例如下：

```
create table tab8(
    sid number,
    sex varchar(20),
    province varchar(20)
)
partition by hash(sex)
subpartition by list(province)
(
    partition p1
    (
        subpartition north values('河南','安徽','河北','山东') in dbs1,
        subpartition south values('广东','海南','广西','江西') in dbs1_1,
        subpartition west values('青海','宁夏','山西') in dbs1_2,
        subpartition east values('上海','浙江','江苏') in dbs2
    ),
    partition p2
);
```

## EXTENT SIZE 选项

EXTENT SIZE 选项可以定义分配到该表的存储 extent 的大小。

EXTENT SIZE 选项




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *first_kilobytes* | 表的第一个 extent 的长度，单位是千字 | 必须返回正整数；最大值是 chunk 大小 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 节；缺省值是 16。 | | |
| *next_kilobytes* | 每个后继 extent 的长度，单位是千字节；缺省值是 16。 | 必须返回正整数；最大值是 chunk 大小 | 表达式 |

**用法**

***first_kilobytes*** 的最小长度（和 ***next_kilobytes*** 的）是您系统的磁盘页大小的四倍。例如，如果您有 2 千字节大小的页的系统，则最小长度是 8 千字节。

如果 CREATE TABLE（或 CREATE TEMP TABLE）语句不包含 IN dbspace 子句，则没有 EXTENT SIZE 规范，且没有 NEXT SIZE 规范，且不会为此表分片存储，直到向表中插入至少一条数据行。第一个 extent 的缺省大小不是 16 千字节就是 4 页面。

下个示例指定第一个 extent 大小为 20 千字节，并允许其余的 extent 使用缺省大小：

```
CREATE TABLE emp_info
        (
        f_name        CHAR(20),
        l_name        CHAR(20),
        position      CHAR(20),
        start_date DATETIME YEAR TO DAY,
        comments      VARCHAR(255)
        )
        EXTENT SIZE 20;
```

如果表中没有数据，则您可以使用 SQL 的 ALTER TABLE MODIFY EXTENT SIZE 或 ALTER TABLE MODIFY NEXT SIZE 语句更改此空表的第一个 extent 和随后的 extent 的大小。然而，对于包含一行或多行的表，并不支持这些操作并不支持 。有关 ALTER TABLE 语句的这些选项的更多信息，请参阅 MODIFY EXTENT SIZE 子句 和 MODIFY NEXT SIZE 子句。

如果需要修改表的 extent 的大小，您可以在生成的卸装表的模式文件中修改 extent 和下一个 extent 大小。例如，要使数据库更加有效率，您可以卸装一个表，在模式文件中修改 extent 的大小，接着创建并装入新表。有关如何优化 extent 的信息，请参阅 *GBase 8s 管理员指南*。

## 表的 COMPRESSED 选项

当数据加载到表或表分片中时，使用 CREATE TABLE 语句的 COMPRESSED 选项启用自动压缩大量数据行。

在使用 COMPRESSED 选项创建表之后，数据库服务器自动创建压缩自动并压缩 2000 行之后的数据行或更多加载到表中或分片中数据。 如果数据通过轻量级追加加载，则前 2000 行和所有连续的行都会被压缩。如果数据通过其它方式加载，则压缩前 2000 行之后的所有连续的行。要压缩初始的 2000 行，请运行带有 table compress 或 fragment compress 参数的 SQL 管理 API task() 或 admin() 函数。

COMPRESSED 选项只对行中的数据启用。COMPRESSED 选项不会启用 dbspace 或索引中的简单大对象的自动压缩。（您可以使用 CREATE INDEX 语句的 COMPRESSED 关键字创建一个压缩的 B-tree 索引。）

以下示例创建了一个设置为可以自动压缩的表：

```
CREATE TABLE cust5 (   ...) COMPRESSED;
```

以下示例还创建了一个名为 t 的表，并定义其第一个和后续 extent 大小设置为自动压缩：

```
CREATE TABLE t(c int, d int) EXTENT SIZE 32 NEXT SIZE 32 COMPRESSED;
```

要禁用自动压缩新行中数据，请在表上运行带有 table uncompress 参数的 SQL 管理 API task() 或 admin() 函数。您可以使用 fragment uncompress 和 fragment compress 参数控制表分片的压缩。

## 延迟 extent 存储分配

如果 IN ***dbspace*** 是新表的唯一的存储规范，则会在创建表时为第一个 extent 缺省分片 16 千字节存储（或者四页面的足够的存储，如果 4 页需要对于 16 千字节）。

但是，如果 CREATE TABLE 语句不包含以下存储规范，则不会第一个 extent 分配存储：

- EXTENT SIZE
- NEXT SIZE
- IN ***dbspace*** 。

在这种情况中，除非第一行已经存储在表中，否则会延迟第一个 extent 的存储分配。

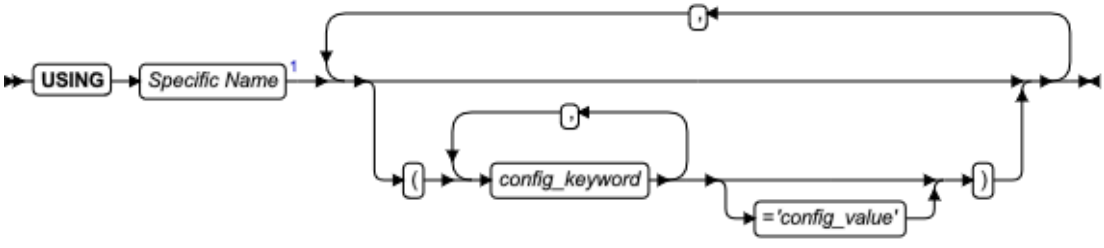
相同存储分配延迟适用于通过不包含以上存储规范列表的 CREATE TEMP TABLE 语句定义的表。

当首次向已延迟 extent 分配的表中插入行时，第一个 extent 的缺省大小为 16 千字节。如果 16 千字节不足 4 个页面，则第一个 extent 大小将会是 4 页。

## USING 存取方法子句

USING 存取方法子句可以指定一个存取方法。

USING 存取方法子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *config_keyword* | 与指定的存取方法关联的配置关键字 | 不大于 18 字节，存取方法必须存在。 | 文字关键字 |
| *config_value* | 指定配置关键字的值 | 不大于 236 字节。必 | 引用字符 |

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| | | 须由存取方法定义。 | 串 |

***主存取方法***是执行 DDL 和 DML 操作的例程集合，如 create 、drop 、insert 、delete 、update 和 scan ，使表可供数据库服务器访问。 GBase 8s 提供了一种内含子的主存取方法。

可以在数据库服务器外的 extspace 或数据库服务器内的 sbspace 中存储并管理虚拟表。（请参阅 存储选项。）可以使用 SQL 语句存储虚拟表。访问虚拟表需要用户定义的主要存取方法。

DataBlade 模块可以提供其它主要存储方法来访问虚拟表。访问虚拟表时，数据库服务器将调用与该存取方法关联的例程而不是内置表例程。有关这些主存取方法的更多信息，请参阅您的存取方法文档。

您可以使用 MI_TAB_AMPARAM 宏从表描述符中检索存取方法的配置值列表（**mi_am_table_desc**）。不是所有的关键字都需要配置值。

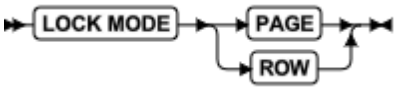
存取方法必须已经存在。例如，如果名为 `textfile` 的存取方法已经存在，则可以用以下语法来指定：

```
CREATE TABLE mybook
        (... )
        IN myextspace
        USING textfile (DELIMITER=':');
```

## LOCK MODE 选项

使用 LOCK MODE 选项指定该表的锁定粒度。

LOCK MODE 选项




下表描述了锁定粒度的可用选项。

| 粒度 | 作用 |
|---|---|
| PAGE | 在一整页的行上获取并释放一个锁<br>这是缺省的锁定粒度。当您知道行分组到各页所依照的顺序与您正在用来处理所有行的顺序相同时，页级别锁定就特别有用。例如，如果您正在按照表的集群索引的顺序来处理表的内容时，页锁定将十分使用。 |
| ROW | 在每一行上获取并释放一个锁<br>行级别锁定提供最高级别的并发性。如果同时使用许多行，则锁定管理开销将变的十分可观。根据您的数据库服务器的配置，也可以超出可用锁的最大数目。但是 GBase 8s 在 32 为平台上可 |

| | |
|---|---|
| | 支持 180 万个锁，或者在 64 位平台上支持 6000 万个锁。只有行级别锁定的表才支持 LAST COMMITTED 隔离级别功能。 |

接着可用使用 ALTER TABLE ... LOCK MODE 语句更改表的锁定方式。

## 优先顺序和缺省行为

在 GBase 8s 中，您不需要每次创建新表时都指定锁定方式。您可以在以下环境中，全局地设定所有新表的锁定粒度：

- 单用户的数据库会话

  可以在当前会话期间，把 **IFX_DEF_TABLE_LOCKMODE** 环境变量设置为指定新表的锁定方式。

- 数据库服务器（数据库服务器上的所有会话）

  如果您是一个 DBA ，则可以设置 ONCONFIG 文件中的 DEF_TABLE_LOCKMODE 配置参数，以缺省数据库服务器中所有新表的锁定方式。

  如果您不是 DBA，可以在运行 **oninit** 之前，设置数据库服务器的 **IFX_DEF_TABLE_LOCKMODE** 环境变量用来指定数据库服务器中所有新表的锁定方式。

CREATE TABLE 语句中的 LOCK MODE 设置优先于 **IFX_DEF_TABLE_LOCKMODE** 环境变量和 DEF_TABLE_LOCKMODE 配置参数。

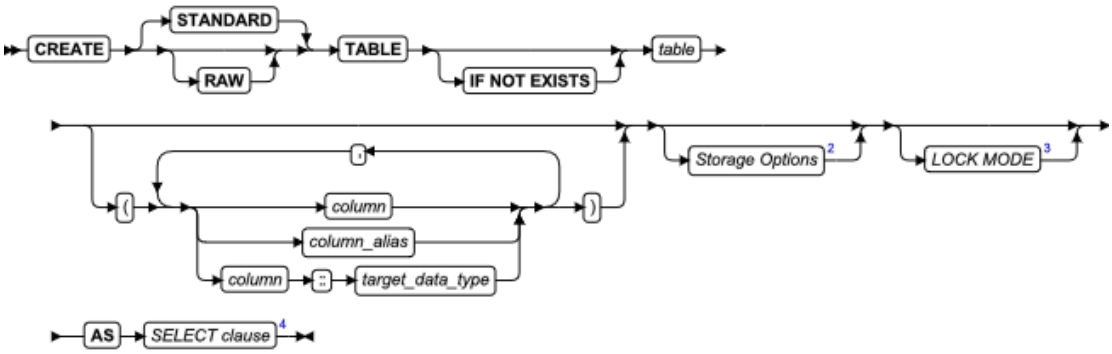
如果 CREATE TABLE 未设置锁定方式，则缺省方式将取决于 **IFX_DEF_TABLE_LOCKMODE** 环境变量或 DEF_TABLE_LOCKMODE 配置参数的设置。有关 **IFX_DEF_TABLE_LOCKMODE** 的信息，请参阅 *《GBase 8s SQL 指南：参考》*。有关 DEF_TABLE_LOCKMODE 配置参数的信息，请参阅 *GBase 8s 管理员参考手册*。

## AS SELECT 子句

使用 CREATE TABLE 语句的 AS SELECT 子句创建新的表并插入指定查询的结果集的数据行。

此语法在功能上与 SELECT 语句的 INTO STANDARD 和 INTO RAW 子句 非常相似。

当您使用 AS SELECT 子句创建新的查询结果表，并使用指定查询返回的限定行填充该表时，只有 CREATE TABLE 语句语法的以下的子集才有效：



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 查询的 FROM 子句中的表的列。这将是结果表中的列名。 | 必须在查询结果集中存在 | 标识符 |
| *column_alias* | 列的别名或显示标签。声明结果表中的列名称。 | 查询的 Projection 子句中的任何非平凡表达式在结果表中需要其列名的别名或显示标签 | 标识符 |
| *table* | 此处为结果表声明的名称 | 在数据库中的物化视图、视图、表、序列和同义词名称中必须唯一。 | 标识符 |
| *target_data_type* | 显式强制转型返回的数据类型。它将会是结果表中列的数据类型。 | 请参阅 对于目标数据类型的规则 | 数据类型 |

**用法**

当使用 CREATE TABLE . . . AS SELECT 语句创建新的永久表来存储查询的结果时，您可以指定该表的日志记录方式为 STANDARD 或 RAW 。如果您都忽略了这些关键字，则缺省为 STANDARD 。

您还可以可选地为此查询结果表指定下列属性：

- 独立存储位置，或者分布存储方案 e
- 它的第一个 extent 和下一个 extent 的存储大小
- 它的 PAGE 或 ROW 锁定粒度

如果您省略了存储或锁定规范，则数据库服务器使用缺省值。

如果在数据库服务器向该表填充大量 AS SELECT 子句中查询返回的行时发生了错误，则该操作会回滚，且不会创建或填充新表。

出现在 AS SELECT 子句的 Projection 列表中的列可以来自任何本地表、视图或来自远程数据库（必须在限定的表名中引用），但是新表必须在本地数据库中创建。

在 grid 环境中，如果 AS SELECT 子句的 FROM 子句中的表与所有参与的指定的 grid 或地区的数据库服务器具有相同的结构，则您可以包含 AS SELECT 子句以从 grid 查询创建表。

**结果表中列的名称**

缺省情况下，新永久表中的列名称是在 Projection 子句的 SELECT 列表中指定的名称。如果星号（*）是 Projection 子句的 SELECT 列表，则星号将扩展为 SELECT 语句的 FROM 子句中对应的表或视图中的所有列名。由 FROM 子句指定表对象中的任何显式或隐式影子列不会由星号规范扩展。

在实现 Enterprise Replication 的系统上，可以使用 ALTER TABLE 语句的 ADD CRCOLS 、ADD REPLCHECK 和 ADD ERKEY 选项将相应的影子列添加到 AS SELECT 子句创建的结果表中。

除了简单列表达式之外， Projection 子句的 SELECT 列表中所有表达式都必须具有显示标签（也称为 *列别名*）。这用作新查询结果表中相应列的标识符。如果列表达式没有显示标签，则结果表使用查询的 FROM 子句中的源表中的列名。

在 CREATE TABLE . . . AS SELECT 语句中，可以使用以下两种方法指定列别名：

- 作为逗号分隔的别名列表，紧跟在 TABLE 关键字后面，类似于 INSERT INTO . . . SELECT FROM 语句
- 作为 Projection 子句中 SELECT 列表的一部分，与结构表中的 SELECT . . . INTO STANDARD 或 SELECT . . . INTO RAW 语句可以创建

如果在 CREATE TABLE . . . AS SELECT 语句中存在 Projection 子句的 SELECT 列表和 TABLE 关键字后面的逗号分隔的别名列表，则列别名的逗号分隔列表优先。在这种情况下，将忽略在 AS SELECT 子句中声明的任何列别名。

在以下情况中 CREATE TABLE . . . AS SELECT 语句会发生错误并失败：

- I 如果未为非重要列表达式声明显示标签或列别名。
- 如果显示标签或列名称与新结果表中的另一列具有相同的名称。
- 除了存储选项和 LOCK MODE 属性之外，CREATE TABLE . . . AS SELECT 语句不能为新表的列定义约束或任何其它特殊属性。
- 如果逗号分隔的别名列表位于 TABLE 关键字之后，但该列表的别名数量少于 Projection 子句的 SELECT 列表中的表达式数量。

但是，如果 TABLE 关键字后面的列别名列表具有比 Projection 子句的 SELECT 列表更多的项目，在这种情况下，数据库服务器将忽略过多的列别名，并且不会发生异常。

**支持的数据类型**

CREATE TABLE . . . AS SELECT 已经支持用户定义的数据类型和所有内置的 GBase 8s 数据类型。

但是，在新的结果表中只允许有一个序列列。在包含了第一个 SERIAL 、SERIAL8 或 BIGSERIAL 类型的列后，所有随后的 SERIAL 、SERIAL8 或 BIGSERIAL 列作为 INTEGER 、INTEGER8 或 BIGINTEGER 列创建。

**结果表上的限制**

与大多数 DDL 语句一样，使用完全限定表名在其它数据库中创建新结果表的尝试会失败，并显示语法错误。类似地，创建于现有表具有相同名称的结果表是一个错误，除非 AS SELECT 子句包含 `IF NOT EXISTS` 关键字。 SELECT 语句的 SELECT INTO . . . TABLE 语法作为子查询的一部分时是无效的。

AS SELECT 子句可以包含在 ORDER BY 子句中而不在 Projection 子句的 SELECT 列表中的列。

**IF NOT EXISTS 关键字**

如果您在 AS SELECT 子句中为查询结果表声明的名称在数据库的永久表、同义词、视图和序列的名称中是唯一的，则无论 AS SELECT 子句是否包含 IF NOT EXISTS 关键字，数据库服务器总会创建查询结果表并填充所有查询返回的行。（如果查询没有返回行，则该结果表为空，但是它的结构已经在数据库的系统目录中注册，并作为一个新的永久表存在。）

如果 AS SELECT 子句包含了 IF NOT EXISTS 关键字，但您为结果表声明的名称在数据库的永久表的名称中并不是唯一的，则不会执行该 AS SELECT 子句定义的查询，并且数据库服务器返回以下消息：

0 row(s) retrieved into table.

如果 AS SELECT 子句忽略了 IF NOT EXISTS 关键字，且您为查询结果表声明的名称在数据库的永久表的名称中并不是唯一的，则不会创建结果表，并且数据库服务器返回一个错误。

**创建和填充结果表的示例**

以下示例创建了一个名为 **rtabl** 的新 RAW 表，用此表来存储连接查询的结果：

```
CREATE RAW TABLE IF NOT EXISTS rtab1
        AS
        SELECT t1col1, t1col2, t2col1
        FROM tab1, tab2
        WHERE t1col1 < 100 and t2col1 > 5;
```

在以上示例中，新的查询结果表 **rtab1** 将包含列 **t1col1** 、**t1col2** 和 **t2col1** 。

下一个示例发生错误 -249 并失败，因为它没有为 col1+5 列表达式声明显示标签：

```
CREATE TABLE IF NOT EXISTS qtab1
        AS
        SELECT col1+5, col2
        FROM tab1;
```

通过在包含 + 运算符的 Projection 子句中为列表表达式声明列别名 **qcol1**，以下修订的查询避免了之前示例返回的 -249 错误：

```
CREATE TABLE IF NOT EXISTS qtab1 (qcol1, col2)
        AS
        SELECT col1+5, col2
        FROM tab1;
```

以上正确的示例创建了标准表 **qtabl** 来存储 AS SELECT 子句的查询结果。

下一示例使用不同但等价的语句来在 AS SELECT 子句中声明相同的 **qcol1** 别名，而不是在列别名列表中声明：

```
CREATE TABLE IF NOT EXISTS qtab1
        AS
        SELECT col1+5 qcol1, col2
        FROM tab1;
```

以上的 CREATE TABLE 语句同样避免了 -249 错误，并创建结果和数据内容与前面示例中 **qtabl** 表相同的结果表。这这些示例中，结果表都有两列， **qcol1** 和 **col2** 。如果 **col1** 是 INTEGER 类型，则 **qcol1** 将是 DECIMAL 类型，返回的数据类型来自 col1+5 表达式。

如语法图所示，CREATE TABLE 语句的 Storage 和 Lock Mode 选项对 AS SELECT 子句有效。以下示例使用 FRAGMENT BY EXPRESSION 关键字为查询结果表定义分布存储策略，其中 **fcol1** 列别名是分片键，ROW 是锁定粒度：

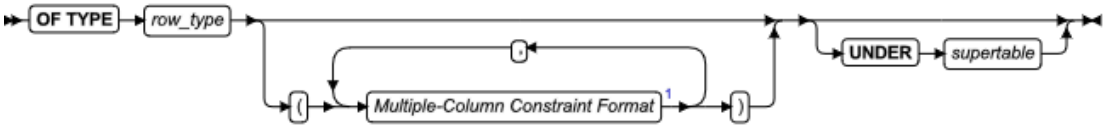
```
CREATE TABLE IF NOT EXISTS permtab (fcol1, col2)
        FRAGMENT BY EXPRESSION
        fcol1 < 300 IN dbs1,
        fcol1 >=300 IN dbs2
        LOCK MODE ROW
        AS SELECT col1::FLOAT, col2
        FROM tab1;
```

任何 **fcol1** 值低于 300 的行会插入到 dbspace **dbs1** 中。具有较大 **fcol1** 值的行会存储在 **dbs2** dbspace 中。

## OF TYPE 子句

使用 OF TYPE 子句为对象关系数据库创建***类型表***。类型表是您将已命名的 ROW 数据类型指定到目标表。

OF TYPE 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *row_type* | 该表所基于的 ROW 类型的名称 | 必须是在本地数据库中注册的已命名 ROW 数据类型 | 标识符 |
| *supertable* | 该表继承其属性的表名称 | 必须作为类型表存在 | 标识符 |

如果您使用 UNDER 子句，则 ***row_type*** 必须从 ***supertable*** 的 ROW 类型派生得到。类型层次结构必须已经存在，其中新表的已命名 ROW 类型是 ***supertable*** 的已命名 ROW 类型的子类型。

***不规则行***是来自表层次结构的一组行的集合，层次结构中的类型表的列数不固定。一些 API ，例如 GBase 8s ESQL/C 和 GBase 8s JDBC Driver，不支持返回 jagged 行的查询。

当创建类型表时，CREATE TABLE 不能为它的列指定名称，因为列名称在创建 ROW 类型时已经声明了。类型表的列对应于指定 ROW 类型的字段。ALTER TABLE 已经不能向类型表添加其它列。

例如，假设您创建一个指定的 ROW 类型 student_t 如下：

```
CREATE ROW TYPE student_t
        (name        VARCHAR(30),
        average      REAL,
        birthdate    DATETIME YEAR TO DAY);
```

如果在 OF TYPE 子句中为表指定 student_t 类型，则该表是一张类型表，该表的列与指定 ROW 类型 student_t 的字段名称和数据类型都相同（顺序也相同）。 例如，以下 CREATE TABLE 语句创建了一个类型是 student_t 的名为 students 的类型表：

```
CREATE TABLE students OF TYPE student_t;
```

students 表有以下列：

```
name        VARCHAR(30)
average     REAL
birthdate DATETIME YEAR TO DAY
```

有关指定 ROW 类型的更多信息，请参阅 CREATE ROW TYPE 语句 。

## 在类型表中使用大对象数据

如果您想要创建的表中包含大对象的列，则使用 BLOB 或 CLOB ，而不要使用 BYTE 或 TEXT 数据类型。为保持向后兼容性，您可以创建包含 BYTE 或 TEXT 字段的指定 ROW 类型，并使用该 ROW 类型将现有的(未归类的)表重新创建为类型表。尽管您可以使用包含 BYTE 或 TEXT 字段的指定 ROW 类型来创建类型表，但是这种 ROW 类型作为列是无效的。然而，可以在类型表和列中使用包含 BLOB 或 CLOB 字段的 ROW 类型。

## 使用 UNDER 子句

使用 UNDER 子句来指示继承（即，将表定义为子表）。子表从它上面的超级表那里继承属性。此外，您还可以为子表定义新属性。

继续 OF TYPE 子句中的示例，以下语句创建了一个类型表 grad_students ，它继承了 students 表的所有列，同时还有对应于 grad_student_t ROW 类型中字段的 adviser 和 field_of_study 这两列：

```
CREATE ROW TYPE grad_student_t
        (adviser          CHAR(25),
        field_of_study CHAR(40)) UNDER student_t;

CREATE TABLE grad_students OF TYPE grad_student_t   UNDER students;
```

使用 UNDER 子句时，子表将继承这些属性：

- 超级表中的所有列
- 超级表上定义的所有约束
- 超级表上定义的所有索引
- 超级表上定义的所有触发器

- 参照完整性约束
- 存取方法
- 存储方法（包含分片存储策略）

  如果子表未定义分片，但它的超级表定义了分片，则该子表将继承超级表的分片。

**提示：** 当子表已经创建后，被添加到超级表的可继承属性将自动被现有子表继承。创建子表前不需要为超级表添加所有可继承的属性。

### *表层级结构上的限制*

继承只在一个方向发生，即从超级表到子表。子表的属性将不被超级表继承。系统目录信息的这节列出继承的数据库对象，在系统目录中没有关于这些子表的信息。

表层次结构中不能有连个表具有相同的数据类型。例如，以下的代码示例的最后一行是无效的，因为表 tab2 和 tab3 不能有相同的行类型（rowtype2）：

```
create row type rowtype1 (...);
create row type rowtype2 (...) under rowtype1;
create table tab1 of type rowtype1;
create table tab2 of type rowtype2 under tab1;
create table tab3 of type rowtype2 under tab1; -- This is not valid.
```

## 表上的存取特权

表上的特权描述了谁可以存取表中的信息以及谁可以创建新表。有关存取特权的更多信息，请参阅 GRANT 语句 章节的描述。

在兼容 ANSI 的数据库中，不存在缺省的表级特权。必须显式地授权这些特权。将 **NODEFDAC** 环境变量设置为 yes 以防止缺省的特权授予不兼容 ANSI 的数据库的新表中的 PUBLIC ，正如 *《GBase 8s SQL 指南：参考》* 中描述的那样。有关特权的更多信息，请参阅 *GBase 8s SQL 教程指南*。

## 系统目录信息

当您创建表时，数据库服务器将每个表的基础信息添加到 **systables** 系统目录表，并将列信息添加到 **syscolumns** 系统目录表。**sysfragments** 系统目录表包含关于分片存储策略和表分片位置的信息。**sysblobs** 系统目录表中包含 dbspace 和简单大对象的位置信息。（**sysmaster** 数据库中的 **syschunks** 表包含智能大对象的位置信息。）

**systabauth** 、**syscolauth** 、**sysfragauth** 、**sysprocauth** 、**sysusers** 和 **sysxtdtypeauth** 表包含不同的 CREATE TABLE 选项所需要的特权。

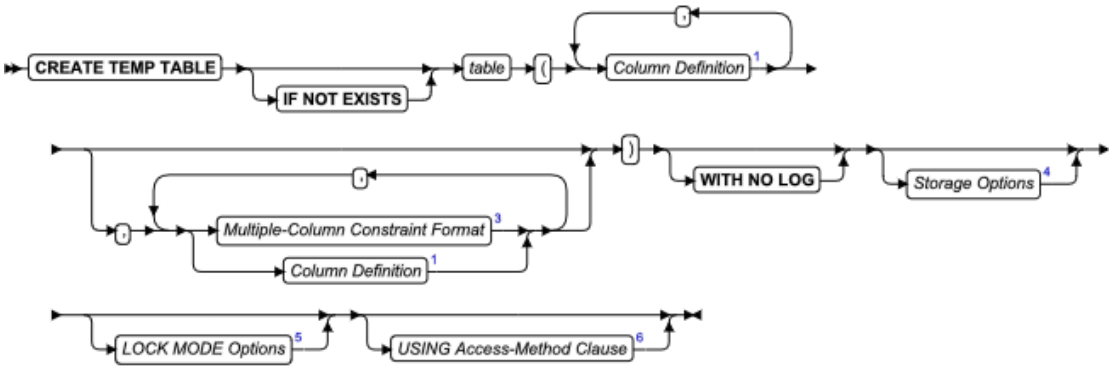
**sysextcols** 、**sysextdfiles** 和 **sysexternal** 表包含有关 CREATE EXTERNAL TABLE 语句注册于数据库中对象的其它信息。

systables 、sysxtdtypes 和 sysinherits 系统目录表提供类型表的信息。系统目录中将记录类型表层次结构、约束、索引和触发器，但不记录继承它们的子表。但是，会为超级表和子表记录分片存储信息。有关继承的更多信息，请参阅 GBase 8s SQL 教程指南。

## 2.46 CREATE TEMP TABLE 语句

使用 CREATE TEMP TABLE 语句在当前数据库中创建临时表。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *table* | 声明表的名称 | 在会话中必须是唯一的。请参阅 命名临时表 | 标识符 |

**用法**

您必须具有数据库上的 Connect 特权才能创建临时表。该临时表只对创建它的用户可见。

如果您包含了可选的 IF NOT EXISTS 关键字，则当指定名称的临时表已经在当前数据库中注册过时，数据库服务器不采取任何操作（而不是向应用程序发送异常）。

您还可以使用 CREATE TEMP TABLE 语句在临时表上定义索引和约束。

在 DB-Access 中，如果您设置了 **DBANSIWARN** 则在 CREATE SCHEMA 语句外使用 CREATE TEMP TABLE 语句会生成警告。

在 ESQL/C 中，如果您使用 **-ansi** 标志或设置 **DBANSIWARN** 环境变量则 CREATE TEMP TABLE 语句生成警告。

### 命名临时表

临时表是与会话而不是数据库关联在一起。创建临时表时，直到删除第一个临时表并结束会话之前都不能使用同样的名称创建另外一个临时表（即使是在另一个数据库中）。

临时表的名称必须遵循 SQL 标识符的要求，但是它不能是限定数据库对象的名称。当您使用 CREATE TEMP TABLE 语句创建临时表时，您不能指定任意授权标识符作为它的***所有者***。不像永久

表，临时表不能在用 ***owner*** 名称，或 ***database*** 名称，或 ***database server*** 名称的限定其标识符的 SQL 语句中被引用。

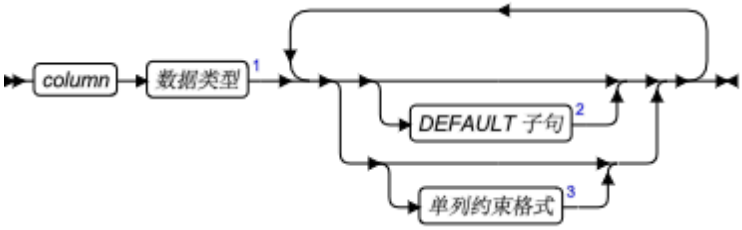
临时表的名称必须与当前数据库中任何其它表、视图、序列对象或同义词的名称都不相同。否则，此临时表会优先于会话中其它任何具有相同名称的永久表。但是，您在此声明的临时表名称不必不同于同一数据库中其它用户声明的临时表名称。

如果您翻出一个跨数据库的 DML 语句，它引用了一个远程的永久表，但您本地的数据库中包含一个相同名称的临时表，DML 语句访问本地的临时表，而非远程的临时表。

## CREATE TEMP TABLE 语句的列定义规范

使用 CREATE TEMP TABLE 语句的 Column Definition 段声明临时表的单列的数据类型和名称（以及缺省值和约束）。

列定义




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 表中列的名称 | 在它的表中必须唯一 | 标识符 |

CREATE TEMP TABLE 语句的这一部分与 CREATE TABLE 语句的相应部分几乎相同。不同之处在于临时表只允许更少的约束类型：

- 不能在列上定义引用约束。
- 数据类型不能是 IDSSECURITYLABEL。
- 临时表不支持 SECURED WITH ***label*** 选项。

与创建永久表一样，对于内置字符类型的列（如 CHAR 、LVARCHAR 、NCHAR 、NVARCHAR 或 VARCHAR），任何显式或缺省存储大小规范都以字节为单位进行解释，除非 SQL_LOGICAL_CHAR 配置参数设置为启用数据类型声明的逻辑字符语义。有关支持多字节代码集的语言环境（如 **UTF-8**）中 SQL_LOGICAL_CHAR 设置的影响的详细信息，请参阅 *GBase 8s 管理员参考手册*，其中单个逻辑字符肯需要多个字节的存储空间。

## 单列约束格式

使用单列约束格式为临时表中的单列创建一个或多个数据完整性约束。

单列约束格式

它是 CREATE TABLE 语句支持的单列约束格式语法的子集。

您可以在这些章节中找到特定约束的详细信息。

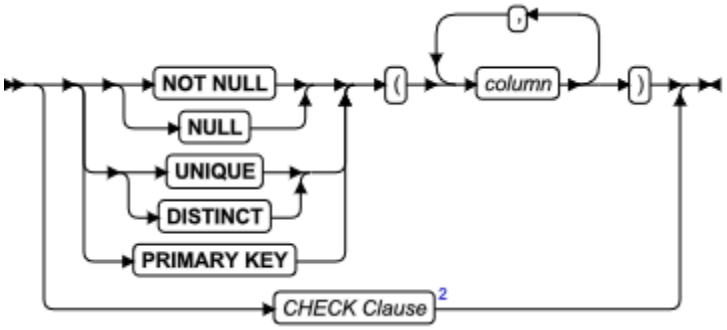
| 约束 | 有关更多信息，请参阅 |
| --- | --- |
| CHECK | CHECK 子句 |
| DISTINCT | 使用 UNIQUE 或 DISTINCT 约束 |
| NOT NULL | 使用 NOT NULL 约束 |
| NULL | 使用 NULL 约束 |
| PRIMARY KEY | 使用 PRIMARY KEY 约束 |
| UNIQUE | 使用 UNIQUE 或 DISTINCT 约束 |

您定义在临时表上的约束总是启用的。

## 多列约束格式

使用多列约束格式将一个或多个列用约束关联起来。它是单列约束格式的备选方案，允许您用约束将多列关联起来 。

多列约束格式




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 列或放置约束的列的名称 | 在表中必须是唯一的，但是在同一数据库中的不同表可以具有相同的名称 | 标识符 |

它是 CREATE TABLE 语句所支持的多列约束格式语法的子集。

它是 CREATE TEMP TABLE 的单列约束格式的备选方案，可用约束将多个列关联起来。您定义在临时表上的约束总是启用的。

您可以在这些章节中找到特定约束的详细信息。

| 约束 | 有关更多信息，请参阅 | 有关示例，请参阅 |
| --- | --- | --- |
| CHECK | CHECK 子句 | 在多个列上定义检查约束 |

| 约束 | 有关更多信息，请参阅 | 有关示例，请参阅 |
|---|---|---|
| DISTINCT | 使用 UNIQUE 或 DISTINCT 约束 | 多列约束格式的示例 |
| PRIMARY KEY | 使用 PRIMARY KEY 约束 | 定义组合的主键和外键 |
| UNIQUE | 使用 UNIQUE 或 DISTINCT 约束 | 多列约束格式的示例 |

另见唯一约束和唯一索引的区别章节。

## 使用 WITH NO LOG 选项

使用 WITH NO LOG 选项减少临时表的事务日志记录的开销。如果您指定 WITH NO LOG，在临时表上的数据操纵语言（DML）操作将不包含在事务日志记录中。

您在临时数据库中创建的所有的临时表都需要 WITH NO LOG 关键字。在一个集群环境中，当您在辅助服务器上创建临时表时需要 WITH NO LOG 关键字。

如果 ONCONFIG 参数 TEMPTAB_NOLOG 设置成 1 ，则临时表的日志记录被禁用，且所有的临时表都缺省为非日志记录的。 该项设置可以提高使用临时表的操作（例如，HDR 操作）的性能。当 TEMPTAB_NOLOG 设置禁用临时表的日志记录时，不需要 WITH NO LOG 选项。有关如何设置 TEMPTAB_NOLOG 参数的更多信息，请参阅 *GBase 8s 管理员参考手册*。

如果在不使用日志记录的数据库中使用 WITH NO LOG 选项，则 CREATE TEMP TABLE 语句的 WITH NO LOG 关键字不会生效。如果您的数据库不支持事务日志记录，所有的表的行为都将表现为已经指定了 WITH NO LOG 选项。

ALTER TABLE 语句不能更改临时表的日志记录的状态。一旦您关闭了临时表上的日志记录，则将无法再打开它；因此，临时表是始终记录日志的或从不记录日志的。

以下临时表在使用日志记录的数据库中不记录日志：

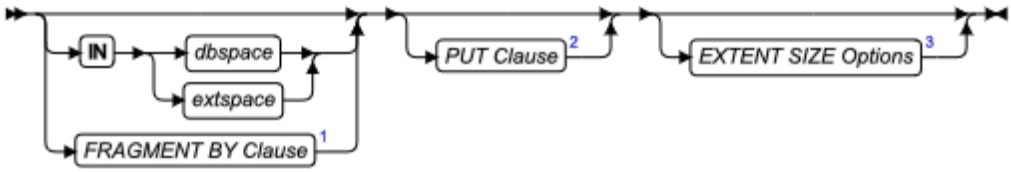
```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))
        WITH NO LOG;
```

类似于所有的数据定义语句（DDL）以上 CREATE TEMP TABLE 语句创建 tab2 是日志记录的。但是，WITH NO LOG 关键字将阻止任何 tab2 上的 DELETE 、INSERT 、LOAD 、MERGE 、SELECT 、UNLOAD 或 UPDATE 操作的事务日志记录。

## 临时表的存储选项

使用 CREATE TEMP TABLE 语句的存储选项指定表的存储位置和分布方案。它是 SQL ANSI/ISO 标准的扩展。

存储选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbspace* | 存储临时表的 Dbspace 或临时 dbspace | 必须已经存在 | 标识符 |
| *extspace* | gspaces 被指定给数据库服务器以外的存储区域的名称 | 必须已经存在 | 请参阅文档以了解您使用的存取方法。 |

只有包含 BLOB 或 CLOB 列的临时表可以包含 PUT 子句作为存储选项。

如果您在 IN 关键字之后指定一个临时 dbspace ，则数据库服务器不会执行任何该临时表的逻辑日志记录或物理日志记录。您无法镜像一个临时 dbspace。

如果您没有指定 extent 大小选项，则缺省的 extent 大小是 8 页面。

要在临时表上创建一个分片的、唯一的索引，您必须在 CREATE TEMP TABLE 语句中为此临时表指定一个显式的基于表达式的分布方案。（不支持通过 ROUND ROBIN 分片索引，对使用 LIST 或 INTERVAL 存储分区策略的表上的唯一索引，自动通过 LIST 或 INTERVAL 分片。）

## 临时表的存储位置

通过 CREATE TEMP TABLE 语句指定的分布方案（可以使用 IN 子句或 FRAGMENT BY 子句）优先于 **DBSPACETEMP** 环境变量或 DBSPACETEMP 配置参数指定的信息。

对于您没有指定显式分布方案的临时表，它的存储位置取决于 **DBSPACETEMP** 环境变量（或 DBSPACETEMP 配置参数）的设置。

- 如果没有设置 **DBSPACETEMP** 和 DBSPACETEMP ，则所有在建立的数据库（或者 **rootdbs** ，如果数据库服务器不在另一个 dbspace 中建立）的同一 dbspace 中创建的临时表都不会有分片。
- 如果临时表只有一个 dbspace 是通过 **DBSPACETEMP** 指定（或通过 DBSPACETEMP ，如果没有设置 **DBSPACETEMP** ），则将在指定的 dbspace 中创建所有的临时表而不分片。
- 如果 **DBSPACETEMP** （或者 DBSPACETEMP ，如果 **DBSPACETEMP** 没有设置）为临时表指定了两个或多个 dbspace ，则每个临时表将在指定的其中一个 dbspace 中创建。

  在不日志记录的数据库中，每个临时表都创建于一个临时的 dbspace 中；在支持事务日志记录的数据库中，临时表创建于标准 dbspace 中。数据库服务器跟踪哪个 dbspace 最近被使

用，并且当它接收到下一个要分配临时存储的请求时，数据库服务器使用下一个可用的 dbspace （以循环的方式）在 dbspace 之间平均分配 I/O 操作。

例如，如果您在日志记录的数据库中创建了三个临时表，DBSPACETEMP 指定 tempspc1 、tempspc2 和 tempspc3 作为该临时表的缺省 dbspace ，第一个在名为 tempspc1 的 dbspace 中，第二个在 **tempspc2** 中，第三个在 **tempspc3** 中（如果这些是临时存储的唯一请求）。

使用 SELECT INTO TEMP 和 WITH NO LOG 创建的临时分片将 DBSPACETEMP 配置参数或 **DBSPACETEMP** 环境变量列出的 dbspace 之间传播。因此，指定多个 dbspace 的 **DBSPACETEMP**（或者 DBSPACETEMP）设置可生成跨临时 dbspace 中所有 dbspace 循环分片。

如您创建了临时表并指定 WITH NO LOG ，则临时表上的操作将不会包含在事务日志记录中。如果 DBSPACETEMP 列表中有一个日志记录的空间，则使用 SELECT .. INTO TEMP WITH NO LOG 选项创建的临时表将在非日志记录的临时 dbspace 中通过循环分布方案分片。例如，如果来自 10 个 dbspace 的列表，只有一个 dbspace 是日志记录，则该表在其它 9 个非日志记录的临时 dbspace 中按照循环分布方案分片。

以下示例显示如果将数据插入到名为 result_tmp 的临时表中，并将用户定义的能返回多行的函数（f_one）的结构输出到文件中：

```
CREATE TEMP TABLE result_tmp( ... );
        INSERT INTO result_tmp EXECUTE FUNCTION f_one();
        UNLOAD TO 'file' SELECT * FROM result_tmp;
```

## 临时表的持续时间

临时表的持续时间取决于它是否进行日志记录。

进行日志记录的临时表将一直存在，直到以下情况发生：

- 应用程序断开连接。
- 对临时表发出 DROP TABLE 语句。
- 数据库已关闭。

当这些事件中的任意一件发生时，将删除临时表。

非日志记录的临时表包含那些用 CREATE TEMP TABLE 的 WITH NO LOG 选项创建的表。

非日志记录的临时表将一直存在，直到以下情况发生：

- 应用程序断开连接。
- 对临时表发出 DROP TABLE 语句。
- 数据库已关闭，且非日志记录的临时表包含至少一个用户定义类型的列，或者一个内置的透明数据类型（ GBase 8s 内置的透明数据类型包括 BLOB, BOOLEAN 、CLOB 、LVARCHAR 和 IDSSECURITYLABEL。）

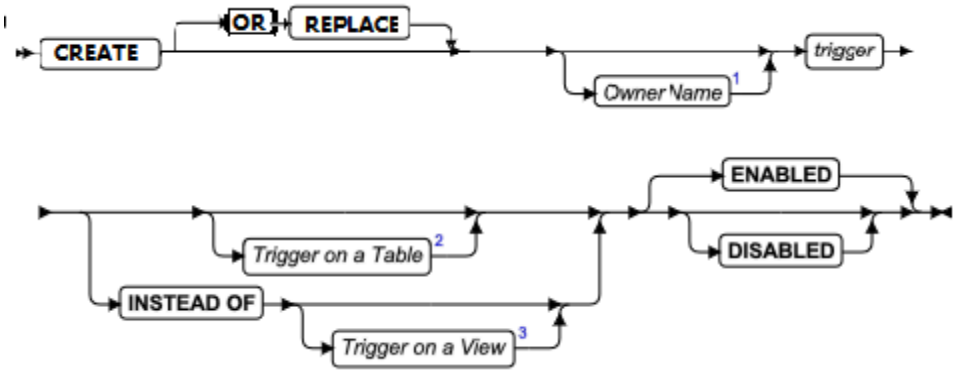
如果非日志记录的临时表不包含任何 UDT 或内置的透明数据类型的列，则当应用程序仍保留连接时，您可以使用此表将数据从一个数据库传输到另一个数据库，因为当数据库已关闭时该表并未销毁。如果要传输的数据包含 UDT 或内置透明数据类型，则您必须使用永久表（或者一些其它策略）。

# 2.47 CREATE TRIGGER 语句

使用 CREATE TRIGGER 语句在表上定义触发器。您还可以使用 CREATE TRIGGER 在视图上定义 INSTEAD OF 触发器。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *trigger* | 此处为新的触发器声明的名称 | 必须在当前数据库中的触发器名称中是唯一的 | 标识符 |

**用法**

除非被禁用。否则当指定的***触发事件***事件发生时，***触发器***将自动执行指定的 SQL 语句集合，称为***触发器操作***。

启动触发器操作的触发事件可以是 INSERT 、DELETE 、UPDATE 或 SELECT 语句。MERGE 语句还可以是 UPDATE 、DELETE 或 INSERT 触发的触发事件。该事件必须指定定义触发器的表或视图。（表上的触发器的 SELECT 或 UPDATE 事件还可以指定一列或多列。）

您可以用两种不同方法使用 CREATE TRIGGER 语句：

- 可以在当前数据库中的表上定义触发器。
- 也可以在当前数据库中的视图上定义 INSTEAD OF 触发器。

作为触发事件的实例的任何 SQL 语句称为***触发语句***。当事件发生时，在表上定义的触发器和在视图上定义的事件在是否执行触发语句方面有所不同：

- 对于表，触发事件和触发器操作都执行。
- 对于视图，只有触发器操作执行，而触发事件不执行。

通过定义指定的 DML 操作（触发事件）使数据库服务器执行特定操作所依据的规则，CREATE TRIGGER 语句可以支持数据库中数据的完整性。以下各节描述了语法元素。

| 子句 | 页 | 作用 |
| --- | --- | --- |
| OR REPLACE 子句 | OR REPLACE | 创建或替换触发器 |
| 定义触发器事件和操作 | 定义触发器事件和操作 | 将触发器的操作与事件关联 |
| 触发器模式 | 触发器方式 | 启用或禁用此触发器 |
| Insert 事件和 Delete 事件 | INSERT 事件和 DELETE 事件 | 定义 Insert 事件和 Delete 事件 |
| Update 事件 | UPDATE 事件 | 定义 Update 事件 |
| Select 事件 | SELECT 事件 | 定义 Select 事件 |
| Action 子句 | Action 子句 | 定义触发器操作 |
| 用于 Delete 的 REFERENCING 子句 | 用于删除的 REFERENCING 子句 | 为已删除的值声明限定符 |
| 用于 Insert 的 REFERENCING 子句 | 用于插入的 REFERENCING 子句 | 为已插入的值声明限定符 |
| 用于 Update 的 REFERENCING 子句 | 用于更新的 REFERENCING 子句 | 为新的和旧的值定义限定符 |
| 用于 Select 的 REFERENCING 子句 | 用于选择的 REFERENCING 子句 | 为结果集值声明限定符 |
| 相关的表操作 | 相关的表操作 | 定义触发器的操作 |
| 触发器操作 | 触发操作 | 定义触发器的操作 |
| 视图上的 INSTEAD OF 触发器 | 视图上的 INSTEAD OF 触发器 | 定义视图上的触发器 |
| INSTEAD OF 触发器的 Action 子句 | INSTEAD OF 触发器的 Action 子句 | 视图上的触发操作 |

## 临时表和永久表之间的差异

与永久表相比，临时表在以下方面不同：

- 可用的约束类型更少。
- 您可以指定的选项更少。
- 它们对于其它用户或会话不可见。
- 不出现在系统目录中。
- 不能保存它们，如临时表的持续时间中所述。

DB-Access 的 INFO 语句和 **Info Menu** 选项无法引用临时表。

## OR REPLACE

指定 OR REPLACE (CREATE OR REPLACE TRIGGER) 将创建一个新触发器或替换同名的现有触发器。

要添加触发器，您必须是该触发器的 Owner 或持有对该数据库的 DBA 权限。

以下示例语句创建触发器 tr1:

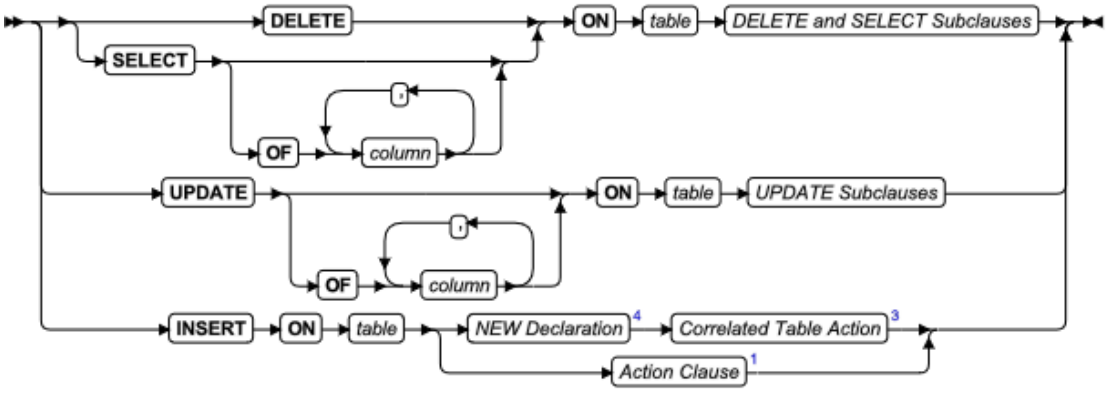
```
create or replace trigger tr1 ……;
```

在该语句完成后，若继续执行此语句，将替换上一个示例中创建的 tr1 触发器。此时，要替换已创建的 tr1 触发器，您必须是此触发器的所有者或持有 DBA 权限的才能成功执行此语句。

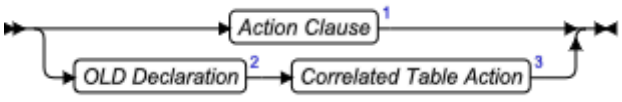
## 定义触发器事件和操作

此语法定义表上或视图上的触发器的事件和操作。

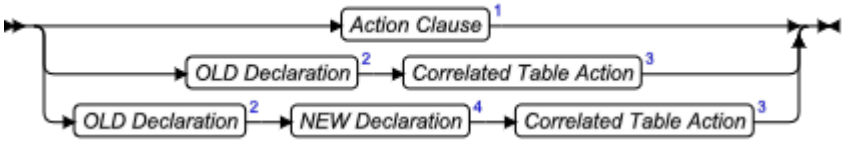
表上的触发器



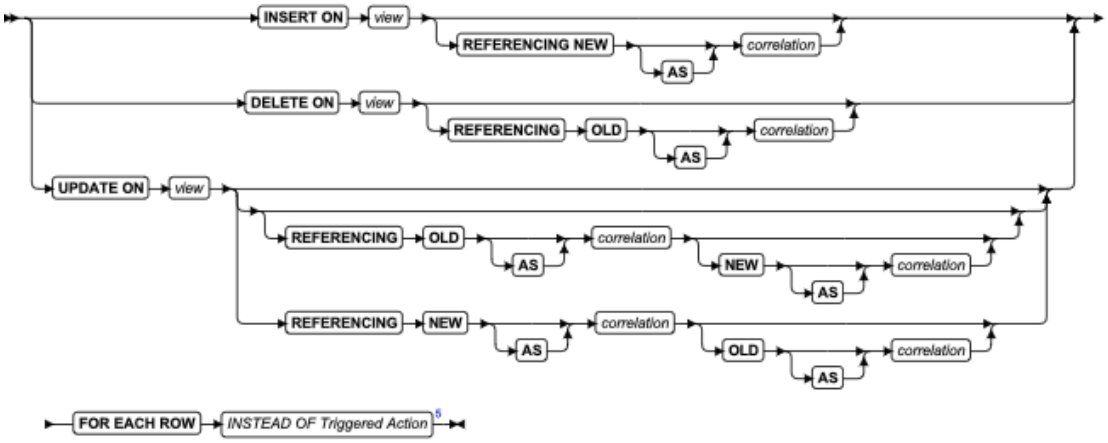

DELETE 和 SELECT 子子句




UPDATE 子子句




视图上触发器




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| column | 触发*表*中的列名称 | 必须存在 | 标识符 |
| correlation | 您此处声明的在触发操作中限定的旧的或新的列值（ *correlation.column*） | 在此触发器中必须是唯一的 | 标识符 |
| *table, view* | 触发表或视图的名称或同义词。*table* 或 *view* 可以包含 *owner.* 限定符。 | 必须存在于当前数据库中 | 标识符 |

主图表（包含*表*或*视图*）的左侧部分定义*触发器事件*（有时称为*触发事件*）。图表的剩余部分声明相关名称并定义*触发器操作*（有时称为*触发器操作*）。（对于表上的触发器，请参阅 Action 子句 和相关的表操作 。有关视图上的 INSTEAD OF 触发器，请参阅 INSTEAD OF 触发器的 Action 子句。）

## 触发器上的限制

要在表上创建触发器（或在视图上创建 INSTEAD OF 触发器），您必须拥有表或视图，或具有 DBA 状态。关于触发器所有者特权和其它用户特权之间的关系，请参阅执行触发操作的特权。

您创建触发器的表必须存在于当前数据库中。您不能在任一以下类型的表上创建触发器：

- 诊断表、违例表或另一个数据库中的表
- 临时表或系统目录表
- CREATE EXTERNAL TABLE 或 CREATE SEQUENCE 语句创建的表对象。

在 DB-Access 中，如果您想将触发器定义为模式的一部分，则请将 CREATE TRIGGER 语句放在 CREATE SCHEMA 语句中。

如果您正将 CREATE TRIGGER 语句嵌入到 GBase 8s ESQL/C 程序中，则您不能在触发器定义中使用主变量。

可以使用 DROP TRIGGER 语句移除现有的触发器。如果使用 DROP TABLE 或 DROP VIEW 语句来从数据库除去触发表或视图，则那些表或视图上的所有触发器也被删除。

当从触发器示例中或从触发器的 Action 子句或 Correlated Action 子句发出 SPL 的 ON EXCEPTION 语句时，该语句不会生效。

触发 BIGSERIAL 、 SERIAL 或 SERIAL8 列递增的 Insert 触发器的触发操作不会更改 SQL 通信区域结构的 **sqlca.sqlerrd[1]** 字段。该触发 INSERT 操作可以成功增加该列的序列计数，但是 **sqlca.sqlerrd[1]** 字段的值仍为零，而不会重置为新的序列值。

不能在指定了 ON DELETE CASCADE 引用约束的表上定义 DELETE 触发器。

UNION 子查询不能是触发事件。如果一个有效的 UNION 子查询指定了 Select 触发器定义的列，该查询成功，但是会忽略此触发器（或者视图上的 INSTEAD OF 触发器）。

数据库服务器不会对一些触发器使用并行处理。对于与触发事件类型相对应的任何 DML 语句，PDQ 在 FOR EACH ROW 部分中自动禁用：

- 在 Select 触发器的 Action 子句中的 SELECT 语句
- 在 Delete 触发器的 Action 子句中的 DELETE 或 MERGE 语句
- 在 Insert 触发器的 Action 子句中的 INSERT 或 MERGE 语句
- 在 Update 触发器的 Action 子句中的 UPDATE 或 MERGE 语句。

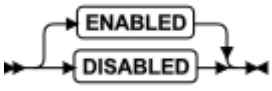
对 PDQ 的此限制的作用域是 FOR EACH ROW 部分。它在 Action 子句的 BEFORE 或 AFTER 部分中的 DML 语句没有作用。

有关视图上 INSTEAD OF 触发器其它限制，请参阅视图上 INSTEAD OF 触发器的限制 。

## 触发器方式

您可以在创建触发器时将触发器方式设置为启用或禁用触发器。

触发器方式




您可以以 ENABLED 或 DISABLED 方式在表或视图上创建触发器。

- 当以 ENABLED 方式创建触发器时，如果发生触发事件，则数据库服务器执行触发操作。（如果在创建触发器时不指定任何方式，则 ENABLED 是缺省方式。）
- 当以 DISABLED 方式创建触发器时，触发事件不会导致执行触发操作。实际上，数据库服务器将忽略该触发器及其操作。即使 **systriggers** 系统目录表维护有关已禁用触发器的信息。

您也可以使用 Database Object Mode 语句的 SET TRIGGERS 选项将现有的触发器设置为 ENABLED 或 DISABLED 方式。

通过 SET TRIGGERS 语句启用 DISABLED 触发器后，当发生触发事件时，数据库服务器可以执行触发操作，但是触发器不逆向执行。对于禁用触发器后和启用触发器前这段时间内选择、插入、删除或更新的行，数据库服务器将不试图为其执行触发。

**警告：** 由于触发器的行为根据其 ENABLED 或 DISABLED 方式而不同，因此禁用触发器时请小心。如果禁用触发器将最终破坏数据库的语义完整性，则请不要禁用触发器。

## 表层级结构中的触发器继承

缺省情况下，任何您定义在 GBase 8s 的类型表上的触发器都会被它的子表继承。

在此版本的 GBase 8s 中，一个表可以继承多个由同一的触发事件启用的触发器，因此这些触发器都是为子表上的相同类型的事件所定义的。

在所有版本的 GBase 8s 中，您在子表上设置的触发器会被它的依赖表继承，但是不会对它的超级表起作用。

当您需要在超级表上启用触发器，但不在其的子表上禁用时，该行为十分重要。在此版本中，禁用在表层次结构中的表上的触发器不会影响继承触发器。例如，以下语句对在表层次结构中低于或高于 *subtable* 的表对象上的触发器没有影响：

SET TRIGGERS FOR ***subtable*** DISABLED

类似地，DROP TRIGGER 语句不能除去继承的触发器，而不移除超级表上的触发器。在这种情况下，您必须改为在子表上定义一个没有 Action 子句的触发器。因为触发器未启用，此空的触发器会覆盖继承的触发器并对子表和子表下的任何子表执行，这些子表不需要进一步覆盖。

## 触发器和 SPL 例程

您不能如在数据操纵语言语句中列出的那样，在 DML（数据操纵语言）语句中调用的 SPL 例程中定义触发器。因此，如果 sp_items 过程包含 CREATE TRIGGER 语句，则以下语句返回错误：

INSERT INTO items EXECUTE PROCEDURE sp_items;

您不能将 SQL 的 CREATE FUNCTION 或 CREATE PROCEDURE 语句与 REFERENCING 子句一起使用来定义包含 FOR ***table*** 或 FOR ***view*** 规范的***触发器例程***。这些 UDR 必须包含在指定的***表***或***视图***中为 OLD 或 NEW 列值声明的 ***correlation*** 名称的 REFERENCING 子句。***表***或***视图***上的触发器可以调用来自 Triggered Action 列表中 FOR EACH ROW 部分的触发器例程。触发器还可以调用来自 Triggered Action 列表 BEFORE 和 AFTER 部分的非触发器例程，但是这些 UDR 不能使用 ***correlation*** 名称来引用 NEW 或 OLD 列值。正如 REFERENCING 子句 中描述的那样，触发器例程中的 REFERENCING 子句支持与 CREATE TRIGGER 语句相同的语法。

由同一触发事件执行的多个触发器可调用多个触发例程，并且这些例程可以通过使用具有相同或不同名称的 SPL 变量访问同一 NEW 或 OLD 列值。当一个触发事件执行多个触发器时，不会授权执行的顺序时，但是所有的 BEFORE 触发器操作都会在 FOR EACH ROW 触发操作之前执行，而所有的 AFTER 触发操作会在所有的 FOR EACH ROW 触发操作后执行。

对于不是触发器例程的 UDR ，SPL 变量在 CREATE TRIGGER 语句中无效。触发器调用的 SPL 例程只能在当前数据库的本地表或本地视图上执行 INSERT 、DELETE 或 UPDATE 操作。另请参阅 SPL 例程的规则以获取关于触发操作中调用的 SPL 例程的附加限制的信息。

## 触发事件

***触发器事件***指定哪个 DML 语句可以启动触发器。事件可以是***表***或***视图***上的 INSERT 、DELETE 或 UPDATE 操作，或是查询表的 SELECT 操作。每个 CREATE TRIGGER 语句必须指定一个触发器事件。作为触发事件的实例的任何 SQL 语句都称为***触发语句***。

对于每个***表***，您只能定义一个由 INSERT 、DELETE 、UPDATE 或 SELECT 语句的激活的触发器。对于每个***视图***，您可以定义定义由 INSERT 、DELETE 或 UPDATE 语句激活的 INSTEAD OF 触发器。同一表或视图上的多个触发器可以被不同类型的触发器事件或同一类型的触发器事件激活。

如果触发表具有指定 ON DELETE CASCADE 的引用约束，则您不能指定 DELETE 事件。

您负责确保触发语句在表上有或没有触发器操作时都返回相同的结果。另请参阅 Action 子句 触发操作部分。

来自外部数据库服务器的触发语句可以激活触发器。

如以下示例所示，**newtab** 上的 Insert 触发器（由 **dbserver1** 管理）由来自 **dbserver2** 的 INSERT 语句激活。该触发器就像在 **dbserver1** 上生成的 INSERT 那样执行。

```
-- Trigger on stores_demo@dbserver1:newtab
        CREATE TRIGGER ins_tr INSERT ON newtab
        REFERENCING new AS post_ins
        FOR EACH ROW(EXECUTE PROCEDURE nt_pct (post_ins.mc));
        -- Triggering statement from dbserver2
        INSERT INTO stores_demo@dbserver1:newtab
        SELECT item_num, order_num, quantity, stock_num, manu_code,
        total_price FROM items;
```

GBase 8s 也支持视图上的 INSTEAD OF 触发器。这些触发器在一个触发 DML 操作引用该指定视图时被启动。INSTEAD OF 触发器使用视图上指定的触发器操作替换该触发器操作事件，而不是执行触发 INSERT 、DELETE 或 UPDATE 操作。对于每种事件类型（INSERT 、DELETE 或 UPDATE），***视图***可以至多定义多个 INSTEAD OF 触发器。

## 带游标的触发事件

对于表上的触发器，如果触发器语句使用游标，则触发操作的每个部分（包括 BEFORE 、FOR EACH ROW 和 AFTER ，如果为该触发器指定了的话）都为游标除了的每行激活。

此行为与触发器语句不使用游标且更新多个行时的情况有所不同。在此情况中，任何由 BEFORE 和 AFTER 触发的操作都只执行一次，但是 FOR EACH ROW 操作列表对于触发语句处理的每一行都执行。关于触发操作的其它信息，请参阅 Action 子句。

## 触发事件上的特权

要将触发 INSERT 、DELETE 、UPDATE 或 SELECT 语句执行为触发事件，您必须在触发表或视图上具有相应的 Insert 、Delete 、Update 或 Select 特权。但是，如果您还不具有指定触发操作中某个 SQL 语句所需的特权的话，触发语句可能仍会失败。当执行触发操作时，数据库服务器对于触发器定义中的每个 SQL 语句都检查您的特权，就像触发器的每个语句都被独立执行那样。关于执行触发操作所需的特权的信息，请参阅执行触发操作的特权 。

## 触发器的性能影响

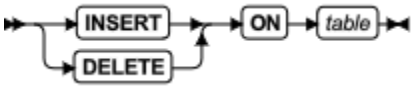
启动触发器的 INSERT 、DELETE 、UPDATE 和 SELECT 语句可能看起来执行得比较慢，因为它们激活附加的 SQL 语句，且用户可能不知道正在发生其它操作。

触发事件的执行时间取决于触发操作的复杂度以及它是否启动其它触发器。随着级联触发器数目增加，时间也会增加。关于启动其它触发器的触发器的更多信息，请参阅级联触发器。

## INSERT 事件和 DELETE 事件

表上的 INSERT 和 DELETE 事件由那些关键字和 ON *table* 子句定义，使用以下语法。

在表上 INSERT 或 DELETE 事件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *table* | 触发表的名称 | 必须在数据库中存在 | 标识符 |

当 INSERT 语句在其 INTO 子句中包含指定的 ***table*** （或 ***table*** 的同义词）时， Insert 触发被激活。同样地，当 DELETE 语句在其 FROM 子句中包含指定的 ***table***（或 ***table*** 的同义词）时，Delete 触发器被激活。

如果 Insert 触发器指定的 ***table*** 是包含 Insert 子句的 MERGE 语句的目标表，则 MERGE 语句也可以激活 Insert 触发器，同样地，如果 Delete 触发器指定的 ***table*** 是包含 Delete 子句的 MERGE 语句的目标表，则 MERGE 语句也可以激活 Delete 触发器。

当 TRUNCATE TABLE 语句删除表的所有行时，它不会激活 Delete 触发器。如果要对一个表定义启用的 Delete 触发器，而您不具有此表的 Alter 特权，则如果您视图移除此表，即使 TRUNCATE 语句不是 Delete 触发器的触发事件，数据库服务器仍会返回错误。（有关删除操作所需的自由访问权的更多信息，请参阅 TRUNCATE 语句。）

对于视图上的触发器，INSTEAD OF 关键字必须紧跟在指定触发事件类型的 INSERT 、DELETE 或 UPDATE 关键字之前，且***视图***（而不是表）的名称或同义词必须跟在 ON 关键字之后。视图上的 INSTEAD OF 触发器部分描述定义 INSTEAD OF 触发事件的语法。

一个表上可以定义多个 Insert 触发器和多个 Delete 触发器。

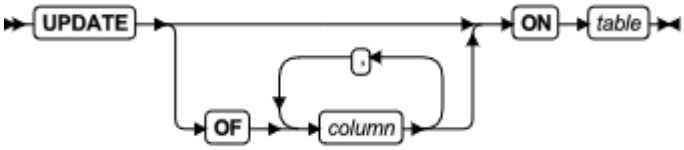
如果您在表层次结构中的子表上定义触发器，且子表支持级联删除，则超级表上的 DELETE 操作将激活子表上的 Delete 触发器。

关于 Insert 触发器和 Delete 触发器的操作上的相关性和限制的信息，另请参阅触发器的再进入一节。

## UPDATE 事件

UPDATE 事件（和 SELECT 事件）可以包含可选的 *column* 列表、

UPDATE 事件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 激发触发器的列 | 必须在触发表中存在 | 标识符 |
| *table* | 触发表的名称 | 必须数据库中存在 | 标识符 |

***column*** 列表是可选的。如果您忽略 OF ***column*** 列表，则更改 ***table*** 的任何列都将激活触发器。

OF ***column*** 子句对于视图上的 INSTEAD OF 触发器是无效的。

在两种情况下，触发表上的 UPDATE 可以激活触发器：

- UPDATE 语句引用 ***column*** 列表中的任何列。
- UPDATE 事件定义未指定 OF ***column*** 列表规范。

无论它更新 ***column*** 列表中的一列还是激活多列，触发 UPDATE 语句都只激活 Update 触发器一次。

如果指定没有 ***columns*** 列的触发器 ***table*** 是 MERGE 语句的目标表，或者如果 MERGE 语句的 Update 子句引用了 Update 触发器的 ***column*** 列表中的列，则 MERGE 语句也可以激活 Update 触发器。

## 定义多个 Update 触发器

一个表上的多个 Update 触发器不能包含相同的列。在以下示例中，trig3 在 items 表上是无效的，因为它的列列表包含 stock_num ，而 stock_num 是 trig1 中的触发列。

```
CREATE TRIGGER trig1 UPDATE OF item_num, stock_num ON items
        REFERENCING OLD AS pre NEW AS post
        FOR EACH ROW(EXECUTE PROCEDURE proc1());
  CREATE TRIGGER trig2 UPDATE OF manu_code ON items
        BEFORE(EXECUTE PROCEDURE proc2());
  CREATE TRIGGER trig3 UPDATE OF order_num, stock_num ON items
```

```
        BEFORE(EXECUTE PROCEDURE proc3());
```

当 UPDATE 语句尝试更新具有不同触发器的多个列时，触发列的列编号确定触发执行的顺序。从编号最小的触发列开始执行，且按顺序一直执行到编号最大的触发列。但是，如果在同一列或同一组列上设置了多个 Update 触发器，则不保证触发器的执行顺序。

以下示例显示表 taba 具有四列（a 、 b 、 c 、 d）：

```
CREATE TABLE taba (a int, b int, c int, d int);
```

将 trig1 定义为列 a 和 c 上的更新，将 trig2 定义为列 b 和 d 上的更新，如以下示例所示：

```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
        AFTER (UPDATE tabb SET y = y + 1);

CREATE TRIGGER trig2 UPDATE OF b, d ON taba
        AFTER (UPDATE tabb SET z = z + 1);
```

以下示例显示 Update 触发器的触发语句：

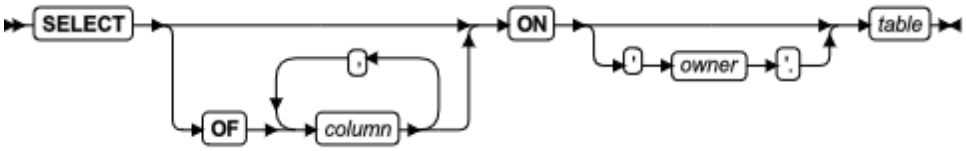
```
UPDATE taba SET (b, c) = (b + 1, c + 1);
```

然后列 **a** 和 **c** 的 **trig1** 首先执行，列 **b** 和 **d** 的 **trig2** 随后执行。在此情况中，触发器中列编号最小的是第 1 列（**a**），第二小的是第 2 列（**b**）。

## SELECT 事件

DELETE 和 INSERT 事件由那些关键字（和 ON *table* 子句），但是 SELECT 和 UPDATE 事件还支持可选的 *column* 列表。

SELECT 事件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 激活触发器的列 | 必须在触发 *table* 中存在 | 标识符 |
| *owner* | *table* 的所有者 | 必须拥有 *table* | 所有者名称 |
| *table* | 触发表的名称 | 必须在数据库中存在 | 标识符 |

如果您在同一个表上定义多个 Select 触发器，则每个触发器的 ***column*** 列表可以是唯一的或者是另一个 Select 触发器的重复。

在这两种情况下，触发表上的 SELECT 可以激活触发器：

- SELECT 语句引用 ***column*** 列表中的任何列。

- SELECT 事件定义未指定 OF ***column*** 列表规范。

（但是，接下来的部分描述可能影响 SELECT 语句是否激活 Select 触发器的其它情况。）

无论它指定 ***column*** 列表中的一列还是激活多列，触发 SELECT 语句都只激活 Select 触发器一次。

Select 触发器的操作不能在触发表上包含 UPDATE 、INSERT 或 DELETE 。Select 触发器的操作可以在不是触发表的其它表上包含 UPDATE 、INSERT 或 DELETE 操作。以下示例在表的一列上定义 Select 触发器：

```
CREATE TRIGGER mytrig
        SELECT OF cola ON mytab REFERENCING OLD AS pre
        FOR EACH ROW (INSERT INTO newtab VALUES('for each action'));
```

您不能对视图上的 INSTEAD OF 触发器指定 SELECT 事件。

## Select 触发器被激活时的情况

在这些情况中，在触发表上的查询激活 Select 触发器：

- SELECT 语句是独立的 SELECT 语句。
- SELECT 语句在选择列表中调用 UDR 中发生。
- SELECT 语句是 Projection 列表中的子查询。
- SELECT 语句是 FROM 子句中的子查询。
- SELECT 语句在 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 调用的 UDR 中发生。
- SELECT 语句从表层结构中的超级表选择数据。在此情况中，SELECT 语句激活层次结构中的超级表和所有子查询的 Select 触发器。

有关不激活 Select 触发器的 SELECT 语句的信息，请参阅 Select 触发器未激活时的情况。

## 独立 SELECT 语句

如果触发列出现在 SELECT 语句的 Projection 子句的选择列表中，则 Select 触发器被激活。

例如，如果 Select 触发器被定义为当表 tab1 的列 col1 被选择时执行，则以下两个独立 SELECT 语句都激活 Select 触发器：

```
SELECT * FROM tab1;
SELECT col1 FROM tab1;
```

## 选择列表中的 UDR 中的 SELECT 语句

如果 UDR 在其语句块中包含 SELECT 语句，则 Select 触发器被 UDR 激活，且 UDR 还显示在 SELECT 语句的 Projection 子句的选择列表中。例如，假设名为 my_rtn 的 UDR 在其语句块中包含此 SELECT 语句：

```
SELECT col1 FROM tab1;
```

限制假设以下 SELECT 语句在其选择列表中调用 my_rtn UDR ：

```
SELECT my_rtn() FROM tab2;
```

当执行 **my_rtn** UDR 时，该 SELECT 语句激活表 **tab1** 的列 **col1** 上定义的 Select 触发器。

## EXECUTE PROCEDURE 和 EXECUTE FUNCTION Call 的 UDR

如果 UDR 在其语句块中包含 SELECT 语句且 UDR 被 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句调用，则 Select 触发器被 UDR 激活。例如，假设名为 my_rtn 的用户定义过程在其语句块中包含以下 SELECT 语句：

```
SELECT col1 FROM tab1;
```

限制假设以下语句调用 my_rtn 过程：

```
EXECUTE PROCEDURE my_rtn();
```

当语句块中的 SELECT 语句被执行时，该语句激活表 **tab1** 的列 **col1** 上定义的 Select 触发器。

## 选择列表中的子查询

Select 触发器可以被 SELECT 语句的 Projection 子句的选择列表中出现的子查询激活。

例如，如果 Select 触发器在 tab1 的 col1 上定义，则以下 SELECT 语句中的子查询激活该触发器：

```
SELECT (SELECT col1 FROM tab1 WHERE col1=1), colx, col y FROM tabz;
```

## SELECT 的 FROM 子句中的子查询

SELECT 的 FROM 子句中的表表达式可以是被不相关子查询引用的表上的触发事件。在以下示例中，指定一个表表达式的子查询是定义在 **tab1** 的 **col1** 上的 Select 触发器的触发事件：

```
SELECT vcol FROM (SELECT FIRST 5 col1 FROM tab1 ORDER BY col1 ) vtab(vcol);
```

## DELETE 或 UPDATE 的 WHERE 子句中的子查询

用 DELETE 语句 或 UPDATE 语句的 WHERE 子句中的子查询语法使用 Condition 的子查询不能是 Select 触发器的触发事件。在以下示例中，该子查询不是定义在 **tab1** 的 **col2** 上的 Select 触发器的触发事件：

```
DELETE tab1 WHERE EXISTS
        (SELECT col2 FROM tab1 WHERE col2 > 1024);
```

但是，在同一示例的 DELETE 操作，激活定义在 tab1 上的 Delete 触发器。tbl1 上的 Select 触发器不会被通过修改子查询的 FORM 子句中引用的表的 DELETE 语句中的子查询激活。

类似地，以下语句中的 WHERE 子句的子查询不是定义在 **tab1** 的 **col3** 上的 Select 触发器的触发事件：

```
UPDATE tab1 SET col3 = col3 + 10
        WHERE col3 > ANY
        (SELECT col3 from tab1 WHERE col3 > 1);
```

相同的示例会激活定义在 tbl 的 col3 上的 Update 触发器，但是此子查询不会更改 Select 触发器。有关 Select 触发器的其它限制，请参阅 Select 触发器未激活时的情况。

## 表层次结构中的 Select 触发器

GBase 8s 数据库中的子表继承超表上定义的 Select 触发器。当您从超级表中选择时， SELECT 语句激活超级表上的 Select 触发器以及表层次结构中的子表上被继承的 Select 触发器。

例如，假设表 **tab1** 是超级表且表 **tab2** 是表层次结构中的子表。如果 Select 触发器 **trig1** 在表 **tab1** 上定义，则表 **tab1** 上的 SELECT 语句为表 **tab1** 中的各行激活 Select 触发器 **trig1**，并为表 **tab2** 中的各行激活继承的 Select 触发器 **trig1** 。

如果您将 Select 触发器添加到子表，则该 Select 触发器不会覆盖该子表从其超表继承的 Select 触发器，但是会增加子表上 Select 触发器的数量。例如，如果 Select 触发器 **trig1** 在超级表 **tab1** 中的列 **col1** 上定义，则子表 **tab2** 继承此触发器。如果您在子表 **tab2** 的列 **col1** 上定义一个名为 **trig2** 的 Select 触发器，和一个来自超表 **tab1** 的 **col1** 列的 SELECT 语句，则此 SELECT 语句为表 tab1 中的各行激活触发器 trig1 ，且为表 tab2 中的各行激活触发器 trig1 和 trig2 。

## Select 触发器未激活时的情况

在某些情况下，触发表上的 SELECT 语句不激活 Select 触发器：

- 如果包含 SELECT 语句的子查询或 UDR 出现在除 FROM 子句或 Projection 子句的 SELECT 语句的其它任意子句中，则 Select 触发器不被激活。
- 例如，如果子查询或 UDR 出现在 SELECT 语句的 WHERE 子句或 HAVING 子句中，则子查询或 UDR 中的 SELECT 语句不激活 Select 触发器。
- 如果 Select 触发器的触发操作调用包含触发 SELECT 语句的 UDR ，则 UDR 中 SELECT 上的 Select 触发器不被激活。不支持级联选择触发器。
- 如果 SELECT 语句在其 Projection 子句中包含内置聚集或用户定义的聚集，则 Select 触发器不被激活。例如，以下 SELECT 语句不会激活 **tab1** 的 **col1** 上定义的 Select 触发器：
- SELECT MIN(col1) FROM tab1;
- 包含集合运算符（包括 INTERSECT 、MINUS 、EXCEPT 、UNION 或 UNION ALL）的 SELECT 语句不会激活 Select 触发器。
- INSERT 的 SELECT 子句不激活 Select 触发器。
- DELETE 或 UPDATEA 语句的 WHERE 子句中的子查询不会激活 DELETE 或 UPDATE 语句正在更新的同一表上的 Select 触发器。
- 如果 SELECT 的 Projection 子句包含 DISTINCT 或 UNIQUE 关键字，则 SELECT 语句不会激活 Select 触发器。
- 滚动游标上不支持 Select 触发器。
- 如果 SELECT 语句引用远程触发表，则 Select 触发器在远程数据库服务器上不被激活。
- 查询的 ORDER BY 列表中的列不激活 Select 触发器（也不激活任何其它触发器），除非它们也列在 Projection 子句中。

最后一条限制的例外是 Select 触发器可以被 FROM 子句的子查询列表中的 ORDER BY 列表中的列激活，无论该相同的列是否出现在 Projection 子句中。在以下示例中，在 ORDER BY 子句中包含 col1 的表表达式（而不是在 Projection 子句中的选择列表的表表达式）是 tab1 的 col1 上定义的 Select 触发器的触发事件：

SELECT vcol FROM (SELECT col2 FROM tab1 ORDER BY col1 ) vtab(vcol);

## Action 子句

Action 子句定义触发器被激活时要执行的 SQL 语句。对于表上的触发器，在 Action 子句中可以有三个部分：BEFORE 、AFTER 和 FOR EACH ROW。

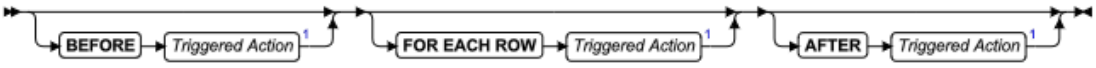
- 在数据库服务器执行触发 DML 操作之前，BEFORE 操作会对每个触发事件执行一次。
- 在触发语句的上下文中，表上的操作执行完毕后，AFTER 操作也会对每个触发 DML 事件执行一次。
- 会对 DML 操作中的插入、更改、删除或选择的每一行执行 FOR EACH ROW 操作，在每一行上的 DML 操作执行之后，但是在数据库服务器将数据写入日志和表之前。

如果一个表具有被同一触发事件激活的多个触发器，则不保证执行触发的顺序，但是所有的 BEFORE 触发操作都在任何 FOR EACH ROW 触发操作之前执行，并且所有的 AFTER 触发操作在 FOR EACH ROW 触发操作后执行。

当您在视图上定义 INSTEAD OF 触发器时，不支持 BEFORE 和 AFTER 关键字，但是 Action 子句的 FOR EACH ROW 部分是有效的。有关在视图上指定触发操作的语法，请参阅视图上的 INSTEAD OF 触发器章节。

Action 子句具有以下语法。

Action 子句




要使触发器对表产生作用，则您必须定义至少一个触发操作，使用 BEFORE 、FOR EACH ROW 或 AFTER 关键字指示何时发生相对于触发事件的执行的操作。

您可以在单个触发器上为这三个选项中的任何一个或所有三个指定操作，但是必须首先指定任何 BEFORE 操作列表，并且最后指定任何 AFTER 操作列表。有关当 REFERENCING 子句也被指定时的 Action 子句的更多信息，请参阅相关的表操作。

## BEFORE 操作

在触发语句执行之前，BEFORE 触发器操作列表只执行一次。即使触发语句不处理任何行，数据库服务器仍然执行 BEFORE 触发器操作。

## FOR EACH ROW 操作

处理触发表的一行后，数据库服务器执行 FOR EACH ROW 触发器操作列表的所有语句；此循环对触发语句处理的每一行重复执行。（但是如果触发语句不插入、删除、更新或选择任何行，则数据库服务器不执行 FOR EACH ROW 触发操作。）

Select 触发器的 FOR EACH ROW 操作列表对于行的每个实例执行一次。例如，同一行可以在结合两个表的查询结果中出现多次。有关引用触发语句过程中指定的值的 FOR EACH ROW 操作的更多信息，请参阅 REFERENCING 子句 。

正如触发器上的限制中所述的那样，对于对应于触发器事件类型的 DM 语句，并行数据的处理在 FOR EACH ROW 触发器操作中是禁用的。例如，数据库服务器不会在 Update 触发器的 Action 子句的 FOR EACH ROW 部分的 UPDATE 语句中应用 PDQ ，也不会在 Delete 触发器的 Action 子句的 FOR EACH ROW 部分的 DELETE 语句中应用。PDQ 处理过程上的限制不适用于 Action 子句的 BEFORE 或 AFTER 部分的 DML 语句。

## AFTER 操作

触发语句的操作完成后，指定的 AFTER 触发器操作执行一次。如果触发器语句不处理任何行，则 AFTER 触发器操作仍将执行。

## 多个触发器的操作

当 UPDATE 或 MERGE 语句激活多个触发器时，触发操作合并。假设 taba 具有列 a 、b 、c 和 d，如该示例所示：

```
CREATE TABLE taba (a INT, b INT, c INT, d INT);
```

接下来。假设您在列 a 和 c 上定义 trig1，在列 b 和 d 上定义 trig2 。如果两个触发器都指定 BEFORE 、FOR EACH ROW 和 AFTER 操作，则按以下顺序执行触发器操作：

1. 触发器的 BEFORE 操作列表（**a** 、 **c**）
2. 触发器的 BEFORE 操作列表（**b** 、**d**）
3. 触发器的 FOR EACH ROW 操作列表（**a** 、**c**）
4. 触发器的 FOR EACH ROW 操作列表（**b**、 **d**）
5. 触发器的 AFTER 操作列表（**a** 、 **c**）
6. 触发器的 AFTER 操作列表（**b** 、 **d**）

数据库服务器将由同一个触发语句激活的所有触发器视为单个触发器。且触发操作是合并操作列表。控制触发器操作的所有规则适用于作为单个列表的合并列表，且两个原始触发器之间没有区别。

### 确保行顺序的独立性

在 FOR EACH ROW 触发操作列表中，结果可能取决于正在处理的行的顺序。通过以下建议，您可以确保结果是独立于行顺序的：

- 避免在 FOR EACH ROW 部分中选择触发表。

  如果触发语句影响触发表中的多个行，则 FOR EACH ROW 部分中的 SELECT 语句的结果随着处理的每一行而变化。该条件也适用于任何级联触发器。请参阅级联触发器。
- 在 FOR EACH ROW 部分中，避免使用从触发表的当前行得到的值更新表。

如果触发操作多次修改表中的任何行，则该行的最终结果取决于触发表中行处理的顺序。

- 避免在同一个 FOR EACH ROW 触发操作（包括任何级联触发操作）中的另一个语句选择的 FOR EACH ROW 部分中修改表。

如果 FOR EACH ROW 操作修改表，则当触发器随后的操作引用表时，该更改可能不完整。在此情况中，结果可能不同，这取决于处理行的顺序。

数据库服务器不实施规则以避免这些情况，因此如果这样做或限制触发操作可以选择的表的集合。而且，大多数触发操作的结果是独立于行顺序的。因此，您负责确保触发操作的结果独立于行顺序。

## REFERENCING 子句

任何事件的 REFERENCING 子句声明可以用于触发表中限定列值的*相关*名（对于 Update 触发器，两个*相关*名称）。这些名称启用 FOR EACH ROW 操作以引用触发事件结果中的新值。

它们还启用 FOR EACH ROW 操作以引用触发事件修改前触发表中存在的旧列值。

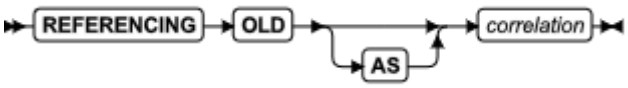
如果该触发操作同时包括了 INSERT 语句和 BEFORE WHEN 或 AFTER WHEN 关键字，则相关名无效。此限制对指定 FOR EACH ROW 关键字不含 BEFORE 或 AFTER 关键字或者不包含 INSERT 语句的触发操作没有影响。

此处为 CREATE TRIGGER 语句描述的 REFERENCING 子句语法在定义例程的 CREATE FUNCTION 和 CREATE PROCEDURE 语句中是可用的，它提供 CREATE FUNCTION 或 CREATE PROCEDURE 语句也包括 FOR ***table_object*** 子句以指定表或视图的 FOR EACH ROW 操作可以调用触发例程。

## 用于删除的 REFERENCING 子句

Delete 触发器的 REFERENCING 子句可以为列中要删除的值声明相关名称。

用于删除的 REFERENCING 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 此处为满足在触发器操作中使用的旧列值声明的名称（ *correlation.column*） | 在此 CREATE TRIGGER 语句中必须唯一 | 标识符 |

***correlation*** 是在触发语句执行前，用于触发表中列值的限定符。***correlation*** 在 FOR EACH ROW 触发操作列表中的作用域中。请参阅相关的表操作。

要在触发操作中使用相关名称以引用旧的列值，请以相关名称和句号（.）作为列名的前缀。例如，如果 NEW ***correlation*** 是 **post** ，请将列 **fname** 的新值引用为 **post.fname**。

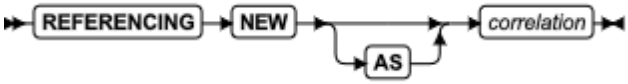
如果触发器事件是 DELETE 语句，则使用 new 相关名作为限定符会产生错误，因为咋 i 该行被删除后该列没有值。要了解控制使用相关名称的规则，请参阅在触发操作中使用相关名称。

只要您定义了 FOR EACH ROW 触发操作，就可以为删除使用 REFERENCING 子句。

## 用于插入的 REFERENCING 子句

Insert 触发器的 REFERENCING 子句可以为列中要插入的值声明相关名称。

用于插入的 REFERENCING 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 此处为满足在触发器操作中使用的新列值声明的名称（ *correlation.column*） | 在此 CREATE TRIGGER 语句中必须唯一 | 标识符 |

***correlation*** 是在执行触发语句后用于新列值的名称。其引用作用域仅限于 FOR EACH ROW 触发操作列表；请参阅相关的表操作。要使用相关名称，请在列名前面加上 ***correlation*** 名称和句号（ **.** ）。因此，如果 NEW ***correlation*** 名称是 **post** ，请将列 **fname** 的新值称为 **post.fname**。

如果触发事件是 INSERT 语句，则使用 old ***correlation*** 名称作为限定符会产生错误，因为插入行前不存在值。要了解控制如何使用相关性的规则，请参阅在触发操作中使用相关名称。只有您定义了 FOR EACH ROW 触发操作，就可以使用 INSERT REFERENCING 子句。

以下示例说明 INSERT REFERENCING 子句的用法。对于插入到 table1 中的每一行，该示例将一行插入到 backup_table1 中。插入到 backup_table1 的 col1 和 col2 中的值是刚插入到 table1 中的值的精确副本。

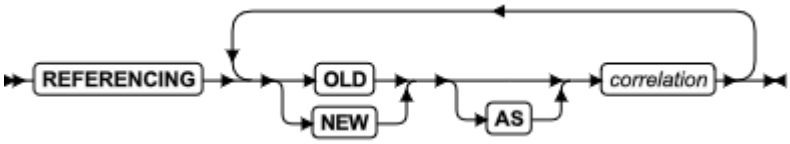
```
CREATE TABLE table1 (col1 INT, col2 INT);
        CREATE TABLE backup_table1 (col1 INT, col2 INT);
        CREATE TRIGGER before_trig
        INSERT ON table1      REFERENCING NEW AS new
        FOR EACH ROW
        (
        INSERT INTO backup_table1 (col1, col2)
        VALUES (new.col1, new.col2)
        );
```

如以上示例所示，INSERT REFERENCING 子句使您能够引用触发操作生成的数据值。

## 用于更新的 REFERENCING 子句

Update 触发器的 REFERENCING 子句可以为列中原始值和已更改的值声明相关名称。

用于更新的 REFERENCING 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 您在此为在触发器操作中使用的旧的或新的列值声明的名称（*correlation.column*） | 在此 CREATE TRIGGER 语句中必须唯一 | 标识符 |

OLD ***correlation*** 是执行触发语句前触发表中的列值的名称；NEW ***correlation*** 标识执行触发语句后的相应值。

您在此声明的 ***correlation*** 名称的引用作用域只限于 FOR EACH ROW 触发器操作列表中。请参阅相关的表操作。

要引用新的或旧的列值，请以 ***correlation*** 名称和句号（.）作为列名的前缀。例如，如果 new ***correlation*** 名称是 **post** ，您可以将列 **fname** 中的新值引用为 **post.fname**。

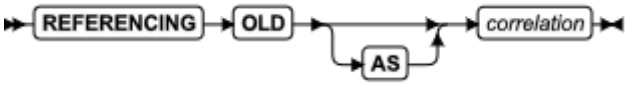
如果触发器事件是 UPDATE 语句，则您可以同时定义 old 和 new ***correlation*** 名称以引用触发 UPDATE 语句之前和之后的列值。要了解控制使用 ***correlation*** 名称的规则，请参阅在触发操作中使用相关名称。

只有您定义了 FOR EACH ROW 触发操作，就可以使用 UPDATE REFERENCING 子句。

## 用于选择的 REFERENCING 子句

Select 触发器的 REFERENCING 子句可以为列中的值声明相关名称。

用于选择的 REFERENCING 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 您在触发操作中此为旧或新列值声明的名称（*correlation.column*） | 在此 CREATE TRIGGER 语句中必须是唯一的 | 标识符 |

该子句具有与用于删除的 REFERENCING 子句相同的语法。您在此声明的 ***correlation*** 名称的引用作用域只限于 FOR EACH ROW 触发操作列表中。请参阅相关的表操作。

您可与通过在列名前加上相关名称和句号(.)使用 ***correlation*** 名称以引用 old 列值。例如，如果 old ***correlation*** 名称是 **pre** ，则您可以将列 **fname** 的旧值引用为 **pre.fname**。

如果触发事件是 SELECT 语句，则使用 new ***correlation*** 名称作为限定符会产生错误，因为在选择该列后该列不具有 new 值。要了解控制使用相关名称的规则，请参阅在触发操作中使用相关名称。

只有您定义了 FOR EACH ROW 触发操作，就可以使用 SELECT REFERENCING 子句。

## 相关的表操作

使用 Correlated Trigger Action 子句定义当触发事件激活表上的触发器时作为触发操作执行的 SQL 语句。

相关的表操作


如果 CREATE TRIGGER 语句包含 INSERT REFERENCING 子句、DELETE REFERENCING 子句、UPDATE REFERENCING 子句或者 SELECT REFERENCING 子句，则您必须在操作子句中包含 FOR EACH ROW 触发操作列表。您还可以包含 BEFORE 和 AFTER 触发操作列表，但是它们是可选的。

有关 BEFORE 、FOR EACH ROW 和 AFTER 触发操作列表的信息，请参阅 Action 子句。

## 触发操作

触发操作指定触发器被激活时，执行 SQL 语句的列表。Action 子句的 BEFORE 、FOR EACH ROW 和 AFTER 部分可以指定同一触发器的不同触发操作列表。

触发操作

对于表上的触发器。触发操作由可选的 WHEN 条件和操作语句构成。您可以为每个 WHEN 子句指定触发操作列表，或者如果您不包含 WHEN 子句的话可以指定单个列表（由一个或多个触发操作构成）。

当 CREATE TRIGGER 语句定义新的触发器时，在触发操作或触发事件的定义中显式引用的数据库对象（例如表、列和 UDR ）必须存在。

**注意：** 当您在 WHEN 条件中或操作语句中指定日期表达式时，请确保对年份指定四位数字而不是两位数字。有关缩写年份的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 中关于 **DBCENTURY** 的描述。该文档还描述了环境变量设置如何影响某些数据库对象的行为。与分片表达式、检查约束、和 UDR 类似，触发器与环境变量的创建时间设置一起存储在系统目录表中，环境变量可能影响诸如 WHEN 条件之类的表达式求值。当对那些数据库对象中的表达式求值时，数据库服务器忽略对这些设置的所有后续更改。

## WHEN 条件

WHEN 条件使触发操作依赖于测试的结果。当您在触发操作中包含 WHEN 条件时，仅当条件所得的值为 true 时，触发操作列表中的语句才执行。如果 WHEN 条件求值为 false 或 unknown，则触发操作列表中的语句不执行。

如果触发操作在 FOR EACH ROW 部分中，则其条件对每一行求值，例如，仅当 WHEN 子句为 true 时以下触发器中的触发操作才执行：

```
CREATE TRIGGER up_price
        UPDATE OF unit_price ON stock
        REFERENCING OLD AS pre NEW AS post
        FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
        (INSERT INTO warn_tab VALUES(pre.stock_num, pre.order_num,
        pre.unit_price, post.unit_price, CURRENT));
```

在 WHEN 条件内执行的 SPL 例程与在数据库操纵语句中调用的 UDR 具有相同的限制。也就是说，SPL 例程不能包含某些 SQL 语句。有关语句受限制的信息，请参阅在数据操纵语句中 SPL 例程的限制。

## 操作语句

触发操作语句可以是 INSERT 、DELETE 、UPDATE 、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句。如果操作列表包含多个语句，并且 WHEN 添加满足（或缺失），则这些语句按它们在列表中出现的顺序执行。

### *作为触发操作的 UDR*

用户定义的函数和过程包括触发例程可以是触发操作。FOR EACH ROW 子句的操作列表可以包含调用 **mi_trigger*( )** 函数的 UDR 。只有 GBase 8s 中的触发例程的上下文中的触发操作能被调用。有关触发例程的语法和调用内容的限制，请参阅 REFERENCING 和 FOR 子句。

您可以使用 EXECUTE FUNCTION 语句调用任何用户定义的函数或触发器函数。使用 EXECUTE PROCEDURE 语句调用热河用户定义的过程或触发器过程。

在 Boolean 表达式有效的上下文中，Boolean 运算符 SELECTING 、INSERTING 、DELETING 和 UPDATING 在触发例程中都有效，且在其它在触发操作语句调用的 UDR 中也是有效的。如果触发事件与符合运算符名称的 DML 操作相匹配，则这些运算符返回 TRUE('t')；否则返回 FALSE('f')。一个触发例程可以设计成为不同种类的触发事件执行不同触发操作，使用这些 Boolean 运算符执行与触发器类型相适合的程序块。

有关使用 SPL 例程作为触发操作的限制，请参阅 SPL 例程的规则和触发器和 SPL 例程。

### *实现一致性结果*

要保证触发语句对于有触发操作或没有时都返回相同的结果，请确保 BEFORE 和 FOR EACH ROW 部分中的触发操作不修改以下子句中引用的任何表：

- WHERE 子句

- UPDATE 语句中的 SET 子句
- SELECT 子句
- 多行 INSERT 语句中的 EXECUTE PROCEDURE 子句或 EXECUTE FUNCTION 子句 。

### *声明 SQL 的关键字作为相关名称*

如果您在触发操作列表中的以下任何子句中使用 INSERT 、DELETE 、UPDATE 或 EXECUTE 关键字作为 *correlation* 标识符，则必须使用***所有者***名称和/或***表***名称限定它们。

- SELECT 语句的 FROM 子句
- EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句的 INTO 子句
- GROUP BY 子句
- UPDATE 语句的 SET 子句

当您在触发操作中包含这些关键字时，如果这些关键字***未***被限定，则会产生语法错误。

如果您使用关键字作为列名，则必须使用表名限定它；例如 **table.update**。如果表名称和列名称都是关键字，则必须使用所有者名称限定它们（例如， ***owner*.insert.update**）。如果所有者名称、表名称和列名称都是关键字，则必须用引号将所有者名称引起来；例如 **'delete'.insert.update**。（这些是将保留字作为标识符的一般规则，而不是触发器的特例。如果避免使用 SQL 的关键字作为标识符，则您的代码会更易阅读和维护。）

唯一的例外是这些关键字是列表汇总的第一个表或列名称时，且您没有限定它们。例如，不需要限定以下语句中的 `delete` ，因为它是 INTO 子句列出的第一列：

```
CREATE TRIGGER t1 UPDATE OF b ON tab1
        FOR EACH ROW (EXECUTE PROCEDURE p2() INTO delete, d);
```

在以下语句显示的示例中，您必须限定列名称或表名称：

- SELECT 语句的 FROM 子句

  ```
  CREATE TRIGGER t1 INSERT ON tab1
     BEFORE (INSERT INTO tab2 SELECT * FROM tab3, 'owner1'.update);
  ```

- EXECUTE PROCEDURE 语句的 INTO 子句

  ```
  CREATE TRIGGER t3 UPDATE OF b ON tab1
        FOR EACH ROW (EXECUTE PROCEDURE p2() INTO
              d, tab1.delete);
  ```

  视图上的 INSTEAD OF 触发器在其已触发的操作中不能包含 EXECUTE PROCEDURE INTO 语句。

- SELECT 语句的 GROUP BY 子句

  ```
  CREATE TRIGGER t4 DELETE ON tab1
        BEFORE (INSERT INTO tab3 SELECT deptno, SUM(exp)
        FROM budget GROUP BY deptno, budget.update);
  ```

- UPDATE 语句的 SET 子句

  ```
  CREATE TRIGGER t2 UPDATE OF a ON tab1
  ```

```
BEFORE (UPDATE tab2 SET a = 10, tab2.insert = 5);
```

## 在触发操作中使用相关名称

当您在触发操作中使用相关名称时应用这些规则：

- 您可以在 FOR EACH ROW 触发操作列表的 SQL 语句中和 WHEN 条件中为旧列和新列值使用相关名称。
- 同一表上的多个触发器的 WHEN 条件和 FOR EACH ROW 子句可以在触发器和触发器例程的 REFERENCING 子句中使用不同的相关的变量来引用同一列的值。
- 旧的和新的相关性名称引用触发语句影响的所有行。
- 在 GROUP BY 、SET 或 COUNT DISTINCT 子句中不能使用相关性名称限定列名称。
- 相关性名称的引用作用域是整个触发器定义。该作用域是静态确定的，这意味着它限制于触发器定义；它不包含作为触发操作的 UDR 中的表名称限定的级联触发器或列，期望在 FOR EACH ROW 子句中调用触发器的例程。

有关在触发器例程中使用相关名称的其它信息，请参阅 SPL 例程的规则。

## 何时使用相关性名称

在 FOR EACH ROW 列表的 SQL 语句中，您必须使用旧的或新的相关性名称限定对触发表中的列的引用，触发该语句有效独立于触发操作。

换句话说，如果 FOR EACH ROW 触发操作列表中的列名称未用相关性名称限定，则即使使用触发表名称限定它，仍会像该语句独立于触发操作那样解释它。对于非限定的列名称，不特别搜索触发表的定义。

例如，假设以下 DELETE 语句是触发器的 FOR EACH ROW 部分中的触发部分：

```
DELETE FROM tab1 WHERE col_c = col_c2;
```

要使该语句有效，则 **col_c** 和 **col_c2** 都必须是来自 **tab1** 的列。如果打算使用 **col_c2** 作为触发表中某列的相关性引用，则必须使用旧的或新的相关性名称限定它。如果 **col_c2** 不是 **tab1** 中的列且未用旧的或新的相关性名称限定，则您将得到错误。

在有效独立于触发操作的语句中，没有 ***correlation*** 限定符的列名称引用数据库中的当前值。

下一个示例中，在触发器 t1 的触发操作中，相关子查询的 WHERE 子句中的 mgr 是触发表中的非限定列。在此情况中，mgr 引用 empsal 中的当前列值，因为 INSERT 语句有效独立于触发操作。

```
CREATE DATABASE db1;
        CREATE TABLE empsal (empno INT, salary INT, mgr INT);
        CREATE TABLE mgr (eno INT, bonus INT);
        CREATE TABLE biggap (empno INT, salary INT, mgr INT);

        CREATE TRIGGER t1 UPDATE OF salary ON empsal
        AFTER (INSERT INTO biggap SELECT * FROM empsal WHERE salary <
```

```
(SELECT bonus FROM mgr WHERE eno = mgr));
```

在触发操作中，来自触发表的非限定列名引用当前列值，但是触发器声明必须有效独立于触发操作。

## 限定值与非限定值的比较

下面的表总结了在发生不同触发器事件后用 *old* 或 *new* 相关名限定 column 名称时，检索的值。

| 触发事件 | *old*.column | *new*.column |
|---|---|---|
| INSERT | 无值（错误） | 插入的值 |
| UPDATE (column updated) | 初始值 | 当前值（U） |
| UPDATE (column not updated) | 初始值 | 当前值（N） |
| DELETE | 初始值 | 无值（错误） |
| SELECT | 初始值 | 无值（错误） |

当相关性名称没有值是，只要 SQL 或 SPL 语句引用未定义的相关名称执行时而不是在声明相关性名称时，会发出错误，当您读取上一个表时引用以下键。

**术语** **含义**

初始值 触发事件前的值

当前值 触发事件后的值

(N) 不能被触发操作更改

(U) 可以被触发语句更新；更新的值可能因为前面的触发操作而与初始值不同。

在 FOR EACH ROW 触发操作列表外，您不能使用旧的或新的相关名限定来自触发表的列；它总是引用数据库中的当前值。

创建触发器时，触发操作列表中的语句使用任何有效的排列顺序，即使执行触发器操作时有效的排列不同。请参阅 SET COLLATION 语句 以获取关于如何指定与 **DB_LOCALE** 所指定的不同的排列顺序。

## 触发器的再进入

在某些情况下触发器可以是再进入的。在这些情况中，触发操作可以引用触发表。换句话说，触发事件和触发操作都在同一个表上操作。下表总结了触发器可以是再进入的情况和触发器不能是再进入的情况：

- Update 触发器的触发操作不能是触发事件更新的表的 INSERT 或 DELETE 。
- 同样，Update 触发器的触发器操作不能是对触发器事件更新的列的 UPDATE 。（但是 Update 触发器的触发器操作可以更新未由触发器事件更新的列。）

  例如，假设以下 UPDATE 语句，更新 **tab1** 的列 **a** 和 **b** ，是触发语句：

  ```
  UPDATE tab1 SET (a, b) = (a + 1, b + 1);
  ```

现在考虑以下示例中的触发操作。第一个 UPDATE 语句可以是有效的触发操作，但是第二个不是，因为它再次更新 **b** 列。

```
UPDATE tab1 SET c = c + 1;       -- OK
UPDATE tab1 SET b = b + 1;       -- INVALID
```

- 如果触发器具有 UPDATE 事件，则触发操作可以是带有 INTO 语句（引用触发事件更新的列或触发表中的任何其它列）的 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句。

  当 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句是触发操作时 UPDATE 触发器的 INTO 子句仅在 FOR EACH ROW 触发操作中有效，并且出现在 INTO 子句中的列名必须来自触发表。

  以下语句说明了 INTO 子句的正确用法：

```
CREATE TRIGGER upd_totpr UPDATE OF quantity ON items
                REFERENCING OLD AS pre_upd NEW AS post_upd
                FOR EACH ROW(EXECUTE PROCEDURE
                calc_totpr(pre_upd.quantity,post_upd.quantity,
                  pre_upd.total_price) INTO total_price);
```

  INTO 关键字之后的列必须在触发列表中，但是不需要被触发事件更新。

  当 INTO 子句出现在 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句中时，当从 UDR 返回值时，数据库服务器立即用该值更新指定的列。

- 如果触发器具有 INSERT 事件，则触发操作不能是引用触发表中的列的 INSERT 或 DELETE 语句。
- 如果触发器具有 INSERT 事件，则触发操作可以是引用触发表中的列的 UPDATE 语句，但是该列不能是触发器事件向其提供值的列。

  如果触发器具有 INSERT 事件，并且触发器操作更新触发表，则两个语句中的列必须互斥。例如，假设触发语句为表 **tab1** 的列 **cola** 和 **colb** 插入值：

```
INSERT INTO tab1 (cola, colb) VALUES (1,10);
```

  现在考虑以下触发操作。第一个 UPDATE 是有效的，但是第二个无效。因为即使触发事件已经为列 **colb** 提供了值，它仍然更新列 **colb** ：

```
UPDATE tab1 SET colc=100; --OK
UPDATE tab1 SET colb=100; --INVALID
```

- 如果触发器具有 INSERT 事件，则触发操作可以是带有 INTO 子句（引用触发事件提供的列或触发事件未提供的列）的 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句。

  当 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句是触发操作时，仅当触发操作出现在 FOR EACH ROW 列表中时，您可以为 INSERT 触发器指定 INTO 子句。在此情况中，INTO 子句只能包含来自触发表的列名称。

  以下语句说明了 INTO 子句的有效用法：

```
CREATE TRIGGER ins_totpr INSERT ON items
 REFERENCING NEW AS new_ins
 FOR EACH ROW (EXECUTE PROCEDURE calc_totpr
```

```
(0, new_ins.quantity, 0) INTO total_price);
```

INTO 关键字之后的列可以是触发事件提供的触发列表中的列，或是触发事件未提供的触发表中的事件。

当 INTO 子句出现在 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句中时，数据库夫服务器立即用从 UDR 返回的值更新指定的列。

- 如果触发操作是 SELECT 语句，则 SELECT 语句可以引用触发表。SELECT 语句可以是以下实例中的触发操作：
  - SELECT 语句出现在 WHERE 子句的子查询中或出现在触发操作语句中。
  - 触发操作是 UDR ，且 SELECT 语句出现在 UDR 中。

## 再进入和级联触发器

触发器不能再进入的情况中递归应用于所有的级联触发器。它被认为是初始触发器的一部分。特别地，该规则表示级联触发器不能更新原始触发语句更新的触发表中的任何列，包括该语句影响的任何非触发列。例如，假设此 UPDATE 语句是触发语句：

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1);
```

在下一个示例的级联触发器中，trig2 在运行时失败，因为它引用触发 UPDATE 语句更新的列 b ：

```
CREATE TRIGGER trig1 UPDATE OF a ON tab1-- Valid
        AFTER (UPDATE tab2 SET e = e + 1);

        CREATE TRIGGER trig2 UPDATE OF e ON tab2-- Invalid
        AFTER (UPDATE tab1 SET b = b + 1);
```

现在考虑以下 SQL 语句，当执行最终的 UPDATE 语句时，列 **a** 被更新且触发器 **trig1** 被激活。

触发操作再次用 EXECUTE PROCEDURE INTO 语句更新列 **a** 。

```
CREATE TABLE temp1 (a INT, b INT, e INT);
        INSERT INTO temp1 VALUES (10, 20, 30);

        CREATE PROCEDURE proc(val iINT) RETURNING INT,INT;
        RETURN val+10, val+20;
        END PROCEDURE;

        CREATE TRIGGER trig1 UPDATE OF a ON temp1
        FOR EACH ROW (EXECUTE PROCEDURE proc(50) INTO a, e);

        CREATE TRIGGER trig2 UPDATE OF e ON temp1
        FOR EACH ROW (EXECUTE PROCEDURE proc(100) INTO a, e);

        UPDATE temp1 SET (a,b) = (40,50);
```

该级联触发器示例有几个问题。首先，更新列 **a** 是否会再次激活触发器 **trig1** ？答案是否定的。因为触发器已被激活，它没有被再次激活。如果触发操作是 EXECUTE PROCEDURE INTO 或

EXECUTE FUNCTION INTO 语句，只有那些从更新的列到之后（在触发器级联中）在该表中互斥的列上定义的触发器才被激活。其它触发器被忽略。

该示例产生的另一个问题是触发器 **trig2** 是否被激活。答案是肯定的。触发器 **trig2** 在列 **e** 上定义。直到现在，表 **temp1** 中的列 **e** 尚未被修改。触发器 **trig2** 被激活。

该示例产生的最后一个问题是执行 **trig2** 中您的触发操作后触发器 **trig1** 和 **trig2** 是否被激活。答案是否定的。两个触发器都不会被激活。在此之前列 **a** 和 **e** 已被更新一次，触发器 **trig1** 和 **trig2** 已被执行一次。数据库服务器忽略这些触发器且不激活它们。关于级联触发器的更多信息，请参阅级联触发器。

正如前面提到的，视图上的 INSTEAD OF 触发器在其它触发器操作中不能包含 EXECUTE PROCEDURE INTO 语句。而且，如果两个视图分别具有带有已定义的操作（在其它视图上执行插入操作）的 INSERT INSTEAD OF 触发器的话，会产生错误。

## SPL 例程的规则

除了触发器的再进入中列出的规则外，以下规则适用于指定为触发操作的 SPL 例程：

- 在只希望有一行的上下文中，SPL 例程不能是游标函数（返回多行的函数）。
- 在 SPL 例程中不能使用旧的或新的相关性名称，除非 CREATE FUNCTION 或 CREATE PROCEDURE 语句包含了将 UDR 定义为触发例程的 REFERENCING 子句。如果您需要在例程中使用相应的值，则必须将它们传递为参数。例程应答独立于触发器，且旧的或新的相关性名称在触发器外不具有任何意义。
- 触发器例程必须包含可以为触发器例程中 SPL 语句可以引用的 OLD 或 NEW 列值声明相关名称的 REFERENCING 子句。
- 触发例程必须包含指定本地数据库中的表或视图的名称的 FOR ***table_object*** 子句，其触发器可以调用此例程。该触发操作不能调用没有指定触发表或视图的触发器例程。
- 只有在 Triggered Action 列表的 FOR EACH ROW 部分中调用的触发器例程才能直接操作在触发器的或触发器例程的 REFERENCING 子句中定义的旧的或新的相关名称。
- 触发器例程只能在触发器定义中 Triggered Action 列表的 FOR EACH ROW 部分调用。
- OLD 或 NEW 值的相关变量可以出现在 SPL 的 IF 语句和 CASE 表达式中。
- 只有 NEW 值的相关变量可以在引用相关变量的 LET 表达式的左边。在这种情况中，SPL 例程的 FOR 子句必须指定表（而非视图），并且调用 SPL 例程的操作的触发器不能是 INSTEAD OF 触发器。
- OLD 和 NEW 值可以在 LET 表达式的右边。
- 只用 FOR EACH ROW 子句中调用的触发器例程可以使用 Boolean 运算符 SELECTING 、INSERTING 、DELETING 和 UPDATING。如果触发事件符合由相同名称的运算符引用的 DM 操作，则返回 TRUE（'t'），否则返回 FALSE（'f'）。
- SPL 的 IF 语句和 SQL 的 CASE 表达式可以指定这些运算符为触发例程中的条件。
- 触发器例程必须用 SPL 语句编写。它们不能使用外部语言编写，例如 C 或 Java™ 语言，但是触发器例程可以包含外部语言的调用，例如用于触发器内省的 **mi_trigger** 应用程序接口。

- 触发器例程不能引用保存点。触发操作对数据值或数据库结构的更改必须整体提交或回滚。GBase 8s 在触发器例程中不支持 ROLLBACK TO SAVEPOINT 语句用于触发操作的部分回滚。

有关 **mi_trigger** API 的更多信息，请参阅 *GBase 8s DataBlade API 程序员指南* 和 *GBase 8s DataBlade API 函数参考* 。

当您使用 SPL 例程作为触发操作时，除非执行该例程，否则例程引用的数据库对象不被检查。

另请参阅 触发器和 SPL 例程 中的 SPL 限制。

## 执行触发操作的特权

如果您不是此触发器的所有者，但是触发器所有者的访问特权包含 WITH GRANT OPTION ，则除了您子句对每个 SQL 语句的特权外还继承所有者（具有授权选项）的特权。如果触发器操作是 UDR ，则您需要 UDR 上的 Execute 特权，或者触发器所有者必须具有授权选项的 Execute 特权。

**重要：** 作为安全预防措施，用户仅保留角色（但是并不是单独授予用户或作为 PUBLIC 组成员的角色）的自由访问权不能通过触发操作或通过触发例程提供对当前数据库以外的表的访问。

然而，当执行 UDR 时，您不继承触发器所有者的特权；相反，您接收随 UDR 授予的特权，它取决于此例程是 DBA 特权例程还是所有者特权 UDR：

- DBA 特权 UDR 的特权

  当使用 DBA 关键字注册 UDR ，且您被授予了 UDR 上的 Execute 特权时，数据库服务器自动为您授予临时 DBA 特权，这些特权仅当您执行 UDR 时才可用。

- 所有者特权 UDR 的特权

  如果创建没有 DBA 关键字的 UDR，但是 UDR 的所有者对于基础数据库对象上的必要特权具有 WITH GRANT OPTION 关键字，当您被授予 UDR 的 Execute 特权时，您会继承这些特权。

对于没有 DBA 特权的 UDR ，UDR 引用的所有非限定数据库对象都被 UDR 所有者的名称隐式限定。

如果 UDR 所有者没有 WITH GRANT OPTION 特权，则当 UDR 执行您在基础数据库对象上具有原始特权。有关 SPL 例程上的更多信息，请参阅 *GBase 8s SQL 教程指南* 。

不具有 INSTEAD OF 触发器的视图只有过 Select （具有授权选项）特权。但是，如果在它上面创建 INSTEAD OF 触发器，则在触发器创建期间该视图具有 Insert （具有授权选项）特权。视图所有者现在只能为其它人授予 Select 和 Insert 特权。这对触发操作是独立的。不必获取过程或函数上的 Execute（具有授权选项）特权。缺省情况限，在操作列表中的每个 UDR 上授予 Execute（具有授权选项）特权。

您可以使用具有触发器的角色。与角色相关的语句（CREATE ROLE 、DROP ROLE 、GRANT 、REVOKE 和 SET ROLE）和 SET SESSION AUTHORIZATION 语句在触发操作调用的 UDR 中有效。当执行触发器时，用户通过启用角色或通过 SET SESSION AUTHORIZATION 语句已经获取的特权不会被放弃。

在复杂的视图（具有来自多个表的列的视图）上，只有所有者或 DBA 可以创建 INSTEAD OF 触发器。当创建触发器时，所有者接收 Select 特权。只有获取必需的 Execute 特权后，视图所有者才能为其它用户授予特权。当删除复杂视图上的触发器时，所有这些特权都被撤销。

## 创建任何人都能使用的触发操作

要使具有执行触发语句特权的任何人都能够执行某个触发器，您可以要求 DBA 创建具有 DBA 特权的 UDR 并为您授予具有 WITH GRANT OPTION 权限的 Execute 特权。

然后您将具有 DBA 特权的 UDR 用作触发操作。任何人都可以执行该触发操作，因为具有 DBA 特权的 UDR 具有 WITH GRANT OPTION 权限。当您激活 UDR 时，数据库服务器为 DBA 应用特权检查规则。

## 级联触发器

数据库服务器允许除 Select 触发器外的其它触发器级联，也就是说，一个触发器的触发器操作可以激活另一个触发器。（有关级联 Select 触发器的限制的更多信息，请参阅 Select 触发器被激活时的情况。）

级联序列中触发器的最大数目是 61：即初始触发器加上最多 60 个级联触发器。当序列中的级联触发器数目超过最大值时，数据库服务器返回错误号码 -748，带有一些信息：

```
Exceeded limit on maximum number of cascaded triggers.
```

下一个示例说明在 stores_demo 数据库中的 manufact 、stock 和 items 表上实施引用完整性的级联触发器序列。当从 manufact 表删除制造商是，第一个触发器 del_manu 从 stock 表中删除该制造商的所有条目。stock 表中的每个 DELETE 激活第二个触发器 del_items，该触发器从 items 表删除该制造商的所有 items。最终，items 表中的每个 DELETE 触发 SPL 例程 log_order，同时在不能再被填充的 orders 表中创建所有订单的记录。

```
CREATE TRIGGER del_manu
        DELETE ON manufact REFERENCING OLD AS pre_del
        FOR EACH ROW(DELETE FROM stock WHERE manu_code = pre_del.manu_code);
        CREATE TRIGGER del_stock
        DELETE ON stock REFERENCING OLD AS pre_del
        FOR EACH ROW(DELETE FROM items WHERE manu_code = pre_del.manu_code);
        CREATE TRIGGER del_items
        DELETE ON items REFERENCING OLD AS pre_del
        FOR EACH ROW(EXECUTE PROCEDURE log_order(pre_del.order_num));
```

当您不使用日志记录时，**manufact** 和 **stock** 表上的引用完整性约束都阻止该示例中的触发器执行。但是，当您使用日志记录时，触发器成功执行，因为约束检查被延迟，直至所有的触发操作（包括级联触发器的操作）都完成。关于执行触发器时如何处理约束的信息，请参阅约束检查。

除了不修改触发 UPDATE 语句别更新的任何列的 UPDATE 语句或 INSERT 语句外，数据库服务器通过不允许您修改任何级联触发器操作中的触发表俩防止触发器循环。INSERT 触发器可以在同一个表上定义 UPDATE 触发操作。

## 约束检查

当您使用日志记录时，数据库服务器延迟触发语句上的约束检查，直至触发操作列表中的语句执行之后。这等同于执行触发语句前执行 SET CONSTRAINTS ALL DEFERRED 语句。完成触发操作后，数据库服务器有效执行 SET CONSTRAINTS ***constraint*** IMMEDIATE 语句以检查延迟的约束。该操作允许您写入触发器以便是触发器操作能够解析触发语句创建的任何违例。有关更多信息，请参阅 SET Database Object Mode 语句。

考虑以下示例，表 child 具有约束 r1，该约束引用表 parent。您定义触发器 trig1 并使用 INSERT 语句激活它。在触发操作中，trig1 检查 parent 是否具有 child 中当前 cola 具有的值的行；如果没有，将插入该行。

```
CREATE TABLE parent (cola INT PRIMARY KEY);
    CREATE TABLE child (cola INT REFERENCES parent CONSTRAINT r1);
        CREATE TRIGGER trig1 INSERT ON child
        REFERENCING NEW AS new
        FOR EACH ROW
        WHEN((SELECT COUNT (*) FROM parent
        WHERE cola = new.cola) = 0)
        -- parent row does not exist
        (INSERT INTO parent VALUES (new.cola));
```

当您将一行插入到引用约束中的子表中时，该行可能在父表中不存在。数据库服务器不立即在触发语句上返回此错误。相反，它允许触发操作通过将相应的行插入父表来解析约束违例。如先前示例所示，您可以在触发操作内检查父行是否存在，如果存在，则可以提供逻辑以绕过 INSERT 操作。

对于没有日志记录的数据库，数据库服务器不延迟触发语句上的约束检查。在此情况中，如果触发语句违反约束，则数据库服务器立即返回错误。

您不能在触发操作中使用 SET Transaction Mode 语句。当您激活触发器时，数据库服务器检查此约束，因为该语句可能在 UDR 中发生。

## 防止触发器相互覆盖

当您使用 UPDATE 语句激活多个触发器时，触发器可能覆盖较早触发器所做的更改。如果您不希望触发操作相互作用，则可以将该 UPDATE 语句分为多个 UPDATE 语句，每个语句更新单个列。

另一个办法是，您可以为需要触发操作的所有列创建单个更新触发器。然后，在触发操作内，您可以测试正被更新的列并以希望的顺序应用操作。但是，这种方法与让数据库服务器应用单个触发器的操作不同，并且有以下缺点：

- 如果触发 UPDATE 语句将列设置为当前值，则您无法检测 UPDATE ，因此触发操作被跳过。您可能希望执行触发操作，即使列值尚未被更改。
- 如果触发器具有 BEFORE 操作，则它应用于所有列，因为您尚无法检测某列是否被更改。

## 远程数据库中的表

您无法在驻留于当前数据库之外的表或视图上创建触发器。但是，您可以在本地表上定义触发器，该表的触发操作操纵了本地服务器实例的另一个数据库中的表，或另一个服务器实例的数据库中的表。

以下示例在本地 GBase 8s 服务器实例 **dbserver1** （它的会话是已连接的）的当前数据库中的 **newtab** 表上定义了 Update 触发器。此处触发操作在远程 **dbserver2** GBase 8s 服务器实例的数据库中的 **items** 表上指定了 UPDATE 操作：

```
CREATE TRIGGER upd_nt UPDATE ON newtab
        REFERENCING NEW AS post
        FOR EACH ROW(UPDATE stores_demo@dbserver2:items
        SET quantity = post.qty WHERE stock_num = post.stock
        AND manu_code = post.mc);
```

总之，存在于本地数据库中的触发器支持本地、跨数据库和跨服务器的触发操作：

- 在本地数据库中表上的本地触发操作
- 在本地服务器实例的另一个数据库的表上跨数据库触发操作
- 在远程服务器实例的数据库的表上的跨服务器触发操作。

定义在远程服务器实例的数据库上的触发器的跨服务器触发操作可以是激活本地数据库中的一个或多个触发器的事件，本地触发器的触发操作不能是跨服务器操作。如果来自远程数据库服务器的 SELECT 、DELETE 、INSERT 、MERGE 或 UPDATE 语句是激活本地触发器（其操作指定了远程数据库实例的数据库的表）的事件，则触发操作失败。

例如，以下触发操作的组合和触发语句在触发语句执行时产生错误：

```
-- Trigger action from dbserver1 to dbserver3:
        CREATE TRIGGER upd_nt UPDATE ON newtab
        REFERENCING NEW AS post
        FOR EACH ROW(UPDATE stores_demo@dbserver3:items
        SET quantity = post.qty WHERE stock_num = post.stock
        AND manu_code = post.mc);

        -- Triggering statement from dbserver2:
        UPDATE stores_demo@dbserver1:newtab
        SET qty = qty * 2 WHERE s_num = 5
        AND mc = 'ANZ';
```

以上的 UPDATE 语句在运行时不会返回错误，因为跨服务器的触发事件不会触发另一个跨服务器操作。

重要： 作为安全预防措施，用户仅保留角色的自由访问权，不能通过视图或触发器通过对单独去数据库外表的访问。跨数据库触发操作和跨除雾器触发操作需要非本地数据库和直接授权给用户或者授权到 PUBLIC 组的表的存取权限。

## 日志记录和恢复

您可以为数据库创建触发器，使其带有或不带有日志记录。如果具有事务日志记录的数据库中的触发器失败，则触发语句和触发操作会回滚，就好像这些操作时触发语句的扩展那样，但是其它事务不回滚。

但是，在不具有事务日志记录的数据库中，当触发语句失败时，您不能回滚。在此情况中，您负责维护数据库中的数据完整性。触发语句的 UPDATE 、INSERT 或 DELETE 操作在 FOR EACH ROW 部分中的触发操作前发生。如果没有日志记录的数据库中的触发操作失败，则应用程序必须将触发语句更改的行恢复到先前值。

如果触发操作调用 UDR，但是 UDR 在异常处理部分终止，则该部分中修改数据的任何操作都随触发语句回滚。在以下部分示例中，当异常处理程序陷入错误时，它在表 logtab 中插入一行：

```
ON EXCEPTION IN (-201)
        INSERT INTO logtab values (errno, errstr);
        RAISE EXCEPTION -201
        END EXCEPTION;
```

但是，当 RAISE EXCEPTION 语句返回错误时，数据库服务器回滚该 INSERT ，因为它是触发操作的一部分。如果 UDR 在触发操作外执行，则 INSERT 不回滚。

实现触发操作的 UDR 不能包含任何 BEGIN WORK 、COMMIT WORK 或 ROLLBACK WORK 语句。如果数据库具有事务日志记录，则您必须在触发语句前开始显式事务，或者该语句本身必须是显式事务。无论哪种情况，UDR 中任何其它与事务相关的语句都无效。

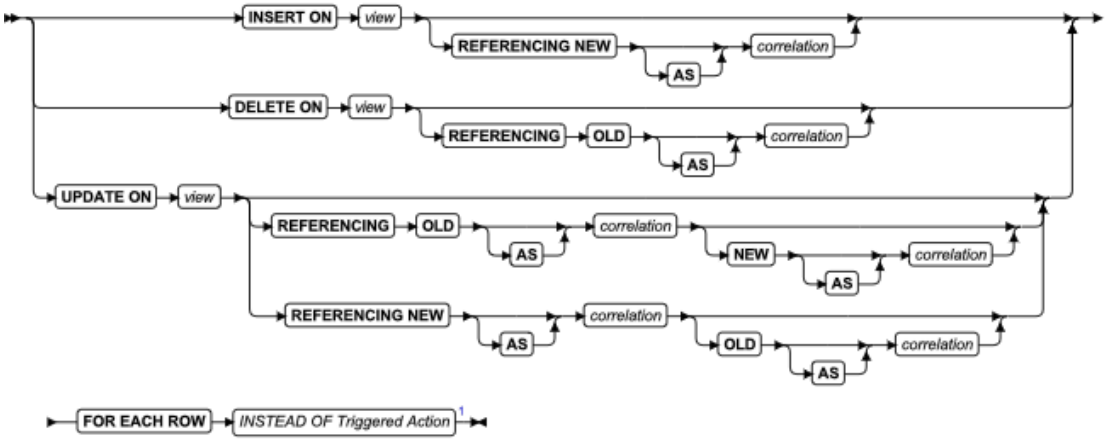
您可以使用触发器实施数据库服务器当前不支持的引用操作。在没有日志记录的数据库中，当触发语句失败时，您负责维护数据完整性。

## 视图上的 INSTEAD OF 触发器

使用 INSTEAD OF 触发器在视图上执行指定的触发操作，而不是执行触发 INSERT 、DELETE 、MERGE 或 UPDATE 语句。

**语法**


视图上的触发器




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *correlation* | 触发操作中限定的旧或新列值的名称（ *correlation.column*） | 在此语句中必须唯一 | 标识符 |
| *trigger* | 在此为触发器声明的名称 | 必须在数据库中的触发器名称中是唯一的 | 标识符 |
| *view* | 触发视图的名称或同义词。可以包含 *owner*. qualifier 。 | 视图或同义词必须存在于当前数据库中 | 标识符 |

您可以使用触发操作更新视图下的表，在某些情况下更新一般“不可更新”的视图。当 INSERT 、DELETE 或 UPDATE 语句引用数据库中的特定列时，您还可以使用 INSTEAD OF 触发器替换其它操作。

在 INSTEAD OF UPDATE 触发器的可选的 REFERENCING 子句中，***新***相关性名称可以出现在***旧***相关性名称之前或之后。

在 GBase 8s 中，CREATE FUNCTION 和 CREATE PROCEDURE 的语句中支持同一 REFERENCING OLD 和 REFERENCING NEW 语法，以在触发例程中定义相关性名称。可以在 REATE FUNCTION 或 CREATE PROCEDURE 语句（定义触发器例程）的 FOR 子句中指定的视图上的 INSTEAD OF 触发器的 Action 子句中调用触发器例程。

指定的***视图***有时称为***触发视图***。此图表的左侧部分（包含***视图*** 规范）定义***触发事件***。该图表的剩余部分定义相关性名称和***触发操作***。

示例

假设 dept 和 emp 是列出部门和员工的表：

```
CREATE TABLE dept (
        deptno INTEGER PRIMARY KEY,
        deptname CHAR(20),
        manager_num INT
        );
```

```
        CREATE TABLE emp (
        empno INTEGER PRIMARY KEY,
        empname CHAR(20),
        deptno INTEGER REFERENCES dept(deptno),
        startdate DATE
        );
ALTER TABLE dept ADD CONSTRAINT(FOREIGN KEY (manager_num)
        REFERENCES emp(empno));
```

下一语句定义 manager_info，它是 dept 和 emp 表中的列构成的视图，包含每个部门中所有的经理：

```
CREATE VIEW manager_info AS
            SELECT d.deptno, d.deptname, e.empno, e.empname
            FROM emp e, dept d WHERE e.empno = d.manager_num;
```

以下 CREATE TRIGGER 语句创建 manager_info_insert，它是设计为向 manager_info 视图中的 dept 和 emp 表插入行 INSTEAD OF 触发器：

```
CREATE TRIGGER manager_info_insert
    INSTEAD OF INSERT ON manager_info      --defines trigger event
    REFERENCING NEW AS n                   --new manager data
    FOR EACH ROW                           --defines trigger action
    EXECUTE PROCEDURE instab(n.deptno, n.empno));

CREATE PROCEDURE instab (dno INT, eno INT)
      INSERT INTO dept(deptno, manager_num) VALUES(dno, eno);
      INSERT INTO emp (empno, deptno) VALUES (eno, dno);
END PROCEDURE;
```

表、视图、触发器和 SPL 例程创建完成之后，数据库服务器将以下 INSERT 语句作为触发事件：

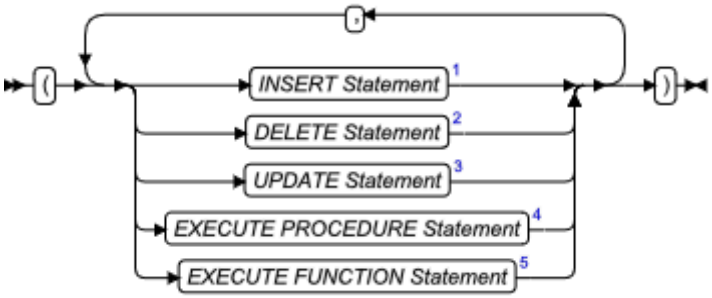
```
INSERT INTO manager_info(deptno, empno) VALUES (08, 4232);
```

此触发 INSERT 语句不会执行，但是该事件会导致触发操作被执行，同时调用 **instab( )** SPL 例程。SPL 例程中的 INSERT 语句向 **manager_info** 视图中的 **emp** 和 **dept** 基本表同时都插入新值。

## INSTEAD OF 触发器的 Action 子句

当遇到指定*视图*的触发事件时，执行触发操作的 SQL 语句，而不是触发语句。视图上定义的触发器在操作子句中支持以下语法。

INSTEAD OF 触发操作




这与表上触发器的触发操作的语法不完全相同，如触发操作部分所述。由于不支持 WHEN (***condition***) ，因此每当遇到 INSTEAD OF 触发事件时都执行同一个触发操作，且只能指定一个操作列表，而不是为每个 ***condition*** 指定独立的列表。

## 视图上 INSTEAD OF 触发器的限制

您必须是***视图***的所有者，或者处于 DBA 状态，以便在视图上创建 INSTEAD OF 触发器。简单视图（仅基于一个表）的所有者具有 Insert 、Update 和 Delete 特权。关于触发器所有者的特权和其它用户的特权间的关系信息，请参阅执行触发操作的特权。

如果多个表在视图下面，则仅其所有者能创建触发器，但是该所有者能将视图上的 DML 特权授权给其他用户。

视图上定义的 INSTEAD OF 触发器不能违反触发器上的限制并且必须遵循以下附加规则：

- 仅可在视图上定义 INSTEAD OF 触发器，而非在表上。
- 视图必须对当前数据库是本地的。
- 视图不能是可更新的视图 WITH CHECK OPTION。
- INSTEAD OF 触发器中没有有效的 SELECT 事件或 WHEN 子句。
- INSTEAD OF 触发器中没有有效的 BEFORE 和 AFTER 操作。
- 在 INSTEAD OF UPDATE 触发器中 OF ***column*** 子句都是无效的。
- 每个 INSTEAD OF 触发器都必须指定 FOR EACH ROW。
- 通过 INSTEAD OF 触发器调用的触发例程不能引用保存点。

视图可以具有任意数量的 INSTEAD OF 触发器以用于定义每种类型的事件（INSERT 、DELETE 或 UPDATE）。

如果 SPL 的 ON EXCEPTION 语句是从 INSTEAD OF 触发器的 Action 子句发出，则它不会生效。

就像表上的触发器，INSTEAD OF 触发器的触发操作向 BIGSERIAL 、SERIAL 或 SERIAL8 列插入序列值，不能更新 SQL 通信区域结构的 **sqlca.sqlerrd[1]** 字段。已触发的 INSERT 操作可以成功的递增该列的序列数目，但是 **sqlca.sqlerrd[1]** 字段的值仍为零，而不会重置为序列值。**sqlca.sqlerrd[1]** 自动可以显示您通过更新的视图直接插入的新的序列值，但是该字段不能显示系列列上的 INSTEAD OF Insert 触发器的操作。

## 更新视图

只有定义视图的 SELECT 语句的以下所有条件都为真，则 INSERT 、DELETE 或 UPDATE 语句可以直接修改视图：

- 视图中的所有列都来自单个表。
- 投影列表中的列都不是聚集值。
- SELECT 投影列表中没有 UNIQUE 或 DISTINCT 关键字。
- 视图定义中没有 GROUP BY 子句和 UNION 运算符。
- 查询不选择计算值和文字值。

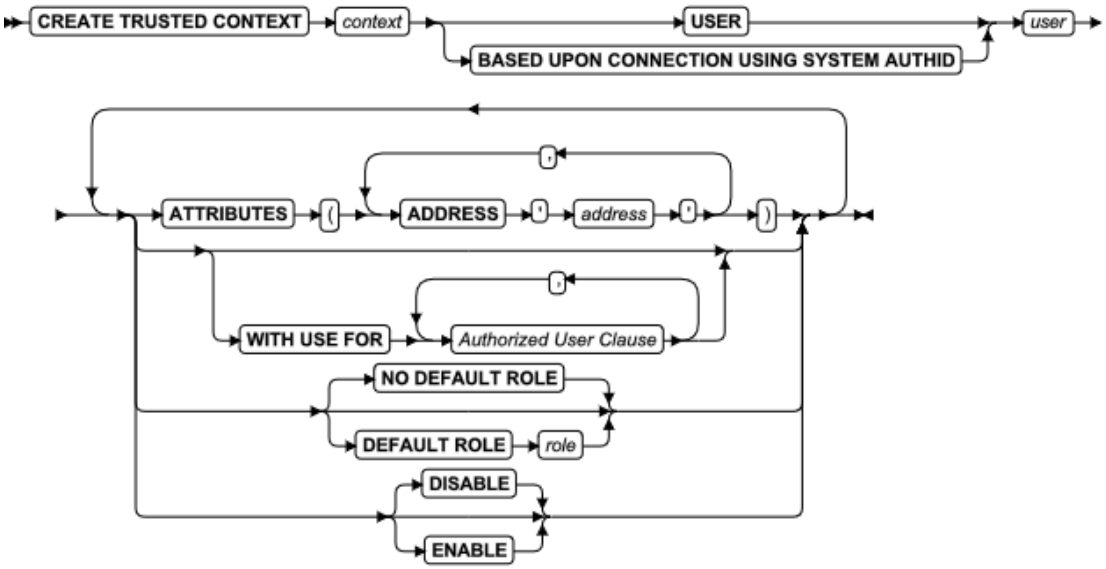
但是，如果触发操作修改基本表，则您可以通过使用 INSTEAD OF 触发器绕过视图上的这些限制。

# 2.48 CREATE TRUSTED CONTEXT 语句

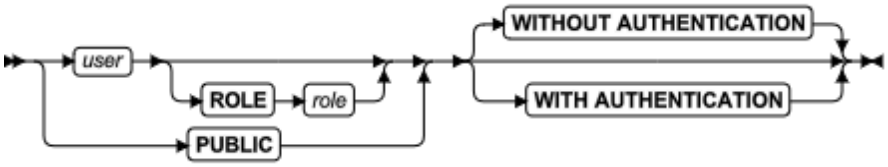
使用 CREATE TRUSTED CONTEXT 语句定义新的可信上下文对象。该语句是 SQL 语言的 ANSI/ISO 标准的扩展。

您必须持有数据库安全管理员（DBSECADM）角色才能运行此语句。

**语法**




Authorized User Clause




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| *address* | 客户端连接到数据库服务器的通信地址 | 在此互信上下文对象的客户端的通信地址中必须是唯一的。有关 *address* 其它的限制，请参阅下面的 ADDRESS attributes 。 | 引用字符串 |
| *context* | 此处为可信上下文对象声明的名称 | 在此数据库服务器示例的可信上下文对象的名称中必须是唯一的，并且不能以字符 SYS 开头 | 标识符 |
| *role* | 现有的用户定义的或内置的角色 | 必须存在于当前数据库中，且必须在此可信上下文对象的属性中是唯一的 | 所有者名称 |
| *user* | 用户的授权标识 | 必须是有效的身授权标识。不能超过 32 字节。不能是发出此语句的用户的授权 ID 。在 WITH USE FOR 子句中必须指定多于一次。 | 所有者名称 |

**用法**

CREATE TRUSTED CONTEXT 语句用于创建可信上下文对象，它运行用户具有可信连接。在 CREATE TRUSTED CONTEXT STATEMENT 中，每个 ATTRIBUTES 、DEFAULT ROLE 、ENABLE 和 WITH USE 子句可以指定多次，每个属性名称和相应的值必须是唯一的。

**USER 子句**

USER 子句指定此 SQL 语句中创建的能建立上下文的系统授权 ID。此语句中的 USER 子句只对 GBase 8s 数据库服务器有效，USER 关键字等价于 BASED UPON CONNECTION USING SYSTEM AUTHID 关键字，在 DB2 的 CREATE TRUSTED CONTEXT 语句中是必需的。这六个关键字可指定 GBase 8s 和 DB2 数据库的系统授权 ID ，但是只有 GBase 8s 支持将 USER 关键字作为它们的同义词。

**ADDRESS 属性**

ATTRIBUTES 子句可以为定义可选上下文对象的数据库指定一个或多个通信地址。以下限制应用于 ALTER TRUSTED CONTEXT 或 CREATE TRUSTED CONTEXT 语句引用的通信地址：

- 每个地址必须在此可选上下文对象的通信地址中是唯一的。
- 每个地址都必须符合 TCP/IP 协议。
- 每个地址必须是 IPv4 地址、IPv6 地址或安全域名。
- IPv4 地址或 IPv6 地址必须是真实的主机地址（不是本地主机），并且不能包含空格。
- 此外，IPv6 地址不能是 IPv4 映射的 IPv6 地址。
- 安全域名不能是动态主机配置协议（DHCP）地址。

如果 *address* 值是安全域的名称，此名称通过域名服务器转换为 IP 地址，其中确定产生 IPv4 或 IPv6 地址。当域名被转换为 IP 地址时，该转换的结果可能是一个或多个 IP 地址集。在这种情况下，如果连接发起的 IP 地址与域名转换的任何 IP 地址下匹配，则数据库服务器将传入连接请求解释为与可信上下文对象的 ADDRESS 属性匹配。

此 ADDRESS 属性可以被多次指定，但是每对 *address* 对于属性集合必须是唯一的。

**注意：**

如果您在 ATTRIBUTES 子句中具有包含 ENCRYPTION 或 WITH ENCRYPTION 应用程序，则可以不管它们，而数据库服务器不会发出 SQL 错误。但是，除了 `WITH ENCRYPTION 'NONE'` 和 `ENCRYPTION 'NONE'`， GBase 8s 数据库服务器不支持 CREATE TRUSTED CONTEXT 语句的这些加密选项。

**WITH USE FOR 子句**

WITH USE FOR 子句指定该可信连接可以被指定的授权标识使用。同一 ***user*** 名称只能在此子句中出现一次，允许由指定用户的列表或 PUBLIC 访问。

例如，假设可信上下文对象被定义为允许由 `PUBLIC WITH AUTHENTICATION` 和 `joe WITHOUT AUTHENTICATION` 访问。如果通过 `joe` 使用可信上下文，则不需要授权。但是，如果由 `george` 使用此可信上下文对象，它只能作为 `PUBLIC` 的成员访问，则需要授权。

WITH AUTHENTICATION 属性指定将基于此可信上下文对象的可信连接上的当前用户切换到此用户需要授权。WITHOUT AUTHENTICATION 属性指定切换当前用户不需要授权。用户的规范会覆盖 PUBLIC 的规范。

这些属性还会影响 ODBC、JDBC 或 ESQL/C 连接的客户端会话期间是否需要认证，其中 SET SESSION AUTHORIZATION 语句在可信连接建立后尝试切换到其它用户 ID 。

**DEFAULT ROLE 属性**

ROLE 对象指定使用可信连接时用户的角色（和特权）。DEFAULT ROLE 标识存在于当前数据库中的角色，并且在当用户没有被定义为可信上下文对象定义的一部分的用户指定角色时使用。NO DEFAULT ROLE 属性将指定没有缺省角色的可信上下文对象。缺省为 NO DEFAULT ROLE。为用户显式指定的角色将覆盖与可信上下文对象相关联的任何缺省角色。

**ENABLE 和 DISABLE 关键字**

ENABLE 关键字指定创建的可信上下文对象处于启用状态。

DISABLE 关键字指定创建的新的可信上下文对象处于禁用状态，并且对于已建立的新的可信连接未启用。

您不能使用 SET Database Object Mode 语句更改可信上下文的 ENABLE 或 DISABLE 属性。如果您需要重置可信上下文的 ENABLED 或 DISABLED 模式，则必须使用 ALTER TRUSTED CONTEXT 语句。

**可信上下文定义的示例**

***示例 1：*** 创建可信环境对象，以便基于此可信环境对象的可信连接上的当前用户可以切换到两个不同的用户 ID。当此连接的当前用户切换为 joe 时，不需要授权。但是，当连接的当前用户切换为 bob 时，需要授权。注意可信上下文对象具有称为 MANAGER 的缺省角色。 这意味着在此可信上下文对象范围内工作的用户将继承与 MANAGER 角色关联的自由访问权。

```
CREATE TRUSTED CONTEXT appserver
        USER wrjaibi
        DEFAULT ROLE MANAGER
        ENABLE
        ATTRIBUTES (ADDRESS '9.26.113.204')
        WITH USE FOR joe WITHOUT AUTHENTICATION,
        bob WITH AUTHENTICATION;
```

***示例 2：*** 创建可信环境对象，以便基于此可信环境对象的可信连接的当前用户可以切换到任何用户 ID ，而不进行认证。

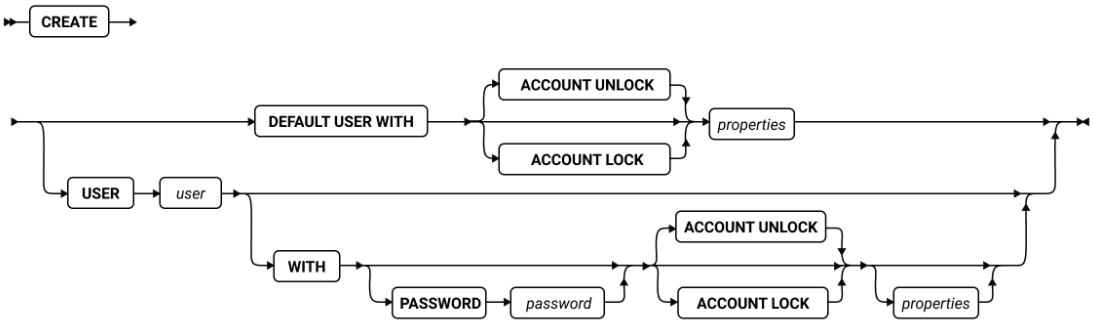
```
CREATE TRUSTED CONTEXT securerole
            USER pbird
            ENABLE
            ATTRIBUTES (ADDRESS 'example.gbase.com')
            WITH USE FOR PUBLIC WITHOUT AUTHENTICATION;
```

## 2.49 CREATE USER 语句 (UNIX™、Linux™)

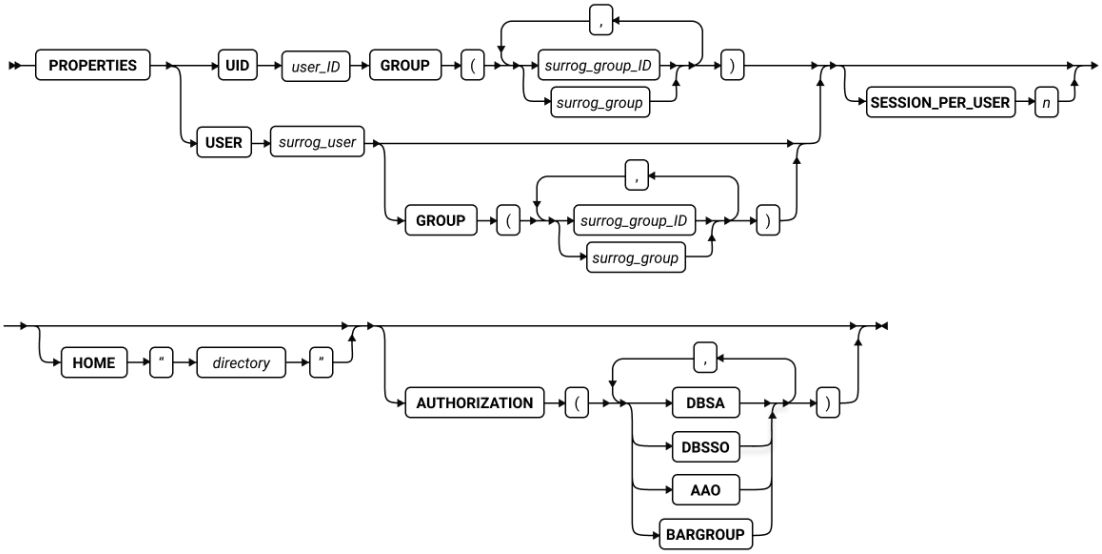
使用 CREATE USER 语句定义内部认证的用户，或将外部认证的用户映射到访问 GBase 8s 资源所需的代理用户属性。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




Properties







| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *directory* | 存储用户文件的目录的路径名称 | 必须是 255 字节或更少，并且必须符合您操作系统的规则。*directory* 还必须：<br>• 属于映射的 *user_ID* 和 *surrog_group_ID*。<br>• 所有者具有读、写和执行权限。 | 引用字符串 |
| *password* | *user* 的内部认证的密码。 | 必须是 6 - 32 字节。 | 引用字符串 |
| *surrog_group* | 具有要映射 *user* 的权限的现有操作系统组（代理组）的名称。*surrog_group* 值必须包含在括号中。 | 至多为 32 字节。<br>必须是 /etc/gbasedbt/allowed.surrogates 文件中指定的代理之一。 | 所有者名称 |
| *surrog_group_ID* | 要映射 *user* 的组标识编号。*surrog_group_id* 值的列表必须包含在括号中。 | *surrog_group_ID* 不能是：<br>• 具有服务器管理特权（DBSA 、DBSSO 、AAO 和 BARGROUP）的组 ID<br>• 组 0 （**root**，有时引用为 **wheel** 或 **system**）<br>• 与组 **bin** 或组 **sys** 相关联的组 ID<br>必须是 /etc/gbasedbt/allowed.surrogates 文件中指定的代理之一。 | 精确数值 |
| *surrog_user* | 在具有要映射 *user* 权限的 GBase 8s 主机计 | 必须符合您操作系统的规则。<br>必须是 | 所有者名称 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 算机上的现有操作系统用户账户（代理用户）的名称。 | /etc/gbasedbt/allowed.surrogates 文件中指定的代理之一。。 | |
| *user* | 您要映射到用户属性的指定用户的授权标识。 | 不能是 PUBLIC。 | 所有者名称 |
| *user_ID* | 要映射 *user* 的用户标识符号。 | 不能是用户 root 或用户 gbasedbt。<br>必须是 /etc/gbasedbt/allowed.surrogates 文件中指定的代理之一。 | 精确数值 |

**用法**

只有 DBSA 才能运行 CREATE USER 语句。在非 root 安装中，除非用户将 DBSA 特权委托给另一个用户，否则安装服务器的用户等价于 DBSA 。

在 CREATE USER 语句定义的用户可以连接到数据库服务器之前，USERMAPPING 配置参数必须设置成为支持映射用户的值。DBSA 可以发出 CREATE USER 语句以将用户映射到与适当级别的授权相对应的属性。

还必须在 **sysusers** 的 SYSUSERMAP 表中输入值，以将用户映射到适当的用户属性，以使 SQL 的映射用户已经正常工作。

CREATE USER 语句的执行可以用 CRUR 审计代码审计。

**PASSWORD 子句**

对于 root 特权的服务器，如果 OS 用户正在连接而还没有设置 USERMAPPING 配置参数，则尽管用户在当前数据库中存在，OS 认证仍会发生。当设置了 USERMAPPING 参数时，内部用户认证优先于 OS 认证。映射的用户都是内部或外部认证的。当创建没有密码的用户时，将创建映射的用户。当创建有密码的用户时，除非在此语句中指定了显式 PROPERTIES 子句，否则使用来在操作系统的属性创建内部认证用户。当 CREATE USER 语句包含 PASSWORD 子句和 PROPERTIES 子句时，此用户是内部认证的用户，但是它具有 PROPERTIES 子句中指定的代理属性。在这种情况中，代理用户或代理组必须在 /etc/gbasedbt/allowed.surrogates file 中列出。

**PROPERTIES 子句**

PROPERTIES 子句可以定义新的用户，并且可以可选地将此用户与包含组合主目录的代理属性相关联。CREATE DEFAULT USER 是 CREATE USER 语句的特例。CREATE DEFAULT USER 语句定义为缺省用户设置的属性。当定义完缺省用户属性之后，您可以通过省略 PROPERTIES 子句创建新的具有缺省用户属性的用户。 映射的用户可以使用代理用户属性连接到数据库服务器，如果它们使用可插入身份验证模块（PAM）、单点登录（SSO）或内部身份验证。属性值不适用于非 root 安

装，但必须像 root 权限服务器一样指定。但是，代理用户和代理组在非 root 安装中在 allowed.surrogates 文件中是不必需的。

**SESSION_PER_USER 子句**

SESSION_PER_USER n：在一个实例中，一个用户可以同时拥有的会话数量，正整数，n 的取值范围为：[1,$2^{31}$ -1]。

例如以下语句创建用户名为 user1001，密码为 123456，os_user 为映射的操作系统用户，限制该用户的会话连接数为 5：

```
CREATE USER user1001 WITH PASSWORD '123456'
PROPERTIES USER os_user SESSION_PER_USER 5;
```

修改用户的会话连接数，以下语句修改用户 user1001 的允许的最大连接数：

```
ALTER USER user1001 MODIFY SESSION_PER_USER 6;
```

**AUTHORIZATION 子句**

AUTHORIZATION 子句授权管理特权的子集。USERMAPPING 配置参数必须设置成 ADMIN 以启用此子句。

**注：**

不推荐使用 AUTHORIZATION 子句（ALTER USER 的 AUTHORIZATION 子句或 GRANT ACCESS TO PROPERTIES ）。此语法在更高版本中不支持角色分离。

**HOME 目录子句**

使用 HOME 关键字为用户文件指定目录是可选的，但在某些情况下，这是非常可取的。如果您没有指定主目录，具有同一主目录的内部认证用户将作为 GBase 8s 主机计算机上的代理用户账户。如果代理用户身份没有设置主目录，则 GBase 8s 为$GBASEDBTDIR/users 的用户文件创建目录。在这之后的情况中，$GBASEDBTDIR/users 目录名称采用这种格式 uid.*ID_number* （如，uid.101）。

**ACCOUNT LOCK 和 ACCOUNT UNLOCK 关键字**

使用 ACCOUNT LOCK 和 ACCOUNT UNLOCK 关键字，DBSA 可以切换禁用和启用指定用户对数据库服务器的访问。

**示例**

示例 1：创建映射用户

以下语句创建了名为 **joe** 的映射用户。

```
CREATE USER joe;
```

如果用户 **joe** 是 OS 用户，则 **joe** 禁用与其用户名称关联的操作系统属性。

如果用户 **joe** 不是 OS 用户，并且已经定义了缺省用户属性，则 **joe** 具有缺省用户的代理属性。如果没有定义缺省语句，则会返回错误。

示例 2：创建内部认证用户：

以下示例创建了名为 **joe** 的内部认证用户，它的密码为 **joebar**：

```
CREATE USER joe WITH PASSWORD "joebar";
```

如果用户 **joe** 不是 OS 用户，并且已经定义了缺省用户属性，则 **joe** 具有缺省用户的代理属性。如果没有定义缺省语句，则会返回错误。

示例 3：使用锁定的账户创建内部认证用户：

以下语句创建具有锁定的账户的名为 **phil** 的内部认证用户：

```
CREATE USER phil WITH PASSWORD "joebar" ACCOUNT LOCK;
```

如果用户 **phil** 不是 OS 用户，并且已经定义了缺省用户属性，则 **phil** 具有缺省用户的代理属性。如果没有定义缺省语句，则会返回错误。

示例 4：创建具有特定属性的内部认证用户：

以下语句创建具有 UID 、组和主目录的内部认证用户 **mary** ：

```
CREATE USER mary WITH PASSWORD "joebar" PROPERTIES UID 44567
GROUP(1234) HOME "/home/pd/osuser";
```

示例 5：创建具有代理用户的映射用户：

以下语句创建具有代理用户 **foo_os** 的映射用户 **bill** ：

```
CREATE USER bill WITH PROPERTIES user "foo_os";
```

用户 **bill** 拥有用户 **foo_os** 的操作系统属性。

示例 6：创建缺省用户：

以下语句创建一个用户（内部命名为 PUBLIC），其中包含代理用户 **tmp** 的属性：

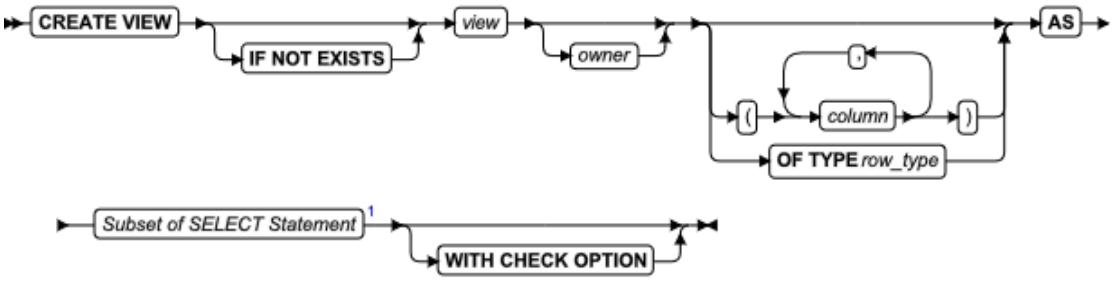
```
CREATE DEFAULT USER WITH PROPERTIES USER "tmp";
```

创建的没有代理属性的其它用户将具有这些属性。

## 2.50 CREATE VIEW 语句

使用 CREATE VIEW 语句创建新的视图，该视图基于驻留在数据库（或本地数据库服务器或不同的数据库服务器中的另一个数据库）中的一个或多个现有表和视图。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 在此处声明的视图中的列的名称。缺省值是 SELECT 投影列表中的名称。 | 请参阅命名视图列。 | 标识符 |
| *owner* | 视图的所有者。如果省略，缺省为发出此语句的用户 ID 。 | 要指定另一个用户 ID 需要 DBA 存取权限。 | 所有者名称 |
| *row_type* | 已归类的视图的已命名行类型 | 必须已经在数据库中存在 | 数据类型 |
| *view* | 在此处声明的视图名称 | 在数据库中的物化视图、视图、表、序列和同义词名称中必须唯一。 | 标识符 |

**用法**

*视图*是虚拟的表，由 SELECT 语句定义。除了以下列表中列出的语句，您可以在任何表名称在语法上有效的 SQL 语句中指定视图的名称或同义词：

- ALTER FRAGMENT
- CREATE INDEX
- CREATE TABLE
- CREATE TRIGGER
- RENAME TABLE
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE
- TRUNCATE
- UPDATE STATISTICS

当您使用 CREATE TRIGGER 语句在视图上定义 INSTEAD OF 触发器时必须指定视图的名称，但是此语法和功能与定义在表上的触发器不同。

通过视图更新禁止在 INSERT 、DELETE 或 UPDATE 语句中使用非可更新视图（其中其它视图是有效的）。

要创建视图，您必须拥有对从中派生视图的所有列的 Select 特权。您可以如同表一样查询视图，并且在以下情况下可如同表一样更新视图，但视图并不是表。

如果您包含可选的 IF NOT EXISTS 关键字，当指定名称的视图已经在当前数据库中注册过，或者指定名称是当前数据库中表、同义词或序列对象的标识符，则数据库服务器不会执行任何操作（而不是向应用程序发送异常）。

视图是由视图定义中的 SELECT 语句在每次查询中引用该视图时返回的行和列的集合组成。

在一些情况下，数据库服务器将用户的 SELECT 语句同定义视图的 SELECT 语句合并，然后执行组合语句。在其它情况下，如果视图定义的复杂性导致数据库服务器创建了一个临时表（称为实现的视图）。有关实现的视图的更多信息，请参阅 *GBase 8s 性能指南* 。

视图反映了对底层表的更改，但是有两个例外：

- 如果 SELECT * 规范定义了视图，则视图中只有那些在由 CREATE VIEW 定义视图时存在于底层表中的列。随后使用 ALTER TABLE 语句添加到底层表的任何新的列不会出现在该视图中。
- 如果 GRANT 或 REVOKE 语句修改了视图定义中引用的表上的自由访问权，则数据库服务器不会将这些存取特权应用到视图上。

要强制将对基础表的访问权限或模式的修改应用于视图，你可以使用 DROP VIEW 和 CREATE VIEW 语句删除并重建此视图。还可以使用 CREATE VIEW 和 CREATE TRIGGER 语句分别重建 DROP VIEW 语句销毁的任何相关视图或 INSTEAD OF 触发器。

视图继承了从中派生视图的表的列的数据类型。数据库服务器通过表达式的特性确定虚拟列的数据类型。

SELECT 语句存储于 **sysviews** 系统目录表中。当您随后在另一个语句中引用视图的时候，数据库服务器会在执行新语句的同时执行定义 SELECT 语句的操作。

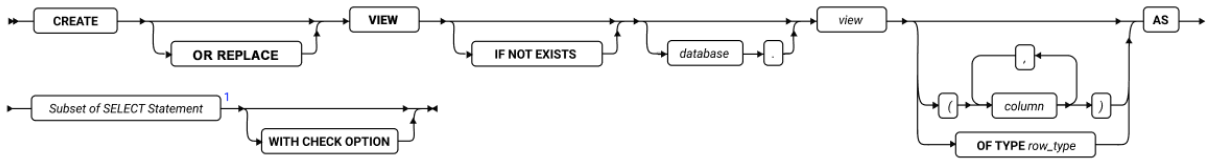
在 DB-Access 中，如果在 CREATE SCHEMA 语句外创建视图，那么在使用 **-ansi** 标志或设置 **DBANSIWARN** 环境变量的情况下，将会收到警告。

下面的语句创建一个基于 person 表的视图。创建类似这样的没有 OF TYPE 子句的视图时，该视图称为***未归类的视图***。

```
CREATE VIEW v1 AS SELECT * FROM person;
```

在 Oracle 模式下，保持 GBase 8s 原有创建视图方式不变的基础上，支持 CREATE OR REPLACE VIEW 的语法结构和使用方式。 具体请参见 CREATE VIEW 中 OR REPLACE 章节内容。

## OR REPLACE




指定 OR REPLACE (CREATE OR REPLACE VIEW) 将创建一个新视图或替换同名的现有视图。

例如，当前数据库是否已经存在 v_tab 视图，都可以使用以下语句创建 v_tab 视图:
create or replace view v_tab as select col from tab;

## 已归类的视图

如果您对已命名的 ROW 类型拥有 Usage 特权，或者您是其所有者或 DBA ,则您可以创建***已归类的视图*** 。如果忽略 OF TYPE 子句，则认为视图中的行是未归类的，并且缺省为一个未命名的 ROW 类型。

如果类型表，已归类的视图是基于命名的 ROW 类型。视图中的每个列相当于命名的 ROW 类型中的一个字段。下面的语句创建了一个基于表 person 的已归类的视图。

CREATE VIEW v2 OF TYPE person_t AS SELECT * FROM person;

要创建已归类的视图，必须包括 OF TYPE 子句。当创建一个已归类的视图时，紧跟在 OF TYPE 关键字之后指定的命名的 ROW 类型必须已存在。

## 视图定义中有效的 SELECT 语句的子集

视图定义中支持大多数 SELECT 语句语法，但有一些例外。

您不能在临时表上创建视图。SELECT 语句的 FROM 子集不可以包括临时表的名称。

CREATE VIEW 语句中由 SELECT 语句引用的表对象可以是永久数据库表、视图或派生的表。该查询可以引用单独的表对象，或连接两个或多个表对象。这些表可以是当前数据库中的表，可以是本地数据库服务器的其它数据库中的表，或者远程服务器实例的数据库中的表。SELECT 语句可以使用用不相关表或相关表的引用在 FROM 语句中定义派生表。这些派生表定义可以包含 LATERAL 关键字和横向表和列引用。

如果取消用户对表的 Select 特权，而在某条定义视图（该视图为同一个用户所拥有）的 SELECT 语句中引用了该表，则此视图会被删除，除非它还包含来自其它数据库中的表的列。

您无法在驻留在远程数据库中的类型表（包括作为表层次结构一部分的任何表）上创建视图。

请勿在 Projection 子句的 Select 列表中使用显示标签。Projection 子句中的显示标签会被解释为列名称。

Hardcoded 值不应在视图定义中使用，但只能在查询视图的后续 SELECT 语句的 WHERE 子句中使用。如果视图中的值不是 hardcoded ，则查询优化器可以始终排除这些文字值，并且可以在更短的时间内完成查询。但是如果相同的值在视图中被 hardcoded ，则查询优化器仍然必须评估每个文字值。

CREATE VIEW 中的 SELECT 语句不包括 SKIP 、FIRST 或 LIMIT 关键字或 INTO TEMP 子句。有关 SELECT 语句语法和用法的完整信息，请参阅 SELECT 语句 。

## 联合视图

一个在其 SELECT 语句中包含 UNION 或 UNION ALL 操作符的视图称为***联合视图***。某些限制应用于联合视图：

- 如果 CREATE VIEW 语句定义了一个联合视图，或者包含 INTERSECT 、MINUS 或 EXCEPT 集合运算符，则您不能在 CREATE VIEW 语句中指定 WITH CHECK OPTION 关键字。
- 在独立的 SELECT 语句中应用于 UNION 或 UNION ALL 操作的所有限制，也应用于联合视图的 SELECT 语句中的 UNION 和 UNION ALL 操作。
- 同样，在独立的 SELECT 语句中应用于 INTERSECT 、MINUS 或 EXCEPT 集合运算符，也应用于定义视图的组合 SELECT 语句中。

关于这些限制的列表，请参阅 对组合的 SELECT 的限制。关于定义联合视图的 CREATE VIEW 语句的示例，请参阅命名视图列。

## 指定视图列

在 ***column*** 列表中指定的列的数目必须与定义视图的 SELECT 语句所返回的列数一致。如果不指定列名称，则视图继承底层表的列名称。在下面的示例中，视图 **herostock** 拥有的列名称与 SELECT 语句的 Projection 子句中的列名称相同：

```
CREATE VIEW herostock AS
SELECT stock_num, description, unit_price, unit, unit_descr
FROM stock WHERE manu_code = 'HRO';
```

当投影列为常量表达式、函数表达式、条件表达式等表达式时，用户必须将对应投影列指定别名。

当 SELECT 语句包括 UNION 操作符并且投影列包括表达式时，创建视图时用户必须为第一个 SELECT 语句的表达式投影列指定别名。最终视图的列名以第一个 SELECT 语句的投影列名为准。例如：

```
CREATE VIEW newview3  AS
SELECT 1101 as tabid, ‘tab1’as tabname FROM tab1
UNION
SELECT 1102, ‘tab2’ FROM tab2;
```

该语句创建的视图 newiew3 具有两列： tabid 、 tabname 。

当 SELECT 语句包括 UNION 操作符并且 SELECT 语句中相应列的名称不相同时，创建视图时也可以不指定别名。最终的视图的列名以第一个 SELECT 语句的投影列名为准。例如：

```
CREATE VIEW newview3   AS
SELECT tab1_id, tab1_name FROM tab1
UNION
SELECT tab2_id, tab2_name FROM tab2;
```

该语句创建的视图 newiew3 具有两列：tab1_id、tab1_name。

但是，在以下环境中必须指定至少一个列名：

- 您还必须指定列名称，以防选择的列含有不带表限定符的重复的列名称。例如，如果 orders.order_num 和 items.order_num 都出现在 SELECT 语句中，则 CREATE VIEW 语句必须提供两个分开的列名以标记它们：

```
CREATE VIEW someorders (custnum,ocustnum,newprice) AS
        SELECT orders.order_num,items.order_num,
                items.total_price*1.5
                FROM orders, items
         WHERE orders.order_num = items.order_num
                AND items.total_price > 100.00;
```

此处 **custnum** 和 **ocustnum** 替换两个相同的列名称。

## 在 SELECT 语句中使用视图

您可以定义一个其列基于其它视图的视图，但必须遵守对创建视图的限制。

## WITH CHECK OPTION 关键字

WITH CHECK OPTION 关键字指示数据库服务器确保通过视图对基础表所作的所有修改均满足视图的定义。

下面的示例创建一个名为 palo_alto 的视图，它使用城市 Palo Alto 中的顾客的 customer 表中的所有信息。由于指定了 WITH CHECK OPTIO 关键字，所以数据库服务器会检查通过 palo_alto 对 customer 表所做的任何修改。

```
CREATE VIEW palo_alto AS
        SELECT * FROM customer WHERE city = 'Palo Alto'
        WITH CHECK OPTION
```

您可以在视图中插入一个不满足视图条件的行（即通过视图不显示的行）。也可以更新视图的行，使其不再满足视图的统计。例如，如果视图是不使用 WITH CHECK OPTION 关键字创建的，那么您可以通过城市为 Los Altos 的视图插入一行，或者通过将城市从 Palo Alto 更改为 Los Altos。

要防止这样的插入和更新，可以在创建视图时添加 WITH CHECK OPTION 关键字。这些关键字要求数据库服务器测试每个插入的或更新的行以确保它满足由视图的 WHERE 子句设置的条件。如果行不满足这些条件，则数据库服务器拒绝操作并以出错形式表明。

即使视图是用 WITH CHECK OPTION 关键字创建的，但您也可以通过视图执行插入和更新操作以更改不是视图定义一部分的列。如果某列未出现在定义视图的 SELECT 语句的 WHERE 子句中，则该列不是视图定义的一部分。

如果 CREATE VIEW 语句中包含 WITH CHECK OPTION 关键字，其中 UNION 、INTERSECT 、MINUS 或 EXCEPT 集合运算符在视图定义中组合两个查询，则 CREATE VIEW 语句失败并显示错误 -940 。

## 通过视图更新

如果视图是构建在一个单独的表上，则在定义该视图的 SELECT 语句不包含任何以下元素的情况下，则该视图是 ***可更新的***：

- 投影列表中是聚集值的列
- 投影列表中使用 UNIQUE 或 DISTINCT 关键字
- GROUP BY 子句
- UNION 操作符
- 选择计算或文字值的查询

您可以从一个单个表中选择计算值的视图执行 DELETE 操作，但是 INSERT 和 UPDATE 操作对这样的视图无效。

在可更新的视图中，可以通过将基础表中的值插入视图来更新这些值。然而，如果视图构建在列有派生值的表上，则该列不能够通过该视图更新。但是，可以更新视图中的其它列。

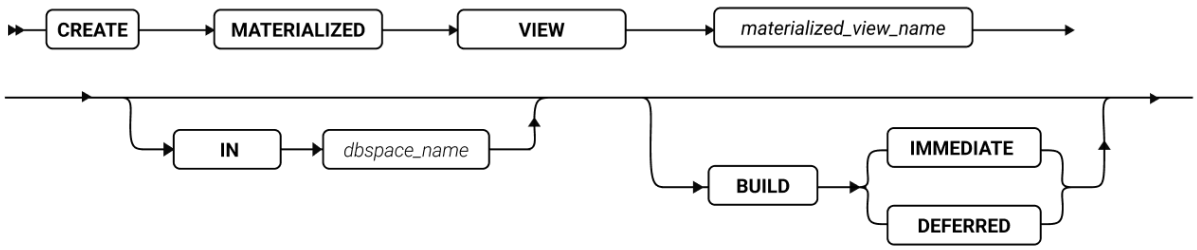
有关使用 INSTEAD OF 触发器更新基于多个表的视图，或者更新其中包括含有聚集值或其它计算值的列的视图的信息，另请参阅更新视图。

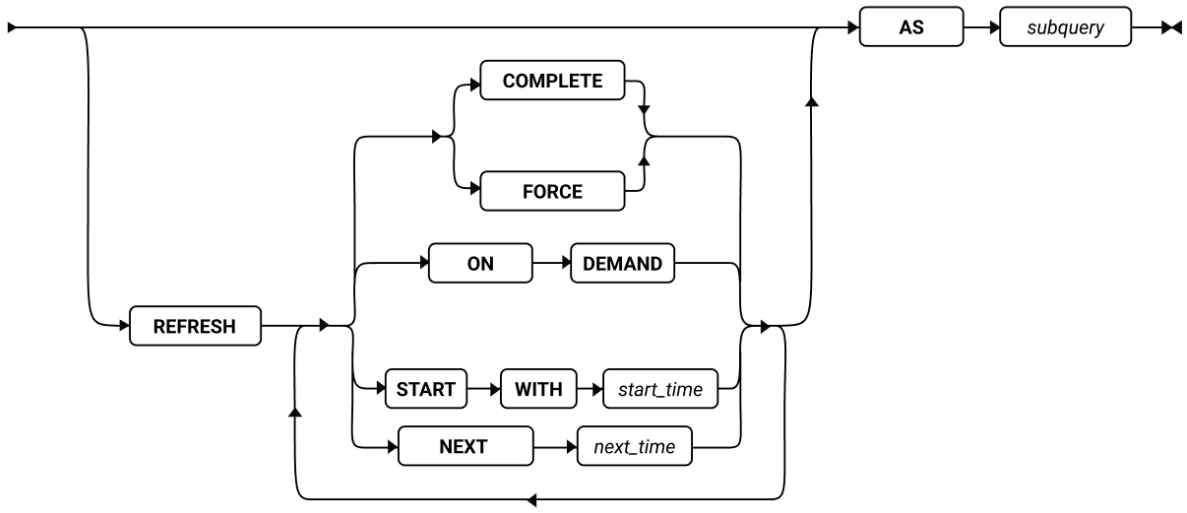
**重要：** 您无法通过使用 WITH CHECK OPTION 关键字创建的视图更新或插入远程表中的行。

# 2.51 CREATE MATERIALIZED VIEW

物化视图的创建

语法：






| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *materialized_view_name* | 此处为创建的物化视图名称 | 在数据库中的物化视图、视图、表、序列和同义词名称中必须唯一。 | 标识符 |
| *dbspace_name* | 指定的 dbspace 名称 | 指定的 dbspace 必须在数据库中存在 | 标识符 |
| *subquery* | 查询语句逻辑 | 同 CREATE VIEW 语句中中查询语句的限制(视图定义中有效的 SELECT 语句的子集) | select 语句 |
| *start_time* | 开始刷新时间 | 有效的日期时间类型 | 标识符 |
| *next_time* | 下一次刷新时间 | 有效的日期时间类型 | 标识符 |

**用法**

物化视图的创建功能，用户可以通过sql语句”create materialized view 物化视图名 … 可选参数 … as 查询语句;” 创建物化视图，语句中的可选参数可以指定创建物化视图的 DBSPACE，设置物化视图是否在创建的同时刷新数据。

- 该功能仅在 GBase 8s 的 ORACLE 模式下支持；
- 物化视图名最大长度123个字节；
- 如果在目录 /etc/sysadmin 中存在 stop 文件需要删除，否则物化视图的定时自动刷新功能无法正常使用。如果需要禁用 ph task 下与物化视图无关的系统任务，可以通过下面语句完成：

```
update ph_task set tk_enable = 'f'
```

```
where tk_name not like 'Refresh_MV%';
```

- 存储位置：

  可以通过 IN dbspace_name 指定物化视图数据存放的 DBSPACE。

- 初始化参数：

  - BUILD IMMEDIATE

    创建物化视图的同时立即生成数据。

  - BUILD DEFERRED

    创建物化视图时不生成数据，但如果同时指定 START WITH 为当前系统时间，物化视图创建完成之后会立即进行刷新。

- 刷新模式：

  - COMPLETE

    当物化视图刷新时会根据查询语句生成的数据进行全量刷新。

  - FORCE

    强制刷新,由于当前版本暂不支持增量刷新功能,所以目前 FORCE 和 COMPLETE 均为全量刷新，缺省会默认使用 FORCE。

- 刷新时机

  - ON DEMAND

    按需刷新，用户可以手动刷新，也可通过 START WITH 和 NEXT 指定时间自动刷新；

  - START WITH ...[NEXT ...]

    - START WITH ... 用户指定物化视图的首次刷新时间，NEXT ...指定下一次刷新时间，后者与前者的时间间隔就是自动刷新的间隔频率；
    - 如果指定 START WITH 省略 NEXT 则首次刷新后不再自动刷新；
    - 如果省略 START WITH 指定 NEXT 则使用当前系统时间与下一次刷新时间的间隔作为刷新间隔；
    - 如果 START WITH 和 NEXT 均未指定则物化视图不会自动刷新。

**示例**

例如，在下面的示例中，创建的物化视图名为: mv_test_nextrefresh 的物化视图，刷新开始时间为物化视图创建时间，下一次刷新时间为 20 秒后,刷新间隔为 20 秒的物化视图

```
create materialized view    mv_test_nextrefresh
refresh on demand start with sysdate next sysdate+1/24/60/3
as
select * from teacher where id > 6;
```

查询刚刚创建的物化视图

```
select * from mv_test_nextrefresh;
```

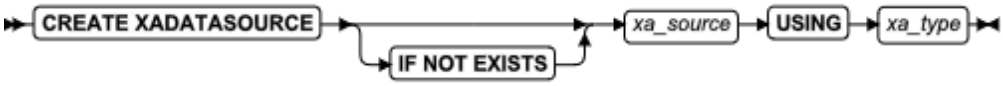
返回结果如下:

```
id    name    subject    id_group
7     a       v          1
8     b       w          1
9     c       x          1
10    d       y          1
```

## 2.52 CREATE XADATASOURCE 语句

使用 CREATE XADATASOURCE 语句创建新的兼容 XA 的数据源并为其在 **sysxadatasources** 系统目录表中创建一个条目。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *xa_source* | 此处为新的 XA 数据源声明的名称 | 在 sysxadatasources 中的 XA 数据源名称中必须是唯一的 | 标识符 |
| *xa_type* | 现有 XA 数据源类型的名称 | 必须已经存在于 sysxasourcetypes 系统目录表的数据库中 | 标识符 |

**用法**

符合 XA 标准的数据源是符合 X/Open DTP XA 标准的外部数据源，用于管理事务管理器和资源管理器之间的交互。要在数据库中注册符合 XA 的数据源，需要执行以下两条 SQL 语句：

- 首先通过使用 CREATE XADATASOURCE 语句创建一个或多个符合 XA 的数据类型。
- 然后用 CREATE XADATASOURCE 语句创建一个或多个符合 XA 数据源的实例。

您可以使用两阶段提交协议将 XA 数据源中的事务与 GBase 8s 事务集成，以确保事务在多个数据库之间一致提交或回滚，并在同一全局事务中管理多个外部 XA 数据源。

高可用集群的辅助服务器上不支持 CREATE XADATASOURCE 语句。

任何用户都可以根据数据库是否符合 ANSI 的状态创建遵循所有者命名规则标准的 XA 数据源。只有使用事务日志记录的数据库才支持 X/Open DTP XA 标准。

如果您包含了可选的 IF NOT EXISTS 关键字，则如果它指定名称的 XA 数据源已在当前数据库中注册，则数据库服务器不会执行任何操作（而不是向应用程序发送异常）。

XA 数据源类型和 XA 数据源实例都会指定到一个数据库。要支持分布式事务，必须在多个与外部 XA 数据源交互的数据库中创建它们。

以下语句创建一个新的 XA 数据源实例，它的名称为 gbasedbt.NewYork ，是 gbasedbt.MQSeries 类别。

```
CREATE XADATASOURCE gbasedbt.NewYork USING gbasedbt.MQSeries;
```

SQL 语句在 XA 环境中无效

当您在尝试在 X/Open 分布式事务处理环境中执行以下任何语句时， GBase 8s 数据库服务器发出错误 -701 ：

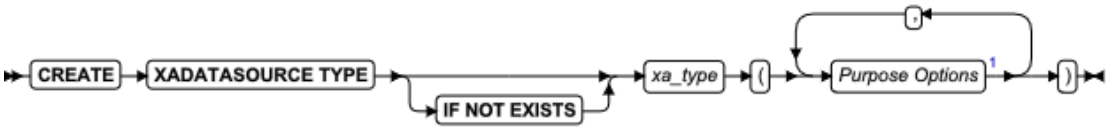
- CLOSE DATABASE
- CREATE DATABASE
- DROP DATABASE
- SET LOG
- SAVEPOINT
- RELEASE SAVEPOINT
- ROLLBACK TO SAVEPOINT

在 XA 环境中，您可以在 xa_open 调用指定当前数据库后执行一个 DATABASE 语句。但是，当选择了该数据库后，在同一会话中不能有其它 DATABASE 语句。如果您创建执行第二个 DATABASE 语句，DATABASE 语句发生错误 -701 并失败。

## 2.53 CREATE XADATASOURCE TYPE 语句

使用 CREATE XADATASOURCE TYPE 语句创建新的符合 XA 的数据源类型，并在 **sysxasourcetypes** 系统目录表中为其创建一个条目。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *xa_type* | 在此处声明新的 XA 数据源类型的名称 | 在 sysxasourcetypes 系统目录表中的 XA 数据源类型名称中必须是唯一的 | 标识符 |

**用法**

CREATE XADATASOURCE TYPE 语句向数据库中添加一个符合 XA 的数据源类型。

高可用集群的辅助服务器上不支持 CREATE XADATASOURCE TYPE 语句。

任何用户都可以根据数据库是否符合 ANSI 的状态创建遵循所有者命名规则标准的 XA 数据源类型。只有使用事务日志记录的数据库才支持 X/Open DTP XA 标准。

要创建数据源类型，您必须声明它的名称并指定 *purpose functions* 和 *purpose values* 作为 XA 源类型的属性。源类型名称后面的大多数的选项将 **sysxasourcetypes** 系统目录表中的列与 UDR 相关联。

如果您包含了可选的 IF NOT EXISTS 关键字，则如果它指定名称的 XA 数据源类型名称已在当前数据库中注册，则数据库服务器不会执行任何操作（而不是向应用程序发送异常）。

XA 数据源类型和 XA 数据源实例都会指定到一个数据库。要支持分布式事务，必须在多个与外部 XA 数据源交互的数据库中创建它们。

以下语句创建新的 XA 数据源类型，名为 MQSeries® ，被用户 gbasedbt 所有。

```
CREATE XADATASOURCE TYPE 'gbasedbt'.MQSeries(
        xa_flags    = 1,
        xa_version  = 0,
        xa_open     = gbasedbt.mqseries_open,
        xa_close    = gbasedbt.mqseries_close,
        xa_start    = gbasedbt.mqseries_start,
        xa_end      = gbasedbt.mqseries_end,
        xa_rollback = gbasedbt.mqseries_rollback,
        xa_prepare  = gbasedbt.mqseries_prepare,
        xa_commit   = gbasedbt.mqseries_commit,
        xa_recover  = gbasedbt.mqseries_recover,
        xa_forget   = gbasedbt.mqseries_forget,
        xa_complete = gbasedbt.mqseries_complete);
```

您需要为上述列出的选项提供一个值或 UDR 名称，但是您列出它们的顺序并不重要。（在此示例中目标选项的顺序与 **sysxasourcetypes** 系统目录表中列名称的顺序相对应。）

此语句成功执行后，您创建了 gbasedbt.MQSeries 类型的实例。以下语句创建了新的实例 gbasedbt.MenloPark ，它符合 XA 数据源类型 gbasedbt.MQSeries ：

```
CREATE XADATASOURCE gbasedbt.MenloPark USING gbasedbt.MQSeries;
```

## 2.54 DATABASE 语句

使用 DATABASE 语句可打开一个能访问的数据库作为当前数据库。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *database* | 数据库的名称 | 数据库必须存在 | 数据库名 |

**用法**

您可以使用 DATABASE 语句选择数据库服务器上的任何数据库。要选择另一台数据库服务器上的数据库，应指定数据库服务器的名称及数据库名。

如果用数据库名称包括当前（或另一个）数据库服务器的名称，则数据库服务器名称不可以大写。（有关指定数据库服务器名称的语法，请参阅数据库名。）

当数据库已打开时，发出 DATABASE 语句会在打开的新的数据库之前关闭当前的数据库。关闭当前数据库会释放数据库服务器的所有游标资源，使所有已声明至该点的游标无效。如果通过 SET SESSION AUTHORIZATION 语句更改了 *user* 规范，则在打开新数据库使会恢复原始的用户名。

如果先前的 CONNECT 语句已经和数据库建立了一个显式连接，而且该连接仍然是当前连接，那么在使用 DISCONNECT 语句关闭该显式连接之前，不能使用 DATABASE 语句（或任何创建隐式连接的语句）。

当前用户（或 PUBLIC）必须拥有对 DATABASE 语句中指定的数据库的“连接”特权。当前用户不可以拥有同数据库中现有角色相同的用户名。

DATABASE 语句在多语句 PREPARE 操作中是一个无效语句。

## DATABASE 执行之后立即设置 SQLCA.SQLWARN （ESQL/C）

在 DATABASE 执行之后，您可以立即通过检查 sqlca 结构中的警告标志标识指定数据库的特征。

- 如果 **sqlca.sqlwarn** 的第一个字段为空白，则不发出任何警告。
- 如果被打开的***数据库***支持事务日志记录，则第二个 **sqlca.sqlwarn** 字段设置为字母 W。
- 如果***数据库***是一个兼容 ANSI 的数据库，则第三个字段设置为 W 。
- 如果***数据库***是 GBase 8s 数据库，则第四个字段设置为 W 。
- 如果***数据库***将所有的浮点数据转换成 DECIMAL 格式，则第五个字段设置为 W。（系统缺少 FLOAT 和 SMALLFLOAT 支持。）
- 如果***数据库***是数据复制对中的次触发器（即在只读方式下运行），则第七个字段设置为 W。
- 如果***数据库***将 **DB_LOCALE** 设置为一个不同于客户机系统上 **DB_LOCALE** 设置的语言环境，则第八个字段设置为 W 。

## EXCLUSIVE 关键字

EXCLUSIVE 关键字以互斥方式打开数据库，并防止除当前用户之外的任何人访问。要允许其他人访问数据库，您必须先执行 CLOSE DATABASE 语句，然后在不带 EXCLUSIVE 关键字的情况下重新打开数据库。

以下语句以互斥方式打开 training 数据库服务器上的 stores_demo 数据库：

```
DATABASE stores_demo@training EXCLUSIVE;
```

如果另一个用户已打开指定的数据库，则拒绝互斥访问并返回一个错误，而且不打开任何数据库。

如果您遇到该错误，但是您无法确认是否有其它用户连接了此数据库，则在 Scheduler API 上运行由传感器或任务导致的非互斥访问。要暂时禁止此 Scheduler，您可以发出此 SQL 管理 API 命令：

```
EXECUTE FUNCTION admin('scheduler shutdown');
```

admin('scheduler shutdown') 例程执行完毕后，重新尝试 DATABASE ... EXCLUSIVE 语句。

有关 Scheduler API 命令的更多信息，请参阅 *GBase 8s 管理员指南*。有关调用 SQL 管理 API 函数必须持有的权限的信息，请参阅 *GBase 8s 管理员参考手册*。

## 2.55 DEALLOCATE COLLECTION 语句

使用 DEALLOCATE COLLECTION 语句释放先前通过 ALLOCATE COLLECTION 语句分配的集合变量的内存。

该语句是 SQL ANSI/ISO 标准的扩展。在 ESQL/C 中使用此语句。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *variable* | 标识要解除为其分配的内存的已归类的或未归类的集合变量的名称 | 必须是已分配的 GBase 8s ESQL/C 集合变量的名称 | 名称必须符合针对变量名称制定的语言特定规则 |

**用法**

DEALLOCATE COLLECTION 语句释放与 ***variable*** 标识的 GBase 8s ESQL/C 集合变量相关联的所有内存。您必须使用 DEALLOCATE COLLECTION 显式地释放集合变量的内存资源。否则，解除分配会在程序结束时自动发生。

DEALLOCATE COLLECTION 语句会释放已归类的和未归类的集合变量的资源。

**提示：** DEALLOCATE COLLECTION 语句只解除为 GBase 8s ESQL/C 集合变量分配的内存。要解除为 GBase 8s ESQL/C 行变量分配的内存，应使用 DEALLOCATE ROW 语句。

如果要解除分配一个不存在的集合变量或不是 GBase 8s ESQL/C 集合变量的变量，会导致错误。一旦解除分配一个集合变量，即可使用 ALLOCATE COLLECTION 来重新分配资源，然后重新使用一个集合变量。

此示例显示了如何使用 DEALLOCATE COLLECTION 语句解除分配给未归类的集合变量 a_set 的资源：

```
EXEC SQL BEGIN DECLARE SECTION;
```

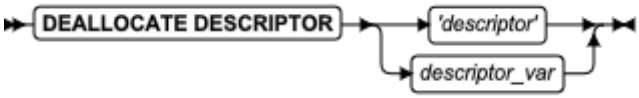
```
client collection a_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_set;
. . .
EXEC SQL deallocate collection :a_set;
```

相关示例，请参阅相关的概念，插入集合游标。

## 2.56 DEALLOCATE DESCRIPTOR 语句

使用 DEALLOCATE DESCRIPTOR 语句可释放先前分配的系统描述符区域。在 GBase 8s ESQL/C 中使用此语句。

**语法**




该语句是 SQL 语言的 ANSI/ISO 标准的扩展。

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 系统描述符区域的名称 | 使用单引号。必须已分配系统描述符区域。 | 引用字符串 |
| *descriptor_var* | 包含系统描述符区域的名称的主变量 | 必须已分配系统描述符区域，并且必须已声明变量 | 名称必须符合特定于语言的规则 |

**用法**

DEALLOCATE DESCRIPTOR 语句会释放所有与 ***descriptor*** 或 ***descriptor_var*** 标识的系统描述符区域相关联的内存。它也会释放所有项描述符（包括用于项描述符中的数据项的内存）。

您可以在解除分配之后重新使用描述符或描述符变量。否则，解除分配会在程序结束时自动发生。

如果对不存在的描述符或描述符变量执行解除分配操作，则会导致错误。

你不可以使用 DEALLOCATE DESCRIPTOR 语句解除对 **sqlda** 结构的分配。只可以用它释放为系统描述符区域分配的内存。

以下示例显示了有效的 DEALLOCATE DESCRIPTOR 语句。第一行使用了一个嵌入变量的名称，第二行使用了一个用引号引起的字符串以标识分配的系统描述符区域。

```
EXEC SQL deallocate descriptor :descname;

EXEC SQL deallocate descriptor 'desc1';
```

## 2.57 DEALLOCATE ROW 语句

使用 DEALLOCATE ROW 语句可释放 ROW 变量的内存。

该语句是 SQL ANSI/ISO 标准的扩展。在 ESQL/C 中使用此语句。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *variable* | 归类的或未归类的行变量 | 必须被声明和分配 | 特定于语言 |

**用法**

DEALLOCATE ROW 释放与 ***variable*** 标识的 GBase 8s ESQL/C 已归类的或未归类的行变量相关联的所有内存。如果没有使用 DEALLOCATE ROW 显式地释放内存资源，程序结束时将自动执行释放。要解除 GBase 8s ESQL/C 集合变量分配的内存，可使用 DEALLOCATE COLLECTION 语句。

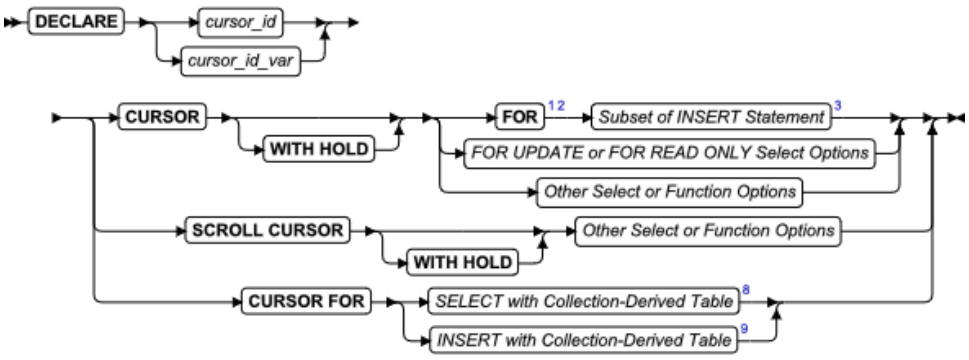
在解除分配 ROW 变量之后，可以使用 ALLOCATE ROW 语句重新分片资源，然后重新使用一个 ROW 变量。下面的示例显示了如何使用 DEALLOCATE ROW 语句解除为 ROW 变量 a_row 分配的资源：

```
EXEC SQL BEGIN DECLARE SECTION; row (a int, b int) a_row;
        EXEC SQL END DECLARE SECTION;
        . . .
        EXEC SQL allocate row :a_row;
        . . .
        EXEC SQL deallocate row :a_row;
```

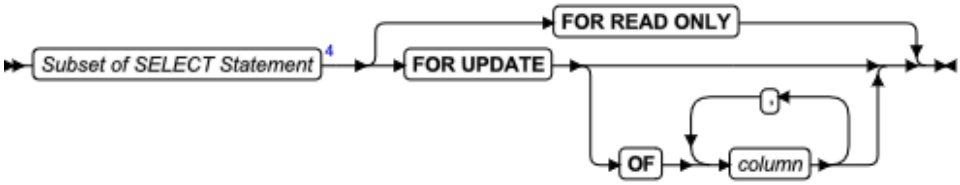
## 2.58 DECLARE 语句

使用动态 SQL 的 DECLARE 语句声明游标，并将它与一个向 GBase 8s ESQL/C 或 SPL 例程返回一组行的 SQL 语句相关联。

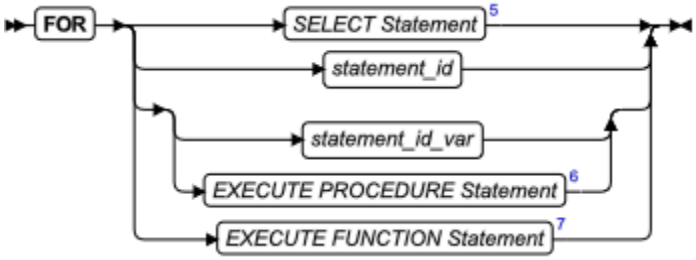
**语法**




FOR UPDATE 或 FOR READ ONLY Select 选项




其它 Select 或 Function 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要使用游标更新的列 | 必须存在，但不需要列在 Projection 子句的 Select 列表中 | 标识符 |
| *cursor_id* | 此处为游标声明的名称 | 在游标的名称以及准备好的对象的名称中（在 SPL 中是变量）必须唯一 | 标识符 |
| *cursor_id_var* | 持有 *cursor_id* 的变量 | 必须有一个字符数据类型 | 特定于语言 |
| *statement_id* | 准备好的语句的名称 | 在 PREPARE 语句中声明 | 标识符 |
| *statement_id_var* | 持有 *statement_id* 的变量 | 必须有一个字符数据类型 | 特定于语言 |

## 用法

正如所指出的除外，下节描述了在 GBase 8s ESQL/C 例程中如何使用 DECLARE 语句。有关 SPL 例程中 DECLARE 语句的语法和语义更多的限制，请参阅在 SPL 例程中声明动态游标。

*游标*是一个与一组行相关联的标识符。DECLARE 语句将游标与以下一个数据库对象相关联：

- 使用一个 SQL 语句，例如 SELECT 、EXECUTE FUNCTION（或 EXECUTE PROCEDURE）或 INSERT。

  这些 SQL 语句中的每一个会创建一个不同类型的游标。有关更多信息，请参阅游标类型的概述。

- 使用准备好的语句的语句标识符（***statement id*** 或 ***statement id variable***）

  您可以准备一个先前的 SQL 语句，并将该准备好的语句与游标相关联。有关更多信息，请参阅将游标与准备好的语句相关联。

- 使用 GBase 8s ESQL/C 程序中的集合变量

  集合变量的名称出现在 SELECT 的 FROM 子句或 INSERT 的 INTO 子句。有关更多信息，请参阅将游标与准备好的语句相关联。

DECLARE 分配一个标识符给游标，指定它的用法并指示 GBase 8s ESQL/C 预处理器为它分配存储器。在程序执行期间，DECLARE 必须在任何其它引用该游标的语句之前执行。

在单个程序中可以共同存在的游标和准备好的对象的数目由可用的内存限制。要避免超出限制，可使用 FREE 语句释放一些准备好的语句或游标。

ESQL/C 程序可以由一个或多个源代码文件组成。在缺省情况下，因为引用游标的作用域是全局到程序，所以在一个源文件中声明的游标可以被另一个文件中语句引用。在多文件程序中，如果希望将游标名称的作用域限制到声明这些游标名称的文件，则必须使用 -local 命令行选项预先处理所有文件。

可以为同一个准备好的语句标识符声明多个游标。例如，下面的 GBase 8s ESQL/C 示例不会返回错误：

- EXEC SQL prepare id1 from 'select * from customer';
- EXEC SQL declare x cursor for id1;
- EXEC SQL declare y scroll cursor for id1;
- EXEC SQL declare z cursor with hold for id1;

如果包含 -ansi 编辑标志（或者如果设置了 DBANSIWARN ），那么会针对使用动态游标名称或动态语句标识符的语句生成警告。一些错误检查会在运行时执行，如以下这些典型检查：

- 将顺序游标作为滚动游标无效的使用
- 使用未声明的游标
- 无效的游标名称或语句名称（空）

建档游标或语句作为标识符指定时，会在编译的时候执行对同一名称的一个游标的多个声明的检查。下面示例使用主变量存储游标名称：

```
EXEC SQL declare x cursor for select * from customer;
        . . .
        stcopy("x", s);
EXEC SQL declare :s cursor for select * from customer;
```

游标使用排列顺序，该顺序在游标被声明时是有效的，即使这不同于运行时会话的排列顺序。

## 游标类型的概述

在典型情况下，对多行数据（或对 GBase 8s ESQL/C 集合变量）的数据操纵语言（DML）操作需要游标。您可以使用 DECLARE 语句声明一下类型的游标：

- ***Select 游标***是与 SELECT 语句相关联的游标。
- ***函数游标***是与 EXECUTE FUNCTION （或 EXECUTE PROCEDURE）语句相关联的游标。
- ***Insert 游标***是与 INSERT 语句相关联的游标。

后面的章节描述了这些游标的类型中的每一中类型。游标也可以有***顺序***、***滚动***和***控制***特征（但是 Insert 游标不可是一个滚动游标）。这些特征确定了游标的结构；请参阅 游标特性。此外，Select 游标或函数游标可以指定***只读***或***更新***方式。有关更多信息，请参阅 Select 游标或 Function 游标。

**提示：** 函数游标的行为与作为更新游标启用的 Select 游标相同。

除了将游标直接与 SQL 语句的内容相关联，DECLARE 语句的 FOR 关键字可以紧跟在准备好的 SQL 语句的标识符后面，并将此游标与动态准备好的 INSERT 、SELECT 、EXECUTE FUNCTION （或 EXECUTE PROCEDURE）语句的结果集相关联。该功能可以使您在不同时刻将不同的语句与同一个游标使用。在这种情况下，游标的类型取决于打开游标时 statement_id 或 statement_id 变量指定的准备好的语句。（有关更多信息，请参阅将游标与准备好的语句相关联。）

## Select 游标或 Function 游标

当 SQL 语句将多组值返回到一个 GBase 8s ESQL/C 程序中时，必须声明一个游标以保存多组或行数据并且以一次访问一行的方式访问这些行。您必须将以下 SQL 语句与游标相关联：

- 如果将 SELECT 语句与游标相关联，则该游标称为 ***Select 游标***。

  Select 游标是一个数据结构，代表 SELECT 语句检索到行的活动集合内特定位置。

- 如果将 EXECUTE FUNCTION（或者 EXECUTE PROCEDURE）语句与游标相关联，则该游标称为***函数游标***。

  函数游标代表用户定义的函数返回的列或值。函数游标的行为与作为更新游标启用的 Select 游标相同。

在 GBase 8s 中，对于后向兼容性，如果一个 SPL 函数是使用 CREATE PROCEDURE 语句创建的，那么可以使用 EXECUTE PROCEDURE 语句创建一个函数游标。对于外部函数，必须使用 EXECUTE PROCEDURE 语句。

当您将 SELECT 或 EXECUTE FUNCTION （或 EXECUTE PROCEDURE）语句与一个游标相关联时，该语句可以包括 INTO 子句。但是，如果准备 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句，您必须省略 PREPARE 语句中的 INTO 子句并使用 FETCH 语句的 INTO 子句从集合游标检索值。

Select 或函数游标可以扫描返回的数据行，并逐行数据移动到接收变量的集合中，如下面的步骤所述：

1. DECLARE

   使用 DECLARE 定义一个游标并将它与一个语句相关联。

2. OPEN

   使用 OPEN 打开该游标。数据库服务器处理查询，直到它定位或构造活动集的第一行。

3. FETCH

   使用 FETCH 从游标检索连续行的数据。

4. CLOSE

   使用 CLOSE 在不再需要游标的活动集时关闭游标。

5. FREE

   使用 FREE 释放为游标分配的资源。

## 使用 FOR READ ONLY 选项

使用 FOR READ ONLY 关键字可将游标定义为只读游标。声明为只读的游标不可以用于更新（或删除）任何它得到的行。

对 FOR READ ONLY 关键字的需要取决于数据库是符合 ANSI 的还是不符合 ANSI 。

因为在不兼容 ANSI 的数据库中，DECLARE 语句定义的游标在缺省情况下是只读游标，所以如果想将游标成为一个只读游标，您不需要指定 FOR READ ONLY 关键字。显式地指定 FOR READ ONLY 关键字的唯一好处是为了更好地编制程序文档。

在兼容 ANSI 的数据库中，如果 SELECT 语句符合与游标相关联的 SELECT 语句的子集 中列出的对更新游标的所有限制，那么通过 DECLARE 语句与 SELECT 语句相关联的游标在缺省情况下是一个更新游标。如果想让一个 Select 游标成为只读，则必须在声明该游标时使用 FOR READ ONLY 关键字。

数据库服务器可以针对只读游标使用比针对更新游标更为宽松的锁定。

下面的示例创建了一个只读游标：

```
EXEC SQL declare z_curs cursor for
        select * from customer_ansi
        for read only;
```

## 使用 FOR UPDATE 选项

使用 FOR UPDATE 选项可声明更新游标。您可以使用更新游标修改（更新或删除）当前的行。

在兼容 ANSI 的数据库中，如果一个 Select 游标不是使用 FOR READ ONLY 关键字声明的，并且该游标遵守在与游标相关联的 SELECT 语句的子集中描述的对更新游标的限制，那么可以使用该 Select 游标更新或删除数据。在声明游标时不需要使用 FOR UPDATE 关键字。

下面的示例声明一个更新游标：

```
EXEC SQL declare new_curs cursor for
        select * from customer_notansi
        for update;
```

在更新游标中，可以更新或是删除活动集中的行。在创建更新游标之后，可以通过使用带 WHERE CURRENT OF 子句的 UPDATE 或 DELETE 语句更新或删除当前选择的行。单词 CURRENT OF 是指最近取得的行；它们替代 WHERE 子句中通常的测试表达式。

更新游标能让您执行使用 UPDATE 语句不可能实现的更新操作，因为做出更新的决定以及新数据项的值取决于行的原始内容。您的程序可以在决定是否更新之前评估或操纵选择的数据。UPDATE 语句无法询问正在更新的表。

您可以指定那些可以被更新的特殊列。这些列不需要出现在 Projection 子句的 Select 列表中。

### *连同列的列表使用 FOR UPDATE*

当声明更新游标时，您可以通过包含 OF 关键字和列的列表将更新操作限制在特定的列。您只可以修改随后 UPDATE 语句中的那些命名的列。列不需要在 SELECT 子句的选择列表中。

下一个示例声明了一个更新游标，并指定该游标只可以更新 customer_notansi 表中的 fname 和 lname 列：

```
EXEC SQL declare name_curs cursor for
        select * from customer_notansi
        for update of fname, lname;
```

缺省情况下，除非声明为 FOR READ ONLY ，否则在兼容 ANSI 的数据库中的 Select 游标是一个更新游标，因此 FOR UPDATE 关键字是可选的。但是，如果希望更新游标只能够修改表中的一些列，则必须在 FOR UPDATE OF ***column*** 列表中指定这些列。

指定列的主要好处是便于文档的编制并防止编程出错。（数据库服务器拒绝更新任何其它列。）另一个好处是在 SELECT 语句满足以下条件时性能得到增强：

- 可以使用索引处理 SELECT 语句。
- 列出的列不是用于处理 SELECT 语句的索引的一部分。

如果打算更新的列是用于处理 SELECT 语句的索引的一部分，则数据库服务器会保留一个列表，其中列有每个更新的行，从而确保没有任何行被更新两次。如果 OF 关键字指定了可以更新的列，则数据库服务器便会确定是否要保留更新行的列表。如果数据库服务器确定保留该列表的工作已不再需要，则性能随之提高。如果不使用 OF ***column*** 列表，数据库服务器通常会维护一个更新行的列表，尽管此列表可能并不需要。

下面的示例包含将更新游标同 DELETE 语句使用删除当前行的 GBase 8s ESQL/C 代码。

每当删除行时，游标仍处于行之间。在删除数据之后，必须在可以引用 DELETE 或 UPDATE 语句中的游标之前使用 FETCH 语句将游标推进到下一行。

```
EXEC SQL declare q_curs cursor for
    select * from customer where lname matches :last_name for update;

EXEC SQL open q_curs;
for (;;)
{
    EXEC SQL fetch q_curs into :cust_rec;
```

```
    if (strncmp(SQLSTATE, "00", 2) != 0)
        break;

    /* Display customer values and prompt for answer */
    printf("\n%s %s", cust_rec.fname, cust_rec.lname);
    printf("\nDelete this customer? ");
    scanf("%s", answer);

    if (answer[0] == 'y')
        EXEC SQL delete from customer where current of q_curs;
    if (strncmp(SQLSTATE, "00", 2) != 0)
        break;
}
printf("\n");
EXEC SQL close q_curs;
```

### *使用 Update 游标进行锁定*

FOR UPDATE 关键字告知数据库服务器更新操作是可能的，并会引发数据库服务器使用比对 Select 游标更严格的锁定。您声明一个更新游标可让数据库服务器知道程序可能更新（或删除）它取得的作为 SELECT 语句一部分的任何行，更新游标为程序取得的行使用***可升级锁定***（也称***写锁***）。在程序修改行之前，会将行锁提升为互斥锁。

使用 WITH HOLD 关键字声明一个更新游标是可能的，但是这样做的唯一原因是将一长串的更新打断成较小的事务。您必须在同一个事务中取得并更新特定的行。

如果操作涉及取和更新大量的行，则数据库服务器维护的锁表会溢出。防止这种溢出的通常方式是锁定整个正在更新的表。如果此操作是不可能的，那么一种替换的方式是通过 Hold 游标来更新,并按时间间隔执行 COMMIT WORK 。但是，您必须谨慎地计划这样的应用程序，因为 COMMIT WORK 会释放所有的锁，甚至是那些通过 Hold 游标放置的锁。

## 与顺序游标相关联的 INSERT 语句的子集

如 DECLARE 语句的程序中所述，要创建一个 Insert 游标，应将顺序游标与 INSERT 语句的限制格式相关联。INSERT 语句必须包括 VALUES 子句；它不可以包含嵌入的 SELECT 语句。

下面的示例包含声明 Insert 游标的 GBase 8s ESQL/C 代码：

```
EXEC SQL declare ins_cur cursor for
        insert into stock values
        (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

Insert 游标简单地插入数据行；它不可以用于取得数据。当打开 Insert 游标时，会在内存中创建一个缓冲区。该缓冲区在程序执行 PUT 语句时接收数据行。只有在缓冲区满的时候才会将这些行写入磁盘。您可以在缓冲区还未满时使用 CLOSE 、FLUSH 或 COMMIT WORK 语句刷新缓冲区（即将它的内容写入数据库）。该主题会在 CLOSE 、FLUSH 和 PUT 语句下做进一步讨论。

您必须在程序结束之前关闭 Insert 游标，从而将任何已缓冲的行插入数据库。如果没有正确关闭游标，则会丢失数据。有关对 INSERT 语法和使用的完整讨论，请参阅 INSERT 语句。

## Insert 游标

当您将 INSERT 语句与一个游标相关联的时候，该游标称为 ***Insert 游标***。Insert 游标是一个数据结构，它代表 INSERT 语句将要添加到数据库的行。INSERT 游标简单地插入数据行；它不可以用于取得数据。要创建 Insert 游标，应将一个游标与 INSERT 语句的限制格式相关联。INSERT 语句必须包括 VALUES 子句；它不可以包含嵌入的 SELECT 语句。

如果想要在 INSERT 操作中将多个行添加到数据库，应创建一个 Insert 游标。Insert 游标允许将大量的插入数据缓存到内存中，并在缓冲区满的时候写入磁盘，具体如以下这些步骤所述：

1. 使用 DECLARE 可为 INSERT 语句定义一个 Insert 游标。
2. 使用 OPEN 语句打开游标。数据库服务器会在内存中创建缓冲区。并将游标的位置确定在插入缓冲区的第一行。
3. 使用 PUT 语句将连续的数据行复制到插入缓冲区。
4. 数据库服务器只有在缓冲区满的时候才会将这些行写到磁盘。您可以在缓冲区还未满的时候使用 CLOSE 、FLUSH 或 COMMIT WORK 语句刷新缓冲区。该主题会在 PUT 和 CLOSE 语句下做进一步讨论。
5. 当不再需要 Insert 游标时使用 CLOSE 语句关闭游标。您必须在查询结束之前关闭 Insert 游标，从而将任何已缓冲的行插入数据库。如果没有正确关闭游标，则会丢失数据。
6. 使用 FREE 语句释放游标。FREE 语句会释放为 Insert 游标分配的资源。

使用 Insert 游标比直接嵌入 INSERT 语句更有效。此进程减少了程序和数据库服务器之间的通信，也加快了插入的速度。

Insert 游标还具有顺序游标特征。要创建 Insert 游标，可将一个顺序游标与 INSERT 语句额限制格式相关联。（有关更多信息，请参阅 Insert 语句。）以下示例包含声明顺序 Insert 游标的 GBase 8s ESQL/C 代码：

```
EXEC SQL declare ins_cur cursor for
insert into stock values (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

## 游标特性

您可以将一个游标声明为***顺序***游标（缺省值）、***滚动***游标（通过使用 SCROLL 关键字）或***保持***游标（通过使用 WITH HOLD 关键字）。SCROLL 和 WITH HOLD 关键字并不互相排斥。后面的章节解释了这些结构特征。

Select 或 Function 游标可以是一个顺序游标或滚动游标。Insert 游标只可以是一个顺序游标。在 ESQL/C 例程中，选择游标、函数游标和 Insert 游标可以有选择地成为保持游标。（在 SPL 例程中，所有的游标都是顺序游标，但是只有 Select 游标可以是保持游标。）

## 缺省创建顺序游标

如果只使用 CURSOR 关键字，则会创建一个顺序游标，它只可以从活动集合按顺序取得下一行。每次打开顺序游标时，顺序游标只可以从活动集合读一次。

如果正在为一个 Select 游标使用顺序游标，则在每次执行 FETCH 语句时，数据库服务器会返回当前行的内容，并且找到活动集合中的下一行。

以下示例在一个不兼容 ANSI 的数据库中创建了一个只读顺序游标，在一个兼容 ANSI 的数据库汇总创建了一个更新顺序游标：

```
EXEC SQL declare s_cur cursor for
        select fname, lname into :st_fname, :st_lname
        from orders where customer_num = 114;
```

Insert 游标也具有顺序游标的特征。要创建一个 Insert 游标，请将顺序游标与 INSERT 语句的限制格式相关联。（有关更多信息，请参阅 Insert 游标。）以下示例声明了一个 Insert 游标：

```
EXEC SQL declare ins_cur cursor for
        insert into stock values
        (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

## 使用 SCROLL 关键字创建滚动游标

使用 SCROLL 关键字创建滚动游标，它可以任何顺序取活动集合的行。

数据库服务器将游标的活动集合保留为一个临时表，直到关闭游标。您可以取得活动集合的第一行、最后一行或任何一个中间行，以及在不必关闭再重新打开游标的情况下重复取得一些行。（请参阅 FETCH。）

多用户系统中，在派生出活动集合的行的表中的行可能在打开游标之后有所更改，并且会在临时表中生成一个副本。如果在一个事务内使用滚动游标，那么通过将隔离级别设置为“可重复读取”或通过在事务期间以共享方式锁定整个表可以防止更改复制的行。（请参阅 SET ISOLATION 和 LOCK TABLE。）

以下示例为 SELECT 语句创建了一个滚动游标：

```
DECLARE sc_cur SCROLL CURSOR FOR SELECT * FROM orders;
```

您可以将 Select 和 Function 游标创建滚动游标，但不可为 Insert 游标创建滚动游标。不可将滚动游标声明为 FOR UPDATE。

## 使用 WITH HOLD 关键字创建保持游标

使用 WITH HOLD 关键字创建 *hold 游标*。Hold 游标允许跨多个事务对行的集合进行不中断的访问。

通常，所有的游标在事务结束时关闭。Hold 游标不会关闭；它在事务结束之后仍保持打开状态。

Hold 游标可以是一个顺序游标或（在 ESQL/C 中）滚动游标。

在 SPL 例程中 WITH HOLD 关键字只对 Select 游标有效。有关 DECLARE 语句在 SPL 例程中的语法，请参阅 在 SPL 例程中声明动态游标。

在 ESQL/C 中，可以使用 WITH HOLD 关键字声明 Select 和函数游标（具有顺序和滚动属性）也可以声明 Insert 游标。这些关键字在 DECLARE 语句中跟在 CURSOR 关键字之后。以下示例为 SELECT 创建了一个顺序 Hold 游标：

```
DECLARE hld_cur CURSOR WITH HOLD FOR
        SELECT customer_num, lname, city FROM customer;
```

您可以如下面的 GBase 8s ESQL/C 代码示例所示使用选择保持游标。此代码段使用一个 Hold 游标作为主游标来扫描一个记录集合，使用一个顺序游标作为细节游标俩指向位于不同表中的记录。主游标扫描的记录是更新细节游标指向的记录的基础。在第一个 WHILE 循环的每个迭代的结束处的 COMMIT WORK 语句将 Hold 游标 **c_master** 保留为打开状态，但关闭顺序游标 **c_detail** 并释放所有锁。这种技术最小化了数据库服务器必须分配给锁和未完成的事务的资源，并且它使其它用户能够立即访问更新的行。

```
EXEC SQL BEGIN DECLARE SECTION;
        int p_custnum, int save_status; long p_orddate;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL prepare st_1 from
        'select order_date from orders where customer_num = ? for update';
        EXEC SQL declare c_detail cursor for st_1;
        EXEC SQL declare c_master cursor with hold for
        select customer_num from customer where city = 'Pittsburgh';

        EXEC SQL open c_master;
        if(SQLCODE==0) /* the open worked */
        EXEC SQL fetch c_master into :p_custnum; /* discover first customer */
        while(SQLCODE==0) /* while no errors and not end of pittsburgh customers */
        {
        EXEC SQL begin work; /* start transaction for customer p_custnum */
        EXEC SQL open c_detail using :p_custnum;
        if(SQLCODE==0) /* detail open succeeded */
        EXEC SQL fetch c_detail into :p_orddate; /* get first order */
        while(SQLCODE==0) /* while no errors and not end of orders */
        {
        EXEC SQL update orders set order_date = '08/15/94'
        where current of c_detail;
        if(status==0) /* update was ok */
        EXEC SQL fetch c_detail into :p_orddate; /* next order */
        }
        if(SQLCODE==SQLNOTFOUND) /* correctly updated all found orders */
        EXEC SQL commit work; /* make updates permanent, set status */
        else /* some failure in an update */
        {
```

```
save_status = SQLCODE; /* save error for loop control */
EXEC SQL rollback work;
SQLCODE = save_status; /* force loop to end */
}
if(SQLCODE==0) /* all updates, and the commit, worked ok */
EXEC SQL fetch c_master into :p_custnum; /* next customer? */
}
EXEC SQL close c_master;
```

使用 CLOSE 语句显式地关闭 Hold 游标，或使用 CLOSE DATABASE 或 DISCONNECT 语句隐式地关闭 Hold 游标。CLOSE DATABASE 语句关闭所有游标。

### *使用保持 Insert 游标*

如果您将保持游标与 INSERT 语句相关联，则可以使用事务将一长串的 PUT 语句打断为较小的 PUT 语句集合。除了等待 PUT 语句填充缓冲区并引发自动写入数据库，您可以执行 COMMIT WORK 语句刷新行缓冲区。使用 Hold 游标，COMMIT WORK 会提交插入的行，但将游标保留为打开状态以供进一步插入。当插入大量的行时，这样方法是期望的，因为暂时的未提交的工作会消耗数据库服务器的资源。

## 与游标相关联的 SELECT 语句的子集

如 DECLARE 语句的语法表所示，不是所有的 SELECT 语句都可以与只读或更新游标相关联。

如果 DECLARE 语句包含 FOR READ ONLY 或 FOR UPDATE 选项，则必须遵守对 DECLARE 语句中包括的 SELECT 语句的某些限制（直接地或作为准备好的语句）。

如果 DECLARE 语句包括 FOR READ ONLY 选项，则 SELECT 语句不可以有 FOR READ ONLY 或 FOR UPDATE 选项。（有关 SELEC 语法和用法的描述，请参阅 SELECT 语句。）

如果 DECLARE 语句包括 FOR UPDATE 选项，则 SELECT 语句必须服从下列限制：

- 该语句只能从一个表中选择数据。
- 该语句不能包含任何聚集函数。
- 该语句还不能包含 FOR UPDATE 关键字。
- 该语句不可以包括任何以下子句或关键字：DISTINCT 、EXCEPT 、FOR READ ONLY 、FOR UPDATE 、GROUP BY 、INTERSECT 、INTO TEMP 、MINUS 、ORDER BY 、UNION 或 UNIQUE。

## 不兼容 ANSI 的数据库中的游标的示例

在不兼容 ANSI 的数据库中，与 SELECT 语句相关联的游标在缺省情况下是一个只读游标。以下的示例在一个不兼容 ANSI 的数据库中声明了一个只读游标：

```
EXEC SQL declare cust_curs cursor for
        select * from customer_notansi;
```

如果想要在程序代码中明确此游标为只读游标，应指定 FOR READ ONLY 选项，如下面的示例所示：

```
EXEC SQL declare cust_curs cursor for
        select * from customer_notansi for read only;
```

如果要将这个游标变为一个更新游标，可在 DECLARE 语句中指定的 FOR UPDATE 选项。此示例声明了一个更新游标：

```
EXEC SQL declare new_curs cursor for
        select * from customer_notansi for update;
```

如果要使一个更新游标只能够修改表中的一些列，则必须在 FOR UPDATE 子句中指定那些列。下面的示例声明了一个只可以更新 customer_notansi 表中 fname 和 lname 列的更新游标：

```
EXEC SQL declare name_curs cursor for
        select * from customer_notansi for update of fname, lname;
```

## 兼容 ANSI 的数据库中的游标的示例

在兼容 ANSI 的数据库中，与 SELECT 语句相关联的游标在缺省情况下是一个更新游标。

下面的示例声明了一个兼容 ANSI 的数据库中的更新游标：

```
EXEC SQL declare x_curs cursor for select * from customer_ansi;
```

要在程序文档中声明此游标为更新游标，您可以指定 FOR UPDATE 选项，如下面的示例所示：

```
EXEC SQL declare x_curs cursor for
        select * from customer_ansi for update;
```

如果希望一个更新游标只能够修改表中的一些列，则必须在 FOR UPDATE 选项中指定这些列。下面的示例声明了一个更新游标，并指定该游标只可以更新 customer_ansi 表中的 fname 和 lname 列：

```
EXEC SQL declare y_curs cursor for
        select * from customer_ansi for update of fname, lname;
```

如果要将一个游标变为只读游标，则必须通过在 DECLARE 语句中指定 FOR READ ONLY 选项来覆盖 DECLARE 语句的缺省行为。下面的示例声明了一个只读游标：

```
EXEC SQL declare z_curs cursor for
        select * from customer_ansi for read only;
```

## 将游标与准备好的语句相关联

PREPARE 语句让您在运行时聚集 SQL 语句的文本，并将语句文本传送到数据库服务器以供执行。如果您期望一个能返回值的动态准备好的 SELECT 、EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句可生成多行数据，在必须将该准备好的语句与一个游标相关联。（请参阅 PREPARE。）

PREPARE 语句的结果是一个语句标识符（***statement id*** 或 ***id variable***），它是一个代表准备好的语句文本的数据结构，要为语句文本声明一个游标，应将该游标与语句标识符相关联。

您可以将一个顺序游标与任何准备好的 SELECT 或 EXECUTE FUNCTION （或 EXECUTE PROCEDURE）语句相关联。您不可以将一个滚动游标与准备好的 INSERT 语句或准备好包括 FOR UPDATE 子句的 SELECT 语句相关联。

在打开、使用和关闭游标之后，可以在相同的语句标识符下准备一个不同的语句。以这种方式，一个单独的游标可能在不同的时刻与不同的语句使用。再次使用游标之前，必须重新声明该游标。

下面的示例包含准备 SELECT 语句并为准备好的语句文本声明顺序游标的 GBase 8s ESQL/C 代码。首先通过返回值的 SELECT 语句准备语句标识符 st_1 ；然后为 st_1 声明游标 c_detail。

```
EXEC SQL prepare st_1 from
        'select order_date
        from orders where customer_num = ?';
        EXEC SQL declare c_detail cursor for st_1;
```

如果想要使用准备好的 SELECT 语句修改数据，可将 FOR UPDATE 子句添加到想要准备的语句文本，如下面的 GBase 8s ESQL/C 示例所示：

```
EXEC SQL prepare sel_1 from
        'select * from customer for update';
        EXEC SQL declare sel_curs cursor for sel_1;
```

更改表结构的 DDL 操作可以使使其相关联的准备语句或相关联的例程引用修改的表的游标无效，除非准备好的对象被重新编译，或者除非例程被重新优化。有关更多信息，请参阅对游标引用的表的 DDL 操作。

DECLARE 语句允许您为 GBase 8s ESQL/C 集合变量声明游标。这样的游标称为***集合游标***。使用集合变量可访问集合（SET 、MULTISET 、LIST）列的元素。当要访问集合变量中的一个或多个元素的时候，请使用游标。

集合派生的表这一段标识了要为其声明游标的集合变量。有关更多信息，请参阅集合派生表。

## 用集合派生的表选择

DECLARE 语句的图表参考了本节。

要为集合变量声明 Select 游标，应将集合派生的表这一段同与集合游标相关联的 SELECT 语句包括在一起。一个 Select 游标允许您从集合变量选择一个或多个元素。（有关对 SELECT 语句和使用的描述，请参阅 SELECT 语句。）

当为集合变量声明一个 Select 游标时，DECLARE 语句有以下限制：

- 它不可以将 FOR READ ONLY 关键字包括为游标方式。

  Select 游标是一个更新游标。
- 它不能包括 SCROLL 或 WITH HOLD 关键字。

  Select 游标必须是一个顺序游标。

另外，与集合游标相关联的 SELECT 语句有以下限制：

- 它不可以包括以下子句或选项：WHERE 、GROUP BY 、ORDER BY 、HAVING 、INTO TEMP 和 WITH REOPTIMIZATION 。
- 它不可以在选择列表中包含表达式。
- 如果集合包含不透明、Distinct、内置或其它集合数据类型的元素，则选择列表必须是一个星号（ * ）。
- 选择列表中的列名称必须是简单的列名称，不带限定符。

  这些列不能使用以下语法：

  database@server:table.column --INVALID SYNTAX
- 它必须在 FROM 子句中指定集合变量的名称。

  您不可以为集合变量指定输入参数（问号（?））。同样您不可以使用实际表格式的“集合派生的表”的段。

### *使用 Select 游标和集合变量*

包括带集合派生的表子句的 SELECT 语句的集合游标提供了对集合变量中的元素的访问。

要选择多个元素：

1. 在 GBase 8s ESQL/C 程序中创建一个客户端集合变量。
2. 使用 DECLARE 语句为 SELECT 语句声明集合游标。
3. 要修改集合变量的元素，可使用 FOR UPDATE 关键字将选择游标声明为一个更新游标。然后您可以使用 DELETE 和 UPDATE 语句的 WHERE CURRENT OF 子句删除或更新集合的元素。
4. 使用 OPEN 语句打开此游标。
5. 使用 FETCH 语句和 INTO 子句从集合游标取得元素。
6. 如果需要，则对取得的数据执行任何更新或删除，并将修改的集合变量保存集合列中。

   一旦集合变量包含正确的元素，即可使用 UPDATE 或 INSERT 语句将集合变量的内容保存在实际的集合列（SET 、MULTISET 或 LIST）。
7. 使用 CLOSE 语句关闭此集合游标。

此 DECLARE 语句为一个集合变量声明一个 Select 游标：

```
EXEC SQL BEGIN DECLARE SECTION;
  client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL declare set_curs cursor for select * from table(:a_set);
```

有关对于 SELECT 语句使用集合游标的扩展的代码示例，请参阅 从集合游标访存。

## 使用集合派生表插入

要为集合变量声明 Insert 游标，应将集合派生的表这一段同与集合游标相关联的 INSERT 语句包括在一起。一个 Insert 游标允许您从集合变量选择一个或多个元素。（有关对 INSERT 语句和使用的描述，请参阅 INSERT 语句。）

该 Insert 游标必须是一个顺序游标。即，DECLARE 语句不能指定 SCROLL 关键字。

当您为集合变量声明 Insert 游标，INSERT 语句的集合派生表子句*必须*包含集合变量的名称。您不能为此集合变量指定一个输入参数（问号（?））。但是，可以在 INSERT 语句的 VALUES 子句中使用输入参数。该参数指示集合元素稍后将由 PUT 语句的 FROM 子句提供。

包括 INSERT 语句和集合派生表子句的集合游标允许您将多个元素插入一个集合变量。

要插入多个元素：

1. 在 GBase 8s ESQL/C 程序中创建一个客户端集合变量。
2. 使用 DECLARE 语句为 INSERT 语句声明集合游标。
3. 使用 OPEN 语句打开游标。
4. 使用 PUT 语句和 FROM 子句将元素放入集合游标。
5. 一旦集合变量包含所有元素，即可对表名称使用 UPDATE 语句或 INSERT 语句将集合变量的内容保存在集合列（SET 、MULTISET 或 LIST）。
6. 使用 CLOSE 语句关闭此集合游标。

此示例为 a_set 集合变量声明了一个 Insert 游标：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection multiset(smallint not null) a_mset;
        int an_element;
        EXEC SQL END DECLARE SECTION;
        ...
        EXEC SQL declare mset_curs cursor for
        insert into table(:a_mset) values (?);
        EXEC SQL open mset_curs;
        while (1)
        {
        ...
        EXEC SQL put mset_curs from :an_element;
        ...
        }
```

要将元素插入集合变量，可使用 FROM 子句的 PUT 语句。有关对 INSERT 语句使用集合游标的代码示例，请参阅插入到 Collection 游标内。

## 使用带事务的游标

要回滚修改，必须在事务内执行修改。只有当执行 BEGIN WORK 语句时，不兼容 ANSI 的数据库中的事务才会开始。

在兼容 ANSI 的数据库中，事务始终有效。

数据库服务器对选择和更新游标强制这些准则以确保可以正常地提交或回滚修改：

- 在事务内打开一个插入或更新游标。
- 在一个事务内包含 PUT 和 FLUSH 语句。

- 在一个事务内修改数据（更新、插入或删除）。

数据库服务器让您打开和关闭保持游标以便在事务之外执行更新操作；但是，应当先取得所有与给定修改有关的行，然后在一个单独的事务中执行所有的修改。您无法在事务之外打开和关闭保持游标或更新游标。

以下示例在事务内使用一个更新游标：

```
EXEC SQL declare q_curs cursor for
        select customer_num, fname, lname from customer
        where lname matches :last_name for update;
        EXEC SQL open q_curs;
        EXEC SQL begin work;
        EXEC SQL fetch q_curs into :cust_rec; /* fetch after begin */
        EXEC SQL update customer set lname = 'Smith'
        where current of q_curs;
        /* no error */
        EXEC SQL commit work;
```

当更新事务内的行时，该行保持为锁定状态，直到游标被关闭或者是事务被提交或回滚。如果在没有任何事务是有效的时候对行进行更新，则在将修改的行写到磁盘时会释放该行锁定。如果在事务之外更新或删除行，则无法回滚该操作。

在使用事务的数据库中，您无法在事务之外打开一个插入游标，除非也使用 WITH HOLD 关键字声明了该插入游标。

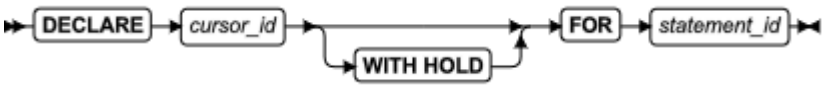
## 在 SPL 例程中声明动态游标

在 SPL 例程中使用 DECLARE 语句声明动态游标的名称，并将此游标与 PREPARE 语句在同一 SPL 例程中声明的准备好的语句的语句标识符相关联。

SQL 的 DECLARE 语句在 SPL 例程中创建的动态游标与 SPL 的 FOREACH 语句在 SPL 例程中创建的直接顺序游标不同。（有关顺序游标的语法和用法，请参阅 FOREACH。）

### 语法

SPL 例程中 DECLARE 语句是 GBase 8s ESQL/C 例程中 DECLARE 支持的语法的子集。




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 此处为动态游标声明的名称 r | 在例程的游标名称、准备好的语句名称和 SPL 变量名称中必须是唯一 | 标识符 |
| *statement_id* | 一个准备好的 SQL 语句的标识符 | 必须已在同一 SPL 例程的 PREPARE 语句中声明 | 标识符 |

用法

在以 SPL 语言编写的 UDR 中，与游标相关联的 *statement_id* 必须已在 PREPARE 语句的同一 UDR 中从这些语句类型之一的单个 SQL 语句的文本中预先声明：

- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- SELECT.

***statement_id*** 指定的准备好的语句文本可以包含问号（ ？ ）作为用户在运行时提供的值的占位符，但 PREPARE 语句中的占位符只能表示数据值，而不能表示 SQL 标识符。

DECLARE 语句可以在 SPL 例程中定义的动态游标类似于 ESQL/C 在其功能中 Select 游标或函数游标，但是具有以下限制：

- DECLARE 在 SPL 例程中定义的游标可以是 Select 游标或函数游标，但是它们不能是 Insert 游标或集合游标。
- 游标或准备好的语句的标识符不能指定为 SPL 变量。因为在 SPL 中，变量、游标和准备好对象的标识符会共享同一命名空间。
- 缺省情况下，SPL 的动态游标是顺序的。它们不能是滚动游标。
- 您使用 WITH HOLD 关键字创建的动态游标的语义与 FOREACH 语句声明的保持游标相同。
- SPL 例程中的 WITH HOLD 关键字只对 Select 游标有效。如果 ***statement_id*** 引用了 EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句的准备好的文本，则 DECLARE 语句失败并发出错误 -26056。
- 在 DECLARE 语句中 ESQL/C 支持的 FOR UPDATE 和 FOR READ ONLY 关键字在 SPL 例程中不支持。使用 SPL 的 FOREACH 语句声明可以模拟 ESQL/C 更新游标的功能的直接游标。（但是当编译 UDR 时，而不是在运行时定义与直接游标相关联的查询。）
- SPL 例程中的 DECLARE 语句在集合派生的表上不支持 SELECT 操作。
- SPL 例程的 DECLARE 语句中的语法错误会在运行时报告，不像 ESQL/C 的语法错误，在例程编译完后报告。

DECLARE 的语句与 SPL 例程中的准备好的语句相关联的动态游标的名称可以由同一 SPL 例程中的动态 SQL 的 OPEN 、CLOSE 、FETCH 和 FREE 语句引用。

在以下程序片段中，声明一个名为 equi_noctis 的游标，并打开、关闭和释放它。

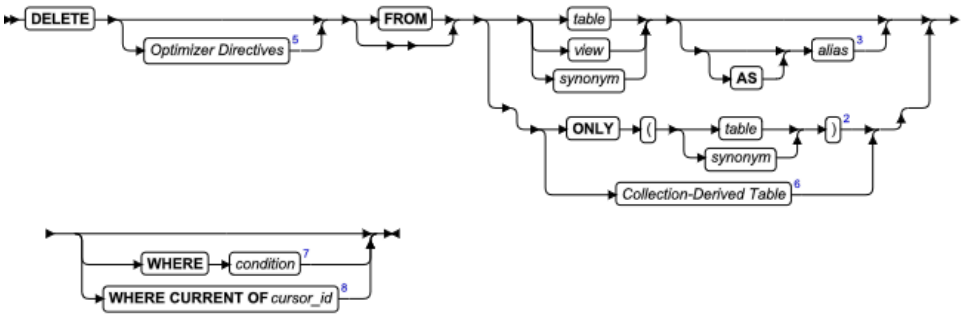
```
CREATE FUNCTION lente
        DEFINE first, last VARCHAR(30);
        . . .
        DATABASE stores_demo;
        LET first = "select * from state";
        LET lsst = "where code < ?";
        PREPARE stmt_1 FROM first || last;
        DECLARE cursor_1 FOR stmt_1;
        OPEN cursor_1
        . . .
```

```
CLOSE cursor_1;
FREE cursor_1;
FREE stmt_1;
...
END FUNCTION;
```

## 2.59 DELETE 语句

使用 DELETE 语句从表中删除一行或多行，或者在 SPL 或 GBase 8s ESQL/C 的集合变量中删除一个或多个元素。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 在此处为表、视图或同义词声明的临时名称 | 如果 WHERE 是 *alias* 的标识符则 AS 关键字必须优先于 *alias* | 标识符 |
| *condition* | 删除行必须满足的逻辑条件 | 不能是 UDR 也不能是相关的子查询 | 条件 |
| *cursor_id* | 先前声明的游标 | 必须已声明为 FOR UPDATE | 标识符 |
| *synonym*, *table*, *view* | 带有要删除行的表、同义词或可更新的视图 | *table* 或 *view*（或 *synonym* 及其指向的表或视图）必须存在 | 数据库对象名 |

**用法**

使用 DELETE 语句移除以下任意类型的数据库对象或程序对象：

- 表中或视图中的行：一行，一组行或所有的行
- 集合数据类型的列中的元素
- 已命名或未命名 ROW 数据类型的列、一个字段或所有字段。

您还可以使用此语句移除 GBase 8s ESQL/C 或 SPL 集合变量或 ROW 变量中的一个或多个元素的值。

要执行 DELETE 语句，您必须拥有数据库上的 DBA 访问权限，或者表的 Delete 存取权限。

在带有显式事务日志记录的数据库中，任何在事务之外执行的 DELETE 语句都将作为一个单独的事务来对待。

如果指定*视图*名称，则该视图必须是可更新的。有关对可更新视图的解释，请参阅通过视图更新。

DELETE 语句不能引用 CREATE EXTERNAL TABLE 语句定义的表对象。

如果您使用不带 WHERE 子句的 DELETE （指定一个条件或游标的活动集合），表中所有的行都会被删除。但是，若要移除表中所有的行，使用 TRUNCATE 语句会比 DELETE 语句更有效率。

在 DB-Access 中，如果您在 SQL 菜单工作时，省略 WHERE 子句，则 DB-Access 会提示您确认是否要删除表中所有的行。如果在命令文件中执行 DELETE 语句，则不会收到提示。

在兼容 ANSI 的数据库中，数据操作语言（DML）语句通常都在事务中。您无法在事务外执行 DELETE 语句。

**FROM 子句**

FROM 关键字优先于目标表的名称是可选的。要删除名为 from 的表中的行，可以设置 **DELIMIDENT** 环境变量并使用引号（" ）分隔 "from"：

```
DELETE "from";
```

另一种方法是，可以使用表所有者的名称限定 from 表的名称：

```
DELETE zelaine.from;
```

但是，如果您避免将 SQL 关键字声明为表、视图或其它数据库对象的标识符，则 SQL 代码会比较容易读取和维护。

**WHERE 子句**

如果使用不带 WHERE 子句（指定一个条件或游标的活动集合）的 DELETE ，则会删除表中所有的行。但是，若要移除表中所有的行，使用 TRUNCATE 语句会比 DELETE 语句更有效率。

在 DB-Access 中，如果您在 SQL 菜单工作时，省略 WHERE 子句，则 DB-Access 会提示您确认是否要删除表中所有的行。如果在命令文件中执行 DELETE 语句，则不会收到提示。

**锁定注意事项**

数据库服务器在事务期间锁定该事务内每个受 DELETE 语句影响的行。该表锁定粒度的可以是 PAGE 级或 ROW 级。

决定锁定粒度的功能具有以下顺序的优先级：

- DEF_TABLE_LOCKMODE 配置参数可以将表的锁定粒度缺省设置为 PAGE 或 ROW。
- 如果 IFX_TABLE_LOCKMODE 环境变量设置成 PAGE 或 ROW，则它的设置会覆盖 DEF_TABLE_LOCKMODE 的缺省值。
- CREATE TABLE 语句的 LOCK MODE 子句会覆盖新表的任何任何缺省锁定粒度。
- ALTER TABLE 语句的 LOCK MODE 子句可以将表的锁定粒度重置为 PAGE 或 ROW，覆盖任何上述的设置。

- LOCK TABLE 语句总是锁定整个表，覆盖指定表的以上列出的锁定粒度规范。

当表的锁定粒度是 ROW 时，数据库服务器对每个受 DELETE 语句影响的页获取一个锁。

如果影响的行的数量很大，且锁定方式为 ROW ，则可能超过操作系统对同时发生的锁的最大数目所置的限制。如果发生这种情况，您可以在执行 DELETE 语句之前减小 DELETE 语句的范围或者以互斥方式用 LOCK TABLE 语句锁定表。

## 对类型表使用 ONLY 关键字

当 DELETE 语句指定一个超级表时，任何满足 WHERE 子句的限定的行都会被删除，缺省情况下，表层次结构中所有超表的子表均会删除。要限制 DELETE 超级表的作用域，您必须在超级表的名称或同义词之前指定 ONLY 关键字。

在以下示例中，任何 name 列的具有 johnson 值的行都会从超级表 super_tab 中删除。但是，super_tab 的子表中的 name 列的具有 johnson 值的行会被保留，因为 ONLY( ) 子句限制了超级表的 DELETE 操作：

```
DELETE FROM ONLY(super_tab)
        WHERE name = "johnson";
```

**警告：** 如果您在超级表上使用 DELETE 语句并省略了 ONLY 关键字和 WHERE 子句，则会删除超级表及其子表的所有行。

如果您计划使用 WHERE CURRENT OF 子句删除游标的活动集合的当前行，则不能指定 ONLY 关键字。

## 级联删除表时的注意事项

当使用 CREATE TABLE 或 ALTER TABLE 语句的 REFERENCES 子句的 ON DELETE CASCADE 选项时，即指定了您想从一个表级联地删除到另一个表。例如，在 **stores_demo** 数据库中，**stock** 表包含作为主键的列 **stock_num**。**catalog** 和 **items** 表中每一个表都包含作为外键的用 ON DELETE CASCADE 选项指定的列 **stock_num** 。当从 **stock** 表执行删除操作是，也会在 **catalog** 和 **items** 表（这两个表通过外键引用）中删除行。

要使 DELETE 操作级联到一个含有对父表的引用约束的表，您只需要拥有对 DELETE 语句中的引用的父表的 Delect 特权。

如果使用级联删除一个或多个子表引用的表执行不带 WHERE 子句的 DELETE 操作，则 GBase 8s 从该表及其任何受影响的子表删除所有的行。（这类似于 TRUNCATE 语句的作用，但是在具有子表引用它的表上 GBase 8s 不支持 TRUNCATE 操作。）

有关如何创建使用级联删除的引用约束的示例，请参阅使用 ON DELETE CASCADE 选项。

## 表有级联删除时对 DELETE 的限制

你不可以使用相关子查询查询中的子表从父表删除行。如果两个子表引用相同的父表，并且一个子表指定级联删除但另一个子表没有指定，那么如果您尝试从父表中删除同时应用于这两个子表的行，则删除失败，并且不会从父表或子表删除任何行。

## 级联删除的锁定和记录日志牵连

在删除期间，数据库服务器会在被引用的表以及正在引用的表的所有符合条件的行上放置锁。

GBase 8s 要求对级联删除进行事务日志记录。如果在不符合 ANSI 的数据库中改变日志记录，即使是临时关闭，那么删除操作也不会级联地执行，因为您无法回滚任何操作。例如，如果删除一个父行，但在删除子行之前系统发生故障，那么数据库将含有悬挂的子记录，这违反了参照完整性。但是，在重新打开日志记录之后，随后的删除是级联的。

### 使用 WHERE 关键字指定条件

使用 WHERE *condition* 子句指定您要从表中删除的行。WHERE 关键字之后的 *condition* 等同于 SELECT 或 UPDATE 语句中的 *condition* 。例如，下一个语句删除了顺序号小于 1034 的 items 表的所有行：

```
DELETE FROM items WHERE order_num < 1034;
```

在 DB-Access 中，如果包含了选择表中所有行的 WHERE 子句，则 DB-Access 不会给出任何提示，并且删除所有行。

如果正在从表层次结构中的超级表删除，则 WHERE 子句的子查询无法引用子表。

当正从子表删除的时候，WHERE 子句中的子查询只可以在 SELECT … FROM ONLY (***supertable***)... 语法中引用超级表。

### DELETE 的 WHERE 子句中的子查询

DELETE 语句的 WHERE 子句中的子查询 FROM 子句可以将 DELETE 语句的 FORM 子句指定的同一个表或视图指定为数据源。仅当所有以下条件为真时，才支持带有引用相同表对象的子查询的 DELETE 操作：

- 该子查询要么返回一行，要么具有不相关列引用。
- 该子查询在 DELETE 语句的 WHERE 子句中，使用 Condition with Subquery 语法。
- 任何子查询中的 SPL 例程不能引用正在修改的表。

除非以上这些条件都满足，否则包含引用同 DELETE 语句修改的相同的表或视图子查询的 DELETE 语句返回错误 -360。

以下示例从 orders 表中删除其中 paid_date 列值满足 WHERE 子句中条件的行的子集。WHERE 子句通过将 IN 运算符应用于子查询返回的行来指定要删除的行，该子查询只选择 orders 表中的行，其中 paid_date 值早于当前日期：

```
DELETE FROM orders WHERE paid_date IN
```

```
    (SELECT paid_date FROM orders WHERE paid_date < CURRENT );
```

该子查询仅包含不相关的列引用，因为其唯一引用的列位于 FORM 子句中指定的表中。上面列出的要求有效，因为子查询的数据源与外部 UPDATE 语句的 FROM 子句指定的顺序表相同。上一个示例说明了 GBase 8s 支持 DELETE 语句的 WHERE 子句中不相关子查询。而不是如何写短 SQL 语句。下一示例使用更简单的语法实现了相同的结果：

```
DELETE orders WHERE paid_date < CURRENT;
```

以下示例从 **stock** 表中删除具有最大 **unit_price** 值的行（或多行）。WHERE 子句通过将等于运算符应用于子查询的结果来确定哪个 **unit_price** 值最大，子查询调用 **unit_price** 列值的内置 **MAX** 聚合函数：

```
DELETE FROM stock WHERE unit_price =
    (SELECT MAX(unit_price) FROM stock );
```

如果作为修改相同表的 DELETE 语句的 WHERE 子句中的子查询的数据源的表上定义了已启用的 Select 触发器，则在 DELETE 语句中执行该子查询不会激活触发器。

DELETE 语句中的子查询可以包含 UNION 或 UNION ALL 运算符。

如果外部 DELETE 语句修改表层次结构中的类型表， GBase 8s 支持在 DELETE 的 WHERE 子句中使用有效子查询的所有以下操作：

- 从带有（ SELECT from parent table ）子查询的父表中 DELETE
- 从带有（ SELECT from child table ）子查询的父表中 DELETE
- 从带有（ SELECT from parent table ）子查询的子表中 DELETE
- 从带有（ SELECT from child table）子查询的子表中 DELETE 。

请参阅 子查询的条件主题以获取有关 DELETE 语句的 WHERE 子句中将多行返回为谓词的子查询的语法的更多信息。

## 为表声明别名

可以为表声明别名。别名可以引用本地或远程表、视图或同义词的完全限定数据库对象名称。

别名是未在数据库的系统目录中注册临时的名称，而且只有在 DELETE 语句运行时才会保留。

如果您声明为别名的名称是关键字 WHERE，则必须使用 AS 关键字来说明语法：

```
DELETE stock AS where
        WHERE manu_code =
    (SELECT manu_code FROM where WHERE manu_code MATCHES 'H*');
```

因为 **where** 是 DELETE 和 SELECT 的关键字，所以之前的示例不易读取。以下示例访问远程表而不声明表的别名：

```
DELETE overstock@cleveland:stock AS ocs
        WHERE manu_code =
        (SELECT manu_code FROM overstock@cleveland:stock
        WHERE manu_code MATCHES 'H*');
```

;

下一个示例在逻辑上等同于前一个 DELETE 语句，但通过将 ocs 声明为引用子查询中的相同表的别名来简化符号：

```
DELETE overstock@cleveland:stock AS ocs
        WHERE manu_code =
    (SELECT manu_code FROM ocs WHERE manu_code MATCHES 'H*');
```

在 Oracle 模式下，保持 GBase 8s 原有别名语法基础上，如果 DELETE 语句的 SQL 关键字包括 NAME、TEMP、ARRAY、LIST、REVERSE、CONTEXT、LENGTH、LOG 作为表或视图别名使用，可以不用关键字 AS 开始它的声明。

例如，在 DELETE 语句中使用关键字 TEMP 作为 customer 表别名：

```
DELETE FROM customer temp WHERE temp.col =1;
```

## 使用 WHERE CURRENT OF 关键字（ESQL/C, SPL）

WHERE CURRENT OF 子句删除一个游标的活动集合的当前行。当包括这个子句时，DELETE 语句会在游标的当前位置处除去活动集合的行。在删除之后，没有任何当前的行存在；您无法使用游标删除或更新行，直到您使用 FETCH 语句（ESQL/C 例程中）或 FOREACH 语句（ SPL 例程中）重新确定游标的位置。

你可以使用更新游标来访问游标的活动集合的当前行。在可以使用 WHERE CURRENT OF 子句之前，您必须通过使用 FOREACH 语句（SPL 中）或使用带 FOR UPDATE 子句（ GBase 8s ESQL/C 中）的 DECLARE 语句。跟在 OF 关键字之后的 ***cursor_id*** 不能通过 SPL 例程中的 DECLARE 声明。

除非使用 FOR READ ONLY 关键字声明所有 Select 游标，否则所有 Select 游标在兼容 ANSI 的数据库中均是潜在的更新游标。您可以将 WHERE CURRENT OF 子句与没有用 FOR READ ONLY 关键字声明的任何 Select 游标一起使用。

如果您只从表层次结构中的一个表进行选择，则不可以使用 WHERE CURRENT OF 。即这个子句对 ONLY 关键字无效。

通过删除集合变量控制的集合派生的表的当前行，WHERE CURRENT OF 子句可用于从集合删除元素。有关更多信息，请参阅集合派生表。

## 删除包含不透明数据类型的行

当删除一些不透明数据类型的时，它们需要特别的处理过程。例如，如果不透明数据类型包含占数据库空间的或多重表示的数据，则它可能提供如何存储该数据的选项：在内部结构中或者在智能大对象中（对于大对象）。

要完成此进程，应调用一个称为 **destroy( )** 的用户定义的支持函数。当使用 DELETE 除去包含这些不透明数据类型的其中一种的行的时，数据库服务器会自动对该不透明数据类型调用 **destroy( )** 。此函数决定如何除去数据，不管它存储在什么地方。有关对 **destroy( )** 支持函数的更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## 删除包含集合数据类型的行

当行包含集合数据类型的列（LIST 、MULTISET 或 SET），您可以在集合中搜寻特定的元素，并且删除在其中发现该元素的一行或多行。

例如，以下的示例从 new_tab 表中删除其中的 set_col 列包含元素 jimmy smith 的任何行：

```
DELETE FROM new_tab WHERE 'jimmy smith' IN set_col;
```

您也可以通过删除集合中一个或多个个别元素使用集合变量删除集合列中的值。有关更多信息，请参阅 集合派生表 和 数据库名 和 从集合中执行删除操作的示例 中的示例。

## 分布式 DELETE 操作中的数据类型

访问另一个 GBase 8s 实例的数据库的 DELETE 语句（或其它 SQL 数据操纵语言语句）只能引用以下数据类型：

- 不透明的内置数据类型
- BOOLEAN
- LVARCHAR
- 不透明的内置数据类型的 DISTINCT
- BOOLEAN DISTINCT
- LVARCHAR DISTINCT
- 出现在此列表中的任何 DISTINCT 数据类型的 DISTINCT

如果 DISTINCT 类型显式地强制转型为内置类型，并且所有的 DISTINCT 类型、它们的数据类型层次结构以及它们的强制转型在按同一方式定义在每个参与操作的数据库中，则跨服务器分布式 DELETE 操作可以支持这些 DISTINCT 类型。

跨服务器的 DML 操作不能引用复杂、大对象或者用户定义的不透明数据类型（UDT），或者不支持的 DISTINCT 类型或内置不透明类型的列或表达式。有关 GBase 8s 在跨服务器 DML 操作中支持的数据类型的其它信息，请参阅跨服务器事务中的数据类型。

但是，访问本地 GBase 8s 实例的其它数据库的分布式操作可以访问上述为跨服务器操作列出的数据类型，也能访问下列其它的数据类型大多数内置的不透明的：

- 大多数***内置的不透明数据类型***，如跨数据库事务中的数据类型 中所列
- 上面一行中引用的内置类型的 DISTINCT
- 上面两行中列出的任何数据类型的 DISTINCT
- 可以显式强制转型为内置数据类型的不透明的用户定义的数据类型（UDT）

如果所有的不透明和 DISTINCT UDT 都显式地强制转型成内置类型，同时所有的 UDT 、DISTINCT 类型，以及强制转型都定义在每个参与操作的数据库中，那么跨数据库的 DELETE 操作支持这些 DISTINCT 和不透明的 UDT 。

分布式 DELETE 不能访问另一个 GBase 8s 实例的数据库，除非这两个服务器在 DBSERVERNAME 或 DBSERVERALIASES 配置参数中定义了 TCP/IP 或 IPCSTR 连接。这一连接类型要求适用于 GBase 8s 实例间的所有通信，即使这两个数据库服务器都驻留在同一台计算机上。

## 兼容 ANSI 的数据库中的 SQLSTATE 值

如果没有任何行满足对 兼容 ANSI 的数据库中的表执行的 DELETE 操作的 WHERE 子句，那么数据库服务器会发出一个警告。可以下列方式之一检测此警告条件：

- GET DIAGNOSTICS 语句将 **RETURNED_SQLSTATE** 字段设置为值 02000。在 SQL API 应用程序中，**SQLSTATE** 变量包含这个相同的值。
- SQL API 应用程序中，**sqlca.sqlcode** 和 **SQLCODE** 变量包含值 100。

如果 DELETE . . . WHERE 语句是多语句预备对象的一部分，并且数据库服务器不返回任何行，则数据库服务器还将 **SQLSTATE** 和 **SQLCODE** 设置为这些值。

## 在不兼容 ANSI 的数据库中的 SQLSTATE 值

在不兼容 ANSI 的数据库中，当数据库服务器没有找到满足 DELETE 语句的 WHERE 子句的行时，不会返回警告。在这种情况下，**SQLSTATE** 代码为 00000 且 **SQLCODE** 代码为零（0）。但是如果 DELETE . . . WHERE 是多语句的预备对象的一部分，且没有返回任何行，则数据库服务器会发出一个警告。它将 **SQLSTATE** 设置为 02000 ，将 **SQLCODE** 值设置为 100。

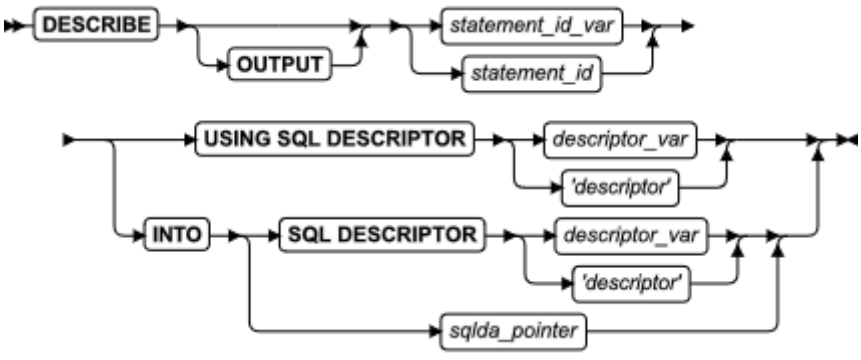
有关返回 **SQLSTATE** 变量的值的 ANSI/ISO 兼容状态的信息，请参阅 SQLSTATE 支持 SQL 的 ANSI/ISO 标准一节。

# 2.60 DESCRIBE 语句

使用 DESCRIBE 语句可在执行准备好的语句之前获得有关其输出参数和其它功能的信息。

在 GBase 8s ESQL/C 中使用此语句。（另见 DESCRIBE INPUT 语句。）

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 系统描述符区域的名称 | 必须已分配系统描述符区域 | 引用字符串 |
| *descriptor_var* | 指定系统描述符区域的主变量 | 必须包含分配的系统描述符区域的名称 | 特定于语言的名称规则 |
| *sqlda_pointer* | 指向 sqlda 结 | 不可以美元符号（$） | 请参阅 *GBase* |

| | | | |
|---|---|---|---|
| | 构 | 或冒号（：）开始。如果使用动态的 SQL，则 sqlda 结构是必需的 | *8s ESQL/C 程序员手册* 中的 sqlda 结构 |
| *statement_id* | 准备好的 SQL 语句的语句标识符 | 必须在先前的 PREPARE 语句中定义 | PREPARE 语句；标识符 |
| *statement_id_var* | 包含 *statement_id* 的值的主变量 | 必须在先前的 PREPARE 语句中定义 | 特定于语言的名称规则 |

**用法**

DESCRIBE 可以在运行时提供有关准备好的语句的信息：

- 准备好的 SQL 语句的类型
- 是 UPDATE 还是 DELETE 语句包含 WHERE 子句
- 对于 EXECUTE 、EXECUTE FUNCTION 、EXECUTE PROCEDURE 、INSERT 、SELECT 或 UPDATE 语句，DESCRIBE 语句也返回值的数目、数据类型和大小，以及查询返回的列或表达式的名称。
- 对于 SELECT 语句，DESCRIBE 还返回查询返回的列或表达式的名称。

使用此信息，您可以编写代码来分配内存，从而控制检索到的值，并且在取得这些值之后显示或处理它们。

## OUTPUT 关键字

OUTPUT 关键字指定只将有关准备好的语句的输出参数的信息存储在 **sqlda** 描述符区域中。如果省略这个关键字，DESCRIBE 可以返回输入参数，但这仅针对 INSERT 语句（如果在数据库服务器已初始化的环境中设置了 **IFX_UPDDESC** 环境变量，这种情况也适用于 UPDATE ）。

## 描述语句类型

DESCRIBE 语句从 PREPARE 语句获取一个语句标识符作为输入。当 DESCRIBE 语句执行时，数据库服务器会设置 **sqlca** 的 SQLCODE 字段的值，以指明语句类型（即语句开始处的关键字）。如果准备好的语句文本包含多个 SQL 语句，则 DESCRIBE 语句返回文本中第一个语句的类型。

SQLCODE 设置为 0 表示一个不带 INTO TEMP 子句的 SELECT 语句。这种情况是最普通的。对于任何其它 SQL 语句，SQLCODE 设置为一个正整数。您可以对照定义的常量名测试该数字。在 GBase 8s ESQL/C 中，常量名定义在 **sqlstypes.h** 头文件中。

DESCRIBE 语句（和 DESCRIBE INPUT 语句）对 SQLCODE 字段的使用不同于任何其它语句，当它成功执行时可能返回一个非零值。如果愿意，您可以修订标准的错误检查例程以使用这种行为。

## 检查 WHERE 子句的存在性

如果 DESCRIBE 语句检测到一个不带 WHERE 子句的 UPDATE 或 DELETE 语句的准备好的语句，则 DESCRIBE 语句将 **sqlca.sqlwarn.sqlwarn4** 变量设置为 W。

当 DELETE 或 UPDATE 语句中没有指定 WHERE 子句时，数据库服务器对整个表执行删除或更新操作。检查 **sqlca.sqlwarn.sqlwarn4** 变量以避免不期望的对表所做的全局更改。

## 描述带运行时参数的语句

如果准备好的语句包含这么一种参数。即在运行时将为该参数提供参数或参数数据类型的数目，那么您可以描述这些输入值。如果准备好的语句文本包括以下一个语句，那么 DESCRIBE 语句会返回一个对表中包括的每个列或表达式的描述：

- EXECUTE FUNCTION （或 EXECUTE PROCEDURE）
- INSERT
- SELECT （不带 INTO TEMP 子句）
- UPDATE

在 GBase 8s 中，必须首先按照 *《GBase 8s SQL 指南：参考》* 中的描述设置 **IFX_UPDDESC** 环境变量，然后才可以使用 DESCRIBE 来获取有关 UPDATE 语句的信息。

描述包括以下信息：

- 列的数据类型，如表中定义
- 列的长度，以字节为单位
- 列或表达式的名称

对于准备好的 INSERT 或 UPDATE 语句，DESCRIBE 只返回动态参数（那些由问号（?）表示的参数）。但是，使用 OUTPUT 关键字会防止返回这些参数。

您可以将为返回的信息所设置的目的地指定为一个新的或现有的系统描述符区域，或是指定为一个指向 **sqlda** 结构的指针。

系统描述符区域符合 X/Open 标准。

## 使用 SQL DESCRIPTOR 关键字

使用 USING SQL DESCRIPTOR 子句可将对准备好的语句列表的描述存储在先前分配的系统描述符区域中。

使用 INTO SQL DESCRIPTOR 子句创建新的系统描述符结构并将语句列表的描述存储在该结构中。

要将一个先前提到的语言描述到一个系统描述符区域，DESCRIBE 会以以下方式更新系统描述符区域：

- 将系统描述符区域中的 COUNT 字段设置为语句列表中值的数目。如果 COUNT 大于系统描述符区域中项描述符的数目，则会导致一个错误。
- 它设置系统描述符区域中的 TYPE 、LENGTH 、NAME 、SCALE 、PRECISION 和 NULLABLE 字段。

- 如果列具有不透明数据类型，则数据库服务器会设置项描述符的 EXTYPEID 、EXTYPENAME 、EXTYPELENGTH 、EXTYPEOWNERLENGTH 和 EXTYPEOWNERNAME 字段。
- 根据 TYPE 和 LENGTH 信息，为每个项描述符的 DATA 字段分配内存。

在执行 DESCRIBE 语句之后，SCALE 和 PRECISION 字段分别包含列的小数位和精度。如果在 SET DESCRIPTOR 语句中设置了 SCALE 和 PRECISION ，并且将 TYPE 设置为 DECIMAL 或 MONEY，则会修改 LENGTH 字段以调整十进制值的小数位和精度。如果没有将 TYPE 设置为 ECIMAL 或 MONEY，则不设置 SCALE 和 PRECISION 的值，并且 LENGTH 不受影响。

您必须使用 SET DESCRIPTOR 语句修改系统描述符区域信息，以显示要接收描述符的值在内存中的地址。可以将数据类型更改为另一种兼容的类型。此更改会在取得数据值时引起数据转换的发生。

您可以在支持 USING SQL DESCRIPTOR 子句（如 EXECUTE 、FETCH 、OPEN 和 PUT）的准备好的语句中使用系统描述符区域。

下面的示例显示了 DESCRIBE 语句中对系统描述符的使用。在第一个示例中，系统描述符是一个用引号引起的字符串；在第二个示例中，它是一个嵌入的变量名称。

```
main()
  {
        . . .
       EXEC SQL allocate descriptor 'desc1' with max 3;
       EXEC SQL prepare curs1 FROM 'select * from tab';
       EXEC SQL describe curs1 using sql descriptor 'desc1';
   }
       EXEC SQL describe curs1 using sql descriptor :desc1var;
```

## 使用 INTO sqlda Pointer 子句

使用 INTO ***sqlda_pointer*** 子句可为 **sqlda** 结构分配内存，并将它的地址存储在一个 **sqlda** 指针中。DESCRIBE 语句用描述信息填充分配的内存。不像 USING 子句，INTO 子句创建新的 **sqlda** 结构以存储来自 DESCRIBE 的输出。

DESCRIBE 语句将 **sqlda.sqld** 字段设置为语句列表中的值的数目。**sqlda** 结构也包含一个数据描述符的数组（**sqlvar** 结构），语句列表中的每个值各有一个相应的数据描述符。执行 DESCRIBE 语句之后，**sqlda.sqlvar** 结构便含有 **sqltype** 、**sqllen** 和 **sqlname** 字段集。

如果列具有不透明数据类型，则 DESCRIBE...INTO 会设置项描述符的 **sqlxid** 、**sqltypename** 、**sqltypelen** 、**sqlownerlen** 和 **sqlownername** 字段。

一旦程序中声明了 **sqlda** 指针，DESCRIBE 语句就会为该指针分配内存。但是，应用程序必须指定 **sqlda.sqlvar.sqldata** 字段的存储区域。

## 描述集合变量

当使用 USING SQL DESCRIPTOR 或 INTO 子句时，DESCRIBE 语句提供有关集合变量的信息。在打开 Select 或 Insert 游标*之后*，必须发出 DESCRIBE 语句，因为 OPEN...USING 语句指定了要使用的集合变量的名称。

下一个 GBase 8s ESQL/C 代码段动态地选择 :a_set 集合变量的元素进入称为 desc1 的系统描述符区域：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection a_set;
        int i, set_count;
        int element_type, element_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL allocate descriptor 'desc1';
EXEC SQL select set_col into :a_set from table1;
EXEC SQL prepare set_id from 'select * from table(?)'

EXEC SQL declare set_curs cursor for set_id;
EXEC SQL open set_curs using :a_set;
EXEC SQL describe set_id using sql descriptor 'desc1';

do
{
        EXEC SQL fetch set_curs using sql descriptor 'desc1';
        ...
        EXEC SQL get descriptor 'desc1' :set_count = count;
        for (i = 1; i <= set_count; i++)
        {
            EXEC SQL get descriptor 'desc1' value :i
                :element_type = TYPE;
            switch
            {
                case SQLINTEGER:
                    EXEC SQL get descriptor 'desc1' value :i
                        :element_value = data;
                ...
            } /* end switch */
        } /* end for */
} while (SQLCODE == 0);

EXEC SQL close set_curs;
EXEC SQL free set_curs;
EXEC SQL free set_id;
EXEC SQL deallocate collection :a_set;
```

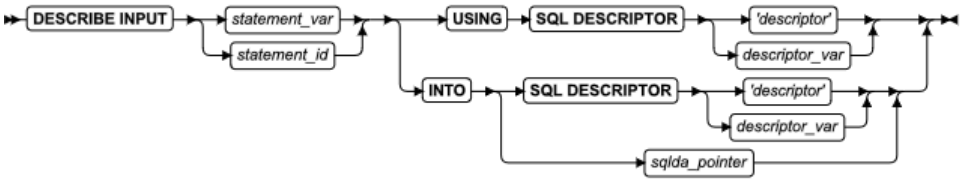
EXEC SQL deallocate desc

## 2.61 DESCRIBE INPUT 语句

使用 DESCRIBE INPUT 语句可在执行一个准备好的语句之前返回输入参数信息。

在 ESQL/C 中使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 系统描述符的名称 | 必须已分配系统描述符区域 | 引用字符串 |
| *descriptor_var* | 指定系统描述符区域的主变量 | 必须包含分配的系统描述符区域的名称 | 特定于语言的名称规则 |
| *sqlda_pointer* | 指向一个 sqlda 结构 | 不可以美元符号（$）或冒号（：）开始。如果使用动态的 SQL，则 sqlda 结构是必需的 | 请参阅 *GBase 8s ESQL/C 程序员手册* 中的 sqlda 结构 |
| *statement_id* | 准备好的 SQL 语句的语句标识符 | 必须在先前执行的 PREPARE 语句中定义 | PREPARE 语句；PREPARE 语句；标识符 |
| *statement_var* | 包含 *statement_id* 的值的主变量 | 变量和 *statement_id* 都必须声明 | 特定于语言的名称规则 |

**用法**

DESCRIBE INPUT 和 DESCRIBE OUTPUT 语句可以将有关准备好的语句的信息返回到 SQL 描述符区域（sqlda）：

- 对于 SELECT 、EXECUTE FUNCTION （或 PROCEDURE）、INSERT 或 UPDATE 语句，DESCRIBE 语句（不带 INPUT 关键字）返回返回值的数目、数据类型和大小以及列或表达式的名称。
- 对于 SELECT 、EXECUTE FUNCTION 、EXECUTE PROCEDURE 、DELETE 、INSERT 或 UPDATE 语句，DESCRIBE INPUT 语句返回准备好的语句的所有输入参数。

**提示：** 考虑到程序同旧应用程序的兼容性，当前版本支持不带 *INPUT* 的 *DESCRIBE* 语句。在新的应用程序中，您应当使用 DESCRIBE INPUT 语句来提供有关 WHERE 子句、子查询以及其它语法上下文动态参数的信息，这些都是在旧格式的 DESCRIBE 中无法提供的信息。

使用此信息，您可以编写代码来分配内存，从而控制那些在取得之后可以显示或处理的检索到的值。

**在使用 DESCRIBE INPUT 获取有关 UPDATE 语句的信息时，不需要设置** IFX_UPDDESC **环境变量。**

## 描述语句类型

该语句从 PREPARE 语句获取一个语句标识符作为输入。DESCRIBE INPUT 执行后，**sqlca** 的 SQLCODE 字段的值指明语句类型（即语句开始处的关键字）。如果准备好的对象包含多个 SQL 语句，则 DESCRIBE INPUT 语句返回文本中第一个语句的类型。

SQLCODE 设置为 0 表示一个不带 INTO TEMP 子句的 SELECT 语句。这种情况是最普通的。对于任何其它 SQL 语句，SQLCODE 设置为一个正整数。您可以对照定义的常量名测试该数字。在 GBase 8s ESQL/C 中，常量名定义在 **sqlstypes.h** 头文件中。

DESCRIBE 语句 和 DESCRIBE INPUT 语句对 SQLCODE 字段的使用不同于任何其它语句，在某些情况下可能返回一个非零值。如果愿意，您可以修订标准的错误检查例程以使用这种行为。

## 检查 WHERE 子句的存在性

如果 DESCRIBE INPUT 语句检测到一个准备好的对象包含不带 WHERE 子句的 UPDATE 或 DELETE 语句，则数据库服务器将 **sqlca.sqlwarn.sqlwarn4** 变量设置为 W。

当 DELETE 或 UPDATE 语句中没有指定 WHERE 子句时，数据库服务器对整个表执行删除或更新操作。DESCRIBE INPUT 执行后检查 **sqlca.sqlwarn.sqlwarn4** 变量以避免不期望的对表所做的全局更改。

## 使用动态运行时参数描述语句

如果准备好的语句指定了这么一种参数集合，即必须在运行时提供它的基数或数据类型，那么您可以描述这些输入值。 如果准备好的语句文本中包括以下一个语句，则 DESCRIBE INPUT 语句会返回对列表中包括的的每个列或表达式的描述：

- EXECUTE FUNCTION（或 EXECUTE PROCEDURE）
- INSERT 或 SELECT
- UPDATE 或 DELETE

该描述包括以下信息：

- 列的数据类型，如表中定义
- 列的长度，以字节为单位
- 列或表达式的名称
- 有关***动态参数*** （在准备好的语句中表示为问号（?）的参数）的信息

如果数据库服务器无法推断表达式参数的数据类型，DESCRIBE INPUT 语句会返回 SQLUNKNOWN 作为数据类型。

您可以将为返回的信息所设的目的地指定为一个新的或现有的系统描述符区域，或是指定为一个指向 **sqlda** 结构的指针。

## 使用 SQL DESCRIPTOR 关键字

使用 INTO SQL DESCRIPTOR 创建新的系统描述符结构并将准备好语句列表的描述在此结构中。

使用 USING SQL DESCRIPTOR 子句将准备好的语句列表的描述存储在先前分配的系统描述区域中。执行 DESCRIBE INPUT . . . USING SQL DESCRIPTOR 语句以以下方式修改现有的系统描述区域：

- 基于 TYPE 和 LENGTH 信息为每个项描述符的 DATA 字段分配内存。
- 将系统描述符区域中的 COUNT 字段设置为语句列表中值的数目。如果 COUNT 大于系统描述符区域中项描述符的数目，则会导致一个错误。
- 它设置系统描述符区域中的 TYPE 、LENGTH 、NAME 、SCALE 、PRECISION 和 NULLABLE 字段。

对于不透明数据类型的列。DESCRIBE INPUT 语句会设置项目描述符的 EXTYPEID 、EXTYPENAME 、EXTYPELENGTH 、EXTYPEOWNERLENGTH 和 EXTYPEOWNERNAME 字段。

DESCRIBE INPUT 语句执行后，SCALE 和 PRECISION 字段分别包含列的小数位和精度。如果在 SET DESCRIPTOR 语句中设置 SCALE 和 PRECISION，并且将 TYPE 设置为 DECIMAL 或 MONEY，则会修改 LENGTH 字段以调整十进制值的小数位和精度。如果没有将 TYPE 设置为 DECIMAL 或 MONEY，则不设置 SCALE 和 PRECISION 的值，并且 LENGTH 不受影响。

您必须使用 SET DESCRIPTOR 语句修改系统描述符区域的信息，以显示要接收描述的值在内存中的地址。可以将数据类型更改为另一种兼容的类型。此更改会在取得数据值时引起数据转换的发生。

您不能在其它支持 USING SQL DESCRIPTOR 子句（例如 EXECUTE 、FETCH 、OPEN 和 PUT ）的语句中使用系统描述符区域。

以下的示例显示了在 DESCRIBE 语句中的使用系统描述符。在第一个示例中，系统描述符是带引号的字符串；在第二个示例中，它嵌入了变量名称。

```
main()
        {
        . . .
        EXEC SQL allocate descriptor 'desc1' with max 3;
        EXEC SQL prepare curs1 FROM 'select * from tab';
        EXEC SQL describe curs1 using sql descriptor 'desc1';
        }
        EXEC SQL describe curs1 using sql descriptor :desc1var;
```

系统描述符区域必须符合 X/Open 标准。

## 使用 INTO sqlda Pointer 子句

使用 INTO ***sqlda_pointer*** 子句可为 **sqlda** 结构分配内存，并将它的地址存储在一个 **sqlda** 指针中。DESCRIBE INPUT 语句用描述信息填充分配的内存。

DESCRIBE INPUT 语句将 **sqlda.sqld** 字段设置为语句列表中的值的数目。**sqlda** 结构也包含一个数据描述符的数组（**sqlvar** 结构），语句列表中的每个值各有一个相应的数据描述符。执行 DESCRIBE 语句之后，**sqlda.sqlvar** 结构便含有 **sqltype** 、**sqllen** 和 **sqlname** 字段集。

如果列具有不透明数据类型，则 DESCRIBE INPUT...INTO 会设置项描述符的 **sqlxid**、**sqltypename** 、**sqlownerlen** 、**sqltypelen** 和 **sqlownername** 字段。

一旦程序中声明了 sqlda 指针，DESCRIBE INPUT 语句就会为该指针分配内存。但是，应用程序必须指定 sqlda.sqlvar.sqldata 字段的存储区域。

## 描述集合变量

如果您使用 INTO 和 USING SQL DESCRIPTOR 子句，DESCRIBE INPUT 语句提供有关集合变量的信息。

在打开 Select 或 Insert 游标后必须执行 DESCRIBE INPUT 语句。否则，DESCRIBE INPUT 无法获取有关集合变量的信息，因为它是指定要使用的集合变量的名称的 OPEN . . . USING 语句。

下一个 GBase 8s ESQL/C 程序段动态地选择 :a_set 集合变量的元素进入称为 desc1 的系统描述符区域：

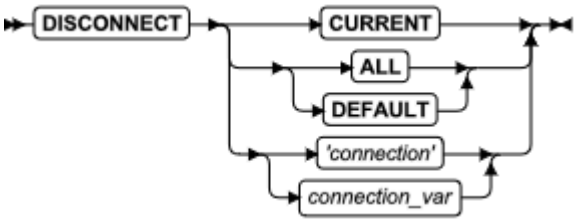
```
EXEC SQL BEGIN DECLARE SECTION;
        client collection a_set;
        int i, set_count;
        int element_type, element_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL allocate descriptor 'desc1';
EXEC SQL select set_col into :a_set from table1;
EXEC SQL prepare set_id from
        'select * from table(?)';


EXEC SQL declare set_curs cursor for set_id;
EXEC SQL open set_curs using :a_set;
EXEC SQL describe set_id using sql descriptor 'desc1';do
{
        EXEC SQL fetch set_curs using sql descriptor 'desc1';
        ...
        EXEC SQL get descriptor 'desc1' :set_count = count;
        for (i = 1; i <= set_count; i++)
        {
```

```
            EXEC SQL get descriptor 'desc1' value :i
                :element_type = TYPE;
            switch
            {
                case SQLINTEGER:
                    EXEC SQL get descriptor 'desc1' value :i
                        :element_value = data;
                ...
            } /* end switch */
        } /* end for */
} while (SQLCODE == 0);

EXEC SQL close set_curs;
EXEC SQL free set_curs;
EXEC SQL free set_id;
EXEC SQL deallocate collection :a_set;
EXEC SQL deallocate descriptor 'desc1';
```

## 2.62 DISCONNECT 语句

使用 DISCONNECT 语句终止应用程序和数据库服务器之间的连接。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *connection* | 指定要终止的连接的字符串 | CONNECT 语句指定的连接名称 | 引用字符串 |
| *connection_var* | 持有连接名称的主变量 | 必须是一个固定长度的字符数据类型 | 特定于语言 |

**用法**

DISCONNECT 终止到数据库服务器的连接。如果数据库是打开的，则在连接断开之前该数据库会关闭。即使只连接到一个特定的数据库，DISCONNECT 也会终止到数据库服务器的连接。如果 DISCONNECT 没有终止当前的连接，则不会更改当前环境的连接上下文。

DISCONNECT 在 PREPARE 语句中作为语句文本是无效的。

在 ESQL/C ，如果您使用 ***connection*** 或 ***connection_var*** 断开连接，那么在指定的连接不是当前连接或休眠连接的情况下，DISCONNECT 会生成一个错误。

## DEFAULT 选项

DISCONNECT DEFAULT 断开缺省连接。

***缺省连接***是指以下一种连接：

- 由 CONNECT TO DEFAULT 语句建立的连接
- 由 DATABASE 或 CREATE DATABASE 语句建立的隐式缺省连接

您可以使用 DISCONNECT 断开缺省连接。如果 DATABASE 语句没有指定数据库服务器（如下例所示），则缺省数据库服务器建立缺省的连接：

```
EXEC SQL database 'stores_demo';
        . . .
        EXEC SQL disconnect default;
```

如果 DATABASE 语句指定了数据库服务器（如下例所示），则向该数据库服务器建立缺省的连接：

```
EXEC SQL database 'stores_demo@mydbsrvr';
        . . .
EXEC SQL disconnect default;
```

在以上任何一种情况下，DISCONNECT 的 DEFAULT 选项会断开这个缺省的连接。有关更多信息，请参阅缺省连接规范。

## 指定 CURRENT 关键字

DISCONNECT CURRENT 语句终止当前的连接。例如，以下程序段中的 DISCONNECT 语句终止到数据库服务器 mydbsrvr 的当前连接：

```
CONNECT TO 'stores_demo@mydbsrvr';
        . . .
DISCONNECT CURRENT;
```

## 当事务活动时

DISCONNECT 在事务期间生成一个错误。事务仍处于活动状态，并且应用程序必须显式地提交或回滚该事务。如果在没有发出 DISCONNECT 的情况下应用程序终止（例如是由于系统故障或程序错误），则会回滚活动的事务。

但是，在兼容 ANSI 的数据库中，如果在非交互方式下没有发出 CLOSE DATABASE 、COMMIT WORK 或 DISCONNECT 语句便退出 DB-Access 时没有遇到错误，则数据库服务器自动提交任何打开的事务。

## 在线程安全环境中断开连接

如果您在线程安全的 GBase 8s ESQL/C 应用程序中发出 DISCONNECT 语句，请记住活动的连接只可以从它活动所在的线程内被断开。因此，一个线程无法断开另一个线程的活动连接。如果进行这种尝试，DISCONNECT 语句会生成一个错误。

但是一旦连接变为休眠状态，任何其它线程即可断开该连接，除非有一个正在进行的事务与使用 CONNECT 的 WITH CONCURRENT TRANSACTION 子句建立的休眠连接相关联。如果休眠的连接不是用 WITH CONCURRENT TRANSACTION 子句建立的，则 DISCONNECT 会试图在断开连接时生成一个错误。

有关对线程安全的 GBase 8s ESQL/C 应用程序中的连接的解释，请参阅 SET CONNECTION 语句。

## 指定 ALL 选项

使用关键字 ALL 终止至该时刻为止由应用程序建立的所有连接。例如，下面的 DISCONNECT 语句断开当前的连接以及所有休眠的连接：

```
DISCONNECT ALL;
```

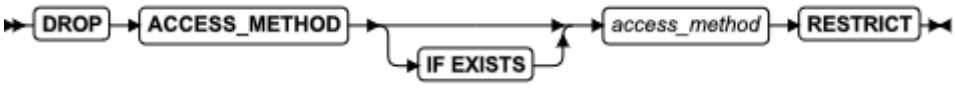
在 GBase 8s ESQL/C 中，ALL 关键字对多线程的应用程序的效果同对单线程应用程序的效果相同。执行 DISCONNECT ALL 语句会引起所有线程中的所有连接终止。但是，如果任何一个连接正在使用中，或者有一个与该连接相关联的正在处理的事务，则 DISCONNECT ALL 语句失败。如果这些条件中的任何一个为真，则不会断开任何连接。

# 2.63 DROP ACCESS_METHOD 语句

使用 DROP ACCESS_METHOD 语句可从数据库除去之前定义的主或辅助存取方法。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *access_method* | 要删除的存取方法的名称 | 必须在 sysams 系统目录表中注册；不能是内置存取方法 | 标识符 |
| *owner* | 存取方法的所有者 | 必须拥有存取方法 | 所有者名称 |

**用法**

RESTRICT 关键字是必需的。如果存在使用存取方法的虚拟表或索引，则无法删除该存取方法。您必须是该存取方法的所有者，或者拥有 DBA 特权，才可删除存取方法。

如果事务正在处理中，则数据库服务器等待移除此存取方法，直到提交或回滚该事务。该交易完成之前，其它任何用户都无法执行该存取方法。

如果您包含可选的 IF EXISTS 关键字，则如果指定名称的存取方法已在当前数据库中注册过，数据库服务器不执行任何操作（而不是向应用程序发生异常）。

**示例**

对于此示例，假设由此语句创建了一个存取方法：

```
CREATE SECONDARY ACCESS_METHOD T_tree
        (
        am_getnext = ttree_getnext,
        am_unique,
        am_cluster,
        am_sptype = 'S'
        );
```

以下语句删除了此存取方法：

```
DROP ACCESS_METHOD T_tree RESTRICT;
```

现有存取方法的详细信息可以使用以下查询在 sysams 系统目录表中找到：

```
SELECT am_name FROM gbasedbt.sysams;
```

## 2.64 DROP AGGREGATE 语句

使用 DROP AGGREGATE 语句删除您用 CREATE AGGREGATE 语句创建的用户定义的集合。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *aggregate* | 要删除的用户定义的聚集的名称 | 先前必须已用 CREATE AGGREGATE 语句创建 | 标识符 |
| *owner* | 聚集的所有者 | 必须拥有此聚集 | 所有者名称 |

**用法**

删除用户定义的聚集不会删除您在 CREATE AGGREGATE 语句中为聚集定义的支持函数。数据库服务器不会对您在语句中使用的用户定义的聚集追踪 SQL 语句的依赖性。例如，您可以删除 SPL 例程中使用的用户定义的聚集。

以下示例删除用户定义的聚集 my_avg：

```
DROP AGGREGATE my_avg;
```

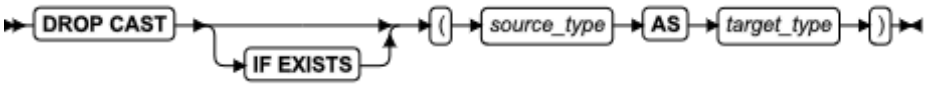
如果您包含了可选的 IF EXISTS 关键字，则如果指定名称的聚集没在当前数据库中注册过，数据库服务器不执行任何操作（而不是向应用程序发送异常）。

## 2.65 DROP CAST 语句

使用 DROP CAST 语句从数据库移除现有的强制转型。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *source_type* | 强制转型接收为输入的数据类型 | 必须存在 | 标识符；数据类型 |
| *target_type* | 由强制转型返回的数据类型 | 必须存在 | 标识符；数据类型 |

**用法**

您必须是强制转型的所有者或拥有 DBA 特权才可使用 DROP CAST。删除强制转型会从 syscasts 系统目录表中除去它的定义，所以不能显式地或隐式地调用强制转型。删除强制转型对于强制转型相关联的用户定义的函数没有影响。使用 DROP FUNCTION 语句可从数据库中除去用户定义的函数。

**警告：** 请勿删除用户 gbasedbt 拥有的内置强制转型。内置数据类型之间的字段转换需要这些强制转型。

在给定数据类型上定义的强制转型也可以用在从该源类型创建的任何 DISTINCT 类型上。如果删除强制转型，则无法再为 DISTINCT 类型调用它，但是删除为 DISTINCT 类型定义的强制转型对为其源类型创建的强制转型没有影响。当创建 DISTINCT 类型时，数据库服务器会自动地定义一个从 DISTINCT 类型到其源类型的显式强制转型以及另一个从源类型到 DISTINCT 类型的显式强制转型。当删除 DISTINCT 类型时，数据库服务器会自动删除这两个强制转型。

如果您包含可选的 IF EXISTS 关键字，则如果在两个指定的数据类型之间的强制转型在当前数据库中没有注册过，数据库服务器不执行任何操作（而不是向应用程序发送异常）。

**示例**

一个强制转型（如 superstores_demo 数据库中的这个强制转型）可以使用 DROP CAST 语句删除 ：

```
DROP CAST (decimal(5,5) AS percent);
```

现有的强制转型的详细信息可以使用以下 SQL 在 syscasts 系统目录表中找到：

```
SELECT routine_name, class, argument_type, result_type FROM Syscasts;
```

## 2.66 DROP DATABASE 语句

使用 DROP DATABASE 语句删除整个数据库，包括系统目录表、对象和数据。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


**用法**

DROP DATABASE 语句是 ANSI/ISO 标准的扩展，该标准没有提供毁坏数据库的语法。

以下语句删除 stores_demo 数据库：

```
DROP DATABASE stores_demo
```

您必须拥有 DBA 特权，或者您是用户 **gbasedbt** 才可成功运行 DROP DATABASE 语句。否则，数据库服务器会发出一条错误消息，并且不会删除数据库。

不能删除当前数据库或当前正由其它用户使用的数据库。数据库的所有当前用户必须先执行 CLOSE DATABASE 语句，这样 DROP DATABASE 才会成功。

DROP DATABASE 语句试图创建到您的想要删除的数据库的隐式连接。如果前面的 CONNECT 语句已经建立了到另一个数据库的显式连接，而且该连接仍然是您的当前连接，则 DROP DATABASE 语句失败，错误为 -1811。在这种情况下，您必须首先使用 DISCONNECT 语句关闭显式连接后，才能执行 DROP DATABASE 语句。

如果您包含可选的 IF EXISTS 关键字，则如果没有指定名称的数据库由所连接的数据库服务器实例管理，则数据库服务器不采取任何操作（而不是向应用程序返回错误）。

DROP DATABASE 语句不可以出现在多语句 PREPARE 中，也不可以出现在 SPL 例程中。

在 DROP DATABASE 操作中，数据库服务器对数据库中的每个表都获取一个锁，并保留这些锁直至整个操作完成。请给您的数据库服务器配置足够的锁以满足这一事实的发生。

例如，如果要删除的数据库有 2500 个表，但是为您的数据库服务器配置的锁定少于 2500 个，则 DROP DATABASE 语句失败。有关如何配置可用于数据库服务器的锁的数目的更多信息，请参阅 *GBase 8s 管理员参考手册* 中对 LOCKS 配置参数的讨论。

在 DB-Access 中，使用 DROP DATABASE 语句时要小心。DB-Access 不会提示您验证是否想要删除整个数据库。

在 ESQL/C 中，您可以在程序或主变量中使用未限定的数据库名称，或可以指定标准的 *database@server* 格式。例如，以下语句删除了名为 gibson95 的数据库服务器的 stores_demo 数据库：

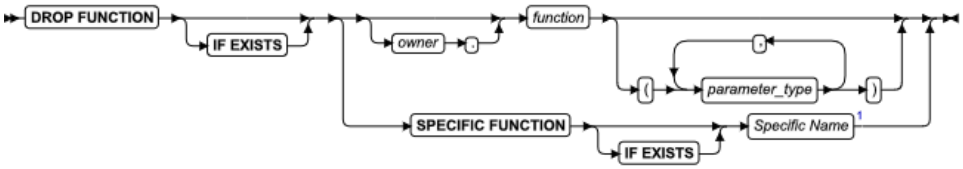
```
EXEC SQL DROP DATABASE stores_demo@gibson95;
```

如果此语句执行成功，则 gibson95 数据库服务器实例继续存在，但该数据库服务器的 stores_demo 数据库不再存在。有关更多信息，请参阅数据库名。

## 2.67 DROP FUNCTION 语句

使用 DROP FUNCTION 语句可从数据库中除去用户定义的函数。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 要删除的用户定义的函数的名称 | 必须存在于数据库中（即已注册）。如果该名称没有唯一地标识函数，你必须输入 *parameter_type* 的一个或多个适当的值 | 标识符 |
| *parameter_type* | 参数的数据类型 | 数据类型（或数据类型列表）必须与那些在创建函数时就在 CREATE FUNCTION 语句中指定的数据类型相同（并且以相同的顺序指定） | 数据类型 |

**用法**

删除用户定义的函数会从数据库中除去函数的文本和可执行的版本。（请确保在数据库外保留一个函数文本副本，以防您在删除此函数后需要重建它。）

如果您不知道 UDR 是用户定义的函数还是用户定义的过程，则可以通过使用 DROP ROUTINE 语句删除 UDR 。

要使用 DROP FUNCTION 语句，您必须是此用户定义函数的所有者（并持有数据库上的 Resource 权限）或者具有 DBA 权限。您还必须持有编写该 UDR 的程序语言的 Usage 权限。要删除外部用户定义的函数，另见 删除外部例程。

您不能使用 DROP ROUTINE 、DROP FUNCTION 或 DROP PROCEDURE 语句删除受保护的例程。有关受保护例程的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 系统目录表 **sysprocedures** 的描述。

您无法从相同的 SPL 函数内删除 SPL 函数。

如果函数定义声明了一个指定的名称，则 GBase 8s 可通过它的 *specific name* 来解析函数，如果在此语句中使用指定的名称，您还必须使用关键字 SPECIFIC，如下例所示：

```
DROP SPECIFIC FUNCTION compare_point;
```

否则，如果 *function* 名称在数据库中不唯一，则您必须指定足够的 *parameter_type* 信息来明确名称 。如果使用参数数据类型标识用户定义的函数，则它们跟在函数名后面，如下例所示：

```
DROP FUNCTION compare (int, int);
```

如果数据库服务器不能解析名义模糊的函数名称（该函数名称的特征符与另一函数的特征符只在未命名的 ROW 类型参数中不同），则数据库服务器返回错误。（当定义了含义模糊的 *function* 后，数据库服务器不能预期此错误。）

如果您包含了可选的 IF EXISTS 关键字，则如果数据库服务器在当前数据库中没有找到与 DROP FUNCTION 语句指定的相符合的函数则数据库不采取任何操作（而不是发出错误）。

确定函数是否存在

在您尝试删除用户定义函数之前，可以通过查询系统目录来检查此函数是否在数据库中存在。以下示例中，SELECT 语句从 **sysprodures** 表中检索标识为 MyFunction 的任何例程：

```
SELECT * FROM sysprocedures WHERE procname = MyFunction;
```

如果此查询返回一行，则名为 MyFunction 的 UDR 注册在当前数据库中。

如果此查询没有返回行，则您不需发出 DROP FUNCTION 语句，但是您可能希望验证 WHERE 子句指定的名称是否正确，以及您是否连接到正确的数据库。

如果此查询返回多行，则在当前数据库中重载例程 MyFunction ，并且您需要检查 MyFunction 例程的属性以确定它们中的哪些（如果有）需要通过 DROP FUNCTION 语句注销。

示例

大多数函数可以使用以下类似的 SQL 语句来删除：

```
DROP FUNCTION best_month;
```

然而，如果您有多个相同名称的函数，则通过函数重载，DROP FUNCTION 以及必须指定函数的特定名称（如果有）或参数列表来唯一标识它。例如，**superstores_demo** 数据库有两个使用以下参数创建的 **last_contact** 函数：

```
CREATE FUNCTION last_contact(cust_name name_t) ...
```

和

```
CREATE FUNCTION last_contact(c_num INT) ...
```

要删除第二个函数，使用以下语句：

```
DROP FUNCTION last_contact(INT);
```

如果使用指定的名称 last_cname_contact 和 last_cnum_contac 创建了上述函数，则删除其中的第二个函数，发出以下语句：

```
DROP SPECIFIC FUNCTION last_cnum_contact;
```

现有函数的详细信息可以在 sysprocedures 系统目录表中找到，如下所示的 SQL 查询：

```
SELECT procname, specificname, paramtypes
        FROM sysprocedures ;
```

### 删除外部函数

以 C 语言或 Java™ 语句编写用户定义的函数（UDF）称为***外部函数***。外部函数必须包含指定共享对象文件名的外部例程引用子句。在缺省情况下，只有 DBSA 授予了内置的 EXTEND 角色的用户可以创建或删除外部函数。有关此功能的附加信息，请参阅 授予 EXTEND 角色。请参阅 语言级权限 章节以获取 C 语言或 Java 语言的 USAGE ON LANGUAGE 子句的语法。

要从共享内存中删除 C 语言例程的可执行版本，请调用 **IFX_UNLOAD_MODULE** 函数。要用另一个例程替换可执行版本的 C 例程，请调用 **IFX_REPLACE_MODULE** 函数。这些内置函数都在 UDR 定义例程中有所描述。

## 2.68 DROP INDEX 语句

使用 DROP INDEX 语句删除索引。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *index* | 要删除的索引的名称 | 必须存在于当前数据库中 | 标识符 |
| *owner* | 索引所有者的名称 | 必须拥有此索引 | 所有者名称 |

**用法**

在典型的联机事务处理（OLTP）环境中，并发应用程序连续到执行 DML 操作的数据库服务器。对于每个查询，优化程序选择基于现有索引、统计和伪指令的计划。然而，在经过众多 OLTP 事务后，所选计划可能不再是查询执行的最佳计划。在这种情况下，删除低效的索引有时可以改进性能。

您必须是 *index* 的所有者或者拥有 DBA 权限才可使用 DROP INDEX 语句。以下示例删除了 joed 所拥有的索引 o_num_ix 。stores_demo 数据库必须是当前数据库。

```
DROP INDEX stores_demo:joed.o_num_ix;
```

您不能使用 DROP INDEX 语句删除唯一约束，也不能删除支持约束的索引；必须使用 ALTER TABLE . . . DROP CONSTRAINT 语句来删除该约束。当您删除约束时，数据库服务器自动删除其存在仅仅是为了支持该约束的任何索引。如果您试图使用 DROP INDEX 删除由唯一的约束共享的索引，数据库服务器会在 sysindexes 系统目录表中给指定的索引重命名，声明一个以下面的格式出现的新名称：

[space]<***tabid***>_<***constraint_id***>

这里 ***tabid*** 和 ***constraint_id*** 分布来自 **systables** 和 **sysconstraints** 系统目录表，则 **sysconstraints.idxname** 列可能更新为类似以下的内容：" 121_13" （其中引号表示空格）。如果这个索引是一个唯一索引，且只有参考约束共享它，则在重新命名之后会降级该索引为一个重复索引。

在某些上下文中，DROP INDEX 语句的替代方法是 SET Database Object Mode 语句，它可以禁用指定的索引而不将其从系统目录中删除。有关此 SQL 语句的更多信息（也可以启用当前禁用的索引），请参阅 SET Database Object Mode 语句。

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的索引，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

**示例**

*stores_demo* 数据库中的索引可以使用以下语句删除：

```
DROP INDEX zip_ix;
```

如果必要，可以将索引名指定为完全限定的四部分对象名称（*database@instance:owner.indexname*），如下例所示：

```
DROP INDEX stores_demo@prod:"gbasedbt".zip_ix ;
```

现有函数的详细信息可以在 sysprocedures 系统目录表中找到，如下例所示：

```
SELECT idxname FROM sysindices ;
```

## DROP INDEX 的 ONLINE 关键字

DBA 通过将 ONLINE 关键字作为 DROP INDEX 语句的最终规范将它包括在内，可以减少非独占错误的风险，并能增加有索引的表的可能性。当正在最小化互斥锁的持续时间时，ONLINE 关键字指示数据库服务器删除索引。当并发用户正在访问表时，可以删除索引。

缺省情况下，DROP INDEX 尝试将互斥锁放在索引的表上，以防止在正在删除索引时所有其它用户访问此表。如果另一用户已经锁定此表，或正以 Dirty Read 隔离级别访问此表，则 DROP INDEX 语句失败。

在发出 DROP INDEX ONLINE 语句之后，查询优化器不会考虑在后续的查询计划或成本估计中使用指定的索引，而且数据库服务器不支持在有索引的表上的任何其它 DDL 操作，直至已经删除了指定的索引之后。然而，在 DROP INDEX ONLINE 语句之前就已经启用的查询操作可以继续访问索引直至查询完成。

当没有其他用户正在访问索引时，数据库服务器删除索引，且 DROP INDEX ONLINE 语句终止执行。

缺省情况下，DROP INDEX ONLINE 语句不会无限期地等待要释放的锁。如果一个或多个并发会话持有表的锁，该语句可能会发生错误 -216 或 -113 而失败，除非您首先发出 SET LOCK MODE TO WAIT 语句以指定无限期等待。否则，DROP INDEX ONLINE 使用 DEADLOCK_TIMEOUT 配置参数指定的锁定的等待期，或者指定先前的 SET LOCK MODE 语句。要避免锁定错误，请在联机删除索引之前执行 SET LOCK MODE TO WAIT （没有特定的限制）。

不能使用 CREATE INDEX 语句声明一个已经具有相同标识的新索引，直至已经删除了指定索引之后。最多有一个 CREATE INDEX ONLINE 或 DROP INDEX ONLINE 语句可以在同一个表上并发地引用索引。

DROP INDEX ONLINE 语句中的索引的表可以是永久的或临时的、记录的或不记录的以及分片的或不分片的。然而，当您删除具有以下属性的索引时，不能指定 ONLINE 关键字：

- 功能索引
- 集群索引
- 虚拟索引
- R-tree 索引

以下语句指示了数据库服务器联机删除索引 idx_01：

```
DROP INDEX IF EXISTS idx_01 ONLINE;
```

## 2.69 DROP OPCLASS 语句

使用 DROP OPCLASS 语句从数据库中删除现有的运算符类。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *opclass* | 要删除的运算符类的名称 | 必须已经由先前的 CREATE OPCLASS 语句创建 | 标识符 |
| *owner* | opclass 所有者的名称 | 必须拥有此运算符类 | 所有者名称 |

**用法**

您必须是运算符类的所有者或拥有 DBA 特权才可使用 DROP OPCLASS 语句。

如果您包含可选的 IF EXISTS 关键字，则如果指定名称的运算符类已经在当前数据库中注册，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

如果数据库包含已在您计划要删除的运算符类上定义的索引，则 RESTRICT 关键字会导致 DROP OPCLASS 失败。因此，在删除运算符类之前，必须使用 DROP INDEX 语句来删除任何从属的索引。

以下 DROP OPCLASS 语句删除了一个称为 abs_btree_ops 的运算符类：

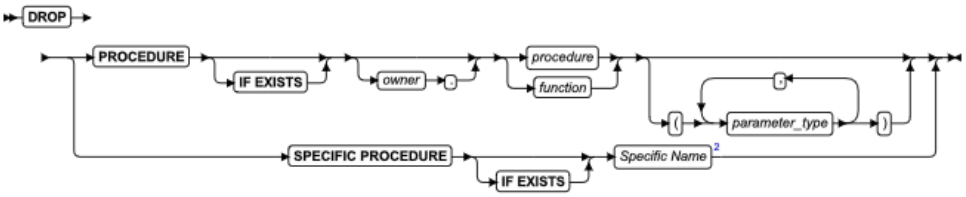
```
DROP OPCLASS abs_btree_ops RESTRICT
```

如果您包含可选的 IF EXISTS 关键字，则如果没有指定名称（或者如果您包含 SPECIFIC 关键字，为指定的特定名称）的函数在当前数据库中注册，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

## 2.70 DROP PROCEDURE 语句

使用 DROP PROCEDURE 语句从数据库中删除用户定义的过程。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 要删除的过程或 SPL 函数的名称 | 必须在数据库中存在（即已注册） | 标识符 |
| *owner* | UDR 所有者的名称 | 必须拥有过程或 SPL 函数 | 所有者名称 |
| *parameter_type* | 参数的数据类型 | 数据类型（或数据类型列表）必须和那些在创建过程中就已指定的数据类型是相同的数据类型（并且具有相同的顺序） | 标识符；数据类型 |
| *procedure* | 要删除的用户定义的过程名称 | 必须在数据库中存在（即已注册） | 数据库对象名 |

**用法**

删除用户定义的过程会除去该过程的文本和可执行的版本。您无法在同一个 SPL 过程内删除 SPL 过程。

不能使用 DROP ROUTINE 、DROP FUNCTION 或 DROP PROCEDURE 语句删除受保护的例程。有关受保护的例程的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 中 **sysprocedures** 系统目录表的描述。

要使用 DROP PROCEDURE 语句，您必须是过程的所有者并且还必须具有数据库的 Resource 权限或 DBA 权限。您必须还需具有编写此 UDR 所使用的程序语言的 Usage 权限。要删除一个外部用户定义的过程，另见 删除外部过程。

如果 *function* 或 *procedure* 名称在数据库中不是唯一的，则您必须指定足够的 *parameter_type* 信息以区分这些名称。如果数据库服务器无法解析一个意义含糊的 UDR 名称，即它的签名与另一个 UDR 的签名只是在一个未命名的 ROW 类型参数中不同，则返回一个错误。（当定义了含义模糊的 ***function*** 或 ***procedure*** 时，数据库服务器不能预期此错误。）

如果不知道 UDR 是用户定义的过程还是用户定义的函数，您可以使用 DROP ROUTINE 语句。有关更多信息，请参阅 DROP ROUTINE 语句。

对于较早的 GBase 8s 版本的向后兼容性，可以使用此语句来删除由 CREATE PROCEDURE 语句创建的 SPL 函数。您可以在过程名称的后面包括参数数据类型，以识别过程：

```
DROP PROCEDURE compare(int, int);
```

如果对用户定义的过程使用了指定的名称，还必须用关键字 SPECIFIC，如下例所示：

```
DROP SPECIFIC PROCEDURE compare_point;
```

如果您包含了可选的 IF EXISTS 关键字，则如果没有指定名称的过程在当前数据库中注册，则数据库服务器不采取任何操作（而不是向应用程序发送错误）。

**确定过程是否存在**

在您尝试删除用户定义的过程之前，可以通过查询系统目录来检查此过程是否在数据库中存在。以下示例中，SELECT 语句从 sysprodures 表中检索标识为 MyProcedure 的任何过程：

```
SELECT * FROM sysprocedures WHERE procname = MyProcedure;
```

如果此查询返回一行，则名为 MyProcedure 的 UDR 注册在当前数据库中。

如果没有返回行，则您无需发出 DROP PROCEDURE 语句，但是您可能希望验证 WHERE 子句指定的名称是否正确，以及您是否连接到正确的数据库。

如果此查询返回多行，则在当前数据库中重载过程 MyProcedure ，并且您需要检查 MyProcedure 过程的属性以确定它们中的哪些（如果有）需要通过 DROP PROCEDURE 语句注销。

## 删除外部过程

以 C 语言或 Java™ 语句编写用户定义的过程（UDP）称为***外部例程***。外部例程必须包含指定共享对象文件名的外部例程引用子句。在缺省情况下，只有 DBSA 授予了内置的 EXTEND 角色的用户可以创建或删除外部例程。您还必须具有编写此 UDR 的外部程序语言的 Usage 特权。有关 EXTEND 角色安全功能的其它信息请参阅 授予 EXTEND 角色。 有关 C 语言或 Java 语言的 USAGE ON LANGUAGE 子句的语法使用，请参阅 语言级权限 。

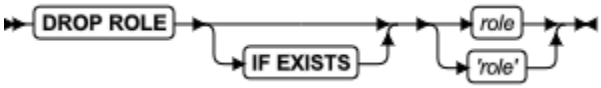
要从共享内存中删除 C 语言例程的可执行版本，请调用 **IFX_UNLOAD_MODULE** 函数。要用另一个例程替换可执行版本的 C 例程，请调用 **IFX_REPLACE_MODULE** 函数。这些内置函数都在 UDR 定义例程中有所描述。

## 2.71 DROP ROLE 语句

使用 DROP ROLE 语句从数据库中删除用户定义的角色。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *role* | 要删除的角色的名称 | 必须在本地数据库中注册，当前 *role* 名称用引号括起时，它是区分大小写的。 | 所有者名称 |

**用法**

DBA 或者用 WITH GRANT OPTION 关键字授予角色的用户可以发出 DROP ROLE 语句。（如 ***user*** 名称一样，***role*** 是授权标识而不是数据库对象，因为 ***role*** 没有所有者。）

如果您包含了可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定的名称的角色 ，则数据库不执行任何操作（而不是向应用程序发送异常）。

在您删除角色后，没有用户可以授权或启用已删除的角色，而且当前角色被删除后，任何被指定了该角色的用户将失去它的特权（如表级别特权或例程级别特权）。除非单独对 PUBLIC 或用户授予相同的权限。如果已删除的角色时用户的缺省角色，则该用户的缺省角色变成 NULL。

以下语句删除了 engineer 角色：

```
DROP ROLE engineer;
```

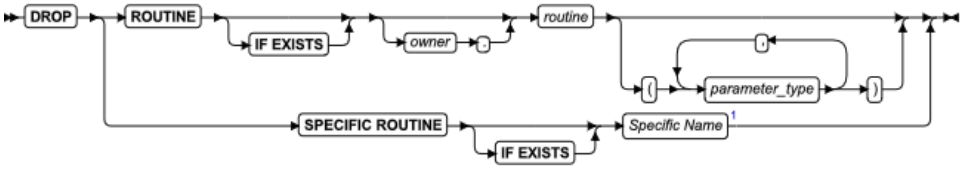
您不能使用 DROP ROLE 语句删除内置的角色，例如 EXTEND 或 GBase 8s 的 DBSECADM 角色。

## 2.72 DROP ROUTINE 语句

使用 DROP ROUTINE 语句从数据库中删除用户定义的例程（UDR）。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | UDR 所有者的名称 | 必须拥有 UDR | 所有者名称 |
| *parameter_type* | *routine* 参数的数据类型 | 数据类型（或数据类型列表）必须与 UDR 定义中的数据类型是相同类型(并且以相同的顺序指定） | 标识符；数据类型 |
| *routine* | 要删除的 UDR 的名称 | UDR 必须在数据库中存在(即已注册） | 标识符 |

### 用法

删除 UDR 会从数据库中除去 UDR 的文本和可执行的版本。如果不指定 UDR 是用户定义的函数还是用户定义的过程，则此语句指导服务器删除指定的用户定义的函数或用户定义的过程。

要使用 DROP ROUTINE 语句，您必须是 UDR 的所有者（并且持有数据库的 Resource 特权），或者您必须具有 DBA 特权。您还必须持有编写此 UDR 的程序语言的 Usage 特权。要删除外部的用户定义的例程，请参阅删除外部例程.。

## 限制

您无法从相同的 SPL 例程内删除 SPL 例程。

您不能使用 DROP ROUTINE 、DROP FUNCTION 或 DROP PROCEDURE 语句删除受保护的例程。有关受保护例程的更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 中系统目录表 **sysprocedures** 的描述。

要使用 DROP ROUTINE 语句注销 UDR，UDR 的类型不能含糊不清。您指定的 UDR 的名称必须引用用户定义的函数或用户定义的过程。如果存在以下任何一个条件，则数据库服务器会返回一个错误：

- 您指定的名称（或参数）同时应用于用户定义的过程和用户定义的函数。
- 您指定的 ***specific*** 名称同时应用于用户定义的过程和用户定义的函数。

如果***例程***名称在数据库内不唯一，则您必须指定足够的 *parameter_type* 信息以区分这些名称。如果数据库服务器无法解析一个意义含糊的 UDR 名称，即它的签名与另一个 UDR 的签名只是在一个未命名的 ROW 类型参数中不同，则返回一个错误。（当定义了含义模糊的 ***function*** 或 ***procedure*** 时，数据库服务器不能预期此错误。）

如果使用参数类型来标识 UDR，则这些参数数据类型应跟在 UDR 名称后面，如下例所示：

```
DROP ROUTINE compare(INT, INT);
```

如果您对 UDR 使用特定的名称，则必须包含关键字 SPECIFIC ，如下例所示：

```
DROP SPECIFIC ROUTINE compare_point;
```

如果您包含了可选的 IF EXISTS 关键字，则如果数据库服务器在当前数据库中没有找到与 DROP ROUTINE 语句指定的相符合的函数则数据库不采取任何操作（而不是发出错误）。

**确定例程是否存在**

在您尝试删除用户定义的例程之前，可以通过查询系统目录来检查此例程是否在数据库中存在。以下示例中，SELECT 语句从 sysprodures 表中检索标识为 MyRoutine 的任何例程：

```
SELECT * FROM sysprocedures WHERE procname = MyRoutine;
```

如果此查询返回一行，则名为 MyRoutine 的 UDR 注册在当前数据库中。

如果没有返回行，则您无需发出 DROP ROUTINE 语句，但是您可能希望验证 WHERE 子句指定的名称是否正确，以及您是否连接到正确的数据库。

如果此查询返回多行，则在当前数据库中重载例程 **MyRoutine** ，并且您需要检查 **MyRoutine** 例程的属性以确定它们中的哪些（如果有）需要通过 DROP ROUTINE 语句注销。

## 删除外部例程

以 C 语言或 Java™ 语句编写用户定义的例程（UDR）称为外部例程。外部例程必须包含指定共享对象文件名的外部例程引用子句。在缺省情况下，只有 DBSA 授予了内置的 EXTEND 角色的用户可以创建或删除外部例程。您还必须具有编写此 UDR 的外部程序语言的 Usage 特权。有关 EXTEND 角色安全功能的其它信息请参阅 授予 EXTEND 角色。有关 C 语言或 Java 语言的 USAGE ON LANGUAGE 子句的语法使用，请参阅 语言级权限 。

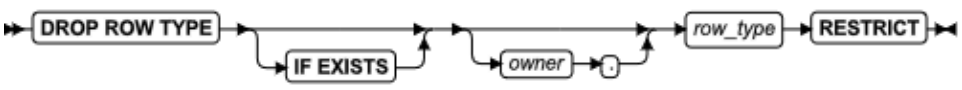
要从共享内存中删除 C 语言例程的可执行版本，请调用 **IFX_UNLOAD_MODULE** 函数。要用另一个例程替换可执行版本的 C 例程，请调用 **IFX_REPLACE_MODULE** 函数。这些内置函数都在 UDR 定义例程中有所描述。

# 2.73 DROP ROW TYPE 语句

使用 DROP ROW TYPE 语句从数据库中删除现有的名为 ROW 的数据类型。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| *owner* | ROW 类型所有者的名称 | 必须是 row_type 的所有者 | 所有者名称 |
| *row_type* | 要删除的现有的 ROW 数据类型的名称 | 必须存在。另见后面的**用法**部分 | 标识符； 数据类型 |

**用法**

DROP ROW TYPE 语句从 **sysxtdtypes** 系统目录表中删除指定 ***row_type*** 的条目。您必须是指定名称的 ROW 数据类型的所有者或具有 DBA 权限才可以 使用 DROP ROW TYPE 语句。

如果您包含可选的 IF EXISTS 关键字，则没有指定名称的 ROW 数据类型在当前数据库中存在，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

如果一个已命名的 ROW 数据类型的名称正在使用中，则不能删除该数据类型。当以下的任何条件为真时，不能删除已命名的 ROW 数据类型：

- 任何现有的表或列正在使用已命名的 ROW 数据类型。
- 已命名的 ROW 数据类型在继承层次结构中是超类型。
- 在已命名的 ROW 数据类型的列上定义了视图。

要从表中删除名为 ROW 类型的列，请使用 ALTER TABLE。

DROP ROW TYPE 语句不能删除未命名的 ROW 数据类型。

### RESTRICT 关键字

RESTRICT 关键字要求同 DROP ROW TYPE 语句使用。如果 ***row_type*** 上的从属性存在，则 RESTRICT 会引起 DROP ROW TYPE 失败。

如果以下任一条件为真，则 DROP ROW TYPE 语句失败并返回错误消息：

- 已命名的 ROW 数据类型用于现有的表或列。

  查看 **systables** 和 **syscolumns** 系统目录表来查找是否任何表或数据类型使用了已命名的 ROW 数据类型。

- 已命名的 ROW 数据类型在继承层次结构中是超类型。

  查找 **sysinherits** 系统目录表来查看哪一个已命名的 ROW 数据类型具有子类型。

以下语句删除了已命名的 ROW 数据类型 employee_t：

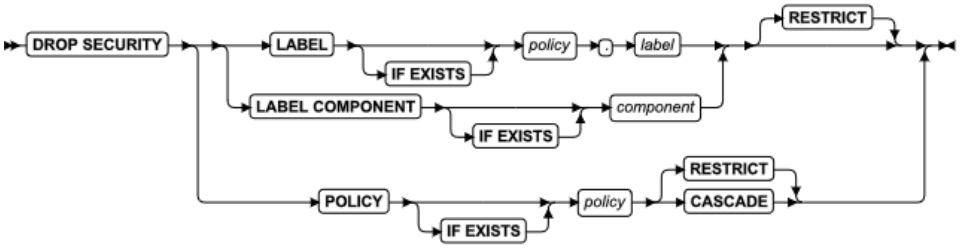
```
DROP ROW TYPE employee_t RESTRICT
```

## 2.74 DROP SECURITY 语句

使用 DROP SECURITY 语句从当前数据库中删除现有的安全对象。此对象可以是安全策略、安全标签或安全标签组件。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *component* | 要删除的安全标签组件 | 必须在数据库中存在 | 标识符 |
| *label* | 要删除的安全标签 | 必须作为指定 *policy* 的标签存在于数据库中 | 标识符 |
| *policy* | 要删除的安全策略 | 必须在数据库中存在 | 标识符 |

**用法**

只有 DBSECADM 能发出此语句。当成功执行 DROP SECURITY 语句后，数据库服务器从系统目录的表中删除引用指定对象的名称和数字标识符的任何行，这些表为：

- 对于安全策略是 **sysecpolicies**
- 对于安全标签是 **sysseclabels**
- 对于安全标签组件是 **sysseclabelcomponents** 。

跟随在 SECURITY 关键字之后的关键字标识正在删除的安全对象的类型。

- SECURITY POLICY ***policy*** 指定一个安全策略
- SECURITY LABEL ***policy.label*** 指定一个安全标签
- SECURITY LABEL COMPONENT ***component*** 指定一个安全标签组件。

没有 SQL 语句可以在不会破坏整个组件的情况下选择性地删除安全标签组件的某些元素。要从数据库中只删除安全标签组件的一部分元素，DBSECADM 可以使用 DROP SECURITY LABEL COMPONENT 语句删除组件，然后使用 CREATE SECURITY LABEL COMPONENT 语句重新定义删除的子句，但不包括任何不再需要的元素。（另一种方法是删除包含已弃用元素的所有安全标签，然后使用 CREATE SECURITY LABEL 语句重新定义具有已删除标签相同的组件的新标签，但不包含这些元素。在这种情况下，已弃用的元素将保留在数据库中，但是没有安全标签会将它们用作组件的值。）

如果您包含了可选的 IF EXISTS 关键字，则如果在当前数据库中未注册指定的安全对象类型和指定名称的安全对象，则数据库服务器不会执行任何操作（而不是向应用程序发送异常）。

**示例**

以下语句指示数据库服务器删除安全标签 **witty**：

```
DROP SECURITY LABEL witty;
```

如果有列被 witty 标签保护或者有用户持有此标签，则该语句失败。

下一个示例指示数据库服务器从数据库中删除安全标签组件 adhesive：

```
DROP SECURITY LABEL COMPONENT adhesive;
```

如果安全策略取决于 adhesive 安全标签组件，则该语句失败。

以下示例指示数据库服务器以 CASCADE 方式删除 best 安全策略：

```
DROP SECURITY POLICY best CASCADE;
```

如果该策略正在包含任何表，则该语句失败。但是，如果该语句成功，则因为 CASCADE 规范它会产生以下附加的影响：

- 所有与 **best** 安全策略的相关联的安全标签都会被删除。
- 所有 **best** 安全策略撤销的豁免权。
- 由于 **best** 安全策略而删除的所有安全标签都将从拥有这些标签的所有用户中撤销。

### 以 RESTRICT 方式或 CASCADE 方式删除安全标签对象

缺省情况下，当删除任何安全对象时，RESTRICT 关键字生效。在 CASCADE 模式下指定删除安全***策略***。如果满足以下任一条件，DBSECADM 不能删除 RESTRICT 模式下的安全策略：

- 表由安全策略保护
- 安全标签取决于此安全策略
- 用户已获得该安全策略规则的豁免。

如果安全策略正在保护表，则该策略不能以 CASCADE 模式删除。当安全策略以 CASCADE 模式成功删除时，则下列安全对象也会被删除或撤销：

- 所有与已删除的安全策略相关联的安全标签
- 所有已删除的安全标签也将从拥有这些标签的用户中撤销
- 撤销删除的安全策略的所有豁免。

如果满足以下任一条件，则不能在 RESTRICT 模式中删除安全标签（RESTRICT 模式是唯一支持的删除安全标签的模式）：

- 列被安全标签保护
- 用户拥有此安全标签。

如果任何安全策略取决于该安全标签组件，则不能在 RESTRICTA 模式中删除安全标签组件，这是唯一支持的删除组件的模式。

## 2.75 DROP SEQUENCE 语句

使用 DROP SEQUENCE 语句从数据库中删除序列对象。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | 序列所有者的名称 | 必须拥有序列对象 | 所有者名称 |
| *sequence* | 序列的名称 | 必须存在于当前数据库中 | 标识符 |

**用法**

此语句从 **syssequences** 系统目录表中删除 ***sequence*** 条目。要删除序列，您必须是它的所有者或拥有对数据库的 DBA 特权。在兼容 ANSI 的数据库中，如果您不是所有者，您必须拥有它的所有者的名（***owner.sequence***）。

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的序列对象，则数据库服务器不执行任何操作（而不是向应用程序发送异常）。

如果删除一个序列。则该序列的名称的任何同义词也会由数据库服务器自动删除。

您不可以使用同义词指定 DROP SEQUENCE 语句中 ***sequence*** 的标识符。

**示例**

假设您使用以下语句创建一个序列：

```
CREATE SEQUENCE Invoice_Numbers
        START 10000 INCREMENT 1 NOCYCLE ;
```

该序列可以由此语句删除：

```
DROP SEQUENCE Invoice_Numbers;
```

可以通过连接 *syssequences* 和 *systables* 系统目录表来查看现有序列的详细信息，如下例所示：

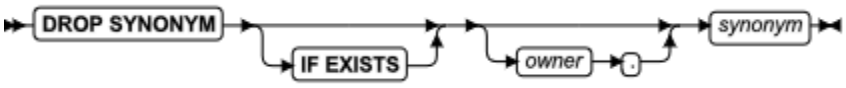
```
SELECT t.tabname SeqName
        FROM Syssequences s, Systables t
        WHERE t.tabid = s.tabid ;
```

## 2.76 DROP SYNONYM 语句

使用 DROP SYNONYM 语句注销现有的同义词。

该语句是 SQL ANSI/ISO 标准的扩展。

语法



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | *synonym* 的所有者 | 必须拥有 *synonym* | 所有者名称 |
| *synonym* | 要删除的同义词 | 该同义词必须存在于当前数据库中 | 标识符 |

**用法**

此语句从 **systables** 、**syssynonyms** 和 **syssyntable** 系统目录表中删除条目。您必须是 ***synonym*** 的所有者或者拥有 DBA 特权才能执行 DROP SYNONYM 语句。删除同义词对同义词指向的表、视图或系列对象都没有影响。

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的同义词则数据库服务器不执行任何操作（而不是向应用程序发送异常）。

下列语句删除用户 cathyg 拥有的同义词 nj_cust：

```
DROP SYNONYM cathyg.nj_cust;
```

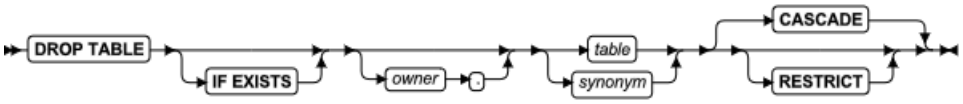
DROP SYNONYM 并不是唯一一个可以注销同义词的 DDL 操作，如果删除表、视图或序列，则同一数据库中的任何同义词以及指代该表、视图或序列的同义词也会被删除。

但是，如果当前数据库中的同义词引用另一个数据库中的已删除表或视图，那么该同义词将保留在系统目录中，直至使用 DROP SYNONYM 语句显式删除该同义词。 您可以在同一数据库中创建另一个表或视图，并声明已删除的表或视图的名称作为其标识符。（如果不是当前数据库中的任何表或对象的名称，您可以在当前数据库中创建一个表、视图或序列对象，并将在其它数据库中的表中删除的表或视图的标识符声明为其名称。）在另一种情况中，旧的同义词现在会引用新的表对象。有关同义词链接的更完整的讨论，请参阅 CREATE SYNONYM 语句描述中的链接同义词一节。

## 2.77 DROP TABLE 语句

使用 DROP TABLE 语句可删除表，以及与之关联的索引和数据。该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | 表所有者的名称 | 必须拥有表 | 所有者名称 |
| *synonym* | 要删除的表的本地同义词 | 该同义词和表必须存在，USETABLENAME 必须设置为 1 | 标识符 |
| *table* | 要删除的表的名称 | 必须在本地数据库的 systables | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 系统目录表中注册 | |

**用法**

您必须是表的所有者或拥有 DBA 特权才能使用 DROP TABLE 语句。

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的表，则数据库服务器不执行任何操作（而不是向应用程序发送异常）。

您无法删除系统目录表。

如果发出 DROP TABLE 语句，则 DB-Access 不会提示您验证是否想要删除整个表。

## DROP TABLE 语句的效果

使用 DROP TABLE 语句要谨慎。当您删除一个表时，也会删除存储在其中的数据、索引或对列的约束（包括对该表的列的所有参考约束）、分配给该表的任何本地同义词、在该表上创建的任何触发器以及给予该表的任何授权。同时您也删除了基于该表的所有视图以及与该表相关联的任何违例和诊断表。

DROP TABLE 不会删除在外部数据库中创建的表的任何同义词。要删除以删除的表的外部的同义词，必须使用 DROP SYNONYM 语句显式地执行此操作。

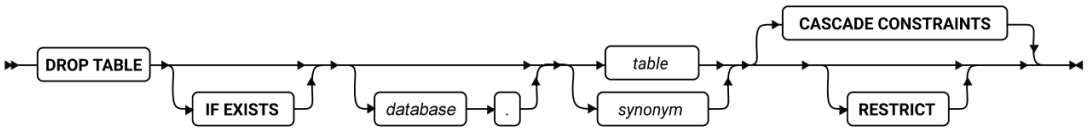
您可以通过设置 **USETABLENAME** 环境变量来防止用户在 DROP TABLE 语句中指定同义词。如果设置了 **USETABLENAME** ，当前任何用户试图指定 DROP TABLE ***synonym*** 时将导致错误。

## 指定 CASCADE 方式

DROP TABLE 中的 CASCADE 关键字会删除相关的数据库对象，包括构建在表上的参考约束、定义在表上的视图以及与表相关联的任何违例和诊断表。

如果表在继承层次结构中是超级表，则 CASCADE 删除所有的子表和超级表。
CASCADE 方式是 DROP TABLE 语句的缺省方式。您也可以使用 CASCADE 关键字显式地指定此方式。

在 Oracle 模式下，保持 GBase 8s 原有级联删除方式不变的基础上，DROP TABLE 支持指定 CASCADE CONSTRAINTS 关键字级联删除。如果不指定方式，RESTRICT 为 DROP TABLE 语句的缺省方式，如果数据库中有构建在被删除表上的视图，数据库返回错误信息。




例如，执行以下语句对 test_tab 表级联删除：

```
DROP TABLE IF EXISTS test_tab CASCADE CONSTRAINTS;
```

### 指定 RESTRICT 方式

RESTRICT 关键字可以控制对以下对象的删除操作：超级表、在该表上定义的视图、含有与表相关联的违例和诊断表。如果任何以下条件为真，则使用 RESTRICT 选项会引起删除操作失败并返回一条出错消息：

- 现有的参考约束引用 ***table*** 。
- 现有的视图定义在 ***table*** 上。
- 任何违例表或诊断表与 ***table*** 相关联。
- ***table*** 在继承层次结构中是超级表。

### 删除包含不透明数据类型的表

当删除一些不透明数据类型时，它们需要特别的处理过程。例如，如果某个不透明类型包含空间或多重表示数据，那么它可能提供如何存储数据的选项：存储于内部结构中或者（对于大对象）智能大对象中。

数据库服务器通过调用称为 **destroy( )** 的用户定义的支持函数删除不透明类型。对行包含不透明类型的表执行 DROP TABLE 语句时，数据库服务器自动调用该类型的 **destroy( )** 函数。在删除表之前，**destroy( )** 函数可以对不透明数据类型的列执行某些确定的函数。有关 **destroy( )** 支持函数的更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

### 无法删除的表

可以删除的表的类型是有限制的。

- 不能删除任何系统目录表。
- 无法删除不在当前数据库中的表。
- 不能删除违例表或诊断表。

在删除表之前，您必须先对违例表和诊断表所关联的基本表发出 STOP VIOLATIONS TABLE 语句。

以下示例删除了当前数据库中的两个表。它们都被当前用户 joed 拥有。且与违例表和诊断表都没有关联，表上也没有定义引用约束或视图。

```
DROP TABLE customer;
DROP TABLE stores_demo@accntg:joed.state;
```

## 2.78 DROP TRIGGER 语句

使用 DROP TRIGGER 语句从数据库中删除触发器定义。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | 触发器所有者的名称 | 必须拥有触发器 | 所有者名称 |
| *trigger* | 要删除的触发器的名称 | 触发器必须存在于当前数据库中 | 标识符 |

**用法**

您必须是触发器的所有者或者具有 DBA 特权才能删除触发器。删除触发器会从数据库中删除该触发器定义的文本和可执行的触发器，描述指定触发器的行会从 **systriggers** 系统目录表中删除。

如果包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的触发器，则数据库服务器不执行任何操作（而不是向应用程序发送异常）。

在复杂视图（带有来自多个表的列的视图）上删除 INSTEAD OF 触发器撤销了对于视图的所有特权（这些特权是在创建触发器时触发器所有者自动接收的），还撤销了触发器所有者授予其它用户的所有特权。（删除单个视图上的触发器不会取消任何特权。）

以下示例删除了 items_pct 触发器：

```
DROP TRIGGER items_pct;
```

如果 DROP TRIGGER 语句出现在由数据操纵语句（DML）调用的 SPL 例程内，则数据库服务器返回一个错误。

当对同一触发事件在同一表或视图上定义多个触发器时，不保证触发器的执行顺序。如果您有一个首先的执行顺序，但是触发器是以其它顺序执行的，则您可能希望删除除了首先运行的触发器之外的所有触发器，然后按照相对顺序（您希望的顺序）重新创建其它触发器， 以便它们按照与其的执行顺序列在系统目录中。

## 2.79 DROP TRUSTED CONTEXT 语句

使用 DROP TRUSTED CONTEXT 语句删除可信上下文对象。

该语句是 SQL ANSI/ISO 标准的扩展。您必须拥有数据库服务器安全管理员（DBSECADM）角色才能删除可信上下文。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *context* | 要删除的可信上下文对象 | 必须存在于当前 GBase 8s 数据库服务器实例中 | 标识符 |

**用法**

当 DROP TRUSTED CONTEXT 语句成功执行后，指定的可信上下文对象被销毁。所有 *context* 的引用都会从 GBase 8s 数据库服务器实例的 sysuser 数据库的这些表中删除：

- systrustedcontext
- systcxattributes
- systcxusers.

如果您删除可信上下文而此可信上下文连接处于活动状态，那么这些连接将保持授信任，直到它们终止，或直到下一个重新尝试。但是，如果尝试在这些可信的连接上切换用户，则会返回错误。

以下 DROP TRUSTED CONTEXT 语句的示例删除了可信上下文对象 cntx1：

```
DROP TRUSTED CONTEXT cntx1;
```

如果 cntx1 不是当前数据库服务器实例的可信上下文对象的名称，则该示例失败。

## 2.80 DROP TYPE 语句

使用 DROP TYPE 语句从数据库中删除用户定义的 Distinct 或 Opaque 数据类型。（您无法使用此语句删除内置数据类型。）

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | 要删除的 Distinct 或 Opaque 数据类型的名称 | 必须是一个本地数据库中现有的用户定义的 Distinct 或 Opaque 类型；不能是内置数据类型 | 标识符 |
| *owner* | 数据类型所有者的名称 | 必须拥有数据类型 | 所有者名称 |

**用法**

要使用 DROP TYPE 语句删除 Distinct 或 Opaque 数据类型，您必须是此数据库类型的所有者或者拥有 DBA 特权。当您使用此语句时，就从数据库（sysxtdtypes 系统目录表中）中删除了数据类型定义。通常，此语句不会删除强制转型的任何定义与该数据类型相关的支持函数的任何定义。

**重要：** 当删除 Distinct 类型时，数据库服务器会自动删除在 Distinct 及其基于的类型之间的两个显式的强制转型。

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册用户定义的 Distinct 或 Opaque 数据类型，则数据库服务器不执行任何操作（而不是向应用程序发送异常）。

如果您尝试删除内置数据类型，则 DROP TYPE 语句失败。如，内置的 Opaque BOOLEAN 或 LVARCHAR 类型、或者内置的 Distinct IDSSECURITYLABEL 类型。

如果数据库包含任何定义引用了 Distinct 或 Opaque 类型的强制转型、列或用户定义的函数，则不能删除该数据类型。

以下语句删除 new_type 数据类型：

```
DROP TYPE new_type RESTRICT;
```

## 2.81 DROP USER 语句 (UNIX™、Linux™)

使用 DROP USER 语句删除内部用户。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *user* | 您要删除的指定用户的认证标识符 | 必须是现有的认证标识符 | 所有者名称 |

**用法**

只有 DBSA 才能运行 DROP USER 语句。在非 root 安装中，安装服务器的用户等同于 DBSA，除非该用户将 DBSA 特权委托给另一个用户。

当指定的用户处于连接的状态时，不建议您运行 DROP USER 语句。

DROP USER 语句的执行可以使用 DRUR 审计代码审计。

**示例**

以下语句删除用户 **bill**：

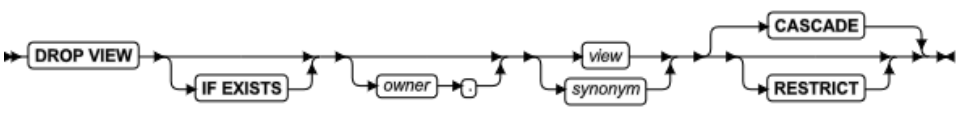
```
DROP USER bill;
```

## 2.82 DROP VIEW 语句

使用 DROP VIEW 语句从数据库中删除视图。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *owner* | 视图所有者的名称 | 必须拥有视图 | 所有者名称 |
| *synonym* | 该语句删除的视图的同义词 | 它指向的 *synonym* 和视图必须在本地数据库中存在 | 标识符 |
| *view* | 要删除的视图的名称 | 必须存在于 systables | 标识符 |

**用法**

要删除视图，您必须是所有者或拥有 DBA 特权。

当删除一个视图时，也会删除依赖于此视图的任何其它视图和 INSTEAD OF 触发器。（您也可以使用 CASCADE 关键字显式地指定这个缺省的行为。）

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的视图，则数据库服务器不执行任何操作（而不是向应用程序发送异常）。

当在 DROP VIEW 语句中使用 RESTRICT 关键字时，如果任何其它的现有视图是定义在*视图*上的，则删除操作失败；否则，会在删除操作中放弃这些视图。

您可以查询 **sysdepend** 系统目录表以确定哪些视图（如果有）依赖于另一个视图。

以下语句删除了名为 cust1 的视图：

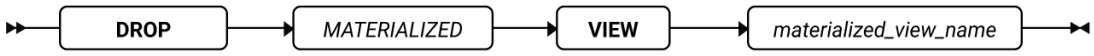
```
DROP VIEW cust1
```

## 2.83 DROP MATERIALIZED VIEW 语句

通过使用 drop materialized view 语句来完成物化视图的删除。

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| materialized_view_name | 物化视图名称 | 数据库中存在 | 标识符 |

**用法**

- 要删除物化视图，您必须是所有者或拥有 DBA 特权。

- 当删除一个物化视图时，也会删除依赖于此物化视图的任何其它视图。

例如下面示例：

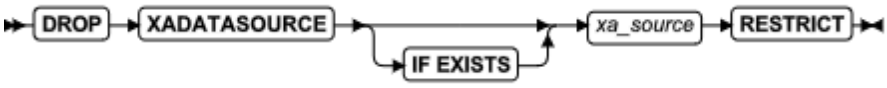
```
drop materialized view mv_test;
```

## 2.84 DROP XADATASOURCE 语句

使用 DROP XADATASOURCE 语句从数据库的系统目录中删除之前定义的符合 XA 的数据源。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *xa_source* | 要删除的符合 XA 的数据源 | 必须存在于 sysxadatasources 系统目录表中 | 标识符 |

**用法**

RESTRICT 关键字是必需的。您必须是 XA 数据源的所有者或者持有 DBA 特权才能删除存取方法。

DROP XADATASOURCE 语句在高可用集群中的辅助服务器上不支持。

如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的 XA 数据源，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

以下语句删除了名为 NewYork 的 XA 数据源实例，它被用户 gbasedbt 拥有。

```
DROP XADATASOURCE gbasedbt.NewYork RESTRICT;
```

如果存取方法正在被当前打开的事务使用，则不能删除该方法。如果已对未完成的事务注册了 XA 数据源，则只能在数据库关闭或会话结束后删除数据源。

## 2.85 DROP XADATASOURCE TYPE 语句

使用 DROP XADATASOURCE TYPE 语句从数据库在删除之前定义的符合 XA 的数据源类型。

该语句是 SQL ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *xa_type* | 要删除的 XA 数据 | 必须存在于 sysxasourcetypes | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 源类型的名称 | 系统目录表中 | |

**用法**

RESTRICT 关键字是必需的。如果虚拟表或索引存在并使用了此数据源，则您不能删除 XA 数据源类型、您必须是用户 **gbasedbt** 或具有 DBA 特权才能删除 XA 数据源类型。

DROP XADATASOURCE TYPE 语句在高可用集群中的辅助服务器上不支持。

以下语句删除了名为 MQSeries® 的 XA 数据源类型，它被用户 gbasedbt 拥有：

```
DROP XADATASOURCE TYPE gbasedbt.MQSeries RESTRICT;
```

您不能删除 XA 数据源类型，直到所有使用数据源类型的 XA 数据源实例都被删除后，您才可以删除 XA 数据源类型。

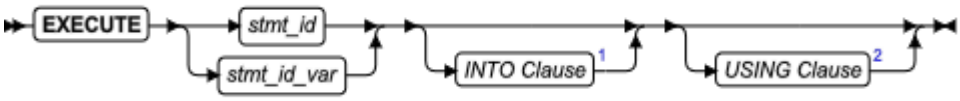
如果您包含可选的 IF EXISTS 关键字，则如果未在当前数据库中注册指定名称的 XA 数据源类型，则数据库服务器不采取任何操作（而不是向应用程序发送异常）。

## 2.86 EXECUTE 语句

使用 EXECUTE 语句来运行一个先前准备好的语句或一个准备好的多语句对象。

请随同 GBase 8s ESQL/C 使用本语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *stmt_id* | 准备好的 SQL 语句的标识符 | 必须在先前 PREPARE 语句中已经声明 | 标识符 |
| *stmt_id_var* | 包含准备好的语句标识符的主变量 | 必须存在，且必须包含一个先前 PREPARE 语句已声明的语句标识符，并必须为字符数据类型 | PREPARE 语句 |

**用法**

EXECUTE 语句向数据库服务器传递一个准备好的 SQL 语句来执行。下列示例展示 GBase 8s ESQL/C 程序内的 EXECUTE 语句：

```
EXEC SQL PREPARE del_1 FROM
    'DELETE FROM customer WHERE customer_num = 119';
EXEC SQL EXECUTE del_1;
```

一旦准备好，SQL 语句可根据需要执行。

在（使用 FREE 语句）释放数据库服务器资源之后，您不可随同 DECLEAR 游标或随同 EXECUTE 语句来使用该语句标识符，直到再次准备该语句。

如果该语句包含问号（?）占位符，则请在执行之前使用 USING 子句来为它们提供指定的值。要获取更多信息，请参阅 USING 子句。

您可执行任意准备好的语句，除了下列列表中的那些：

- 返回超过一行的准备好的 SELECT 语句

  当您使用准备好的 SELECT 语句来返回多行数据时，必须使用游标来检索这些数据行。作为替代方法，您可 EXECUTE 准备好的 SELECT INTO TEMP 语句来取得相同的结果。

  要获取更多关于游标的信息，请参阅 DECLARE 语句。

- 返回多于一行的 SPL 函数的准备好的 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句

  当您准备 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句来调用返回多行的 SPL 函数时，必须使用游标来检索这些数据行。

  要获取更多关于如何执行 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句的信息，请参阅 PREPARE 语句。

在准备触发 INSERT、DELETE 或 UPDATE 之后，如果您创建或删除触发器，则当您执行准备好的语句时，该语句返回错误。

## 语句标识符的作用域

程序可由一个或多个源代码文件组成。缺省情况下，语句标识符引用的作用域对于程序是全局的。在一个文件中创建的语句标识符可从另一个文件引用。

在多文件程序中，如果您想要将语句标识符的引用作用域限定到所执行的文件，则可以 **-local** 命令行选项预处理所有文件。

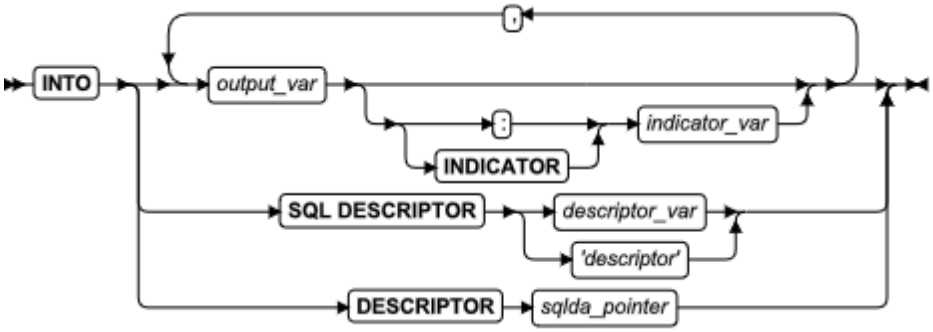
### INTO 子句

使用 INTO 子句来保存这些 SQL 语句的返回值：

- 准备好的单 SELECT 语句，该语句仅返回选择列表中那些列的一行列值
- SPL 函数的准备好的 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句，该语句仅返回一组值

EXECUTE 语句的 INTO 子句有下列语法：

INTO 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 标识系统描述符区域的带引号字符串 | 必须已分配。使用单（’ ）引号 | 引用字符串 |
| *descriptor_var* | 标识系统描述符区域的主变量 | 必须已分配系统描述符区域 | 特定于语言 |
| *indicator_var* | 主变量，如果相应的 *parameter_var* 为 NULL 值，或发生截断，则该变量收到返回码 | 不可为 DATETIME 或 INTERVAL 数据类型 | 特定于语言 |
| *output_var* | 主变量，其内容替代准备好的语句中的问号（?）占位符 | 必须为字符数据类型 | 特定于语言 |
| *sqlda_pointer* | 指向 sqlda 结构的指针，定义值的数据类型和内存位置，来替换准备好的对象中的问号（?）占位符 | 不可以美元符号（$）或冒号（ :）开头。动态的 SQL 需要 sqlda 结构 | DESCRIBE INPUT 语句 |

这非常类似于 USING 子句 的语法。

INTO 子句为更复杂和更长的语法提供简明有效的替代方法。此外，通过将值放入可显示的变量之内，INTO 子句简化和增强检索和显示数据值的能力。例如，如果您使用 INTO 子句，则无需使用游标来从表中检索值。

您可将返回的值存储在输出变量中，或存储在输出 **sqlda** 指针中。

## 对 INTO 子句的限制

如果您执行返回多于一行的准备好的 SELECT 语句，或执行返回多于一组返回值的 SPL 函数的准备好的 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句，则会收到错误消息。此外，如果您准备和声明一个语句然后尝试执行那个语句，则会收到错误消息。

您不可从表列中选择 NULL 值并将那个值放入输出变量内。如果您事先知道表列包含 NULL 值，则在选择该数据之后，检查与该列相关联的指示符变量来确定该值是否为 NULL。

要随同 EXECUTE 语句使用 INTO 子句：

1. 声明 EXECUTE 语句使用的输出变量。
2. 使用 PREPARE 来准备 SELECT 语句或来准备 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句。
3. 使用 EXECUTE 语句，随同 INTO 子句，来执行 SELECT 语句或来执行 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句。

## 以参数替代占位符

在您执行准备好的语句之前，您可执行下列任一项来替代语句中的问号占位符：

- 主变量名称（如果在编译时知道参数的数量和数据类型）
- 标识系统描述符区域的系统描述符
- 作为指向 **sqlda** 结构指针的描述符

后面的章节描述这些指定参数的每一选项。

## 保存主变量或程序变量中的值

如果您知道在运行时提供的返回值的数量及其数据类型，则可在程序中将 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句返回的值定义为主变量。随同 INTO 关键字使用这些主变量，后跟变量的名称。这些变量与返回值相匹配，从左至右一一对应。

你必须为 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）返回的每一个值提供一个变量名称。每一变量的数据类型必须与来自准备好的语句的相应返回值兼容。

## 将值保存在系统描述符区域中

如果您在运行时不知道要提供的返回值的数量或其数据类型，则可将输出值与系统描述符区域相关联。系统描述符区域描述一个或多个值的数据类型和内存位置。

系统描述符区域符合 X/Open 标准。

要指定系统描述符区域为输出值的位置，请使用 EXECUTE 语句的 INTO SQL DESCRIPTOR 子句。每次运行 EXECUTE 语句时，系统描述符区域所描述的值都存储在系统描述符区域中。

下列示例展示如何使用系统描述符区域来执行 GBase 8s ESQL/C 中的准备好的语句：

```
EXEC SQL allocate descriptor 'desc1';
...
sprintf(sel_stmt, "%s %s %s",
    "select fname, lname from customer",
    "where customer_num =",
    cust_num);
```

```
EXEC SQL prepare sel1 from :sel_stmt;
EXEC SQL execute sel1 into sql descriptor 'desc1';
```

COUNT 字段对应于准备好的语句返回的值的数量。COUNT 的值必须小于或等于当以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域时所指定的发生次数的值。

你可以 GET DESCRIPTOR 语句获取域的值，并以 SET DESCRIPTOR 语句设置该值。

要获取更多信息，请参阅 *GBase 8s ESQL/C 程序员手册* 中关于系统描述符区域的讨论。

## 在 sqlda 结构（ESQL/C）中保存值

如果在运行时您不知道返回的输出值的数量或其数据类型，则可从 **sqlda** 结构关联输出值。**sqlda** 结构罗列一个或多个返回值的数据类型和内存位置。要指定 **sqlda** 结构作为返回值的位置，请使用 EXECUTE 语句的 INTO DESCRIPTOR 子句。每次 EXECUTE 语句运行时，数据库服务器将 **sqlda** 结构描述的返回值放在该 **sqlda** 结构内。

下一个示例使用 sqlda 结构来执行准备好的语句：

```
struct sqlda *pointer2;
...
sprintf(sel_stmt, "%s %s %s",
    "select fname, lname from customer",
    "where customer_num =",
    cust_num);
EXEC SQL prepare sel1 from :sel_stmt;
EXEC SQL describe sel1 into pointer2;
EXEC SQL execute sel1 into descriptor pointer2;
```

**sqlda.sqld** 值指定 **sqlvar** 发生次数所描述的输出值的数量。这个数量必须对应于 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句返回的值的数量。

要获取更多信息，请参阅 *GBase 8s ESQL/C 程序员手册* 中关于 **sqlda** 的讨论。

这个示例在 GBase 8s ESQL/C 中随同 EXECUTE 语句使用 INTO 子句：

```
EXEC SQL prepare sel1 from 'select fname, lname from customer
     where customer_num =123';
EXEC SQL execute sel1 into :fname, :lname using :cust_num;
```

下一示例使用 INTO 子句来返回多行数据：

```
EXEC SQL BEGIN DECLARE SECTION;
int customer_num =100;
char fname[25];
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare sel1 from 'select fname from customer
     where customer_num=?';
for ( ;customer_num < 200; customer_num++)
     {
     EXEC SQL execute sel1 into :fname using customer_num;
```

```
printf("Customer number is %d\n", customer_num);
printf("Customer first name is %s\n\n", fname);
}
```

## sqlca 记录和 EXECUTE

在 EXECUTE 语句之后，sqlca 可反映两种结果：

- **sqlca 可反映 EXECUTE 语句内的错误。**

  例如，当在准备好的语句中的 UPDATE …WHERE 语句处理零行时，数据库服务器设置 **sqlca** 为 100。
- **sqlca** 可反映所执行语句的成功或失败。

## 以 EXECUTE 返回的 SQLCODE 值

如果准备好的语句在执行时未能访问任何行，则数据库服务器返回 **SQLCODE** 零 (0) 值。

然而，对于多语句的准备好的对象，如果下列列表中的任何语句都未能访问行，则返回的 SQLCODE 值为 SQLNOTFOUND ( = 100)：

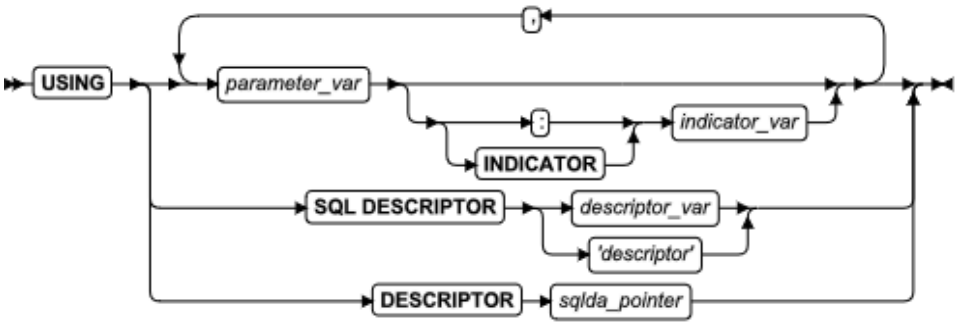
- INSERT INTO ***table*** SELECT … WHERE
- SELECT…WHERE … INTO TEMP
- DELETE … WHERE
- UPDATE … WHERE

在符合 ANSI 的数据库中，如果您准备并执行上述列表中任何语句，且未返回行，则返回的 **SQLCODE** 值为 SQLNOTFOUND ( = 100)。

### USING 子句

使用 USING 子句来指定要替代准备好的语句中的问号（?）占位符的值。在 EXECUTE 语句中提供在准备好的语句中替代问号（?）占位符的值，此操作有时称为 ***参数化*** 准备好的语句。

USING 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *descriptor* | 标识系统描述符区域的带引号字符串 | 必须已经分配了系统描述符区域。 | 引用字符串 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 使用单（ ’ ）引号。 | |
| *descriptor_var* | 标识系统描述符区域的主变量 | 必须已经分配了系统描述符区域 | 特定于语言 |
| *indicator_var* | 如果相应的 *parameter_var* 为 NULL 值，或如果发生截断，则收到返回码的主变量 | 不可为 DATETIME 或 INTERVAL 数据类型 | 特定于语言 |
| *parameter_var* | 主变量，其内容替代准备好的语句中的问号（?）占位符 | 必须为字符数据类型 | 特定于语言 |
| *sqlda_pointer* | 指向 sqlda 结构的指针，定义值的数据类型和内存位置，来替代准备好的对象中的问号（?）占位符 | 不可以美元符号（$）或冒号（ :）开头。动态的 SQL 需要 sqlda 结构 | DESCRIBE INPUT 语句 |

这非常类似于 INTO 子句 的语法。

如果在运行时您直到要提供的参数的数量或其数据类型，则可在程序中将 EXECUTE 语句所需要的参数定义为主变量。

如果在运行时您不知道要提供的参数的数量，则可关联来自系统描述符区域或 **sqlda** 结构的输入值。这两种描述符结构都描述用以替代问号（?）占位符的一个或多个值的数据类型和内存位置。

## 通过主变量或程序变量提供参数

通过以后跟变量名称的 USING 关键字打开游标来向数据库服务器传递参数。这些变量与准备好的语句问号（?）占位符相匹配，从左至右一一对应。 您必须为每一占位符提供一个存储参数变量。每一变量的数据类型必须与准备好的语句所需要的相应值兼容。

下列示例执行在 GBase 8s ESQL/C 中准备好的 UPDATE 语句：

```
stcopy ("update orders set order_date = ?
    where po_num = ?", stm1);
EXEC SQL prepare statement_1 from :stm1;
EXEC SQL execute statement_1 using :order_date, :po_num;
```

## 通过系统描述符提供参数

您可创建描述一个或多个值的数据类型和内存位置的系统描述符区域，然后在 EXECUTE 语句的 USING SQL DESCRIPTOR 子句中指定该描述符 。

每次运行 EXECUTE 语句时，使用系统描述符区域描述的值来替代 PREPARE 语句中的问号（?）占位符。

COUNT 字段对应于准备好的语句中动态参数的数量。COUNT 的值必须小于或等于项描述符的数量，这个数量时当以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域时指定的。

下列示例展示如何使用系统描述符来执行 GBase 8s ESQL/C 中准备好的语句：

```
EXEC SQL execute prep_stmt using sql descriptor 'desc1';
```

### 通过 sqlda 结构（ESQL/C）提供参数

您可在 EXECUTE 语句的 USING DESCRIPTOR 子句中指定 **sqlda** 指针。

每次运行 EXECUTE 语句时，使用描述符结构描述的值来替代 PREPARE 语句中的问号（?）占位符。

**sqlda.sqld** 值指定 **sqlvar** 的发生次数中描述的输入值数量。该数量必须对应于准备好的语句中动态参数的数量。

下列示例显示如何使用 sqlda 结构来执行 GBase 8s ESQL/C 中准备好的语句：

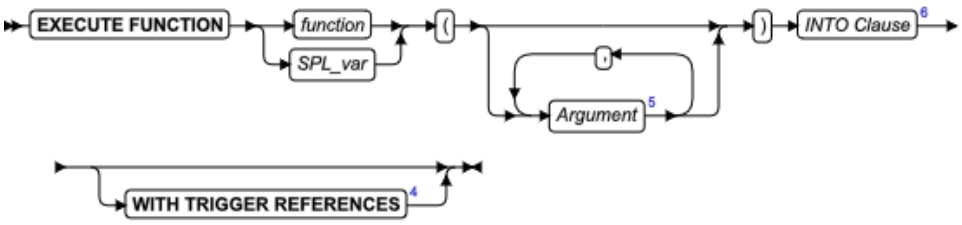
```
EXEC SQL execute prep_stmt using descriptor pointer2;
```

## 2.87 EXECUTE FUNCTION 语句

使用 EXECUTE FUNCTION 语句来调用用户定义的函数或返回值的内建例程。

该语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *function* | 要执行的用户定义函数的名称 | 必须在数据库中注册 | 数据库对象名 |
| *SPL_var* | 包含要执行的 SPL 例程名称的变量 | 必须为包含现有 SPL 函数的非 NULL 名称的 CHAR、VARCHAR、NCHAR 或 NVARCHAR 数据类型 | 标识符 |

**用法**

EXECUTE FUNCTION 语句以参数调用用户定义函数（UDF），并指定结果要返回到哪里。

外部的 C 或 Java™ 语言函数仅返回一个值。

SPL 函数可返回一个或多个值。

您不可使用 EXECUTE FUNCTION 语句来调用不返回值的任何类型的用户定义过程。而是使用 EXECUTE PROCEDURE 或 EXECUTE ROUTINE 语句来执行过程。

您必须对用户定义函数拥有 Execute 权限。

要获取更多信息，请参阅 GRANT 语句。

在支持隐式事务的符合 ANSI/ISO 的数据库中，在缺省情况下，EXECUTE FUNCTION 不开启新的事务。然而，被调用函数内的 SQL 语句可开启新的事务。

## 否定函数及其伴随函数

如果返回 BOOLEAN 值的 UDF 有伴随函数，则执行该函数的任何用户必须对该函数及其伴随函数都有 Execute 权限。例如，如果函数有否定函数，则执行该函数的任何用户都必须对该函数及其否定函数 都有 Execute 权限。此外，伴随函数必须与其否定函数有相同的所有者。

要获取关于如何将 UDF 指定为其否定函数的伴随函数的信息，请参阅 NEGATOR。

## EXECUTE FUNCTION 语句的工作机制

对于要随同 EXECUTE FUNCTION 语句执行的用户定义的函数（UDF），必须存在下列条件：

- 限定的函数名称或函数特征符（带有参数列表的函数名称），在名称空间或数据库内必须是唯一的。
- 在当前数据库中，该函数必须存在。

如果 EXECUTE FUNCTION 指定的参数少于用户自定义的函数预期，则未指定的参数称为***丢失的***。将丢失的参数初始化为它们相应的缺省值，如果已经定义了这些缺省值。在 例程参数列表 中描述为参数指定缺省值的语法。

在下列条件下，EXECUTE FUNCTION 返回错误：

- EXECUTE FUNCTION 指定的参数多于 UDF 预期。
- 丢失一个或多个参数，且没有缺省值。
- 完全限定的函数名称或函数特征符不是唯一的。
- 找不到指定名称的函数或您指定的特征符。
- EXECUTE FUNCTION 尝试调用用户定义的过程。

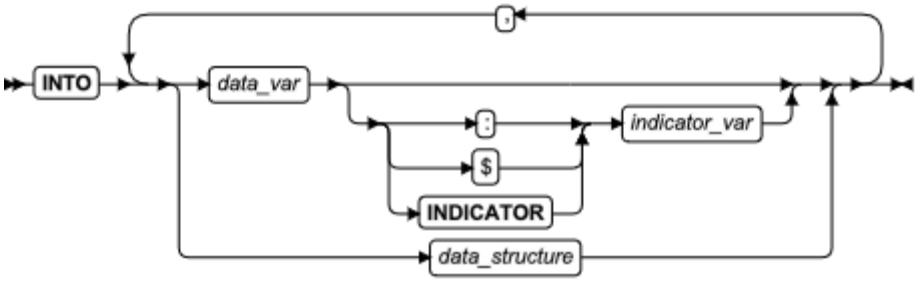
如果在数据库内 ***function*** 名称不唯一，则必须指定足够的 ***parameter_type*** 信息来明确该名称。要获取更多关于调用函数时如何指定参数的信息，请参阅 参数 一节。

在一些 DDL 语句中，外部 UDR 的 ***specific name*** 有效，但在调用该函数的上下文中无效。

如果 GBase 8s 不可解析模糊的函数名称，该名称的特征符不同于仅在未命名 ROW 类型参数中的另一例程的特征符，则返回错误。（当定义该模糊函数时，数据库服务器不可预料到该错误。）

**INTO 子句**

INTO 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_structure* | 声明为主变量的结构 | 结构的单个元素必须与返回值的数据类型兼容 | 特定于语言 |
| *data_var* | 用于接收用户定义的函数返回值的变量 | 请参阅 数据变量。 | 特定于语言 |
| *indicator_var* | 如果相应的 *data_var* 收到 NULL 值，则用于存储返回码的程序变量 | 如果相应的 *data_var* 可能为 NULL，则使用指示符变量 | 特定于语言 |

您必须随同 EXECUTE FUNCTION 包括 INTO 子句来指定接收用户定义的函数返回值的变量。如果该函数返回值多于一个，则这些值以您指定的顺序返回到变量列表中。

如果 EXECUTE FUNCTION 语句独立（即，它不是 DECLARE 语句的一部分且不使用 INTO 子句），则它必须执行非游标函数。 非游标函数仅返回一行值。下列示例展示 GBase 8s ESQL/C 中的 SELECT 语句：

```
EXEC SQL EXECUTE FUNCTION
    cust_num(fname, lname, company_name) INTO :c_num;
```

## 数据变量

如果您在 GBase 8s ESQL/C 程序内发出 EXECUTE FUNCTION 语句，则 ***data_var*** 必须为主变量。在 SPL 例程内，***data_var*** 必须为 SPL 变量。

如果您在 CREATE TRIGGER 内发出 EXECUTE FUNCTION 语句，则 ***data_var*** 必须为触发器表中或另一表中的列名。

## 带有指示符变量（ESQL/C）的 INTO 子句

如果存在从用户定义的函数返回的数据为 NULL 的可能性，则您应使用指示符变量。要获取更多关于指示符变量的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

## 带有游标的 INTO 子句

如果 EXECUTE FUNCTION 调用返回多行值的 UDF，则它必须执行游标函数。游标函数可返回一行或多行值，且必须与要执行的函数游标相关联。

如果 SPL 函数返回多于一行，或返回集合数据类型，则您必须以游标访问这些行或集合元素。

要返回多于一行值，必须将一个外部函数（用 C 或 Java™ 语言编写）定义为迭代函数。要获得关于迭代函数的更多信息，请参阅 *GBase 8s DataBlade API 程序员指南*。

在 SPL 例程中，如果 SELECT 返回多于一行，则您必须使用 FOREACH 语句来逐个地访问这些行。SELECT 语句的 INTO 子句可存储取得的值。要获取更多信息，请参阅 FOREACH。

要返回多行值，SPL 函数必须在其 RETURN 语句中包括 WITH RESUME 关键字。关于如何写 SPL 函数的更多信息，请参阅 *GBase 8s SQL 教程指南*。

在 GBase 8s ESQL/C 程序中，DECLARE 语句可声明函数游标，且 FETCH 语句可从游标逐个地返回行。 您可将 INTO 子句放在 EXECUTE FUNCTION 语句中或放在 FETCH 语句中，但您不可两个都放。下列 GBase 8s ESQL/C 代码示例展示您可使用 INTO 子句的不同方法：

- 使用 EXECUTE FUNCTION 语句中的 INTO 子句：

```
EXEC SQL declare f_curs cursor for
    execute function get_orders(customer_num)
    into :ord_num, :ord_date;
EXEC SQL open f_curs;
while (SQLCODE == 0)
    EXEC SQL fetch f_curs;
EXEC SQL close f_curs;
```

使用 FETCH 语句中的 INTO 子句：

```
EXEC SQL declare f_curs cursor for
    execute function get_orders(customer_num);
EXEC SQL open f_curs;
while (SQLCODE == 0)
    EXEC SQL fetch f_curs into :ord_num, :ord_date;
EXEC SQL close f_curs;
```

## PREPARE ... EXECUTE FUNCTION ... INTO 的备用选择

在 ESQL/C 中，您不可准备包括 INTO 子句的 EXECUTE FUNCTION 语句。然而，对于类似的功能，请遵循这些步骤：

1. 准备不带有 INTO 子句的 EXECUTE FUNCTION 语句。
2. 为准备好的语句声明函数游标。
3. 打开游标。
4. 执行带有 INTO 子句的 FETCH 语句来将返回值放到程序变量内。

作为备用选择，您可执行下列操作：

1. 为未首先准备语句的 EXECUTE FUNCTION 语句声明游标，并在声明游标时在 EXECUTE FUNCTION 中包括 INTO 子句。
2. 打开游标。
3. 不使用 FETCH 语句的 INTO 子句，从游标访存返回的值。

### SPL 函数的动态例程名称规范

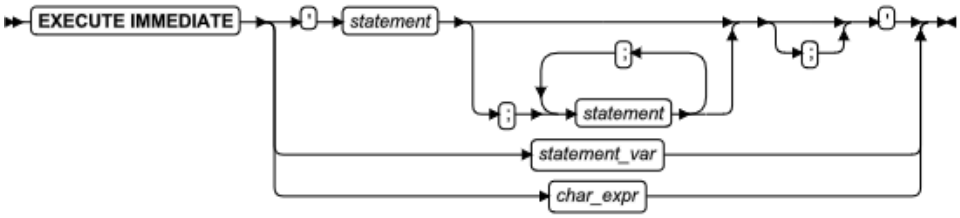
*动态例程名称规范*简化 SPL 函数的编写，该函数调用另一个直到运行时才知道其名字的 SPL 例程。要在 EXECUTE FUNCTION 语句中指定 SPL 例程的名称，而不是罗列 SPL 例程的显示名称，则可使用 SPL 变量来保留例程名称。要了解更多关于如何动态地执行 SPL 函数的信息，请参阅 *GBase 8s SQL 教程指南*。

## 2.88 EXECUTE IMMEDIATE 语句

使用 EXECUTE IMMEDIATE 语句来执行等同于 PREPARE、EXECUTE 和 FREE 语句实现的那些任务，但只作为单个操作。

请随同 GBase 8s ESQL/C 和 SPL 使用该动态 SQL 语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *char_expr* | 取值为字符数据类型的表达式 | 必须取值为 CHAR、LVARCHAR、NCHAR、NVARCHAR 或 VARCHAR 数据类型 | 表达式 |
| *statement* | 有效 SQL 语句的文本 | 请参阅下列的 *statement_var* 的相同章节 | 请参阅本章节。 |
| *statement_var* | 包含 *statement* 的变量或（在 ESQL/C 中）以分号分隔的语句列表 | 必须是先前声明的 CHAR、NCHAR、NVARCHAR 或 VARCHAR （或在 SPL 中，LVARCHAR）类型的变量。请参阅 EXECUTE IMMEDIATE 和限制性语句 和对有效语句的限制。 | Language specific |

**用法**

EXECUTE IMMEDIATE 语句动态地执行在程序执行期间构造的单个 SQL 语句（或在 ESQL/C 例程中，以分号分隔的 SQL 语句列表）。例如，您可从程序输入获取数据库的名称，将 DATABASE 语句构建为程序变量，然后使用 EXECUTE IMMEDIATE 来执行该语句，以打开指定的数据库。

在 ESQL/C 例程内，由变量或引用的字符串指定的语句文本可包括多个 SQL 语句，如果用分号（;）分隔符来分隔连续的语句的话。然而， 在 SPL 例程中，仅可包括一个语句。***statement*** 不可为 SPL 语句，但可为在 EXECUTE IMMEDIATE 和 r 限制性语句 或 对有效语句的限制 章节中未列出的任何 SQL 语句。

如果跟在 IMMEDIATE 关键字后的参数有效，则被分析并执行；然后立即释放所有数据结构和内存资源。除非您使用 EXECUTE IMMEDIATE，不然这些操作需要单独的 PREPARE、EXECUTE 和 FREE 语句。

如果会话环境值（诸如发出 EXECUTE IMMEDIATE 语句的 ESQL/C 或 SPL 例程的 EXTDIRECTIVES、OPTCOMPIND 或 USELASTCOMMITTED 设置）与相应的 ONCONFIG 参数值不同，则会话环境值覆盖它们。

在支持隐式事务的符合 ANSI/ISO 的数据库中，在缺省情况下，EXECUTE IMMEDIATE 语句不开启新的事务。然而，执行指定的 SQL 语句文本可开启新的事务。

## EXECUTE IMMEDIATE 和限制性语句

不可跟在下列 SQL 语句之后执行 EXECUTE IMMEDIATE 语句。

- CLOSE
- CONNECT
- DECLARE
- DISCONNECT
- EXECUTE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- FETCH
- FLUSH
- FREE
- GET DESCRIPTOR
- GET DIAGNOSTICS
- OPEN
- OUTPUT
- PREPARE
- PUT
- SELECT
- SET AUTOFREE

- SET CONNECTION
- SET DEFERRED_PREPARE
- SET DESCRIPTOR
- WHENEVER

对于 EXECUTE PROCEDURE，该限制仅引用与返回一个或多个值的调用。

EXECUTE IMMEDIATE 支持作为语句文本的唯一的 SELECT 语句形式为 `SELECT ... INTO TEMP` ***table***。要了解 SELECT 语句中 INTO TEMP ***table*** 子句的语法，请参阅 INTO table 子句。

此外，ESQL/C 不可使用 EXECUTE IMMEDIATE 语句来执行包含由分号分隔的多个 SQL 语句的文本中的下列语句：

- CLOSE DATABASE
- CREATE DATABASE
- DATABASE
- DROP DATABASE
- SELECT（SELECT INTO TEMP 除外）

EXECUTE IMMEDIATE 语句不可处理包括问号（?）符号作为占位符的 SQL 语句文本。请使用 PREPARE 语句和或者游标或者 EXECUTE 语句来执行动态构建的 SELECT 语句。

（在 SPL 例程中，EXECUTE IMMEDIATE 语句仅可执行单个 SQL 语句。如果紧跟在 IMMEDIATE 关键字后的参数赋值为多个 SQL 语句的列表，或为 NULL 值，或为非有效 SQL 语句的文本，则数据库服务器发出运行时错误。）

## 对有效语句的限制

下列限制应用于那些包含在紧跟在 EXECUTE IMMEDIATE 关键字之后的字符表达式、引用字符串或语句变量中的语句：

- SQL 语句不可包含主语言注释。
- 主语言变量的名称不像在准备好的文本中那样被识别。
- 您可使用的唯一标识符是在当前数据库的系统目录中注册的名称，诸如表名称和列名称。
- 该语句不可引用主变量列表或描述符；不可包含任何问号（?）占位符，这允许随同 PREPARE 语句使用。
- 该文本不可包括任何嵌入的 SQL 语句前缀，诸如美元符号（$）或关键字 EXEC SQL。
- 虽然不是必需的，在语句文本中可包括 SQL 语句结束符（;）。
- 该文本不可包括任何嵌入的 SQL 语句前缀，诸如美元符号（$）或关键字 EXEC SQL。

  EXECUTE IMMEDIATE 不可处理输入主变量，这对集合变量是必需的。请使用 EXECUTE 语句或游标来处理对集合变量的准备好的访问。

## 处理来自 EXECUTE IMMEDIATE 语句的例外

在编译 EXECUTE IMMEDIATE 语句时，如果 GBase 8s ESQL/C 分析器发现语法错误，则发出编译错误，并不产生可执行的 UDR，直到语法正确并编译。如果分析器接受 EXECUTE IMMEDIATE 语法且 UDR 编译成功，但在执行 EXECUTE IMMEDIATE 语句时在调用 UDR 期间发生例外，则数据库服务器在运行时发出错误。WHENEVER 语句可捕获运行时错误，在 UDR 的程序逻辑中一些其他例外处理机制也可捕获。

对于用 SPL 语言写的例程，在运行时为 SQL 表达式赋值，而不是在编译或优化例程时。如果跟在 IMMEDIATE 关键字之后的表达式指定无效的 SQL 语句文本，则 GBase 8s 发出运行时例外，而不是编译错误。在 SPL 例程中的任何运行时错误条件之后，程序控制转到 ON EXCEPTION 语句块（如果定义的话）；否则，UDR 的执行异常终止，并返回给调用上下文一个错误。要获取关于如何在 SPL 例程中处理运行时错误的信息，请参阅 SPL 语句 ON EXCEPTION 的描述。（另请参阅内建 SQL 函数 SQLCODE。）

## EXECUTE IMMEDIATE 语句的示例

下列 ESQL/C 示例展示 GBase 8s ESQL/C 中的 EXECUTE IMMEDIATE 语句。两个示例都是用包含 CREATE DATABASE 语句的主变量。

```
sprintf(cdb_text1, "create database %s", usr_db_id);
EXEC SQL execute immediate :cdb_text1;

sprintf(cdb_text2, "create database %s", usr_db_id2);
EXEC SQL execute immediate :cdb_text2;
```

下一个示例展示 SPL 程序片断，声明本地 SPL 变量并分配给它们两个 DDL 语句文本的部分。然后发出 EXECUTE IMMEDIATE 语句来删除名为 DYN_TAB 的表，指定 SPL 变量中的 DROP TABLE 语句文本。本示例中的第二个 EXECUTE IMMEDIATE 语句创建一个同名的表，在此指定字符表达式中的 CREATE TABLE 语句文本，该表达式将两个 SPL 变量的内容连在一起。

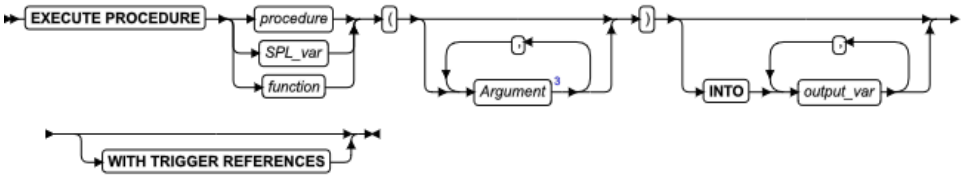
```
CREATE PROCEDURE myproc()
DEFINE COLS      VARCHAR(22);
DEFINE CRTOPER VARCHAR(16);
DEFINE DRPOPER VARCHAR(16);
DEFINE TABNAME VARCHAR(16);
DEFINE QRYSTR   VARCHAR(100);
...
LET CRTOPER = "CREATE TABLE ";
LET DRPOPER = "DROP TABLE ";
LET TABNAME = "DYN_TAB";
LET COLS = "(ID INT, NAME CHAR(20))";
LET QRYSTR = DRPOPER || TABNAME;
EXECUTE IMMEDIATE QRYSTR;

EXECUTE IMMEDIATE CRTOPER || TABNAME || COLS;
```

```
END PROCEDURE;
```

## 2.89 EXECUTE PROCEDURE 语句

使用 EXECUTE PROCEDURE 语句来调用用户定义的过程或内建例程。该语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 要执行的 SPL 例程 | 必须存在 | 数据库对象名 |
| *output_var* | 从 UDR 接收返回值的主变量或程序变量 | 在 CREATE TRIGGER 语句的上下文中，必须包含触发器表中或另一表中的列名称 | 特定于语言 |
| *procedure* | 要执行的用户定义过程 | 必须存在 | 数据库对象名 |
| *SPL_var* | 包含要执行的 SPL 例程名称的变量 | 必须是字符数据类型，包含 SPL 例程的非 NULL 名称。 | 标识符 |

**用法**

EXECUTE PROCEDURE 语句调用用户定义过程并指定其参数。

为了与早期的 GBase 8s 版本相兼容，您可使用 EXECUTE PROCEDURE 语句来执行 CREATE PROCEDURE 语句定义的 SPL 函数。

如果 EXECUTE PROCEDURE 语句返回多行，则必须在 SPL 例程的 FOREACH 循环内处理该结果集，或者通过 ESQL/C 例程的游标另行访问。

在支持隐式事务的符合 ANSI/ISO 数据库中，在缺省情况下，EXECUTE PROCEDURE 不开启新的事务。然而，在被调用过程内的 SQL 语句可开启新的事务。

### 错误的原因

EXECUTE PROCEDURE 在下列条件下返回错误。

- 它的参数比被调用过程预期的多。

- 丢失一个或多个参数，且没有缺省值。
- 完全限定过程名称或例程特征符不唯一。
- 找不到指定名称或特征符的过程。

如果在数据库内 ***procedure*** 名称不唯一，则必须指定足够的 ***parameter_type*** 信息来明确该名称。请参阅 参数 获取在调用过程时如何指定参数的附加信息。（在 GBase 8s 中，外部 UDR 的 ***specific name*** 在 DDL 语句中有效，但在调用该过程的上下文中无效。）

## 使用 INTO 子句

使用 INTO 子句来指定存储 SPL 函数返回值的存储位置。

如果 SPL 函数返回多个值，则这些值以您指定的顺序返回到变量列表内。如果 SPL 函数返回多行或集合数据类型，则必须以游标访问这些行或集合元素。

您不可准备有 INTO 子句的 EXECUTE PROCEDURE 语句。要获取更多信息，请参阅 PREPARE ... EXECUTE FUNCTION ... INTO 的备用选择。

## WITH TRIGGER REFERENCES 关键字

在您使用 EXECUTE PROCEDURE 语句来调用触发器过程时，必须包括 WITH TRIGGER REFERENCES 关键字。

触发器过程是一个 SPL 例程，EXECUTE PROCEDURE 仅可从触发器定义的 Action 子句的 FOR EACH ROW 部分来调用该例程。该 REFERENCING 子句声明关联变量的名称，该过程可使用这些变量来在触发器事件发生时引用行中的 ***old*** 列，或在触发器修改了行之后引用列的 ***new*** 值。FOR 子句指定在其上定义触发器的表或视图。

## 调用触发器过程的示例

```
CREATE TABLE tab1 (col1 INT,col2 INT);
CREATE TABLE tab2 (col1 INT);
CREATE TABLE temptab1
    (old_col1 INTt, new_col1 INT, old_col2 INT, new_col2 INT);


        /* 在本示例中，从 INSERT 触发器调用下列过程。
        */
CREATE PROCEDURE proc1()
REFERENCING OLD AS o NEW AS n FOR tab1;

IF (INSERTING) THEN    -- INSERTING Boolean operator
        LET n.col1 = n.col1 + 1;     -- You can modify new values.
        INSERT INTO temptab1 VALUES(0,n.col1,1,n.col2);
END IF

IF (UPDATING) THEN    -- UPDATING Boolean operator
```

```
        -- you can access relevant old and new values.
        INSERT INTO temptab1 values(o.col1,n.col1,o.col2,n.col2);
END IF

if (SELECTING) THEN    -- SELECTING Boolean operator
        -- you can access relevant old values.
        INSERT INTO temptab1 VALUES(o.col1,0,o.col2,0);
END IF

if (DELETING) THEN     -- DELETING Boolean operator
        DELETE FROM temptab1 WHERE temptab1.col1 = o.col1;
END IF

END PROCEDURE;
```

该示例说明触发操作可不同于触发事件的 DML 操作。虽然该过程在 Insert 触发器调用它时插入一行，并在 Delete 触发器调用它时删除一行，但如果由 Select 触发器或由 Update 触发器调用它，则还执行 INSERT 操作。

本示例中的 **proc1( )** 触发器过程使用仅在触发器例程中才有效的 Boolean 条件运算符。仅当从 INSERT 触发器的 FOR EACH ROW 操作调用该过程时，INSERTING 运算符才返回真值。该过程还可从其触发器事件为 UPDATE、SELECT 或 DELETE 的其他触发器调用，因为如果在相应的触发事件类型触发的操作中调用该过程，则 UPDATING、SELECTING 和 DELETING 操作符返回真值（t）。

下列语句在 tab1 上定义 Insert 触发器，从 FOR EACH ROW 部分调用 proc1( ) 作为其触发的操作，并执行激活该触发器的 INSERT 操作：

```
CREATE TRIGGER ins_trig_tab1 INSERT ON tab1 REFERENCING NEW AS post
        FOR EACH ROW(EXECUTE PROCEDURE proc1() WITH TRIGGER REFERENCES);
```

请注意该触发器的 REFERENCING 子句为 NEW 子句声明一相关名称，不同于触发器过程声明的相关名称。这些名称无需匹配，因为在触发器过程中声明的相关名称是以那个过程作为其引用的作用域。下列语句激活 ins_trig_tab1 触发器，执行 proc1( ) 过程。

```
INSERT INTO tab1 VALUES (111,222);
```

由于触发器过程将 col1 的新值加 1，因此插入的值为 (112, 222)，而不是触发事件指定的值。

## SPL 过程的动态例程名称规范

***动态例程名称规范***简化 SPL 例程的编写，该例程调用另一个直到运行时才能知道其名称的 SPL 例程。

要指定 EXECUTE PROCEDURE 语句中的 SPL 例程的名称，而不是罗列 SPL 例程的显式名称，您可使用 SPL 变量来保留例程名称。

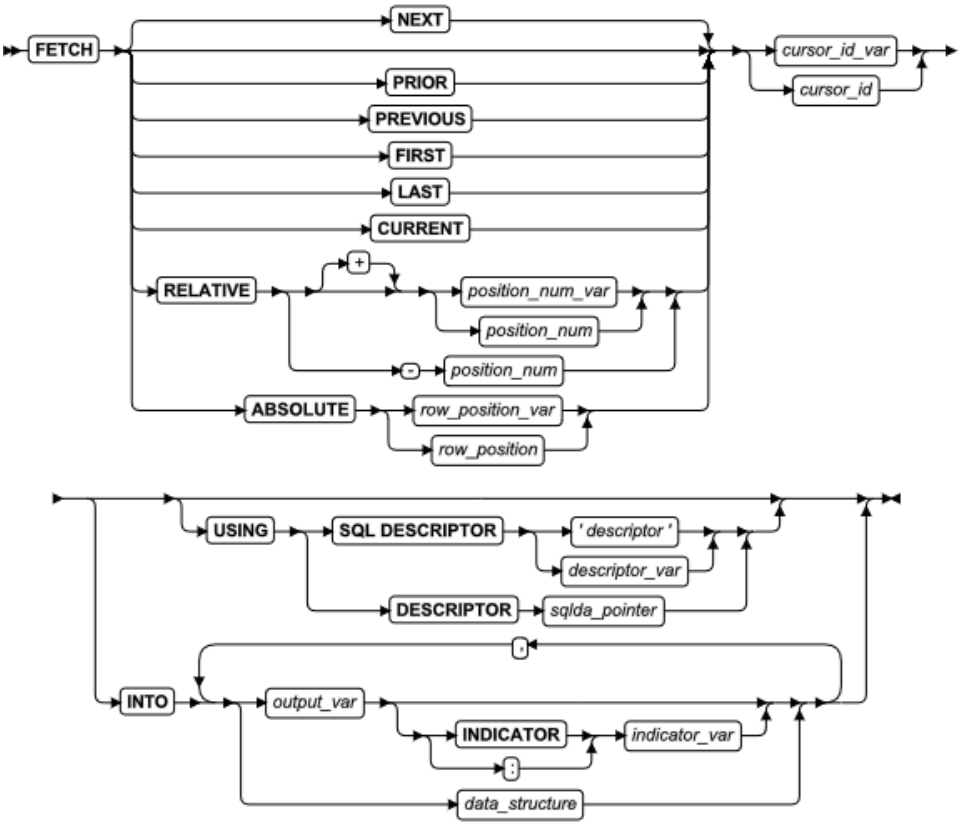
如果 SPL 变量命名返回值的 SPL 例程，则包括 EXECUTE PROCEDURE 的 INTO 子句来指定 ***receiving variable***（或多个变量）以保留 SPL 函数返回的一个值（或多个值）。要获取更过关于如何动态地执行 SPL 过程的信息，请参阅 ***《GBase 8s SQL 指南：教程》***。

## 2.90 FETCH 语句

使用 FETCH 语句来将游标移到活动的集合中的新行，并从内存检索该行值。

请随同 GBase 8s ESQL/C 并随同 SPL 使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 要检索行的游标 | 必须打开 | 标识符 |
| *cursor_id_var* | 存储 *cursor_id* 的主变量 | 必须为字符数据类型 | 特定于语言 |
| *data_structure* | 作为主变量的结构 | 必须存储访存的值 | 特定于语言 |
| *descriptor* | 系统描述符区域 | 必须已分配 | 引用字符串 |
| *descriptor_var* | 存储 *descriptor* 的 | 必须已分配 | 特定于语 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 主变量 | | 言 |
| *indicator_var* | 返回码的主变量，如果 *output_var* 可为 NULL 值 | 请参阅 使用指示符变量。 | 特定于语言 |
| *output_var* | 访存的值的主变量 | 必须存储来自行的值 | 特定于语言 |
| *position_num* | 相对于当前行的位置 | 值 0 访存当前行 | 精确数值 |
| *position_num_var* | 主变量（= *position_num*） | 值 0 访存当前行 | 特定于语言 |
| *row_position* | 活动的集合中的序数位置 | 必须为大于 1 的整数 | 精确数值 |
| *row_position_var* | 主变量（= *row_position*） | 必须为 1 或更大 | 特定于语言 |
| *sqlda_pointer* | 指向 sqlda 结构的指针 | 不可以 $ 开头，也不可以 : 开头 | 请参阅 ESQL/C。 |

**用法**

除非另有注明，后续章节都描述在 GBase 8s ESQL/C 例程中如何使用 FETCH 语句。要获取更多关于在 SPL 例程中 FETCH 语句的更严格语法和语义的信息，请参阅 从 SPL 例程中的动态游标访存。

数据库服务器如何创建、存储和访存活动的行集合的成员，取决于是否将游标声明为顺序游标，或声明为滚动游标。FETCH 语句可在 SPL 例程中引用的所有游标都是顺序游标。

在 X/Open 模式下，如果指定游标方向值（诸如 NEXT 或 RELATIVE），则发出警告消息，指出语句不符合 X/Open 标准。

## 使用顺序游标的 FETCH

顺序游标仅可从活动的集合按顺序访存下一行。唯一可用的选项为缺省选项 NEXT。每次打开表时顺序游标仅可通读表一次。下列 GBase 8s ESQL/C 示例说明使用顺序游标的 FETCH 语句：

```
EXEC SQL FETCH seq_curs INTO :fname, :lname;
EXEC SQL FETCH NEXT seq_curs INTO :fname, :lname;
```

在程序打开顺序游标时，数据库服务器处理对第一行数据的定位或构造点的查询。数据库服务器的目标是尽可能少地占用资源。

由于顺序游标仅可检索下一行，因此数据库服务器可频繁地创建活动的集合，一次一行。

对于每一 FETCH 操作，数据库服务器返回当前行的内容，并定位到下一行。如果数据库服务器必须创建整个活动的集合来确定哪一行为下一行，则这种一次一行的策略不可行。（可能是 SELECT 语句包括 ORDER 子句的那种情况）。

## 使用滚动游标的 FETCH

这些 GBase 8s ESQL/C 示例说明使用滚动游标的 FETCH 语句：

```
EXEC SQL fetch previous q_curs into :orders;
EXEC SQL fetch last q_curs into :orders;
EXEC SQL fetch relative -10 q_curs into :orders;
printf("Which row? ");
scanf("
EXEC SQL fetch absolute :row_num q_curs into :orders;
```

滚动游标可访存活动的集合中的任意行，或者通过指定绝对行位置，或者通过指定相对偏移量。使用下列游标位置选项来指定您想要检索的特定行。

| 关键字 | 作用 |
|---|---|
| NEXT | 检索活动的集合中的下一行 |
| PREVIOUS | 检索活动的集合中的前一行 |
| PRIOR | 检索活动的集合中的前一行（与 PREVIOUS 同义。） |
| FIRST | 检索活动的集合中的第一行 |
| LAST | 检索活动的集合中的最后一行 |
| CURRENT | 检索活动的集合中的当前行（与前面的 FETCH 语句从滚动游标返回的行相同） |
| RELATIVE | 检索相对于活动的集合中当前游标位置的第 ***n*** 行，其中 ***position_num***（或 ***position_num_var***） 提供 ***n***。负值指示当前游标位置之前的第 ***n*** 行。如果 ***position_num*** = 0，则访存当前行。 |
| ABSOLUTE | 检索活动的集合中的第 ***n*** 行，其中 ***row_position_var*** （或 ***row_position***）= ***n***。绝对行位置从 1 开始计数。 |

**提示：** 请不要将行位置值与 rowid 值相混淆。rowid 值是基于在其表中的位置，并在重建表之前保持有效。行位置值（ABSOLUTE 关键字检索到的值）是该行在游标的活动的集合中的相对位置；下一次打开游标时，可能选择不同的行。

## 数据库服务器如何实现滚动游标

由于数据库服务器不可预测程序下一次会请求哪一行，因此数据库服务器必须保留活动的集合中的所有行，直到该滚动游标关闭为止。在滚动游标打开时，数据库服务器实现活动的集合作为临时表，虽然它不可能立即植入此表。

第一次访存行时，数据库服务器将它复制到临时表内并将它返回到程序。

在第二次访存行时，可以从临时表进行。这种方案使用最少的资源，以防程序在访存所有行之前放弃查询。从不被访存的那些行通常不从数据库复制，或被保存在临时表中。

## 指定值在内存中的返回位置

来自查询的选择列表或者执行用户定义的函数的每一值必须被返回到内存位置内。您可以下列方式之一指定这些目的地：

- 使用 SELECT 语句的 INTO 子句。
- 使用 EXECUTE 函数（或 EXECUTE PROCEDURE）语句的 INTO 子句。
- 使用 FETCH 语句的 INTO 子句。
- 使用系统描述符区域。
- 使用 **sqlda** 结构。

## 使用 INTO 子句

如果您将 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句与函数游标相关联，则该语句可包含 INTO 子句来指定变量来接收返回的值。仅当写 SELECT、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句作为游标声明的一部分时，才可使用这种方法；请参阅 DECLARE 语句。在这种情况下，FETCH 语句不可包含 INTO 子句。

下列示例使用 SELECT 语句的 INTO 子句来指定 GBase 8s ESQL/C 中的程序变量：

```
EXEC SQL declare ord_date cursor for
    select order_num, order_date, po_num
        into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

如果您准备 SELECT 语句，则 SELECT ***不可***包括 INTO 子句，因此必须使用 FETCH 语句的 INTO 子句。

在您动态地创建 SELECT 语句时，不可使用 INTO 子句，因为您不可在准备好的语句中命名主变量。

如果您确定投影列表中值的数量和数据类型，则可使用 FETCH 语句中的 INTO 子句。然而，如果用户输入生成了查询，则可能无法确定被选择的值的数量和数据类型。在这种情况下，您必须或者使用系统描述符，或者使用指向 **sqlda** 结构的指针。

## 使用指示符变量

如果返回的数据可能为空，则请使用指示符变量。

***indicator_var*** 参数是可选的，但如果可能存在 ***output_var*** 的值为 NULL 的情况，则使用指示符变量。

如果您指定不带 INDICATOR 关键字的指示符变量，则不可在 ***output_var*** 与 ***indicator_var*** 之间加空格。

要获取更多关于在 ***indicator_var*** 之前加前缀的规则的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

主变量不可为 DATETIME 或 INTERVAL 数据类型。

## 在需要 FETCH 的 INTO 子句时

在 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）省略 INTO 子句时，在访存行时必须指定数据目标。

例如，要动态地执行 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句，SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）不可在 PREPARE 语句中包括其 INTO 子句。因此，FETCH 语句必须包括 INTO 子句来将数据检索到一组变量内。这种方法让您在不同的内存位置中存储不同的行。

仅可使用 FETCH 语句中的 INTO 子句来访存到程序数组元素内。如果您使用程序数组，则必须罗列数据名称和在 *data_structure* 中数据的特定元素。在您声明游标时，请不要在 SQL 语句内引用数据元素。

**提示：** 如果您确定在 Projection 子句的选择列表中值的数量和数据类型，则可使用 FETCH 语句中的 INTO 子句。

在下列 GBase 8s ESQL/C 示例中，一系列完整的行被访存到程序数组内。每一 FETCH 语句的 INTO 子句指定数组元素和数据名称：

```
EXEC SQL BEGIN DECLARE SECTION;
    char wanted_state[2];
    short int row_count = 0;
    struct customer_t{
    {
        int       c_no;
        char      fname[15];
        char      lname[15];
    } cust_rec[100];
EXEC SQL END DECLARE SECTION;

main()
{
    EXEC SQL connect to'stores_demo';
    printf("Enter 2-letter state code: ");
    scanf ("%s", wanted_state);
    EXEC SQL declare cust cursor for
        select * from customer where state = :wanted_state;
    EXEC SQL open cust;
    EXEC SQL fetch cust into :cust_rec[row_count];
    while (SQLCODE == 0)
    {
        printf("\n%s %s", cust_rec[row_count].fname,
            cust_rec[row_count].lname);
```

```
        row_count++;
        EXEC SQL fetch cust into :cust_rec[row_count];
    }
    printf ("\n");
    EXEC SQL close cust;
    EXEC SQL free cust;
}
```

## 使用系统描述符区域（X/Open）

当您不知道在运行时 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句返回的返回值的数量或其数据类型时，可使用系统描述符区域来存储输出值。系统描述符区域描述一个或多个返回值的数据类型和内存位置，并符合 X/Open 标准。

关键字 USING SQL DESCRIPTOR 将系统描述符区域的名称引入您访存行的内容或用户定义的函数的返回值内。然后您可使用 GET DESCRIPTOR 语句来将 FETCH 语句返回的值从系统描述符区域传送到主变量内。

下列示例展示有效的 FETCH…USING SQL DESCRIPTOR 语句：

```
EXEC SQL allocate descriptor 'desc';
    ...
EXEC SQL declare selcurs cursor for
    select * from customer where state = 'CA';
EXEC SQL describe selcurs using sql descriptor 'desc';
EXEC SQL open selcurs;
while (1)
    {
    EXEC SQL fetch selcurs using sql descriptor 'desc';
```

您还可使用 **sqlda** 结构来动态地提供参数。

## 使用 sqlda 结构

在您不知道 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句返回的值的数量或其数据类型时，可使用指向 **sqlda** 结构的指针来存储输出值。

此结构包含为一个选择的值指定数据类型和内存位置的数据描述符。关键字 USING DESCRIPTOR 引入指向 sqlda 结构的指针。

**提示：** 如果您知道在选择列表中所有值的数量和数据类型，则可使用 FETCH 语句中的 INTO 子句。要获取更多信息，请参阅 在需要 FETCH 的 INTO 子句时。

要指定 sqlda 结构作为参数位置：

1. 声明 **sqlda** 指针变量。
2. 使用 DESCRIBE 语句来填充 **sqlda** 结构。
3. 分配内存来保留数据值。

4. 使用 FETCH 的 USING DESCRIPTOR 子句来指定 **sqlda** 结构作为您访存返回值的目标位置。

下列示例展示 FETCH USING DESCRIPTOR 语句：

```
struct sqlda *sqlda_ptr;
...
EXEC SQL declare selcurs2 cursor for
    select * from customer where state = 'CA';
EXEC SQL describe selcurs2 into sqlda_ptr;
...
EXEC SQL open selcurs2;
while (1)
    {
    EXEC SQL fetch selcurs2 using descriptor sqlda_ptr;
    ...
```

**sqld** 值指定 **sqlda** 结构的 **sqlvar** 结构的出现个数中描述的输出值的数量。此数量必须对应于从准备好的语句返回的值的数量。

## 为更新而访存行

FETCH 语句通常不锁定已访存的行。这样，您的程序收到该访存的行之后，另一进程可立即修改（更新或删除）它。在下列情况下，锁定访存的行：

- 在您将隔离级别设置为 Repeatable Read 时，以读锁锁定访存的每一行，直到当前会话结束为止。其他程序还可读锁定的行。
- 在您将隔离级别设置为 Cursor Stability 时，锁定当前行。
- 在符合 ANSI 的数据库中，Repeatable Read 为缺省的隔离级别；您可将它设置为其他值。
- 当您通过更新游标（声明 FOR UPDATE 的游标）访存时，以可提升锁锁定访存的每一行。其他程序可读取锁定的行，但其他程序不可放置可提升锁或写锁；因此，如果另一用户试图使用 UPDATE 或 DELETE 语句的 WHERE CURRENT OF 子句修改该行，则它保持不变。

当您修改行时，锁升级为写锁并保持直到游标关闭或事务结束为止。如果您不修改行，则数据库服务器的行为取决于已设置的隔离级别。一旦访存另一行，数据库服务器即释放未更改的行的锁，除非您正在使用 Repeatable Read 隔离。（请参阅 SET ISOLATION 语句。）

**重要：** 在没有使用 Repeatable Read 隔离或 Repeatable Read 隔离不可用时，您可在附加的行上保持锁定。当您的程序正在读其他行时，以未更改的数据更新该行来保持对它的锁定。您必须在应用的上下文中评估此技术对性能的影响，且必须了解增加死锁的可能。

在您使用显式事务时，请确保在单个事务内既访存也修改行；即，FETCH 和后续的 UPDATE 或 DELETE 语句都必须在 BEGIN WORK 语句与下一个 COMMIT WORK 语句之间。

## 从集合游标访存

集合游标允许您访问 GBase 8s ESQL/C 集合变量的个别元素。要声明集合游标，请使用 DECLARE 语句，并将集合派生表段包括在与游标关联的 SELECT 语句中。在您以 OPEN 语句打开集合游标之后，该游标允许您来访问集合变量的元素。

要从集合游标每次访存一个元素，请使用 FETCH 语句和 INTO 子句。FETCH 语句标识以集合变量关联的集合游标。INTO 子句标识保留从集合游标访存的元素值的主变量。INTO 子句中的主变量的数据类型必须与集合的元素类型相配。

假设您有一名为 children 的表，具有下列结构：

```
CREATE TABLE children
(
    age             SMALLINT,
    name              VARCHAR(30),
    fav_colors                SET(VARCHAR(20) NOT NULL),
)
```

下列 GBase 8s ESQL/C 代码段展示如何从 child_colors 集合变量访存元素：

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection child_colors;
    varchar one_favorite[21];
    char child_name[31] = "marybeth";
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate collection :child_colors;
/* Get structure of fav_colors column for untyped
 * child_colors collection variable */
EXEC SQL select fav_colors into :child_colors
    from children
    where name = :child_name;
/* Declare select cursor for child_colors collection
 * variable */
EXEC SQL declare colors_curs cursor for
    select * from table(:child_colors);
EXEC SQL open colors_curs;
do
{
    EXEC SQL fetch colors_curs into :one_favorite;
    ...
} while (SQLCODE == 0)
EXEC SQL close colors_curs;
EXEC SQL free colors_curs;
EXEC SQL deallocate collection :child_colors;
```

在访存集合元素之后，您可以 UPDATA 或 DELETE 语句修改该元素。要获取更多信息，请参阅本文档中的 UPDATE 和 DELETE 语句。您还可以 INSERT 语句将新元素插入到集合变量内。要获取更多信息，请参阅 INSERT 语句。

## 检查 FETCH 的结果

您可使用 **SQLSTATE** 变量来检查每一 FETCH 语句的结果。数据库服务器在每一 SQL 语句之后设置 **SQLSTATE** 变量。如果成功地返回一行，则 **SQLSTATE** 变量包含值 00000。如果未找到

行，则数据库服务器设置 **SQLSTATE** 代码为 02000，表明 未找到数据，且当前行不变。下列条件将 **SQLSTATE** 代码设置为 02000，表明 未找到数据：

- 活动的集合未包含行。
- 在游标指向活动的集合中的最后一行或越过该行时，发出 FETCH NEXT 语句。
- 在游标指向活动的集合中的第一行时，发出 FETCH PRIOR 或 FETCH PREVIOUS 语句。
- 在活动的集合中不存在第 ***n*** 行时，发出 FETCH RELATIVE ***n*** 语句。
- 在活动的集合中不存在第 ***n*** 行时，发出 FETCH ABSOLUTE ***n*** 语句。

数据库服务器从系统诊断区域的 **RETURNED_SQLSTATE** 域复制 **SQLSTATE** 代码。 GBase 8s 的客户机/服务器通信协议，诸如 SQLI 和 DRDA®，支持 **SQLSTATE** 代码值。要获取这些代码的列表，并获取关于如何获得消息文本的消息，请参阅 使用 SQLSTATE 错误状态代码。您可使用 GET DIAGNOSTICS 语句来直接地检验 **RETURNED_SQLSTATE** 域。系统诊断区域还可包含附加的错误信息。

您还可使用 SQL 通信区域（**sqlca**）的 **SQLCODE** 变量来确定相同的结果。

### 从 SPL 例程中的动态游标访存

使用 SPL 例程中的 FETCH 语句来检索指定的动态游标活动的集合的下一行，检索到一个在同一 SPL 例程中声明的 SPL 变量的有序列表内。

### 语法

SPL 例程中 FETCH 语句的语法是在 GBase 8t ESQL/C 例程中 FETCH 支持的语法的子集。


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 动态游标的名称 | 必须是打开的且已经在同一 SPL 例程中声明 | 标识符 |
| *output_var* | 存储来自该行的访存的值的 SPL 变量 | 必须在调用上下文中已经本地地或全局地声明，且必须为与访存的列值相兼容的数据类型 | 标识符 |

恰如在 ESQL/C 例程中那样，输出变量的列表必须与列值的数量、顺序和数据类型相一致，这些列值是 SQL 语句关联的由特定的游标返回的那些行。

所有 SPL 游标都是顺序游标。您的 UDR 必须包括检测游标的活动的集合到头的逻辑，因为在 SPL 中 NOTFOUND 条件不会自动地产生例外。

内建的 **SQLCODE** 函数，仅可从 SPL 例程调用的函数，可返回 FETCH 操作的状态代码。

对引用顺序选择游标或功能游标的 FETCH 语句的其他 ESQL/C 限制，也适用于 SPL 中的 FETCH 操作。

SPL 例程中的 FETCH 语句不支持下列 ESQL/C 特性：

- 指定为主变量的游标名称
- 位置规范或位置关键字（需要滚动游标）
- 随同描述符或随同 **sqlda** 指针的 USING 子句。

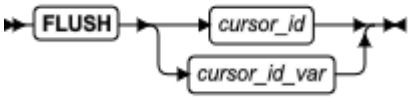
在 SPL 语言中，不需要指示符变量。如果 FETCH 操作收到 NULL 值，则将收到访存的值的 SPL 变量设置为 NULL。

FETCH 语句仅可引用 DECLARE 语句定义的动态游标。***cursor_id*** 不可指定 SPL 的 FOREACH 语句声明的直接游标的名称。

## 2.91 FLUSH 语句

使用 FLUSH 语句来将 PUT 语句缓冲的行强制写到数据库。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 游标名称 | 必须已经声明 | 标识符 |
| *cursor_id_var* | 保留 *cursor_id* 值的主变量 | 必须为字符数据类型 | 特定于语言 |

**用法**

随同 GBase 8s ESQL/C 使用此语句，这是对 SQL 的 ANSI/ISO 标准的扩展。

PUT 语句添加行到缓冲区，在缓冲区变满时，将缓冲区的内容写到数据库。在缓冲区未满时，使用 FLUSH 语句来强制插入。

如果程序终止而未关闭该游标，则缓冲区保持为未刷新。从上一次刷新丢失起，将行放入缓冲区内。不要预期程序结束会自动地关闭游标并刷新缓冲区。下列示例展示操作名为 icurs 的游标的 FLUSH 语句：

```
FLUSH icurs
```

**示例**

下列示例假设名为 *next_cust* 的函数返回有关新客户的信息，或返回空数据表示输入结束：

```
EXEC SQL BEGIN WORK;
EXEC SQL OPEN new_custs;
```

```
while(SQLCODE == 0)
{
      next_cust();
      if(the_company == NULL)
            break;

      EXEC SQL PUT new_custs;
}

if(SQLCODE == 0)      /* if no problem with PUT */
{
      EXEC SQL FLUSH new_custs;
      /* write any rows left */

      if(SQLCODE == 0)      /* if no problem with FLUSH */
            EXEC SQL COMMIT WORK;    /* commit changes */
}
else
      EXEC SQL ROLLBACK WORK; /* else undo changes */
```

此示例中的代码重复地调用 *next_cust*。在返回非空数据时，PUT 语句将返回值发送到行缓冲区。在缓冲区填满时，自动地将缓冲区包含的那些行发送到数据库服务器。在 *next_cust* 不再有数据返回时，循环正常结束。

## 检查 FLUSH 语句时出错

SQL 通信区域（**sqlca**）结构包含关于每一 FLUSH 语句的成功信息以及成功地插入的行数。每一 FLUSH 语句的结果在 **sqlca** 的这些字段描述：**sqlca.sqlcode**、**SQLCODE** 和 **sqlca.sqlerrd[2]**。

在您以插入游标使用数据缓冲时，直到刷新该缓冲区才会发现错误。例如，仅在刷新缓冲区时，才能发现与列的预定数据类型不兼容的输入值。在发现错误时，缓冲区中位于错误之后的任何行都*不会*插入；它们从内存中丢失。

如果未发生错误，则 SQLCODE 域或设置为错误代码或设置为零（0）。SQLERRD 数组的第三个元素设置为成功地插入到数据库内的行数：

- 如果一个行块成功地插入到数据库内，则 **SQLCODE** 设置为零（0），**SQLERRD** 设置为行数。
- 如果在 FLUSH 语句插入一个行块时发生错误，则 **SQLCODE** 显示是哪个错误，**SQLERRD** 包含成功地插入的行数。（从缓冲区废弃未插入的行。）

**提示：** 在您遇到 SQLCODE 错误时，还存在相应的 SQLSTATE 错误。 GBase 8s 的客户机/服务器通信协议，诸如 SQLI 和 DRDA®，支持 SQLSTATE 代码值。要获取这些代码的列表，以及关于如何取得消息文本的信息，请参阅 使用 SQLSTATE 错误状态代码。

**要对实际插入到数据库内的行数以及尚未插入的行数计数**

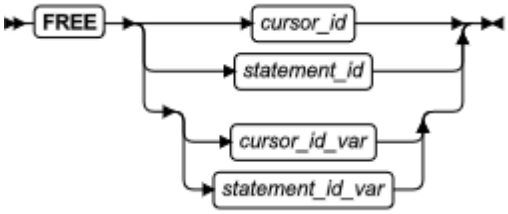
.准备两个整数变量，例如，**total** 和 **pending**。

1. 在游标打开时，设置两个变量为 0。
2. 每次执行 PUT 语句时，增大 **total** 和 **pending**。
3. 只要执行 FLUSH 语句时，或关闭游标时，从 **pending** 抽取 **SQLERRD** 数组的第三个字段。

## 2.92 FREE 语句

使用 FREE 语句来释放那些分配给准备好的语句或分配给游标的资源。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 游标的名称 | 必须已经声明 | 标识符 |
| *cursor_id_var* | 持有 cursor_id 值的主变量 | 必须为字符数据类型 | 特定于语言 |
| *statement_id* | 准备好的 SQL 语句的标识符 | 必须在先前的 PREPARE 语句中定义 | PREPARE 语句 |
| *statement_id_var* | 存储准备好的对象名称的主变量 | 必须声明为字符数据类型。 | PREPARE 语句 |

**用法**

随同 GBase 8s ESQL/C 或随同 SPL 使用此语句，这是对 SQL 的 ANSI/ISO 标准的扩展。

FREE 释放为准备好的语句或为游标分配的（ESQL/C 的）数据库服务器和应用开发工具。

如果您为准备好的语句声明游标，则 FREE ***statement_id***（或 ***statement_id_var***）仅释放应用开发工具中的资源；游标仍可使用。仅当释放游标时，才释放数据库服务器中的资源。

如果您准备了语句（但没有为它声明游标），则 FREE ***statement_id***（或 FREE ***statement_id_var***）释放在应用开发工具和数据库服务器中的资源。

在您释放语句之后，不可执行它或为它声明游标，直到您再次准备它。

下列 GBase 8s ESQL/C 示例展示那些用于释放隐式准备好的语句的语句序列：

```
EXEC SQL prepare sel_stmt from 'select * from orders';
```

```
...
EXEC SQL free sel_stmt;
```

下列 GBase 8s ESQL/C 示例展示那些用于释放显式准备好的语句的资源的语句序列。此示例中的第一个 FREE 语句释放游标。此示例中的第二个 FREE 语句释放准备好的语句。

```
sprintf(demoselect, "%s %s",
    "select * from customer ",
    "where customer_num between 100 and 200");
EXEC SQL prepare sel_stmt from :demoselect;
EXEC SQL declare sel_curs cursor for sel_stmt;
EXEC SQL open sel_curs;
...
EXEC SQL close sel_curs;
EXEC SQL free sel_curs;
EXEC SQL free sel_stmt;
```

如果您为准备好的语句声明游标，则释放游标仅释放数据库服务器中的资源。要为该语句释放应用开发工具中的资源，请使用 FREE ***statement_id***（或 FREE ***statement_id_var***）。如果未为准备好的语句声明游标，则释放它会释放应用开发工具和数据库服务器中的资源。要查看释放游标的 FREE 语句的 ESQL/C 示例，请参阅前面的示例。

释放游标之后，不可打开它直到再次声明为止。游标应在其被释放之前明确地关闭它。

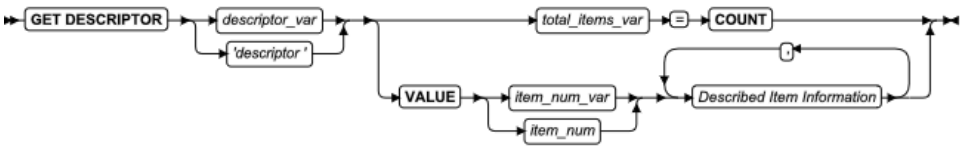
在 SPL 例程执行完毕时，数据库服务器自动地释放例程中通过 PREPARE 或 DECLARE 语句已经分配给游标或准备好的语句的任何资源，如果这些资源尚未由 FREE 语句释放的话。

SPL 例程中的 FREE 语句不可引用 SPL 的 FOREACH 语句可声明的直接游标的 ***cursor_id***。

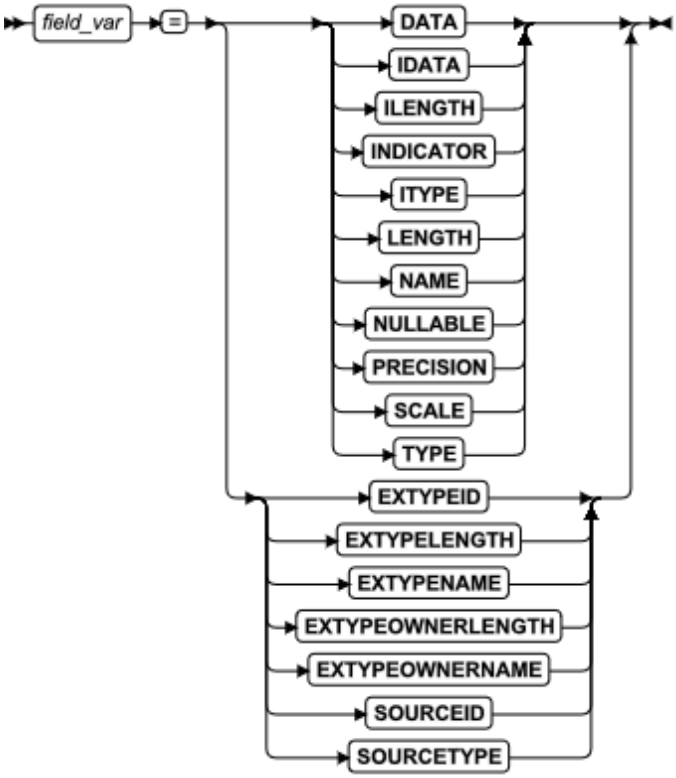
## 2.93 GET DESCRIPTOR 语句

使用 GET DESCRIPTOR 语句来从系统描述符区域读取

**语法**




描述的项信息




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 标识系统描述符区域（SDA）的带引号字符串 | 必须已经分配了系统描述符区域 | 引用字符串 |
| *descriptor_var* | 存储 *descriptor* 值的变量 | 与 *descriptor* 相同的限制 | 特定于语言 |
| *field_var* | 从 SDA 接收域内容的主变量 | 类型必须为可接收指定的 SDA 域的值 | 特定于语言 |
| *item_num* | 在 SDA 中描述的项的无符号序数 | 0 ≤ *item_num* ≤（在 SDA 中的项描述符的数量） | 精确数值 |
| *item_num_ var* | 存储 *item_num* 的主变量 | 必须为整数数据类型 | 特定于语言 |
| *total_items_var* | 存储在 SDA 中描述的项的数量的主变量 | 必须为整数数据类型 | 特定于语言 |

**用法**

请随同 GBase 8s ESQL/C 使用此语句。

使用 GET DESCRIPTOR 语句来完成下列所有任务：

- 确定系统描述符区域描述了多少项。
- 确定系统描述符区域中描述的每一列或表达式的特征。
- 将值从系统描述符区域复制到 FETCH 语句之后的主变量。

在您以 DESCRIBE . . . USING SQL DESCRIPTOR 语句描述 EXECUTE FUNCTION、INSERT、SELECT 或 UPDATE 之后，可使用 GET DESCRIPTOR。

您在 GET DESCRIPTOR 语句中引用的主变量必须在程序的 BEGIN DECLARE SECTION 中声明。

如果在赋值给任何指定的主变量期间发生错误，则主变量的内容未定义。

**示例**

下列 ESQL/C 示例展示如何随同主变量使用 GET DESCRIPTOR 语句来确定名为 desc1: GET DESCRIPTOR 的系统描述符区域中描述了多少项

```
main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        int h_count;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL allocate descriptor 'desc1' with max 20;

    /* 这部分程序会将 SELECT 或 INSERT 语句准备到 s_id 语句 id 内。
    */

    EXEC SQL describe s_id using sql descriptor 'desc1';
    EXEC SQL get descriptor 'desc1' :h_count = count;
```

下列 ESQL/C 示例使用 GET DESCRIPTOR 来从 demodesc 系统描述符区域获取数据类型信息：

```
EXEC SQL get descriptor 'demodesc' value
    :index :type = TYPE,
    :len = LENGTH,
    :name = NAME;
printf("Column %d: type = %d, len = %d, name = %s\n",
    index, type, len, name);
```

下列 ESQL/C 示例展示您可如何将数据从 DATA 域赋值到访存之后的主变量（结果）内。对于此示例，预先约定所有返回值具有相同的数据类型：

```
EXEC SQL get descriptor 'demodesc' :desc_count = count;
.. .
EXEC SQL fetch democursor using sql descriptor 'demodesc';
for (i = 1; i <= desc_count; i++)
    {
    if (sqlca.sqlcode != 0) break;
    EXEC SQL get descriptor 'demodesc' value :i :result = DATA;
```

```
        printf("%s ", result);
}
printf("\n");
```

## 使用 COUNT 关键字

使用 COUNT 关键字来确定在系统描述符区域中描述了多少项。

下列 GBase 8s ESQL/C 示例展示如何随同主变量使用 GET DESCRIPTOR 语句来确定在名为 desc1 的系统描述符区域中描述了多少项：

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
int h_count;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc1' with max 20;


   /* 这部分程序会将 SELECT 或 INSERT 语句准备到 s_id 语句 id 内。
   */
EXEC SQL describe s_id using sql descriptor 'desc1';

EXEC SQL get descriptor 'desc1' :h_count = count;
...
}
```

## 使用 VALUE 子句

使用 VALUE 子句来获取关于描述的列或表达式的信息，或来接收数据库服务器在系统描述符区域中返回的值。

***item_num*** 必须大于零（0）但不大于在以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域时指定的项描述符的数量。

# 在 DESCRIBE 之后使用 VALUE 子句

在您描述 SELECT、EXECUTE FUNCTION（或 EXECUTE PROCEDURE）、INSERT 或 UPDATE 语句之后，将由 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句返回的值的特征，或者 INSERT 或 UPDATE 语句中每一列的特征返回到系统描述符区域。系统描述符区域中的每一值都描述一个返回的列或返回的表达式的特征。

下列 GBase 8s ESQL/C 示例使用 GET DESCRIPTOR 来从 demodesc 系统描述符区域获取数据类型信息：

```
EXEC SQL get descriptor 'demodesc' value :index
        :type = TYPE,
```

```
        :len = LENGTH,
        :name = NAME;
printf("Column %d: type = %d, len = %d, name = %s\n",
       index, type, len, name);
```

数据库服务器返回到 TYPE 字段内的值是定义的整数。要计算返回的数据类型的值，请测试特定的整数值。要获取关于整数数据类型值的附加信息，请参阅 设置 TYPE 或 ITYPE 字段。

在 X/Open 模式下，X/Open 代码返回到 TYPE 字段。您不可混用这两种模式，因为会导致错误。例如，如果在 X/Open 模式下未定义特定的数据类型，但为 GBase 8s 产品定义了特定的数据类型，则执行 GET DESCRIPTOR 语句可导致错误。

在 X/Open 模式下，如果使用 ILENGTH、IDATA 或 ITYPE，则产生错误消息。该消息说明这些字段不是系统描述符区域的标准 X/Open 字段。

如果访存的值的 TYPE 为 DECIMAL 或 MONEY，在执行 DESCRIBE 语句之后，数据库服务器将列的精度和小数位信息返回到 PRECISION 和 SCALE 字段内。如果 TYPE ***不是*** DECIMAL 或 MONEY，则取消定义 SCALE 和 PRECISION 字段。

## 在 FETCH 之后使用 VALUE 子句

每一次您的程序访存行时，它必须将访存的值复制到主变量内，以便可使用该数据。要完成此任务，在每次访问选择列表中的每一值之后，请使用 GET DESCRIPTOR 语句。如果在选择类表中存在三个值，则在每次访存之后，您需要使用三个 GET DESCRIPTOR 语句（假设您想要读取全部三个值）。这三个 GET DESCRIPTOR 语句的每一项编号为 1、 2 和 3。

下列 GBase 8s ESQL/C 示例展示您可如何在访存之后将数据从 DATA 域复制到主变量（**result**）内。对于此示例，预先确定所有的返回值为相同的数据类型：

```
EXEC SQL get descriptor 'demodesc' :desc_count = count;
.. .
EXEC SQL fetch democursor using sql descriptor 'demodesc';
for (i = 1; i <= desc_count; i++)
    {
    if (sqlca.sqlcode != 0) break;
    EXEC SQL get descriptor 'demodesc' value :i :result = DATA;
    printf("%s ", result);
    }
printf("\n");
```

## 访存 NULL 值

在访存之后您使用 GET DESCRIPTOR，且访存的值为 NULL 时，将 INDICATOR 字段设置为 -1 来指示 NULL 值。如果 INDICATOR 指示 NULL 值，则不定义 DATA 的值。复制 DATA 到其中的主变量有不可预测的值。

## 使用 LENGTH 或 ILENGTH

如果您的 DATA 或 IDATA 字段包含字符串，则必须指定 LENGTH 的值。如果您指定 LENGTH=0，则自动地将 LENGTH 设置为字符串的最大长度。DATA 或 IDATA 字段可能包含文字字符串或从 CHAR 或 VARCHAR 数据类型的字符变量派生的字符串。这提供了一种方法来动态地确定 DATA 或 IDATA 字段中的字符串长度。

如果 DESCRIBE 语句在 GET DESCRIPTOR 语句之前，则自动地将 LENGTH 设置为在表中指定的字符串字段的最大长度。

对于 ILENGTH，此信息相同。在您创建不符合 X/Open 标准的动态程序时，请使用 ILENGTH。

## 描述不透明类型列

当要访存的列有不透明类型作为其数据类型时，DESCRIBE 语句设置下列项描述符字段：

- EXTYPEID 字段存储不透明类型的扩展的 ID。此整数是与 **sysxtdtypes** 系统目录表的相对应的 **extended_id** 列中的值。
- EXTYPENAME 字段存储不透明类型的名称。此字符值是与 **sysxtdtypes** 系统目录表中 **extended_id** 相匹配的行的 **name** 列中的值。
- EXTYPELENGTH 字段存储不透明类型名称的长度。此整数是数据类型名称的长度（以字节计）。
- EXTYPEOWNERNAME 字段存储不透明类型所有者的名称。此字符值是与 **sysxtdtypes** 系统目录表中 **extended_id** 相匹配的行的 **owner** 列中的值。
- EXTYPEOWNERLENGTH 字段存储 EXTTYPEOWNERNAME 字段中值的长度。此整数是以字节计的该不透明类型的所有者名的长度。

随同 GET DESCRIPTOR 语句使用这些字段名称来获取关于不透明列的信息。

## 描述 distinct 类型列

在要访存的列有以 distinct 类型作为其数据类型时，DESCRIBE 语句设置下列项描述符字段：

- SOURCEID 字段存储 source 数据类型的扩展标识符。

  此整数具有 **sysxtdtypes** 系统目录表的行的 **source** 列的值，其 **extended_id** 值与 distinct 数据类型的值相匹配。仅当 source 数据类型是不透明数据类型时才设置此字段。

- SOURCETYPE 字段存储 source 数据类型的数据类型常量。

  此值是 DISTINCT 数据类型的 source 类型的数据类型的数据类型变量（来自 **sqltypes.h** 文件）。该 SOURCETYPE 字段的代码罗列在 SET DESCRIPTOR 语句中的 TYPE 字段的描述之中。（要获取更多信息，请参阅 设置 TYPE 或 ITYPE 字段。）此整数值必须对应于 **sysxtdtypes** 系统目录表的行的 **type** 列中的值，其 **extended_id** 值与 DISTINCT 数据类型的值相匹配。

请随同 GET DESCRIPTOR 语句使用这些字段名称来获取关于 distinct 类型列的信息。

# 2.94 GET DIAGNOSTICS 语句

使用 GET DIAGNOSTICS 语句来返回关于最近执行的 SQL 语句的诊断信息。

**语法**


**用法**

请随同 GBase 8s ESQL/C 使用此语句。

GET DIAGNOSTICS 语句检索数据库服务器记录在名为***诊断区域***中的指定的状态信息。使用 GET DIAGNOSTICS 不会更改诊断区域的内容。

GET DIAGNOSTICS 语句使用下列两个子句中的一个：

- Statement 子句返回关于最近的 SQL 语句生成的错误和警告的计数和溢出信息。
- EXCEPTION 子句提供关于最近的 SQL 语句生成的错误和警告的特定信息。

## 使用 SQLSTATE 错误状态代码

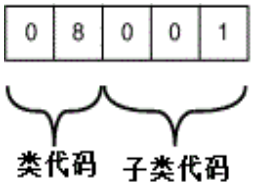
在执行 SQL 语句时，自动地生成错误状态代码。此代码表示 success、failure、warning 或 no data found。该错误状态代码存储在名为 **SQLSTATE** 的内建变量中。

## 类和子类代码

**SQLSTATE** 状态代码是一个仅可包含数字和大写字母的五字符的字符串。

SQLSTATE 状态代码的前两个字符表示类。SQLSTATE 代码的后三个字符表示子类。图 1 展示 SQLSTATE 代码的结构。此示例使用值 08001，此处 08 是类代码，001 是子类代码。值 08001 表示 unable to connect with database environment 错误。

图: SQLSTATE 代码的结构




下列表时解释类代码值的快速参考。

**SQLSTATE 类代码值　　结果**

| | |
|---|---|
| 00 | 成功 |
| 01 | 成功但有警告 |
| 02 | 找不到数据 |
| > 02 | 错误或警告 |

## SQLSTATE 支持 SQL 的 ANSI/ISO 标准

返回到 SQLSTATE 变量的所有状态代码都符合 ANSI，除了以下几种情况：

- 带有 01 类代码和以 I 开头的子类代码的 **SQLSTATE** 代码是特定于 GBase 8s 的警告消息。
- 带有 IX 类代码和任何子类代码的 **SQLSTATE** 代码是特定于 GBase 8s 的错误消息。
- 带有以从 5 到 9 中的一个数字或从 I 到 Z 中的一个大写字母开头的 **SQLSTATE** 代码，表示 SQL 的 ANSI/ISO 标准当前未定义的条件。唯一的例外是类代码为 IX 的 **SQLSTATE** 代码，这些代码是特定于 GBase 8s 的错误消息。

GBase 8s 的客户端/服务器通信协议，诸如 SQLI 和 DRDA®，支持这些 **SQLSTATE** 代码值。

## SQLSTATE 代码列表

此表描述与 SQLSTATE 变量相关联的类代码、子类代码和所有有效警告和错误代码的含义。

| 类 | 子类 | 含义 |
|---|---|---|
| 00 | 000 | 成功。 |
| 01 | 000 | 成功但有警告。 |
| 01 | 002 | 断开连接错误。事务回滚。 |
| 01 | 003 | 在集合函数中消除的 NULL 值。 |
| 01 | 004 | 字符串数据，右截断。 |
| 01 | 005 | 项描述符区域不足。 |
| 01 | 006 | 未调用的权限。 |
| 01 | 007 | 未授予的权限。 |
| 01 | I01 | 数据库有事务。 |
| 01 | I03 | 选择了符合 ANSI 的数据库。 |
| 01 | I04 | 选择了 GBase 8s 数据库服务器。 |
| 01 | I05 | 使用了浮点到十进制转换。 |
| 01 | I06 | GBase 8s 扩展到符合 ANSI 的语法。 |

| 类 | 子类 | 含义 |
|---|---|---|
| 01 | I07 | UPDATE 或 DELETE 语句没有 WHERE 子句。 |
| 01 | I08 | 使用 ANSI 关键字作为游标名称。 |
| 01 | I09 | 投影列表的基数与 INTO 列表的基数不相等。 |
| 01 | I10 | 运行在备用模式下的数据库服务器。 |
| 01 | I11 | 打开 Dataskip。 |
| 02 | 000 | 找不到数据。 |
| 07 | 000 | 动态的 SQL 错误。 |
| 07 | 001 | USING 子句与动态参数不匹配。 |
| 07 | 002 | USING 子句与目标规范不匹配。 |
| 07 | 003 | 不可执行游标规范。 |
| 07 | 004 | 动态参数需要 USING 子句。 |
| 07 | 005 | 准备好的语句不是游标规范。 |
| 07 | 006 | 违背受限的数据类型属性。 |
| 07 | 008 | 无效的描述符计数。 |
| 07 | 009 | 无效的描述符索引。 |
| 08 | 000 | 连接异常。 |
| 08 | 001 | 数据库服务器拒绝连接。 |
| 08 | 002 | 连接名称在用。 |
| 08 | 003 | 连接不存在。 |
| 08 | 004 | 客户端无法建立连接。 |
| 08 | 006 | 事务已回滚。 |
| 08 | 007 | 事务状态未知。 |
| 08 | S01 | 通信失败。 |
| 0A | 000 | 不支持的特性。 |
| 0A | 001 | 多服务器事务。 |
| 21 | 000 | 基数违反。 |
| 21 | S01 | 插入数据列表与列列表不匹配。 |
| 21 | S02 | 派生表的程度与列列表不匹配。 |

| 类 | 子类 | 含义 |
|---|---|---|
| 22 | 000 | 数据异常。 |
| 22 | 001 | 字符串数据，右截断。 |
| 22 | 002 | NULL 值，无指示符参数。 |
| 22 | 003 | 分数值超范围。 |
| 22 | 005 | 赋值错误。 |
| 22 | 027 | 数据异常调整错误。 |
| 22 | 012 | 除以零（0）。 |
| 22 | 019 | 无效的转义字符。 |
| 22 | 024 | 未终止的字符串。 |
| 22 | 025 | 无效的转义顺序。 |
| 23 | 000 | 完整性约束未被。 |
| 24 | 000 | 无效的游标状态。 |
| 25 | 000 | 无效的事务状态。 |
| 2B | 000 | 从属的权限描述符仍存在。 |
| 2D | 000 | 无效的事务终止。 |
| 26 | 000 | 无效的 SQL 语句标识符。 |
| 2E | 000 | 无效的连接名称。 |
| 28 | 000 | 无效的用户授权规范。 |
| 33 | 000 | 无效的 SQL 描述符名称。 |
| 34 | 000 | 无效的游标名称。 |
| 35 | 000 | 无效的异常号。 |
| 37 | 000 | 在 PREPARE 或 EXECUTE IMMEDIATE 中的语法错误或访问违背。 |
| 3C | 000 | 重复的游标名称。 |
| 40 | 000 | 事务回滚。 |
| 40 | 003 | 语句完成未知。 |
| 42 | 000 | 语法错误或访问违背。 |
| S0 | 000 | 无效的名称。 |

| 类 | 子类 | 含义 |
|---|---|---|
| S0 | 001 | 基础表或视图表已存在。 |
| S0 | 002 | 未找到基础表。 |
| S0 | 011 | 索引已存在。 |
| S0 | 021 | 列已存在。 |
| S1 | 001 | 内存分配失败。 |
| IX | 000 | GBase 8s 保留的错误消息。 |

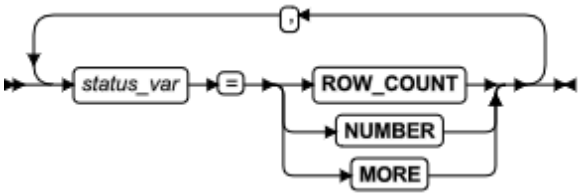
## 在应用中使用 SQLSTATE

您可在程序中使用名为 **SQLSTATE** 的内建变量，无需声明。**SQLSTATE** 包含对错误处理不可或缺的状态代码，程序每一次执行 SQL 语句都会生成状态代码。自动地创建 **SQLSTATE**。您可检测 **SQLSTATE** 变量来确定 SQL 语句是否成功。如果 **SQLSTATE** 变量指出语句失败，则可执行 GET DIAGNOSTICS 语句来获取附加的错误信息。

要获取展现如何在程序中使用 **SQLSTATE** 变量的示例，请参阅 使用 GET DIAGNOSTICS 进行错误检查。

### Statement 子句

Statement 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *status_var* | 主变量，用于接收关于指定的状态域名称的最近 SQL 语句的状态信息 | 必须与域的数据类型相匹配 | 特定于语言 |

在检索计数和溢出信息时，GET DIAGNOSTICS 可将这三个语句域的值存入对应的主变量。主变量数据类型必须与请求的域的数据类型相同。下列关键字表示这三个域。

| 域名称关键字 | 域数据类型 | 域内容 | ESQL/C 主变量数据类型 |
|---|---|---|---|
| MORE | 字符 | Y 或 N | char[2] |
| NUMBER | 整数 | 1 至 35,000 | int |
| ROW_COUNT | 整数 | 0 至 999,999,999 | int |

## 使用 MORE 关键字

使用 MORE 关键字来确定最近执行的 SQL 语句是否导致数据库服务器的下列操作：

- 将它检测到的所有异常存储在诊断区域中

  如果如此，则 GET DIAGNOSTICS 返回值 N。
- 检测到的异常多于它存储在诊断区域中的异常

  如果如此，则 GET DIAGNOSTICS 返回值 Y。（MORE 的值通常为 N。）

## 使用 ROW_COUNT 关键字

ROW_COUNT 关键字返回最近执行的 DML 语句处理的行数。ROW_COUNT 对这些行计数：

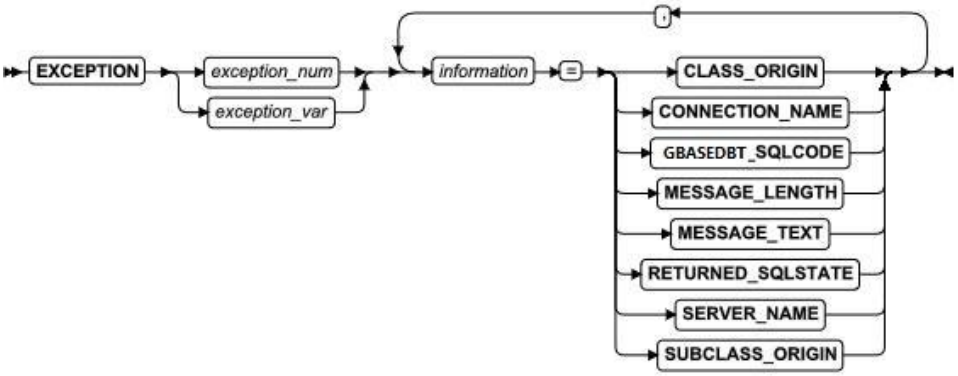
- 插入到表内的行
- 在表中更新的行
- 从表删除的行

## 使用 NUMBER 关键字

NUMBER 关键字返回最近执行的 SQL 语句发生的异常的数量。根据计数得到的异常数，NUMBER 字段可保留从 1 至 35,000 的值。

## EXCEPTION 子句

Exception 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *exception_num* | 异常的数量 | 从 1 至 35,000 的整数 | 精确数值 |
| *exception_var* | 存储 *exception_num* 的变量 | 必须为 SMALLINT 或 INT | 特定于语言 |
| *information* | 接收指定的异常域的值的主变量 | 数据类型必须与指定的字段的数据类型向匹配 | 特定于语言 |

***exception_num*** 文字表示从 Statement 子句中的 NUMBER 字段返回的异常的数量的异常值之一。

在检索异常信息时，GET DIAGNOSTICS 将七个字段中的每一个的值写到相应的主变量内。这些域定位在诊断区域中，并从最近的 SQL 语句产生的异常派生而来。

主变量数据类型必须与请求的域的数据类型相同。下表描述这七个异常信息域。

| 域名称关键字 | 域数据类型 | 域内容 | ESQL/C 主变量数据类型 |
|---|---|---|---|
| RETURNED_SQLSTATE | 字符 | SQLSTATE 值 | char[6] |
| GBASEDBT_SQLCODE | 整数 | 特定于 GBase 8s 的状态代码 | int4 |
| CLASS_ORIGIN | 字符 | 字符串 | char[255] |
| SUBCLASS_ORIGIN | 字符 | 字符串 | char[255] |
| MESSAGE_TEXT | 字符 | 字符串 | char[255] |
| MESSAGE_LENGTH | 整数 | 数值 | int |
| SERVER_NAME | 字符 | 字符串 | char[255] |
| CONNECTION_NAME | 字符 | 字符串 | char[255] |

应用通过编号指定异常，或使用无符号的整数，或使用整数主变量（小数位为 0 的精确数值）。值为 1 的异常对应于由最近的 SQL 语句而不是 GET DIAGNOSTICS 设置的 **SQLSTATE** 值。其他异常编号与由 SQL 语句产生的其他异常之间的关联未定义。这样，不存在设置顺序，异常值依次顺序填充到诊断区域。您通常会得到至少一个异常，即使 **SQLSTATE** 值表示成功。

如果在 GET DIAGNOSTICS 语句内发生错误（即，如果请求了无效的异常编号），则将 GBase 8s 内部的 **SQLCODE** 和 **SQLSTATE** 变量设置为那个异常的值。此外，未定义 GET DIAGNOSTICS 字段。

## 使用 RETURNED_SQLSTATE 关键字

RETURNED_SQLSTATE 关键字返回该异常描述的 **SQLSTATE** 值。

## 使用 GBASEDBT_SQLCODE 关键字

GBASEDBT_SQLCODE 关键字返回特定于 GBase 8s 的状态代码。在全局的 **SQLCODE** 变量中也可得到相同的值。要获取更多信息，请参阅 *GBase 8s ESQL/C 程序员手册* 中对 **SQLCODE** 变量的讨论。

## 使用 CLASS_ORIGIN 关键字

使用 CLASS_ORIGIN 关键字来检索 RETURNED_SQLSTATE 值的类部分。如果 SQL 的 ISO 标准定义该类，则 CLASS_ORIGIN 的值等于 ISO 9075。否则，由 CLASS_ORIGIN 返回的值是由 GBase 8s 定义的，且不可为 ISO 9075。术语 ANSI SQL 和 ISO SQL 是同义词。

## 使用 SUBCLASS_ORIGIN 关键字

SUBCLASS_ORIGIN 关键字返回 RETURNED_SQLSTATE 子类上的数据。（如果该子类定义 ISO 标准，则此值为 ISO 9075。）

## 使用 MESSAGE_TEXT 关键字

MESSAGE_TEXT 关键字返回异常的消息文本（例如，错误消息）。

## 使用 MESSAGE_LENGTH 关键字

MESSAGE_LENGTH 关键字返回以字节计的当前消息文本字符串的长度。

## 使用 SERVER_NAME 关键字

SERVER_NAME 关键字返回与 CONNECT 或 DATABASE 语句相关的数据库服务器的名称。在发生任何下列事件时，GET DIAGNOSTICS 更新 SERVER_NAME 字段：

- CONNECT 语句成功地执行。
- SET CONNECTION 语句成功地执行。
- DISCONNECT 语句成功地终止当前连接。
- DISCONNECT ALL 语句失败。

然而，在这些事件之后，不更新 SERVER_NAME 字段：

- CONNECT 语句失败。
- DISCONNECT 语句失败（但这不包括 DISCONNECT ALL 语句）。
- SET CONNECTION 语句失败。

SERVER_NAME 字段保留在先前的 SQL 语句中设置的值。如果在执行的第一个 SQL 语句上发生任何上述情况，则 SERVER_NAME 字段为空白。

### SERVER_NAME 域的内容

在您执行下列语句之后，SERVER_NAME 域包含不同的信息。

| 执行的语句 | SERVER_NAME 域内容 |
| --- | --- |
| CONNECT | 包含您连接到得或不能连接到的数据库服务器的名称。如果您没有当前连接，或使用缺省连接，则域为空白。 |
| DATABASE | 包含指定的数据库所在的数据库服务器的名称。 |

DISCONNECT　包含您断开连接的或未能断开连接的数据库服务器的名称。如果您切断连接，然后执行非当前连接的 DISCONNECT 语句，则 SERVER_NAME 域保持不变。

DISCONNECT ALL　如果该语句执行成功，则将此域设置为空白。如果该语句失败，则 SERVER_NAME 包含您未断开连接的所有数据库服务器的名称。（此信息不表示连接仍存在。）

SET CONNECTION　包含您切换到的或未切换到的数据库服务器的名称

如果 CONNECT 成功，则将 SERVER_NAME 设置为下列值之一：

- **GBASEDBTSERVER** 值（如果该连接是连接到缺省的数据库服务器，因为 CONNECT 未指定数据库服务器）
- 数据库服务器的名称（如果该连接是连接到特定的数据库服务器）

## 使用 CONNECTION_NAME 关键字

使用 CONNECTION_NAME 关键字来返回您在 CONNECT 或 SET CONNECTION 语句中指定的连接的名称。

## 在更新 CONNECTION_NAME 关键字时

在发生下列情况时，GET DIAGNOSTICS 更新 CONNECTION_NAME 域

- CONNECT 语句成功地执行。
- SET CONNECTION 语句成功地执行。
- 在当前的连接中 DISCONNECT 语句成功地执行。

  GET DIAGNOSTICS 以空白填充 CONNECTION_NAME 域，因为不存在当前连接。

- DISCONNECT ALL 语句失败。

## 在未更新 CONNECTION_NAME 时

在下列情况下，不更新 CONNECTION_NAME 域：

- CONNECT 语句失败。
- DISCONNECT 语句失败（但这不包括 DISCONNECT ALL 语句）。
- SET CONNECTION 语句失败。

CONNECTION_NAME 域保留在先前的 SQL 语句中设置的值。如果在执行第一个 SQL 语句时发生上述任何情况，则 CONNECTION_NAME 域为空白。

隐式连接没有名称。在 DATABASE 语句成功地创建隐式连接之后，CONNECTION_NAME 域为空白。

### CONNECTION_NAME 域的内容

CONNECTION_NAME 域包含依赖于先前执行的 CONNECT、SET CONNECTION、DISCONNECT 或 DISCONNECT ALL 语句的连接信息。

在您执行下列语句之后，CONNECTION_NAME 域包含不同的信息

| 执行的语句 | CONNECTION_NAME 域内容 |
| --- | --- |
| CONNECT | 包含连接或未连接的连接名称，在 CONNECT 语句中指定。对于无当前连接或缺省连接，该区域为空白。 |
| SET CONNECTION | 包含切换或未切换的连接名称，在 SET CONNECTION 语句中指定。 |
| DISCONNECT | 包含断开连接或未断开连接的连接名称，在 DISCONNECT 语句中指定。如果您成功地断开连接，然后执行非当前连接的 DISCONNECT 语句，则 CONNECTION_NAME 域保持不变。 |
| DISCONNECT ALL | 如果成功地执行 DISCONNECT ALL 语句，则不包含信息。如果未成功地执行该语句，则 CONNECTION_NAME 与包含在 CONNECT 语句中指定的未断开连接的所有连接的名称。然而，此信息不意味着连接仍存在。 |

如果 CONNECT 成功，则 CONNECTION_NAME 取这些值之一：

- 在 CONNECT 语句中指定的数据库环境的名称，如果 CONNECT 语句不包括 AS 子句
- 连接的名称（在 AS 关键字之后声明了的标识符），如果 CONNECT 语句包括 AS 子句

## 使用 GET DIAGNOSTICS 进行错误检查

GET DIAGNOSTICS 从诊断区域中的各种域返回值。对于您想要访问的诊断区域中的每一域，必须支持兼容数据类型的主变量。

下列示例展示如何使用 GET DIAGNOSTICS 语句来显示错误信息。该示例展示名为 disp_sqlstate_err( ) 的 GBase 8s ESQL/C 错误显示例程：

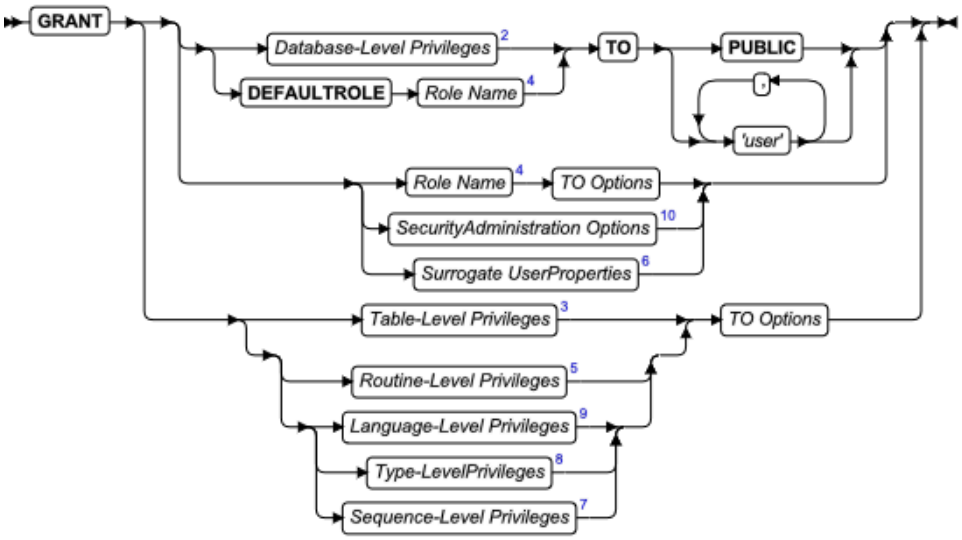
```
void disp_sqlstate_err()
{
int j;
EXEC SQL BEGIN DECLARE SECTION;
    int exception_count;
    char overflow[2];
    int exception_num=1;
    char class_id[255];
    char subclass_id[255];
    char message[255];
    int messlen;
    char sqlstate_code[6];
    int i;
EXEC SQL END DECLARE SECTION;
    printf("--------------------------------");
    printf("------------------------\n");
    printf("SQLSTATE:
    printf("SQLCODE: %d\n", SQLCODE);
```

```
    printf("\n");
    EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
    printf("EXCEPTIONS:    Number=%d\t", exception_count);
    printf("More? %s\n", overflow);
    for (i = 1; i <= exception_count; i++)
    {
        EXEC SQL get diagnostics    exception :i
            :sqlstate_code = RETURNED_SQLSTATE,
            :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
            :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
        printf("- - - - - - - - - - - - - - - - - - -\n");
        printf("EXCEPTION %d: SQLSTATE=%s\n", i, sqlstate_code);
        message[messlen-1] ='\0';
        printf("MESSAGE TEXT: %s\n", message);
        j = stleng(class_id);
        while((class_id[j] == '\0') ||
              (class_id[j] == ' '))
            j--;
        class_id[j+1] = '\0';
        printf("CLASS ORIGIN:
        j = stleng(subclass_id);
        while((subclass_id[j] == '\0') ||
              (subclass_id[j] == ' '))
            j--;
        subclass_id[j+1] = '\0';
        printf("SUBCLASS ORIGIN:
    }
    printf("--------------------------------");
    printf("------------------------\n");
}
```

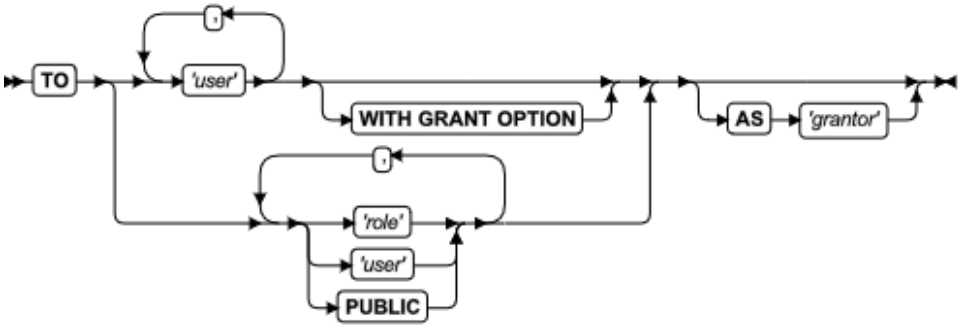
## 2.95 GRANT 语句

使用 GRANT 语句来给用户和其他角色指定访问权限和角色。拥有 DBSECADM 角色的用户可使用此语句来从基于标签的访问控制（LBAC）安全规则指定用户安全标签和豁免。

**语法**




TO 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *grantor* | 用户的授权标识符，该用户可使用 REVOKE 来取消此 GRANT 语句的作用。如果省略 AS 子句，则缺省值是发出此语句的用户的登录名 | 必须为有效的 *user* 名称(非 *role* 名称)。在 Windows™ 上，*user* 名称不可超过 20 字节。在其他平台上，该限制为 32 字节。 | 所有者名称 |
| *role* | 现有角色的名称，您将一个或多个访问权限授予该角色，或您指定另一角色给该角色 | 在该数据库中必须存在 | 所有者名称 |
| *user* | 用户的授权标识符，您将一个或多个访问权限授予该用户，或您将指定角色给该用户 | 同 *grantor* | 所有者名称 |

## 用法

GRANT 语句扩展到其他用户特定的自主访问权限或 LBAC 标签和豁免，这些通常仅归于 DBA 或对象的创建者。后续的 GRANT 语句不影响已授予用户的那些权限。

您可使用 GRANT 语句进行如下操作：

- 授权其他人来使用或管理您创建的数据库
- 允许其他人来查看、改变或删除您创建的表、同义词、视图或序列对象
- 允许其他人来使用数据类型或 SPL 语言，或来执行您创建的用户定义的例程（UDR）
- 将角色及其权限分配给用户、给 PUBLIC，或给其他角色
- 将缺省角色分配给一个或多个用户或给 PUBLIC
- 如果您具有 DBSECADM 角色，则从 LBAC 安全策略的规则将 LBAC 安全标签或豁免分配给用户，

您可将权限授予先前创建的角色或授予内嵌角色。您可将角色授予 PUBLIC、授予个别用户，或授予另一角色。

如果您用引号括起 ***grantor***、 ***role*** 或 ***user***，则名称是区分大小写的，并完全按您输入的形式存储。在符合 ANSI 的数据库中，如果您不使用引号做定界符，则以大写字母存储该名称。

仅在 Windows 上，数据库服务器不支持包含多于 20 字符的 ***user*** 名称。

您授予的权限保持有效直到您以 REVOKE 语句取消为止。仅权限的 grantor 可调用那个权限。grantor 是发出 GRANT 语句的人，除非 AS ***grantor*** 子句将调用那些权限的权利转给其他用户。

仅对象的所有者或以 WITH GRANT OPTION 关键字明确地授予权限的用户可授予对象的权限。有 DBA 权限还不够。然而，作为 DBA，您可通过使用 AS ***grantor*** 子句代表另一用户授予权限。对于数据库对象的权限，其所有者不是操作系统识别的用户（例如，用户 **gbasedbt**），AS ***grantor*** 子句是有用的。

关键字 PUBLIC 将特定的的权限或角色扩展到 PUBLIC 组或连接到该数据库的所有用户。如果您想要将 PUBLIC 已持有的权限仅限制到用户的子集，则必须首先从 PUBLIC 取消那些权限。

要为已由表达式分段的表的一个或多个分段授予权限，请参阅 GRANT FRAGMENT 语句。

## 数据库级权限

数据库级访问权限影响对数据库的访问。仅个别用户，而不是角色，可持有数据库权限。

数据库级权限


当您以 CREATE DATABASE 语句创建数据库时，您是所有者并自动地收到所有数据库级权限。

数据库对所有其他用户保持为不可访问，直到您，作为 DBA，将数据库权限授予他们。

作为数据库所有者，您还自动地收到对该数据库中所有表的表级权限。要获取关于表级权限的更过信息，请参阅 表级权限。

**建议：** 仅用户 **gbasedbt** 可直接地修改系统目录表。然而，除了在数据库服务器文档中特别说明的之外，请勿直接地使用 DML 语句来插入、删除或更新系统目录表的行，因为修改这些表中的数据可损坏数据库的完整性。

当数据库级权限与表级权限冲突时，限制性更大的特权优先。

从最低至最高，数据库访问级别为 Connect、Resource 和 DBA。请使用相应的关键字来授予访问权限的级别。

| 权限 | 作用 |
|---|---|
| CONNECT | 让您查询和修改数据<br>您可修改数据库模式，如果您拥有想要修改的数据库对象。任何有 Connect 权限的用户都可执行下列操作：<br>• 以 CONNECT 语句或另一连接语句连接到该数据库<br>• 执行 SELECT、INSERT、UPDATE 和 DELETE 语句，如果用户有必要的表级权限<br>• 创建视图，如果用户有对底层表的 Select 权限<br>• 创建同义词<br>• 创建临时表及创建临时表上的索引<br>• 改变或删除表或索引，如果用户拥有该表或索引（或有该表上的 Alter、Index 或 References 权限）<br>• 授予表或视图的权限，如果该用户拥有该表（或已经由 WITH GRANT OPTION 关键字被授予对表的权限） |
| RESOURCE | 让您扩展数据库的结构。除了 Connect 权限的能力之外，Resource 权限的持有者可执行下列功能：<br>• 创建新标<br>• 创建新索引<br>• 创建新 UDR<br>• 创建新数据类型 |
| DBA | 有 Resource 权限的所有能力，且可执行下列附加的操作：<br>• 将任何数据库级权限授予另一用户，包括 DBA 权限<br>• 将任何表级权限授予另一用户或角色<br>• 将角色授予用户或授予另一角色<br>• 取消权限，您在 REVOKE 语句的 AS 子句中将其 grantor 指定为 ***revoker***<br>• 当注册 UDR 时，限制 DBA 的 Execute 权限<br>• 执行 SET SESSION AUTHORIZATION 语句<br>• 创建任何数据库对象<br>• 创建表、视图和索引，指定另一用户为这些对象的所有者<br>• 改变、删除或重命名数据库对象，不管谁拥有它们<br>• 执行 UPDATE STATISTICS 语句的 DROP DISTRIBUTIONS 选项<br>• 执行 DROP DATABASE 和 RENAME DATABASE 语句 |

用户 **gbasedbt** 有改变系统目录表所需的权限，包括 **systables** 表。

下列示例使用 PUBLIC 关键字来将当前活动的数据库的 Connect 权限授予任何用户：

```
GRANT CONNECT TO PUBLIC;
```

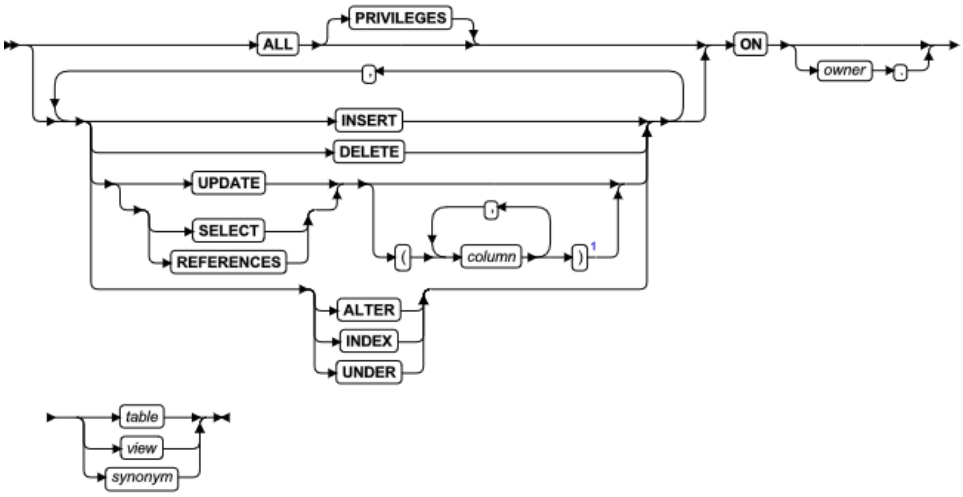
您不可将数据库级权限授予一个角色。仅个别的用户或 PUBLIC 可持有数据库级权限。

## 表级权限

当您以 CREATE TABLE 语句创建表时，您是该表的所有者，且自动地收到所有的表级权限。您不可将所有权转给另一用户，但您可将表级权限授予另一用户或授予一角色。（然而，请参阅 RENAME TABLE 语句，它可更改表的名称和所有权。）

有数据库级 DBA 权限的用户自动地收到那个数据库中每一表的所有表级权限。

表级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 授予对列的 References、Select 或 Update 权限。缺省的范围是 *table*、*view* 或 *synonym* 的所有列。 | 必须为 *table*、*view* 或 *synonym* 的列 | 标识符 |
| *owner* | 拥有该 *table*、*view* 或 *synonym* 的用户名称 | 必须为有效的授权标识符 | 所有者名称 |
| *synonym*, *table*, *view* | 授予权限的同义词、表或视图。 | 在当前数据库中必须存在 | 标识符 |

GRANT 语句可罗列下列关键字中一个或多个，来指定您授予同样的用户或角色的表权限。

| 权限 | 作用 |
|---|---|

| 权限 | 作用 |
| --- | --- |
| INSERT | 让您插入行 |
| DELETE | 让您删除行 |
| SELECT | 让您访问 SELECT 语句中的任何列。您可通过罗列列来将 Select 权限限定到一列或多列。 |
| UPDATE | 让您访问 UPDATE 语句中的任何列。您可通过罗列列来将 Update 权限限定到到一列或多列。 |
| REFERENCES | 让您定义对列的引用约束。您必须有 Resource 权限来利用 References 权限。（然而，您可在 ALTER TABLE 语句期间添加参考约束，而无需持有数据库的 Resource 权限。）您仅需 References 权限来指出级联删除。您无需 Delete 权限来在表上放置级联删除。您可通过罗列列来将 References 权限限定到一列或多列。 |
| INDEX | 让您创建永久索引。您必须有 Resource 权限来使用 Index 权限。（有 Connect 权限的任何用户可在临时表上创建索引。） |
| ALTER | 让您添加或删除列，修改列数据类型，添加或删除约束，将表的锁定模式由 PAGE 修改为 ROW，或为您的表添加或删除对应的 ROW 数据类型。 它还让您启用或禁用索引、约束和触发器，如 SET Database Object Mode 语句 中所描述。<br>您必须有 Resource 权限来使用 Alter 权限。此外，对于受 ALTER TABLE 语句影响的任何用户定义的数据类型，您还需要 Usage 权限。 |
| UNDER | 让您创建类型表之下的子表。 |
| ALL | 提供以上列出的所有权限。PRIVILEGES 关键字是可选的。 |

您可通过指定权限适用的列来缩小 Select、Update 或 References 权限的范围。

指定关键字 PUBLIC 为 ***user***，如果您想要 GRANT 语句应用于所有用户。

以下一些简单的示例展示如何以 GRANT 语句来授予表级权限。

下列语句将删除和选择表 **customer** 中的任何列的值的权限授予用户 **mary** 和 **john**。它还授予 Update 权限，但仅限于列 **customer_num**、**fname** 和 **lname**：

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
    ON customer TO mary, john;
```

要将上述相同的权限授予给所有授权的用户，请使用关键字 PUBLIC，如下列示例所示：

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
    ON customer TO PUBLIC;
```

对于 GBase 8s 数据库，假设名为 mary 的用户已经创建了名为 tab1 的类型表。在缺省情况下，仅用户 mary 可创建 tab1 表之下的子表。如果 mary 想要将在 tab1 表之下创建子表的能力授予名为 john 的用户，则 mary 必须输入下列 GRANT 语句：

```
GRANT UNDER ON tab1 TO john;
```

收到 **tab1** 表上的 Under 权限之后，用户 **john** 可创建 **tab1** 之下的一个或多个子表。

## ALL 关键字的作用

ALL 关键字将所有可能的表级权限授予指定的用户。如果对于 grantor 任何或所有表级权限都不存在，则带有 ALL 关键字的 GRANT 语句成功（如同 SQLCODE 设置为零，即使对于 grantor 在表上可能的权限设置为空）。然而，在这种情况下，返回下列 SQLSTATE 警告：

```
01007 - Privilege not granted.
```

例如，假设用户 **ted** 在 **customer** 上有 Select 和 Insert 权限，将那些权限授予其他用户。

用户 ted 想要将所有表级权限授予用户 tania。因此，用户 ted 发出下列 GRANT 语句：

```
GRANT ALL ON customer TO tania;
```

此语句成功地执行，但由于下列原因返回 SQLSTATE 代码 01007：

- 该语句成功地将 Select 和 Insert 权限授予用户 **tania**，因为用户 **ted** 有那些权限和将那些权限授予其他用户的权利。
- ALL 关键字暗含的其他权限不可由 **ted** 授予，因此没有授予用户 **tania**。

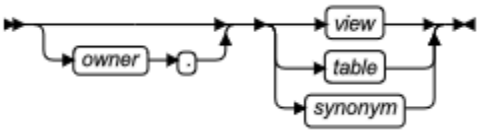
如果您以 ALL 关键字授予所有表级权限，则仅当该表为类型表时该权限才包括 Under 权限。ALL 权限的授予不包括 Under 权限，如果该表不是基于 ROW 类型的。

如果表所有者授予传统的关系表 ALL 权限，并在后来将那个表更改为类型表，则表所有者必须明确地授予 Under 权限来允许其他用户创建它之下的子表。

## 表引用

通过制定表的或视图的名称或现有的同义词，您直接授予表级权限，可以 *owner* 名称来限定。

表引用




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | 拥有 *table*、*view* 或 *synonym* 的用户的名称 | 必须为有效的授权标识符 | 所有者名称 |
| *synonym*, *table*, *view* | 对其授予权限的同义词、表或视图 | *table*、*view* 或 *synonym* 必须在数据库中存在 | 标识符 |

对其授予权限的对象必须位于当前数据库中。

对于 CREATE EXTERNAL TABLE 语句已经在当前数据库中注册的表对象，支持 Select 权限和 Insert 权限，但不可授予或取消对其他表或列的访问权限。

在符合 ANSI 的数据库中，如果 ***owner*** 未加引号，则数据库以小写字母方式存储所有者名称。

## 对表和同义词的权限

在符合 ANSI 的数据库中，如果您创建表，则仅有您作为该表的所有者拥有任何表级权限，直到您明确地将它们授予其他人为止。

然而，当您在*不*符合 ANSI 的数据库中创建表时，PUBLIC 获得那个表的 Select、Insert、Delete、Under 和 Update 权限。（当设置为 yes 时，**NODEFDAC** 防止 PUBLIC 自动地获得这些表级权限。）

要允许某些用户访问，或仅访问不符合 ANSI 的数据库中的某些列，您必须明确地取消 PUBLIC 在缺省情况下获得的权限，然后仅授予您想要授予的权限。例如，这一系列语句将对整个 customer 表的权限授予用户 john 和 mary，但限定 PUBLIC 仅对那个表中的四列有 Select 权限：

```
REVOKE ALL ON customer FROM PUBLIC;
GRANT ALL ON customer TO john, mary;
GRANT SELECT (fname, lname, company, city) ON customer TO PUBLIC;
```

## 对视图的权限

您对表或列必须至少有 Select 权限来创建在那个表上的视图。对那些在当前数据库中仅引用表的视图，如果视图所有者失去对该视图的任何基本表的 Select 权限，则删除该视图。

您对该视图有的权限与您对为该视图提供数据的一表或多表的权限相同。例如，如果您从仅有 Select 权限的表创建视图，则您可从该视图选择数据，但不可删除或更新数据。要获取更多关于如何创建视图的信息，请参阅 CREATE VIEW 语句。

当您创建视图时，PUBLIC 不会自动地获得对您创建的视图的任何权限。仅有您有权通过那个视图访问表数据。即使是对该视图的基本表有权限的用户，也不会自动地获得对该视图的权限。

仅当您是底层基本表的所有者，或您获得了对基本表的这些权限有权授予那些（通过 WITH GRANT OPTION 关键字指定的）权限，您才可对视图授予（或取消）权限。您必须在自己的权限之内明确地授予那些权限，因为在创建视图时，PUBLIC 没有自动地获得对它的任何权限。

视图的创建者可明确地将对该视图的 Select、Insert、Delete 和 Update 权限授予其他用户或授予一角色。您不可对视图授予 Index、Alter、Under 或 References 权限（也不可指定视图的 ALL 关键字，因为 ALL 包含 Index、References 和 Alter 权限）。

当 GRANT 或 REVOKE 语句更改对任何表的自主访问权限，而现有视图的定义中引用该表，则数据库服务器不自动地将那些权限变更应用到该视图。要将新表访问权限应用到依赖于那一表的视图，您可使用 DROP VIEW 和 CREATE VIEW 语句来删除并重新创建该视图。

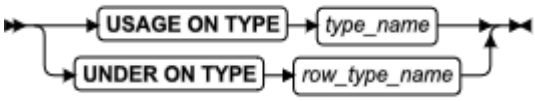
在此情况下，如果其他视图的定义引用您删除的视图，或如果在那一视图上定义了 INSTEAD OF 触发器，则您还可使用 CREATE VIEW 和 CREATE TRIGGER 语句来分别重新创建那些由 DROP VIEW 语句破坏的相依赖的视图和 INSTEAD OF 触发器。

## 类型级权限

您可对不是内建数据类型的数据类型指定两种权限：

- 对用户定义的数据类型的 Usage 权限
- 对命名的 ROW 类型的 Under 权限

类型级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *row_type_name* | 授予 Under 权限的命名的 ROW 类型 | 命名的 ROW 数据类型必须存在 | 标识符；数据类型 |
| *type_name* | 授予 Usage 权限的用户定义的类型 | 用户定义的数据类型必须存在。 | 标识符；数据类型 |

要看到在用户定义的数据类型上存在什么权限，请在 **sysxtdtypes** 系统目录表检查每一 UDT 的 ***owner***，并在 **sysxtdtypeauth** 系统目录表检查在 UDT 上持有权限的任何其他用户或角色。要获取关于系统目录表的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

然而，对于所有***内建数据类型***，PUBLIC 自动地获得这些访问权限且不可取消。

## USAGE 权限

您拥有您创建的任何用户定义的数据类型（UDT）。作为所有者，您自动地获得那种数据类型的 Usage 权限，并可将 Usage 权限授予其他人，以便他们可在 SQL 语句中引用该类型名称或数据。DBA 还可授予 UDT 的 Usage 权限。

下列示例授予用户 **mark** 访问权限来使用 **widget** 用户定义的类型：

```
GRANT USAGE ON TYPE widget TO mark;
```

如果您将 Usage 权限授予有 Alter 权限的用户（或角色），则被授予者可向包含您的 UDT 的值的表添加列。

没有来自 GRANT 语句的权限，任何用户都可发出引用内建数据类型的 SQL 语句。相反，用户必须从 GRANT 语句获得显示的 Usage 权限，才能使用 distinct 数据类型，即使该 distinct 类型基于内建类型。

要获得更多关于用户定义的类型的信息，请参阅 CREATE OPAQUE TYPE 语句、CREATE DISTINCT TYPE 语句，在 *《GBase 8s SQL 指南：参考》*中对数据类型的讨论。

## UNDER 权限

您拥有您创建的任何命名的 ROW 类型。如果您想要其他用户能够创建此命名的 ROW 类型之下的子类型，则必须授予这些用户对您的命名的 ROW 类型的 Under 权限。

例如，假设您创建名为 **rtype1** 的 ROW 类型：

```
CREATE ROW TYPE rtype1 (cola INT, colb INT);
```

如果您想要名为 **kathy** 的另一用户能够创建此命名的 ROW 类型之下的子类型，则必须授予用户 **kathy** 对此命名的 ROW 类型的 Under 权限：

```
GRANT UNDER ON ROW TYPE rtype1 TO kathy;
```

现在，用户 **kathy** 可创建 **rtype1** ROW 类型之下的另一 ROW 类型，即使 **kathy** 不是 **rtype1** ROW 类型的所有者：

```
CREATE ROW TYPE rtype2 (colc INT, cold INT) UNDER rtype1;
```

要获取更多关于命名的 ROW 类型的信息，请参阅 CREATE ROW TYPE 语句，以及 *《GBase 8s SQL 指南：参考》*中数据类型的讨论。

## 例程级权限

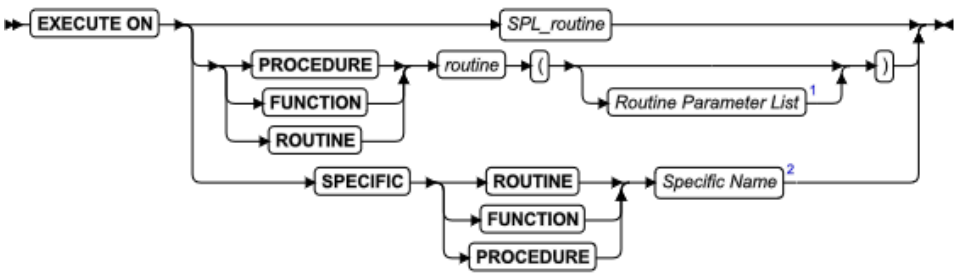
当您创建用户定义的例程(UDR)，您成为该 UDR 的所有者，您自动地获得对那个 UDR 的 Execute 权限。

Execute 权限允许您以 EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句，任何一个都适用，或以 SPL 例程中的 CALL 语句来调用该 UDR。 Execute 权限还允许您使用表达式中的用户定义的函数，如下示例所示：

```
SELECT * FROM table WHERE in_stock(partnum) < 20;
```

对于需要对给定的 UDR 有 Execute 权限的用户、角色或 PUBLIC 组的成员，GRANT 语句支持下列语法：

例程级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *routine* | 用户定义的例程 | 必须存在 | 标识符 |
| *SPL_routine* | SPL 例程 | 必须在数据库中是唯一的 | 标识符 |

下列语句将对 delete_order 例程的 Execute 权限授予用户 finn：

GRANT EXECUTE ON ROUTINE delete_order TO finn;

您是否必须显式地授予 Execute 权限，有赖于下列条件：

- 如果您有 DBA 级权限，则可使用 CREATE FUNCTION 或 CREATE PROCEDURE 的 DBA 关键字来限定那些有 DBA 权限的用户的缺省 Execute 权限。您必须显式地将对那个 UDR 的 Execute 权限显式地授予那些没有 DBA 权限的用户。
- 如果您有 Resource 数据库级权限但没有 DBA 权限，则当您创建 UDR 时不可使用 DBA 关键字：
  - 当您在*不*符合 ANSI 的数据库中创建 UDR 时，PUBLIC 可执行那个 UDR。您不需为其他获得 Execute 权限的用户发出 GRANT 语句。
  - 设置 **NODEFDAC** 环境变量为 yes 以防 PUBLIC 执行 UDR，直到您显式地授予 Execute 权限为止。
- 在符合 ANSI 的数据库中，UDR 的创建者必须显式地将该 UDR 的 Execute 权限授予其他用户，使之能够执行它。

在 GBase 8s 中，如果两个或多个 UDR 同名，则使用下列列表中的关键字来指定用户列表可执行那些 UDR 中的哪些。

| 关键字 | 用户可执行的 UDR |
|---|---|
| SPECIFIC | 由 ***specific name*** 标识的 UDR |
| FUNCTION | 任何带有指定的 ***routine name*** （以及与 ***routine 参数列表***相匹配的参数类型，如果指定的话）的函数 |
| PROCEDURE | 任何带有指定的 ***routine name*** (以及与***参数列表***相匹配的参数类型，如果指定的话）的过程 |
| ROUTINE | 带有指定的 ***routine name*** (以及与***例程参数列表***相匹配的参数类型，如果指定的话）的函数或过程 |

如果 GBase 8s 的用户定义的函数和用户定义的过程都有相同的名称和相同的参数数据类型的类表，则可以关键字 ROUTINE 给二者授予 Execute 权限。

要将 Execute 权限限定到有相同标识符的几个例程中的一个，请使用 FUNCTION、PROCEDURE 或 SPECIFIC 关键字。

要将 Execute 权限限定到接受特定数据类型为参数的 UDR，请包括例程参数列表或使用 SPECIFIC 关键字来引入 UDR 的特定名称。

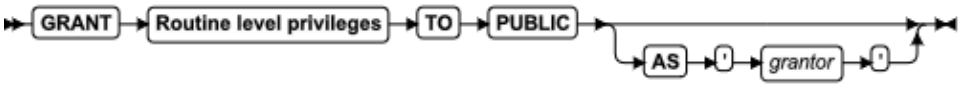
如果外部函数有 negator 函数，则在其他用户可执行该外部函数之前，您必须对外部函数及其 negator 函数都授予 Execute 权限。

用户必须对该语言持有 Usage 权限，该用户定义的例程由该语言编写并以 CREATE FUNCTION、CREATE FUNCTION FROM、CREATE PROCEDURE、CREATE PROCEDURE FROM 或 CREATE ROUTINE FROM 语句来注册 UDR。要获取更多关于注册 UDR 的要求的信息，请参阅 使用 CREATE FUNCTION 时必需的特权。

## 将 Execute 权限授予 PUBLIC

GRANT 语句支持将访问权限授予 PUBLIC 组的语法，该组包括持有对数据库的 Connect 权限的所有用户。

GRANT EXECUTE TO PUBLIC




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *grantor* | UDR 的所有者 | 不可为角色 | 所有者名称 |

此语句使得 PUBLIC 组中的每个用户都可以执行指定的例程。它覆盖 NODEFDAC 环境变量，如果该变量的设置阻止 PUBLIC 组在缺省情况下获得对例程的 Execute 权限的话。此语句还使那些不持有 DBA 权限的用户可以执行指定的例程，不论是否以 DBA 关键字创建了那个例程。

仅持有 DBA 权限的用户可指定 AS ***grantor*** 子句。指定的 grantor 必须为指定的例程的所有者，罗列在 **sysprocedures** 系统目录表的 **owner** 列中。grantor 不可为角色的名称或 PUBLIC 关键字。

在符合 ANSI 的数据库中，需要 AS ***grantor*** 子句，而不是作为可选项，如果发出 GRANT EXECUTE 语句的 DBA 不是指定的例程的所有者的话。

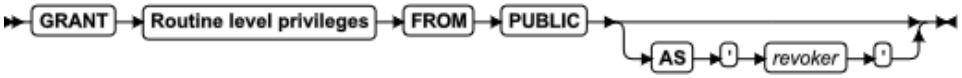
仅在可选的 AS ***grantor*** 子句中指定的用户可使用 SQL 的 REVOKE 语句来从 PUBLIC 组取消 Execute 权限

在可选的 AS ***grantor*** 子句中指定的用户可使用 REVOKE 语句来从 PUBLIC 组取消 Execute 权限。

## 将 Execute 权限从 PUBLIC 取消

REVOKE 支持下列从 PUBLIC 组取消对特定的例程的访问权限的语法，该组包括对数据库持有 Connect 权限的所有用户。

REVOKE EXECUTE TO PUBLIC




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *revoker* | UDR 的所有者 | 不可为角色 | 所有者名称 |

此语句防止 PUBLIC 组在缺省情况下获得对指定的例程的 Execute 权限。(然而，个别的持有 DBA 权限的用户，或拥有该例程的用户，或被个别地授予了或通过角色获得了对此例程的 Execute 权限的用户，他们不受此语句的影响。

仅持有 DBA 权限的用户可指定 AS ***revoker*** 子句。指定的取消者必须为指定的例程的所有者，罗列在 **sysprocedures** 系统目录表的 **owner** 列中。该名称不可为角色的名称或 PUBLIC 关键字。

在符合 ANSI 的数据库中，需要 AS ***revoker*** 子句，而不是可选的，如果发出 REVOKE EXECUTE 语句的 DBA 不是指定的例程的所有者的话。

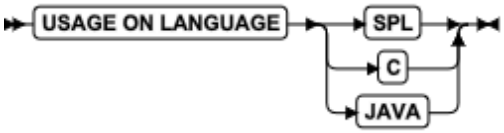
在 PUBLIC 组在缺省情况下持有对所有者权限的例程的 Execute 权限的数据库中，在将对执行指定的例程的自主访问权限可授权给用户的子集或授权给一个或多个角色之前，必须成功地执行 REVOKE EXECUTE ON PUBLIC 语句。否则，仅有 DBA 权限的用户或该例程的所有者可启动它。

## 语言级权限

GBase 8s 还支持***语言级权限***，该权限指定 UDR 的编程语言，已经被授予了给定语言的 Usage 权限的用户可将它注册在数据库中。

这是对编程语言授予 Usage 权限的 USAGE ON LANGUAGE 子句的语法：

语言级权限




SPL、C 和 JAVA 关键字可指定 USAGE ON LANGUAGE 子句中的编程语言。每一 GRANT USAGE ON LANGUAGE 语句至多可指定一种编程语言。在缺省情况下，将对 SPL 的 Usage 权限授予 PUBLIC。

当用户执行 CREATE FUNCTION 或 CREATE PROCEDURE 语句来注册以 SPL、C 或 Java™ 语言编写的 UDR 时，数据库服务器验证用户是否拥有对编写 UDR 所用语言的 Usage 权限。如果 IFX_EXTEND_ROLE 配置参数已经启用了内建 EXTEND 角色，则只有还持有那种角色的用户可注册或删除以 C 语言或以 Java 语言编写的 UDR，即使用户持有对那些语言的 USAGE ON LANGUAGE 权限。

GRANT USAGE ON LANGUAGE 语句可将对编程语言的 Usage 权限授予给用户的受限组。下列示例将对 C 语言的 Usage 权限授予名为 developers 的用户定义的角色：

```
GRANT USAGE ON LANGUAGE C TO developers;
```

如果上述示例成功地执行，则持有 developers 作为当前角色的用户可创建或删除 C 例程（如果他们还持有 EXTEND 角色，或者如果 IFX_EXTEND_ROLE 参数设置为 0 或 Off 的话）。

要获取关于这些语句需要的其他访问权限的信息，请参阅 CREATE FUNCTION 语句 和 CREATE PROCEDURE 语句。

## 在存储过程语言中的 Usage 权限

在缺省情况下，将对 SPL 的 Usage 权限授予 PUBLIC。仅用户 **gbasedbt**、DBA 或获得了 Usage 权限 WITH GRANT OPTION 的用户可将对 SPL 的 Usage 权限授予另一用户。

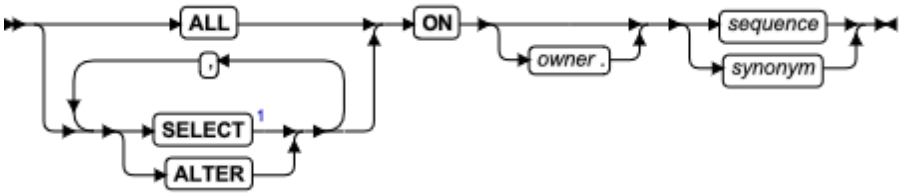
在下列示例中，假设从 PUBLIC 取消了对 SPL 的 Usage 权限，且 DBA 想要将对 SPL 的 Usage 权限授予名为 developers 的角色：

```
GRANT USAGE ON LANGUAGE SPL TO developers;
```

## 序列级权限

虽然 GBase 8s 以表的方式实现序列对象，但仅可对序列授予表级权限的子集（ 页 表级权限）。您可对序列授予 Select 和/或 Alter 权限：

序列级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | sequence 的所有者（或 *synonym* 的所有者） | 必须为所有者 | 所有者名称 |
| *sequence* | 对其授予权限的序列 | 必须存在 | 标识符 |
| *synonym* | 序列对象的同义词 | 必须存在 | 标识符 |

在当前数据库中必须存在该序列对象。您可以有效的 ***owner*** 名称限定该 ***sequence*** 或 ***synonym*** 标识符，但远程 ***database***（或 ***database@server***）的名称不是有效的限定符。当您将 ALTER、SELECT 或 ALL 授予用户或授予 PUBLIC（但不授予角色）作为对序列对象的权限时，可包括 WITH GRANT OPTION 关键字。

## Alter 权限

您可将对序列的 Alter 权限授予另一用户或角色。Alter 权限使得指定的用户或角色能够以 ALTER SEQUENCE 语句修改序列的定义，或以 RENAME SEQUENCE 语句重命名该序列。

下列语句将对 cust_seq 序列对象的 Alter 权限授予用户 mark：

```
GRANT ALTER ON cust_seq TO mark;
```

## Select 权限

您可将对序列的 Select 权限授予另一用户或角色。Select 使得指定的用户和角色能够在 SQL 语句中使用 ***sequence*.CURRVAL** 和 ***sequence*.NEXTVAL** 表达式来读取或（分别地）增大序列的值。

下列语句将对 cust_seq 序列对象的 Select 权限授予用户 mark：

```
GRANT SELECT ON cust_seq TO mark;
```

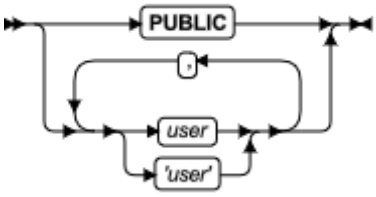
## ALL 关键字

您可指定 ALL 关键字来将对序列对象的 Alter 和 Select 权限都授予另一用户或角色，或用户或角色的列表。

## 用户列表

您可将权限授予个别的用户或授予用户的列表。您还可指定 PUBLIC 关键字来给所有用户授予权限。

用户列表




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *user* | 您正在授予权限的用户或角色的登录名 | 必须是有效的授权标识符 | 所有者名称 |

下列示例在不符合 ANSI 的数据库中，将对 table1 的 Insert 表级权限授予用户 mary：

```
GRANT INSERT ON table1 TO mary;
```

在符合 ANSI 的数据库中，如果您不包括引号作为括起 ***user*** 的限定符，则以大写字母存储用户名。

您可指定 ALL 关键字来将对序列对象的 Alter 和 Select 权限都授予另一用户或角色，或用户或角色的列表。

## 角色名称

您可使用 GRANT 语句来将一个或多个用户（或者所有用户，使用 PUBLIC 关键字）与可以描述它们行为的 ***role*** 名称关联。在您声明并授予角色之后，您授予那个角色的权限因而授予给当前与那个角色相关联的所有用户。

角色名称


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *role* | 授予的角色，或对其授予权限的另一角色 | 必须存在。如果用引号括起来，则 *role* 区分大小写。 | 所有者名称 |

您还可将现有的角色授予另一角色。此操作将被授予角色的任何权限都授予有接收角色的所有用户。

## 将角色授予用户或另一角色

在 GRANT 语句中可使用角色之前，您必须在数据库中注册角色。要获取更多信息，请参阅 CREATE ROLE 语句。

DBA 有权限将新角色授予另一用户。如果用户得到角色 WITH GRANT OPTION，则那个用户可将该角色授予其他用户或另一角色。用户保持授予他们的角色，直到 REVOKE 语句将他们的登录名与此角色名称的关联中断。

**重要：** CREATE ROLE 和 GRANT 语句不激活角色。非缺省的角色不起作用，直到 SET ROLE 启用它。角色的授予者和被授予者可发出 SET ROLE 语句。

下列示例展示将 payables 角色授予或激活给执行应付账款功能的员工组所需要的操作。首先，DBA 创建角色 payables，然后将它授予 maryf。

```
CREATE ROLE payables;
GRANT payables TO maryf WITH GRANT OPTION;
```

DBA 或 maryf 可以下列语句激活该角色：

```
SET ROLE payables;
```

用户 maryf 有 WITH GRANT OPTION 权限来将 payables 授予其他支付账款的员工。

```
GRANT payables TO charly, gene, marvin, raoul;
```

如果您将一个角色授予权限给另一角色，则接受的角色具有已经授予给两个角色的权限的组合集。下列示例将角色 petty_cash 授予角色 payables：

```
CREATE ROLE petty_cash;
SET ROLE petty_cash;
GRANT petty_cash TO payables;
```

在成功地执行所有这些语句之后，如果用户 raoul 使用 SET ROLE 语句来将 payables 作为其当前角色，那么（任何 REVOKE 操作的作用除外）他持有下列访问权限的组合集：

- 授予 **payables** 角色的权限
- 授予 **petty_cash** 角色的权限
- 单个地授予 **raoul** 的权限
- 授予 PUBLIC 的权限

如果您尝试将角色授予自己，或者直接地或者间接地，数据库服务器都会生成错误。（然而，如果要获取对此规则的重要例外的信息，请参阅 DBSECADM 子句 的描述。）

如果您在将角色分配给另一角色的 GRANT 语句中包括 WITH GRANT OPTION 关键字，则数据库服务器还生成错误。

## 将权限授予角色

您可将表级和例程级访问权限授予角色，如果您有权限来将这些相同的权限授予登录名或 PUBLIC 的话。您还可将类型级权限授予角色。角色不可持有数据库级权限。

**重要：** 用户定义的角色的范围（以及 GRANT 语句分配给该角色的自主访问权限的范围）为当前数据库。当 GRANT DEFAULT ROLE 或 SET ROLE 语句激活角色时，该角色及其权限仅在当前数据库生效。作为安全预防措施，用户仅从角色获得的自主访问权限，不可通过视图或通过触发器的动作提供对当前数据库之外的表的访问。

与授予用户相比，给角色授予权限的语法受到更多限制：

- 您可指定 AS ***grantor*** 子句。

  以这种方式，有该角色的任何用户都可取消这些相同的权限。要获取更多信息，请参阅 AS grantor 子句。

- 你不可包括 WITH GRANT OPTION 子句。

  反过来，角色不可将相同的访问权限授予另一用户。

此样例将 supplier 表上的 Insert 权限授予角色 payables：

```
GRANT INSERT ON supplier TO payables;
```

已经被授予 **payables** 角色的任何用户，以及通过发出 SET ROLE 语句成功地激活它的用户，现在可插入行到 **supplier** 内。

## 授予缺省的角色

DBA 或数据库的所有者（缺省情况下，用户 **gbasedbt**）可以 GRANT DEFAULT ROLE 语句为一个或多个用户，或为 PUBLIC 定义***缺省的角色***。当该用户连接到数据库时，激活一个缺省的角色。不需要 SET ROLE 语句来激活缺省的角色。

如果用户通过客户端应用访问数据库，该应用不可更改访问权限也不可设置角色，则缺省的角色非常有用。

缺省的角色可为被分配了那个角色的所有用户指定一系列访问权限，如下列示例中所示：

```
CREATE ROLE accounting;
GRANT ALTER, INSERT, SELECT ON stock TO accounting;
GRANT DEFAULT ROLE accounting TO mary, asok, vlad;
```

最后的语句提供给用户 **mary**、**asok** 和 **vlad** 以 **accounting** 作为他们的缺省的角色。如果这些用户中的任何人连接到数据库，则那个用户激活 **accounting** 角色持有的任何权限，除了用户作为个别用户或作为 PUBLIC 已经拥有的任何角色之外。

该角色必须已经存在，且该用户必须有访问权限来设置该角色。如果先前不曾将该角色授予用户，它会被作为设置缺省的角色的一部分被授予。

如果既未为用户也未为 PUBLIC 定义缺省的角色，那么不设置角色，且用户的现有权限有效。

下列示例展示缺省的角色是如何可分配给所有用户的：

```
DATABASE hrdb;
CREATE ROLE emprole;
GRANT CONNECT TO PUBLIC;
GRANT SELECT ON emptab TO emprole;
GRANT emprole TO PUBLIC;
GRANT DEFAULT ROLE emprole TO PUBLIC;
```

**注：** 使用 GRANT DEFAULT ROLE 是在 sysdbopen( ) 过程中发出 SET ROLE 语句的一种备用方法。然而，当用户建立连接时，使用 sysdbopen( ) 过程定义的缺省的角色优先于任何其他角色。

为用户或为 PUBLIC 更改缺省的角色仅影响新的数据库连接。在当前分配的角色之下，现有的连接继续运行。如果将缺省的角色授予了 ***user***，且另一缺省的角色授予了 PUBLIC，则在连接的时候，授予 ***user*** 的缺省的角色优先。

不可将缺省的角色分配给另一角色。因为角色不是跨数据库定义的，必须为每一数据库分配缺省的角色。在 GRANT DEFAULT ROLE 语句中的 TO 关键字之后，除了 ***user-list*** 之外没有有效的选项。如果您尝试包括 AS ***grantor*** 子句或 WITH GRANT OPTION 子句，则数据库服务器发出错误。

## 授予 EXTEND 角色

如果 IFX_EXTEND_ROLE 配置参数设置为 ON 或 1，仅持有 EXTEND 角色的用户（以及还持有对数据库的 Resource 权限和对用于编写 UDR 的编程语言的 Usage 权限的用户）可创建或删除以 C 或 Java™ 外部语言编写的 UDR，这些语言可支持共享库。

数据库服务器管理员（DBSA），缺省情况下用户 **gbasedbt**，可以 GRANT EXTEND TO ***user-list*** 语句将 EXTEND 角色授予一个或多个用户或授予 PUBLIC。

由于 EXTEND 为内建的角色，所以 SET ROLE 语句不需要 EXTEND 角色来使之生效。用户持有 EXTEND 角色就足够了，无需使用 SET ROLE 来启用它。

例如，假设用户 **max** 持有对数据库的 Resource 权限，且已经通过 GRANT USAGE ON LANGUAGE C 语句被授予了对 C 语言的 Usage 权限。下列语句将 EXTEND 角色授予用户 **max**：

```
GRANT EXTEND TO 'max';
```

此语句使得用户 **max** 能够创建或删除以 C 语言编写的 UDR，而不要求 **max** 发出 SET ROLE EXTEND 语句。（此处引号在授权标识符 **max** 中保持小写字母。）然而，在用户 **max** 可创建或删除以 Java 语言编写的 UDR 之前，有效的 GRANT USAGE ON LANGUAGE JAVA 语句的 TO

子句必须指定或者 'max'，或者 PUBLIC，或者 **max** 持有的用户定义的角色的名称（以及 **max** 已经使用 SET ROLE 语句来指定为他的当前角色）。

在不需要此安全特性的数据库中，DBSA 可通过在 ONCONFIG 文件中将 IFX_EXTEND_ROLE 配置参数设置为 OFF 或 0 来禁用对谁可创建或删除外部 UDR 的限制。当 IFX_EXTEND_ROLE 设置为 OFF 或 0 时，任何持有 Resource 权限的用户（以及还持有对以其编写该 UDR 的编程语言的 Usage 权限的用户）都可创建或删除外部 UDRs。

对任何要创建或删除外部 UDR 的用户而言，对数据库的 Resource 权限和对外部语言的 Usage 权限都是需要的，不管 IFX_EXTEND_ROLE 配置参数设置如何，或该用户是否持有 EXTEND 角色。用户 **gbasedbt**、DBA 或任何已经获得 Usage 权限 WITH GRANT OPTION 的用户都可将对 SPL、C 和 Java 语言的 Usage 权限授予 PUBLIC。要获取关于授予 Resource 权限的信息，请参阅 数据库级权限。要获取关于授予对编程语言的 Usage 权限的信息，请参阅 语言级权限。

## WITH GRANT OPTION 关键字

WITH GRANT OPTION 关键字将权限或角色传递给 user，随同将相同的权限或角色授予其他用户的权利。

请您创建以您开头的权限链，并扩展至 ***user*** 以及 ***user*** 随后将授予权限的权利传递给的任何用户。如果您包括 WITH GRANT OPTION，则可不再控制权限的分发。

下列示例将对 cust_seq 序列对象的 Alter 和 Select 权限授予用户 mark，随同将那些权限授予其他用户的能力：

```
GRANT ALL ON cust_seq TO mark WITH GRANT OPTION;
```

如果您从 ***user*** 将使用 WITH GRANT OPTION 授予的权限取消，则切断该权限链。即当您从 ***user*** 取消权限时，自动地取消了从 ***user*** 或从 ***user*** 创建的链获取了权限的所有用户的权限（除非其他用户给 ***user***、或从 ***user*** 获得授权的用户授予了相同的权限集）。

下列示例展示此情况。作为表 items 的所有者，您发出下列语句来将访问权限授予用户 mary：

```
REVOKE ALL ON items FROM PUBLIC;
        GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION;
```

用户 mary 使用其权限来授予用户 cathy 和 paul 访问此表的权限：

```
GRANT SELECT, UPDATE ON items TO cathy;
        GRANT SELECT ON items TO paul;
```

稍后，您从用户 mary 取消对 items 表的访问权限：

```
REVOKE SELECT, UPDATE ON items FROM mary;
```

此单一语句有效地取消用户 **mary**、**cathy** 和 **paul** 对 **items** 表的所有权限。如果您想要与另一用户创建权限链作为权限源，请使用 AS ***grantor*** 子句。

在 GBase 8s 中，WITH GRANT OPTION 关键字仅对用户有效。如果将角色作为权限或另一角色的被授权者，则为无效。您不可在语句中指定 WITH GRANT OPTION 给 PUBLIC 组授予权限。

“数据库服务器管理员”不可在 GRANT EXTEND 或 GRANT DBSECADM 语句中包括 WITH GRANT OPTION 关键字。DBSA 不可将授予内建的 EXTEND 或 DBSECADM 角色的权限分派给另一用户。如果多个用户需要这些权限，则应在安装数据库服务器时将他们包括在 **DBSA** 组中。

除了 GRANT DBSECADM 语句之外，GRANT 语句的其他安全管理选项都不支持 WITH GRANT OPTION 关键字。要获取关于这些语句及其语法的详细信息，请参阅 安全管理选项。

## AS grantor 子句

当您将自主访问权限授予其他用户、角色或 PUBLIC 时，在缺省情况下，您是可取消那些权限的用户。AS ***grantor*** 子句使您建立另一用户作为您正在授予权限的源。

当您使用 AS ***grantor*** 子句时，AS ***grantor*** 子句中提供的登录名取代相应系统目录表中您的登录名。如果您有对该数据库的 DBA 权限，则您可使用此子句。

在您使用此子句之后，仅指定的 ***grantor*** 可取消当前 GRANT 操作产生的影响。即使 DBA 也不可取消权限，除非那个 DBA 作为授予了该权限的用户被罗列在系统目录表中。

下列示例展示此情况。您是 DBA，且您将对 items 表的所有权限授予用户 tom，随同授予所有权限的权利：

```
REVOKE ALL ON items FROM PUBLIC;
GRANT ALL ON items TO tom WITH GRANT OPTION;
```

下一示例展示不同的情况。您还可将 Select 和 Update 权限授予用户 jim，但您指定作为用户 tom 授予了该权限。（数据库服务器在 systabauth 系统目标表中的记录显示用户 tom 为那些权限的授予者，而不是您。）

```
GRANT SELECT, UPDATE ON items TO jim AS tom;
```

随后，您决定从用户 user tom 取消对 items 表的权限，于是您发出下列语句：

```
REVOKE ALL ON items FROM tom;
```

然而，如果不是如此，您试图以类似的语句从用户 jim 取消权限。则数据库服务器返回错误，如下例所示：

```
REVOKE SELECT, UPDATE ON items FROM jim;

580: Cannot revoke permission.
```

因为数据库服务器记录显示原始的授权者为用户 **tom**，且您不可取消该权限，所以您收到错误。虽然您是 DBA，但您不可取消另一用户授予了的权限。

在 GRANT DEFAULT ROLE 语句内，AS ***grantor*** 子句无效。

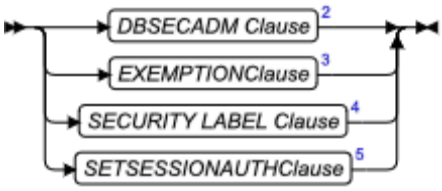
要了解 AS ***grantor*** 子句处需要的上下文，而不是可选项，请参阅 将 Execute 权限授予 PUBLIC。

## 安全管理选项

结合 REVOKE 语句，GRANT 通过指定哪些用户或角色持有访问数据库内的数据库或对象所需要的权限， 支持 GBase 8s 的自主访问控制（DAC）数据安全特性。

GRANT 语句的“安全管理选项”，就像 REVOKE 语句的对应选项，支持附加的数据安全特性集，称为基于标签的访问控制（LBAC）。 这些特性使得 GBase 8s 能够允许或拒绝对访问受保护的数据，这是基于将包含在数据对象中的行安全标签或列安全标签与已经授予正在搜索访问的用户的用户安全标签和其他凭证相比较来完成的。

安全管理选项



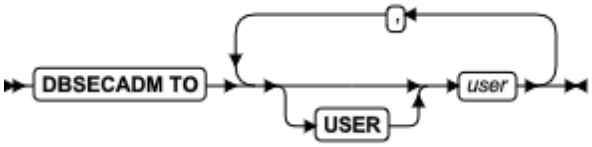

这些 GRANT 语句安全管理选项的使用限于：

- 仅”数据库服务器管理员（DBSA），缺省为用户 **gbasedbt**，或（在 UNIX™ 上）**DBSA** 组的成员，或（在 Windows™ 上）**Gbasedbt-Admin** 组的成员，可使用 GRANT DBSECADM 语句来授予 DBSECADM 角色。
- 仅持有 DBSECADM 角色的用户可发出 GRANT EXEMPTION、GRANT SECURITY LABEL 或 GRANT SETSESSIONAUTH 语句，或相应的 REVOKE 语句。

## DBSECADM 子句

GRANT DBSECADM 语句使得被授予 DBSECADM 角色的用户能够发出 DDL 语句，可创建、更改、重命名或删除安全对象，包括安全策略、安全标签和安全组件。

DBSECADM 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *user* | 被授予该角色的用户 | 必须为用户的权限标识符 | 所有者名称 |

DBSECADM 角色是仅 DBSA 可授予的内建的角色。用户定义的角色的范围是创建该角色所在的数据库，与用户定义的角色不同，BSECADM 角色的范围是 GBase 8s 实例的所有数据库。在同一服务器的其他数据库中，DBSA 不必重新发出 GRANT DBSECADM 语句，就像 GBase 8s 的所有内建的角色一样，当授予 DBSECADM 角色时，即启用该角色，无需通过 SET ROLE 语句激活，且保持有效直到被取消为止。

仅持有 DBSECADM 角色的用户可发出下列创建或修改安全对象的 SQL 语句：

- ALTER SECURITY LABEL COMPONENT
- CREATE SECURITY LABEL

- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- DROP SECURITY LABEL
- DROP SECURITY LABEL COMPONENT
- DROP SECURITY POLICY
- RENAME SECURITY LABEL
- RENAME SECURITY LABEL COMPONENT
- RENAME SECURITY POLICY

仅持有 DBSECADM 角色的用户可使用下列 SQL 语句来引用那些被安全策略保护的表：

- ALTER TABLE ... ADD SECURITY POLICY
- ALTER TABLE ... ADD ... IDSSECURITYLABEL [DEFAULT ***label***]
- ALTER TABLE ... ADD ... [COLUMN] SECURED WITH
- ALTER TABLE ... DROP SECURITY POLICY
- ALTER TABLE ... MODIFY ... [COLUMN] SECURED WITH
- ALTER TABLE ... MODIFY ... DROP COLUMN SECURITY
- CREATE TABLE ... COLUMN SECURED WITH
- CREATE TABLE ... IDSSECURITYLABEL [DEFAULT ***label***]
- CREATE TABLE ... SECURITY POLICY

不持有 DBSECADM 角色的用户也不可发出下列 GRANT 和 REVOKE 语句：

- GRANT EXEMPTION
- GRANT SECURITY LABEL
- GRANT SETSESSIONAUTH
- REVOKE EXEMPTION
- REVOKE SECURITY LABEL
- REVOKE SETSESSIONAUTH

可跟在 TO 关键字之后的 USER 关键字是可选的，且没有作用，但 DBSA 在 GRANT DBSECADM 中指定的任何授权标识符都必须是单个用户的标识符，而不是角色或 PUBLIC 组的标识符。

***user*** 可为发出此 GRANT DBSECADM 语句的 DBSA。这是对通用限制的一个重要例外，通用限制是指 GRANT 语句的 TO 子句（就如 REVOKE 语句中的 FROM 子句一样）不可显式地引用发出该语句的用户的授权标识符。与其他角色不同，GRANT 语句可指定访问权限、用户安全标签和规则的豁免，您可给自己授予 DBSECADM 角色，如果您是用户 **gbasedbt**，或 DBSA 组的成员或（在 Windows™ 上）如果您是 **Gbasedbt-Admin** 组的成员。

在下列示例中，DBSA 将 DBSECADM 角色授予用户 niccolo：

```
GRANT DBSECADM TO niccolo;
```

如果此语句成功地执行，则用户 niccolo 可执行以上罗列的 LBAC 操作，如果 niccolo 还持有对数据库和对于那些 SQL 语句引用的数据库对象的自主访问权限的话。

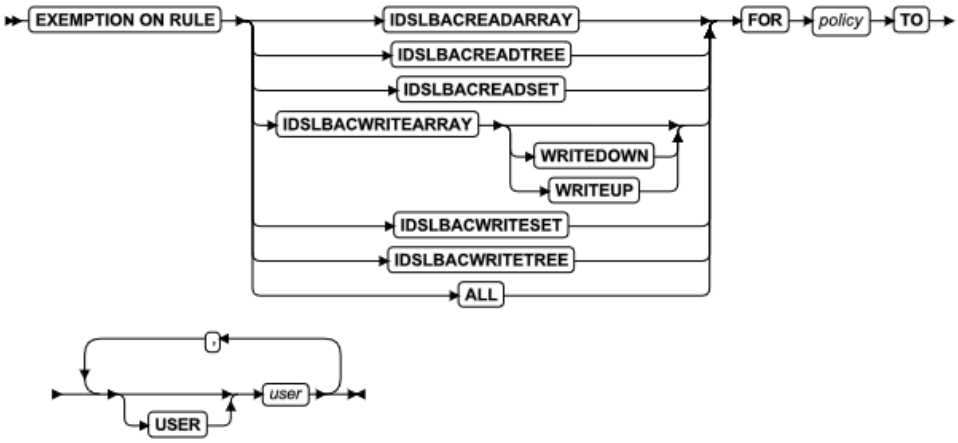
授予用户 DBSECADM 角色之后，仅 DBSA 可取消它。

要了解对 LBAC 安全对象的讨论，请参阅您的 GBase 8s 安全指南。

## EXEMPTION 子句

GRANT EXEMPTION 语句通过禁用指定的安全策略的一个或全部规则来修改指定用户（或用户列表）的安全凭证

EXEMPTION 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *policy* | 从中授予豁免的安全策略 | 在数据库中必须存在 | 标识符 |
| *user* | 要授予其豁免的用户 | 必须为用户的授权标识符 | 所有者名称 |

仅持有 DBSECADM 角色的用户可发出 GRANT EXEMPTION 语句。

### *授予豁免的规则*

跟在 ON 关键字之后的关键字指定授予豁免的安全策略（其标识符跟在 FOR 关键字之后）的预定义 LBAC 访问规则。当授予其豁免的用户访问由指定的安全策略保护的表时，授予其豁免的访问规则不再适用。 要了解对与安全策略相关的读访问和写访问的预定义规则的描述，请参阅章节 与安全策略相关的规则。

下列 GRANT EXEMPTION 语句的关键字标识特定的 IDSLBACRULES 规则，此语句可从此豁免用户：

- IDSLBACREADARRAY 从指定的安全策略的 **IDSLBACREADARRAY** 规则豁免用户。那个规则要求每一用户安全标签的数组组件必须大于或等于相应的数据行安全标签的数组组件。
- IDSLBACREADSET 从指定的安全策略的 **IDSLBACREADSET** 规则豁免用户。那个规则要求用户安全标签的每一集合组件必须包括该数据行安全标签的集合组件。
- IDSLBACREADTREE 从指定的安全策略的 **IDSLBACREADTREE** 规则豁免用户。那个规则要求用户安全标签的每一树组件必须至少包括该数据行安全标签的树组件中的元素之一，或一个这样元素的其他祖先。

- IDSLBACWRITEARRAY WRITEDOWN 从指定的安全策略的 **IDSLBACWRITEARRAY** 规则的一个方面豁免用户。那个规则要求用户安全标签的每一数组组件必须等于数据行安全标签的数组组件。持有此豁免的用户可写到其数组组件级别低于用户的标签中的级别的行。然而，该用户不可写到其数组组件级别标签高于该用户的标签中级别的行。
- IDSLBACWRITEARRAY WRITEUP 从指定的安全策略的 **IDSLBACWRITEARRAY** 规则的一个方面豁免用户。持有此豁免的用户可写到其数组组件级别高于该用户的标签中的级别的行。然而，该用户不可写到其标签数组组件级别低于该用户的标签中的级别的行。
- IDSLBACWRITEARRAY（无 WRITEDOWN 或 WRITEUP 关键字）从指定的安全策略的 **IDSLBACWRITEARRAY** 规则豁免用户。持有此豁免的用户可写到行，不管相应的行标签的数据组件。
- IDSLBACWRITESET 从指定的安全策略的 **IDSLBACWRITESET** 规则豁免用户。那个规则要求该用户安全标签的每一集合组件必须包括该数据行安全标签的集合组件。
- IDSLBACWRITETREE 从指定的安全策略的 **IDSLBACWRITETREE** 规则豁免用户。那个规则要求该用户安全标签的每一树组件必须包括该数据行安全标签的树组件中至少一个元素，或一个这样元素的一个祖先。
- ALL 从指定的安全策略的所有 **IDSLBACRULES** 规则豁免用户。这种形式的豁免要求将数据加载到受保护的表内。

在下列示例中，DBSECADM 从 MegaCorp 安全策略的所有规则授予豁免给用户 manoj 和 sam：

```
GRANT EXEMPTION ON RULE ALL FOR MegaCorp TO manoj, sam;
```

## *安全策略和豁免的被授予者*

豁免仅适用于其名称跟在 FOR 关键字之后的安全策略的规则。受保护的表可有多个安全标签，但最多只能有一个安全策略。

如果在数据库中不存在指定的策略，则 GRANT EXEMPTION 语句失败并报错。

可跟在 TO 关键字之后的 USER 关键字是可选的，且没有作用，但在 GRANT EXEMPTION 语句中指定的任何授权标识符都必须是单个用户的标识符，而不是角色的标识符。 此 ***user*** 不可为发出同一 GRANT EXEMPTION 语句的 DBSECADM。

在下列示例中，DBSECADM 从 MegaCorp 安全策略的规则 IDSLBACREADARRAY 授予用户 lynette 豁免：

```
GRANT EXEMPTION ON RULE IDSLBACREADARRAY FOR MegaCorp TO lynette;
```

此豁免绕过指定策略的安全标签的所有数组组件的读访问规则。

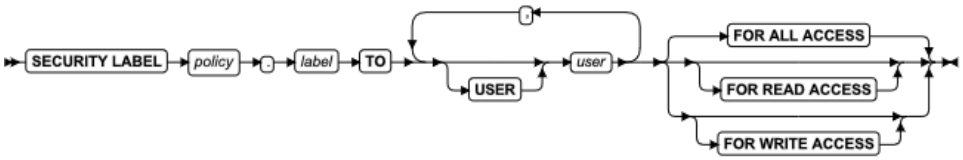
当 GRANT EXEMPTION 语句成功地授权豁免给用户时，该数据库服务器更新系统目录的 **syssecpolicyexemptions** 表来注册新的豁免（或多个豁免，如果在 TO 关键字之后罗列了几个用户的话）。

要了解 LBAC 安全对象的讨论，请参阅您的 GBase 8s 。

# SECURITY LABEL 子句

GRANT SECURITY LABEL 语句将安全标签授予用户或用户列表。

SECURITY LABEL 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 现有安全标签的名称 | 必须存在作为指定的安全 *policy* 的标签 | 标识符 |
| *policy* | 此 *label* 的安全策略 | 必须在该数据库中已存在 | 标识符 |
| *user* | 要将该标签授予的用户 | 必须为用户的授权标识符 | 所有者名称 |

仅持有 DBSECADM 角色的用户可发出 GRANT SECURITY LABEL 语句。

安全标签是始终与安全策略关联的数据库对象。那个策略定义构成安全标签的有效的安全组件的集合。该标签存储安全策略的每一组件的一个或多个值的集合。

DBSECADM 可将安全标签与下列实体关联：

- 数据库表的一列，***列安全标签***可保护该列
- 数据库表的一行，***行安全标签***可保护该行
- 用户，其***用户安全标签***（以及已经授予该用户的安全策略的规则的任何豁免）称为该用户的***安全凭证***。

当持有特定安全策略的安全标签的用户尝试访问同一安全策略的行安全标签所保护的行，数据库服务器将该用户安全标签中值的集合与该行安全标签中值的集合进行对比，来决定该用户是否该被允许访问该数据。类似地，在决定用户的凭证是否该允许其访问受保护的列时，LBAC 考虑用户安全标签与列安全标签。

GRANT SECURITY LABEL 语句是 DBSECADM 将用户与安全标签相关联的机制。通过仅 DBSECADM 可执行的 CREATE TABLE 或 ALTER TABLE 语句的选项，受保护的表中的数据值与行安全标签或列安全标签相关联，而不是通过 GRANT SECURITY LABEL 语句）。

可跟在 TO 关键字之后的 USER 关键字是可选的，而且没有作用，但在 GRANT SECURITY LABEL 语句中指定的任何授权标识符都必须是单个用户的标识符，而不是角色的标识符。

## *访问规范*

授予其安全标签的用户列表可可选地后跟关键字，这些关键字指定对该标签的安全策略保护的数据的访问类型

- FOR WRITE ACCESS

  这些关键字将标签限定为 **IDSLBACRULES**，即 **IDLSBACWRITEARRAY**、**IDLSBACWRITESET** 和 **IDLSBACWRITETREE** 的写访问规则。这些规则影响对受保护数据的 INSERT、DELETE 和 UPDATE 操作。

- FOR READ ACCESS

  这些关键字将标签限定为 **IDSLBACRULES**，即 **IDLSBACWREADARRAY**、**IDLSBACREADSET** 和 **IDLSBACREADTREE** 的读访问规则。这些规则影响对受保护数据的 SELECT、DELETE 和 UPDATE 操作。

- FOR ALL ACCESS

  这些关键字将标签应用到上列所有读和写访问规则。如果 GRANT SECURITY LABEL 语句不包括 FOR ... ACCESS 规范，则此选项作为缺省项生效。

要获取更多关于这些基于标签的读和写访问的 IDSLBACRULES 规则的信息，请参阅 与安全策略相关的规则。要获取关于可以为特定安全策略授予对这些规则的豁免的信息，请参阅 授予豁免的规则。

如果授予用户的读访问安全标签与写访问安全标签不同，那么为这些安全标签组件赋予的值必须服从下列这些规则：

- 对于类型 ARRAY 的安全标签组件，在两个安全标签中的值必须相同。
- 对于类型 SET 的安全标签组件，在用于 WRITE 访问的安全标签中给定的值，必须是在用于 READ 访问的安全标签中给定值的子集。如果所有的值相同，则视为子集，且是允许的。
- 对于类型 TREE 的安全标签组件，用于写访问的安全标签的树组件中的每个元素，必须是用于读访问的安全标签的树组件中的一个元素或一个元素的后代。

总之，当 DBSECADM 尝试将读访问的安全标签授予已持有写访问的安全标签的用户时，或者反之，则读标签不可比写标签更具限制性。否则，GRANT SECURITY LABEL 语句失败并报错。

对于同一安全策略，可将不多于两个标签授予用户。如果授予同一策略的两个标签，则一个标签必须为读访问，且另一个为写访问。如果 DBSECADM 尝试将读访问的安全标签授予用户，而该用户已持有基于同一安全策略的读访问的安全标签，则 GRANT SECURITY LABEL 语句失败并报错。如果两个标签都是写访问的且基于同一安全策略，则导致类似的失败。

在这两种情况下，在可将相同的访问模式和相同的安全策略授予第二个标签之前，必须通过 REVOKE SECURITY LABEL 语句显式地取消第一个安全标签。此规则的唯一例外情况是，如果两个标签都指定组件元素的同一个值。因为在此情况下，两个标签在功能上相同，且不发生错误。

## *用户安全标签的规则*

下列规则影响那些通过 GRANT SECURITY LABEL 语句授予用户的安全标签：

- ***user*** 不可为发出此 GRANT SECURITY LABEL 语句的 DBSECADM。
- 没有安全标签的用户有 NULL 或零标签。没有安全标签的用户不可访问受保护表中的数据，除非该用户持有对该策略的必要的豁免。
- 在缺省情况下，受保护表的 IDSSECURITYLABEL 列不可有 NULL 值。没有安全标签的用户不可将数据插入到带有行保护的表内，即使该用户已被授予了对该安全策略的必要的豁

免，除非在 INSERT 语句中显式地指定该行标签。要了解如何在 INSERT 语句中显式地指定安全标签，请参阅 安全标签支持函数。

- 对下列数据库表类型，用户安全标签不起作用，因为这些表不可由安全策略来保护：
  - “虚拟表接口”表，
  - 带有“虚拟表接口”索引的表，
  - 在类型表层级中的表，
  - 临时表。

## *授予用户安全标签的示例*

下列三个语句分别地创建三个名为 level、compartments 和 groups 的安全标签组件：

```
CREATE SECURITY LABEL COMPONENT
    level ARRAY ['TS','S','C','U'];

CREATE SECURITY LABEL COMPONENT
    compartments SET {'A','B','C','D'};

CREATE SECURITY LABEL COMPONENT
    groups TREE ('G1' ROOT,
                 'G2' UNDER ROOT,
                 'G3' UNDER ROOT);
```

下列语句基于上面三个组件创建名为 secPolicy 的安全策略：

```
CREATE SECURITY POLICY secPolicy COMPONENTS
    level, compartments, groups;
```

下列语句创建名为 secLabel1 的安全标签：

```
CREATE SECURITY LABEL secPolicy.secLabel1
    COMPONENT level 'S',
    COMPONENT compartments 'A', 'B',
    COMPONENT groups 'G2';
```

下列语句创建名为 secLabel2 的安全标签：

```
CREATE SECURITY LABEL secPolicy.secLabel2
    COMPONENT level 'S',
    COMPONENT compartments 'B',
    COMPONENT groups 'G2';
```

下列语句创建名为 secLabel3 的安全标签：

```
CREATE SECURITY LABEL secPolicy.secLabel3
    COMPONENT level 'S',
    COMPONENT compartments 'A',
    COMPONENT groups 'G3';
```

下列语句创建名为 secLabel4 的安全标签：

```
CREATE SECURITY LABEL secPolicy.secLabel4
    COMPONENT level 'TS',
    COMPONENT compartments 'A',
    COMPONENT groups 'G1';
```

下列语句授予用户 sam 读访问的安全标签：

```
GRANT SECURITY LABEL secPolicy.secLabel1
    TO sam FOR READ ACCESS;
```

下列语句授予用户 sam 写访问的安全标签。由于满足以上给出的规则，此语句成功。

```
GRANT SECURITY LABEL secPolicy.secLabel2
    TO sam FOR WRITE ACCESS;
```

下列语句授予用户 lynette 读访问的安全标签：

```
GRANT SECURITY LABEL secPolicy.secLabel1
    TO lynette FOR READ ACCESS;
```

下列语句尝试授予用户 sam 写访问的安全标签。由于违反有关树组件的规则，此语句失败。

```
GRANT SECURITY LABEL secPolicy.secLabel3
    TO sam FOR WRITE ACCESS;
```

下列语句尝试授予用户 sam 写访问的安全标签。由于违反有关数组组件的规则，此语句失败。

```
GRANT SECURITY LABEL secPolicy.secLabel4
    TO sam FOR WRITE ACCESS;
```

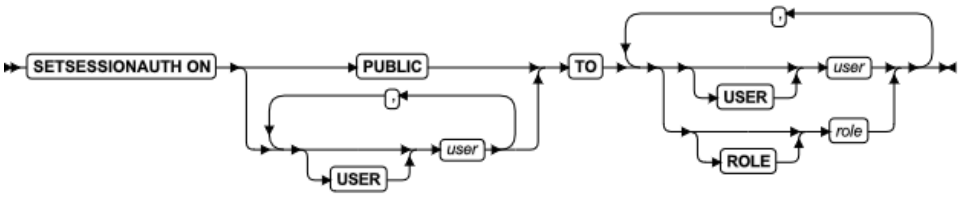
当 GRANT SECURITY LABEL 语句成功地将安全标签授予用户时，数据库服务器更新系统目录的 **sysseclabelauth** 表来注册该安全标签的新的持有者。

要了解 LBAC 安全对象的讨论，请参阅您的 GBase 8s 安全指南。

## SETSESSIONAUTH 子句

GRANT SETSESSIONAUTH 语句将 SETSESSIONAUTH 权限授予一个或多个用户或角色。此权限允许持有者使用 SET SESSION AUTHORIZATION 语句来设置会话授权为 PUBLIC 或为指定用户的列表中的任一个。

*SETSESSIONAUTH 子句*




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *role* | 要授予其该权限的角色 | 必须为角色的授权标识符 | 所有者名称 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *user* | 在 TO 关键字之后，要授予其该权限的用户。在 ON 关键字之后，表示可在 SET AUTHORIZATION 语句中指定其被授权者身份的用户。 | 必须为用户的授权标识符 | 所有者名称 |

仅持有 DBSECADM 角色的用户可授予 SETSESSIONAUTH 权限。SETSESSIONAUTH 权限和 DBA 权限都需要执行 SET AUTHORIZATION 语句。

跟在 ON 关键字之后的用户或 PUBLIC 规范指定在使用 SET SESSION AUTHORIZATION 语句时，该 SETSESSIONAUTH 权限的被授予者可使用谁的身份。此可为用户或 PUBLIC，但不可为角色。如果指定 PUBLIC，那么该权限的被授予者可使用任何数据库用户的身份。

可跟在 TO 关键字之后的 USER 和 ROLE 关键字是可选的。**user** 或 **role** 都不可为发出 GRANT SETSESSIONAUTH 语句的 DBSECADM 角色的持有者。

*下列示例授予用户 sam 将会话授权设置为用户 lynette 和 manoj 的能力：*

```
GRANT SETSESSIONAUTH ON lynette, manoj TO sam;
```

下一示例授予用户 *lynette* 将会话授权设置为 PUBLIC 的能力：

```
GRANT SETSESSIONAUTH ON PUBLIC TO lynette;
```

仅持有 DBSECADM 角色的用户可取消 SETSESSIONAUTH 权限。要了解 LBAC 安全对象的讨论，请参阅您的 GBase 8s 安全指南。

## 代理用户属性（UNIX™、Linux™）

使用 GRANT 语句的 ACCESS TO PROPERTIES 子句来将用户映射到访问 GBase 8s 资源所需要的代理用户属性。

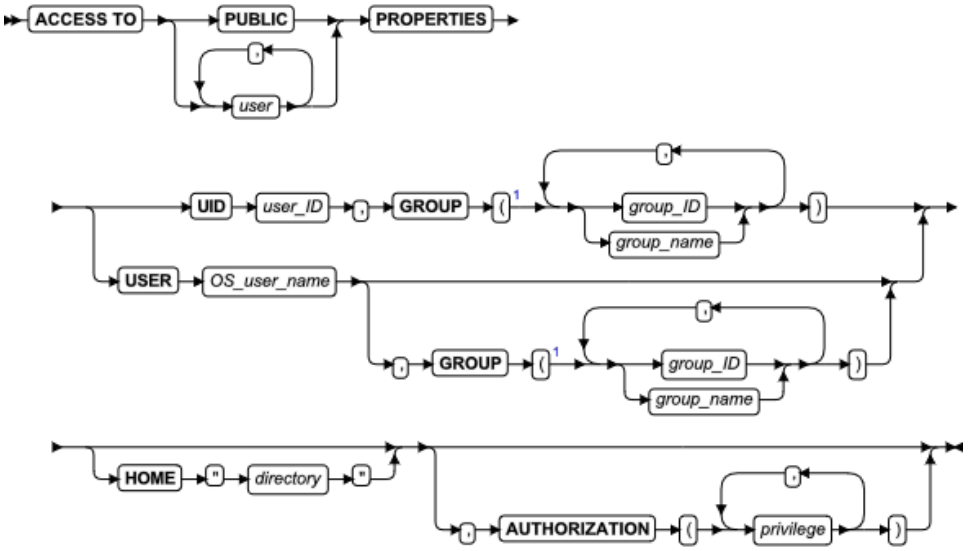
**注：**

仅 DBSA 可外部地将认证的用户映射到有效的代理用户属性。如果 USERMAPPING 配置参数设置为启用支持被映射的用户，则 DBSA 发出 GRANT ACCESS TO PROPERTIES 语句来将用户映射到与适当的授权级别对应的属性。

被映射的用户可以代理用户属性连接到 GBase 8s ，如果他们以可插拔的认证模块（PAM）或单点登录（SSO）认证的话。

被映射的用户可以代理用户属性连接到 GBase 8s ，如果他们以可插拔的认证模块（PAM）认证的话。

ACCESS TO PROPERTIES 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *directory* | 存储用户文件的目录的路径名称。 | 长度不超过 255 字节，且必须符合您的操作系统的规则。*directory* 还必须：<br>• 属于被映射的 ***user_ID*** 和 ***group_ID***<br>• 有所有者的读、写和执行许可<br>• 没有 PUBLIC 写许可 | 引用字符串 |
| *group_ID* | 您想要将 *user* 映射到的组标识符编号。*group_id* 值的列表或您指定的值必须括在括号中。 | *group_ID* 不可为：<br>• 带有管理权限（DBSA、DBSSO、AAO 和 BARGROUP）的组 ID<br>• 组 0（**root**，有时称为 **wheel** 或 **system**）<br>• 与组 **bin** 或组 **sys** 关联的组 ID<br>组 ID 必须在 /etc/gbasedbt/allowed.surrogates 文件中存在。 | 精确数值 |
| *group_name* | 现有的操作系统组的名称，该组有您想要映射 *user* 到的许可。*group_name* 值的列表必须括在括号中。 | 长度不可超过 32 字节。<br>组名必须在 /etc/gbasedbt/allowed.surrogates 文件中存在。 | 所有者名称 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *privilege* | 分配给 *user* 的管理权限。有效值如下：<br>• DBSA<br>• DBSSO<br>• AAO<br>• BARGROUP<br>*privilege* 值或多个值必须括在括号中。 | USERMAPPING 配置参数必须设置为 ADMIN 来以 AUTHORIZATION 关键字授予服务器管理权限。 | 引用字符串 |
| *user* | 您正在映射到用户属性的指定用户的授权标识符。 | 必须为认证的授权标识符 | 所有者名称 |
| *user_ID* | 您想要将 *user* 映射到的那个用户标识符编号。 | *user_ID* 不可为属于用户 root 或用户 gbasedbt 的用户。<br>用户 ID 必须在 /etc/gbasedbt/allowed.surrogates 文件中存在。 | 精确数值 |
| *OS_user_name* | GBase 8s 主计算机上的现有的 OS 用户账号的名称，有您想要将 *user* 映射的许可。 | 必须符合您的操作系统的规则。<br>用户名必须在 /etc/gbasedbt/allowed.surrogates 文件中存在。 | 所有者名称 |

**用法**

最佳实践是将 ***user*** 映射到特定的 OS 用户名，保留该用户名仅作为代理用户身份。您可以 GROUP 关键字添加与代理用户身份相关联的组，并以 HOME 关键字更改 home 目录。如果操作系统管理员已在 /etc/gbasedbt/allowed.surrogates 文件中指定了可接受的代理，则您仅可将用户映射到指定的 OS 用户或组。

如果您将 ***user*** 映射到用户 ID 编号，那么请记住不要以相同的编号在 GBase 8s 主计算机上创建用户账号。

USERMAPPING 配置参数必须设置为 ADMIN，以便以 ADMINISTRATOR 关键字分配 ***user*** 一服务器管理权限。

**注：**

不推荐使用此 AUTHORIZATION 子句（以及 ALTER USER、CREATE USER 或 CREATE DEFAULT USER 语句的 AUTHORIZATION 子句）。不同的语法将在未来的版本中支持角色分离。

不可在同一语句中同时使用 PUBLIC 与 AUTHORIZATION 关键字，因为不可将服务器管理员权限授予 PUBLIC 组。

以 HOME 关键字指定用户文件的目录是可选的，但在有些情况下，强烈建议这样做。当将一个在外部认证了的用户映射到代理用户名，但未在 GRANT ACCESS TO 语句中指定 HOME 目录时，被映射的用户与 GBase 8s 主计算机上的用户账户有相同的 home 目录。当以不设置 home 目录的方式将用户映射到代理用户身份时，则 GBase 8s 在 $GBASEDBTDIR/users 中为用户文件创建目录。在后面这种情况下，$GBASEDBTDIR/users 中的目录名称采用 uid.***ID_number*** 的形式（例如，uid.101）。

**示例**

此章节中的语法和解释是对下列环境的示例，此处的缩写词 GID 是 ***组 ID 编号***的缩写，缩写词 UID 是 ***用户 ID 编号*** 的缩写：

- 在 GBase 8s 主计算机上，有一个有 OS 账号的用户 **fred**。用户 **fred** 以 UID 3000、GID 3000（**users**）、辅助组 200(**staff**) 和 home 目录 /home/fred 访问数据库服务器。
- 在同一台计算机上，存在用户 **dbuser** 的 OS 账户。此账户被锁定，因此 **dbuser** 不可登录。**dbuser** 仅为了代理用户映射的目的而存在，有 UID 3050、GID 4000（**ifx_user**）和 home 目录 /home/dbuser。
- 组 **ifx_user** 有 GID 4000，包括用户 **bill** 和 **eileen**。
- 建立被映射的用户的管理员知道，在 /etc/passwd （或其同义词）中没有 UID 101 的条目，且在 /etc/group（或其同义词）中没有 GID 10011 或 10101 的条目。
- 用户 **bob** 在 GBase 8s 主计算机上没有 OS 账户，但可通过 PAM 或 LDAP 认证。配置数据库服务器来通过 PAM 或 LDAP 模块接受认证。
- onconfig 文件中的 USERMAPPING 参数设置为 ADMIN。

将在外部认证了的用户映射到代理用户名称：

管理员通过发出下列 GRANT 语句将 **bob** 映射到用户 **fred** 已经存在的数据库服务器访问权限：

```
GRANT ACCESS TO bob PROPERTIES USER fred;
```

将 GBase 8s 访问授予所有在外部认证了的用户：

在此环境中，GBase 8s 主计算机上的用户 **dbuser** 账户的目的是将数据库服务器访问授权给被映射的用户。有这么一种情况，有许多被映射的用户且他们不需要知道在 home 目录中创建的用户文件，管理员可能发现将 PUBLIC 映射到 **dbuser** 代理用户身份非常高效且足够安全。管理员可以下列 GRANT ACCESS 语句，将所有认证了的用户（PUBLIC）映射到为 **dbuser** 建立的权限：

```
GRANT ACCESS TO PUBLIC PROPERTIES USER dbuser;
```

**注：** 将 PUBLIC 映射到代理用户身份是为某些被映射的用户设计的，这些用户不创建或不关注 home 目录上的用户文件，诸如在零售 Web 网站上访问 GBase 8s 数据库的顾客。如果您想要将像 dbuser 这样关注数据库服务器功能的用户映射到代理用户身份， 则推荐以指定的 home 目录分别地映射这些用户，如下例所示：

```
GRANT ACCESS TO bob PROPERTIES USER dbuser HOME "/home/dbuser/bob";
```

将在外部认证了的用户映射到 UID-GID 对：

管理员将 **bob** 映射到代理用户身份，由一个通过运行下列语句启用数据库服务器访问的 UID-GID 对构成：

GRANT ACCESS TO bob PROPERTIES UID 101, GROUP (10011);

因为未指定特定的目录，所以以名称 uid.101 创建 $GBASEDBTDIR/users 之下的目录，且此路径将被用作 home 目录。UID 101 和 GROUP (10011) 是匿名的，因为在各自的 /etc 目录中没有条目指定可访问 GBase 8s 的 UID 和 GID。

抑或，管理员可将 **bob** 映射到代理用户身份，这是一匿名 UID 和一显式组的综合体，如下例所示：

GRANT ACCESS TO bob PROPERTIES 101, GROUP (ifx_user);

由于 **ifx_user** 组包括成员 **bill** 和 **eileen**，所以该组不是匿名的。

将在外部认证了的用户映射到有服务器管理权限的代理用户身份：

在下列示例中，管理员将 DBSA 权限授予 **bob**：

```
GRANT ACCESS TO bob PROPERTIES USER fred, GROUP (ifx_user), AUTHORIZATION (dbsa);
```

将 UID 3000（**fred**）和 GID 3000（**users**）、200（**staff**）和额外的组 1000（**ifx_user**）分配给用户 **bob**。通过以不同的权限（DBSSO、AAO 或 BARGROUP）取代 dbsa，授予 **bob** 的管理角色可能不同。如果在 onconfig 文件中将 USERMAPPING 参数设置为 BASIC，那么通过此语句可能不将 DBSA 权限授予 **bob**。如果 USERMAPPING 设置为 OFF，那么 **bob** 可能根本不能连接到该数据库服务器。

**示例**

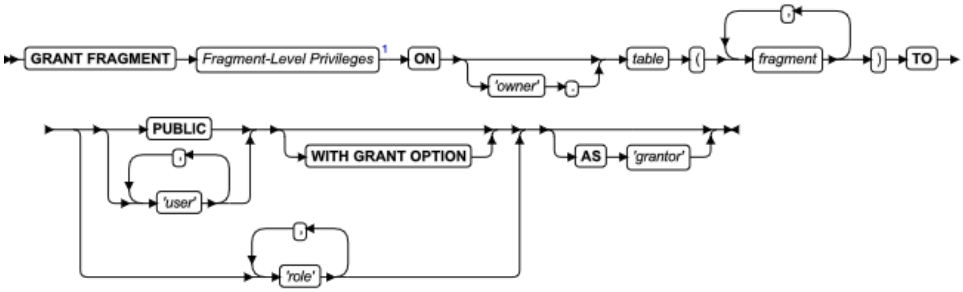
下列 GRANT 语句是有效的 ACCESS TO PROPERTIES 子句的示例，使用假定的值。这些示例未表现 ACCESS TO PROPERTIES 子句的完整语法和可能的语义。

- `GRANT ACCESS TO bob PROPERTIES USER fred;`
- `GRANT ACCESS TO PUBLIC PROPERTIES USER dbuser;`
- `GRANT ACCESS TO bob PROPERTIES USER dbuser HOME "/home/dbuser/bob";`
- `GRANT ACCESS TO bob PROPERTIES UID 101, GROUP (10011);`
- `GRANT ACCESS TO bob PROPERTIES 101, GROUP (ifx_user);`
- `GRANT ACCESS TO bob PROPERTIES USER fred, GROUP (ifx_user), AUTHORIZATION (DBSA);`

## 2.96 GRANT FRAGMENT 语句

使用 GRANT FRAGMENT 语句来对本地数据库中的表分片分配权限，如果该表是通过表达式分片的话。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *fragment* | 分片的名称 | 必须存在；不可用引号定界 | 标识符 |
| *grantor* | 可取消该权限的用户 | 同 *user* | 所有者名称 |
| *owner* | 拥有 *table* 的用户 | 必须为 *table* 的所有者 | 所有者名称 |
| *role* | 获得权限的角色 | 在 sysusers 中必须存在 | 所有者名称 |
| *table* | 对其授予分片权限的分片表 | 必须存在且必须通过表达式分片 | 标识符 |
| *user* | 要将权限授予其的那个用户 | 必须为有效的授权标识符 | 所有者名称 |

**用法**

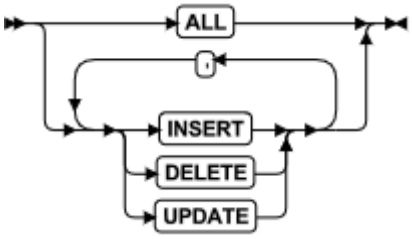
此语句是对 SQL 的 ANSI/ISO 标准的扩展。

使用 GRANT FRAGMENT 语句来将对表的个别分段的任何插入、更新和删除访问权限授予用户（或角色）。GRANT FRAGMENT 语句仅对根据基于表达式的分布方案分片的表是有效的。要了解对此类分片策略的说明，请参阅 表达式分布方案。

## 分片级权限

跟在 FRAGMENT 关键字之后的一个关键字或多个关键字指定***分片级权限***，这些权限是表级权限的一个逻辑子集：

分片级权限



这些关键字对应于下列分片级权限：

| 关键字 | 对被授予者的作用 |
| --- | --- |
| ALL | 获得对分片的插入、删除和更新权限 |
| INSERT | 可将行插入到分片内 |
| DELETE | 可从分片删除行 |
| UPDATE | 可更新在分片中的和在任何列中的行。 |

## 分片级授权的定义

在符合 ANSI 的数据库中，所有者隐式地获得对一新创建的表的所有表级权限，但其他用户得不到这些权限。

对一分片表有表权限的用户隐式地拥有对该表的所有分片的权限。这些权限不记录在 **sysfragauth** 系统目录表中。

当在不符合 ANSI 的数据库中创建分片表时，表的所有者隐式地获得该表上所有表级权限，且在缺省情况下其他用户（即 PUBLIC）获得所有分片级权限。在 **systabauth** 系统目录表中显式地记录授予 PUBLIC 的权限。

然而，如果您使用 REVOKE 语句来撤销现有的表级权限，则可使用 GRANT FRAGMENT 语句来给用户、角色或 PUBLIC 恢复指定的对该分片的一些子集的表级权限。

不论数据库是否符合 ANSI，您都可使用 GRANT FRAGMENT 语句来对通过表达式分片的表的一个或多个分片授予显式 Insert、Update 和 Delete 权限。在 **sysfragauth** 系统目录表中显式地记录 GRANT FRAGMENT 语句授予的权限。

通过 GRANT FRAGMENT 语句对表分片授予的 Insert、Update 和 Delete 权限统称为***分片级权限***或***分片级授权***。

## 分片级授权在语句验证中的作用

分片级权限使得用户能够对表分片执行 INSERT、DELETE 和 UPDATE 数据操作语言（DML）语句，即使被授权者对整个表缺乏 Insert、Update 和 Delete 权限。缺乏表权限的用户可在授权的分片中插入、删除和更新行，这是用于数据库服务器验证 DMIL 语句的算法决定的。此算法由下列检查构成：

1. 当用户执行 INSERT、DELETE 或 UPDATE 语句时，数据库服务器首先检查该用户对尝试的操作是否有必要的表权限。如果表权限存在，则继续处理该语句。
2. 如果不存在表权限，则数据库服务器检查该表是否通过表达式分片。如果该表未通过表达式分片，则数据库服务器返回错误给用户。此错误表示用户没有权限执行该语句。

3. 如果该表通过表达式分片，则数据库服务器检查该用户是否对尝试的操作持有必要的分片权限。如果该用户持有要求的分片权限，则数据库服务器继续处理该语句。如果不存在分片权限，则数据库返回错误给用户。此错误表示用户没有执行该语句的权限。

## 分片级权限的持续时间

分片级权限的持续时间与分片策略的持续时间关联在一起成为一个整体。

如果您通过 DROP TABLE 语句或通过 ALTER FRAGMENT 语句的 INIT、DROP 或 DETACH 子句删除分片策略，则还删除存在于受影响的分片的所有权限。类似地，如果您删除表的分片，则还删除存在于该分片的任何权限。

作为 ALTER FRAGMENT 语句的 DETACH 或 INIT 子句的产物而创建的表，当这些分片是分片表的一部分时，不保持以前的一个分片或多个分片拥有的权限。相反，这些表采用缺省的表权限。

如果一个定义分片权限的表更改，更改为采用轮转法策略或其他表达式策略的表，则还删除该分片权限，且该表采用缺省的表权限。

## 指定分片

您可以括在紧跟在 ON ***table*** 规范之后的括号内的名称（或名称列表）指定一个分片或一逗号分隔的分片列表。您不可使用引号来定界分片名称。如果您未包括分片，或如果指定的表的分片与您罗列的分片不相匹配，则数据库服务器发出错误。

必须通过其名称引用每一 ***fragment***。如果当您创建该分片时，未声明显式的标识符，则它的名称缺省为它所在其中的 dbspace 的名称。

通过 gspaces 实用程序成功地重命名 dbspace 之后，仅新的名称是有效的。 GBase 8s 自动地更新系统目录中现有的分片策略来替换新的 dbspace 名称，但您必须在 GRANT FRAGMENT 语句中指定新的名称来引用那个缺省名称为重命名的 dbspace 的名称的分片。

## TO 子句

跟在 TO 关键字之后的一个或多个用户或角色的列表标识被授权者。您可指定 PUBLIC 关键字来将指定的分片级权限授予所有用户。

您不可使用 GRANT FRAGMENT 来给自己授予分片级权限，既不可直接授予也不可通过角色授予。

如果您用引号括起 ***user*** 或 ***role***，则该名称区分大小写，且安全按输入形式存储。在符合 ANSI 的数据库中，如果您不使用引号括起 ***user*** 或 ***role***，则该名称以大写字母存储。

下列语句将对 part1 中的 customer 表的分片的 Insert、Update 和 Delete 权限授予用户 larry：

```
GRANT FRAGMENT ALL ON customer (part1) TO larry;
```

下列语句将对 part1 和 part2 中 customer 表的分片的 Insert、Update 和 Delete 权限授予用户 millie：

```
GRANT FRAGMENT ALL ON customer (part1, part2) TO millie;
```

要将对表的所有分片的权限授予相同的一个或多个用户，您可使用 GRANT 语句，而不使用 GRANT FRAGMENT 语句。您还可使用 GRANT FRAGMENT 语句达到此目的。

假设 customer 表通过表达式分片为三个分片，且这些分片驻留在名为 part1、part2 和 part3 的 dbspace 中。您可使用下列语句之一来将对该表的所有分片的 Insert 权限授予用户 helen：

```
GRANT FRAGMENT INSERT ON customer (part1, part2, part3) TO helen;
GRANT INSERT ON customer TO helen;
```

## 向用户或用户列表授予权限

您可将分片级权限授予单个用户或多个用户的列表。

下列语句将对 part3 中的 customer 表的分片的 Insert、Update 和 Delete 权限授予用户 oswald：

```
GRANT FRAGMENT ALL ON customer (part3) TO oswald;
```

下列语句将对 part3 中的 customer 表的分片的 Insert、Update 和 Delete 权限授予用户 jerome 和 hilda：

```
GRANT FRAGMENT ALL ON customer (part3) TO jerome, hilda;
```

## 授予权限或权限列表

当您在 GRANT FRAGMENT 语句中指定分片级权限时，您可指定一种权限、多种权限的列表或所有权限。

下列语句将对 part1 中的 customer 表的 Update 权限授予用户 ed：

```
GRANT FRAGMENT UPDATE ON customer (part1) TO ed;
```

下列语句将对 part1 中的 customer 表的 Update 和 Insert 权限授予用户 susan：

```
GRANT FRAGMENT UPDATE, INSERT ON customer (part1) TO susan;
```

下列语句将对 part1 中的 customer 表的 Insert、Update 和 Delete 权限授予用户 harry：

```
GRANT FRAGMENT ALL ON customer (part1) TO harry;
```

## WITH GRANT OPTION 子句

如同在其他 GRANT 语句中一样，WITH GRANT OPTION 关键字指定被授权者可将相同的分片级权限授予其他用户。如果 TO 子句指定 ***role*** 作为被授权者，则 WITH GRANT OPTION 是无效的。要获取附加的信息，请参阅 WITH GRANT OPTION 关键字。

下列语句将对 part3 中的 customer 表的分片的 Update 权限授予用户 george，并赋予 george 将对同一分片的 Update 权限授予其他用户的权利：

```
GRANT FRAGMENT UPDATE ON customer (part3) TO george WITH GRANT OPTION;
```

## AS grantor 子句

GRANT FRAGMENT 语句的 AS ***grantor*** 子句可指定权限的授权者。仅当您有对数据库的 DBA 权限，您才可使用此子句。当您包括 AS ***grantor*** 子句时，数据库服务器罗列指定为 ***grantor*** 的用户或角色作为 **sysfragauth** 系统目录表的 **grantor** 列中的权限的授权者。

在下一示例中，DBA 将对 part3 分片中的 customer 表的分片的 Delete 权限授予用户 martha，并使用 AS *grantor* 子句来指定罗列在 sysfragauth 中的用户 jack 作为该权限的授权者：

```
GRANT FRAGMENT DELETE ON customer (part3) TO martha AS jack;
```

在前一示例中 AS ***grantor*** 子句的作用之一是，用户 **jack** 可执行 REVOKE FRAGMENT 语句来取消 **martha** 持有的 Delete 分片级权限，如果此语句是在 **part3** 中对 **customer** 行 **martha** 的分片权限的唯一来源的话。

## 省略 AS grantor 子句

当 GRANT FRAGMENT 不包括 AS ***grantor*** 子句时，发出该语句的用户为指定的分片权限的缺省的授权者。

在下一示例中，用户将对 part3 中的 customer 表的分片的 Update 权限授予用户 fred。因为此语句未指定 AS *grantor* 子句，所以在缺省情况下，将发出该语句的用户罗列在 sysfragauth 系统目录表中作为该权限的授权者。

```
GRANT FRAGMENT UPDATE ON customer (part3) TO fred;
```

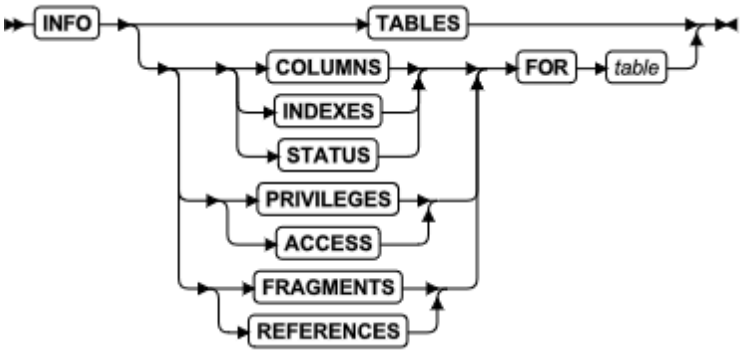
如果您省略 GRANT FRAGMENT 的 AS ***grantor*** 子句，或如果您指定您自己的登录名作为 ***grantor***，则您可在以后使用 REVOKE FRAGMENT 语句来取消您授予指定的用户的权限。例如，如果您将对 **part3** 中的 **customer** 表的分片的 Delete 权限授予用户 **martha**，但指定用户 **jack** 作为该权限的授权者，则用户 **jack** 可从用户 **martha** 取消那一权限，但您不可从用户 **martha** 取消那一权限。

DBA，或该分片的所有者，可使用 REVOKE FRAGMENT 语句的 AS 子句取消对该分片的权限。

# 2.97 INFO 语句

使用 INFO 语句来罗列在当前数据库中所有用户定义的表的名称，或来显示关于特定表的信息。

**语法**



此语句是对 SQL 的 ANSI/ISO 标准的扩展。您仅可随同 DB-Access 使用此语句。

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *table* | 您对其寻找信息的表 | 必须存在 | 数据库对象名 |

**用法**

INFO TABLES 语句罗列在当前数据库中所有用户定义的表的名称。可紧跟在 INFO 关键字之后的其他关键字指示 DB-Access 来显示其名称紧跟在 FOR 关键字之后的那个 ***table*** 的各种属性。要从多于一个关键字选项显示信息，请发出多个 INFO 语句。

INFO 语句支持的关键字选项可显示下列信息：

- **TABLES 关键字**

  使用 TABLES（不随同 FOR 子句）来罗列当前数据库中每个表的标识符，不包括系统目录表。每一用户定义的表按下列格式之一罗列：

  - 如果您是 **cust_calls** 表的所有者，则它显示为 **cust_calls**。
  - 如果您不是 **cust_calls** 表的所有者，则该所有者的授权标识符在表名称之前，诸如 **'june'.cust_calls**。

- **COLUMNS 关键字**

  使用 COLUMNS 来显示在指定的表中的列的名称和数据类型，显示每一列是否允许为 NULL 值。

- **INDEXES 关键字**

  使用 INDEXES 来显示指定的表的每一索引的名称、所有者和类型，以及集群状态，并罗列建立了索引的列。

- **FRAGMENTS 关键字**

  使用 FRAGMENTS 来显示分片策略和存储分片表的分片的 dbspace 的名称。如果以基于表达式的分布方案对表分片，则 INFO 语句还显示这些表达式。

- **ACCESS 或 PRIVILEGES 关键字**

  使用 ACCESS 或 PRIVILEGES 来显示指定的表的用户、角色和 PUBLIC 当前持有的自主访问权限。（在此上下文中，这两个关键字是同义词。）

- **REFERENCES 关键字**

  使用 REFERENCES 来显示在指定的表的列上可定义引用约束的用户的 References 访问权限。对于数据库级权限，请使用 SELECT 语句来查询 **sysusers** 系统目录表。

- **STATUS 关键字**

使用 STATUS 来显示关于指定的表的所有者、行长度、行和列的数目、创建日期和审计跟踪状态的信息。

使用 SQL 的 INFO 语句的另一种方法是使用 DB-Access 的 **SQL** 菜单或 **Table** 菜单的 **Info** 命令来显示相同的和附加的信息。

示例

使用下列示例来罗列数据库中的用户表：

```
INFO TABLES;
```

要显示关于特定的表的信息，请使用语法：

INFO ***info_keyword*** FOR ***table***

此处，*table* 为表名称，且 *info_keyword* 为 INFO 语句的七个关键字选项之一，除了 TABLES 之外。 例如，要显示表 customer 的列的名称，请使用此语句：

```
INFO COLUMNS FOR customer;
```

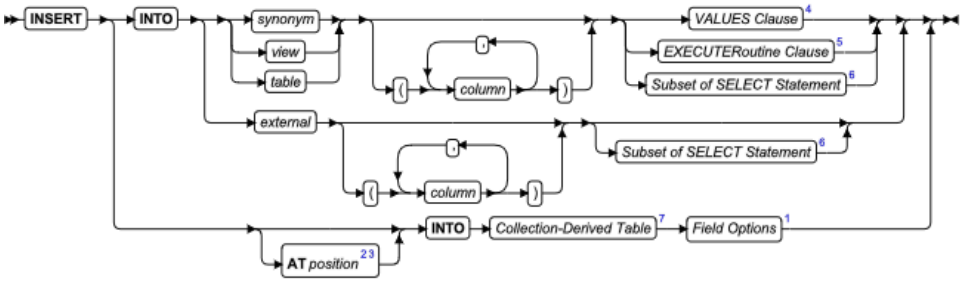
此示例产生下列输出：

```
Column name          Type            Nulls
customer_num         serial          no
fname                char(15)        yes
lname                char(15)        yes
company              char(20)        yes
address1             char(20)        yes
address2             char(20)        yes
city                 char(15)        yes
state                char(2)         yes
zipcode              char(5)         yes
phone                char(18)        yes
```

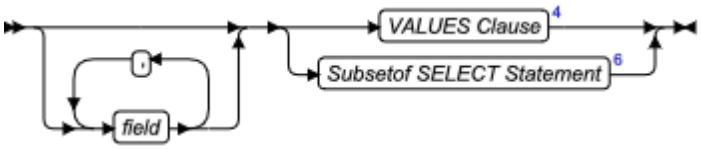
## 2.98 INSERT 语句

使用 INSERT 语句来向表或视图内插入一个或多个新行，或向 SQL 或 GBase 8s ESQL/C 集合变量内插入一个或多个元素。

**语法**




域选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要接收新值的列 | 请参阅 指定列。 | 标识符 |
| *external* | 要向其内插入数据的外部表 | 必须存在 | 数据库对象名 |
| *field* | 命名的或未命名的 ROW 数据类型的域 | 必须已在数据库中定义 | 字段定义 |
| *position* | 要将 LIST 数据类型的元素插入的位置 | 精确整数或 INT 或 SMALLINT 类型 SPL 变量。 | 精确数值 |
| *synonym, table, view* | 要将数据插入其中的表、视图或同义词 | 它指向的同义词或视图及该表必须存在 | 数据库对象名 |

### 用法

要将数据插入到表内，您必须或拥有该表或有对该表的 Insert 权限（请参阅 GRANT 语句）。要将数据插入视图内，您必须有所需要的 Insert 权限，且该视图必须满足在 通过视图插入行 中说明的要求。

如果该表或视图有数据完整性约束，则被插入的行必须满足该约束条件。如果不满足，则数据库服务器返回错误。如果将检查模式设置为 IMMEDIATE，则在每一 INSERT 语句的末尾检查所有指定的约束。如果将检查模式设置为 DEFERRED，则不检查所有指定的约束，直到该事务提交为止。

## 指定列

如果您未显式地指定一个或多个列，则使用列顺序将数据插入到这些列内，该顺序是在创建表或最后改变表时建立的。列顺序罗列在 **syscolumns** 系统目录表中。

在 GBase 8s ESQL/C 中，您可使用带有 INSERT 语句的 DESCRIBE 语句来标识该列顺序以及表中列的数据类型。

在 INSERT INTO 子句中指定的列的数目必须等于，或隐式地或显式地在 VALUES 子句中或由 SELECT 语句指定的值的数目。如果您指定列表，则列按照您罗列的列的顺序接收数据。跟在 VALUES 关键字之后的第一个值插入到罗列的第一列内，第二个值插入到罗列的第二列内，以此类推。

如果您从列列表省略一列，且该列没有与之相关联的缺省值，则当执行 INSERT 语句时，数据库服务器在该列中放一个 NULL 值。

## 使用 AT 子句（ESQL/C、SPL）

使用 AT 子句来在集合变量中指定的位置插入 LIST 元素。在缺省情况下， GBase 8s 在 LIST 集合的末尾添加一新的元素。

如果您指定的位置大于列表中元素的数目，则数据库服务器将该元素添加到列表的末尾。您必须指定至少为 1 的位置值，因为列表中的第一个元素在位置 1。

下列 SPL 示例在列表中指定的位置插于值：

```
CREATE PROCEDURE test3()
    DEFINE a_list LIST(SMALLINT NOT NULL);
    SELECT list_col INTO a_list FROM table1 WHERE id = 201;
    INSERT AT 3 INTO TABLE(a_list) VALUES( 9 );
    UPDATE table1 VALUES list_col = a_list WHERE id = 201;
END PROCEDURE;
```

假设在此 INSERT 之前， **a_list** 包含了元素 {1,8,4,5,2}。在此 INSERT 之后，**a_list** 包含元素 {1,8,9,4,5,2}。新元素 9 插入在列表中的位置 3 。要获得更过关于将值插入到集合变量内的信息，请参阅 集合派生表。

## 通过视图插入行

您可通过 *single-table* 视图插入数据，如果您有对该视图的 Insert 权限。 要执行此操作，可仅从一表定义 SELECT 语句，且它不包含下列任何组件：

- DISTINCT 关键字
- GROUP BY 子句
- 派生的值（也称之为虚拟列）
- 聚集值

如果未指定缺省值，则在视图中未定义的底层表中的列收到缺省值或 NULL 值。如果这些列之一没有缺省值，且不允许 NULL 值，则 INSERT 失败。

您可使用数据完整性约束来防止用户将值插入到不符合视图定义 SELECT 语句的底层表内。要获取进一步的信息，请参阅 WITH CHECK OPTION 关键字。

如果 INSTEAD OF 触发器在其 Action 子句中指定有效的 INSERT 操作，则您可通过 ***single-table*** 或 ***multiple-table*** 视图插入行。要获得关于如何创建通过视图插入的 INSTEAD OF 触发器的信息，请参阅 视图上的 INSTEAD OF 触发器。

如果几个用户正在将敏感信息输入到单个表内，则内建的 USER 函数可限制他们的视图每一用户仅插入指定的行。下列示例包含视图和实现此效果的 INSERT 语句：

```
CREATE VIEW salary_view AS
    SELECT lname, fname, current_salary FROM salary WHERE entered_by = USER;

INSERT INTO salary VALUES ('Smith', 'Pat', 75000, USER);
```

## 使用游标插入行

在 GBase 8s ESQL/C 中，如果您将游标与 INSERT 关联，则必须使用 OPEN、PUT 和 CLOSE 语句来执行 INSERT 操作，对于有事务但不符合 ANSI 的数据库，您必须在一个事务内发出这些语句。

如果您正在使用与 INSERT 语句关联的游标，则在您将行写入磁盘之前缓冲它们。在下列条件下，刷新插入缓冲区：

- 缓冲区已满。
- 执行 FLUSH 语句。
- CLOSE 语句关闭该游标。
- 在不符合 ANSI 的数据库中，OPEN 语句隐式地关闭游标，然后重新打开它。
- COMMIT WORK 语句终结该事务。

当刷新插入缓冲区时，在将这些行发送到数据库服务器之前，客户端处理器执行适当的数据转换。当数据库服务器收到该缓冲区时，它转换任何用户定义的数据类型，然后开始将这些行插入一次到数据库内。如果在数据库服务器将缓冲的行插入到数据库内期间遇到错误，则在最后一次成功地插入了的行之后的任何缓冲的行都被废弃。

## 将行插入到不带有事务的数据库内

如果您正在无事务日志记录地将行插入到数据库内，则一旦操作失败，您必须采取显式操作来恢复插入的行。例如，如果在插入一些行之后 INSERT 失败，则那些成功地插入了的行保留在表中。您不可自动地从失败的数据库插入恢复，因为不存在事务日志。

## 将行插入到带有事务的数据库内

如果您正在将行插入到数据库内，且正在使用显式的事务，在使用 ROLLBACK WORK 语句来撤销 INSERT。如果您在 INSERT 之前未执行 BEGIN WORK，且 INSERT 失败，则数据库服务器自动地回滚从 INSERT 开始以来产生的任何数据修改。如果您正在使用显式的事务，且 INSERT 失败，则数据库服务器自动地撤销 INSERT 的影响。

在符合 ANSI 的数据库中，事务是隐式的，且所有数据库更改都发生在一个事务之内。在此情况下，如果 INSERT 语句失败，则使用 ROLLBACK WORK 语句来撤销这些插入。

您以 RAW 日志记录类型创建的表无日志记录。这样，即使数据库使用日志记录，原始表也是不可恢复的。

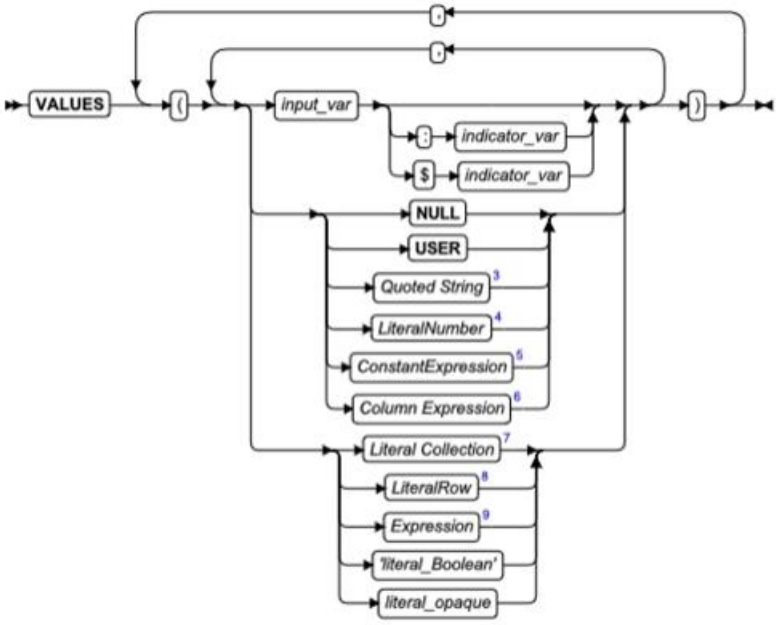
您以事务插入的那些行保持锁定，直到该事务结束为止。事务的结束，或是通过 COMMIT WORK 语句对数据库做出所有修改，或是通过 ROLLBACK WORK 语句对数据库不做任何修改。如果*单个* INSERT 语句影响许多行，则您可超过允许的并发锁的最大数目。要防止出现此情况，或者每个事务少插入些行，或者在您执行 INSERT 语句之前锁定页面（或整个表）。

## VALUES 子句

VALUES 子句可指定要插入到一列或多列内的值。当您使用 VALUES 子句时，您可一次仅插入一行或多行。

跟在 VALUES 关键字之后的每一值都指定给罗列在 INSERT INTO 子句中的对应列（或如果未指定列的列表，则以列的顺序罗列）。 如果您正在将引用的字符串插入到列内，则可无差错地插入的最大长度是 256 字节。

VALUES 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *indicator_var* | 如果 SQL 语句返回 NULL 给 *input_var*，则要显示的变量 | 请参阅 *GBase 8s ESQL/C 程序员手册*。 | 特定于语言 |
| *input_var* | 持有要插入的值的变量。此可为集合变量。 | 可包含任何 VALUES 子句的值选项 | 特定于语言 |
| *literal_opaque* | 不透明数据类型的文字表示 | 必须被不透明数据类型的 input 支持函数识别 | 请参阅不透明类型的文档。 |
| *literal_Boolean* | 作为单个字符的 BOOLEAN 值的文字表示 | 或是 't'（TRUE）或是 'f'（FALSE） | 引用字符串 |

在多行插入时，在 VALUES 子句中使用多个逗号分隔的列值列表，其中列表括在括号内，并用逗号分隔，列表数为插入数据行数。对于存在行触发器的表，每一行都会触发相关的触发器；同样如果目标表具有约束，那么每一行都会进行相应的约束检查，只要有一行不满足约束，所有的值都不能插入成功。

例如，在 user_info 表中新增两条数据：

INSERT INTO user_info (user_account,user_name,user_age,user_class) VALUES
('00001','张三','20','计算机系'),

('00002','李四','19','计算机系');

在 GBase 8s ESQL/C 中，如果您使用 ***input_var*** 变量来指定该值，则可向表内插入长于 256 字节的字符串。

要了解在 VALUES 子句中有效的关键字和精确值的类型，请参考 常量表达式。

## 考虑数据类型

INSERT 语句放入到列内的值无需与接收它的列具有相同的数据类型。然而，这两种数据类型必须兼容。如果数据库服务器有方法将一种数据类型强制转型为另一种，则两种数据类型是***兼容的***。***强制转型***是数据库服务器将一种数据类型转换为另一种的机制。

数据库服务器尽其所能地执行数据转换。如果数据不可转换，则 INSERT 操作失败。如果目标数据类型不可持有指定的值，则数据转换也失败。例如，您不可将整数 123456 插入到定义为 SMALLINT 数据类型的列内，因为此数据类型不能持有那么大的数。

对于数据库服务器提供的强制转型的总结，请参阅 *《GBase 8s SQL 指南：参考》*。要了解关于如何创建用户定义的强制转型的信息，请参阅本文档中的 CREATE CAST 语句以及 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

在使用非缺省语言环境的数据库中，如果 **GL_DATETIME** 环境变量有非缺省的设置，则在 INSERT 语句可正确地将本地化的 DATETIME 值插入到数据库表内，或视图内，或 EXTERNAL 表对象内之前，必须将 **USE_DTENV** 环境变量设置为 1。

## 将值插入到串行列之内

您可插入连续的数字、显式值或重置在 SERIAL、BIGSERIAL 或 SERIAL8 列中值的显式值：

- 要插入连续的串行值

  为 INSERT 语句中的串行列指定零（0）。在此情况下，数据库服务器指定下一最高值。

- 要插入显式值

  首先验证在表中没有重复的非零值，之后指定非零值。如果该串行列单独地索引，或有唯一约束，且表中已有重复的值，则导致错误。如果该值大于当前的最大值，您会在序列中创建间隔。

- 要创建序列中的间隔（即，重置串行值）

  在列中指定大于当前最大值的正值。

  另外，您可使用 ALTER TABLE 语句的 MODIFY 子句来重置串行列的下一值。

要了解更多信息，请参阅 更改下一个顺序值。

在串行列中 NULL 值无效。

在 GBase 8s 中，将序列值插入到作为表层级中一部分的表内，以您插入的值更新包含该串行计数器的层级中的所有表。您可将此值表示为零（0）作为下一最高值，或表示为特定的正整数。

## 将值插入到 Opaque 类型列内

GBase 8s 支持在 VALUES 子句中指定 opaque 数据类型的文字值作为引用的字符串的 INSERT 操作。您可使用此语法来将 opaque 的 UDT 插入到本地数据库的表的列内，或到本地实例的其他数据库中的表的列内。

当插入某些 opaque 数据类型时，需要特殊处理。例如，如果 opaque 数据类型包含占据空间的数据或多重表示的数据，则它可能提供如何存储该数据的选择：在内部结构中，或对于大对象，在智能大对象中。

这是通过调用名为 assign( ) 的用户定义的支持函数来完成的。当您在其行包含这些 opaque 类型之一的表上执行 INSERT 时，数据库服务器自动地为该类型调用 assign( ) 函数。assign( ) 函数可决定如何存储该数据。要了解更多关于 assign( ) 支持函数的信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南 。

## 将值插入到集合列内

您可使用 VALUES 子句来将值插入到集合列内。要获取更多信息，请参阅 集合构造函数。

例如，假设您定义 tab1 表如下：

```
CREATE TABLE tab1
    (
    int1 INTEGER,
    list1 LIST(ROW(a INTEGER, b CHAR(5)) NOT NULL),
    dec1 DECIMAL(5,2)
    );
```

下列 INSERT 语句将行插入到 tab1 内：

```
INSERT INTO tab1 VALUES
    (
    10,
    LIST{ROW(1,'abcde'),
        ROW(POW(3,3), '=27'),
        ROW(ROUND(ROOT(126)), '=11')},
    100
    );
```

在本示例中，集合列 **list1** 有三个元素。每一元素为带有一 INTEGER 字段和一 CHAR(5) 字段的未命名的行类型。第一个元素由两个精确值组成，一个整数（1）和一个引用的字符串（abcde）。第二个和第三个元素还使用引用的字符串来指出第二个字段，但以表达式指定第一个字段的值。

不管您使用什么方式来将值插入到集合列内，都不可将 NULL 元素插入到该列内。因而您使用的表达式不可等于 NULL。如果您尝试插入其中的集合包含 NULL 元素，则数据库服务器返回错误。

您还可使用集合变量来将一个或多个集合元素的值插入到集合列内。要获取更多信息，请参阅 集合派生表。

使用 VALUES 子句来将值插入到命名的或未命名的 ROW 类型列内，如下例所示：

```
CREATE ROW TYPE address_t
    (
    street CHAR(20),
    city CHAR(15),
    state CHAR(2),
    zipcode CHAR(9)
    );
CREATE TABLE employee
    (
    name ROW ( fname CHAR(20), lname CHAR(20)),
    address address_t
    );
```

下一示例在 name 和 address 列中插入精确值：

```
INSERT INTO employee VALUES
    (
        ROW('John', 'Williams'),
        ROW('103 Baker St', 'Tracy','CA', 94060)::address_t
    );
```

INSERT 使用 ROW 构造函数来生成 **name** 列（未命名的 ROW 数据类型）和 **address** 列（命名的 ROW 数据类型）的值。当您为命名的 ROW 数据类型指定值时，必须使用 CAST AS 关键字或双冒号（**::** ）运算符，以 ROW 数据类型的名称来将该值强制转型为命名的 ROW 数据类型。

要了解 ROW 构造函数的语法，请参阅“表达式”部分中的 构造函数表达式。要了解关于命名的 ROW 和未命名的 ROW 数据类型的精确值的信息，请参阅 Literal Row。

当您在 VALUES 子句中使用 ROW 变量时，该 ROW 变量必须包含每一字段值的值。要获取更多信息，请参阅 插入到行变量（ESQL/C、SPL）内。

您可使用 GBase 8s ESQL/C 主变量来以两种方式插入 *nonliteral* 值：

- 将整个 ROW 类型插入到列内。使用 VALUES 子句中的 **row** 变量来一次为 ROW 列中的所有字段插入值。
- ROW 类型的单个字段。要在 ROW 类型列中插入非精确值，请将这些元素插入到 **row** 变量内，然后在 UPDATE 语句的 SET 子句中指定该 **collection** 变量。

## 将值插入到 ROW 类型列内

使用 VALUES 子句来将值插入到命名的或未命名的 ROW 类型列内，如下例所示：

```
CREATE ROW TYPE address_t
    (
    street CHAR(20),
    city CHAR(15),
    state CHAR(2),
    zipcode CHAR(9)
    );
CREATE TABLE employee
    (
    name ROW ( fname CHAR(20), lname CHAR(20)),
    address address_t
    );
```

下一示例在 name 和 address 列中插入精确值：

```
INSERT INTO employee VALUES
    (
        ROW('John', 'Williams'),
        ROW('103 Baker St', 'Tracy','CA', 94060)::address_t
    );
```

INSERT 使用 ROW 构造函数来生成 **name** 列（未命名的 ROW 数据类型）和 **address** 列（命名的 ROW 数据类型）的值。当您为命名的 ROW 数据类型指定值时，必须使用 CAST AS 关键字或双冒号（**::** ）运算符，以 ROW 数据类型的名称来将该值强制转型为命名的 ROW 数据类型。

要了解 ROW 构造函数的语法，请参阅“表达式”部分中的 构造函数表达式。要了解关于命名的 ROW 和未命名的 ROW 数据类型的精确值的信息，请参阅 Literal Row。

当您在 VALUES 子句中使用 ROW 变量时，该 ROW 变量必须包含每一字段值的值。要获取更多信息，请参阅 插入到行变量（ESQL/C、SPL）内。

您可使用 GBase 8s ESQL/C 主变量来以两种方式插入 *nonliteral* 值：

- 将整个 ROW 类型插入到列内。使用 VALUES 子句中的 **row** 变量来一次为 ROW 列中的所有字段插入值。
- ROW 类型的单个字段。要在 ROW 类型列中插入非精确值，请将这些元素插入到 **row** 变量内，然后在 UPDATE 语句的 SET 子句中指定该 **collection** 变量。

## 分布式 INSERT 操作中的数据类型

访问另一 GBase 8s 实例的 INSERT 语句（或任何其他 SQL 数据操纵语言语句） 都仅可引用下列数据类型：

- 透明的内建的数据类型
- BOOLEAN
- LVARCHAR
- 透明的内建的数据类型的 DISTINCT
- BOOLEAN 的 DISTINCT
- LVARCHAR 的 DISTINCT

- 出现在此列表内的任何 DISTINCT 数据类型的 DISTINCT。

跨服务器的分布式 INSERT 操作可支持这些 DISTINCT 类型，仅当 DISTINCT 类型被显式地强制转型为内建的类型时，而在每一参与的数据库中，以完全相同的方式定义所有 DISTINCT 类型、其数据类型层级及其强制转型。

跨服务器 DML 操作不可引用复杂、大对象的列或表达式，也不可引用用户定义的数据类型(UDT)，不可引用不支持的 DISTINCT 或内建的 opaque 类型。要获得关于在跨服务器 DML 操作中 GBase 8s 支持的数据类型的信息，请参阅 跨服务器事务中的数据类型。

然而，访问本地 GBase 8s 实例的其他数据库的分布式操作可访问上列的跨服务器数据类型，以及下列数据类型：

- 大部分***内建的 opaque 数据类型***，如 跨数据库事务中的数据类型 中所列
- 在上一行中引用的内建的类型的 DISTINCT
- 罗列在上面两行中的任何数据类型的 DISTINCT
- 可被显式地强制转型为内建的数据类型的 opaque 用户定义的数据类型（UDT）。

跨数据库 INSERT 操作可支持这些 DISTINCT 和 opaque UDT，仅当所有的 opaque UDT 和 DISTINCT 类型都被显式地强制转型为内建的类型， 且在每一参与的数据库中以完全相同的方式定义所有 opaque UDT、DISTINCT 类型、数据类型层级和强制转型。

分布式 INSERT 事务不可访问另一 GBase 8s 实例的数据库，除非两个数据库在其 DBSERVERNAME 或 DBSERVERALIASES 配置参数以及在 **sqlhosts** 文件或 SQLHOSTS 注册子键中都定义 TCP/IP 或 IPCSTR 连接。支持相同类型连接（或 TCP/IP 或 IPCSTR）的参与服务器都需要适用于在 GBase 8s 实例之间的任何通信，即使都驻留在同一计算机上。

## 在 VALUES 子句中使用表达式

随同 GBase 8s，您可将除了列表达式之外的任何类型表达式插入到列中。例如，您可插入返回当前日期、日期和时间、当前用户的登录名或当前数据库驻留的数据库服务器名称的内建的函数。

TODAY 关键字返回系统日期。CURRENT 或 SYSDATE 关键字返回系统日期和时间。USER 或 CURRENT_USER 关键字返回包含当前用户的登录账户名称的字符串。SITENAME 或 DBSERVERNAME 关键字返回当前数据库驻留的数据库服务器名称。下列示例使用内建的函数来插入数据：

```
INSERT INTO cust_calls (customer_num, call_dtime, user_id,
                        call_code, call_descr)
    VALUES (212, CURRENT, USER, 'L', '2 days');
```

要获取更多信息，请参阅 表达式。

## 插入 NULL 值

当您执行 INSERT 语句时，数据库服务器将 NULL 值插入到您未提供值的任何列内，以及没有缺省值的和未显式地罗列的所有列内。您还可在 VALUES 子句中指定 NULL 关键字来指示应指定 NULL 值的列。

下列示例将值插入到 orders 表的三个列内：

```
INSERT INTO orders (orders_num, order_date, customer_num) VALUES (0, NULL, 123);
```

在此示例中，在 **order_date** 列中显式地输入 NULL 值，且***未***显式地罗列在 INSERT INTO 子句内的 **orders** 表的所有其他列都以 NULL 值填充。

## 将值插入到受保护的表内

在使用基于标签的访问控制（LBAC）的数据库中，INSERT 语句的 INTO 子句可引用被安全策略保护的表，如果该用户持有保护该表的标签的安全策略的足够的凭证，且持有对该表的 Insert 权限的话。

然而，未持有安全标签的用户不可将数据插入到有 LBAC 行保护的表内，即使该用户已经被授予了从该安全策略的规则所需要的豁免，除非在 INSERT 语句的 VALUES 子句中指定受保护表的行标签。数据操作语言语句可提供受保护的表的行标签，通过调用任何三个内建的函数，其第一个参数指定安全策略的名字，且其附加的参数为下列之一：

- 安全标签的名称
- 表中 IDSSECURITYLABEL 列的名称。
- 该标签中安全策略组件的名称及其元素的值

例如，下列 INSERT 语句调用内建的 SECLABEL_BY_NAME 函数，以便将新行插入到名为 tab002 的表内，该表受到 MegaCorp 安全策略的名为 Decca 的行标签的保护：

```
INSERT INTO tab002
    VALUES (SECLABEL_BY_NAME('Megacorp', 'Decca'), 45, 'A.C.Debussy');
```

此 INSERT 操作能否成功取决于该用户的安全凭证是否充分，相对于 Decca 标签的组件值，来启用对 tab002 表的写访问。

要了解关于通过调用 **SECLABEL_BY_NAME** 或类似的内建的函数访问受保护的表的 INSERT 语句的附加的示例，请参阅 安全标签支持函数。要了解关于 LBAC 安全策略、安全标签、读和写的访问规则以及从那些规则的豁免的通用信息，请参阅您的 GBase 8s 安全指南。

## 截断的 CHAR 值

在不符合 ANSI 的数据库中，如果您为 CHAR(n) 列或变量赋值，且那个值的长度超过 n 个字符，则数据库服务器截断最后的那些字符，不发出错误。例如，假设您定义此表：

```
CREATE TABLE tab1 (col_one CHAR(2);
```

数据库服务器在下列 INSERT 语句中截断数据值，分别为 ″jo″ 和 ″sa″，但不返回错误：

```
INSERT INTO tab1 VALUES ("john");
INSERT INTO tab1 VALUES ("sally");
```

因此，在不符合 ANSI 的数据库中，当插入的或更新的值超过声明的长度 ***n*** 时，不强制要求 CHAR(***n***) 列或变量的语义完整性。（但是，在符合 ANSI 的数据库中，在发生字符数据截断时，数据库服务器发出错误 -1279。）

# SELECT 语句的子集

如 INSERT 语句 语法图中指出的那样，在 INSERT 语句内的查询中，并非 SELECT 语句的所有子句和选项都可以使用。 在 INSERT 语句中不支持下列 SELECT 子句和选项：

- FIRST 和 LIMIT
- INTO TEMP、INTO RAW 和 INTO STANDARD 结果表选项
- UNION、UNION ALL、INTERSECT、MINUS 和 EXCEPT 设置运算符。

在符合 ANSI 的数据库中，如果此语句有一不返回行的 WHERE 子句，则 **sqlca** 返回 SQLNOTFOUND (100)。

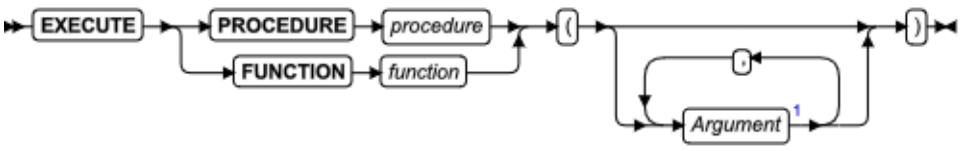
如果多重语句准备的对象的一部分的 INSERT 语句未插入行，则 **sqlca** 为符合 ANSI 的数据库和不符合 ANSI 的数据库都返回 SQLNOTFOUND (100)。在不符合 ANSI 的数据库中，如果没有行满足 WHERE 子句，则 **sqlca** 返回零（0）。

在 GBase 8s 中，如果您正在将值插入表层级中的一个超级表内，则子查询可引用一子表。如果您正在将值插于表层级中的子表内，则如果子查询仅引用超级表，则子查询可引用该超级表。即，子查询必须使用 SELECT…FROM ONLY (***supertable***) 语法。

## Execute Routine 子句

您可指定 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句来插入用户定义的函数返回的值。

Execute Routine 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function*, *procedure* | 要插入数据的用户定义的函数或过程 | 必须存在 | 数据库对象名 |

当您使用用户定义的函数来插入列值时，该函数的返回值必须与该罗列的列有一一对应。即，该函数返回的每一值必须是列列表中以对应的 ***column*** 期望的数据类型。

为了向后兼容起见， GBase 8s 可使用 EXECUTE PROCEDURE 关键字来执行以 CREATE PROCEDURE 创建的 SPL 函数。

如果被调用的 SPL routine 扫描或更新 INSERT 语句的目标表，则数据库返回错误。即，该 SPL 例程不可向您正在向其中插入的行的表选择数据。

如果被调用的 SPL 例程包含特定的 SQL 语句，则数据库服务器返回错误。要获取关于在数据操作语句内调用的 SPL 例程中不可使用的 SQL 语句的信息，请参阅 在数据操纵语句中 SPL 例程的限制。

## 由 SPL、C 和 Java 函数返回的值的数量

一 SPL 函数可返回一个或多个值。请确保返回值的数量与表中列的数量或 INSERT 语句的列列表中列的数量相匹配。这些列必须具有与 SPL 函数返回的值相兼容的数据类型。

用 C 或 Java™ 语言编写的一外部函数可仅返回一个值。请确保您在 INSERT 语句的列列表中仅指定一列。这些列必须有与该外部函数返回的值相兼容的数据类型。该外部函数可为迭代符函数。

下列示例显示如何将数据插入名为 result_tmp 的临时表，以便输出到一返回多行的用户定义的函数（f_one）的结果文件：

```
CREATE TEMP TABLE result_tmp( ... );
INSERT INTO result_tmp EXECUTE FUNCTION f_one();
UNLOAD TO 'file' SELECT * FROM foo_tmp;
```

## 插入到行变量（ESQL/C、SPL）内

INSERT 语句不支持“集合派生的表”段中的行变量。然而，您可使用 UPDATE 语句量将新的字段值插入到行变量内。例如，下列 GBase 8s ESQL/C 代码片断将新行插入到 rectangles 表（由 将值插入到 ROW 类型列内 定义）：

```
EXEC SQL BEGIN DECLARE SECTION;
    row (x int, y int, length float, width float) myrect;
EXEC SQL END DECLARE SECTION;


...
EXEC SQL update table(:myrect)
    set x=7, y=3, length=6, width=2;
EXEC SQL insert into rectangles values (12, :myrect);
```

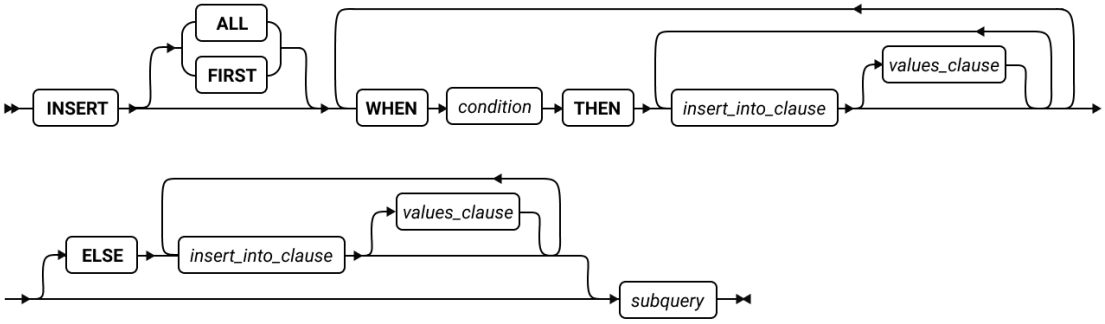
要获取更多信息，请参阅 更新 Row 变量（ESQL/C）。

## 使用 INSERT 作为动态管理语句

在 GBase 8s ESQL/C 中，您可使用 INSERT 语句来处理以下情况：您需要编写可插入在您编译时还不知道其结构的数据的代码。要获取更多信息，请参考 *GBase 8s ESQL/C 程序员手册* 的动态管理章节。

# 2.99 INSERT ALL/FIRST 语句

INSERT ALL 是指将同一批数据插入到符合条件的若干张表中，INSERT FIRST 是指将同一批数据插入到第一个符合条件的表中。也就是说，当表达式第一个条件为 TRUE 之后，INSERT FIRST 会跳过后面的条件并结束插入，而 INSERT ALL 会继续执行接下来若干条件为 TRUE 的操作。




| 子句 | 限制 |
| --- | --- |
| ALL/FIRST | 省略时为 ALL，ALL 和 FIRST 不能同时出现。 |
| WHEN condition | condition 是返回结果为布尔类型的算数表达式，不支持省略，支持一个或多个条件表达式，多个表达式可用 AND、OR 等逻辑运算符和谓词连接。表达式中出现的列名需要是 subquery 子句返回的列，运算符两边都支持列名。 |
| THEN into | 表示条件满足时，将数据插入到 into 子句的表中，into 子句不支持省略。into 子句支持在表后括号中罗列本表列名，列名省略时表示需要对表中所有列依次顺序插入数据。 |
| values | 用于指定需要插入的值，支持列名、函数、运算符、常量值等形式，此处的列名为 subquery 子句返回的列，如果 subquery 子句对投影列使用了别名，则 values 子句中的列名指的是 subquery 子句中投影列的别名。<br>如果 into 子句的表中本次插入值的列的数量少于 subquery 子句返回列的数量，则需要在 into 子句表后括号中罗列列名并在 values 子句中指定需要插入的值；如果 into 子句的表中本次插入值的列的数量少于 subquery 子句返回列的数量，则需要在 into 子句表后括号中罗列列名并在 values 子句中指定需要插入的值；values 子句支持省略，省略时，into 子句中表列的数量需要和 subquery 子句返回的列数量一致。 |
| WHEN…THEN… | 此子句支持多次出现：WHEN…THEN…组合使用时，当表达式第一个条件为 TRUE 之后，INSERT FIRST 会跳过后面的条件并结束插入，而 INSERT ALL 会继续执行接下来若干条件为 TRUE 的操作。 |
| ELSE | 当 WHEN 子句根据条件表达式判断为假而都没有执行后，执行 ELSE 子句中的 into 子句部分，其用法与 WHEN 子句中的相同。ELSE 子句可以省略。 |
| subquery | subquery 子句返回的是一个数据集，支持 SELECT…FROM…、WHERE、UNION 等。 |

例如，可以通过以下代码，根据数据某列的值进行条件判断后插入数据：

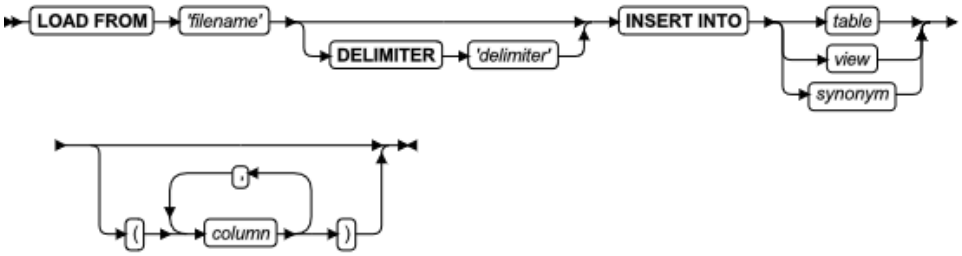
```
INSERT ALL
WHEN FMonth=1 THEN
INTO t1
WHEN FMonth=2 THEN
INTO t2
ELSE
INTO t3
SELECT FProductID, FProductName, FMonth FROM TPruduct;
```

## 2.100 LOAD 语句

使用 LOAD 语句将数据从操作系统文件插入到现有的表或视图内。

**语法**

仅 DB-Access 支持 LOAD 语句。




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要从 *filename* 接收数据值的列 | 请参阅 INSERT INTO 子句。 | 标识符 |
| *delimiter* | 在装入文件的每一行内要分隔数据值的字符。缺省的*定界符*为管道（\|）符号。 | 请参阅 DELIMITER 子句。 | 引用字符串 |
| *filename* | 要读取的文件的路径和文件名。缺省路径为当前目录 | 请参阅 LOAD FROM 文件。 | 特定于操作系统规则 |
| *synonym*, *table*, *view* | 要从 *filename* 向其中插入数据的表的同义词 | 所指向的 *Synonym* 和 *table* 或 *view* 必须存在 | 数据库对象名 |

**用法**

此语句是对 SQL 的 ANSI/ISO 标准的扩展。您仅可随同 DB-Access 使用此语句。

LOAD 语句向表追加新行。它不覆盖现有的数据。您添加的行不可与现有的行具有相同的键。

要使用 LOAD 语句，您必须有对您想要插入数据的表的 Insert 权限。要了解关于数据库级和表级权限的信息，请参阅 GRANT 语句。

您使用非缺省的语言环境的数据库中，如果 **GL_DATETIME** 环境变量有一非缺省的设置，则在 LOAD 语句可将本地化的 DATETIME 值正确地插入到数据库表，或插入到视图，或插入到 CREATE EXTERNAL TABLE 语句定义的对象内之前，**USE_DTENV** 环境变量必须设置为 1。要了解更多关于 **GL_DATETIME**、**GL_DATE**、**DBTIME** 和 **USE_DTENV** 环境变量的信息，请参考 *GBase 8s GLS 用户指南*。

## LOAD FROM 文件

LOAD FROM 包含要被装入到特定的表或视图内的数据。该装入文件的缺省路径名为当前的目录。

您可使用 UNLOAD 语句创建的文件作为 LOAD FROM 文件。（要了解各种数据类型在 UNLOAD TO 文件中如何展示的描述，请参阅 UNLOAD TO 文件。）

如果您在 INSERT INTO 子句中未包括列的列表，则该文件中的字段必须与为表指定的列在数量、顺序和数据类型方面相匹配。

该文件的每一行必须有相同的字段数量。 您定义的字段长度必须小于或等于为相应的列指定的长度。仅指定那些可转换为对应列的数据类型的值。下表指出数据库服务器期望您如何在 LOAD FROM 文件中表示数据类型（当您使用缺省的语言环境 U.S. English 时）。

| 数据的类型 | 输入格式 |
|---|---|
| 空 | 在定界符之间的一个或多个空字符。您可在不对应于字符列的字段中包括前导空白。 |
| BOOLEAN | t 或 T 表示 TRUE 值，f 或 F 表示 FALSE 值。 |
| COLLECTIONS | 集合必须有大括号括起其值，且用字段定界符分开每一元素。要获取更多信息，请参阅 装入复杂数据类型。 |
| DATE | 下列格式的字符串： *mm/dd/year*。您必须声明月份为两位数字。如果年份在 20 世纪中，则可使用两位数字表示年份。（您可以 **DBCENTURY** 环境变量指定另一世纪算法。）该值必须为一实际日期；例如，2 月 30 日是非法的。如果您以 **GL_DATE** 或 **DBDATE** 环境变量指定一种不同的日期格式，则可使用此格式。要获取更多关于环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》* 和 *GBase 8s GLS 用户指南*。 |
| DECIMAL、MONEY、FLOAT | 可包括开始和/或结尾的货币符号以及千分位和十进制分隔符的值。您的语言环境或 **DBMONEY** 环境变量可指定货币格式。 |
| NULL | 在定界符之间没有任何符号 |

| 数据的类型 | 输入格式 |
| --- | --- |
| ROW 类型（命名的或未命名的） | ROW 类型必须以括号和分隔每一元素的字段定界符括起其值。要获取更多信息，请参阅 装入复杂数据类型。 |
| 简单大对象（TEXT、BYTE） | 直接地从 LOAD TO 文件装入的 TEXT 和 BYTE 列。要获取更多信息，请参阅 装入简单大对象。 |
| 智能大对象（CLOB、BLOB） | 从独立的操作系统文件装入的 CLOB 和 BLOB 列。在 LOAD FROM 文件中的 CLOB 或 BLOB 列的字段包含此独立文件的名称。要获取更多信息，请参阅 装入智能大对象。 |
| 时间 | 以 *year-month-day hour:minute:second.fraction* 格式的字符串。您不可使用 DATETIME 或 INTERVAL 值的数据类型关键字或限定符。年份必须是 4 位数字，其月份必须是 2 位数字。**DBTIME** 或 **GL_DATETIME** 环境变量可指定其他终端用户格式。 |
| 用户定义的数据格式（opaque 类型） | 如果需要特殊的处理来将 LOAD FROM 文件中的数据复制到该 opaque 类型的内部格式，则关联的 opaque 类型必须定义了导入支持函数。导入二进制支持函数还可能需要二进制格式的数据。LOAD FROM 文件数据的格式必须是导入或导入二进制支持函数期望的格式。如果在数据库中写入数据之前需要特殊处理，则关联的 opaque 类型必须有赋值支持函数。请参阅 装入 opaque 类型列。 |

要获取更多关于 **DB*** 环境变量的信息，请参考 *《GBase 8s SQL 指南：参考》*。要获取更多关于 **GL*** 环境变量的信息，请参考 *GBase 8s GLS 用户指南*。

如果您正在使用非缺省的语言环境，则 DATE、DATETIME、MONEY 的格式，以及 LOAD FROM 文件中的数字列值必须与该语言环境支持的这些数据类型的格式相兼容。要获取更多信息，请参阅 *GBase 8s GLS 用户指南*。

下列示例展示名为 **new_custs** 的输入文件的内容：

```
0|Jeffery|Padgett|Wheel Thrills|3450 El Camino|Suite 10|Palo Alto|CA|94306||
0|Linda|Lane|Palo Alto Bicycles|2344 University||Palo Alto|CA|94301|
    (415)323-6440
```

此数据文件传递下列信息：

- 通过指定零（0）来表示串行字段
- 使用管道（|），缺省的定界符
- 将 NULL 值赋值给第一行的 **phone** 字段和第二行的 **address2** 字段

  用两个之间不含任何符号的定界符表示 NULL 值。

下列语句将值从 new_custs 文件装入到 jason 拥有的 customer 表内：

```
LOAD FROM 'new_custs' INSERT INTO jason.customer;
```

如果您包括任何下列特殊字符作为字段的值的一部分，则必须在该字符之前使用一反斜杠（ \ ）转义字符：

- 反斜杠
- 定界符
- 在 VARCHAR 或 NVARCHAR 列的值中任何位置的换行符
- 在 TEXT 值的值的末尾的换行符

请不要使用反斜杠字符（ \ ）作为列分隔符。它作为转义字符来通知 LOAD 语句，下一个字符要解释为数据的一部分，而不是有特别的意义。

与字符列相对应的域可包含比允许该字段定义的字符数更多的字符。忽略多余的字符。

如果您正在装入包含 VARCHAR 数据类型的文件，请注意下列信息：

- 如果您 LOAD 语句数据中，字符字段（包括 VARCHAR）不长于列大小，则忽略多余的字符。
- 使用反斜杠（ \ ）来转义包括 VARCHAR 的所有字符字段中的嵌入的定界符和反斜杠字符。
- 请不要在 LOAD FROM 文件中使用下列字符作为定界字符：数字（0 至 9）、字母 a 至 f 以及 A 至 F、反斜杠（ \ ）字符或换行 NEWLINE（CTRL-J）符。

## 装入简单大对象

数据库服务器直接地从 LOAD FROM 文件装入简单大对象(BYTE 和 TEXT 列)。在您装入 BYTE 和 TEXT 数据时，请记住下列限制：

- 您不可在 BYTE 字段中有开头和结尾的空白
- 使用反斜杠（\）转义 TEXT 字段中的精确定界符和反斜杠字符的特殊意义。
- 正在装入到 BYTE 列内的数据必须为 ASCII 十六进制格式。BYTE 列不可包含前导空白。
- 请不要在 LOAD FROM 文件中使用下列字符作为定界字符：数字（0 至 9）、字母 a 至 f 和 A 至 F、反斜杠（\）字符或 NEWLINE（CTRL-J）字符。

为了在非缺省的语言环境中装入 TEXT 列，数据库服务器为该数据处理任何需要的代码集转换。另请参阅 *GBase 8s GLS 用户指南*。

如果您正在装入包含 BYTE 或 TEXT 数据类型的文件，小于 10 KB 的对象临时地存储在内存中。您可以 **DBBLOBBUF** 环境变量将该 10 KB 的设置调整为更大的设置。大于缺省值或 **DBBLOBBUF** 设置的简单大对象存储在临时文件中。要了解更多关于 **DBBLOBBUF** 环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 装入智能大对象

数据库服务器从客户端计算机上独立的操作系统文件装入智能大对象（BLOB 和 CLOB 列）。要了解此文件的结构，请参阅 卸载智能大对象。

在 LOAD FROM 文件中，CLOB 或 BLOB 列值如下所示：

***start_off*, *length*, *client_path***

在此格式中，***start_off*** 为该客户端文件内的智能大对象值的起始偏移量（以十六进制计），***length*** 为 BLOB 或 CLOB 值的长度（以十六进制计），***client_path*** 为客户端文件的路径名。在这些值之间不可出现空格。

例如，要装入 /usr/apps/clob9ce7.318 文件中的 512 字节长且偏移量 256 的 CLOB 值，数据库服务器期望在 LOAD FROM 文件中出现如下的 CLOB 值：

|100,200,/usr/apps/clob9ce7.318|

如果要装入整个客户端文件，则在 LOAD FROM 文件内出现如下的 CLOB 或 BLOB 列值：

***client_path***

例如，要装入占据整个文件 /usr/apps/clob9ce7.318 的 CLOB 值，数据库服务器期望在 LOAD FROM 文件中出现下列 CLOB 值：

|/usr/apps/clob9ce7.318|

在 DB-Access 中，在从 DB-Access 中执行的文件内 USING 子句有效。在交互模式下，DB-Access 提示您输入密码，因此不使用 USING 关键字和 ***validation_var***。

对于 CLOB 列，数据库服务器为该数据处理任何需要的代码集转换。另请参阅 *GBase 8s GLS 用户指南*。

## 装入复杂数据类型

在 LOAD FROM 文件中，复杂数据类型如下所示：

- 以适当的构造函数（SET、MULTISET 或 LIST）引入集合，且其元素括在大括号（{ }）中，并以逗号分隔，如下：

  ***constructor*{*val1* , *val2* , . . . }**

  例如，要将 SET 值 {1, 3, 4} 装入到其数据类型为 SET(INTEGER NOT NULL) 的列内，则 LOAD FROM 文件的相应字段显示为：

  |SET{1,3,4}|

- 以 ROW 构造函数引入行类型（命名的和未命名的），以圆括号括起其字段，并以逗号分隔，如下：

  ROW(***val1* , *val2* , ...** )

例如，要装入 ROW 值 (1, 'abc')，LOAD FROM 文件的相应的字段显示为：

```
|ROW(1,abc)|
```

## 装入 opaque 类型列

当插入某些 opaque 数据类型时，它们需要特殊的处理。例如，如果 opaque 数据类型包括占据空间的或多重表示的数据，则它可能提供如何存储该数据的选项：是在内部的结构之中，对于大对象，或在智能大对象中。

通过调用名为 **assign( )** 的用户定义的支持函数来完成此处理。当您在其行包含这些 opaque 类型之一的表上执行 LOAD 语句时，数据库服务器自动地为该类型调用 **assign( )** 函数。**assign( )** 函数可确定如何存储数据。要了解更多关于 **assign( )** 支持函数的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## DELIMITER 子句

使用 DELIMITER 子句来指定分隔在输入文件中的行中的每一列中包含的数据的定界符。 您可指定 (CTRL-I) 或空格（= ASCII 32）作为定界符号。您不可使用下列项作为定界符号：

- 反斜杠（\）
- NEWLINE 字符 (CTRL-J)
- 十六进制数（0 至 9、a 至 f、A 至 F）

如果您省略此子句，则数据库服务器检查 **DBDELIMITER** 环境变量。要获得关于如何设置 **DBDELIMITER** 环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

如果尚未设置 **DBDELIMITER** 环境变量，则缺省的定界符为管道（|）。

下列示例指定分号（ ;）作为定界符号。该示例使用 Windows™ 文件命名约定。

```
LOAD FROM 'C:\data\loadfile' DELIMITER ';'
   INSERT INTO orders;
```

## INSERT INTO 子句

使用 INSERT INTO 子句来指定要将新数据装入其中的表、同义词或视图。

您必须指定列名称仅当下列条件之一为真：

- 您不是将数据装入所有列内。
- 输入文件与这些列的缺省顺序不匹配（当创建表时指定的顺序）。

INTO 子句不可指定 CREATE EXTERNAL TABLE 语句定义的表对象。

下列示例标识 price 和 discount 列作为向其中添加数据的仅有的列。该示例使用 Windows™ 文件命名约定。

```
LOAD FROM 'C:\tmp\prices' DELIMITER ','
   INSERT INTO norman.worktab(price,discount)
```

# 2.101 LOCK TABLE 语句

使用 LOCK TABLE 语句来控制通过其他进程访问表。xzzz

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | *synonym* 或 *table* 的所有者 | 必须为指定的对象的所有者 | 所有者名称 |
| *synonym* | 要被锁定的表的同义词 | 同义词以及指向它的表必须存在 | 标识符 |
| *table* | 要被锁定的表 | 请参阅 用法 的第一段。 | 标识符 |

**用法**

此语句是对 SQL 的 ANSI/ISO 标准的扩展。

如果下列任一为真，则您可使用 LOCK TABLE 来锁定表：

- 您是该表的所有者。
- 您有对该表或对该表中列的 Select 权限，或通过直接授权，或通过授权给 PUBLIC 或给您的当前角色。

如果该表已被另一进程在 EXCLUSIVE 模式下锁定，或如果您请求 EXCLUSIVE 锁而另一用户已经以 SHARE 模式锁定了同一张表，则 LOCK TABLE 语句失败。

SHARE 关键字以***共享模式***锁定表。共享模式允许其他进程对该表的***读***访问，但拒绝***写***访问。如果以共享模式锁定表，则其他进程不可更新或删除数据。

EXCLUSIVE 关键字以***排他模式***锁定表。此模式拒绝其他进程对表进行读访问，也拒绝写访问。 在下列语句期间，排他模式锁定自动地发生：

- ALTER FRAGMENT
- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE
- TRUNCATE

**在一些 DDL 操作中的 ONLINE 关键字**

在某些 ALTER FRAGMENT、DROP INDEX 和 CREATE INDEX 操作期间，包括 ONLINE 关键字在内，当并发的会话尝试访问同一表时，可减低运行时出错的风险。要获取更多关于支持 ONLINE 关键字选项的那些 DDL 语句的锁定行为的信息，请参阅这些主题：

- 在 ATTACH 操作中使用 ONLINE 关键字
- 在 DETACH 操作中使用 ONLINE 关键字
- 在 MODIFY 操作中使用 ONLINE 关键字
- CREATE INDEX 的 ONLINE 关键字
- DROP INDEX 的 ONLINE 关键字.

**在辅助服务器上的 LOCK TABLE 语句行为**

在高可用性集群中，您可从可更新的辅助服务器上设置表的排他锁。对于来自辅助服务器的排他模式锁请求，会话可读该表但不可更新它。此行为类似于辅助服务器上的共享访问模式；即，当一会话在给定的表上有排他锁时，其他会话不可获取那个表上的共享或排他锁。

在只读的辅助服务器上，发出 LOCK TABLE 语句的会话不锁定该表，且数据库服务器不向客户端返回错误。

集群中的共享模式锁的行为与单独服务器的相同。在成功地运行 LOCK TABLE 语句之后，用户可读该表但不可改变它，直到释放该锁为止。

## 对带有共享锁的表的并发访问

在成功地执行指定 IN SHARE MODE 关键字的 LOCK TABLE 语句之后，其他用户可读该表，但不可改变他的数据，直到释放该锁为止。在支持事务日志记录的数据库中，SELECT 语句可在罗列在 FROM 子句中的每一表上隐式地放置一共享锁，以便防止其他用户修改那些表，直到提交或回滚该查询为止。

## 对带有排他锁的表的并发访问

成功地执行带有 IN EXCLUSIVE MODE 选项的 LOCK TABLE 语句之后，其他用户不可获得对该指定表的锁。然而，当您尝试对那个表进行 DDL 操作时，如果一并发的会话（例如，通过打开游标）正在访问同一表，则您可能收到 RSAM error -106。此错误还可影响某些 DDL 语句自动地在这些表上放置的隐式的锁。

这可能是因为表锁不排除表访问。排他锁防止其他用户获得锁，但不防止那些等待释放该排他锁的写操作打开该表，或对该表的 Dirty Read 操作。您可设置 **IFX_DIRTY_WAIT** 环境变量来指定 DDL 等待指定的秒数，以便 Dirty Read 操作提交或回滚。

当表中的一行或多行被排他锁锁定时，对其他用户的影响部分地取决于他们的事务隔离级别。除了 Dirty Read 隔离级别之外的所有其他隔离级别中的其他用户可能遇到锁定错误，比如，由于在指定的时间限制内未释放锁，或由于发生死锁情况，导致事务失败。

在行级锁定影响一些行的表上，通过启用事务来在行级锁定的表中读取数据的最近提交的版本，可降低锁定冲突的风险，而不是等待提交或回滚在那行上持有该锁的事务。这可通过几种不同的方法实现，包括：

- 从单个事务发出此 SQL 语句

  SET ISOLATION TO COMMITTED READ LAST COMMITTED;
- 对于使用 Committed Read 或 Read Committed 隔离级别的所有会话，请将 USELASTCOMMITTED 配置参数设置为 'ALL' 或 'COMMITTED READ'，或者另发出带有 'ALL' 或 'COMMITTED READ' 作为会话环境选项的 SET ENVIRONMENT USELASTCOMMITTED 语句。
- 对于使用 Dirty Read 或 Read Uncommitted 隔离级别的所有会话，请将 USELASTCOMMITTED 配置参数设置为 'ALL' 或 'DIRTY READ'，或者另发出带有 'ALL' 或 'DIRTY READ' 作为会话环境选项的 SET ENVIRONMENT USELASTCOMMITTED 语句。
- 对于在数据库中为其定义 ***user*.sysdbopen( )** 过程的用户，DBA 可定义那个过程来包括 SET ENVIRONMENT USELASTCOMMITTED 语句，此语句带有 'ALL' 或 'COMMITTED READ' 作为会话环境选项，且还发出 SET ISOLATION 语句来设置 Committed Read 作为隔离级别。
- 对于在数据库中不存在为其定义 ***user*.sysdbopen( )** 过程的用户，DBA 可定义 PUBLIC.**sysdbopen** 过程，该过程指定相同的 SET ENVIRONMENT USELASTCOMMITTED 和 SET ISOLATION 语句。

仅当行级锁定有效时，而不是当另一会话持有对整个表的排他锁时，此 LAST COMMITTED 隔离特性才有用。当 LOCK TABLE 应用表级锁时，此特性对于指定的表不可用。要了解更多关于此 LAST COMMITTED 特性，为了并发访问有些行被排他锁锁定的表，且为了在可支持此特性的表的种类上的限制的信息，请参阅 Committed Read 的 LAST COMMITTED 选项。

## 带有事务日志记录的数据库

如果您以事务日志记录创建的数据库，则 LOCK TABLE 语句成功，仅当在事务之内执行它。在您可执行 LOCK TABLE 语句之前，必须发出 BEGIN WORK 语句。

在符合 ANSI 的数据库中，事务是隐式的。如果指定的表尚未被另一进程锁定，则 LOCK TABLE 语句成功。

下列准则适用于在事务内使用 LOCK TABLE 语句：

- 您不可锁定系统目录表。
- 您不可在事务内在共享的和排他的表锁之间切换。例如，一旦您在共享模式下锁定该表，不可将该锁定模式升级为排他。
- 如果在访问表中的一行之前，您发出 LOCK TABLE 语句，且 PDQ 为生效，则不为该表设置行锁。以此方式，您可覆盖行级锁定并避免超过在数据库服务器配置中定义的锁定的最大数目。（但如果 PDQ 生效，则您可能用尽锁，发生错误 -134，除非您的 ONCONFIG 文件的 LOCKS 参数指定足够大的锁定数。）
- 在完成事务之后，自动地释放所有行和表锁。 在使用事务日志记录的数据库中，UNLOCK TABLE 语句失败。

- 同一用户可显式地使用 LOCK TABLE 来并发地锁定最多 32 个表。（使用 SET ISOLATION 来指定适当的隔离级别，诸如 Repeatable Read，如果在单个事务期间您需要从多于 32 个表锁定行的话。）

下列示例展示如何在以事务日志记录创建了的数据库中更改表的锁定模式：

```
BEGIN WORK;
LOCK TABLE orders IN EXCLUSIVE MODE;
 ...
COMMIT WORK;
BEGIN WORK;
LOCK TABLE orders IN SHARE MODE;
 ...
COMMIT WORK;
```

**警告：** 建议您在事务中不要使用无日志记录的表。如果您需要在事务中使用无日志记录的表，或在排他模式下锁定该表，或设置隔离级别为 Repeatable Read，以防并发问题。

### 无事务日志记录的数据库

在无事务日志记录（通过省略 CREATE DATABASE 语句中的 WITH LOG 关键字）创建了的数据库中，在任一下列事件之后，释放通过 LOCK TABLE 语句设定了的表锁：

- 执行 UNLOCK TABLE 语句。
- 用户关闭该数据库。
- 用户从应用程序退出。

要更改对表的锁定模式，请以 UNLOCK TABLE 语句释放该锁，然后发出新的 LOCK TABLE 语句。

下列示例展示如何在无日志记录的表中更改锁定模式：

```
LOCK TABLE orders IN EXCLUSIVE MODE;
    . . .
UNLOCK TABLE orders;
    . . .
LOCK TABLE orders IN SHARE MODE;
```

### 锁定粒度

锁定表的缺省的粒度是在***页***级，或者您在 **IFX_TABLE_LOCKMODE** 环境变量中指定的任何级别（或 PAGE 或 ROW），若未设定，则通过 ONCONFIG 文件中的 DEF_TABLE_LOCKMODE 设置。CREATE TABLE 或 ALTER TABLE 语句的 LOCK MODE 子句可通过指定 PAGE 或 ROW 覆盖缺省的锁定粒度。仅行级锁支持 GBase 8s 的 LAST COMMITTED 特性。

然而，LOCK TABLE 语句通常锁定整个表，覆盖对该表的所有其他锁定粒度规范。

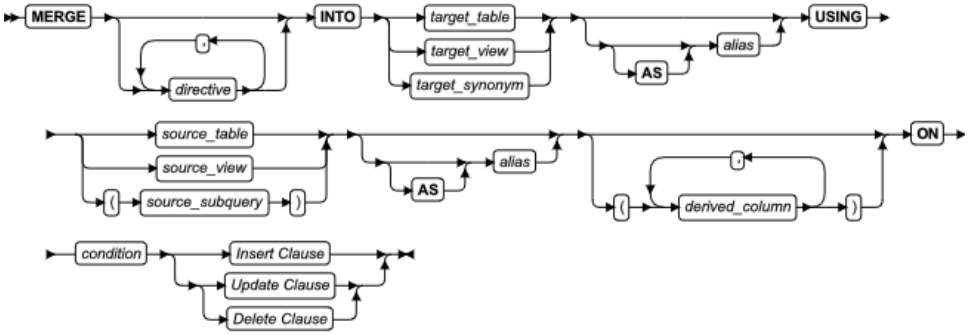
在所有这些上下文中，数据“锁模式”都表示***锁定粒度***。然而，在 SET LOCK MODE 语句的上下文中，“锁模式”指的是当进程尝试访问另一进程已经锁定的行或表时，数据库服务器的行为。

## 2.102 MERGE 语句

使用 MERGE 语句，通过在单个 SQL 语句内综合 UPDATE 或 DELETE 操作与 INSERT 操作，来将数据从源表转移到目标表内。您还可使用此语句来将源表与目标表合并，然后对目标表仅执行 UPDATE 操作，仅执行 DELETE 操作，或仅执行 INSERT 操作。

MERGE 语句以 GBase 8s 扩展支持 SQL 的 ANSI/ISO 标准。

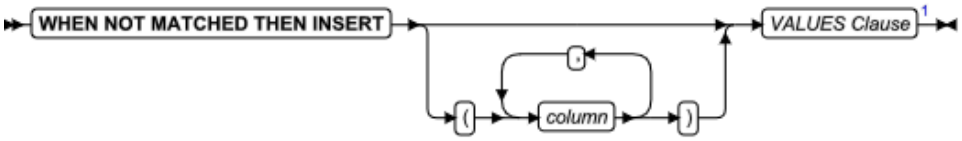
**语法**




Update 子句


Delete 子句


Insert 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 您为 ***target*** 或 ***source*** 表对象在此声明的临时名称 | 源别名与目标别名必须不同。如果有可能含糊，则在 *alias* 之前加上 AS 关键字。 | 标识符 |
| *column* | 向其内插入源数据的目标对象中的列 | 在目标对象中必须存在 | 标识符 |
| *condition* | 适用于 ***source*** 与 ***target*** 表的结合中的行的 Boolean 条件 | 可引用 ***source*** 和 ***target*** 对象中的数据值 | 条件 |
| *derived_column* | 如果该源对象为派生 | SET 和 VALUES 子句 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 的表，则您在此声明其名称 | 可引用此名称。 | |
| *directive* | 查询优化器命令 | 命令必须有效。 | 优化程序伪指令 |
| *source_table*, *source_view*, *source_subquery* | 包含要被重定位的数据的表（或查询的结果） | 对象必须存在。另请参阅 对 MERGE 的源表和目标表的限制。 | 数据库对象名；SELECT 语句 |
| *target_table*, *target_view*, *target_synonym* | 表的名称或同义词，或向其中插入、更新或删除数据的可更新的视图 | 请参阅 对 MERGE 的源表和目标表的限制。 | 数据库对象名 |

**用法**

GBase 8s 的 MERGE 语句是数据操纵语言(DML)语句，将源表对象与目标表或视图结合。您在 ON 关键字之后指定的 *condition* 决定在目标上的 UPDATE 或 DELETE 操作中使用源对象的哪一行，以及在目标上的 INSERT 操作中使用哪一行。MERGE 语句不更改源对象。

条件必须后跟 Delete 或 Update 子句的 WHEN MATCHED THEN 关键字，或后跟 Insert 子句的 WHEN NOT MATCHED THEN 关键字，或后跟 Update（或 Delete）以及 Insert 子句。

- 如果您指定 Update 和 Insert 两个子句，则 MERGE 语句可在目标对象上执行 INSERT 和 UPDATE 两个操作。
- 如果您指定 Delete 和 Insert 两个子句，则 MERGE 语句可在目标对象上执行 INSERT 和 DELETE 两个操作。
- 如果您未指定 Insert 子句，则不执行 INSERT 操作，但 Update 子句必须在满足该条件的源行的目标对象上指定 UPDATE 操作（抑或 Delete 子句必须指定 DELETE 操作）。
- 如果您未指定 Update 子句且未指定 Delete 子句，则不执行 UPDATE 或 DELETE 操作，但 Insert 子句必须为不匹配该对象的源行指定目标对象上的 INSERT 操作。

如果未指定 Delete 子句，未指定 Update 子句，且未指定 Insert 子句，则 MERGE 语句失败并提示错误。

MERGE 语句可对目标对象产生下列影响：

- 如果包括 Update 子句，则 MERGE 语句根据 SET 子句的规范，以源表中该条件取值为真的行的数据更新目标表或视图中的行。
- 如果包括 Delete 子句，则 MERGE 语句从目标表或视图删除该条件取值为真的行。
- 如果包括 Insert 子句，则 MERGE 语句根据 VALUES 子句的规范，以源表中该条件取值为假的行数据，向目标表或视图内插入新行。

然而，单个 MERGE 语句仅可有这三个影响中的两个，因为 Delete 子句与 Update 子句是互斥的。

对于大型表上的操作，请确保在您的系统上可获得这些资源：

- 足够数量的锁
- 用于中间的结合结果的足够的临时 dbspace 存储
- 用于 MERGE 语句的结果的充足的 dbspace 存储。

在高可用性集群配置中，您可从主服务器或从可更新的辅助服务器发出 MERGE 语句。

**优化器命令和子查询**

您可在 MERGE 关键字之后，可选地指定一个或多个查询优化器命令，诸如访问方法命令、结合顺序命令和结合方法命令来指定源表与目标表如何结合。在 MERGE 语句中，诸如 EXPLAIN 和 AVOID_EXECUTE 这样的面向目标的命令也有效。

在 MERGE 语句内，子查询还可包括优化器命令来控制该执行计划的其他方面。在 MERGE 语句中的下列上下文中子查询是有效的：

- 在 ON 子句的 ***condition*** 中
- 在 Update 子句的 SET 子句中
- 在 Insert 子句的 VALUES 子句中
- 在 USING 子句中如果指定源查询，可在 SELECT 语句支持子查询的地方包括任何上下文中的子查询。

然而，如果包括引用目标表的子查询，则 MERGE 语句失败并报错。

在支持外部命令的数据库中，查询优化器还可对源与目标表的外部结合应用外部命令，或对 MERGE 语句内的子查询。

跟在 ON 关键字之后的 *condition* 为源和目标表对象指定结合过滤器。 基于目标和源表的外部结合，此 ON 子句过滤器确定 MERGE 语句中匹配的行和不匹配的行。

- 如果 MERGE 语句包括 Update 子句，且 ON 子句条件取值为真，则在目标中更新相应的行。
- 如果 MERGE 语句包括 Delete 子句，且 ON 子句条件取值为真，则从目标删除相应的行。
- 如果 MERGE 语句包括 Insert 子句，且 ON 子句条件取值为假，则将相应的源行插入到目标内。

对与条件相匹配的行上的 MERGE 语句的 Update 操作，服从该 SET 子句的 UPDATE 语句规则。要获取在目标表中指定被更新的值的语法的详细信息，请参阅 SET 子句。

对与条件相匹配的行上的 MERGE 语句的 Delete 操作，服从 DELETE 语句规则。要了解从目标表删除值的详细信息，请参阅 使用 WHERE 关键字指定条件。

对与条件不相匹配的行上的 Insert 操作服从 VALUES 子句的 INSERT 语句规则。要了解将值插入到目标表中的语法的详细信息，请参阅 VALUES 子句。

**错误处理**

如果在 MERGE 语句正在执行过程中发生错误，则回滚整个语句。

对于支持事务日志记录的数据库，您可包括错误处理逻辑，包括包括定义一个或多个保存点的 MERGE 语句的事务中的 ROLLBACK TO SAVEPOINT 语句在该事务部分回滚到保存点之后，在目标表中保持 MERGE 语句的 INSERT、DELETE 或 UPDATE 操作的影响，如果在该事务的保存点级的语句的文本顺序中，该 MERGE 语句在保存点的前面的话。如果在该事务内，该 MERGE 语句跟在指定的保存点之后，则回滚 MERGE 的影响。

在符合 ANSI 的数据库中，数据操纵语言（DML）语句通常在事务之中。这些数据库不支持事务之外的 MERGE 语句。

**约束检查**

在 MERGE 操作中，强制对目标对象启用数据完整性约束。

- 如果检查模式设置为 DEFERRED，则直到提交该事务之后，才检查这些约束。
- 如果目标表的约束检查模式设置为 IMMEDIATE，则在所有 UPDATE（或 DELETE）和 INSERT 操作完成之后，检查唯一约束和引用约束。在 UPDATE、DELETE 和 INSERT 操作期间，检查 NOT NULL 和检查约束。

要获取关于设置约束检查模式的信息，请参阅主题 SET Transaction Mode 语句。

如果以 ON DELETE CASCADE 关键字定义了目标表上的引用约束，则 MERGE 语句的 DELETE 子句还对目标表的孩子表的行执行级联删除。

然而，如果启用了的引用约束已在目标和源表之间建立了父子关系，则 Delete 合并失败。MERGE 语句不可在其源表上执行级联删除。要获取更多信息，请参阅主题 表有级联删除时对 DELETE 的限制。

如果 START VIOLATIONS 语句已在目标表上定义了活动的违反表，则 MERGE 可对目标、违反和诊断表有下列影响：

- 或删除或更新与结合条件相匹配的目标表中的符合的行。
- 目标表还收到 MERGE 成功地插入的符合的不相匹配的行。
- 违反表收到不符合的行。
- 诊断表收到关于不符合的行不能满足约束的原因的信息，以及对目标表的 MERGE 操作期间的唯一索引的信息。

要在目标表上启用违反表和诊断表，SET Database Object Mode 语句必须设置约束或目标表的唯一索引为 ENABLED 或 FILTERING 模式。要了解更多信息，请参阅主题 与 SET Database Object Mode 语句的关系 和 SET Database Object Mode 语句。

**使用带有触发器的 MERGE 语句**

目标对象可为在其上定义 Update、Delete 或 Insert 触发器的表。如果在目标表上 Update 触发器和 Insert 触发器（或 Delete 触发器和 Insert 触发器）都启用，则 MERGE 作为两个触发器的触发事件，如果 MERGE 语句在目标上执行 UPDATE（或 DELETE）和 INSERT 操作的话。

如果 MERGE 语句包括激活 Update（或 Delete）和 Insert 触发器的操作，则当 MERGE 操作启动时，两个触发器的 BEFORE 触发器活动都执行。类似地，在 MERGE 操作的结尾，两个触发器的 AFTER 触发器活动都执行。每处理一行，都激活 FOR EACH ROW 触发器活动。

恰如对任何 DML 语句那样，数据库服务器将同一 MERGE 语句激活的所有触发器视同一个单个触发器，且结果触发器活动是合并的活动列表。控制一个触发器活动的所有规则适用于作为一个列表的合并的列表，且对这两个原始触发器一视同仁。要获取更多信息，请参阅 多个触发器的操作。

然而，目标对象不可为在其上定义启用的 INSTEAD OF 触发器的视图。在您可使用那个视图作为 MERGE 语句的目标之前，您必须禁用或删除该 INSTEAD OF 触发器。

在触发器的定义中，不可直接指定 MERGE 语句作为触发器活动。然而，在触发的活动中调用的 SPL 触发器例程可发出 MERGE 语句。

**安全策略和安全审计**

如果源对象或其任何列被基于标签的访问控制（LBAC）安全策略所保护，则发出该 MERGE 语句的用户必须有安全标签（或必须持有安全策略豁免），提供其足以在 MERGE 操作中读取该源表的凭证。

如果目标对象或其任何列被基于标签的安全策略所保护，则发出 MERGE 语句的用户必须有安全标签（或持有安全策略豁免），提供其足以在 SET 子句或 VALUES 子句指定的目标对象列中写的凭证，或足以从包括受保护的数据的目标删除行的凭证。

如果源和目标表都受到保护，则它们必须受到同一安全策略的保护。MERGE 语句不可结合那些受不同的 LBAC 安全策略保护的表。

在使用安全审计功能来记录活动的 GBase 8s 实例上，可能潜在地修改或显示数据或审计配置，在对 MERGE 语句的审计跟踪中未定义特定的审计事件助记符：

- Delete 子句指定的活动记录为 DELETE 事件。
- Insert 子句指定的活动记录为 INSERT 事件。
- Update 子句指定的活动记录为 UPDATE 事件。

## 对 MERGE 的源表和目标表的限制

哪些表对象可为 MERGE 语句的源或目标，这取决于该表对象的属性，以及发出 MERGE 语句的对象所持有的访问权限。

对于当前会话连接到的数据库，目标表必须为本地的，但您可指定一远程表作为源表，或在 UPDATA 操作的 SET 子句的子查询中，或在 INSERT 操作的 VALUES 子句的子查询中。

下列章节标识对源表和目标表的附加的限制。

**对源表的限制**

源对象可为 STANDARD、RAW、TEMP、EXTERNAL 或集合派生的表或视图的名称或同义词。它可与目标对象在同一数据库中，或在本地 GBase 8s 实例的不同数据库中，它或可为由不同的 GBase 8s 实例管理的远程表。

如果源是由查询的结果所定义的集合派生的表，则 USING 子句可声明派生的列的名称，MERGE 语句的 SET 和 VALUES 子句可引用这些列。

发出 MERGE 语句的用户必须持有对源对象的数据库的 Connect 访问权限（或更高的权限），其必须还持有对源对象的 Select 权限（或更高的权限）。可分别地授予该用户这些访问权限，或该用户作为 PUBLIC 组的成员而持有它们，或如果当前的或缺省的角色或 PUBLIC 持有那些权限，则可通过用户的角色持有权限。

如果源对象或其任意列受到基于标签的安全策略保护，则发出 MERGE 语句的用户必须有提供其读取该源对象的充足凭证的安全标签（或必须持有安全策略豁免）。如果该用户的凭证不足以读取受保护的列，根据标准基于标签的访问控制（LBAC）规则，则该 MERGE 语句仅可处理该源数据的一个子集。如果此子集为空，则 MERGE 语句不可从源对象向目标表内插入任何值。

下列限制适用于源表对象：

- 源不可为在其上定义启用的 SELECT 触发器的视图。
- 源不可为与目标表在同一表层级中的类型化表
- 在 Delete 合并中，源不可与目标有孩子表关系，由启用的引用约束定义，如果以 ON DELETE CASCADE 关键字定义了那个约束的话。（然而，孩子表关系不影响 Delete 合并，除非目标表约束指定级联删除。）

**对目标表的限制**

目标表对象必须在当前会话连接到的同一 GBase 8s 实例的数据库中。它可为 STANDARD、RAW 或 TEMP 表，或可更新的视图的名称或同义词。如果目标在表层级内为可更新的，则 Delete 子句还删除该目标表的所有子表中相应的行。

发出 MERGE 语句的用户必须持有对目标对象的数据库的 Connect 访问权限（或更高的权限），且必须持有对目标对象的 Insert 权限和 Update 或 Delete 权限，如果 MERGE 语句包括相应的 Insert、Update 或 Delete 子句的话。

下列限制适用于 MERGE 语句的目标表。如果那个表有任何下列的属性，则 MERGE 操作返回错误。

- 目标不可为该源表所在同一表层级中的类型化表。
- 目标不可为“虚拟表接口”（VTI）表。
- 目标不可为 CREATE EXTERNAL TABLE 语句定义的对象。
- 目标不可在远程 GBase 8s 实例的数据库中。
- 目标不可为系统目录表。
- 目标不可为在其上定义启用的 INSTEAD OF 触发器的视图。
- 目标不可为只读视图。
- 目标不可为伪表（在系统数据库中的常驻内存的对象，比如 **sysmaster** 或 **sysadmin** 数据库）。

- 目标不可为同一 MERGE 语句的任何子查询的数据源，包括 ON 子句中的、SET 子句中的或 VALUES 子句中的子查询。
- 如果 MERGE 语句包括 DELETE 子句，则目标不可与源表有父表关系，如果通过指定 ON DELETE CASCADE 关键字的启用的引用约束定义此关系的话。

**合并的行长度的限制**

MERGE 语句合并的源表和目标表的行长度（= 源表的行大小 + 目标表的行大小）不能大于 40 M 字节。否则，MERGE 语句失败并报错。

**对分布式 MERGE 语句的限制**

如果源表与目标表不在同一数据库中，则两个数据库都必须满足对跨数据库和跨服务器 DML 操作的兼容性要求：

- 如果一个数据库符合 ANSI，则其他数据库必须也符合 ANSI。
- 如果一个数据库不符合 ANSI 但是用显式的事务日志记录，则其他数据库必须也支持显式的事务日志记录。
- 如果一个数据库不支持事务日志记录，则其他数据库也必须不支持。
- 两个数据库必须有相同的 NLSCASE 敏感性设置。

例如，分布式 MERGE 语句不可在区分大小的数据库中指定源表，而在创建为 NLSCASE INSENSITIVE 的数据库中指定目标表，不论表是否包括 NCHAR 或 NVARCHAR 列。

## 处理重复的行

在执行 MERGE 时，目标表中的同一行不可被更新或被删除一次以上。在执行了 MERGE 语句之前，请勿尝试更新或删除尚未存在的目标中的任何行。即，没有同一 MERGE 语句插入到目标内的行的更新或删除。

下列 MERGE 语句的示例使用事务表 new_sale 作为源表，从其来在事实表中插入或更新行。在此样例中的结合条件测试 new_sale.cust_id 列值与 sale.cust_id 列值是否相匹配。

```
MERGE INTO sale USING new_sale AS n
    ON sale.cust_id = n.cust_id
    WHEN MATCHED THEN UPDATE
        SET sale.salecount = sale.salecount + n.salecount
    WHEN NOT MATCHED THEN INSERT (cust_id, salecount)
        VALUES (n.cust_id, n.salecount);
```

要执行此 MERGE 语句，数据库服务器结合目标表和源表，并应用指定的相等条件来处理结合的结果：

- 对于满足条件的行（因为 **sale.cust_id** 值与 **new_sale.cust_id** 值相匹配），MERGE 根据 SET 子句的指定，更新 **sale.salecount** 列值。
- 对于不满足条件的行（因为在 **sale** 表中没有行有与 **new_sale.cust_id** 相同的 **cust_id** 值），MERGE 根据 VALUES 子句的指定，将包含 **new_sale.cust_id** 和 **new_sale.salecount** 值的新行插入 **sale** 表内。

对于先前示例中的 MERGE 语句，假设 sale 目标包含两条记录，而 new_sale 源表包含三条记录。

表 1. “sale”表中的记录

| cust_id | sale_count |
|---|---|
| Tom | 129 |
| Julie | 230 |

表 2. “new_sale”表中的记录

| cust_id | sale_count |
|---|---|
| Tom | 20 |
| Julie | 3 |
| Julie | 10 |

当通过指定表达式 sale.cust_id = new_sale.cust_id 作为匹配条件将 **new_sale** 合并至 **sale** 内时，MERGE 语句返回错误，因为它尝试超过一次更新 **sale** 目标表中的记录之一。

**分布式 MERGE 操作中的数据类型**

如果源表或视图（或任何在源查询中引用的表对象）指定在 GBase 8s 实例的数据库中的表对象，而不是管理目标表的数据库的本地实例，则 MERGE 语句仅可访问远程数据库中的下列数据类型的列：

- 非 opaque 的内建的数据类型
- BOOLEAN
- LVARCHAR
- 非 opaque 的内建的数据类型的 DISTINCT
- BOOLEAN 的 DISTINCT
- LVARCHAR 的 DISTINCT
- 出现在此列表中的任何 DISTINCT 数据类型的 DISTINCT。

跨服务器的分布式 MERGE 操作可支持这些 DISTINCT 类型，仅当将 DISTINCT 类型显式地强制转型为内建的类型，且在每一参与的数据库中以完全相同的方式定义所有 DISTINCT 类型、其数据类型层级及其强制转型。要获取关于 GBase 8s 在跨服务器 DML 操作中支持的数据类型的附加信息，请参阅 跨服务器事务中的数据类型。

MERGE 不可访问另一 GBase 8s 实例的数据库，除非两个数据库实例都支持 TCP/IP 或 IPCSTR 连接，这定义在它们的 DBSERVERNAME 或 DBSERVERALIASES 配置参数中，以及在 **sqlhosts** 文件或 SQLHOSTS 注册子键中。此连接类型的要求适用于 GBase 8s 实例之间的任何通信，即使两个数据库服务器位于同一台计算机上。

然而，访问在本地 GBase 8s 实例的其他数据库中的表对象的 MERGE 操作，可访问前述列表中任何跨服务器的数据类型以及这些附加的数据类型：

- 大部分内建的 opaque 数据类型，如 跨数据库事务中的数据类型 中所列

- 同一内建的 opaque 类型的 DISTINCT
- 在前面两行中的任何数据类型的 DISTINCT
- 显式地强制转型为内建的数据类型的 opaque 用户定义的数据类型（UDT）。

MERGE 语句还支持通用客户端 API 中的 “分布式关系数据库架构”™（DRDA®）协议。对于 MERGE 可从远程数据库通过 DRDA 协议返回的 GBase 8s 数据类型，要查看 DRDA 所支持（以及不支持）的 GBase 8s 数据类型的列表，请参阅 *GBase 8s 管理员指南*。

## MERGE 语句的示例

在此部分中的示例包括在结合的结果集上展现结合条件和各种 DML 操作的 MERGE 语句。

### 示例

下列 MERGE 语句包括 Update 和 Insert 子句，并使用相等谓词作为结合条件：

```
MERGE INTO customer c
    USING ext_customer e
    ON c.customer_num=e.customer_num
WHEN MATCHED THEN
    UPDATE SET c.fname = e.fname,
               c.lname = e.lname,
               c.company = e.company,
               c.address1 = e.address1,
               c.address2 = e.address2,
               c.city = e.city,
               c.state = e.state,
               c.zipcode = e.zipcode,
               c.phone = e.phone
WHEN NOT MATCHED THEN
    INSERT (c.fname, c.lname, c.company, c.address1, c.address2,
               c.city, c.state, c.zipcode, c.phone)
          VALUES
             (e.fname, e.lname, e.company, e.address1, e.address2,
               e.city, e.state, e.zipcode, e.phone);
```

下一示例在 ON 子句中指定多个谓词：

```
MERGE INTO customer c
    USING ext_customer e
    ON c.customer_num=e.customer_num
        AND c.fname=e.fname AND c.lname=e.lname
WHEN MATCHED THEN
    UPDATE SET c.fname = e.fname,
               c.lname = e.lname,
               c.company = e.company,
               c.address1 = e.address1,
               c.address2 = e.address2,
               c.city = e.city,
```

```
            c.state = e.state,
            c.zipcode = e.zipcode,
            c.phone = e.phone
WHEN NOT MATCHED THEN
    INSERT
        (c.fname, c.lname, c.company, c.address1, c.address2,
         c.city, c.state, c.zipcode, c.phone)
        VALUES
        (e.fname, e.lname, e.company, e.address1, e.address2,
         e.city, e.state, e.zipcode, e.phone);
```

下列 MERGE 语句执行 Update 结合，不带 Insert 子句：

```
MERGE INTO customer c
USING ext_customer e
ON c.customer_num=e.customer_num
WHEN MATCHED THEN
UPDATE SET c.fname = e.fname,
            c.lname = e.lname,
            c.company = e.company,
            c.address1 = e.address1,
            c.address2 = e.address2,
            c.city = e.city,
            c.state = e.state,
            c.zipcode = e.zipcode,
            c.phone = e.phone ;
```

下列 MERGE 语句仅在结合条件之后包括 Delete 子句：

```
MERGE INTO customer c
    USING ext_customer e
        ON c.customer_num=e.customer_num
        WHEN MATCHED THEN
            DELETE ;
```

下一 MERGE 示例仅包括 Insert 子句：

```
MERGE INTO customer c
USING ext_customer e
ON c.customer_num=e.customer_num AND c.fname=e.fname
    AND c.lname=e.lname
WHEN NOT MATCHED THEN
    INSERT
        (c.fname, c.lname, c.company, c.address1, c.address2,
         c.city, c.state, c.zipcode, c.phone)
         VALUES
          (e.fname, e.lname, e.company, e.address1, e.address2,
           e.city, e.state, e.zipcode, e.phone);
```

下一示例展示 WHEN MATCHED 和 WHEN NOT MATCHED 规范可以任何顺序出现：

```
MERGE INTO customer c
    USING ext_customer e
    ON c.customer_num=e.customer_num AND c.fname=e.fname AND c.lname=e.lname
WHEN NOT MATCHED THEN
        INSERT
            (c.fname, c.lname, c.company, c.address1, c.address2,
             c.city, c.state, c.zipcode, c.phone)
        VALUES
            (e.fname, e.lname, e.company, e.address1, e.address2,
             e.city, e.state, e.zipcode, e.phone)
WHEN MATCHED THEN UPDATE
          SET c.fname = e.fname,
                c.lname = e.lname,
                c.company = e.company,
                c.address1 = e.address1,
                c.address2 = e.address2,
                c.city = e.city,
                c.state = e.state,
                c.zipcode = e.zipcode,
                c.phone = e.phone ;
```

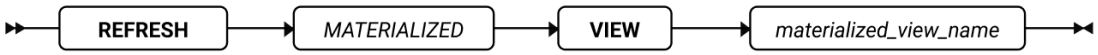
下列 MERGE 指定在 USING 子句中查询定义的派生的表为其源：

```
MERGE INTO customer c
    USING (SELECT * from ext_customer e1, orders e2
              WHERE e1.customer_num=e2.customer_num ) e
    ON c.customer_num=e.customer_num AND c.fname=e.fname
        AND c.lname=e.lname
    WHEN NOT MATCHED THEN
        INSERT (c.fname, c.lname, c.company, c.address1, c.address2,
                    c.city, c.state, c.zipcode, c.phone)
        VALUES (e.fname, e.lname, e.company, e.address1, e.address2,
                    e.city, e.state, e.zipcode, e.phone)
    WHEN MATCHED THEN
        UPDATE SET c.fname = e.fname,
                c.lname = e.lname,
                c.company = e.company,
                c.address1 = e.address1,
                c.address2 = e.address2,
                c.city = e.city,
                c.state = e.state,
                c.zipcode = e.zipcode,
                c.phone = e.phone ;
```

## 2.103 REFRESH MATERIALIZED VIEW

使用调用 refresh materialized view 物化视图名；来完成物化视图刷新。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| materialized_view_name | 物化视图名称 | 数据库中存在 | 标识符 |

## 用法

要刷新物化视图，您必须是所有者或拥有 DBA 特权。

目前物化视图暂不支持 DML 操作，只能通过 refresh 语法刷新物化视图。

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

例如下面示例:

```
create materialized view    mv_test_nextrefresh
refresh on demand start with sysdate NEXT sysdate+1/24/60/3
as
select * from teacher where id > 6;
```

查询刚刚创建的物化视图

```
select * from mv_test_nextrefresh;
```

返回结果如下:

```
    id    name    subject   id_group
    7     a       v         1
    8     b       w         1
    9     c       x         1
10  d       y         1
```

对基表插入一条新的数据

```
insert into teacher values(11,'d','d',1);
```

然后执行

```
refresh materialized view mv_test_nextrefresh;
```

对物化视图进行刷新,显示执行成功之后

查询物化视图

```
select * from mv_test_nextrefresh;
```

返回结果如下:

```
        id    name    subject   id_group
        7     a       v         1
        8     b       w         1
```

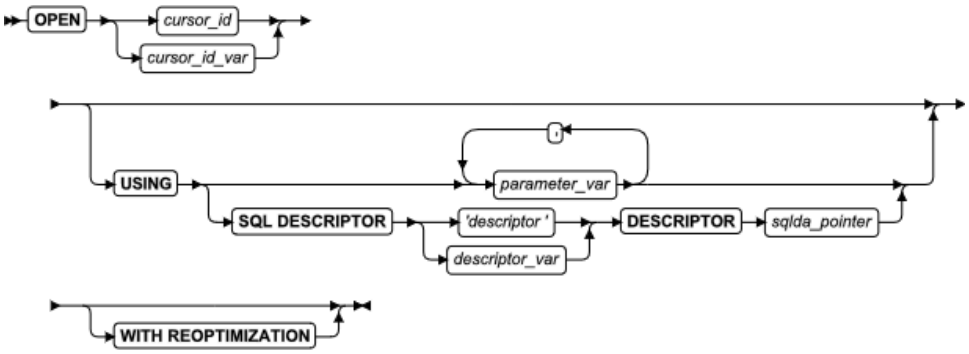
```
        9   c       x       1
        10  d       y       1

11  e       z       1
```

说明刷新物化视图成功

## 2.104 OPEN 语句

使用 OPEN 语句来激活游标。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 游标的名称 | 必须已由 DECLARE 语句声明 | 标识符 |
| *cursor_id_var* | 主变量 = *cursor_id* | 必须为字符数据类型 | 特定于语言 |
| *descriptor* | 系统描述符区域的名称 | 必须已分配 | 引用字符串 |
| *descriptor_var* | 标识系统描述符区域的主变量 | 必须已分配了系统描述符区域 | 引用字符串 |
| *parameter_var* | 在准备好的 SQL 语句中以其内容替代问号（?）占位符的主变量 | 必须为字符或集合数据类型 | 特定于语言 |
| *sqlda_pointer* | 指向 sqlda 结构的指针，定义数据类型和在准备好的语句中要代替问号（?）的值的内存位置 | 不可以美元（$）符号或冒号（:）开头。您必须以动态的 SQL 语句使用 sqlda 结构。 | DESCRIBE 语句 |

**用法**

请随同 GBase 8s ESQL/C 或随同 SPL 使用此语句。

***游标***是与返回有序的值集的 SQL 语句相关联的标识符。OPEN 语句激活 DELARE 语句定义的游标。

可通过游标相关联的 SQL 语句进行归类：

- ***Select 游标***：与 SELECT 语句相关联的游标
- ***Function 游标***：与 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句相关联的游标
- ***Insert 游标***：与 INSERT 语句相关联的游标
- ***Collection 游标***：在集合变量上操作的 Select 或 Insert 游标。

在以 SPL 语言编写的 UDR 中，OPEN 语句仅可引用 Select 或 Function 游标，且这些必须指定游标的标识符，而不是存储 ***cursor_id*** 的变量。OPEN 语句不可引用 SPL 的 FOREACH 语句已经声明的直接游标。

依赖于游标相关联的语句，数据库服务器执行的特定操作有所不同。在 ESQL/C 中，当您将先前的语句与游标直接地关联（即，您未准备该语句，且将该语句标识符与游标关联），OPEN 语句隐式地准备该语句。（这不是 SPL 例程中 OPEN 的特性，在 SPL 例程中 DECLARE 语句将游标与现有的准备好的语句的标识符相关联，而不是直接地与 SQL 语句文本相关联。）

在符合 ANSI 的数据库中，如果您试图打开一已经打开的游标，则会收到错误代码。

## 打开 Select 游标

当您打开以 SELECT… FOR UPDATE 语法创建的 Select 游标或更新游标时，将 SELECT 语句以 USING 子句中指定的任何值传递到数据库服务器。数据库服务器处理该查询到定位或构造活动集合的第一行的点。下列示例展示 GBase 8s ESQL/C 中的简单 OPEN 语句：

```
EXEC SQL declare s_curs cursor for select * from orders;
        EXEC SQL open s_curs;
```

SPL 例程不可引用 OPEN 语句中的更新游标。

## 在事务内部打开 Update 游标

如果您正在带有显式事务的数据库中工作，则必须在事务内打开更新游标。如果您使用 WITH HOLD 选项声明了该游标，则放弃此要求。

## 打开 Function 游标

当您打开 Function 游标时，随同在 USING 子句中指定的任何值，将 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句传递到数据库服务器。

将 USING 子句中的值作为参数传递给用户定义的函数。必须声明此用户定义的函数来接受值。（如果先前准备了该语句，则在准备的时候将该语句传递给数据库服务器。）数据库服务器执行该函数来指出它返回第一个值集的位置。

下列示例展示 GBase 8s ESQL/C 中的一个简单的 OPEN 语句：

```
EXEC SQL declare s_curs cursor for
    execute function new_func(arg1,arg2)
    into :ret_val1, :ret_val2;
EXEC SQL open s_curs;
```

## 重新打开 Select 或 Function 游标

仅当数据库服务器打开 Select 游标或 Function 游标时，它才计算在 OPEN 语句的 USING 子句中指定的值。在打开游标时，在 USING 子句中对程序变量的后续更改不更改游标的活动集合。

在符合 ANSI 的数据库中，如果您试图打开一已打开的游标，则会收到错误代码。

在不符合 ANSI 的数据库中，后续的 OPEN 语句关闭该游标，然后重新打开它。当数据库服务器重新打开游标时，它基于 USING 子句中变量的当前值创建新的活动集合。如果自从先前的 OPEN 语句以来这些变量已经更改，则重新打开该游标可生成一完全不同的活动集合。

即使这些变量的值未改变，在下列情况下，该活动集合中的值也可不同：

- 如果用户定义的函数采用了与在 Function 游标上的先前的 OPEN 语句不同的执行路径
- 自从 Select 游标上的先前的 OPEN 语句以来，如果表中的数据更改了

当数据库服务器打开 Select 或 Function 游标时，它可动态地处理大多数查询，无需预先取回所有行。因此，如果其他用户在同一时间正在修改该用表正在处理的表，则活动集合可能反映这些活动的结果。

对于一些查询，当数据库服务器打开该游标时，它计算整个活动集合。这些查询包括具有下列特性的那些：

- 要求排序的查询：那些带有 ORDER BY 子句或带有 DISTINCT 或 UNIQUE 关键字的
- 要求散列的查询：那些带有结合或带有 GROUP BY 子句的

对于这些查询，在游标正在处理时其他用户对该表的任何更改都不反映在活动集合中。

## 与 Select 和 Function 游标相关的错误

由于数据库服务器首次看到该查询，它可能检测错误。在这种情况下，它不实际地返回数据的第一行，但它重置 **SQLCODE** 变量和 **sqlca** 的 **sqlca.sqlcode** 字段。该值或为负数或为零，如下表所示。

| 代码值 | 含义 |
|---|---|
| 负数 | 在 SELECT 语句中检测到错误 |
| 零 | SELECT 语句有效 |

与 ESQL/C 例程不同，SPL 例程没有对 **sqlca** 结构的直接访问。ESQL/C 例程必须显式地调用内建的 **SQLCODE** 函数来访问与 OPEN 引用的游标相关联的 SELECT、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句的返回代码。

如果 SELECT、SELECT…FOR UPDATE、EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句有效，但没有与其条件相匹配的行，则第一个 FETCH 返回值 100（SQLNOTFOUND），表示未找到行。

提示： 当您遇到 **SQLCODE** 错误时，还存在相应的 **SQLSTATE** 错误值。要了解关于如何查看消息文本的信息，请参考 使用 SQLSTATE 错误状态代码。

## 打开 Insert 游标（ESQL/C）

当您打开 Insert 游标时，该游标将 INSERT 语句传递给数据库服务器，数据库服务器检查关键字和列名称的有效性。数据库服务器还为插入缓冲区分配内存来保留新数据。（请参阅 DECLARE 语句。）

对与 INSERT 语句相关联的游标的 OPEN 语句包括 USING 子句。

## 打开 Insert 游标的示例

下列 GBase 8s ESQL/C 示例展示带有 Insert 游标的 OPEN 语句：

```
EXEC SQL prepare s1 from
    'insert into manufact values ('npr', 'napier')';
EXEC SQL declare in_curs cursor for s1;
EXEC SQL open in_curs;
EXEC SQL put in_curs;
EXEC SQL close in_curs;
```

## 重新打开 Insert 游标

当您重新打开已打开的 Insert 游标时，会有效地刷新插入缓冲区；存储在插入缓冲区中的任何行都会写到数据库表内。数据库服务器首先关闭导致刷新的游标，然后重新打开该游标。要获取关于如何检查错误和对插入的行计数的信息，请参阅 错误检查。

在符合 ANSI 的数据库中，如果您试图打开一已打开的游标，则会收到错误代码。

## 打开 Collection 游标（ESQL/C）

您可声明集合变量上的 Select 和 Insert 游标。这些游标称为 ***Collection 游标***。您必须使用 OPEN 语句来激活这些游标。

使用 OPEN 语句的 USING 子句中的集合变量的名称。要获取更多关于以集合变量使用 OPEN ... USING 的信息，请参阅 从集合游标访存 和 插入到 Collection 游标内。

## USING 子句

当游标与一个包括问号（?）占位符和准备好的语句相关联时，需要 USING 子句，如下：

- 在其 WHERE 子句中带有输入参数的 SELECT 语句
- 带有输入参数作为其用户定义的函数的参数的 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句
- 在其 VALUES 子句中带有输入参数的 INSERT 语句（在 ESQL/C 中）。

在 SPL 例程中，您必须指定这些参数作为 SPL 变量。

在 ESQL/C 中，您可以下列方式之一支持这些参数的值：

- 您可指定一个或多个主变量。
- 您可指定系统描述符区域。
- 您可指定指向 **sqlda** 结构的指针。

要获取更多信息，请参阅 PREPARE 语句。

如果您知道在运行时要提供的参数的数目和顺序及其数据类型，则可在程序中通过该语句将所需参数定义为主变量。通过以 USING 关键字打开游标，您按位置将参数传递给数据库服务器，后跟依其序列顺序的变量的名称。从左至右，这些变量以一一对应的方式与 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句问号（?）占位符相匹配。

在变量的列表中，您不可包括 ESQL/C 的指示符变量。要使用指示符变量，您必须包括 SELECT 或 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句文本作为 DECLARE 语句的一部分，而不是准备好的语句的标识符。

您必须为每一占位符提供一个主变量名称。每一变量的数据类型必须与准备好的语句所需要的相应的类型相兼容。下列 GBase 8s ESQL/C 代码片段打开 Select 游标并在 USING 子句中指定主变量：

```
sprintf (select_1, "%s %s %s %s %s",
    "SELECT o.order_num, sum(total price)",
    "FROM orders o, items i",
    "WHERE o.order_date > ? AND o.customer_num = ?",
    "AND o.order_num = i.order_num",
    "GROUP BY o.order_num");
EXEC SQL prepare statement_1 from :select_1;
EXEC SQL declare q_curs cursor for statement_1;
EXEC SQL open q_curs using :o_date, :o.customer_num;
```

下列示例展示在 GBase 8s ESQL/C 代码片段中，带有 EXECUTE FUNCTION 语句的 OPEN 语句的 USING 子句：

```
stcopy ("EXECUTE FUNCTION one_func(?, ?)", exfunc_stmt);
EXEC SQL prepare exfunc_id from :exfunc_stmt;
EXEC SQL declare func_curs cursor for exfunc_id;
EXEC SQL open func_curs using :arg1, :arg2;
```

## 指定系统描述符区域（ESQL/C）

如果您不知道在运行时要提供的参数的数目及其数据类型，则可从系统描述符区域关联输入值。系统描述符区域描述要代替问号（?）占位符的一个或多个值的数据类型和内存位置。

系统描述符区域符合 X/Open 标准。

使用 SQL DESCRIPTOR 关键字来引入系统描述符区域的名称作为参数的位置。

系统描述符中的 COUNT 字段对应于准备好的语句中的动态参数的数目。COUNT 的值必须小于或等于当以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域时指定的项描述符的数目。您可以 GET DESCRIPTOR 语句获取字段的值，并以 SET DESCRIPTOR 语句设置该值。

下列示例展示 OPEN ... USING SQL DESCRIPTOR 语句：

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL open selcurs using sql descriptor 'desc1';
```

正如该示例显示的，在 OPEN 语句中引用系统描述符区域之前，您必须分配它。

## 指定指向 sqlda 结构的指针（ESQL/C）

如果您不知道在运行时要提供的参数的数目，或其数据类型，可从 **sqlda** 结构关联输入值。**sqlda** 结构罗列要替代问号（?）占位符的一个或多个值的数据类型和内存位置。

使用 DESCRIPTOR 关键字来引入指向 **sqlda** 结构的指针，作为参数的位置。

**sqlda** 值指定在 **sqlvar** 的并发中描述的输入值的数目。此数目必须对应于准备好的语句中的动态参数的数目。

## 指定指向 sqlda 结构的指针的示例

下列示例展示 OPEN ... USING DESCRIPTOR 语句：

```
struct sqlda *sdp;
...
EXEC SQL open selcurs using descriptor sdp;
```

## 使用 WITH REOPTIMIZATION 选项（ESQL/C）

使用 WITH REOPTIMIZATION 关键字来重新优化您的查询计划。当您准备 SELECT、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句时，数据库服务器使用查询计划来优化该查询。如果后来您修改与准备好的语句相关联的数据，则可折中那个语句的查询的有效性。换句话说，如果您更改数据，您可能重新优化您的查询。要确保您的查询的优化，可再次准备该语句，或使用 WITH REOPTIMIZATION 选项再次打开游标。

通常您应使用 WITH REOPTIMIZATION 选项，因为它提供下列较再次准备语句的优势：

- 仅重建查询计划，而不是整个语句
- 使用较少的资源
- 减少开销
- 需要更少的时间

在处理 OPEN 游标语句之前，WITH REOPTIMIZATION 选项强制数据库服务器来优化查询设计计划。

下列示例使用 WITH REOPTIMIZATION 关键字：

```
EXEC SQL open selcurs using descriptor sdp with reoptimization;
```

## OPEN 与 FREE 之间的关系

数据库服务器为准备好的语句分配资源并打开游标。如果您执行 FREE ***statement_id*** 或 FREE ***statement_id_var*** 语句，则仍可打开与释放的语句 ID 相关联的游标。然而，如果您以 FREE ***cursor_id*** 或 FREE ***cursor_id_var*** 语句释放资源，则不可使用游标，除非您在此声明该游标。

类似地，如果您对一个或多个游标使用 SET AUTOFREE 语句，则当程序关闭特定的游标时，数据库服务器自动地释放与游标相关的资源。在这种情况下，您不可使用游标，除非您再次声明该游标。

## 对游标引用的表的 DDL 操作

各种 DDL 语句可删除、重命名或更改在定义游标的 DECLARE 语句中直接地（或通过准备好的语句的标识符间接地）引用的表的模式。对该游标的后续的 OPEN 操作可能失败并报 error -710，或可能产生意外的结果。更改列的数目或列的数据类型会有此影响，且对引用其模式已被修改的表的任何 SPL 例程，用户通常必须重新发出 DESCRIBE 语句、PREPARE 语句和（对于与例程关联的游标）UPDATE STATISTICS 语句。

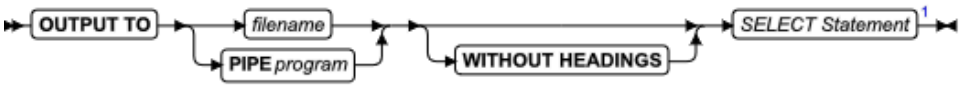
然而，当为准备好了的对象和对引用表的 SPL 例程（例程引用那些 ALTER TABLE、CREATE INDEX 或 DROP INDEX 操作已经修改了的表）启用了自动重新编译时，如果添加或删除索引，则这些限制不适用。这是 GBase 8s 的缺省的行为。要获取更多关于在模式更改之后启用或禁用自动重新编译的更多信息，请参阅对 SET ENVIRONMENT 语句的 IFX_AUTO_REPREPARE 选项的描述。要获取更多关于 AUTO_REPREPARE 配置参数的信息，请参阅 *GBase 8s 管理员参考手册*。

然而，当 AUTO_REPREPARE 配置参数和 IFX_AUTO_REPREPARE 会话环境变量设置为禁用准备好的对象的重新编译时，将索引添加到在 DECLARE 语句中直接地或间接地引用的表，可类似地使相关的游标无效。指定该无效游标的后续的 OPEN 语句失败，即使它们包括 WITH REOPTIMIZATION 关键字。在禁用自动重新编译时，如果将索引添加到与游标相关联的表，则在您打开该游标之前，必须再次准备该语句并再次声明该游标。对于与调用 SPL 例程相关联的游标，对于那些引用已经添加或删除了索引的表的例程，您必须运行 UPDATE STATISTICS 语句。您不可简单地重新打开基于不再有效的准备好的语句的游标。

# 2.105 OUTPUT 语句

使用 OUTPUT 语句来将查询的结果发送到操作系统文件或程序。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *filename* | 写查询结果的位置的路径和文件名。缺省的路径为当前的目录。 | 可指定新的或现有的文件。如果文件存在，则查询结果覆盖该文件的当前的内容。 | 必须符合您的操作系统的规则。 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *program* | 要接收查询结果作为输入的程序的名称。 | 程序必须存在，必须为操作系统所知，且必须能读取查询的结果。 | 必须符合您的操作系统的规则。 |

**用法**

OUTPUT 语句将查询结果写到操作系统文件中，或将查询结果管道到另一程序。您可可选地指定从查询输出省略列标题。此语句为 SQL 的 ANSI/ISO 标准的扩展。您仅可随同 DB-Access 使用此语句。

### 将查询结果发送到文件

要将查询的结果发送到操作系统文件，请指定该文件的全路径名。如果该文件已存在，则输出覆盖当前的内容。

下列示例展示如何将查询的结果发送到操作系统文件。该示例使用 UNIX™ 文件命名约定。

```
OUTPUT TO /usr/april/query1
    SELECT * FROM cust_calls WHERE call_code = 'L'
```

### 显示无列标题的查询结果

要显示无列标题的查询的结果，请使用 WITHOUT HEADINGS 关键字。

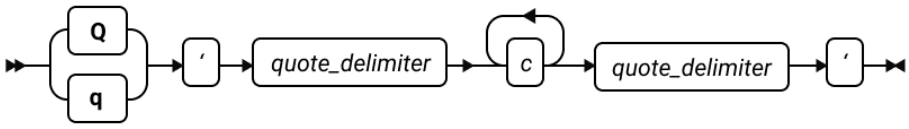
### 将查询结果发送给另一程序

您可使用关键字 PIPE 来将查询结果发送到另一程序，如下例所示：

```
OUTPUT TO PIPE more
    SELECT customer_num, call_dtime, call_code
        FROM cust_calls;
```

## 2.106 Q 转义字符语句

使用 Q 前缀加单引号再加分隔符的形式实现字符的转义，分隔符包围的字符为需要转义的部分。分隔符支持字母、数字、下划线、可见字符。Q 前缀加单引号再加分隔符的形式是对 SQL 中单引号转义的扩展，但凡支持单引号转义的语法均支持替换成此种形式。




用法：Q 前缀不区分大小写，Q 与 q 执行结果一样；Q 作为前缀指示转义即将开始，后面的两个单引号在前面和后面成对使用；quote_delimiter 分隔符可以是任意的单字节或者多字节字符，但不能是空格、TAB 制表符、回车符。如果分隔符也出现在需要转义的字符 c 中，要避免在此字符后紧接出现一个单引号；以下字符作为分隔符使用时需要区分左右：[] {} <> ()。

例如：

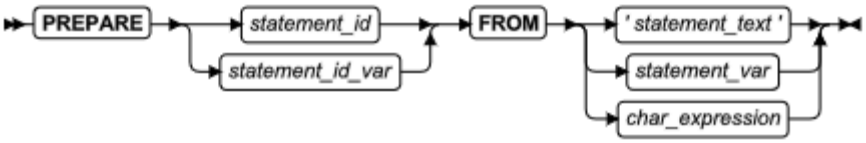
```
SELECT Q'<what's new>' FROM dual
```

返回结果为：what's new

## 2.107 PREPARE 语句

使用 PREPARE 语句可在运行时解析、验证和生成一个或多个 SQL 语句的执行计划。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *char_expression* | 计算得到单个 SQL 语句的文本的表达式 | 必须为 SELECT、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句 | 表达式 |
| *statement_id* | 为准备好的对象在此声明的标识符 | 在游标和准备好的对象的名称之中（以及 SPL 中，在变量之中）必须是唯一的。 | 标识符 |
| *statement_id_var* | 存储 *statement_id* 的主变量 | 必须先前已声明为字符数据类型 | 特定于语言 |
| *statement_text* | 要准备的 SQL 语句的文本 | 请参阅 准备多个 SQL 语句 和 语句文本。 | 引用字符串. |
| *statement_var* | 存储一个或多个 SQL 语句的主变量 | 必须为字符数据类型。如果 SQL 语句包含“集合派生的表”段，则无效。 | 特定于语言 |

**用法**

请在 ESQL/C 或 SPL 例程中使用此语句。

PREPARE 语句启用您的程序来在运行时收集一个（或对于 ESQL/C，多于一个）SQL 语句的文本，来声明结果的***准备好的对象***的标识符，并使之可运行。以三个步骤实现此 SQL 的动态形式：

1. PREPARE 语句接收语句文本为输入，或作为引用的字符串，或 ESQL/C 字符变量，或（在 SPL 中）作为字符表达式计算的值。语句文本可包含问号（?）占位符来表示要到执行该语句时定义的值。

2. OPEN 语句（以及在 ESQL/C 例程中，EXECUTE 语句）可提供所需要的输入值并一次或多次执行准备好的语句。

3. 稍后，可使用 FREE 语句释放那些分配给准备好的语句的资源。

要获取更多关于在准备好的语句中以运行时的值替换占位符的信息，请参阅章节 准备接收参数的语句。

当您创建准备好的对象时的整理顺序为当前，执行那个对象时的整理顺序也一样，即使该会话的（或 **DB_LOCALE** 的）执行时整理是不同的。

## 限制

在单个程序内准备好的对象的数目受可用内存的限制。这些包括在 PREPARE 语句（***statement_id*** 或 ***statement_id_var***）中声明的语句标识符，以及声明了的游标。要避免超限，请使用 FREE 语句来释放一些语句或游标。

在 SPL 例程中，准备好的对象可包括不超过一 SQL 语句的文本，那语句必须或为 EXECUTE FUNCTION、EXECUTE PROCEDURE，或为 SELECT 语句，但 SELECT 语句不可包括 INTO ***variable***、INTO TEMP 或 FOR UPDATE 子句。

在 SPL 例程中指定语句文本的表达式必须计算为 CHAR、LVARCHAR、NCHAR、NVARCHAR 或 VARCHAR 数据类型。您必须显式地强制转型为这些任何其他文本数据类型的表达式类型，诸如 UDT。

要了解在 ESQL/C 例程中对字符串中的 SQL 语句的限制，请参阅 在单一语句准备中受限的语句 和 在多语句准备好的对象中的受限语句。

## 声明语句标识符

PREPARE 将语句文本发送到数据库服务器，数据库服务器分析该语句文本。如果文本不含语法错误，则数据库服务器将其翻译为内部形式。为了以后执行，将此翻译了的语句保存在 PREPARE 语句分配的数据结构中。该结构的名称是在 PREPARE 语句中赋给该语句标识符的值。后续的 SQL 语句可通过使用与在 PREPARE 语句中使用的相同的标识符来引用该结构。

后续的 FREE 语句释放分配给了该语句的数据库服务器资源。在您以 FREE 释放这些资源之后，您不可在 DELCARE 语句或（在 ESQL/C 中）以 EXECUTE 语句使用该语句标识符，直到您再次准备该语句为止。

当例程退出时，自动地释放 SPL 例程为准备好的对象定义的数据库服务器资源。

# 语句标识符的作用域

ESQL/C 程序可由一个或多个源代码文件构成。缺省情况下，对程序而言，语句标识符的引用的作用域是全局的。因此，在一个文件中准备的语句标识符可从另一文件引用。

在多文件程序中，如果您想要将语句标识符的引用的作用域限制到其被准备的文件中， 则以 **-local** 命令行选项预先处理所有文件。

## 释放语句标识符

语句标识符一次仅可表示一个 SQL 语句或(在 ESQL/C 中)一个以分号分隔的 SQL 语句的列表。新的 PREPARE 语句可指定现有的语句标识符，如果您想要将该标识符绑定到不同的 SQL 语句文本的话。

PREPARE 语句支持动态的语句标识符名称，允许您来准备语句标识符作为标识符，或(在 ESQL/C 中)作为包含字符串的数据类型的主变量。下列的第一个示例显示被指定作为主变量的语句标识符。第二个示例指定语句标识符作为字符串。

```
stcopy ("query2", stmtid);
EXEC SQL prepare :stmtid from 'select * from customer';

EXEC SQL prepare query2 from 'select * from customer';
```

该变量必须为字符数据类型。在 C 中，它必须被声明为 **char**。

在 SPL 例程中，在本地作用域中自动地定义 PREPARE 语句声明的语句标识符。请不要尝试将语句标识符声明为有本地的或全局的作用域。对于同一会话调用的任何其他 SPL 例程，在一个 SPL 例程中定义的语句标识符都是不可见的。SPL 语句标识符与 SPL 变量和游标名称分享同一命名空间。

## 语句文本

可在 PREPARE 语句中指定语句文本

- 作为引用的字符串
- 或作为存储在 ESQL/C 程序变量中的文本
- 或（在 SPL 例程中）作为字符表达式。

下列限制适用于语句文本：

- 文本仅可包含 SQL 语句。它不可包含来自主编程语言的语句或注释。
- 文本可包含前面有双连字号（--），或括在大括号（{ }）中或括在 C 风格的斜杠和星号（/* */）定界符中的注释。

  这些符号引入或括起 SQL 注释。要获取更多关于 SQL 注释符号的信息，请参阅 如何输入 SQL 注释。

- 文本可包含或单一的 SQL 语句，或(在 ESQL/C 例程中）一系列由分号（;）分隔的语句。

  要了解不可准备的 SQL 语句的列表，请参阅 在单一语句准备中受限的语句。要了解更多关于如何准备多 SQL 语句的信息，请参阅 准备多个 SQL 语句。

- 文本不可包括嵌入的 SQL 语句前缀或结束符，诸如美元符号（$）或词语 EXEC SQL。

- 不像在准备好的文本中那样识别主语言变量。

  因此，您不可准备包括 INTO 子句的 SELECT（或 EXECUTE FUNCTION 或 EXECUTE PROCEDURE）语句，因为 INTO 子句需要主语言变量。

- 您仅可使用的标识符是在数据库中定义的那些名称，诸如表和列的名称。要获取更多关于如何在语句文本中使用标识符的信息，请参阅 以 SQL 标识符准备语句。
- 请使用问号（?）作为占位符来表示当语句执行时提供数据的位置，如在此 GBase 8s ESQL/C 示例中：

```
EXEC SQL prepare new_cust from
   'insert into customer(fname,lname) values(?,?)';
```

要获取更多关于如何使用问号作为占位符的信息，请参阅 准备接收参数的语句。

如果准备好的语句包含“集合派生的表”段或 GBase 8s ESQL/C 集合变量，则在您可为该 PREPARE 语句组装文本方面有一些附加的限制。要获取关于动态的 SQL 的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。SPL 例程不可使用动态的 SQL 语句来处理包含“集合派生的表”段的准备好的语句。

## 在 SPL 例程中的 PREPARE 语句的示例

GBase 8s SPL 语言支持单一语句的准备好的对象。

例如，下列 SQL 和 SPL 语句执行这些任务：

1. 创建 **cities** 表。
2. 以四行数据填入 **cities** 表。
3. 创建定义准备好的语句和游标来查询 **cities** 的 **order_city** SPL 例程：

```
CREATE TABLE cities       -- defines a table
(
  id INT,
  city_name CHAR(50)
);

INSERT INTO cities VALUES (1, 'Chicago');
INSERT INTO cities VALUES (2, 'New York');
INSERT INTO cities VALUES (3, 'San Francisco');
INSERT INTO cities VALUES (4, 'Atlanta');

UPDATE STATISTICS HIGH;

CREATE PROCEDURE order_city() -- defines a UDR
RETURNING INT, CHAR(50);
DEFINE c_num INT;
DEFINE c_name CHAR(50);
DEFINE c_query VARCHAR(250);
```

```
LET c_query =
 "SELECT id, city_name FROM cities ORDER BY city_name;";

PREPARE c_stmt FROM c_query;
DECLARE c_cur CURSOR FOR c_stmt;

OPEN c_cur ;
while (1 = 1)
   FETCH c_cur INTO c_num, c_name;
   IF (SQLCODE != 100) THEN
      RETURN c_num, c_name WITH RESUME;
   ELSE
      EXIT;
   END IF
END WHILE

CLOSE c_cur;
FREE c_cur;
FREE c_stmt;

END PROCEDURE;
```

下列 SQL 语句调用 **order_city** 例程：

```
EXECUTE PROCEDURE order_city();
```

如果从 **dbaccess** 实用程序调用 **order_city** 函数，则显示此输出：

```
(expression) (expression)
             4 Atlanta
             1 Chicago
             2 New York
             3 San Francisco
```

## 准备并执行用户定义的例程

准备用户定义的例程（UDR）的方式依赖于该 UDR 是用户定义的过程还是用户定义的函数：

- 要准备用户定义的过程，请准备执行该过程的 EXECUTE PROCEDURE 语句。
- 要执行用户准备的过程，请使用 EXECUTE 语句。
- 要准备用户定义的函数，请准备执行该函数的 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句。

您不可在 PREPARE 语句中包括 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）的 INTO 子句。

如何执行准备好的用户定义的函数，依赖于它是仅返回一组值还是多组值。对于仅返回一组值的用户定义的函数，请使用 EXECUTE 语句。

要执行返回多于一组返回值的用户定义的函数，您必须将该 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句与一游标相关联。

## 在单一语句准备中受限的语句

通常，您可准备任何数据操作语言（DML）语句。

在 GBase 8s 中，您可准备任何单一的 SQL 语句，除下列语句之外：

- ALLOCATE COLLECTION
- ALLOCATE DESCRIPTOR
- ALLOCATE ROW
- CLOSE
- CONNECT
- CREATE FUNCTION FROM
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DEALLOCATE COLLECTION
- DEALLOCATE DESCRIPTOR
- DEALLOCATE ROW
- DECLARE
- DESCRIBE
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- FLUSH
- FREE
- GET DESCRIPTOR
- GET DIAGNOSTICS
- INFO
- LOAD
- OPEN
- OUTPUT
- PREPARE
- PUT
- SET AUTOFREE
- SET CONNECTION
- SET DEFERRED_PREPARE
- SET DESCRIPTOR

- UNLOAD
- WHENEVER

您可准备 SELECT 语句。如果 SELECT 包括 INTO TEMP 子句，则 ESQL/C 程序可执行随同 EXECUTE 语句的准备好的语句。如果它不包括 INTO TEMP 子句，则该语句返回数据行。请使用 DECLARE、OPEN 和 FETCH 游标语句来存取行。

在 ESQL/C 中，准备好的 SELECT 语句可包括 FOR 子句。随同 DECLARE 语句使用此子句来创建一更新游标。下一示例展示在 GBase 8s ESQL/C 中带有 FOR UPDATE 子句的 SELECT 语句：

```
sprintf(up_query, "%s %s %s",
    "select * from customer ",
    "where customer_num between ? and ? ",
    "for update");
EXEC SQL prepare up_sel from :up_query;
EXEC SQL declare up_curs cursor for up_sel;
EXEC SQL open up_curs using :low_cust,:high_cust;
```

## 在参数已知时准备语句

在一些准备好的语句中，在准备语句时所有必需的信息都已知。下列在 GBase 8s ESQL/C 中的示例展示从常量数据准备了的两个语句：

```
sprintf(redo_st, "%s %s",
    "drop table workt1; ",
    "create table workt1 (wtk serial, wtv float)" );
EXEC SQL prepare redotab from :redo_st;
```

## 准备接收参数的语句

在一些语句中，在准备语句时参数还未知，因为每次执行该语句时可插入不同的值。在这些语句中，在当执行语句时必须提供参数的地方，您可使用问号（?）占位符。

下列 GBase 8s ESQL/C 示例中的 PREPARE 语句展示问号（?）占位符的一些使用：

```
EXEC SQL prepare s3 from
    'select * from customer where state matches ?';
EXEC SQL prepare in1 from 'insert into manufact values (?,?,?)';
sprintf(up_query, "%s %s",
    "update customer set zipcode = ?"
    "where current of zip_cursor");
EXEC SQL prepare update2 from :up_query;
EXEC SQL prepare exfunc from
    'execute function func1 (?, ?)';
```

对于表达式，但不对于 SQL 标识符，您可使用占位符来延缓计算值，直到运行时为止，除了在 以 SQL 标识符准备语句 中注明之外。

下列 GBase 8s ESQL/C 代码片段的示例从名为 **demoquery** 的变量准备语句。变量中的文本包括一问号（?）占位符。该准备好的语句与一游标相关联，且当打开该游标时，OPEN 语句的 USING 子句为该占位符提供值：

```
EXEC SQL BEGIN DECLARE SECTION;
    char queryvalue [6];
    char demoquery   [80];
EXEC SQL END DECLARE SECTION;

EXEC SQL connect to 'stores_demo';
sprintf(demoquery, "%s %s",
        "select fname, lname from customer ",
        "where lname > ? ");
EXEC SQL prepare quid from :demoquery;
EXEC SQL declare democursor cursor for quid;
stcopy("C", queryvalue);
EXEC SQL open democursor using :queryvalue;
```

在与游标以及 EXECUTE 语句（所有其他的准备好的语句）关联的两个 OPEN 语句中，USING 子句都可用。

您可使用问号（?）占位符来表示 GBase 8s ESQL/C 或 SPL 集合变量的名称。

## 以 SQL 标识符准备语句

通常，当准备语句时，您必须在语句文本中显式地指定 SQL 标识符。在少数特殊情况下，您可为 SQL 标识符使用问号（?）占位符：

- 对于 DATABASE 语句中的数据库名称。
- 对于 CREATE DATABASE 语句的 IN ***dbspace*** 子句中的 dbspace 名称。
- 对于使用游标名称的语句中的游标名称。

# 从用户输入获取 SQL 标识符

如果准备好的语句需要标识符，但在您编写准备好的语句时该标识符未知，则您可构造从用户输入接收 SQL 标识符的语句。

下列 GBase 8s ESQL/C 示例提示用户输入表的名称，并在 SELECT 语句中使用该名称。由于此名称在运行时之前未知，因此该表列的数目和数据类型也未知。因此，程序不可提前分配主变量来接收每一行的数据。取而代之的是，此程序片断描述语句到一 **sqlda** 描述符内并以该描述符访存每一行。该访存将每一行放入程序动态地提供的内存位置内。

如果程序检索活动集合中的所有行，则 FETCH 将置于被访存的每一行的循环中。如果 FETCH 语句检索超过一个数据值（列），则在 FETCH 之后存在另一循环，对每一数据值执行一些活动：

```
#include <stdio.h>
EXEC SQL include sqlda;
EXEC SQL include sqltypes;
char *malloc( );
```

```
main()
{
    struct sqlda *demodesc;
    char tablename[19];
    int i;
EXEC SQL BEGIN DECLARE SECTION;
    char demoselect[200];
EXEC SQL END DECLARE SECTION;

/*   该程序选择给定的 tablename 的所有列。
          交互地提供该 tablename。  */

EXEC SQL connect to 'stores_demo';
printf( "This program does a select * on a table\n" );
printf( "Enter table name: " );
scanf( "%s", tablename );
sprintf(demoselect, "select * from %s", tablename );

EXEC SQL prepare iid from :demoselect;
EXEC SQL describe iid into demodesc;

/* 打印描述返回的内容 */

for ( i = 0;   i < demodesc->sqld; i++ )
    prsqlda (demodesc->sqlvar + i);
/* 指定数据指针。 */

for ( i = 0;   i < demodesc->sqld; i++ )
    {
    switch (demodesc->sqlvar[i].sqltype & SQLTYPE)
        {
        case SQLCHAR:
            demodesc->sqlvar[i].sqltype = CCHARTYPE;
            /* 为空结束符制造空间 */
            demodesc->sqlvar[i].sqllen++;
            demodesc->sqlvar[i].sqldata =
                malloc( demodesc->sqlvar[i].sqllen );
            break;

        case SQLSMINT:      /* 失败 */
        case SQLINT:        /* 失败 */
        case SQLSERIAL:
            demodesc->sqlvar[i].sqltype = CINTTYPE;
            demodesc->sqlvar[i].sqldata =
                malloc( sizeof( int ) );
```

```
            break;
        /*   对每一类型亦然。    */
        }
    }

/* 为选择声明和打开游标。 */
EXEC SQL declare d_curs cursor for iid;
EXEC SQL open d_curs;

/* 每次将被选择的行访存到 demodesc 内。 */
for( ; ; )
    {
    printf( "\n" );
    EXEC SQL fetch d_curs using descriptor demodesc;
    if ( sqlca.sqlcode != 0 )
          break;
    for ( i = 0;   i < demodesc->sqld; i++ )
        {
        switch (demodesc->sqlvar[i].sqltype)
            {
            case CCHARTYPE:
                printf( "%s: \"%s\n", demodesc->sqlvar[i].sqlname,
                    demodesc->sqlvar[i].sqldata );
                break;
            case CINTTYPE:
                printf( "%s: %d\n", demodesc->sqlvar[i].sqlname,
                    *((int *) demodesc->sqlvar[i].sqldata) );
                break;
            /* 对每一类型亦然…… */
                }
        }
    }
EXEC SQL close d_curs;
EXEC SQL free d_curs;
/*   释放数据内存。    */

for ( i = 0;   i < demodesc->sqld; i++ )
    free( demodesc->sqlvar[i].sqldata );
free( demodesc );

printf ("Program Over.\n");
}

prsqlda(sp)
    struct sqlvar_struct *sp;
```

```
{
printf ("type = %d\n", sp->sqltype);
printf ("len = %d\n", sp->sqllen);
printf ("data = %lx\n", sp->sqldata);
printf ("ind = %lx\n", sp->sqlind);
printf ("name = %s\n", sp->sqlname);
}
```

## 准备多个 SQL 语句

在 ESQL/C 中，您可执行几个 SQL 语句作为一个活动，如果您将它们包括在同一 PREPARE 语句中。将多语句文本作为一个单元处理；不是顺序地处理这些活动。因此，多语句文本不可包括那些依赖于该文本中先前的语句中发生的活动的语句。例如，您不可创建表并将值插入到同一准备好的语句块中的那个表内。

如果多语句准备中的一个语句返回错误，则停止执行这个准备好的语句。数据库服务器不执行任何余下的语句。在多数情况下，编译的产品返回关于错误的错误状态信息，但不指出文本中的哪一语句导致错误。您可使用 **sqlca** 中的 **sqlca.sqlerrd[4]** 字段来发现错误的偏移量。

在多语句准备中，如果在下列语句中从 WHERE 子句未返回行，则数据库服务器返回 SQLNOTFOUND (100)：

- UPDATE … WHERE …
- SELECT INTO TEMP … WHERE …
- INSERT INTO … WHERE …
- DELETE FROM … WHERE …

在下一示例中，四个 SQL 语句被准备到称为 **query** 的单一 GBase 8s ESQL/C 字符串内。以分号分隔单独的语句。

单个 PREPARE 语句可准备该四个语句执行，且单个 EXECUTE 语句可执行与 qid 语句标识符相关联的语句：

```
sprintf (query,   "%s %s %s %s %s %s %s",
    "update account set balance = balance + ? ",
        "where acct_number = ?;",
    "update teller set balance = balance + ? ",
        "where teller_number = ?;",
    "update branch set balance = balance + ? ",
        "where branch_number = ?;",
    "insert into history values (?, ?);";
EXEC SQL prepare qid from :query;

EXEC SQL begin work;
EXEC SQL execute qid using
        :delta, :acct_number, :delta, :teller_number,
        :delta, :branch_number, :timestamp, :values;
EXEC SQL commit work;
```

此处需要分号（**;**）作为在 **query** 持有的文本中每一 SQL 语句之间的 SQL 语句结束符号。

## 在多语句准备好的对象中的受限语句

除了在 在单一语句准备中受限的语句 中罗列的作为例外的语句，您不可在多语句的准备好的对象的文本中使用下列语句：

- CLOSE DATABASE
- CREATE DATABASE
- DATABASE
- DROP DATABASE
- RENAME DATABASE
- SELECT（*随同一个例外*）

在多语句准备中，下列语句的类型也是无效的：

- 在执行多语句序列期间可导致当前数据库关闭的语句
- 包括对 TEXT 或 BYTE 主变量引用的语句

通常，您在多语句准备中不可使用 SELECT 语句。唯一在多语句准备中被允许的 SELECT 语句的形式为带有 INTO 临时表子句的 SELECT 语句。

## 为了效率而使用准备好的语句

要提高执行效率，您可在循环中使用 PREPARE 语句和 EXECUTE 语句，来消除冗余解析和优化所引起的开销。例如，每次循环运行时，都解析位于 WHILE 循环内的 UPDATE 语句。如果您在该循环之外准备 UPDATE 语句，则仅解析该语句一次，消除开销并提高了语句执行的速度。下列示例展示如何准备一 GBase 8s ESQL/C 语句来提高性能：

```
EXEC SQL BEGIN DECLARE SECTION;
    char disc_up[80];
    int cust_num;
EXEC SQL END DECLARE SECTION;
main()
{
    sprintf(disc_up, "%s %s","update customer ",
        "set discount = 0.1 where customer_num = ?");
    EXEC SQL prepare up1 from :disc_up;
    while (1)
        {
        printf("Enter customer number (or 0 to quit): ");
        scanf("%d", cust_num);
        if (cust_num == 0)
            break;
        EXEC SQL execute up1 using :cust_num;
        }
}
```

如同 SQL 语句高速缓存一样，准备好的语句可降低重新优化同一查询计划的频度，从而节约在一些上下文中的资源。准备好的语句和语句高速缓存 部分讨论综合地使用准备好的 DML 语句、游标和 SQL 语句高速缓存或提升查询性能的替代技术。

## 对在准备好的对象中引用的表的 DDL 操作

各种 DDL 语句可删除、重命名或更改准备好的对象引用的表的模式，但后续的执行该准备好的对象的尝试可能失败并报错误 -710，或可能导致预料不到的结果。

然而，当为直接地引用表的准备好的对象和例程启用自动的重新编译时，ALTER TABLE、CREATE INDEX 或 DROP INDEX 操作已更改了这些表，如果添加或删除索引，则这些限制不是必然适用的。这是 GBase 8s 的缺省行为。在更改表的模式之后，使用 SET ENVIRONMENT IFX_AUTO_REPREPARE 语句来启用或禁用自动的重新编译，且即使当启用自动的重新编译时，数据库服务器发出错误 -710 的地方的上下文。要获得更多关于这方面的信息，请参阅 IFX_AUTO_REPREPARE 环境选项。

然而，当 AUTO_REPREPARE 配置参数和 IFX_AUTO_REPREPARE 会话环境变量设置为禁用自动的重新编译时，将索引添加到准备好的语句间接地应用的表，可类似地导致准备好的语句无效。如果游标引用无效的准备好的语句，则即使 OPEN 语句包括 WITH REOPTIMIZATION 关键字，后续的 OPEN 语句也失败。在禁用自动的重新编译时，如果在间接地引用的表上添加索引，则您必须再次准备该语句并再次声明游标。如果游标是基于不再有效的准备好的语句，则您不可简单地重新打开游标。

**相关的语句**

相关的语句：CLOSE 子句、DECLARE 语句、DESCRIBE 语句、EXECUTE 语句、FREE 语句、OPEN 语句、SET AUTOFREE 语句 和 SET DEFERRED_PREPARE 语句

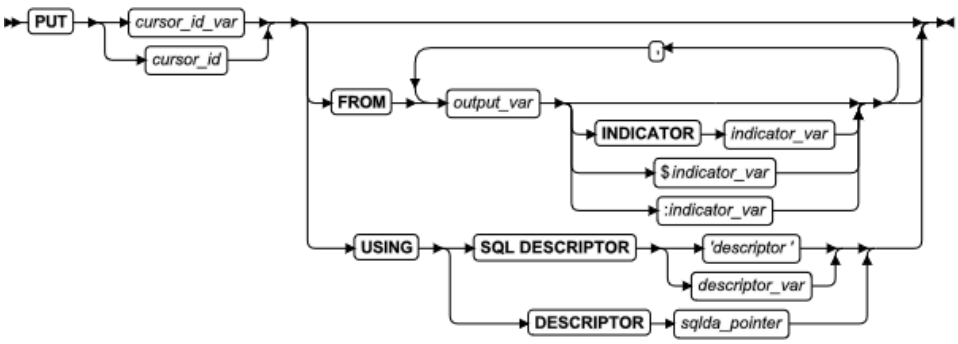
要获取关于与 PREPARE 语句相关的基本概念的信息，请参阅 *GBase 8s SQL 教程指南*。

要获取更多与 PREPARE 语句相关的高级概念的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

# 2.108 PUT 语句

为了以后插入到数据库内，使用 PUT 语句来在插入缓冲区中存储一行。

**语法**



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | 游标的名称 | 必须是打开的 | 标识符 |
| *cursor_id_var* | 主变量 = *cursor_id* | 必须为字符类型；游标必须是打开的 | 特定于语言 |
| *descriptor* | 系统描述符区域的名称 | 必须已分配 | 引用字符串 |
| *descriptor_var* | 包含 *descriptor* 的主变量 | 必须已分配 | 引用字符串 |
| *indicator_var* | 如果相应的 *output_var* 收到 NULL 值，则为要收到返回代码的主变量 | 不可为 DATETIME 或 INTERVAL 数据类型 | 特定于语言 |
| *output_var* | 其内容代替准备好的 INSERT 语句中的问号（?）占位符的主变量 | 必须为字符数据类型 | 特定于语言 |
| *sqlda_pointer* | 指向 sqlda 结构的指针 | 第一个字符不可为（$）或（:）符号 | DESCRIBE 语句 |

**用法**

此语句为对 SQL 的 ANSI/ISO 标准的扩展。您可随同 ESQL/C 使用此语句。

PUT 将行存储在打开游标时创建的***插入缓冲区***中。

如果当该语句执行时该缓冲区没有空间存储新行，则将被缓冲的行成块写到数据库，并清空缓冲区。因此，一些 PUT 语句执行导致将行写到数据库，而一些不写。您可使用 FLUSH 语句来将缓冲的行写到数据库，而不添加新行。在关闭 Insert 游标之前，CLOSE 语句写所有余下的行。

如果当前的数据库使用显式的事务，您必须在一事务之内执行 PUT 语句。

下列示例使用 GBase 8s ESQL/C 中的 PUT 语句：

```
EXEC SQL prepare ins_mcode from
    'insert into manufact values(?,?)';
EXEC SQL declare mcode cursor for ins_mcode;
EXEC SQL open mcode;
EXEC SQL put mcode from :the_code, :the_name;
```

PUT 语句不是 X/Open SQL 语句。因此，如果您在 X/Open 模式下编译 PUT 语句，则会得到警告信息。

## 提供插入的值

插入的行中的值可来自下列来源之一：

- 写到 INSERT 语句内的常量值
- 在 INSERT 语句中命名的程序变量
- 在 PUT 语句的 FROM 子句中的程序变量
- 在由 **sqlda** 结构寻址的内存中准备的值，或系统描述符区域，然后在 PUT 语句的 USING 子句中指定。

***descriptor*** 或 ***sqlda_pointer*** 引用的系统描述符区域或 **sqlda** 结构必须定义每一值的数据类型和内存位置，对应于准备好的 INSERT 语句中的问号（?）占位符。

# 在 INSERT 中使用常量值

VALUES 子句罗列插入的列的值。这些值中的一个或多个可为常量（即，数字或字符串）。

当*所有*插入的值都是常量时，PUT 语句有一特殊的作用。PUT 语句仅增大计数器，而不创建行并放入缓冲区中。当您使用 FLUSH 或 CLOSE 语句来清空缓冲区时，将一行和重复计数发送到数据库服务器，插入那个编号的行。在下列 GBase 8s ESQL/C 示例中，将 99 个空客户记录插入到 customer 表内。因为所有值都是常量，直到该游标关闭才会发生磁盘输出。（customer_num 的常量零导致生成 SERIAL 值。）下列示例将 99 个空客户记录插入到客户表内：

```
int count;
EXEC SQL declare fill_c cursor for
    insert into customer(customer_num) values(0);
EXEC SQL open fill_c;
for (count = 1; count <= 99; ++count)
    EXEC SQL put fill_c;
EXEC SQL close fill_c;
```

# 在 INSERT 中命名程序变量

当您将 INSERT 语句与游标（在 DECLARE 语句中）相关联时，请创建 Insert 游标。在 INSERT 语句中，您可在 VALUES 子句中命名程序变量。当执行每一 PUT 语句时，使用那个时刻的程序变量的内容来填写插入到缓冲区内的行。

如果您正在（随同 INSERT 使用 DECLARE）创建 Insert 游标，则必须在该 VALUES 子句中仅使用程序变量。在准备好的语句的上下文中不识别变量名称；您通过其语句标识符将准备好的语句与游标关联。

下列 GBase 8s ESQL/C 示例展示 Insert 游标的使用。代码包括下列语句：

- DECLARE 语句将名为 ins_curs 的游标与 INSERT 语句相关联，该 INSERT 语句将数据插入到 customer 表内。
- VALUES 子句指定名为 **cust_rec** 的数据结构；GBase 8s ESQL/C 预处理器将 **cust_rec** 转化为值的列表，每一结构的组件一个。
- OPEN 语句创建一缓冲区。

- 用户定义的函数(未在此示例内定义)从用户输入获取客户信息并将其保存在 **cust_rec** 中。
- PUT 语句从 **cust_rec** 结构的当前内容组成一行，并将其发送到行缓冲区。
- CLOSE 语句将留在行缓冲区中的任何行都插入到 **customer** 表内，并关闭 Insert 游标：

```
int keep_going = 1;
EXEC SQL BEGIN DECLARE SECTION
    struct cust_row { /* fields of a row of customer table */ } cust_rec;
EXEC SQL END DECLARE SECTION
EXEC SQL declare ins_curs cursor for
        insert into customer values (:cust_row);
EXEC SQL open ins_curs;
while ( (sqlca.sqlcode == 0) && (keep_going) )


  {
keep_going = get_user_input(cust_rec); /* ask user for new customer */
    if (keep_going )                          /* user did supply customer info
*/
        {
        cust_rec.customer_num = 0;              /* request new serial value */
        EXEC SQL put ins_curs;
        }
    if (sqlca.sqlcode == 0)                   /* no error from PUT */
        keep_going = (prompt_for_y_or_n("another new customer") =='Y')
    }
EXEC SQL close ins_curs;
```

如果插入的数据可能为 NULL ，则请使用指示符变量。

当准备 INSERT 语句时(请参阅 PREPARE 语句)，您不可在它的 VALUES 子句中使用程序变量，但可通过问号(?)占位符表示值。请在 PUT 语句的 FROM 子句中罗列程序变量来提供缺少的值。

下列 GBase 8s ESQL/C 示例罗列在 PUT 语句中的主变量：

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
    char ins_comp[80];
    char u_company[20];
EXEC SQL END DECLARE SECTION;

main()
{
    EXEC SQL connect to 'stores_demo';
    EXEC SQL prepare ins_comp from
        'insert into customer (customer_num, company) values (0, ?)';
    EXEC SQL declare ins_curs cursor for ins_comp;
    EXEC SQL open ins_curs;
```

```
    while (1)
        {
        printf("\nEnter a customer: ");
         gets(u_company);
        EXEC SQL put ins_curs from :u_company;
        printf("Enter another customer (y/n) ? ");
        if (answer = getch() != 'y')
            break;
        }
    EXEC SQL close ins_curs;
    EXEC SQL disconnect all;
}
```

指示符变量是可选的，但如果可能存在包含 NULL 值的 ***output_var*** ，则您应使用指示符变量。如果您指定未带 INDICATOR 关键字的指示符变量，则不可在 ***output_var*** 与 ***indicator_var*** 之间放空格。

## 命名 PUT 中的程序变量

当准备 INSERT 语句时（请参阅 PREPARE 语句），您不可在它的 VALUES 子句中使用程序变量，但可通过问号（?）占位符表示值。请在 PUT 语句的 FROM 子句中罗列程序变量来提供缺少的值。

下列 GBase 8s ESQL/C 示例罗列在 PUT 语句中的主变量：

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
    char ins_comp[80];
    char u_company[20];
EXEC SQL END DECLARE SECTION;

main()
{
    EXEC SQL connect to 'stores_demo';
    EXEC SQL prepare ins_comp from
        'insert into customer (customer_num, company) values (0, ?)';
    EXEC SQL declare ins_curs cursor for ins_comp;
    EXEC SQL open ins_curs;

    while (1)
        {
        printf("\nEnter a customer: ");
         gets(u_company);
        EXEC SQL put ins_curs from :u_company;
        printf("Enter another customer (y/n) ? ");
        if (answer = getch() != 'y')
            break;
        }
```

```
    EXEC SQL close ins_curs;
    EXEC SQL disconnect all;
}
```

指示符变量是可选的，但如果可能存在包含 NULL 值的 ***output_var*** ，则您应使用指示符变量。如果您指定未带 INDICATOR 关键字的指示符变量，则不可在 ***output_var*** 与 ***indicator_var*** 之间放空格。

## 使用 USING 子句

如果您不知道在运行时要提供的参数的数目或其数据类型，则可从系统描述符或 **sqlda** 结构关联输入值。这些描述符结构都描述一个或多个替代问号（?）占位符的值的数据类型和内存位置。

每次执行 PUT 语句，都使用描述符结构描述的值替代 INSERT 语句中的问号（?）占位符。此过程类似于随同变量列表使用 FROM 子句，除了您的程序可完全控制数据值的内存位置之外。

## 指定系统描述符区域

SQL DESCRIPTOR 选项指定系统描述符区域的名称。

系统描述符区域中的 **COUNT** 域对应于准备好的语句中动态参数的数目。**COUNT** 的值必须小于或等于当以 ALLOCATE DESCRIPTOR 分配系统描述符区域时指定的项描述符的数量。您可以 GET DESCRIPTOR 语句取得域值，并以 SET DESCRIPTOR 语句设置该值。

系统描述符区域符合 X/Open 标准。

下列 GBase 8s ESQL/C 示例展示如何从系统描述符区域关联值：

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL put selcurs using sql descriptor 'desc1';
```

## 指定 sqlda 结构

使用 DESCRIPTOR 选项来引入指向 sqlda 结构的指针的名称。下列 GBase 8s ESQL/C 示例展示如何从 sqlda 结构关联值：

```
EXEC SQL put selcurs using descriptor pointer2;
```

## 插入到 Collection 游标内

Collection 游标允许您访问集合变量的单个元素。 要声明 Collection 游标，请使用 DECLARE 语句并包括在您将其与游标关联的 INSERT 语句中的“集合派生的表”段。一旦您以 OPEN 语句打开 Collection 游标，该游标可将元素放在集合变量中。

要一次一个地将元素放到 Insert 游标内，请使用 PUT 语句和 FROM 子句。 PUT 语句指定与该集合变量相关联的 Collection 游标。 FROM 子句标识要被插入到游标内的元素值。 在 FROM 子句中的任何主变量的数据类型必须与该集合的元素类型相匹配。

Important： 集合变量存储集合的元素。然而，它与数据库列没有内在的联系。一旦集合变量包含正确的元素，那么您必须以 INSERT 或 UPDATE 语句将该变量保存到实际的集合列内。

假设您有一名为 children 的表，其模式如下：

```
CREATE TABLE children
(
    age         SMALLINT,
    name        VARCHAR(30),
    fav_colors SET(VARCHAR(20) NOT NULL)
);
```

下列 GBase 8s ESQL/C 程序片断展示如何使用 Insert 游标将元素放入名为 child_colors 的集合变量内：

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection child_colors;
    char *favorites[]
    (
        "blue",
        "purple",
        "green",
        "white",
        "gold",
        0
    );
    int a = 0;
    char child_name[21];
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :child_colors;

/* 获取 fav_colors 列的结构，对于 untyped
 * child_colors 集合变量 */
EXEC SQL select fav_colors into :child_colors
    from children where name = :child_name;
/* 为 child_colors 集合变量声明插入游标
 * 并打开此游标 */
EXEC SQL declare colors_curs cursor for
    insert into table(:child_colors)
    values (?);
EXEC SQL open colors_curs;
/* 使用 PUT 来将 favorite-color 值收集
 * 到游标内 */
while (fav_colors[a])
{
    EXEC SQL put colors_curs from :favorites[:a];
```

```
    a++
    ...
}
/* 刷新游标内容到集合变量 */
EXEC SQL flush colors_curs;
EXEC SQL update children set fav_colors = :child_colors;

EXEC SQL close colors_curs;
EXEC SQL deallocate collection :child_colors;
```

在 FLUSH 语句执行之后，集合变量 **child_colors** 包含元素 {"blue", "purple", "green", "white", "gold"}。在此程序片断的末尾的 UPDATE 语句将新的集合保存到数据库的 **fav_colors** 列内。没有此 UPDATE 语句，就不会将新的集合插入到集合列。

## 写缓冲了的行

要打开 Insert 游标，OPEN 语句创建插入缓冲区。PUT 语句将一行放到此插入缓冲区内。仅当必要时，才将缓冲了的行成块插入到数据库表内；此过程称为***刷新缓冲区***。在下列任何事件之后，都会刷新缓冲区：

- 缓冲区太满，以至于不能在 PUT 语句开始时保持新行。
- FLUSH 语句执行。
- CLOSE 语句关闭游标。
- OPEN 语句指定已打开的游标，在重新打开它之前关闭它。（此隐式 CLOSE 语句刷新缓冲区。）
- COMMIT WORK 语句执行。
- 缓冲区包含 BYTE 或 TEXT 语句（在单个 PUT 语句之后刷新）。

如果程序没有关闭 Insert 游标就终止，则缓冲区保持未刷新。自从上一次刷新以来插入到该缓冲区内的行丢失。请不要依赖于程序的末尾来关闭游标并刷新缓冲区。

## 错误检查

sqlca 结构包含每一 PUT 语句的成功信息，以及使您能对插入了的行进行计数的信息。在 **sqlca** 的下列字段中包含每一 PUT 语句的结果：**sqlca.sqlcode**、**SQLCODE** 和 **sqlca.sqlerrd[2]**。

带有 Insert 游标的数据缓冲区意味着直到刷新缓冲区时才发现错误。例如，仅当刷新缓冲区时，才会发现输入值与所想要的列的数据类型不兼容。当发现错误时，***不***插入那些在错误之前未被插入的那些缓冲了的行；它们会从内存丢失。

如果未发生错误，则 SQLCODE 域设置为 0；否则设置为错误代码。sqlerrd 数组的第三个元素设置为成功地插入到了数据库内的行的数目：

- 如果将任何行放到插入缓冲区内，但***未***写到数据库，则 **SQLCODE** 和 **sqlerrd** 设置为 0（**SQLCODE** 是因为未发生错误，**sqlerrd** 是因为未插入行）。
- 如果在 PUT 语句执行期间，缓冲了的行块写到数据库，则 **SQLCODE** 设置为 0，且 **sqlerrd** 设置为成功地插入到数据库内的行的数目。

- 如果在将缓冲了的行写到数据库时发生错误，则 **SQLCODE** 指出错误，且 **sqlerrd** 包含成功地插入行的数目。（从缓冲区废弃未缓冲的行。）

**提示：** 当您遇到 SQLCODE 错误时，还存在 SQLSTATE 错误。请参阅 GET DIAGNOSTICS 语句了解如何获取消息文本的细节。

**要对在数据库中挂起和插入的行的数目计数**

1. 准备两个整数变量（例如，**total** 和 **pending**）。
2. 当打开游标时，将两个变量设置为 0。
3. 每次执行 PUT 语句，增大 **total** 和 **pending**。
4. 无论何时执行 PUT 或 FLUSH 语句，或关闭游标，从 **pending** 减去 **SQLERRD** 数组的第三个字段。

在任何时候，（**total - pending**）都表示实际插入的行数。 如果没有语句失败，则在关闭游标之后 **pending** 包含零。 如果在 PUT、FLUSH 或 CLOSE 期间发生错误，则保留在 **pending** 中的值为未插入的（被废弃的）行的数目。

相关的语句

相关的语句：ALLOCATE DESCRIPTOR 语句、CLOSE 子句、DEALLOCATE DESCRIPTOR 语句、FLUSH 语句、DECLARE 语句、GET DESCRIPTOR 语句、OPEN 语句、PREPARE 语句 和 SET DESCRIPTOR 语句

要获得关于 PUT 语句的面向任务的讨论，请参阅 *GBase 8s SQL 教程指南*。

要获得更多关于错误检查、系统描述符区域和 **sqlda** 结构的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

## 2.109 RELEASE SAVEPOINT 语句

使用 RELEASE SAVEPOINT 语句来销毁指定的保存点。RELEASE SAVEPOINT 语句符合 SQL 的 ANSI/ISO 标准。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *savepoint* | 要销毁的保存点的名称 | 必须在当前的保存点级别中存在 | 标识符 |

**用法**

**限制：** 在此语句成功地执行之后，不再可能回滚到指定的保存点（或到 RELEASE SAVEPOINT 语句与指定的保存点之间的任何其他保存点）。

RELEASE SAVEPOINT 语句销毁指定的保存点。还销毁在当前的保存点级别中那个保存点与 RELEASE SAVEPOINT 语句之间的任何保存点设置。然而，在当前的保存点级别中设置早于指定的保存点的保存点继续为活动的。

在下列上下文中，RELEASE SAVEPOINT 语句失败并报错：

- 无 SQL 事务是打开的。
- 在当前的保存点级别中，不存在带有指定的名称的保存点。
- 该语句为触发器的活动的一部分。
- 该语句为 XA 事务的一部分。
- 启用客户端 API 的 autocommit 事务模式。
- 该语句为跨服务器的分布式 SQL 事务的一部分，在该事务中，参与的数据库服务器之一不支持保存点。
- 在 DML 语句内调用的 UDR 内发出该语句。

RELEASE SAVEPOINT 销毁的任何保存点的标识符可在同一保存点级别的后续的 SAVEPOINT 语句中重用，即使通过包括 UNIQUE 关键字的 SAVEPOINT 语句设置了该释放的保存点。

由于保存点为程序对象，而不是数据库对象，因此 RELEASE SAVEPOINT 语句对数据库或其系统目录表没有直接的影响。然而，RELEASE SAVEPOINT 可间接地影响用户表和系统目录，如果它更改后续的 ROLLBACK TO SAVEPOINT 操作的作用域，该操作在当前保存点级别的不同部分内，取消对数据库的未提交的更改，如下例所示。

下列程序片断设置名为 sp45 的保存点，然后释放它：

```
BEGIN WORK;
CREATE DATABASE third_base IN db3 WITH BUFFERED LOG;
SAVEPOINT sp46;
CREATE TABLE tab1 ( col1 INT, col2 CHAR(24));
SAVEPOINT sp45 UNIQUE;
...
CREATE TABLE tab2 ( col1 INT8, col2 LVARCHAR(24000));
SAVEPOINT sp44;
...
RELEASE SAVEPOINT sp45;
ROLLBACK TO SAVEPOINT;
```

在此示例中 RELEASE SAVEPOINT 语句的作用是销毁两个保存点，sp45 和 sp44。如果仅保留当前的保存点级别中的保存点为 sp46，则后续的 ROLLBACK TO SAVEPOINT 语句取消那些创建了 tab1 和 tab2 的 DDL 语句，并取消对那些在 ROLLBACK 语句之前的表的任何 DML 操作。然而，回滚不取消创建 third_base 数据库的 CREATE DATABASE 语句。没有 RELEASE SAVEPOINT 语句，创建了 tab1 的 CREATE TABLE 语句可能已被取消，因为 GBase 8s 可能已经将 sp44 处理作为 ROLLBACK 语句的 TO SAVEPOINT 子句的缺省的保存点。

## 2.110 RENAME COLUMN 语句

使用 RENAME COLUMN 语句来更改列的名称。RENAME COLUMN 语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_column* | 您在此声明来替代 *old_column* 的名称 | 在 *table* 中的列名称中必须是唯一的。另请参阅 影响触发器的方式。 | 标识符 |
| *old_column* | 要重命名的列 | 在表内必须存在 | 标识符 |
| *owner* | 表的所有者 | 必须为表的所有者 | 所有者名称 |
| *table* | 包含 *old_column* 的表 | 必须注册在当前数据库中 | 标识符 |

**用法**

如果任何下列条件为真，则您可重命名表的列：

- 您拥有该表或有对该表的 Alter 权限。
- 您有对该数据库的 DBA 权限。

该列可在 CREATE EXTERNAL TABLE 语句定义的表对象中。

**示例**

下列示例将新名称 c_num 赋予 customer 表中的 customer_num 列：

```
RENAME COLUMN customer.customer_num TO c_num;
```

## 影响视图和检查约束的方式

如果您重命名出现在视图中的列，则更新在 **sysviews** 系统目录表中的视图定义的文本，来反映新的列名称。如果您重命名出现在检查约束中的列，则更新在 **syschecks** 系统目录表中的检查约束的文本，来反映新的列名称。

## 影响触发器的方式

如果您重名出现在触发器定义内的列，则仅在下列情况下才用新的名称代替它：

- 当它作为触发器的 FOR EACH ROW 活动子句内的相关名称的一部分出现时
- 当它作为 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句的 INTO 子句中的相关名称的一部分出现时

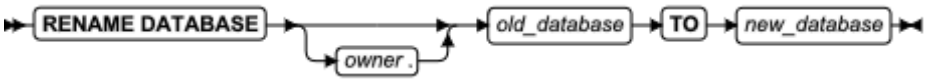
- 当它作为 UPDATE 子句中的触发器列出现时

当触发器执行时，如果数据库服务器遇到在该表中不再存在的列名称时，在返回错误。

## 2.111 RENAME DATABASE 语句

使用 RENAME DATABASE 语句来更改数据库的名称。此语句为对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_database* | 您在此为 *old_database* 声明的新名称 | 必须在当前数据库服务器的数据库名称之中为唯一的；当发出此语句时，必须未被任何用户打开 | 数据库名 |
| *old_database* | *new_database* 替代的名称 | 必须在当前数据库服务器上存在，但它不可为当前数据库的名称 | 数据库名 |
| *owner* | *old_database* 的所有者 | 必须为该数据库的所有者 | 所有者名称 |

**用法**

如果下列条件之一为真，则您可重命名数据库：

- 您创建了该数据库。
- 您有对该数据库的 DBA 权限。

然而，如果指定的数据库包含任何下列对象，则 RENAME DATABASE 语句失败，并报错 -9874：

- 虚拟表
- 虚拟索引
- R-tree 索引
- 在用户定义的主访问方法中或在用户定义的辅助访问方法中，引用数据库的当前名称的 DataBlade。

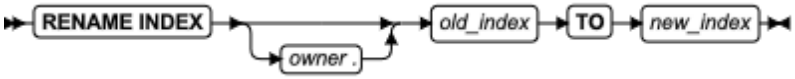
您仅可重命名您当前连接到的数据库服务器的数据库。

您不可从 SPL 例程之内重命名数据库。

## 2.112 RENAME INDEX 语句

使用 RENAME INDEX 语句来更改现有的索引的名称。此语句为对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_index* | 您在此为该索引声明的新名称 | 名称对于该数据库必须为唯一的（或如果 *old_index* 在临时表上，则是对于该会话） | 标识符 |
| *old_index* | *new_index* 替代的索引名称 | 必须存在，但不可为下列中的任何之一： -- 在系统目录表上的索引 -- 系统生成的约束索引 -- “虚拟索引接口”（VII） | 标识符 |
| *owner* | 索引的所有者 | 必须为 *old_index* 的所有者 | 所有者名称 |

**用法**

如果您是该索引的所有者或有对该数据库的 DBA 权限，则可重命名索引。

当您重命名索引时，数据库服务器更改 **sysindexes**、**sysconstraints**、**sysobjstate** 和 **sysfragments** 系统目录表中的索引名称。（但对于临时表上的索引，不更新系统目录表。）

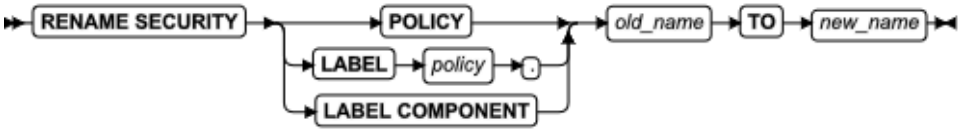
不可重命名系统目录表上的索引。如果您想更改实施约束的系统生成的索引的名称，请使用 ALTER TABLE ... DROP CONSTRAINT 语句来删除该约束，然后再使用 ALTER TABLE ... ADD CONSTRAINT 语句来定义新的约束，新约束与您删除了的约束有相同的定义，但对于您声明的新名称。

在缺省情况下，重新优化使用该重命名了的索引的 SPL 例程，当重命名该索引之后下一次执行它们时。然而，当启用自动的重编译时，如果该重命名了的索引与直接地引用了的表相关联，则在下一次使用该重命名了的索引时，自动地重编译 SPL 例程。然而，如果仅间接地引用该表，执行可失败，并报错 -710。要获取更多关于在更改被引用的表的模式之后，启用或禁用自动的重编译的信息，请参阅 IFX_AUTO_REPREPARE 环境选项。要获取更多关于 AUTO_REPREPARE 配置参数的信息，请参阅您的 *GBase 8s 管理员参考手册*。

## 2.113 RENAME SECURITY 语句

使用 RENAME SECURITY 语句来更改现有的安全对象的名称。该对象可为安全策略，或安全标签，或安全标签组件。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_name* | 您在此为该安全对象声明的新名称 | 在该数据库中的安全对象的标识符之中必须为唯一的，且必须不同于 *old_name* | 标识符 |
| *old_name* | *new_name* 替代的当前名称 | 在该数据库中必须作为安全对象的标识符存在 | 标识符 |
| *policy* | *old_name* 标签的安全策略 | 必须为安全标签 *old_name* 的安全策略 | 标识符 |

**用法**

此语句为对 SQL 的 ANSI/ISO 标准的扩展。

仅 DBSECADM 可发出此语句。在该重命名了的安全对象注册在其中的系统目录的表中，RENAME SECURITY 语句以指定的 *new_name* 替代 *old_name*：

- **sysecpolicies.secpolicyname** 对于安全策略
- **sysseclabels.seclabelname** 对于安全标签
- **sysseclabelcomponents.compname** 对于安全标签组件。

然而，此语句不更改该重命名了的安全对象的 sysecpolicies.secpolicyid、sysseclabels.seclabelid 或 sysseclabelcomponents.compid 的数值。

该关键字或跟在 SECURITY 关键字之后的关键字表示正在重命名的安全对象的类型。在下例中，新的标识符 honesty 作为安全策略的名称替代 best：

```
RENAME SECURITY POLICY best TO honesty;
```

在下例中，新的标识符 transparent 作为 honesty 安全策略的标签的名称替代 opaque：

```
RENAME SECURITY LABEL honesty.opaque TO transparent;
```

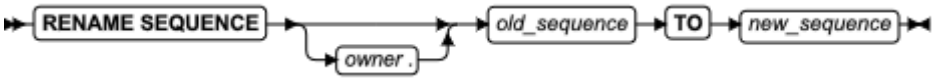
在下一个示例中，新的标识符 accountant 作为安全标签组件的名称替代 architect：

```
RENAME SECURITY LABEL COMPONENT architect TO accountant;
```

## 2.114 RENAME SEQUENCE 语句

使用 RENAME SEQUENCE 语句来更改序列的名称。此语句为对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_sequence* | 您在此为现有的序列声明的新名称 | 必须为该数据库中序列、表、视图和同义词的名称之中唯一的 | 标识符 |
| *old_sequence* | 序列的当前名称 | 在当前的数据库中必须存在 | 标识符 |
| *owner* | 序列的所有者 | 必须为该序列的所有者 | 所有者名称 |

**用法**

要重命名序列，您必须为该序列的所有者，对该序列有 ALTER 权限，或对该数据库有 DBA 权限。

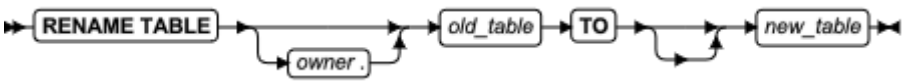
您不可使用同义词来指定该序列的名称。

在不符合 ANSI 的数据库中，*new_sequence* 的名称（或在符合 ANSI 的数据库中，*owner.new_sequence* 的组合）必须在该数据库中的序列、表、视图和同义词之中为唯一的。

## 2.115 RENAME TABLE 语句

使用 RENAME TABLE 语句来更改表的名称。RENAME TABLE 语句为对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_table* | *old_table* 的新名称 | 在该数据库的序列、表、视图和同义词的名称之中必须为唯一的 | 标识符 |
| *old_table* | *new_table* 替代的名称 | 必须为在当前的数据库中注册的表的名称（非同义词） | 标识符 |
| *owner* | 该表的当前所有者 | 必须为该表的所有者。 | 所有者名称 |

**用法**

要重命名表，您必须为该表的所有者，或有对该表的 ALTER 权限，或有对该数据库的 DBA 权限。

如果 *old_table* 为同义词，而不是表的名称，则发生错误。

***old_table*** 可为 CREATE EXTERNAL TABLE 语句定义的对象。

重命名了的表保留在当前的数据库中。您不可使用 RENAME TABLE 语句来将表从当前的数据库移到另一数据库，也不能重命名位于另一数据库中的表。

您不可通过重命名表来更改表 *owner*。如果您试图为该表的新名称指定 *owner*. 标识符，则发生错误。

当更改表所有者时，您必须同时指定旧的所有者和新的所有者。

在符合 ANSI 的数据库中，如果您不是 *old_table* 的所有者，则必须指定 *owner.old_table* 作为该表的旧的名称。

如果通过当前数据库中的视图引用 *old_table*，则在 **sysviews** 系统目录表中更新该视图定义来反映新的表名称。要获取更多关于 **sysviews** 系统目录表的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

如果 *old_table* 为触发器表，则数据库服务器采取这些活动：

- 替代在触发器定义中的表的名称，但不在任何触发器活动之中表名称出现的任何地方替代它
- 如果新的表名称与该触发器定义的 REFERENCING 子句中的相关名称相同，则返回错误

当执行该触发器时，如果它遇到不存在的表的表名称，则数据库服务器返回错误。

**使用 RENAME TABLE 来重新组织表**

当您需要重新组织现有的表的模式时，RENAME TABLE 语句可为 ALTER TABLE 语句的有用的替代手段。例如，假如您决定更改 stores 演示数据库的 items 表中列的顺序。您可通过下列这些步骤重新组织 items 表，将 quantity 列从第五个位置移到第三个位置：

1. 创建新表 **new_table**，在第三个位置包含列 **quantity**。
2. 以来自当前的 **items** 表的数据填充该表。
3. 删除旧的 **items** 表。
4. 以标识符 **items** 重命名 **new_table**。

下列示例使用 RENAME TABLE 语句作为最后的步骤：

```
CREATE TABLE new_table
    (
    item_num            SMALLINT,
    order_num           INTEGER,
    quantity           SMALLINT,
    stock_num           SMALLINT,
    manu_code            CHAR(3),
    total_price        MONEY(8)
    );
INSERT INTO new_table
    SELECT item_num, order_num, quantity, stock_num,
          manu_code, total_price FROM items;
```

```
DROP TABLE items;
RENAME TABLE new_table TO items;
```

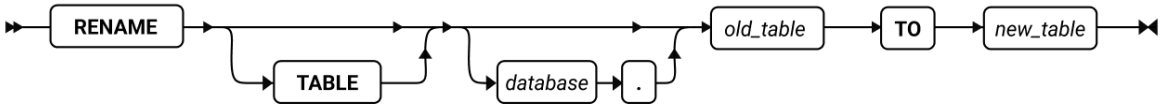
## RENAME 语句重命名表

新增该功能为兼容 Oracle 的 SQL 语法需要。

该功能仅在 GBase 8s 的 ORACLE 模式下支持。

使用 RENAME 语句来更改表的名称。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *old_table* | 需要重命名的表 | 必须为在当前的数据库中注册的表的名称（非同义词） | 标识符 |
| *new_table* | 在此声明来代替 *old_table* 的名称 | 在该数据库的序列、表、视图和同义词的名称之中必须唯一 | 标识符 |
| *database* | 库名 | 需重命名的表必须在库中 | 表所在库的名称 |

**用法**

重命名一个表，必须具有 ALTER 该表的权限；

新表名在该数据库的表的名称之中必须为唯一的；

旧表名必须为在当前的数据库中注册的表的名称（非同义词）；

重命名了的表保留在当前的数据库中。不可使用 RENAME 语句来将表从当前的数据库移到另一数据库，也不能重命名位于另一数据库中的表。

- 索引、视图、同义词、约束(包括外键)、权限在表重命名后，会跟随着变更，无需单独处理。触发器、存储过程的逻辑文本部分在表重命名后会失效，需要单独处理。

示例：

例如：当前库下的表 employee 重命名为 employee_test：

```
rename employee to employee_test;
```

例如：test 库下的表 employee 重命名为 employee_test：

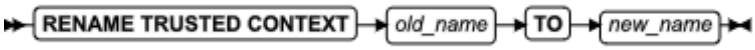
```
rename test.employee to employee_test;
```

## 2.116 RENAME TRUSTED CONTEXT 语句

使用 RENAME TRUSTED CONTEXT 语句来更改可信的上下文对象的名称。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。您必须持有数据库安全管理员（DBSECADM）角色来重命名可信的上下文。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *old_name* | *new_name* 替代的可信的上下文标识符 | 必须为数据库服务器的现有的可信的上下文对象 | 标识符 |
| *new_name* | 您在此为可信的上下文声明的新名称 | 必须为一部分名称，无限定符。它不可以字符 "SYS" 开头，且必须不标识在数据库服务器上业已存在的可信的上下文。 | 标识符 |

**用法**

***new_name*** 和 ***old_name*** 不可包括限定符，诸如 ***owner***、***database*** 或 ***dbserver***。

成功地执行 RENAME TRUSTED CONTEXT 语句之后，所有对 *old_name* 的引用都会被 GBase 8s 数据库服务器实例的 sysuser 数据库中的这些表中的 *new_name* 所替代：

- systrustedcontext
- systcxattributes
- systcxusers.

此外，那些尝试通过引用该 ***old_name*** 来建立到该数据库的连接的应用程序将会失败，除非已经声明该 ***old_name*** 作为新的可信的上下文对象的标识符。

如果在此上下文的可信的连接为活动的时候，您重命名该可信的上下文，则那些连接保持为可信的，直到它们终止为止，或直到下一重用尝试为止。然而，如果尝试着切换在这些可信的连接上的用户，则返回错误。

下例是一个完整的 RENAME TRUSTED CONTEXT 语句，以 cntx2 作为 cntx1 可信的上下文的新名称替代安全对象标识符 cntx1：

```
RENAME TRUSTED CONTEXT cntx1 TO cntx2;
```

在下列任一情况下，此示例失败：

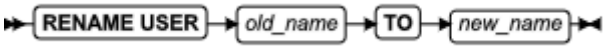
- 如果 cntx1 不是当前数据库服务器实例的可信的上下文对象的名称，
- 或如果 cntx2 已是同一数据库服务器的现有的可信的上下文对象的名称。

## 2.117 RENAME USER 语句 (UNIX™、Linux™)

使用 RENAME USER 语句来更改数据库服务器的 non-root 安装的内部用户的名称。

此语句为对 SQL 语言的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *old_name* | 您正在重命名的特定用户的授权标识符。 | 必须为现有的授权标识符 | 所有者名称 |
| *new_name* | 特定用户的授权标识符。 | 不可为现有的授权标识符 | 所有者名称 |

**用法**

仅 DBSA 可运行 RENAME USER 语句。随同 non-root 安装，安装该服务器的用户等同于 DBSA，除非该用户将 DBSA 权限授予不同的用户。

在连接上的用户为活动的时，请不要重命名用户。运行该语句并不将授权给旧的用户名的任何数据库级或表级权限转给新的用户名。

可以 RNUR 审计代码审计 RENAME USER 语句的执行。

USERMAPPING 配置参数必须设置为 BASIC 或 ADMIN。

您还必须在 **sysusers** 数据库的 SYSUSERMAP 表中输入值，来以适当的用户属性映射用户，以便映射了的 SQL 的用户语句正确地工作。

**示例**

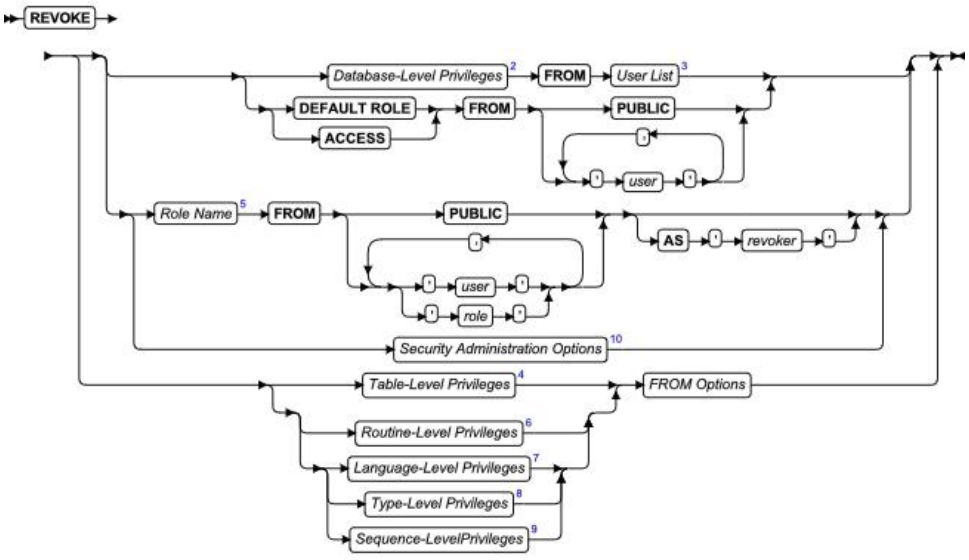
下列语句将用户 **bill** 重命名为 **bob**：

```
RENAME USER bill TO bob;
```

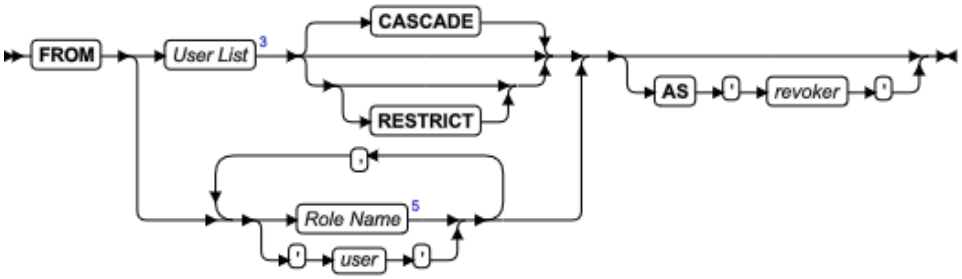
## 2.118 REVOKE 语句

使用 REVOKE 语句来取消由用户、由角色或由 PUBLIC 持有的访问权限或角色，或从安全策略的规则取消用户安全标签或豁免。

## 语法




FROM 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *revoker* | 要取消的权限的授予者的授权标识符 | 必须为指定的权限的授予者 | 所有者名称 |
| *role* | 您从其取消另一角色的角色 | 必须存在 | 所有者名称 |
| *user* | 您取消其角色（或缺省的角色）的用户 | 必须存在 | 所有者名称 |

## 用法

要取消已经通过表达式分片的表的一个或多个分片上的权限，请参阅 REVOKE FRAGMENT 语句。

对于您尝试取消一些数据库对象上的权限，如果任何下列条件为真，则您可取消权限：

- 您授予它们，且不指定另一用户作为授予者。
- GRANT 语句指定了您作为授予者。
- 您正在从 PUBLIC 对您拥有的对象取消权限，且当您创建该对象时缺省地授予了那些权限。
- 您有数据库级 DBA 权限且您在 AS 子句中指定了该权限的授予者的用户名称。

REVOKE 语句可取消用户，或 PUBLIC，或角色当前持有的任何下列访问权限或角色：

- 对数据库的权限（但角色不可持有数据库级权限）
- 对表、同义词、视图或序列对象的权限
- 对用户定义的数据类型（UDT）、用户定义的例程（UDR），或对 SPL 语言的权限
- 非缺省的角色，或 PUBLIC 或用户的缺省的角色。

您不可从您自己取消权限。您不可从另一用户取消授予者状态。要取消通过 GRANT 语句的 AS *grantor* 子句授予给另一用户的权限，您必须有 DBA 权限，且您必须使用 AS 子句来指定作为 **revoker** 的用户。

如果您在引号中括起了 *revoker*、 *role* 或 *user*，则名称是区分大小写的，完全按照您输入的样子存储名称。在符合 ANSI 的数据库中，如果您不使用引号作为定界符，则以大写字母存储该名称。

## 从映射了的用户取消数据库服务器访问（UNIX™、Linux™）

使用 REVOKE ACCESS FROM 语句来从特定的映射了的用户移除代理用户属性。

仅用户 **gbasedbt** 或 DBSA 可运行 REVOKE ACCESS FROM 语句。

REVOKE ACCESS FROM 语句不影响通过 OS 级账户在主计算机上访问数据库服务器的 GBase 8s 用户账号名称的任何访问权限。

示例：

用户 **gbasedbt** 或 DBSA 可在支持映射了的用户的系统上运行下列语句，且映射了的用户之一为用户 **bob**：

此语句完全地移除了用户 **bob** 对数据库服务器的访问，除了当下列之一或全都为真之外：

- PUBLIC 被映射到代理用户属性。在此情况下，用户 **bob** 仍保持与 PUBLIC 组持有的相同的访问权限。
- 在数据库服务器访问的 GBase 8s 主计算机上，用户 **bob** 还是用户账户。

## 数据库级权限

数据库级权限的三个同心层，Connect、Resource 和 DBA，对数据库访问和控制依次递增的授权。仅拥有 DBA 权限的用户可授予或取消数据库级权限。

数据库级权限


由于权限的层级组织（如本节稍后描述的权限定义中所述的那样），如果您从拥有 DBA 权限的用户取消 Resource 权限或 Connect 权限，则该语句没有作用。如果您从有 DBA 权限的用户取消该 DBA 权限，则该用户保留对该数据库的 Connect 权限。要拒绝有 DBA 或 Resource 权限的用户访问数据库，您必须先取消 DBA 或 Resource 权限，然后在单独的 REVOKE 语句中取消 Connect 权限。

类似地，如果您从有 Resource 权限的用户取消 Connect 权限，则该语句没有作用。如果您从用户取消 Resource 权限，则该用户保留对数据库的 Connect 权限。

仅用户或 PUBLIC 可持有数据库级权限。您不可从角色取消这些权限，因为角色不可持有数据库级权限。

下表罗列每一数据库级权限的关键字。

| 权限 | 作用 |
| --- | --- |
| DBA | 有 Resource 权限的所有能力，且可执行下列附加的操作：<br>• 将包括 DBA 权限在内的所有数据库级权限授予另一用户。<br>• 将任何表级权限授予另一用户或角色。<br>• 将角色授予用户或另一角色。<br>• 取消其授予者为您在 REVOKE 语句的 AS 子句中作为 ***revoker*** 的权限。<br>• 当注册 UDR 时，限制 DBA 的 Execute 权限。<br>• 执行 SET SESSION AUTHORIZATION 语句。<br>• 创建任何数据库对象。<br>• 创建表、视图和索引，指定另一用户作为这些对象的所有者。<br>• 变更、删除或重命名数据库对象，不管其所有者是谁。<br>• 执行 UPDATE STATISTICS 语句的 DROP DISTRIBUTIONS 选项。<br>• 执行 DROP DATABASE 和 RENAME DATABASE 语句。 |
| RESOURCE | 让您扩展数据库的结构。除了 Connect 权限的能力之外，Resource 权限的持有者还可执行下列操作：<br>• 创建新表。<br>• 创建新索引。<br>• 创建新的用户定义的例程。<br>• 创建新的数据类型。<br>• 授予用户注释权限（COMMENT ANY TABLE）。 |
| CONNECT | 如果您有此权限，则可查询和修改数据，如果您拥有您想要修改的数据库对象，则可修改该数据库模式。持有 Connect 权限的用户可执行下列操作：<br>• 以 CONNECT 语句或另一连接语句连接到数据库。<br>• 执行 SELECT、INSERT、UPDATE 和 DELETE 语句，只要用户有必要的表级权限。 |

| 权限 | 作用 |
| --- | --- |
| | • 创建视图，只要用户有对基础表的 Select 权限。<br>• 创建同义词。<br>• 创建临时表，并在临时表上创建索引。<br>• 变更或删除表或索引，如果用户拥有该表或索引（或有对表的 Alter、Index 或 References 权限）。<br>• 授予对表的权限，如果用户拥有该表（或通过 WITH GRANT OPTION 关键字被授予了对该表的权限)。 |

**提示**：要确定哪些用户有对数据库的 DBA 权限，请从 DB-Access 或您的应用系统运行此查询：

```
select username,usertype from sysusers;
```

输出展示用户名（例如，public 和 gbasedbt）后跟下列代码之一：

- D = DBA 权限
- C = Connect 权限
- R = Resource 权限

## 表级权限

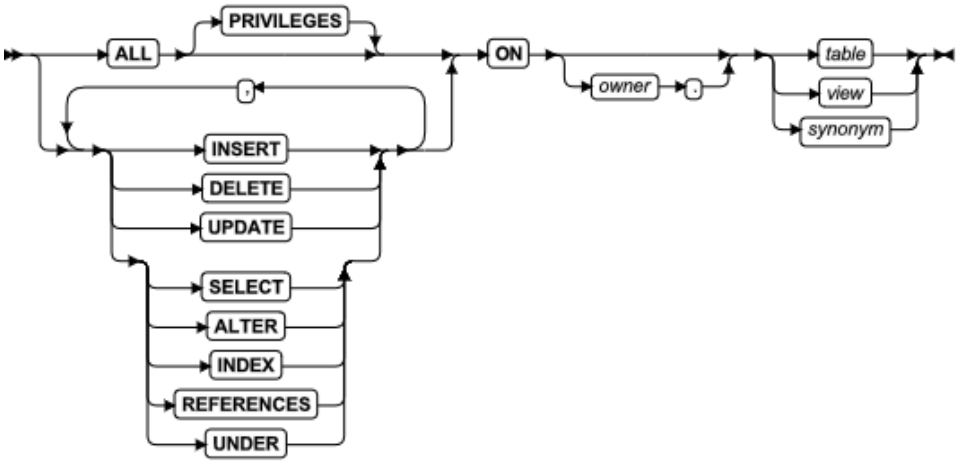
表级权限，也称为***表权限***，指定用户或角色可在数据库中的表或视图上执行哪些操作。您可使用同义词来指定您在其上授予或取消表权限的表或视图。

可在表或视图的列的子集上授予 Select、Update 和 References 权限，但仅可对所有列取消权限。对于在定义同一用户拥有的视图的 SELECT 语句中引用的表，如果从用户取消 Select 权限，则删除那个视图，除非它还包括来自另一数据库中表的列。

对于 CREATE EXTERNAL TABLE 语句已经在当前的数据库中注册了的表对象，仅支持 Select 权限和 Insert 权限；不可授予或取消其他表或列访问权限。

使用下列语法来指定从一个或多个用户或角色取消哪些表级权限：

表级权限



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *owner* | 拥有 *table*、*view* 或 *synonym* 的用户的名称 | 必须为有效的授权标识符 | 所有者名称 |
| *synonym*、*table*、*view* | 在其上授予权限的同义词、表或视图 | 在当前的数据库中必须存在 | 标识符 |

在一 REVOKE 语句中，您可罗列一个或多个下列关键字来指定对指定的表要从用户或角色取消的权限。

| 权限 | 在 REVOKE 之后的作用 |
| --- | --- |
| INSERT | 用户不可插入行。 |
| DELETE | 用户不可删除行。 |
| SELECT | 用户不可通过 SELECT 语句显示检索的数据。 |
| UPDATE | 用户不可更改列值。 |
| INDEX | 用户不可创建永久的索引。您必须有 Resource 权限来利用 Index 权限。（但任何有 Connect 权限的用户都可在临时表上创建索引。） |
| ALTER | 持有者不可添加或删除列、修改列数据类型、添加或删除约束、将表的锁定模式由 PAGE 更改为 ROW，也不可添加或删除相应的命名了的 ROW 类型表。用户还不可启用或禁用索引、约束以及触发器，如 SET Database Object Mode 语句 中所述。 |
| REFERENCES | 用户不可引用引用约束中的列。您还必须有对该数据库的 Resource 权限来利用表上的 References 权限。（然而，您可在 ALTER TABLE 语句期间添加引用约束，而不持有对该数据库的 Resource 权限。)取消 References 权限不允许级联 DELETE 操作。 |
| UNDER | 用户不可创建在类型表之下的子表。 |
| ALL | 这将移除以上罗列的所有表权限。（此处的 PRIVILEGES 关键字是可选的。） |

如果用户从两个不同的授予者收到相同的权限，且一授予者取消该权限，则被授予者仍有该权限，直到第二个授予者也取消该权限为止。例如，如果您和 DBA 都将对您的表的 Update 权限授予 **ted**，则您和 DBA 必须都取消 Update 权限来阻止 **ted** 更新您的表。

然而，如果用户 **ted** 通过角色或作为 PUBLIC 持有相同的权限，则此 REVOKE 操作不能阻止 **ted** 行使 Update 权限。

## 何时在 GRANT 之前使用 REVOKE

您可使用 REVOKE 与 GRANT 的组合来以特定的用户替代 PUBLIC 作为被授予者，并移除对一些列的表级权限。

### *以指定的用户替代 PUBLIC*

如果表所有者将权限授予 PUBLIC，则该所有者不可从任何特定的用户取消同一权限。例如，假设 PUBLIC 对您的 customer 表有缺省的 Select 权限。假设您发出下列语句，尝试阻止 ted 访问您的表：

```
REVOKE ALL ON customer FROM ted;
```

此语句导致 ISAM 错误消息 111，No record found，因为系统目录表（**syscolauth** 或 **systabauth**）未包含名为 **ted** 的用户的表级权限条目。此 REVOKE 操作不阻止 **ted** 保持所有赋予 PUBLIC 的对 **customer** 表的表级权限。

要限制表级权限，首先以 PUBLIC 关键字取消该权限，然后将它们重新授予某适当的用户和角色列表。下列语句从所有用户取消对 customer 表的 Index 和 Alter 权限，然后特定地将这些权限授予用户 mary：

```
REVOKE INDEX, ALTER ON customer FROM PUBLIC;
GRANT INDEX, ALTER ON customer TO mary;
```

### *限制对特定列的访问*

不同于 GRANT，REVOKE 语句没有语法来对表中的列的子集指定权限。要从用户取消对列的 Select、Update 或 References 权限，您必须对该表的所有列取消该权限。要提供对您先前已在其上授予了权限的一些列的访问，请发出新的 GRANT 语句来恢复对特定列的适当的权限。

下一示例取消 PUBLIC 对特定列的 Select 权限：

```
REVOKE SELECT ON customer FROM PUBLIC;
GRANT SELECT (fname, lname, company, city) ON customer TO PUBLIC;
```

在下一示例中，mary 首先获得对 customer 中四列的引用能力，然后该表拥有者将引用限定为两列：

```
GRANT REFERENCES (fname, lname, company, city) ON customer TO mary;
        REVOKE REFERENCES ON customer FROM mary;
GRANT REFERENCES (company, city) ON customer TO mary;
```

## ALL 关键字的作用

ALL 关键字取消所有表级权限。如果对于被取消者任何或所有表级权限不存在，则带有 ALL 关键字的 REVOKE 执行成功但返回下列 SQLSTATE 代码：

```
01006--Privilege not revoked
```

例如，假设用户 hal 有对 customer 表的 Select 和 Insert 权限。用户 jocelyn 想要从用户 hal 取消所有表级权限。于是用户 jocelyn 发出下列 REVOKE 语句：

```
REVOKE ALL ON customer FROM hal;
```

此语句执行成功但返回 SQLSTATE 代码 01006。返回该 SQLSTATE 警告是因为下列二者都为真：

- 该语句成功地从用户 **hal** 取消 Select 和 Insert 权限，因为用户 **hal** 有那些权限。
- 返回 **SQLSTATE** 代码 01006 是因为用户 **hal** 缺少通过 ALL 关键字隐含的其他权限，但这些权限还未被取消。

ALL 关键字指示数据库管理器来取消所有可能的权限，包括无任何权限。如果从其取消权限的用户没有对该表的权限，则 REVOKE ALL 语句仍然成功，因为它从该用户取消所有可能的权限（在此情况下，根本没有权限）。

### *ALL 关键字对 UNDER 权限的作用*

如果您对类型表取消 ALL 权限，则 Under 权限包括在被取消的权限之中。如果您对不是基于 ROW 类型的表取消 ALL 权限，则 Under 权限不包括在被取消的权限之中。（仅可对类型表授予 Under 权限。）

## 未提交的事务的影响

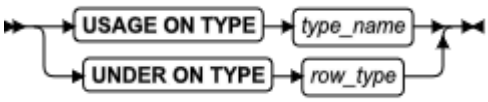
REVOKE 对取消权限的表在 **systables** 系统目录表中的条目上放置一排他的行锁。直到包含 REVOKE 语句的事务终止，才释放此锁。当另一事务尝试对此表准备 SELECT 语句，而第一个事务是打开的，则并发事务失败，因为指定的表的 **systables** 行仍然排他地锁定着。直到第一个事务或提交或回滚，要准备 SELECT 语句的尝试才可成功。

## 类型级权限

您可对数据类型取消两权限：

- 对用户定义的数据类型的 Usage 权限
- 对命名的 ROW 类型的 Under 权限

类型级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *row_type* | 对其取消 Under 权限的命名的 ROW 类型 | 必须存在 | 数据类型， |
| *type_name* | 对其取消 Usage 权限的用户定义的类型 | 必须存在 | 数据类型， |

## Usage 权限

任何用户都可引用 SQL 语句中的内建的数据类型，但不可引用基于内建的数据类型的 DISTINCT 数据类型。用户定义的数据类型的创建者或 DBA 必须显式地授予对 UDT 的 Usage 权限，包括基于内建的数据类型的 DISTINCT 数据类型。

带有 USAGE ON TYPE 关键字的 REVOKE 移除您稍早授予另一用户、PUBLIC 或角色的 Usage 权限。

下列语句从用户 mark 移除使用 widget 用户定义的类型的权限：

```
REVOKE USAGE ON TYPE widget FROM mark;
```

## Under 权限

您拥有您创建的任何命名的 ROW 数据类型。如果您想要其他用户能够在此命名的 ROW 类型之下创建子类型，则您必须授予这些用户对于您的命名的 ROW 类型的 Under 权限。如果您稍后想要移除这些用户在该命名的 ROW 类型之下创建子类型的能力，则您必须从这些用户取消 Under 权限。带有 UNDER ON TYPE 关键字的 REVOKE 语句移除您稍早授予这些用户的 Under 权限。

例如，假设您创建了名为 rtype1 的 ROW 类型：

```
CREATE ROW TYPE rtype1 (cola INT, colb INT);
```

如果您想要另一名为 kathy 的用户能够在此命名的 ROW 类型之下创建子类型，则您必须将对于此命名的 ROW 类型的 Under 权限授予用户 kathy：

```
GRANT UNDER ON TYPE rtype1 TO kathy;
```

现在，即使 kathy 不是 rtype1 ROW 类型的所有者，用户 kathy 也可在 rtype1 ROW 类型之下创建另一 ROW 类型：

```
CREATE ROW TYPE rtype2 (colc INT, cold INT) UNDER rtype1;
```

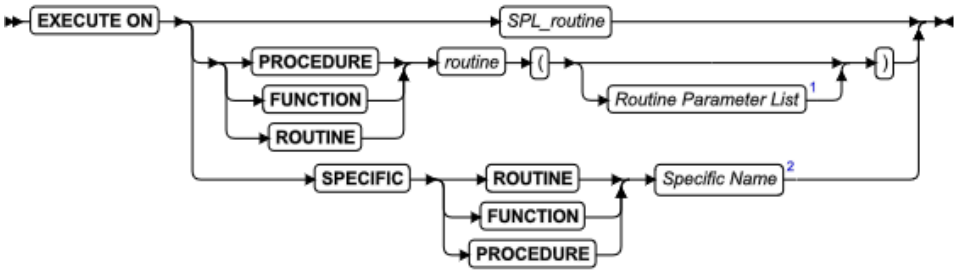
如果您稍后想要移除用户 kathy 在 rtype1 ROW 类型之下创建子类型的能力，则请输入下列语句：

```
REVOKE UNDER ON TYPE rtype1 FROM kathy;
```

## 例程级权限

如果您从用户取消对 UDR 的 Execute 权限，则那个用户不可再以任何方式执行那个 UDR。要获取用户可如何执行 UDR 的详细信息，请参阅 例程级权限。

例程级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *routine* | 用户定义的例程 | 必须存在 | 标识符 |
| *SPL_routine* | SPL 例程 | 在该数据库中必须为唯一的 | 标识符 |

在符合 ANSI 的数据库中，***owner*** 名称必须限定 ***routine*** 名称，除非发出 REVOKE 语句的用户为该例程的所有者。

下列示例取消用户 **mark** 对由 **luke** 所拥有的 **delete_order** 例程的 Execute 权限：

```
REVOKE EXECUTE ON ROUTINE luke.delete_order FROM mark;
```

在 GBase 8s 中，任何您授予 Execute 权限的取反函数都需要单独的、显式的 REVOKE 语句。

当您在任何下列环境之下创建 UDR，都不会缺省地授予 PUBLIC Execute 权限。因此，在您可取消它之前，您必须显式地授予 Execute 权限：

- 您在符合 ANSI 的数据库中创建 UDR。
- 您有 DBA 权限且在 CREATE 关键字之后指定 DBA 来将 Execute 权限限定给拥有 DBA 数据库级权限的用户。
- **NODEFDAC** 环境变量设置为 yes 来防止 PUBLIC 收到任何未被显式地授予的权限。

但如果您在没有任何那些条件生效的情况下创建 UDR，则 PUBLIC 可无需 GRANT EXECUTE 语句即可执行您的 UDR。要限定谁可执行您的 UDR，请通过 FROM PUBLIC 取消 Execute 权限，并将它授予用户（请参阅 用户列表）或角色（请参阅 角色名称）。

在 GBase 8s 中，如果两个或多个 UDR 有相同的名称，则请从此列表使用关键字来指定用户列表可不再执行那些 UDR 中的哪些。

| **关键字** | **防止用户执行的 UDR** |
|---|---|
| SPECIFIC | 通过 ***specific name*** 标识的 UDR |
| FUNCTION | 任何带有指定的 ***routine name*** 的函数（以及与 ***routine 参数列表***相匹配的参数类型，如果指定的话） |
| PROCEDURE | 任何带有指定的 ***routine name*** 的过程（以及与 ***routine 参数列表***相匹配的参数类型，如果指定的话） |

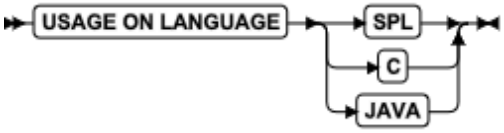
| | |
|---|---|
| ROUTINE | 带有指定的 ***routine name*** 的函数或过程（以及与 ***routine 参数列表***相匹配的参数类型，如果指定的话） |

## 语言级权限

要注册或删除用 SPL、C 或 Java™ 语言编写的 UDR，用户必须持有对用以编写该例程的编程语言的 Usage 权限。

这是对于指定要取消的语言级权限的 USAGE ON LANGUAGE 子句的语法：

语言级权限




每一 REVOKE USAGE ON LANGUAGE 语句可指定不多于一种编程语言。

当用户注册以 SPL、C 或 Java 语言编写的 UDR 时，数据库服务器验证该用户是否有对用以编写该 UDR 的语言的 Usage 权限。 如果该用户不具权限，则 CREATE FUNCTION 或 CREATE PROCEDURE 语句失败并报错。 如果 IFX_EXTEND_ROLE 配置参数已启用内建的 EXTEND 角色，则仅还持有那个角色的用户可注册或删除以 C 语言或以 Java 语言编写的 UDR，即使用户持有对那些语言的 USAGE ON LANGUAGE 权限。

要从用户或角色取消对编程语言的 Usage 权限，请发出包括 USAGE ON LANGUAGE 关键字和指定该编程语言的关键字的 REVOKE 语句。如果此语句成功，则任何您在 FROM 子句中指定的用户或角色都可不再注册那些以该语言编写的 UDR。例如，如果您从 PUBLIC 取消对 SPL 的缺省的 Usage 权限，则从所有用户剥夺创建 SPL 例程的能力（除了那些已被单独地授予了对该 SPL 语言的 Usage 权限的用户，或通过角色持有那 Usage 权限的用户之外）：

```
REVOKE USAGE ON LANGUAGE SPL FROM PUBLIC;
```

您可发出 GRANT USAGE ON LANGUAGE 语句来给受限的组恢复对 SPL 的 Usage 权限，比如，给名为 developers 的角色：

```
GRANT USAGE ON LANGUAGE SPL TO developers;
```

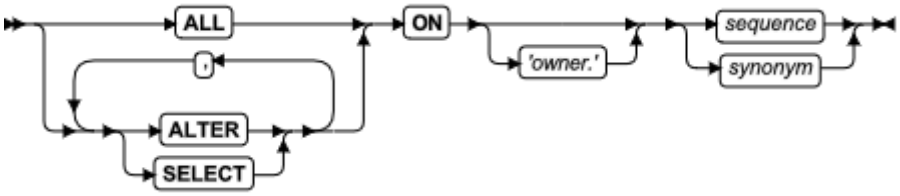
## 序列级权限

虽然 GBase 8s 以表的形式实现序列对象，但仅可对序列授予或取消表权限的下列子集（如 表级权限 中所述）：

- Select 权限
- Alter 权限

使用下列语法来指定要对序列对象取消的权限：

序列级权限




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | 序列或其同义词的所有者 | 必须为所有者 | 所有者名称 |
| sequence | 要取消其权限的序列 | 必须存在 | 标识符 |
| *synonym* | 序列对象的同义词 | 必须指向序列 | 标识符 |

序列必须驻留在当前的数据库中。(您可以有效的 ***owner*** 名称限定 ***sequence*** 或 ***synonym*** 标识符，但远程的 ***database*** 的名称(或 ***database@server***)为无效限定符。)取消序列级权限的语法是对 SQL 的 ANSI/ISO 标准的扩展。

## Alter 权限

您可从另一用户、从 PUBLIC 或从角色取消对序列的 Alter 权限。Alter 权限使得指定的用户或角色能以 ALTER SEQUENCE 语句修改序列的定义，或以 RENAME SEQUENCE 语句重命名该序列。

下列 REVOKE 语句取消分别授予用户 mark 对 cust_seq 序列对象的任何 Alter 权限：

```
REVOKE ALTER ON cust_seq FROM mark;
```

## Select 权限

您可从另一用户、从 PUBLIC 或从角色取消对序列的 Select 权限。Select 权限使得用户或角色能使用 SQL 语句中的 sequence.CURRVAL 和 sequence.NEXTVAL 来访问和增大序列的值。

下列 REVOKE 语句取消单独地授予用户 mark 对 cust_seq 序列对象的任何 Select 权限：

```
REVOKE SELECT ON cust_seq FROM mark;
```

## ALL 关键字

您可使用 ALL 关键字来从另一用户、从 PUBLIC 或从角色同时取消 Alter 和 Select 权限。

下例取消用户 mark 对 cust_seq 序列对象持有的任何 Alter 和 Select 权限：

```
REVOKE ALL ON cust_seq FROM mark;
```

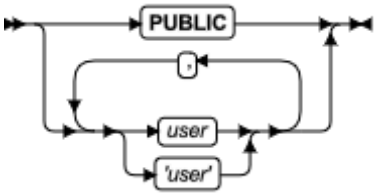
在此语句执行之后，mark 是否仍可访问 cust_seq 取决于该用户是否仍持有授予 PUBLIC 的对 cust_seq 的 Alter 或 Select 权限，或他是否持有已授予了的对 cust_seq 的未取消的权限的角色。

## 用户列表

跟在 REVOKE 的 FROM 关键字之后的授权标识符（或 PUBLIC 关键字）指定失去取消了的权限或取消了的角色的用户。如果您使用 PUBLIC 关键字作为用户列表，则 REVOKE 语句从 PUBLIC 取消指定的权限或角色，从而从所有用户到尚未显式地授予权限或角色的用户，或从未通过已收到的角色或权限而持有一些其他角色的用户取消它们。

*user list* 可包含单个用户或多个用户的授权标识符，以逗号分隔。如果您使用 PUBLIC 关键字作为用户列表，则 REVOKE 语句从所有用户取消指定的权限。

用户列表




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *user* | 您正在取消其权限的用户的登录名 | 必须为有效的授权标识符 | 所有者名称 |

在该列表中用户名称的拼写与在 GRANT 语句中的拼写完全一致。您可有选择地在该列表中使用括起每一用户名称的引号来保留大小写。在符合 ANSI 的数据库中，如果您不使用引号来定界 ***user***，则以大写字母存储该用户的名称，除非在初始化数据库服务器之前 **ANSIOWNER** 环境变量设置成了 1。

当您指定登录名时，可使用 REVOKE 语句和 GRANT 语句来有选择地确保数据库对象的各种类型。要了解示例，请参阅 何时在 GRANT 之前使用 REVOKE。

## 角色名称

仅 DBA 或通过 WITH GRANT OPTION 被授予了角色的用户可取消角色或其权限。用户不可取消自身的角色。

角色名称


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *role* | 有这些属性之一的角色：<br>• 失去现有的权限或角色<br>• 被用户或被另一角色失 | 必须存在。如果括在引号之间，则 *role* 区分大小写。 | 所有者名称 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 去 | | |

紧跟在 REVOKE 关键字之后，***role*** 的名称指定要从用户列表取消的角色。然而，在 FROM 关键字之后，***role*** 的名称指定要从其取消访问权限（或另一角色）的角色。如果没有其他的 REVOKE 选项与 ***user*** 或 ***role*** 规范相冲突，则同一 FROM 子句可同时包括 ***user*** 和 ***role*** 名称。要从角色取消对角色的权限的语法是对 SQL 的 ANSI/ISO 标准的扩展。

当您在 REVOKE 语句的 FROM 关键字之后包括 ***role***，则从那个角色取消指定的权限（或另一角色），但有那个角色的用户保留那些单独地授予给他们的任何权限或角色。

如果您将 ***role*** 括在引号之间，则该名称区分大小写，且完全按照您输入的形式存储。在符合 ANSI 的数据库中，如果您不使用引号作为定界符，则以大写字母形式存储 ***role***。

当您取消以 WITH GRANT OPTION 关键字授予了用户的角色时，您同时取消该角色及授予它的选项。

下列示例展示 REVOKE *role* 的作用：

- 移除用户或从指定的角色中包含的项移除另一角色：

```
REVOKE accounting FROM mary;
REVOKE payroll FROM accounting;
```

- 从角色移除一个或多个访问权限：

```
REVOKE UPDATE ON employee FROM accounting;
```

当您从角色取消表级权限时，您不可包括 RESTRICT 或 CASCADE 关键字。

## 取消缺省的角色

DBA 或数据库的所有者可以 GRANT DEFAULT ROLE 语句为一个或多个用户或为 PUBLIC 定义***缺省的角色***。不同于非缺省的角色，当用户连接到数据库时，缺省的角色自动地生效。直到 SET ROLE 语句激活该角色，非缺省的角色才生效。缺省的角色可为被授予了那个缺省的角色的所有用户指定一系列访问权限。相反地，REVOKE DEFAULT ROLE 语句为指定的 *user-list* 取消当前的缺省的角色作为缺省的角色，如下列程序片段所示：

```
CREATE ROLE accounting;
GRANT USAGE ON LANGUAGE SPL TO accounting;
GRANT ALL PRIVILEGES ON receivables TO accounting;
GRANT DEFAULT ROLE accounting TO mary;
. . .
REVOKE DEFAULT ROLE FROM mary;
```

最后的语句从用户 **mary** 移除任何她仅通过她的缺省的角色所持有的任何访问权限。在此示例中，该缺省的角色为 **accounting**，但由于在给定的时间点对于单个用户（或 PUBLIC 组）可仅有一个角色，所以在 REVOKE DEFAULT ROLE 语句中不指定缺省的角色的名称。如果 **mary** 发出 SET ROLE DEFAULT 语句，则直到授予他某新的缺省的角色之后，它才会有效。

在您执行指定一个或多个用户或 PUBLIC 的 REVOKE DEFAULT ROLE 之后，仅通过缺省的用户，才能取消那些用户持有的任何权限。（但此语句不取消单独地授予了用户的任何权限，或通过另一角色授予了用户的权限，或 PUBLIC 持有的权限。）

在 REVOKE DEFAULT ROLE 成功地取消 ***user*** 的缺省的角色之后，***user*** 的缺省的角色成为 NULL，且从系统目录移除该缺省的角色信息。（在此上下文中，NULL 与 NONE 是同义词。）

如果 REVOKE DEFAULT ROLE 指定尚未授予缺省的角色的用户，则不发出警告。

除了 ***user-list*** 之外，在 REVOKE DEFAULT ROLE 语句中的 FROM 关键字之后的选项都无效。

## 取消 EXTEND 角色

REVOKE EXTEND FROM ***user-list*** 语句取消指定用户的 EXTEND 角色。在启用 IFX_EXTEND_ROLE 配置参数的数据库中，取消此角色防止指定的用户创建或删除外部 UDR。用户是否持有 EXTEND 角色对创建或删除以 SPL 语言编写的 UDR 没有作用。

仅数据库服务器管理员（DBSA），缺省为用户 **gbasedbt**，可通过发出 GRANT EXTEND TO ***user-list*** 语句将内建的 EXTEND 角色授予一个或多个用户或授予 PUBLIC。（由于 EXTEND 为内建的角色，因此持有它的用户不需要以 SET ROLE 语句激活它，且 DROP ROLE 语句不可销毁 EXTEND 角色。）

如果 IFX_EXTEND_ROLE 配置参数设置为 ON 或为 1，则不持有 EXTEND 角色的用户不可创建或删除以 C 或 Java™ 语言编写的 UDR，这两种语言都支持共享库。下列示例从用户 max 取消 EXTEND 角色：

```
REVOKE EXTEND FROM 'max';
```

这防止用户 **max** 创建或删除外部的 UDR，即使 **max** 为它后续试图删除的那个 UDR 的所有者。

在不需要此安全特性的数据库中，DBSA 可通过设置 ONCONFIG 文件中的 IFX_EXTEND_ROLE 参数为 OFF 或为 0 来禁用对可创建或删除外部的 UDR 的用户的限制。但不论启用或禁用 IFX_EXTEND_ROLE，创建或删除外部的 UDR 的用户还必须持有下列访问权限：

- 对在其中注册 UDR 的数据库的 Resource 权限或 DBA 权限。
- 对以其编写 UDR 的外部编程语言的 Usage 权限。

要获取关于 Resource 权限的信息，请参阅 数据库级权限。要获取 SQL 的 GRANT USAGE ON LANGUAGE C 和 GRANT USAGE ON LANGUAGE JAVA 的语法，请参阅 语言级权限。

## 取消 WITH GRANT OPTION 授予的权限

如果您从 ***user*** 取消您使用 WITH GRANT OPTION 关键字授予的权限或角色，则切断由那个 ***user*** 授予的权限的链条。

这样，当您从用户或从角色取消权限时，您还取消了在下列上下文中 GRANT 语句所产生的相同的权限：

- 由您的被授予者发出的
- 允许，因为您的被授予者指定了 WITH GRANT OPTION 子句

- 允许，因为后续的被授予者使用 WITH GRANT OPTION 子句授予了相同的权限或角色

在给特定的用户指定权限的 GRANT 语句中，仅 WITH GRANT OPTION 子句是有效的。被授予者不可为 PUBLIC 组或角色。

下列示例展示权限的取消。假设您，作为表 items 的所有者，发出下列语句来将访问权限授予用户 mary：

```
REVOKE ALL ON items FROM PUBLIC;
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION;
```

然后，用户 mary 使用她的新权限来授予用户 cathy 和 paul 对 items 表的访问：

```
GRANT SELECT, UPDATE ON items TO cathy;
        GRANT SELECT ON items TO paul;
```

稍后，您从用户 mary 取消对 items 表的权限：

```
REVOKE SELECT, UPDATE ON items FROM mary;
```

此单一语句有效地从用户 **mary**、**cathy** 和 **paul** 取消对 **items** 表的所有权限。

CASCADE 关键字与此缺省的情况有相同的作用。

## AS 子句

若没有 AS 子句，执行 REVOKE 语句的用户必须为正被取消的权限的授予者。DBA 或该对象的所有者可使用 AS 子句来指定另一用户（必须为该权限的授予者）作为该权限的取消者。

AS 子句提供唯一的机制，可取消对其 ***owner*** 为诸如 **gbasedbt** 这样的授权标识符的数据库对象的权限，该标识符还不是操作系统已知的有效的用户账户。

要了解 AS ***revoker*** 子句需要的，而不是可选的，上下文，请参阅 将 Execute 权限从 PUBLIC 取消。

## CASCADE 关键字对 UNDER 权限的作用

如果您以 CASCADE 选项取消对类型表的 Under 权限，则从指定的用户移除 Under 权限，并从数据库删除那个用户在该类型表之下创建的任何子表。

如果当那个数据类型正在使用时，您以 CASCADE 选项取消对命名的 ROW 类型的 Under 权限，则 REVOKE 失败。对 CASCADE 选项的缺省的行为会发生例外，因为数据库服务器支持仅带有 RESTRICT 关键字的 DROP ROW TYPE 语句。

例如，假设用户 **jeff** 创建名为 **rtype1** 的 ROW 类型，并将对那个 ROW 类型的 Under 权限授予用户 **mary**。现在，用户 **mary** 在 ROW 类型 **rtype1** 之下创建名为 **rtype2** 的 ROW 类型，并将对 ROW 类型 **rtype2** 的 Under 权限授予用户 **andy**。然后，用户 **andy** 在 ROW 类型 **rtype2** 之下创建名为 **rtype3** 的 ROW 类型。

如果现在用户 **jeff** 试图以 CASCADE 选项从用户 **mary** 取消对 ROW 类型 **rtype1** 的 Under 权限，则 REVOKE 语句失败，因为 ROW 类型 **rtype2** 仍在被 ROW 类型 **rtype3** 所使用。

## 以 RESTRICT 选项控制 REVOKE 的作用域

当任何下列依赖存在时，RESTRICT 关键字导致 REVOKE 语句失败：

- 视图依赖于您正在尝试取消的 Select 权限。
- 外键约束依赖于您尝试取消的 References 权限。
- 您尝试从一用户取消权限，该用户后来将此权限授予了另一用户或角色。

如果 REVOKE 指定有将该权限授予其他用户的权限，但尚未使用那项权利的用户，则 REVOKE 不会失败。例如，假设当用户 clara 将对 customer 表的 Select 权限授予用户 ted 时，它指定 WITH GRANT OPTION。进一步假设用户 ted，接着将对 customer 表的 Select 权限授予用户 tania。clara 发出了的下列语句不起作用，因为 ted 已使用了它的授权来授予 Select 权限：

```
REVOKE SELECT ON customer FROM ted RESTRICT;
```

相反，如果用户 ted 未将 Select 权限授予 tania 或任何其他用户，则同样的 REVOKE 语句成功。即使 ted 确将 Select 权限授予另一用户，下列语句的每一条均成功：

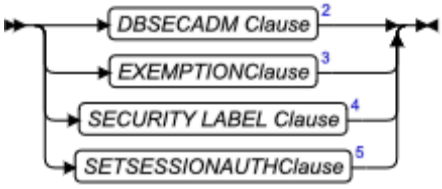
```
REVOKE SELECT ON customer FROM ted CASCADE;
REVOKE SELECT ON customer FROM ted;
```

## 安全管理选项

结合 GRANT 语句，REVOKE 语句通过指定哪些用户或角色持有需要访问数据库或数据库之内的对象的权限，支持 GBase 8s 的自主访问控制（DAC）数据安全特性。

REVOKE 语句的“安全管理选项”，与 GRANT 语句的对应选项相似，支持一系列附加的数据安全特性，称为基于标签的访问控制（LBAC）。 这些特性使得 GBase 8s 能基于将包含在数据对象中的行安全标签或列安全标签与用户标签及其他已经授予了正在寻求访问的用户的其他凭证进行比较，允许或拒绝对受保护的数据的访问。

安全管理选项



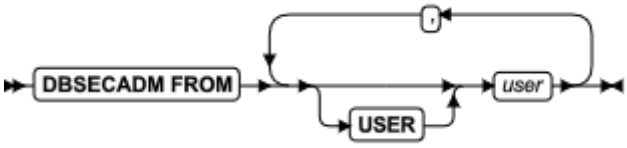

使用这些 REVOKE 语句安全管理选项是受限的：

- 仅“数据库服务器管理员”（DBSA），缺省为用户 **gbasedbt**，可使用 REVOKE DBSECADM 语句来取消 DBSECADM 角色。
- 仅持有 DBSECADM 角色的用户可发出 REVOKE EXEMPTION、REVOKE SECURITY LABEL 或 REVOKE SETSESSIONAUTH 语句。

# DBSECADM 子句

REVOKE DBSECADM 语句防止被授予了 DBSECADM 角色的用户发出可创建、改变、重命名或删除安全对象的 DDL 语句，安全对象包括安全策略、安全标签和安全组件。

DBSECADM 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *user* | 要从其取消角色的用户 | 必须为用户的授权标识符 | 所有者名称 |

DBSECADM 角色是仅 DBSA 可取消的内建角色。与用户定义的角色不同，DBSECADM 角色的作用域是 GBase 8s 实例的所有数据库。用户定义的角色的作用域是在其中创建该角色的数据库。DBSA 不必在同一服务器的其他数据库中重新发出 REVOKE DBSECADM 语句。

仅持有 DBSECADM 角色的用户可发出下列创建或更改安全对象的 SQL 语句：

- ALTER SECURITY LABEL COMPONENT
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- DROP SECURITY LABEL
- DROP SECURITY LABEL COMPONENT
- DROP SECURITY POLICY
- RENAME SECURITY LABEL
- RENAME SECURITY LABEL COMPONENT
- RENAME SECURITY POLICY

仅持有 DBSECADM 角色的用户可使用下列 SQL 语句来引用受安全策略保护的表：

- ALTER TABLE ... ADD SECURITY POLICY
- ALTER TABLE ... ADD ... IDSSECURITYLABEL [DEFAULT ***label***]
- ALTER TABLE ... ADD ... [COLUMN] SECURED WITH
- ALTER TABLE ... DROP SECURITY POLICY
- ALTER TABLE ... MODIFY ... [COLUMN] SECURED WITH
- ALTER TABLE ... MODIFY ... DROP COLUMN SECURITY
- CREATE TABLE ... COLUMN SECURED WITH
- CREATE TABLE ... IDSSECURITYLABEL [DEFAULT ***label***]
- CREATE TABLE ... SECURITY POLICY

不持有 DBSECADM 角色的用户也不可发出下列 GRANT 和 REVOKE 语句：

- GRANT EXEMPTION
- GRANT SECURITY LABEL
- GRANT SETSESSIONAUTH
- REVOKE EXEMPTION
- REVOKE SECURITY LABEL
- REVOKE SETSESSIONAUTH

可跟在 FROM 关键字之后的 USER 关键字是可选的，且不起作用，但 DBSA 在 REVOKE DBSECADM 语句中指定的任何授权标识符必须为单个用户的标识符，而不是角色的标识符。***user*** 不可为发出此 REVOKE DBSECADM 语句的 DBSA.

在下例中，DBSA 取消用户 **niccolo** 的 DBSECADM 角色：

```
REVOKE DBSECADM FROM niccolo;
```

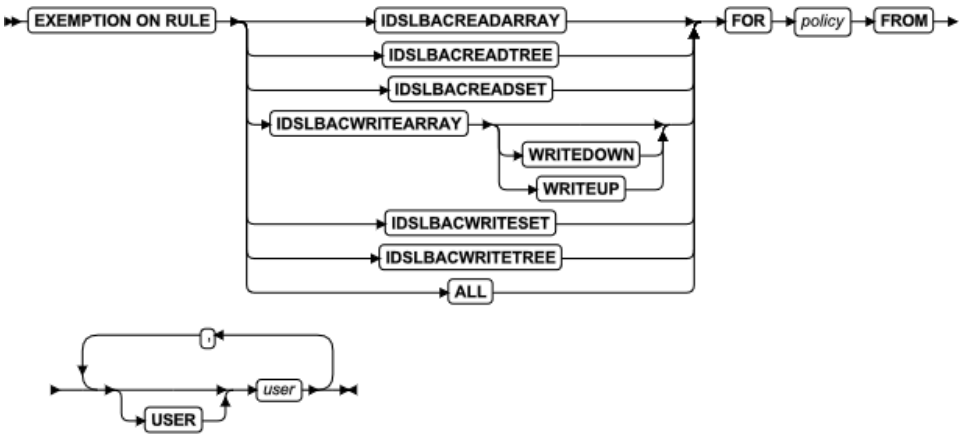
如果此语句执行成功，则用户 **niccolo** 可不再执行以上罗列的操作。

在取消 DBSECADM 角色之后，仅 DBSA 可再次将它授予被取消了它的用户。

## EXEMPTION 子句

REVOKE EXEMPTION 语句通过启用该用户已被豁免的指定的安全策略的一个或所有规则，修改指定的用户（或用户列表）的安全凭证。

EXEMPTION 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *policy* | 被取消豁免的安全策略 | 在数据库中必须存在 | 标识符 |
| *user* | 要被取消豁免的用户 | 必须为用户的授权标识符 | 所有者名称 |

仅持有 DBSECADM 角色的用户可发出 REVOKE EXEMPTION 语句。

## *有关取消豁免的规则*

跟在 ON 关键字之后的关键字指定取消豁免的安全策略（其标识符跟在 FOR 关键字之后）的预定义的访问规则。当从其取消豁免的用户访问受指定的策略保护的表时，应用取消豁免的访问规则。要了解与安全策略相关联的读访问和写访问的预定义规则的描述，请参阅 与安全策略相关的规则部分。

下列 REVOKE EXEMPTION 语句的关键字标识此语句可适用于以前豁免的用户的特定的 IDSLBACRULES 规则：

- IDSLBACREADARRAY 适用于指定的安全策略的 **IDSLBACREADARRAY** 规则的用户。对于无豁免的用户，此规则要求用户安全标签的每一数组组件必须大于或等于数据行安全标签的相应的数组组件。
- IDSLBACREADSET 适用于指定的安全策略的 **IDSLBACREADSET** 规则的用户。对于无豁免的用户，此规则要求该用户安全标签的每一集合组件必须包括数据行安全标签的集合组件。
- IDSLBACREADTREE 适用于指定的安全策略的 **IDSLBACREADTREE** 规则的用户。对于无豁免的用户，此规则要求该用户安全标签的树组件必须包括数据行安全标签的树组件中的至少一个元素，或包括一个这样元素的祖先。
- IDSLBACWRITEARRAY WRITEDOWN 从指定的安全策略的 **IDSLBACWRITEARRAY** 规则的一个方面豁免该用户。失去此豁免的用户不可写到受标签保护的行，该标签包括低于在该用户的标签中级别的数组组件级别。
- IDSLBACWRITEARRAY WRITEUP 从指定的安全策略的 **IDSLBACWRITEARRAY** 规则的一个方面豁免该用户。失去此豁免的用户不可写到受标签保护的行，该标签包括高于在该用户的标签中的级别的数组组件级别。
- IDSLBACWRITEARRAY （不带有 WRITEDOWN 或 WRITEUP 关键字）适用于指定的安全策略的 **IDSLBACWRITEARRAY** 规则的用户。失去此豁免的用户不可写到其数组组件级别高于或低于在该用户的标签中的级别的行。
- IDSLBACWRITESET 适用于指定的安全策略的 **IDSLBACWRITESET** 规则的用户。对于无豁免的用户，那个规则要求该用户安全标签的每一集合组件必须包括数据行安全标签的集合组件。
- IDSLBACWRITETREE 适用于指定的安全策略的 **IDSLBACWRITETREE** 规则的用户。对于无豁免的用户，那个规则要求该用户安全标签的每一树组件必须包括数据行安全标签的树组件中至少一个元素，或包括一个这样元素的祖先。
- ALL 从指定的安全策略的所有 **IDSLBACRULES** 规则取消豁免。

在下例中，DBSECADM 从用户 manoj 和 sam 取消对 MegaCorp 安全策略的所有规则的豁免：

```
REVOKE EXEMPTION ON RULE ALL FOR MegaCorp FROM manoj, sam;
```

### *安全策略和豁免的被授予者*

豁免仅适用于单个安全策略的规则，其名称跟在 FOR 关键字之后。由于受到保护的表可有多个安全标签，但只有一个安全策略，因此撤销豁免可防止没有充分的安全凭证的用户访问有指定的安全策略保护的表中的数据。

如果在数据库中不存在指定的策略，则 REVOKE EXEMPTION 语句失败并报错。

可跟在 FROM 关键字之后的 USER 关键字是可选的，且没有作用，但在 REVOKE EXEMPTION 语句中指定的任何授权标识符必须是单个用户的标识符，而不是角色的标识符。此 ***user*** 不可为发出同一 REVOKE EXEMPTION 语句的 DBSECADM。

在下例中，DBSECADM 从用户 lynette 取消对 MegaCorp 安全策略的规则 IDSLBACREADARRAY 的豁免：

```
REVOKE EXEMPTION ON RULE IDSLBACREADARRAY FOR MegaCorp FROM lynette;
```

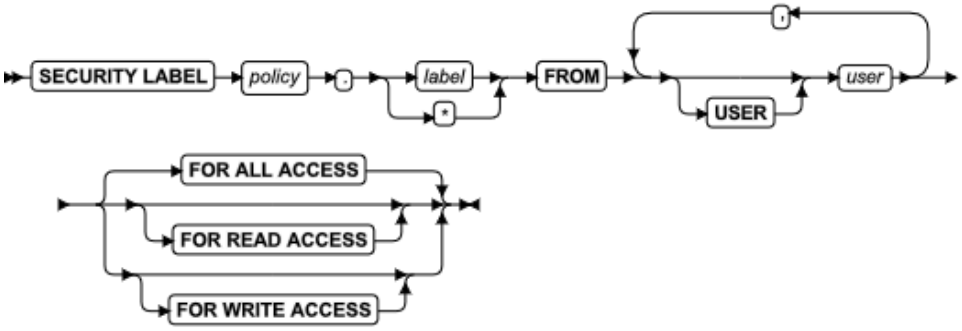
此豁免恢复对所有后续读操作的数组组件的读存取规则，用户 lynette 尝试在受指定的策略的安全标签保护的表进行读操作。

当 REVOKE EXEMPTION 语句成功地取消用户的豁免时，数据库服务器更新系统目录的 **syssecpolicyexemptions** 表来取消注册被取消的豁免（或如果在 FROM 关键字之后罗列几个用户，则为多个豁免）。

## SECURITY LABEL 子句

REVOKE SECURITY LABEL 语句取消由一个或多个用户持有的安全标签（或指定的安全策略的所有安全标签）。

SECURITY LABEL 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 现有的安全标签的名称 | 必须存在指定的安全 *policy* 的标签 | 标识符 |
| *policy* | 此 *label* 的安全策略 | 必须在数据库中已存在 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *user* | 从其取消标签的用户 | 必须为用户的授权标识符 | 所有者名称 |

仅持有 DBSECADM 角色的用户可发出 REVOKE SECURITY LABEL 语句。

安全标签是总与安全策略相关联的数据库对象。那个策略定义构成该安全标签的一系列有效的安全组件。该标签存储该安全策略的每一组件的一个或多个值的集合。

DBSECADM 可将安全标签与下列实体相关联：

- 数据库表的列，***列安全标签***可保护列
- 数据库表的行，***行安全标签***可保护行
- 用户，其***用户安全标签***（以及从已经授予了该用户的安全策略的规则的任何豁免）称为该用户的***安全凭证***。

当持有特定的安全策略的安全标签的用户尝试访问受到同一安全策略的行安全标签保护的行时，数据库服务器将用户安全标签的值的集合与行安全标签的值的集合相比较，以确定该用户是否应被允许访问该数据。类似地，LBAC 考虑用户安全标签与列安全标签，以确定该用户的凭证是否应被允许访问受保护的列。

GRANT SECURITY LABEL 和 REVOKE SECURITY LABEL 语句使得 DBSECADM 能控制用户与标签的关联。(通过仅 DBSECADM 可执行的 CREATE TABLE 或 ALTER TABLE 语句的选项，而不是通过 GRANT SECURITY LABEL 语句，将受保护的表中的数据值与行安全标签或列安全标签相关联。)

紧跟在 LABEL 关键字之后， ***policy.**** 规范中的星号（*）指示数据库服务器来取消 ***policy*** 的每一安全标签。如果您未用星号指定 ***policy.label***，则那个 ***label*** 必须为指定的 ***policy*** 的安全标签的名称。在此情况下，如果该语句成功，则仅从用户列表取消那个安全标签。

跟在 FROM 关键字之后的 USER 关键字是可选的，但在 REVOKE SECURITY LABEL 语句中指定的任何授权标识符都必须是单个用户的标识符，而不是角色的标识符。

## *访问规范*

从其取消安全标签的用户的列表可可选地后跟关键字，以指定对该标签的安全策略保护的数据的访问的类型。

- FOR WRITE ACCESS

  这些关键字将该标签限定到 **IDSLBACRULES** 的写访问规则，即 **IDLSBACWRITEARRAY**、**IDLSBACWRITESET** 和 **IDLSBACWRITETREE**。

- FOR READ ACCESS

  这些关键字将该标签限定到 **IDSLBACRULES** 的读访问规则，即 **IDLSBACWREADARRAY**、**IDLSBACREADSET** 和 **IDLSBACREADTREE**。

- FOR ALL ACCESS

  这些关键字将该标签应用到以上罗列的所有读和写访问规则。如果 REVOKE SECURITY LABEL 不包括 FOR ... ACCESS 规范，则此选项作为缺省值生效。

要获取更多关于这些基于标签的读和写访问的 IDSLBACRULES 规则的信息，请参阅 与安全策略相关的规则。要获取更多关于对这些可为特定的安全策略授予的规则的豁免的信息，请参阅 有关取消豁免的规则。

### *取消用户安全标签的示例*

下列三个语句分别地创建三个名为 level、compartments 和 groups 的安全标签组件：

```
CREATE SECURITY LABEL COMPONENT
    level ARRAY ['TS','S','C','U'];

CREATE SECURITY LABEL COMPONENT
    compartments SET {'A','B','C','D'};

CREATE SECURITY LABEL COMPONENT
    groups TREE ('G1' ROOT,
                 'G2' UNDER ROOT,
                 'G3' UNDER ROOT);
```

下列语句基于上述三个组件创建名为 secPolicy 的安全策略：

```
CREATE SECURITY POLICY secPolicy COMPONENTS
    level, compartments, groups;
```

下列语句创建名为 secLabel1 的安全标签：

```
CREATE SECURITY LABEL secPolicy.secLabel1
    COMPONENT level 'S',
    COMPONENT compartments 'A', 'B',
    COMPONENT groups 'G2';
```

下列语句将此对读访问的安全标签授予用户 sam：

```
GRANT SECURITY LABEL secPolicy.secLabel1
    TO sam FOR READ ACCESS;
```

下列语句从用户 sam 取消对读访问的安全标签。

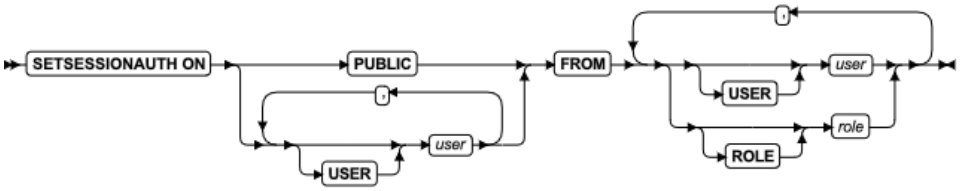
```
REVOKE SECURITY LABEL secPolicy.secLabel1
    FROM sam FOR READ ACCESS;
```

当 REVOKE SECURITY LABEL 语句成功地取消由用户持有了的安全标签时，数据库服务器更新系统目录的 **sysseclabelauth** 表来从那些持有那个安全标签的用户列表移除该用户。

## SETSESSIONAUTH 子句

REVOKE SETSESSIONAUTH 语句从一个或多个用户或角色取消 SETSESSIONAUTH 权限。SETSESSIONAUTH 权限允许还持有 DBA 权限的用户使用 SET SESSION AUTHORIZATION 语句来将会话授权设置为一系列指定的用户之一。

SETSESSIONAUTH 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *role* | 要从其取消权限的角色 | 必须为角色的授权标识符 | 所有者名称 |
| *user* | 在 FROM 关键字之后，是要从其取消权限的用户。在 ON 关键字之后，是被授权者可在 SET AUTHORIZATION 语句中指定其身份的用户。 | 必须为用户的授权标识符 | 所有者名称 |

仅持有 DBSECADM 角色的用户可取消 SETSESSIONAUTH 权限。

跟在 ON 关键字之后的用户或 PUBLIC 规范，指定在使用 SET SESSION AUTHORIZATION 语句时， SETSESSIONAUTH 权限的被授权者不再能够使用谁的身份。这可为用户或 PUBLIC，但不可为角色。如果指定 PUBLIC，则该权限的被授权者不再有能力使用任意数据库用户的身份。

可跟在 FROM 关键字之后的 USER 和 ROLE 关键字是可选的。***user*** 或 ***role*** 都不可为发出 REVOKE SETSESSIONAUTH 语句的 DBSECADM 角色的持有者。FROM 子句不可指定 PUBLIC。

下例授予用户 sam 将会话授权设置为用户 lynette 和 manor 的能力：

```
REVOKE SETSESSIONAUTH ON lynette, manoj TO sam;
```

下一示例从用户 lynette 取消将会话授权设置为 PUBLIC 的能力：

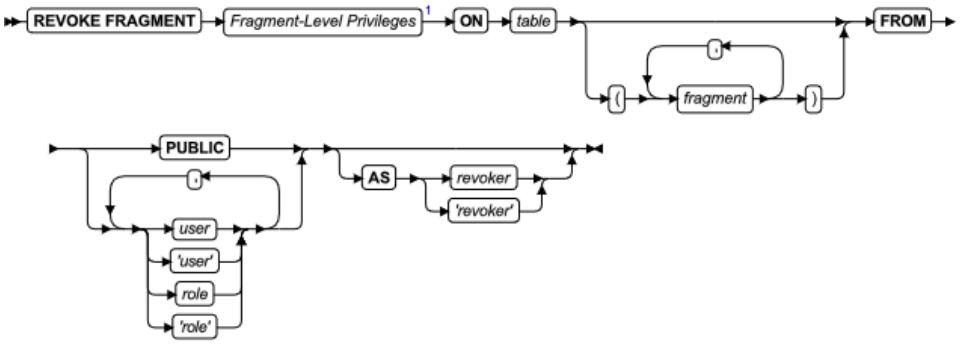
```
REVOKE SETSESSIONAUTH ON PUBLIC FROM lynette;
```

此语句取消的权限的 PUBLIC 作用域已使得用户 lynette 能使用她在 SET SESSION AUTHORIZATION 语句中指定的任何用户的访问权限和安全凭证。

# 2.119 REVOKE FRAGMENT 语句

使用 REVOKE FRAGMENT 语句来从一个或多个用户或角色取消 Insert、Update 或 Delete 分片级权限，这些权限是对那些已通过表达式分片了的表的单个分片授予的。此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *fragment* | 分片或存储一分片的 dbspace 的名称。缺省为 *table* 的所有分片。 | 必须存在且必须存储表的分片 | 标识符 |
| *revoker* | 为要取消的权限的授予者得用户（其未正在执行此语句） | 必须为分片级权限的授予者 | 所有者名称 |
| *role* | 要从其取消权限的角色 | 必须在数据库中存在 | 所有者名称 |
| *table* | 要取消其分片级权限的分片了的表 | 必须存在且必须通过表达式分片 | 数据库对象名 |
| *user* | 要从其取消权限的用户 | 必须为有效的授权标识符 | 所有者名称 |

**用法**

REVOKE FRAGMENT 语句是对表分片指定权限的 REVOKE 语句的特例。使用 REVOKE FRAGMENT 语句来从一个或多个用户或角色取消 Insert、Update 或 Delete 权限。DBA 可使用此语句来取消其所有者为另一用户的分片的权限。

REVOKE FRAGMENT 语句仅对通过基于表达式的分布式方案分片的表为有效的。要获取对此分片策略的解释，请参阅 表达式分布方案。

## 指定分片

如果您未指定 ***fragment***，则取消对 ***table*** 的所有分片的权限。您可紧跟在 ON ***table*** 规范之后指定一个分片或括在括号之间以逗号分隔的分片列表。

必须通过其名称引用每一 ***fragment***。当您创建该分片时，如果您未声明显式的标识符，则其名称缺省为其驻留在其中的 dbspace 的名称。

在使用 gspaces 实用程序成功地重命名 dbspace 之后，仅新的名称是有效的。 GBase 8s 自动地更新在系统目录中的现有的分片策略，来替换新的 dbspace 名称，但您必须在 REVOKE FRAGMENT 语句中指定新的名称来引用其缺省的名称为重命名了的 dbspace 的名称的分片。

## FROM 子句

您可指定 PUBLIC 关键字来从 PUBLIC 取消指定的分片级权限，从而从已经被显式地授予了该权限的所有用户或不持有他们已通过其收到了该权限的角色取消该权限。

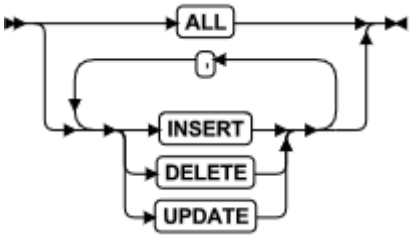
如果您以引号括起 ***user*** 或 ***role***，则该名称区分大小写，且完全按输入的样子存储。在符合 ANSI 的数据库中，如果您不使用引号括起 ***user*** 或括起 ***role***，则以大写字母缺省地存储该名称，虽然您可在 ***owner*** 规范中将 ANSIOWNER 环境变量设置为保留小写字符字母。

当您在 REVOKE FRAGMENT 的 FROM 子句中包括 ***role*** 时，从那个角色取消指定的分片权限。然而，有那个角色的用户保留单独地授予了他们或 PUBLIC 而持有的任何分片权限。

## 分片级权限

跟在 FRAGMENT 关键字之后的一个或多个关键字指定***分片级权限***，这是表级权限的一个逻辑子集：

分片级权限




您可单个地或成批地取消分片级权限。下列关键字指定您可取消的分片级权限。

| 关键字 | 作用 |
| --- | --- |
| INSERT | 防止用户在分片中插入行 |
| DELETE | 防止用户在分片中删除行 |
| UPDATE | 防止用户在分片中更新行 |
| ALL | 对分片取消 Insert、Delete 和 Update 权限 |

如果您在 REVOKE FRAGMENT 语句中指定 ALL 关键字，则指定的用户和角色失去他们当前对指定的分片拥有的所有分片级权限。例如，假设用户当前对表的一个分片有 Update 权限。如果您使用 ALL 关键字来从此用户取消对此分片的所有当前权限，则该用户失去他或她对此分片拥有的 Update 权限。

要了解分片级权限与表级权限之间的区别，请参阅 分片级授权的定义 和 分片级授权在语句验证中的作用 部分。

## AS 子句

如果没有 AS 子句，则执行 REVOKE 语句的用户必须是正在被取消的权限的授予者。DBA 或该分片的所有者可使用 AS 子句来指定另一用户（其必须为该权限的授予者）作为对分片的权限的取消者。

AS 子句提供了唯一可通过其对分片取消权限的机制，其 ***owner*** 为操作系统不可知的有效用户账号。

## REVOKE FRAGMENT 语句的示例

下列示例基于 **customer** 表。它们都假设通过表达式将 **customer** 表分片成三个名为 **part1**、**part2** 和 **part3** 的分片。

### 取消对一个分片的权限

下列语句从用户 ed 取消对 part1 中的 customer 表的分片的 Update 权限：

```
REVOKE FRAGMENT UPDATE ON customer (part1) FROM ed;
```

下列语句从用户 susan 取消对 part1 中的 customer 表的分片的 Update 和 Insert 权限：

```
REVOKE FRAGMENT UPDATE, INSERT ON customer (part1) FROM susan;
```

下列语句取消对 part1 中的 customer 表的分片当前授予了用户 harry 的所有权限:

```
REVOKE FRAGMENT ALL ON customer (part1) FROM harry;
```

### 取消对多于一个分片的权限

下列语句取消对 part1 和 part2 中的 customer 表的分片当前授予用户 millie 的所有权限：

```
REVOKE FRAGMENT ALL ON customer (part1, part2) FROM millie;
```

### 从多于一个用户取消权限

下列语句取消对 part3 中的 customer 表的分片当前授予用户 jerome 和 hilda 的所有权限：

```
REVOKE FRAGMENT ALL ON customer (part3) FROM jerome, hilda;
```

### 不指定分片而取消权限

下列语句从用户 mel 取消对此用户当前对其有权限的所有分片的所有当前的权限：

```
REVOKE FRAGMENT ALL ON customer FROM mel;
```

**相关的语句**

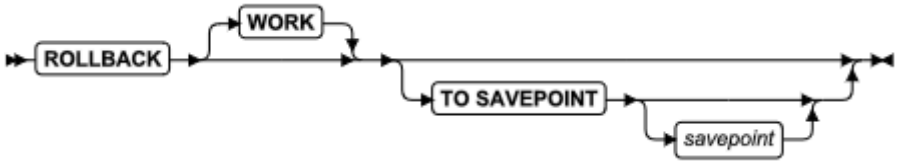
相关的语句：GRANT FRAGMENT 语句 和 REVOKE 语句

要了解对分片级和表级权限的讨论，请参阅 分片级权限 部分。

# 2.120 ROLLBACK WORK 语句

使用 ROLLBACK WORK 语句来有意地取消当前事务的全部或部分，撤销从该事务开始以来发生的任何更改，或在该 ROLLBACK WORK 语句与指定的或缺省的保存点之间发生的任何更改。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *savepoint* | 定界回滚的范围的保存点的名称 | 必须在当前的事务中存在。 | 标识符 |

**用法**

ROLLBACK WORK 语句仅在支持事务日志记录的数据库中是有效的。仅可回滚日志记录了的操作。仅在多语句操作的结尾使用 ROLLBACK WORK。

ROLLBACK WORK 语句将数据库恢复到在该事务开始的被取消的部分之前已存在的状态。

在不符合 ANSI 的数据库中，BEGIN WORK 语句启动事务。您可以 COMMIT WORK 语句终止该事务，或以 ROLLBACK WORK 语句取消该事务的全部或部分。在不符合 ANSI 的数据库中，当没有事务正在挂起时，如果您发出 ROLLBACK WORK 语句，则 GBase 8s 发出错误。

在符合 ANSI 的数据库中，多语句事务是隐式的。您不需要以 BEGIN WORK 语句标记事务的起始。您仅需要以 COMMIT WORK 语句标记每一事务的结尾，或以 ROLLBACK WORK 语句取消该事务。当没有事务正在挂起时，如果您发出 ROLLBACK WORK 语句，则接受该语句但不起作用。

ROLLBACK WORK 语句将数据库恢复到在该事务开始的被取消的部分之前存在的状态。除非您包括 TO SAVEPOINT 关键字，不然，ROLLBACK WORK 取消整个事务。

ROLLBACK WORK 语句释放取消了的事务持有的所有行和表锁。

在 GBase 8s ESQL/C 和 SPL 中，ROLLBACK WORK 语句关闭所有打开的游标，除了那些通过包括 WITH HOLD 关键字被声明作为***保持游标***的游标。在提交或回滚事务之后，保持游标保持打开。

如果您在 WHENEVER 语句调用的 SPL 例程内部使用 ROLLBACK WORK 语句，则请在 ROLLBACK WORK 语句之前指定 WHENEVER SQLERROR CONTINUE 和 WHENEVER SQLWARNING CONTINUE。如果 ROLLBACK WORK 语句遇到错误或警告，则此步骤防止程序发生循环。

如果程序异常地终止，则隐式地回滚当前的事务。

## WORK 关键字

WORK 关键字在 ROLLBACK WORK 语句中是可选的。下列两个语句是等价的：

```
ROLLBACK;

ROLLBACK WORK;
```

## TO SAVEPOINT 子句

可选的 TO SAVEPOINT 子句指定部分回滚。此子句可将回滚的范围限定到在 ROLLBACK 语句与指定的或缺省的保存点之间的当前保存点级别的操作。如果在 SAVEPOINT 关键字之后未指定 ***savepoint***，则回滚终止在当前保存点级别之内的最近设置保存点。

当 ROLLBACK WORK TO SAVEPOINT 语句执行成功时，在保存点之前的 DDL 和 DML 语句的任何影响依然保持，但取消通过跟在保存点之后的语句对数据库的模式的更改或对其数据值的更改。这些取消了的语句所需要的任何锁依然保持，但在事务末尾被释放。销毁在指定的保存点与 ROLLBACK 语句之间的任何保存点，但通过 ROLLBACK 语句引用的保存点（以及在被引用的保存点之前的任何保存点）继续存在。程序控制传递到紧跟在 ROLLBACK 语句之后的语句。

如果省略 TO SAVEPOINT 子句，则 ROLLBACK 语句回滚整个事务，以及被释放的事务之内的所有保存点。

如果在当前的事务中指定的 ***savepoint*** 不存在，则数据库服务器发出例外。

在紧跟在 TRUNCATE 语句之后的 ROLLBACK 语句中，TO SAVEPOINT 子句是无效的。在此情况下，尝试进行的部分回滚失败并报错。要取消 TRUNCATE 语句已经对表造成的未提交的更改，请发出 ROLLBACK WORK 作为下一语句，但不带 TO SAVEPOINT 子句。

下列程序片断将当前的事务会滚到名为 pt109 的保存点：

```
BEGIN WORK;
DROP TABLE tab03;
CREATE TABLE tab03 (col1 CHAR(24), col2 DATE);
SAVEPOINT pt108;
...
INSERT INTO   tab03 VALUES ('First day of autumn', '09/23/2012');
SAVEPOINT pt109;
...
DELETE FROM tab03 WHERE col2 < '12/09/2009';
SAVEPOINT pt110;
...
ROLLBACK TO SAVEPOINT pt109;
```

在此示例中的 ROLLBACK 语句有这些作用：

- 取消删除 **col2** 日期值早于 2009 年 12 月 9 日的任何行的 DML 操作。
- 释放保存点 **pt110**，以及在 **pt109** 与 ROLLBACK 语句之间的任何其他保存点。
- 取消当前事务之内按照 SQL 语句的词典顺序跟在保存点 **pt109** 之后的操作对数据库的任何其他更改。

然而，不释放保存点 pt108，因为在该事务中它被设置早于 pt109。未被此部分回滚取消的是，在设置了 pt109 保存点之前该事务的任何未提交的 DDL 或 DML 操作的影响，包括表 tab03 的创建以及向那个表添加行的 INSERT 操作。这些会在部分回滚之后依然保持，将另一部分回滚的可能性挂起到保存点，以及整个事务的最终提交或回滚。

**相关的语句**

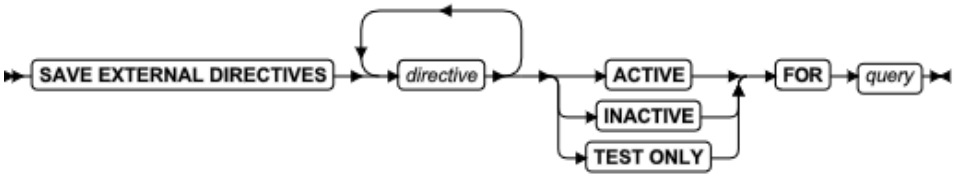
相关的语句：BEGIN WORK 语句、COMMIT WORK 语句、RELEASE SAVEPOINT 语句 和 SAVEPOINT 语句。

要了解对事务和 ROLLBACK WORK 的讨论，请参阅 *GBase 8s SQL 教程指南*。

## 2.121 SAVE EXTERNAL DIRECTIVES 语句

使用 SAVE EXTERNAL DIRECTIVES 语句来创建指定的查询的外部优化程序伪指令，并将伪指令保存在数据库中。这些伪指令自动地应用于同一查询的后续实例。此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *directive* | 对 *query* 有效的优化程序伪指令 | 必须对查询有效且通过注释指示符定界 | 优化程序伪指令 |
| *query* | 有效的 SELECT 语句的文本 | NULL 字符串无效 | SELECT 语句 |

**用法**

SAVE EXTERNAL DIRECTIVES 将一个或多个优化程序伪指令与查询关联，并在 **sysdirectives** 系统目录表中存储此关联的一条记录，以便随后用于与指定的查询字符串相匹配的查询。此语句在优化程序伪指令的列表与查询的文本之间建立关联，但不执行指定的查询。

如果 SAVE EXTERNAL DIRECTIVES 语句指定多于一个优化程序伪指令，则使用在连续的伪指令之间使用空格字符（ASCII 32）作为分隔符，如下例所示：

```
SAVE EXTERNAL DIRECTIVES /*+ USE_INDEX */ /*+ ORDERED */ ACTIVE FOR
    SELECT * FROM systables;
```

与在查询中不同，在 SAVE EXTERNAL DIRECTIVES 语句的伪指令列表中，逗号（,）不是有效的分隔符。

仅 DBA 或用户 **gbasedbt** 可执行 SAVE EXTERNAL DIRECTIVES。保存在数据库中的优化程序伪指令称为*外部伪指令*。

## 外部优化程序伪指令

SAVE EXTERNAL DIRECTIVES 语句将其与查询的文本相关联的外部伪指令在一些查询中可提高性能，该查询优化程序的缺省的行为不令人满意。

外部优化程序伪指令类似于嵌入在查询内的 inline 优化程序伪指令。然而，与 inline 伪指令不同，无需修改或重新编译现有的应用程序即可应用外部伪指令。

## 为会话启用或禁用外部伪指令

在会话期间，如果在配置文件中 EXT_DIRECTIVES 参数设置为 0，或 SET ENVIRONMENT 语句中的 EXTDIRECTIVES 关键字设置为 0、OFF 或 off，则 GBase 8s 忽略外部伪指令。

此外，当 IFX_EXTDIRECTIVES 环境变量设置为 0 时，客户端系统可为它的当前会话禁用外部伪指令。

下表展示对于在客户端系统上的有效的 IFX_EXTDIRECTIVES 设置的各种组合，以及在 GBase 8s 上，有效的 EXT_DIRECTIVES 配置参数设置的各种组合，是否禁用（OFF）或启用（ON）外部伪指令：

表 1. IFX_DIRECTIVES 设置与 EXT_DIRECTIVES 配置参数设置的组合

| **在客户端系统上的 IFX_EXTDIRECTIVES 设置** | **EXT_DIRECTIVES = 0** | **EXT_DIRECTIVES = 1** | **EXT_DIRECTIVES = 2** |
|---|---|---|---|
| IFX_EXTDIRECTIVES 未设置 | OFF | OFF | ON |
| IFX_EXTDIRECTIVES = 1 | OFF | ON | ON |
| IFX_EXTDIRECTIVES = 0 | OFF | OFF | OFF |

当初始化数据库服务器时，如果 EXT_DIRECTIVES 设置为 1 或 2，则服务器启用外部伪指令。单个的会话可通过设置 **IFX_EXTDIRECTIVES** 启用或禁用外部伪指令，如表所示。1 或 2 之外的任何设置都解释为零，禁用此特性。

当启用外部伪指令时，通过 ACTIVE、INACTIVE 或 TEST ONLY 关键字指定单个的外部伪指令的状态。（但仅在其上伪指令为有效的查询可从外部伪指令获益。）

您还可使用 SET ENVIRONMENT 语句的 EXTDIRECTIVES 选项来启用或禁用会话期间的外部伪指令。您使用 EXTDIRECTIVES 选项指定的内容重写在 ONCONFIG 文件中的 EXT_DIRECTIVES 配置参数中指定的外部伪指令值。

为了启用或禁用在 ONCONFIG 文件中的值，且：

- 要启用会话期间的外部伪指令，则指定 1、on 或 ON 作为 SET ENVIRONMENT EXTDIRECTIVES 的值。
- 要禁用会话期间的外部伪指令，请指定 0、off 或 OFF 作为 SET ENVIRONMENT EXTDIRECTIVES 的值。

在会话期间，要启用在 EXT_DIRECTIVES 配置参数中和在客户端侧 IFX_EXTDIRECTIVES 环境变量中的缺省的值，请指定 DEFAULT 作为 SET ENVIRONMENT 语句的 EXTDIRECTIVES 选项的值。

要获取更多关于使用 SET ENVIRONMENT 语句的 EXTDIRECTIVES 选项的信息，请参阅 SET ENVIRONMENT 语句。

## 伪指令规范

SAVE EXTERNAL DIRECTIVES 语句中的每一 *directive* 规范必须遵循“用户程序伪指令”段的语法，如 优化程序伪指令 中所描述的那样，除了如果您指定多于一个伪指令之外，您必须通过空格字符在伪指令列表中分隔它们，而不是通过逗号（,），如下例所示：

```
SAVE EXTERNAL DIRECTIVES /*+ AVOID_INDEX (table1 index1)*/ /*+ FULL(table1) */
        ACTIVE FOR
        SELECT /*+ INDEX( table1 index1 ) */   col1, col2
        FROM table1, table2 WHERE table1.col1 = table2.col1;
```

此示例将 AVOID_INDEX 和 FULL 伪指令与指定的查询相关联。当外部伪指令应用于与该 SELECT 语句相匹配的查询时，查询优化程序忽略 inline INDEX 伪指令。

## ACTIVE、INACTIVE 和 TEST ONLY 关键字

您必须包括 ACTIVE、INACTIVE 或 TEST ONLY 关键字选项之一来启用、禁用或限制外部伪指令的范围：

- 如果启用外部伪指令，ACTIVE 关键字将伪指令的列表应用于任何与 ***query*** 字符串相匹配的任何后续的查询。
- INACTIVE 关键字导致 GBase 8s 忽略伪指令。（它与 **sysdirectives** 中的查询相关联，但它是休眠的，没有作用。）
- 如果启用外部伪指令，则 TEST ONLY 关键字仅将该伪指令应用于 DBA 或 **gbasedbt** 执行的匹配的查询。由任何用户执行的查询都不可使用 TEST ONLY 外部伪指令。

直到 DBA 或用户 **gbasedbt** 为了那个伪指令，将 **sysdirectives.active** 系统目录表值由 0（INACTIVE）更改为 1（ACTIVE）或 2（TEST ONLY），INACTIVE 伪指令才起作用。外部伪指令没有 SQL 标识符，但 DBA 可引用 UPDATE 语句中的 **sysdirectives.id** 列来指定要更新哪个外部伪指令。

或者，DBA 或用户 **gbasedbt** 可从 **sysdirectives** 删除 INACTIVE 或 TEST ONLY 行，并使用 SET EXTERNAL DIRECTIVES 语句来重新定义被删除的伪指令，不过现在指定 ACTIVE 关键字。这可给予其他用户访问 DBA 已验证的 TEST ONLY 伪指令。

## 查询规范

在 SAVE EXTERNAL DIRECTIVES 中跟在 FOR 关键字之后的 ***query*** 规范必须指定有效的 SELECT 语句的语法，如 SELECT 语句 中描述的那样。如果 ***query*** 文本还包括任何 inline 优化程序伪指令，则当将外部伪指令应用到该查询时，忽略该 inline 伪指令。

当启用外部伪指令且 **sysdirectives** 系统目录表不为空时，数据库服务器将每一查询与每个 ACTIVE 外部伪指令的 ***query*** 文本相比较，对于由 DBA 或用户 **gbasedbt** 执行的查询，与每个 TEST ONLY 外部伪指令相比较。如果已将外部伪指令应用于查询了，则从 SET EXPLAIN 语句的输出指示那个查询 "EXTERNAL DIRECTIVES IN EFFECT"。

外部伪指令的目的是提高那些与 ***query*** 字符串相匹配的查询的性能，但使用这些伪指令可潜在地降低其他查询的速度，如果查询优化程序必须将大量的活动的外部伪指令的 query 字符串与每个 SELECT 语句的文本相比较的话。为此， GBase 推荐 DBA 不允许 sysdirectives 表累计超过一定数量的 ACTIVE 行。（避免对其他查询产生无意的性能影响的另一方法是禁用此特性。）

如果多于一个 SET EXTERNAL DIRECTIVES 语句将活动的外部伪指令与同一查询相关联，则结果不可预料，因为优化程序使用其 ***query*** 字符串相匹配的查询的第一 sysdirectives 行。

### 相关的语句

要获取关于优化程序伪指令及其语法的信息，请参阅 优化程序伪指令 中的“优化程序伪指令”段。

要获取关于 **sysdirectives** 表和 **IFX_EXTDIRECTIVES** 环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

# 2.122 SAVEPOINT 语句

使用 SAVEPOINT 语句来声明在当前的 SQL 事务之内新的保存点的名称，并设置在该事务之内 SQL 语句的词典顺序之内的新保存点的位置。SAVEPOINT 语句符合 SQL 的 ANSI/ISO 标准。

### 语法


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *savepoint* | 在此为新的保存点声明的名称 | 不可为在相同的保存点级别中现有的唯一保存点的名称 | 标识符 |

### 用法

您可在 SQL 事务中使用 SAVEPOINT 语句以 DB-Access 和 SPL、C 和 Java™ 例程来支持错误处理。您可定义保存点来将单个复杂的事务分隔成它的组件 SQL 语句的较小的逻辑子集。在那个事务之内，可更有效地回滚跟在每一保存点之后的语句的子集，比起如果您在多个事务中已使用了多个 COMMIT WORK 和 ROLLBACK WORK 语句的话。

SAVEPOINT 语句在当前的事务之内按照语句的词典顺序在当前的位置设置指定的保存点。在 SAVEPOINT 语句执行成功之后，引用此保存点的后续的 ROLLBACK TO SAVEPOINT 语句可取消对数据库任何未提交的更改，这些更改是跟在新的保存点之后但在 ROLLBACK TO SAVEPOINT 语句之前的。

如果在同一事务内的现有的保存点与 SAVEPOINT 语句指定的名称相同，则销毁现有的保存点，除非下列条件之一为真：

- 在不同的保存点级别设置了现有的保存点。
- 以 UNIQUE 关键字选项声明了现有的保存点名称。在此情况下 SAVEPOINT 语句失败并报错，除非在不同的保存点级别设置了现有的 UNIQUE 保存点。

销毁一保存点来为另一保存点重用它的名称，与释放该保存点不同。重用保存点名称仅销毁一个保存点。以 RELEASE SAVEPOINT 语句释放保存点会释放指定的保存点以及后续已设置了的所有保存点。

**UNIQUE 选项**

此可选的关键字指定应用程序不要打算在另一 SAVEPOINT 语句中重用此保存点的名称，在此保存点在当前的保存点级别之内是活动的时候。

如果保存点已存在，是在当前的保存点界别之内以相同的名称和以 UNIQUE 关键字设置了的，则 SAVEPOINT 语句失败并报错，且不销毁现有的保存点。

**保存点级别**

GBase 8s 支持构造嵌套的保存点级别。单个 SQL 事务可有多个保存点级别。在执行 SPL 例程或外部 UDR 期间，自动地创建新的保存点级别。递归地调用相同的 SPL 例程或 UDR 还增长当前事务的保存点级别。

当在其被创建的 UDR 中完成执行时，保存点级别终止。当保存点级别终止时，自动地释放在它之内的所有保存点。父保存点级别继承任何 DDL 或 DML 修改（即，到在其内创建了刚刚终止的那个保存点级别），并受任何针对该父保存点级别发出的保存点相关的语句支配。

下列规则适用于保存点级别之内的活动：

- 仅可在保存点被创建的那个保存点级别之内引用保存点。您不可释放、销毁或回滚到在当前保存点级别之外创建的保存点。

保存点名称的唯一性仅在当前的保存点级别之内是强制的。在其他保存点级别中为活动的保存点的名称可在当前的保存点级别中重用，而不影响其他保存点级别中的那些保存点。

**在分布式 SQL 事务中的保存点**

如果所有参与的数据库支持事务日志记录，则保存点在支持事务的单个 GBase 8s 实例的跨数据库分布式 SQL 事务中是有效的。在跨数据库 SQL 事务中还支持保存点，包括在高可用性集群中的操作，如果所有参与的 GBase 8s 实例支持保存点，且在该事务中访问的所有数据库都使用日志记录的话。

然而，如果在跨数据库事务中的任何参与的数据库服务器不支持保存点，且在可支持保存点的协调者与不支持保存点的从属服务器之间建立了连接，则在分布式会话之内的任何 ROLLBACK TO SAVEPOINT 语句都失败并报错。

**保存点的保持**

保存点是 SQL 事务之内，而不是数据库对象之内的位置标记。在同一事务之内，任何下列事件之一都销毁现有的保存点 **S**：

- 执行 COMMIT WORK 或 ROLLBACK WORK（无 TO SAVEPOINT 子句）语句。
- 执行在同一保存点级别中指定 **S** 的 RELEASE SAVEPOINT 语句。
- 执行 ROLLBACK TO SAVEPOINT 或 RELEASE SAVEPOINT 语句，指定早于在同一保存点级别中的 **S** 建立了的保存点。
- 在同一保存级别中 SAVEPOINT 语句指定相同的名称作为 **S**，且 **S** 不是以 UNIQUE 关键字创建的。

对保存点的限制

在下列上下文中不支持保存点和保存点级别：

- 在不支持事务日志记录的数据库中
- 在触发器活动中
- 在 XA 全局事务中
- 在启用 AUTOCOMMIT 连接属性的应用程序或 UDR 中。

此外，在 DML 语句之内调用的 UDR 中，SAVEPOINT 语句（如 RELEASE SAVEPOINT 和 ROLLBACK WORK TO SAVEPOINT 语句）是无效的，如下例所示：

```
SELECT first_1 foo() FROM systables;
```

在此，**foo( )** 例程不可设置保存点。

**相关的语句**

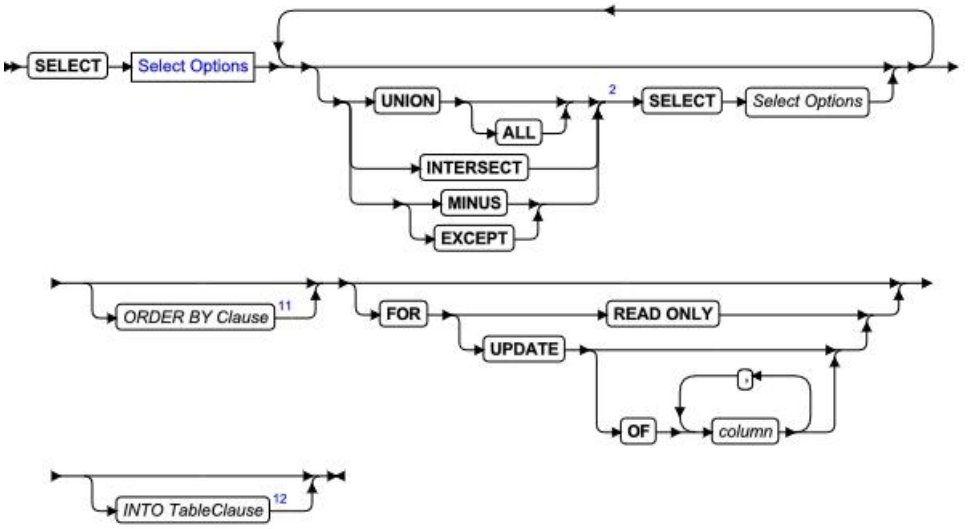
相关的语句：COMMIT WORK 语句、RELEASE SAVEPOINT 语句 和 ROLLBACK WORK 语句

## 2.123 SELECT 语句

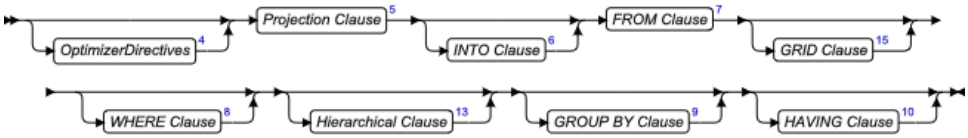
使用 SELECT 语句来从数据库或从 SPL 或 GBase 8s ESQL/C 集合变量检索值。SELECT 操作称为***查询***。

满足该查询的特定的查询条件的行或值称为*符合条件的*行或值。在应用任何附加的逻辑条件之后，该查询检索到的其调用上下文称为该查询的*结果集*。此结果集可为空。

**语法**




Select 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 在 FETCH 之后可被更新的列的名称 | 必须在 FROM 子句表中，但不需要在 Projection 子句的选择列表中 | 标识符 |

**用法**

SELECT 语句可从当前数据库中，或当前数据库服务器的另一数据库中，或另一数据库服务器的数据库中的表返回数据。 仅 SELECT 关键字、Projection 子句和 FROM 子句是必需的规范。

对于包括 CONNECT BY 子句的层级查询，FROM 子句仅可指定单个表，该表必须驻留在连接到当前会话的 GBase 8s 数据库服务器实例的本地数据库中。

对于包括 GRID 子句的查询，在 GRID 子句指定的每个节点上，FROM 子句指定的每一表的实例必须有相同的模式、相同的数据库语言环境和相同的代码集。

SELECT 语句只能引用 CREATE EXTERNAL TABLE 语句已指定的一个外部表。在复合查询中，仅可在最外部的查询中指定此外部表。您不可在子查询中引用外部表。

您需要对该数据库的 Connect 访问权限来执行查询，以及对该查询要从其检索行的表对象的 Select 权限。

SELECT 语句可包括各种基本的子句，标识在下列列表中。

| 子句 | 作用 |
|---|---|
| 优化程序伪指令 | 指定该查询应如何实现 |

| 子句 | 作用 |
| --- | --- |
| Projection 子句 | 指定要从数据库读的项的列表 |
| INTO 子句 | 指定接收结果集的变量 |
| FROM 子句 | 指定 Projection 子句项的数据源 |
| 表或视图的别名 | 查询中表或列的临时名称 |
| 表表达式 | 定义派生的表作为查询数据源 |
| 横向的派生的表 | 定义查询中相关联的表引用 |
| ONLY 关键字 | 排除孩子表作为类型表的查询中的数据源 |
| 迭代器函数 | 反复地返回值作为数据源的函数 |
| 符合 ANSI 的连接 | 符合 ISO/ANSI 语法标准的连接查询 |
| GBase 8s 扩展外连接 | 基于隐式的 LEFT OUTER 连接的查询语法 |
| GRID 子句 | 指定存储网格查询的表的节点 |
| 使用 ON 子句 | 指定连接条件作为连接前过滤器 |
| SELECT 的 WHERE 子句 | 对符合条件的行和连接后过滤器设置条件 |
| 层级查询子句 | 为层级数据的查询设置条件 |
| GROUP BY 子句 | 将行的组组合到汇总结果内 |
| HAVING 子句 | 对汇总结果设置条件 |
| ORDER BY 子句 | 根据列值对符合条件的行排序 |
| ORDER SIBLINGS BY 子句 | 在每个级别为兄弟层级数据排序 |
| FOR UPDATE 子句 | 在 FETCH 之后启用结果集的更新 |
| FOR REArD ONLY 子句 | 在 FETCH 之后禁用结果集的更新 |
| INTO TEMP 子句 | 将结果集放到临时表内 |
| INTO EXTERNAL 子句 | 在外部表中存储查询结果集 |
| INTO STANDARD 和 INTO RAW 子句 | 在永久数据库表中存储查询结果集 |

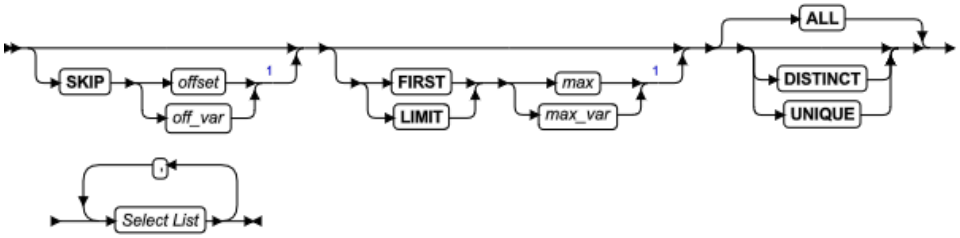
| 子句 | 作用 |
| --- | --- |
| UNION ALL 运算符 | 组合两个 SELECT 语句的结果集 |
| UNION 运算符 | 与 UNION ALL 相同，但废弃重复的行 |
| INTERSECT 运算符 | 从两个查询结果集返回不同的公共行 |
| MINUS 运算符 | 仅返回两个查询返回的第一个不同的行。 |

接下来的部分描述这些和 SELECT 语句的其他子句。

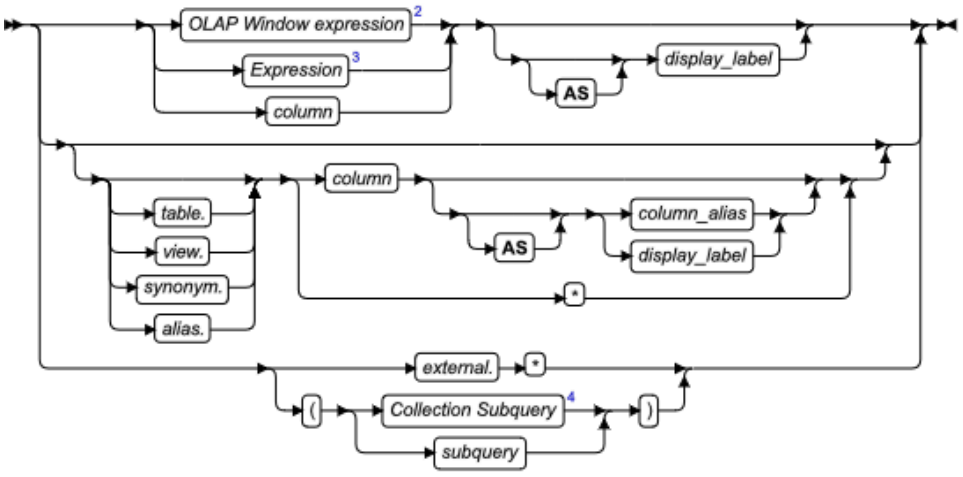
## Projection 子句

Projection 子句（有时称为 ***Select 子句***）指定要检索的数据库对象或表达式的列表，并可设置对符合条件的行的限制。（***select 列表***有时也称为 ***projection 列表***。）

Projection 子句




Select 列表




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *alias* | 临时*表*或*视图*名称。请参阅 FROM 子句。 | 仅当 FROM 子句为 *table* 或 *view* 声明 *alias* 时才有效 | 标识符 |
| *column_alias* | 您在此为 *column* 声明的临时标识符 | 在此查询中的 *columns* 和 *column_alias* 名称 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 之中必须是唯一的。仅 GROUP BY 子句可引用 *column_alias*。 | |
| *column* | 从其检索数据的列 | 在 FROM 子句引用的数据源中必须存在 | 标识符 |
| *display_label* | 在此为 *column* 或为表达式声明的临时名称 | 请参阅 声明显示标签 | 标识符 |
| *external* | 从其检索数据的外部表 | 必须存在 | 数据库对象名 |
| *max* | 指定要返回的行的最大数目的整数（> 0） | 如果 *max* > 符合条件的行的数目，则返回所有相匹配的行 | 精确数值 |
| *max_var* | 存储 *max* 的值的主变量或本地 SPL 变量 | 与 *max* 相同；在准备好的对象和 SPL 例程中有效 | 依赖于语言 |
| *offset* | 指定在结果集的第一行之前要排除多少符合条件的行的整数（> 0） | 不可为负数。如果 *offset* >（符合条件的行的数目），则不返回行 | 精确数值 |
| *off_var* | 存储偏移量的值的主变量或本地 SPL 变量 | 与 *offset* 相同；在准备好的对象和在用户定义的例程中有效 | 依赖于语言 |
| *subquery* | 嵌入的查询 | 在 Projection 子句之内的子查询不可包括 SKIP、FIRST、INTO TEMP 或 ORDER BY 子句。 | SELECT 语句 |
| *table, view, synonym* | 要从其检索数据的表、视图或同义词的名称 | 同义词以及它执行的表或视图必须存在 | 数据库对象名 |

星号（*）指定按照其定义的顺序在*表*或*视图*中的所有列。要以另一顺序存取所有列或列的子集，您必须显式地指定单独的 ***column*** 名称。如果 FROM 子句仅指定单个数据源，则单个的星号（*）可为有效的 Projection 子句。

SKIP、FIRST、LIMIT、MIDDLE、DISTINCT 和 UNIQUE 规范可将结果限定到符合条件的行的子集，如以下部分所解释。

# 符合条件的行的顺序

要执行查询，数据库服务器构建查询计划并检索与 WHERE 子句条件相匹配的所有符合条件的行。（此处，*行*指的是值的一个集合，如在 select 列表中指定的那样，来自 FROM 子句指定的表或连接的表的单个记录。）如果该查询没有 ORDER BY 子句，则符合条件的行按照它们的检索的顺序排列，每一执行可能都不一样；否则，它们的排列遵循 ORDER BY 规范，如 ORDER BY 子句 中描述的那样。

如果 Projection 子句包括任何下列选项，则查询是否指定 ORDER BY 可影响到哪些行在结果集中：

- FIRST 选项
- SKIP 和 LIMIT 选项

# 使用 SKIP 选项

SKIP *offset* 选项指定要排除多少符合条件的行，对于 *offset* SERIAL8 范围内的一个整数，从符合条件的第一行计数。下列示例从除了前 10 行之外的所有行检索值：

SELECT SKIP 10 a, b FROM tab1;

您还可使用主变量来指定要排除多少行。在 SPL 例程中，您可使用输入参数或本地变量来提供此值。

当您以 ORDER BY 子句在查询中使用 SKIP 选项时，可排除前 *offset* 行，根据 ORDER BY 条件这些行有最低的值。如果 ORDER BY 子句包括 DESC 关键字，则您还可使用 SKIP 来排除带有最高值的行。例如，下列查询返回 orders 表的所有行，除了最旧的 50 个订单之外：

SELECT SKIP 50 * FROM orders ORDER BY order_date;

在此，如果在 **orders** 表中只有不到 50 行，则结果集为空。***offset*** = 0 不是无效，但在那种情况下，SKIP 选项无作用。

您还可使用 SKIP 选项来限制准备好了的 SELECT 语句的、UNION 查询的结果集，在其结果集定义集合派生的表的查询中，以及在触发器的事件或活动中。

您可以一起使用 SKIP 与 FIRST 选项来指定在结果集中的哪些以及多少符合条件的行，如在 使用带有 FrIRST 选项的 SKIP 选项 部分中的示例展示的那样。

SKIP 在下列上下文中无效：

- 在视图的定义中
- 在嵌套的 SELECT 语句中
- 在子查询中。

# 使用 FIRST 选项

FIRST *max* 选项指定结果集包括不多于 *max* 行（或正好 *max*，如果 *max* 不大于符合条件的行的数目的话）。 不返回满足选择条件的任何附加的行。下例最多从表 tab1 检索 10 行：

SELECT FIRST 10 a, b FROM tab1;

GBase 8s 可使用主变量或在本地变量中指定 ***max*** 的 SPL 输入参数的值。

随同 ORDER BY 子句，您可检索符合条件的行的前 ***max*** 行。例如，下列查询找到薪酬最高的 10 名雇员：

```
SELECT FIRST 10 name, salary FROM emp ORDER BY salary DESC;
```

您可使用在查询中的 FIRST 选项，该查询的结果集在另一 SELECT 语句的 FROM 子句之内定义集合派生的表（CDT）。下列查询指定有不多于 10 行的 CDR:

```
SELECT *
        FROM TABLE(MULTISET(SELECT FIRST 10 * FROM employees
        ORDER BY employee_id)) vt(x,y), tab2
        WHERE tab2.id = vt.x;
```

在 FROM 子句中包括表表达式的查询中，FIRST 和 SKIP 关键字还有效：

```
SELECT * FROM (SELECT SKIP 2 FIRST 8 col1 FROM tab1 WHERE col1 > 50 );
```

下一示例将 FIRST 选项用于 UNION 表达式的结果：

```
SELECT FIRST 10 a, b FROM tab1 UNION SELECT a, b FROM tab2;
```

在任何下列上下文中，FIRST 选项都无效：

- 在视图的定义中
- 在嵌套的 SELECT 语句中
- 在子查询中，除了在 FROM 子句中指定表表达式的那些子查询之外
- 在 SPL 例程之内的单 SELECT 中（此处 ***max*** = 1）
- 将嵌套的 SELECT 语句用作表达式的情况下

## LIMIT 关键字

LIMIT 是在 Projection 子句中 FIRST 关键字的关键字同义词。然而，您不可在 FIRST 为有效的其他语法上下文中以 LIMIT 替换 FIRST，比如在 FETCH 语句中。

## 使用 TOP 选项

TOP 选项支持以下两种方式：

- TOP ***N*** ：指定取得结果集中前 ***N*** 条记录。
- TOP ***M,N*** ：指定取得结果集中第 ***M*** 条记录之后的 ***N*** 条记录。

您可检索前 ***N*** 个符合条件的行。例如，下列查询将找到薪酬最高的 10 名雇员：

```
SELECT TOP 10 name, salary FROM employee ORDER BY salary DESC;
```

下列查询将找到薪酬第二高的雇员：

```
SELECT TOP 1,1 name, salary FROM employee ORDER BY salary DESC;
```

## 使用 SKIP、FIRST、TOP、LIMIT 或 MIDDLE 作为列名称

如果没有整数紧跟在 FIRST 关键字之后，则数据库服务器将 FIRST 解释为列标识符。例如，如果表 T 有列 first、second 和 third，则下列查询会从名为 first 的列返回数据：

```
SELECT first FROM T
```

同样的考虑也适用于 TOP、SKIP 和 LIMIT 关键字。如果 Projection 子句中的 LIMIT 关键字之后没有字面整数也没有整数变量，则 GBase 8s 将 LIMIT 解释作为列名称。如果 FROM 子句中的数据源不具有带有该名称的列，则查询失败并报错。

## 使用带有 FIRST 选项（或 TOP 选项）的 SKIP 选项

如果带有 SKIP *offset* 选项的 Projection 子句还包括 FIRST 、TOP 或 LIMIT，则结果集将以符合条件的行集合中顺序位置为（*offset* + 1）的行开始，而不是以第一行开始。除非符合条件的行少于（*offset* + *max*），否则位置（*offset* + *max*）上的行是结果集中的最后一行，。下列示例忽略了表 tab1 的前 50 行，但返回最多 10 行的结果集，以第 51 行开始：

```
SELECT SKIP 50 FIRST 10 a, b FROM tab1;
```

下一示例在查询中使用 SKIP 和 FIRST 来将不多于 5 行从表 **tab1** 插入到表 **tab2** 内，以第 11 行开始：

```
INSERT INTO tab2 SELECT SKIP 10 FIRST 5 * FROM tab1;
```

也可以使用 TOP 关键字代替 FIRST 关键字，以下示例的效果等同于上一示例：

```
INSERT INTO tab2 SELECT SKIP 10 TOP 5 * FROM tab1;
```

下列集合子查询仅返回第 11 至第 15 之间的符合条件的行作为集合派生的表，通过列 a 中的值排列这 5 行的顺序，并将此结果集存储在临时表中。

```
SELECT * FROM TABLE (MULTISET
(SELECT SKIP 10 FIRST 5 a FROM tab3
                    ORDER BY a)) INTO TEMP;
```

下列 INSERT 语句包含一个集合子查询，该集合子查询的结果将定义集合派生表。这些行按列 a 中的值排列顺序，并将插入到表 tab1 内。

```
INSERT INTO tab1 (a)
SELECT * FROM TABLE (MULTISET (SELECT SKIP 10 FIRST 5 a
        FROM tab3 ORDER BY a));
```

将 FIRST 或 LIMIT 或 TOP 和 SKIP 选项与 ORDER 子句结合在一起的查询可对符合条件的行强加唯一顺序，因此按 ***max*** 的值增大 ***offset*** 值的连续查询可将符合条件的行划分为 ***max*** 行的不相连子集。这可支持需要固定页面大小的 web 应用程序，而无需游标管理。

仅当所有参与的数据库服务器都支持 SKIP 和 FIRST 选项（或 TOP 选项），您才可在分布式查询中使用这些特性。

# 允许重复

您可应用 ALL、UNIQUE 或 DISTINCT 关键字来指示是否返回重复的值，如果存在的话。如果您在 Projection 子句中未指定任何这些关键字，则在缺省情况下返回所有符合条件的行。

| 关键字 | 作用 |
| --- | --- |
| ALL | 指定返回所有符合条件的行，不论是否存在重复。（这是缺省的规范。） |
| DISTINCT | 从结果集排除重复的符合条件的行 |
| UNIQUE | 排除重复。（此处 UNIQUE 是 DISTINCT 的同义词。这是对 ANSI/ISO 标准的扩展。） |

例如，下一查询返回从 items 的行中的 stock_num 和 manu_code 列的所有唯一的有顺序的值对。如果几行有相同的值对，则那个值对仅在结果集中显示一次：

```
SELECT DISTINCT stock_num, manu_code FROM items;
```

要获得数据库服务器如何标识在有 NLCASE INSENSITIVE 属性的数据库中重复的 NCHAR 和 NVARCHAR 值的信息，请参阅 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

在每一查询或子查询级别内，您可指定 SELECT 语句的 DISTINCT 或 UNIQUE 关键字不超过一次。 下列示例在查询和子查询中都使用 DISTINCT ：

```
SELECT DISTINCT stock_num, manu_code FROM items
        WHERE order_num = (SELECT DISTINCT order_num FROM orders
        WHERE customer_num = 120);
```

上述示例是有效的，因为在每一 SELECT 语句中使用 DISTINCT 未超过一次。

如果查询在 Projection 子句中包括 DISTINCT 或 UNIQUE 关键字（而不是 ALL 关键字或无关键字），该子句的 Select 列表还包括一个其参数列表以 DISTINCT 或 UNIQUE 开头的合计函数，则数据库服务器发出错误，如下例所示：

```
SELECT DISTINCT COUNT(DISTINCT ship_weight)
        FROM orders;
```

即，对于 Projection 子句，以及对于将结果集限制为唯一值的合计函数，它在同一查询中是无效的。（在上例中，以 UNIQUE 替换 DISTINCT 关键字中的一个不能避免此错误。）

### 以多个合计表达式的查询

如果 Projection 子句不指定 SELECT 语句的 DISTINCT 或 UNIQUE 关键字，则该查询可包括多个内建的合计函数，每一函数包括 DISTINCT 或 UNIQUE 关键字作为参数列表中的第一个规范，如下例所示：

```
SELECT COUNT (DISTINCT customer_num),
        COUNT (UNIQUE order_num),
        AVG(DISTINCT ship_charge) FROM orders;
```

在同一查询级别中，对 DISTINCT 或 UNIQUE 合计表达式的支持适用于内建的合计函数，但不适用于 CREATE AGGREGATE 语句定义的用户定义的合计（UDA）函数。如果在同一查询中多于一个 UDA 表达式的参数列表以 DISTINCT 或 UNIQUE 关键字开头，则数据库服务器发出错误。

## 在 NLSCASE INSENSITIVE 数据库中重复的行

在以 NLSCASE INSENSITIVE 选项创建的数据库中，NCHAR 或 NVARCHAR 数据类型的列和表达式在大写和小写字母之间没有差别，因此，有相同的字符序列的这些数据类型的字符串，但有字母大小写区别，取值重复。

在数据库内已加载了相同的字符串，包括 ALL、DISTINCT 或 UNIQUE 关键字的查询返回的结果可能不同于同一查询从区分大小写的数据句库返回的结果。例如，在有 NLSCASE INSENSITIVE 属性的数据库中将 NVARCHAR 字符串 "aCe"、"ACE"，和 "AcE" 算作完全相同的，但在区分大小写的数据库中，同样的三个字符串会作为不同的值进行处理。

然而，通过使用 ALL、DISTINCT 或 UNIQUE 关键字来包括或排除重复的行，在 NLSCASE SENSITIVE 和在 NLSCASE INSENSITIVE 数据库中，CHAR、LVARCHAR 和 VARCHAR 类型的字符串都做相同的处理。要获取更多关于带有 NLSCASE INSENSITIVE 属性的数据库的信息，请参阅 指定 NLSCASE 区分大小写 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## 分布式查询中的数据类型

其唯一数据源是会话连接到的本地数据库中的表和视图的那些查询，可从注册在本地数据库中的任何内建的或用户定义的数据类型的列或表达式返回值。引用其他数据库中的表或视图的查询称为***分布式查询***，其它们可访问的数据类型是 GBase 8s 在本地查询中支持的数据类型的子集。

在分布式查询之中，对数据类型的限制依赖于参与的数据库服务器的数目。

- 如果查询访问的所有数据库都是同一 GBase 8s 实例的数据库，则该查询称为***跨数据库***分布式查询。
- 如果该查询访问多个 GBase 8s 实例的数据库，则该查询称为***跨服务器***分布式查询。

在这两类分布式查询中，所有参与的数据库都必须有相同的符合 ANSI/ISO 的状态。如果所有参与的服务器都支持 SKIP 选项，则跨服务器分布式查询可使用 SKIP 和 FIRST 选项；否则该查询失败并报错。大多数情况下，所有跨服务器操作需要参与的数据库服务器实例都支持指定该操作的 SQL 语法。

### *跨数据库事务中的数据类型*

仅访问本地 GBase 8s 实例的数据库的分布式查询（以及其他分布式 DML 操作或函数调用）可访问下列类别的数据类型：

- 非 opaque 的***内建的数据类型***，包括这些：
    - BIGINT
    - BIGSERIAL

  - BYTE
  - CHAR
  - DATE
  - DATETIME
  - DECIMAL
  - FLOAT
  - INT
  - INTERVAL
  - INT8
  - MONEY
  - NCHAR
  - NVARCHAR
  - SERIAL
  - SERIAL8
  - SMALLFLOAT
  - SMALLINT
  - TEXT
  - VARCHAR
- 大多数***内建的 opaque 数据类型***，包括这些：
  - BLOB
  - BOOLEAN
  - CLIENTBINVAL
  - CLOB
  - IFX_LO_SPEC
  - IFX_LO_STAT
  - INDEXKEYARRAY
  - LVARCHAR
  - POINTER
  - RTNPARAMTYPES,
  - SELFUNCARGS
  - STAT
  - XID
- 显示地强制转型为上列任何内建的类型的用户定义的类型（UDT）
- 在前面的列表中任何内建类型的 DISTINCT。

仅当所有 UDT 和 DISTINCT 类型都显式地强制转型为内建的数据类型，本地 GBase 8s 实例的跨数据库分布式操作才可基于内建的数据类型返回 UDT 和 DISTINCT 类型。

在参与该分布式查询中的每一数据库中，所有 opaque UDT、DISTINCT 类型、数据类型层级和强制转型都必须有完全相同的定义。对于使用上列数据类型作为参数或作为返回的数据类型的跨服务器 UDR 中的查询或其他 DML 操作，该 UDR 还必须在每一参与的数据库中有相同的定义。

如果跨数据库的分布式查询（或任何其他跨数据库 DML 操作）引用在另一包括任何下列数据类型的列的 GBase 8s 实例的另一数据库中的表、视图或同义词，则该分布式查询失败并报错：

- LOLIST
- IMPEX
- IMPEXBIN
- SENDRECV
- 上列的任何 opaque 数据类型的 DISTINCT。
- 复合的类型（命名的或未命名的 ROW、COLLECTION、LIST、MULTISET 或 SET）

### *跨服务器事务中的数据类型*

跨两个或多个 GBase 8s 实例的服务器的分布式查询（或任何其他的分布式 DML 操作或函数调用）不可返回复合的或大对象数据类型，也不可返回大多数用户定义的数据类型（UDT）或 opaque 数据类型。跨服务器分布式查询、DML 操作和函数调用仅可返回下列数据类型：

- 任何非 opaque 的内建的数据类型
- BOOLEAN
- LVARCHAR
- 非 opaque 的内建的类型的 DISTINCT
- BOOLEAN 的 DISTINCT
- LVARCHAR 的 DISTINCT
- 在此列表中任何出现在上面的 DISTINCT 类型的 DISTINCT。

跨服务器的分布式查询仅可支持 DISTINCT 数据类型，如果显式地将它们强制转型为内建的类型，且在参与该分布式查询的每一数据库中，则用完全相同的方式定义所有 DISTINCT 类型、它们的数据类型层级和它们的强制转型。对于那些使用前面罗列的数据类型作为参数或作为返回的数据类型的查询或其他跨服务器的 UDR 中的 DML 操作，该 UDR 必须在每个参与的数据库中有相同的定义。

存储安全标签对象的内建的 DISTINCT 数据类型 IDSSECURITYLABEL，可由持有足够的安全凭证的用户跨服务器地和跨数据库操作地对受保护的数据进行访问。就像对受保护的数据进行本地操作一样，在数据库服务器已经将保护数据安全的数据的安全标签与发出该查询的用户的安全凭证进行比较之后，访问由安全策略保护的远程表的那些分布式查询可仅返回 **IDSLBACRULES** 允许的符合条件的行。

要获取更过关于在跨数据库 DML 操作中 GBase 8s 支持的数据类型的附加信息，请参阅 分布式查询中的数据类型。要获取关于在跨服务器操作中有效的 DISTINCT 数据类型的表层级的信息，请参阅 分布式操作中的 DISTINCT 类型。

如果跨服务器查询（或任何其他跨服务器 DML 操作）引用包括任何下列数据类型的列的另一 GBase 8s 实例的数据库中的表、视图或同义词，则查询失败并报错。：

- BLOB
- CLOB
- INDEXKEYARRAY
- POINTER
- RTNPARAMTYPES
- SELFUNCARGS
- IFX_LO_SPEC
- IFX_LO_STAT
- STAT
- CLIENTBINVAL
- 用户定义的 OPAQUE 类型
- 复合的类型（命名的或未命名的 ROW、COLLECTION、LIST、MULTISET 或 SET）
- 任何以上罗列的 opaque 或复合的数据类型的 DISTINCT。

跨服务器查询不可访问另一 GBase 8s 实例的数据库，除非两个服务器都在它们的 DBSERVERNAME 或 DSERVERALIASES 配置参数中和在 **sqlhosts** 信息中定义 TCP/IP 或 IPCSTR 连接。对两个参与的服务器都要支持相同的连接类型（TCP/IP 抑或 IPCSTR）的要求也适用于 GBase 8s 实例之间的任何通信，即使二者位于同一台计算机上。

## Select 列表中的表达式

在选择列表中，您可使用任何基本类型的表达式（列、常量、内建的函数、聚集函数和用户定义的例程）及其组合。在 表达式 中描述表达式类型。以下部分展示在选择列表中的简单表达式的示例。

您可通过加、减、乘、除算术运算符将简单的数值表达式连接组合起来。然而，如果您组合列表达式与聚集函数，则必须在 GROUP BY 子句中包括该列表达式。（另请参阅 GROUP BY 与 Projection 子句之间的依赖。）

通常，您不可在选择列表中使用变量（例如，在 GBase 8s ESQL/C 应用程序中的主变量）本身。然而，如果以算术运算符或连接运算符将它与变量相连接，则选择列表中的变量是有效的。

在 FOREACH SELECT 语句中，当 FROM 子句中的表为远程表时，您不可使用选择列表中的 SPL 变量本身或随同列名称使用。当 FROM 子句中的表为本地表时，您可使用 SPL 变量自身，或随同选择列表中的常量使用。

在 GBase 8s 的分布式查询中，表达式中的值（以及表达式返回的值）是受限的，如 跨服务器事务中的数据类型 所述。在同一 GBase 8s 实例的其他数据库中将其返回值用作表达式的任何 UDR，必须定义在每一参与的数据库中。

在 Projection 子句中，布尔操作符 NOT 不是有效的。

## *选择列*

列表达式是在 SELECT 语句中最常用的表达式。要获取列表达式的语法和使用的完整描述，请参阅列表达式。下列示例在 Projection 子句中使用列表达式：

```
SELECT orders.order_num, items.price FROM orders, items;
SELECT customer.customer_num ccnum, company FROM customer;
SELECT catalog_num, stock_num, cat_advert [1,15] FROM catalog;
SELECT lead_time - 2 UNITS DAY FROM manufact;
```

## *选择常量*

如果您在 projection 列表中包含常量表达式，则对查询返回的每一行返回相同的值（除了当常量表达式为 NEXTVAL 之外）。要了解常量表达式的语法和使用的完整描述，请参阅 常量表达式。下列示例展示选择列表之内的常量表达式：

```
SELECT 'The first name is', fname FROM customer;
SELECT TODAY FROM cust_calls;
SELECT SITENAME FROM systables WHERE tabid = ;1
SELECT lead_time - 2 UNITS DAY FROM manufact;
SELECT customer_num + LENGTH('string') from customer;
```

## *选择内建的函数表达式*

内建的函数表达式使用对查询中每一行求值的函数。所有内建的函数表达式都需要参数。这些表达式包含时间函数和长度函数，当它们随同列名称作为参数使用时。下列示例展示 Projection 子句的选择列表之内的内建的函数表达式：

```
SELECT EXTEND(res_dtime, YEAR TO SECOND) FROM cust_calls;
SELECT LENGTH(fname) + LENGTH(lname) FROM customer;
SELECT HEX(order_num) FROM orders;
SELECT MONTH(order_date) FROM orders;
```

## *选择聚集函数表达式*

聚集函数返回对一系列被查询的行的一个值。此值依赖于 SELECT 语句指定的 WHERE 子句的行的集合。如果缺少 WHERE 子句，则聚集函数采用的值依赖于 FROM 子句构成的所有行。

下列示例展示 projection 列表中的聚集函数：

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013;
SELECT COUNT(*) FROM orders WHERE order_num = 1001;
SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer;
```

然而，如果 Projection 子句未指定 SELECT 语句的 DISTINCT 或 UNIQUE 关键字，则查询可包括一个或多个包括 DISTINCT 或 UNIQUE 关键字作为参数列表的第一个规范的聚集函数：

```
SELECT SUM(DISTINCT total_price) FROM items WHERE order_num = 1013;
SELECT COUNT(DISTINCT *) FROM orders WHERE order_num = 1001;
SELECT MAX(LENGTH(fname) + LENGTH(UNIQUE lname)) FROM customer;
```

然而，如果 Projection 子句和聚集函数表达式都在同一查询中指定 DISTINCT 或 UNIQUE 关键字，则数据库服务器发出错误。

对于包括聚集函数表达式的网格查询，您必须在子查询中指定 GRID 子句，每一网格服务器计算的聚集表达式的值为商，这个商的分母在参与的网格服务器之间不同。

请不要将 SQL 聚集函数与“联机分析处理”（OLAP）窗口聚集函数混淆，它们是不同类别的函数。

当聚集函数表达式紧跟在 OVER 子句之后时，数据库服务器尝试将它解释为 OLAP 聚集函数。有些 OLAP 聚集函数与 SQL 聚集函数同名（并支持相同语法的子集），但这两类函数的行为不同。

SQL 聚集函数可嵌套在 OLAP 聚集函数之中。例如，在 dollars 为 sales 表 中的列的上下文中，下列查询是有效的：

```
SELECT AVG(SUM(dollars)) OVER() FROM sales;
```

在上述示例中，SUM 函数是 SQL 聚集函数，且包含的 AVG 函数是 OLAP 窗口函数。查询处理的顺序规定总是在分组和聚集操作之后、最后的 ORDER BY 操作之前计算 OLAP 函数。

## *选择 OLAP 窗口表达式*

您可在 Projection 子句的选择列表中包括 OLAP 窗口表达式。

“联机分析处理”（OLAP）函数可返回对查询或子查询的整个结果集、对 OLAP 定义的符合条件的行的分区的子集的排名、行号和聚集函数信息。您可使用 OLAP 规范来定义对于数据的检测维度结果集的分区之内移动窗口，以及标识模式、趋势和数据集内的例外。

包括 OLAP 窗口表达式的查询返回该查询的结果集中的行，以及 OLAP 窗口函数的结果，如果那些函数有任何返回的话。

OLAP 窗口聚集函数表达式可作为另一 OLAP 窗口聚集函数的参数。然而，OLAP 窗口聚集不可为非分析的聚集函数的参数。

## *选择用户定义的函数表达式*

用户定义的函数扩展了您可用的函数的范围，并允许您对您选择的每一行执行子查询。

下列示例为每一 customer_num 调用 get_orders() 用户定义的函数，并显示 n_orders 标签之下的返回的值：

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
        FROM customer;
```

如果 SELECT 语句中的 SPL 例程包含某些 SQL 语句，则数据库服务器返回错误。要获取关于在查询之内调用的 SPL 例程中不可使用哪些 SQL 语句的信息，请参阅 在数据操纵语句中 SPL 例程的限制。

要获取用户定义的函数表达式的完整语法，请参阅 用户定义的函数。

## *选择使用算术运算符的表达式*

您可将数值表达式与算术运算符组合来生成复合的表达式。您不可将包含聚集函数的表达式与列表达式组合。这些示例展示在 Projection 子句中的选择列表之内使用算术运算符的表达式：

```
SELECT stock_num, quantity*total_price FROM customer;
SELECT price*2 doubleprice FROM items;
SELECT count(*)+2 FROM customer;
SELECT count(*)+LENGTH('ab') FROM customer;
```

### *选择 ROW 字段*

您可以 *row.field* 表示法来选择命名的或未命名的 ROW 类型列的特定字段。例如，假设您有下列表结构：

```
CREATE ROW TYPE one (a INTEGER, b FLOAT);
        CREATE ROW TYPE two (c one, d CHAR(10));
        CREATE ROW TYPE three (e CHAR(10), f two);

        CREATE TABLE new_tab OF TYPE two;
        CREATE TABLE three_tab OF TYPE three;
```

下列示例展示在选择列表中为有效的表达式：

```
SELECT t.c FROM new_tab t;
        SELECT f.c.a FROM three_tab;
        SELECT f.d FROM three_tab;
```

您还可在字段名的位置输入星号（*）来表示被选择的 ROW 类型列的所有字段。

例如，如果 my_tab 表有一包含四个字段的名为 rowcol 的 ROW 类型列，则下列 SELECT 语句检索 rowcol 列的所有四个字段：

```
SELECT rowcol.* FROM my_tab;
```

您还可通过仅指定列名称来从 row 类型列检索所有字段。此示例与前一查询有相同的作用：

```
SELECT rowcol FROM my_tab;
```

您不仅可随同 ROW 类型列使用 ***row.field*** 表示法，还可随同取值结果为 ROW 类型值的表达式使用。要获取更多信息，请参阅表达式部分中的“列表达式”。

## 声明显示标签

您可为 Projection 子句的选择列表中的任何列或列表达式声明显示标签。仅在 SELECT 语句正在执行时，此临时名称有效。

在 DB-Access 中，显示标签显示为 SELECT 语句的输出中那列的标题。

在 GBase 8s ESQL/C 中，***display_label*** 的值保存在 **sqlda** 结构的 **sqlname** 字段中。要获取更多关于 **sqlda** 结构的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

如果您的显示标签是 SQL 关键字，则请使用 AS 关键字来阐明语法。例如，要使用 UNITS、YEAR、MONTH、DAY、HOUR、MINUTE、SECOND 或 FRACTION 作为显示标签，请随同该显示标签使用 AS 关键字。下列语句使用随同 minute 的 AS 作为显示标签：

```
SELECT call_dtime AS minute FROM cust_calls;
```

要了解 SQL 的关键字，请参阅 GBase 8s 的 SQL 关键字。

如果您使用 INTO Table 子句来创建临时的或永久的表来存储查询结果，则必须为不是简单类表达式的在选择列表中的任何数据库对象或表达式声明显示标签。在临时的或永久的表中，显示标签用作列的名称。

如果您正在使用 SELECT 语句来定义视图，请不要使用显示标签。而要在 CREATE VIEW 列列表中指定想要的标签名称。

## 声明列别名

您可为 Projection 子句的选择列表中的任何列声明别名。GROUP BY 子句可通过列的别名引用它。仅在 SELECT 语句正在执行时，此临时名称才有效。

如果您的别名是 SELECT 语句的 SQL 关键字，则请使用 AS *column_alias* 关键字来阐明语法。例如，要使用 FROM 作为表别名，必须在该别名前紧接着 AS 关键字，来避免语法错误。下列语句使用带有以 from 的 AS 作为别名：

```
SELECT status AS from FROM stock GROUP BY from;
```

下列等同的查询声明 pcol 作为别名，并在 GROUP BY 子句中使用那个别名：

```
SELECT pseudo_corinthian AS pcol FROM architecture GROUP BY pcol;
SELECT pseudo_corinthian pcol FROM architecture GROUP BY pcol;
```

在 Oracle 模式下，保持 GBase 8s 原有声明列别名语法基础上，如果 SELECT 语句的 SQL 关键字包括 NAME、TEMP、ARRAY、LIST、REVERSE、CONTEXT、LENGTH、LOG 作为列别名使用，可以不用关键字 AS 开始它的声明。

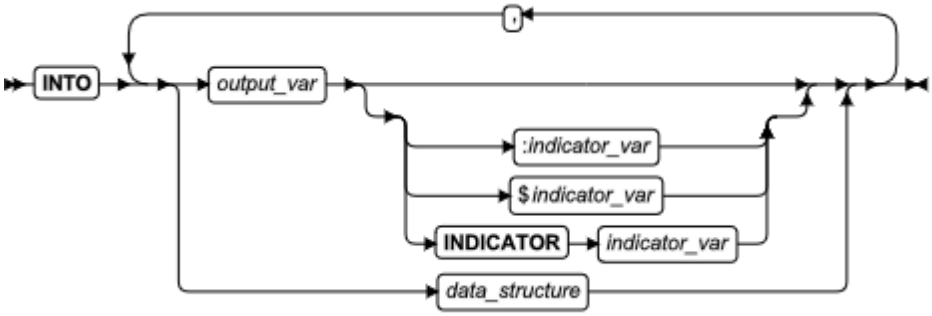
例如，查询 customer 表使用关键字 temp 作列别名：

```
SELECT col temp FROM customer;
```

## INTO 子句

在 SPL 例程或 GBase 8s ESQL/C 程序中使用 INTO 子句来指定程序变量或主变量来接收 SELECT 检索的数据。

INTO 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_structure* | 声明了作为主变量的结构 | 必须能够存储正在选择的值的元素的数据类型 | 特定于语言 |
| *indicator_var* | 如果相应的 *output_var* 收到 NULL 值，则来接收返回代码的程序变量 | 可选的；如果相应的 *output_var* 的值由为 NULL 的可能性，则使用指示符变量 | 特定于语言 |
| *output_var* | 接收相应的选择列表项的值的程序或主变量。可为集合变量 | 接收变量的顺序必须与 Projection 子句的选择列表中相应的项的顺序相匹配 | 特定于语言 |

INTO 子句指定一个或多个接收查询返回的值的变量。如果它返回多个值，则以您指定这些变量的顺序将它们赋予变量列表。

如果 SELECT 语句是孤立的（即，不是 DECLARE 语句的一部分，且不使用 INTO 子句），则必须为单 SELECT 语句。*单* SELECT 语句仅返回一行。

接收的变量的数目必须等于 Projection 子句的选择列表中项的数目。每一接收的变量的数据类型应与选择列表中相应的列或表达式的数据类型相兼容。

当接收的变量的数据类型与被选择的项不相匹配时，要了解数据库服务器采取的活动，请参阅 ESQL/C 中的警告。

下列示例展示 GBase 8s ESQL/C 中的单 SELECT 语句：

```
EXEC SQL select fname, lname, company
        into :p_fname, :p_lname, :p_coname
        from customer where customer_num = 101;
```

在 SPL 例程中，如果 SELECT 返回多于一行，则您必须使用 FOREACH 语句来分别地访问这些行。SELECT 语句的 INTO 子句持有获取的值。要获取更多信息，请参阅 FOREACH。

## 带有指示符变量的 INTO 子句

如果存在从查询返回的数据值为 NULL 的可能性，则请在 INTO 子句中使用 ESQL/C 指示符变量。要获取更多信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

## 带有游标的 INTO 子句

如果 SELECT 语句返回多于一行，则您必须在 FETCH 语句中使用游标来分别地存取这些行。您可将 INTO 子句放在 FETCH 语句中，而不是在 SELECT 语句中，但您不应将其同时放在两个语句中。

下列 GBase 8s ESQL/C 代码示例展示您可使用 INTO 子句的不同的方式。如两个示例所示，您必须首先使用 DECLARE 语句来声明游标。

***在 SELECT 语句中使用 INTO 子句***

```
EXEC SQL declare q_curs cursor for
    select lname, company
        into :p_lname, :p_company
        from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
    EXEC SQL fetch q_curs;
EXEC SQL close q_curs;
```

***使用 FETCH 语句中的 INTO 子句***

```
EXEC SQL declare q_curs cursor for
    select lname, company from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
    EXEC SQL fetch q_curs into :p_lname, :p_company;
EXEC SQL close q_curs;
```

## 准备 SELECT ... INTO 查询

在 GBase 8s ESQL/C 中，您不可准备带有 INTO 子句的查询。您可准备不带有 INTO 子句的查询，为准备好的查询声明游标，打开游标，然后使用带有 INTO 子句的 FETCH 语句来获取程序变量之内的游标。

或者，您可为查询声明游标，而不准备该查询，并当您声明该游标时在该查询中包括 INTO 子句。然后打开该游标并获取该游标，而不使用 FETCH 语句的 INTO 子句。

## 使用带有 INTO 子句的数组变量

在 GBase 8s ESQL/C 中，如果您随同包含 INTO 子句的 SELECT 语句使用 DECLARE 语句，且该变量为数组元素，则可以整数字面值或变量来标识该数组的单个元素。当声明该游标时，确定用作下标的变量的值；随后该下标变量表现为常量。

下列 GBase 8s ESQL/C 代码示例为 SELECT ... INTO 语句声明游标，使用变量 i 和 j 作为数组 a 的下标。在您声明该游标之后，SELECT 语句的 INTO 子句等同于 INTO a[5], a[2]。

```
i = 5
j = 2
EXEC SQL declare c cursor for
select order_num, po_num into :a[i], :a[j] from orders
where order_num =1005 and po_num =2865;
```

您还可在 FETCH 语句中使用程序变量来指定 INTO 子句中程序的元素。在每一获取操作时为程序变量求值，而不是当您声明该游标时求值。

## 错误检查

如果收到的变量的数据类型与被选择的项的数据类型不相匹配，如果可能，则将被选择的项的数据类型转换为该变量的数据类型。如果不可能转换，则发生错误，并在 **status** 变量、**sqlca.sqlcode** 或 **SQLCODE** 中返回负数。在此情况下，程序变量中的值是无法预知的。

在符合 ANSI 的数据库中，如果罗列在 INTO 子句中的变量的数目不同于 Projection 子句的选择列表中项的数目，则会收到错误。

### *ESQL/C 中的警告*

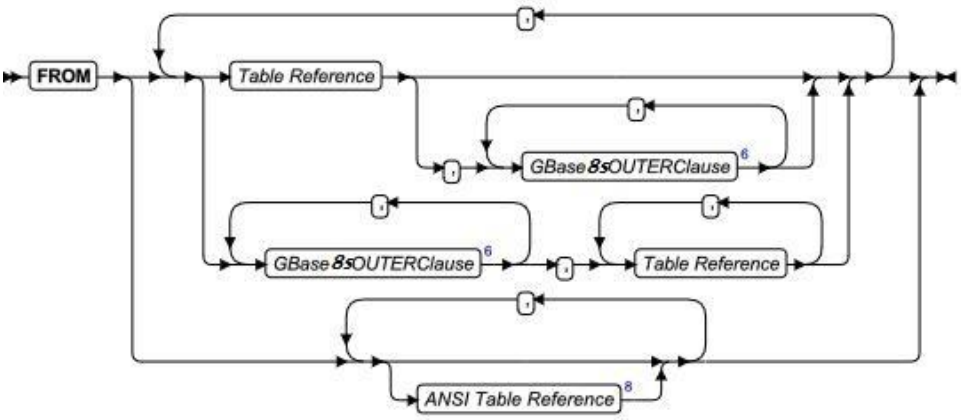
在 GBase 8s ESQL/C 中，如果罗列在 INTO 子句中的变量的数目不同于 Projection 子句中项的数目，则在 **sqlwarn** 结构中返回警告：**sqlca.sqlwarn.sqlwarn3**。转换的变量的实际数目少于这两个数目。要获取更多关于 **sqlwarn** 结构的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

## FROM 子句

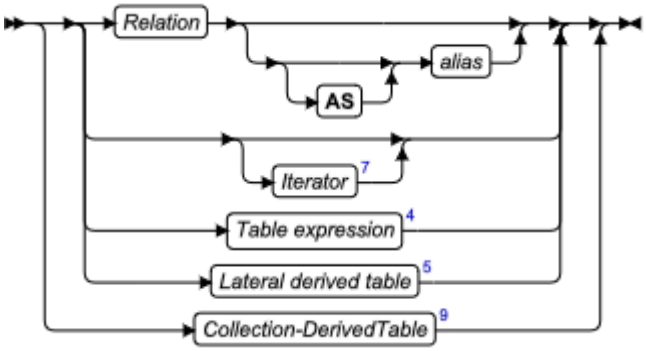
SELECT 语句的 FROM 子句罗列要从其检索数据的表对象。

FROM 子句有此语法：

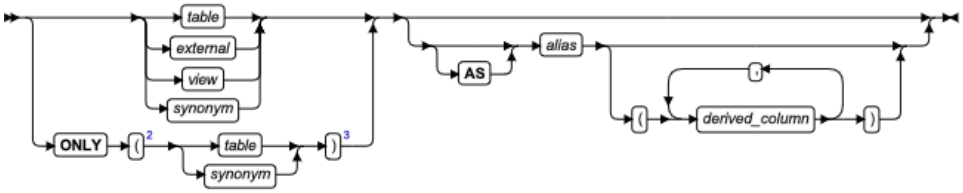
FROM 子句




表引用




关系




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 在此查询中的表、视图或派生的表的临时名称 | 请参阅 AS 关键字。 | 标识符 |
| *derived_column* | 在表表达式中派生的列的临时名称 | 除非底层集合为 ROW 类型，否则您声明的 *derived_column* 名称不能超过一个 | 标识符 |
| *external* | 从其检索数据的外部的表 | 必须存在，但不可为外连接中的外表 | 数据库对象名 |
| *synonym*、*table*、*view* | 从其检索数据的表的同义词 | 同义词和表或它指向的视图必须存在 | 数据库对象名 |

每个 SELECT 语句都需要 FROM 子句，不论是否需要任何数据源。如果您的查询使用数据库服务器来对一不需要数据源的表达式求值，则 FROM 子句可引用您在其上持有充足的访问权限的当前数据库中任何现有的表，如下例所示：

```
SELECT ATANH(SQRT(POW(4,2) + POW(5,2))) FROM systables;
```

如果 FROM 子句指定多个数据源，则该查询称为***连接***，因为它的结果集可从几个表引用连接行。要了解更多关于连接的信息，请参阅 连接表的查询。

## 表或视图的别名

您可为 FROM 子句中的表或视图声明别名。如果你这么做，则必须在 SELECT 语句的其他子句中使用该别名来引用该表或视图。您还可使用别名来缩短查询。

下列示例展示 FROM 子句的典型使用。第一个查询从 customer 表选择所有列和行。第二个查询在 customer 和 orders 表之间使用连接，来选择所有已下了订单的客户。

```
SELECT * FROM customer;
        SELECT fname, lname, order_num FROM customer, orders
        WHERE customer.customer_num = orders.customer_num;
```

下一示例等同于前一示例中的第二个查询，但它在 FROM 子句中声明别名，并在 WHERE 子句中使用它们：

```
SELECT fname, lname, order_num FROM customer c, orders o
        WHERE c.customer_num = o.customer_num;
```

别名（有时称为***相关名称***）对自连接特别有用。要获取更多关于自连接的信息，请参阅 自连接。在自连接中，您必须在 FROM 子句中罗列表名称两次，并为该表名称的两个示例各自声明一个不同的别名。

### *AS 关键字*

如果您使用可能发生歧义的此作为别名（或作为显示标签），则必须以关键字 AS 开始它的声明。如果您使用任何关键字 ORDER、FOR、AT、GROUP、HAVING、INTO、NOT、UNION、WHERE、WITH、CREATE 或 GRANT 作为表或视图的别名，则需要此关键字。

如果下一示例未包括了 AS 关键字来表明 not 是显示标签，而不是操作符，则数据库服务器会发出错误：

```
CREATE TABLE t1(a INT);
        SELECT a AS not FROM t1;
```

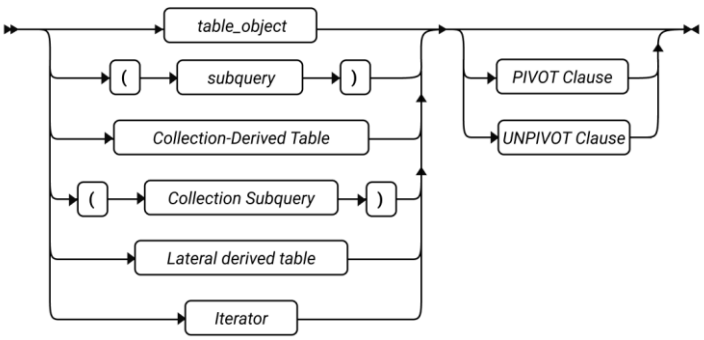
如果您未为集合派生的表声明别名，则数据库服务器为它指定一个与实现相关的名称。

## 表表达式

*表表达式*（有时称为*派生的表*）是表或视图的名称，或对一系列行求值的规范。这些行通常是嵌入到嵌套的 SELECT 语句中，或一些其他 SQL 语句中的查询的结果。

表表达式可有下列语法：

表表达式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *subquery* | 外查询可用其结果的嵌套的查询 | 请参阅下文**用法**说明 | SELECT 语句 |
| *table_object* | 名称、同义词，或表、视图或 EXTERNAL 表的别名 | 必须存在，或必须引用 SELECT 语句创建的派生的表 | 标识符 或 数据库对象名 |

**用法**

表表达式可为简单的或复合的：

- 简单的表表达式

  *简单的*表表达式是在保持查询结果的正确性时，其基础查询可包含进主查询内的表达式。

- 复合的表表达式

  *复合的*表表达式是在保持查询结果的正确性时，其基础查询不可包含进主查询内的表达式。数据库服务器将这样的表表达式具体化成在主查询中使用的临时表。在 FROM 子句中指定聚集、集合操作符或 ORDER BY 子句的子查询作为复合的表表达式实现，通常需要比简单的表表达式更多的数据库服务器资源。

在两种情况下，该表表达式作为常规的 SQL 查询求值，且它的结果可被认为是逻辑表。此逻辑表及其列可就如普通的基础表那样使用，但它不是持久的。它仅在引用它的查询的执行期间存在。

**对表表达式的限制**

表表达式与常规的 SELECT 语句有相同的语法，但有在其他上下文中适用于子查询的大部分限制。表表达式不可包括显式地创建结果表的 SELECT INTO 子句。

GBase 8s 不支持“通用化的键”索引。它支持在 CREATE TRIGGER 语句的触发器活动中的表表达式，并作为 Select 触发器的触发事件。 GBase 8s 还支持表表达式中的 ORDER BY 子句。

GBase 8s 支持迭代器函数作为 FROM 子句表标识符。然而，SPL 的 CALL 语句不可在 FROM 子句中的子查询内调用迭代器 TABLE 函数。

除了这些限制之外，任何有效的 SQL 查询都可为表表达式。表表达式可嵌套在另一表表达式之内，且可在它的定义中包括表和视图。您可在 CREATE VIEW 语句中使用表表达式来定义视图。

**相关的子查询和派生的表**

*相关的子查询*是引用未罗列在其 FROM 子句中的表的列的子查询。相反，仅引用罗列在其 FROM 子句中的表中的列的子查询是*不相关的子查询*。

在下列示例中，在其 FROM 子句中定义派生的表的不相关的子查询，在其 WHERE 子句中包含相关的子查询：

```
SELECT * FROM (SELECT * FROM t1
        WHERE a IN (SELECT b FROM t2 WHERE t1.a = t2.b));
```

此处，在第一个 WHERE 子句中的子查询是相关的子查询，因为它引用表 t1 的列 a，但它的 FROM 子句仅指定表 t2。

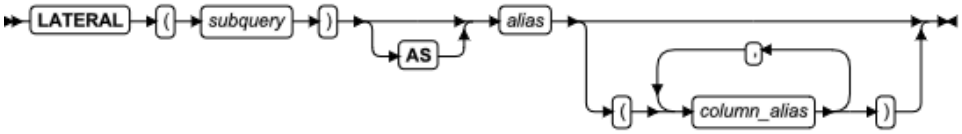
在 FROM 子句表表达式中，GBase 8s 还支持 ORDER BY 子句，这在 FROM 子句之外的子查询中不是有效的。在表表达式中通过 ORDER BY 子句指定的列或表达式不需要包括在 Projection 子句中。

## *横向的派生的表*

在定义派生的表的 FORM 子句中的任何查询都必须紧跟在 LATERAL 关键字之后，如果那个查询引用任何其他表或列，在同一 FROM 子句中该表或列出现得早于定义该派生的表的那个查询的话。

横向的派生的表，以及可在它们的语法中声明的表和列别名的引用的范围，是 SQL 语言的 ISO/ANSI 标准的一部分。此为 FROM 子句中横向的派生的表的语法：

LATERAL 派生的表




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 在此为 *subquery* 结果的派生的表声明的临时名称 | 请参阅 AS 关键字。 | 标识符 |
| *column_alias* | 在此为派生的表中列声明的临时的名称 | | 标识符 |
| *subquery* | 指定要检索的行 | 可为不相关的或相关的 | SELECT 语句 |

### 用法

如果其结果集为派生的表的 ***subquery*** 引用任何早于在同一 FROM 子句中出现的任何表或列，则需要 LATERAL 关键字。此处，*早于*意味着在 FROM 子句中的从左至右的语法令牌顺序中“在派生的表的左边”。以 LATERAL 关键字定义的派生的表称为*横向的派生的表*。

这支持对 FROM 子句中的其他表中列的引用，而不是仅对随后的派生的表中列的引用。可提升在连接一个或多个派生的表的 SELECT 语句中的性能。在 DELETE、UPDATE 和 CREATE VIEW 语句之内的派生的表中，横向的表和列引用也是有效的。

如果在派生的表中，已解析了所有不相关的表和列引用，则在这些派生的表的 FROM 子句中不需要 LATERAL 关键字。

### 横向的派生的表的示例

下列查询在 FROM 子句中包括横向的派生的表，其中 t1_a 是横向的相关引用：

```
SELECT * FROM t1 ,
        LATERAL (SELECT t2.a AS t2_a
        FROM t2 WHERE t2.a = t1.a);
```

在下一示例中，d.deptno 是横向的相关引用：

```
SELECT d.deptno, d.deptname,
    empinfo.avgsal, empinfo.empcount
    FROM department d,
    LATERAL (SELECT AVG(e.salary) AS avgsal,
    COUNT(*) AS empcount
    FROM employee e
    WHERE e.workdept=d.deptno) AS empinfo;
```

在此，列表达式的 avgsal 和 empcount 别名以及 empinfo 横向的表引用出现在外查询的 projection 列表中，使用关联 deptno，从 **department** 表和派生的表连接符合条件的行。

**对横向相关的引用的限制**

下列限制适用于横向的关联的表和列引用：

- 不可在 ANSI FULL OUTER JOIN 查询中使用它们。
- 不可在 ANSI RIGHT OUTER JOIN 查询中使用它们。
- 不可在 GBase 8s 扩展 OUTER JOIN 查询中使用它们。

## *可用性和性能考虑*

虽然通过视图、子查询可实现与表表达式等同的功能，但表表达式简化查询的公式，使得语法更加灵活和直观，并支持 SQL 的 ANSI/ISO 标准。

查询优化器不具体化 FROM 子句指定的简单的表表达式。与使用 GBase 8s 扩展 TABLE (MULTISET ( SELECT ... )) 语法来在 FROM 子句中指定等同的派生的表的查询的性能比起来，在 FROM 子句中对表表达式使用 ANSI/ISO 语法的查询的性能至少一样好，设置如 UNION、INTERSECT 或 MINUS 操作符 这样的操作符，或将 ORDER BY 规范作为复合的表表达式实现，可造成比简单的表表达式更大的开销。请使用 SET EXPLAIN 语句来检测查询计划和表表达式的预计成本。

下列是有效的表表达式的示例：

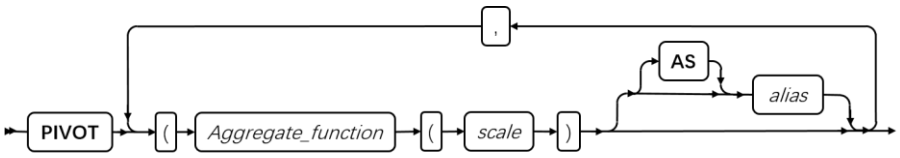
```
SELECT * FROM (SELECT * FROM t);

SELECT * FROM (SELECT * FROM t) AS s;

SELECT * FROM (SELECT * FROM t) AS s WHERE t.a = s.b;

SELECT * FROM (SELECT * FROM t) AS s, (SELECT * FROM u) AS v WHERE s.a = v.b;

SELECT * FROM (SELECT SKIP 2 col1 FROM tab1 WHERE col1 > 50 ORDER BY col1   DESC);
```

```
SELECT * FROM (SELECT col1,col3 FROM tab1
    WHERE col1 < 50 GROUP BY col1,col3 ORDER BY col3 ) vtab(vcol0,vcol1);

SELECT * FROM (SELECT * FROM t WHERE t.a = 1) AS s,
OUTER
(SELECT * FROM u WHERE u.b = 2 GROUP BY 1) AS v WHERE s.a = v.b;

SELECT * FROM (SELECT a AS colA FROM t WHERE t.a = 1) AS s,
OUTER
(SELECT b AS colB FROM u WHERE u.b = 2 GROUP BY 1) AS v
    WHERE s.colA = v.colB;

CREATE VIEW vu AS SELECT * FROM (SELECT * FROM t);

SELECT * FROM ((SELECT * FROM t) AS r) AS s;
```

# PIVOT 子句

PIVOT 子句用于查询中，将指定列的列值作为查询结果集的列名，实现将行旋转为列的表格查询，并在旋转过程中聚合数据。

PIVOT 子句





| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *scale* | 表达式 | 聚集函数的表达式 | 标识符 |
| *alias* | 别名 | 为聚集函数指定的别名； | 标识符 |

**用法**

- pivot 语法支持子查询、create view 、insert…select、with as 场景；
- 聚集函数的参数列和 pivot_for 子句的参数列必须为 select 子句的投影列，且必须为 select 子句中表的列；
- pivot 函数本身具备 group by 功能。即：按照旋转列分组，分组范围为 in 子句记录范围。隐式 group by 规则保持不变，且列的个数不能超过 255；
- pivot 子句的位置需紧跟 from 后；
- 表的数据类型不支持大对象、集合；
- 包含虚拟列的表不支持 pivot 转换；
- pivot_in 子句中参数为浮点值且未定义别名时，作为列名时会有精度损失。

### *aggregate_function*

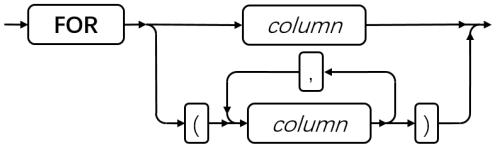
聚集函数定义 pivot 函数行转列的过程中，聚集操作的函数和处理对象。

- aggregate_function 支持的聚集函数包括：AVG、COUNT、MAX、MIN、STDDEV、VARIANCE、SUM；
- 当 pivot 子句中包含多个聚集函数时，可以为每个聚集函数定义别名，如果不定义别名，默认以‘_1’、‘_2’做区分；
- 聚集函数的参数支持为列名、纯数字或纯数字字符串参与的表达式（函数）。

## *pivot_for 子句*

pivot_for 子句定义行转列标准，即依据哪个列进行行转列。

pivot_for 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***column*** | 列名 | 在当前表中必须存在 | 标识符 |

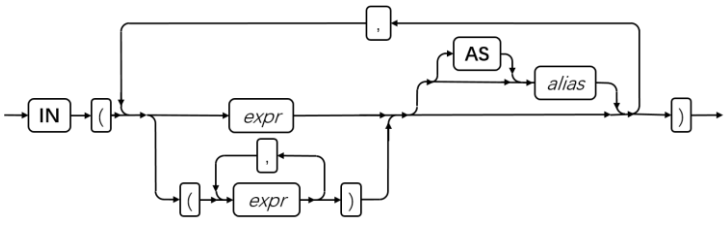
**用法**

- pivot_for 子句不支持表达式、函数，但支持投影列的别名；
- pivot_for 子句中的列个数与 pivot_in 子句中的表达式个数需保持一致。

## *pivot_in 子句*

pivot_in 子句定义行转列的列取值和顺序，即 pivot_in 子句中的名称及顺序作为转换后的表列名和顺序。

pivot_in 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***expr*** | 表达式 | 定义为行转列的列取值 | 标识符 |
| ***alias*** | 别名 | 为表达式指定的别名 | 标识符 |

**用法**

- pivot_in 子句中的表达式仅支持常量表达式、引用字符串；
- pivot_in 子句不支持重复列名或表达式；
- pivot 不支持 XML；
- 输出列数即 pivot_in 子句中表达式的个数与聚集函数的个数乘积不能超过 1024(即支持列数)。

### *转换新列名命名规则*

pivot 函数目标表列名命名规则为：pivot_in 子句参数名与聚集函数别名的组合。

具体规则如下:

- 聚集函数只有一个且未设置别名，pivot_in 子句中参数仅有一列，则新列名为参数名。例如：pivot…in ('a','b','c','d')，则新列名为 a，b，c，d；
- 聚集函数只有一个且未设置别名，pivot_in 子句中参数为多列，则新列名为多列的组合，各列之间以下划线（_）相连。例如：pivot…in((1,'a'),(4,'b'),(6,'c'),(8,'d')，则新列名为：1_a,…；
- 聚集函数只有一个且未设置别名，pivot_in 子句中参数设置了别名，则新列名为参数设置的别名。例如：pivot…in((1,'a') as aa)，则新列名为：aa；
- 聚集函数有多个且未设置别名，pivot_in 子句中参数仅有一列，则新列名为参数名与‘_1’，‘_2’…的组合，其中第一个聚集函数缺省，以参数名为列名。例如：pivot (sum(nums),sum(amount_sold) for name in('a','b')，则新列名为 a，b，a_1，b_1；
- 聚集函数设置别名，pivot_in 子句中参数为多列，则新列名为多列的组合，再与多个聚集函数的别名分别组合，以下划线（_）相连。例如： pivot (sum(nums) as nums,sum(amount_sold) as amount_sold for name in((1,'a'),(4,'b'))，则新列名为 1_a_nums，1_a_amount_sold，4_b_nums，4_b_amount_sold；
- 生成的新列名长度不允许超过数据库列名最大长度 128 字符，超过最大长度的按 128 截断，不报错；

例如，以下代码创建了一张名为 demo 的表，并向表中插入了 8 条数据：

```
create table demo(id int,quarter int,name varchar(20),nums int,amunt_sold decimal(8,2));

insert into demo values(1, 1,'a', 1000,200);

insert into demo values(2, 2,'a', 2000,100.12);

insert into demo values(3, 3,'a', 4000,156);

insert into demo values(4,1, 'b', 5000,234);

insert into demo values(5, 3,'b', 3000,345);

insert into demo values(6, 3,'c', 3500,222);

insert into demo values(7,1, 'd', 4200,111);

insert into demo values(8, 2,'d', 5500,555);


select * from demo;
```

result:

id quarter name nums amunt_sold

1 1 a 1000 200

2 2 a 2000 100.12

3 3 a 4000 156

4 1 b 5000 234

5 3 b 3000 345

6 3 c 3500 222

7 1 d 4200 111

8 2 d 5500 555

select 语句中的投影列仅包含 pivot_in 子句指定的列，结果集返回 1 行，如下列查询返回所有人的总 nums，并将 name 字段行转列显示：

```
select * from (select name,nums from demo)
PIVOT (sum(nums) for name in('a','b','c','d'));
```

result:

A B C D

7000 8000 3500 9700

select 语句中的投影列除 pivot_in 子句指定的列外，还包含表中其他的列，结果集返回多行，如下列查询实现按 quarter 统计所有人的总 nums，并将 name 字段行转列，显示别名：

```
select * from (select quarter,name,nums from demo)
PIVOT (sum(nums) for name in('a' as no,'b','c','d'));
```

result:

quarter no b c d

1 1000 5000 (NULL) 4200

2 2000 (NULL) (NULL) 5500

3 4000 3000 3500 (NULL)

下列查询实现多列转换，对（id, name）列进行行转列操作，如下列查询实现按 quarter 统计特定 id，name 组合的总 nums，并将 id 字段、name 字段行转列，显示别名:

```
select * from (select id,quarter,name,nums from demo)
PIVOT (sum(nums) as nums for (id,name) in((1,'a'),(4,'b'),(6,'c'),(8,'d')));
```

result:

| quarter | 1_a_nums | 4_b_nums | 6_c_nums | 8_d _nums |
|---|---|---|---|---|
| 1 | 1000 | 5000 | (NULL) | (NULL) |
| 2 | (NULL) | (NULL) | (NULL) | 5500 |
| 3 | (NULL) | (NULL) | 3500 | (NULL) |

下列查询实现按 quarter 统计 a 和 b 的总 nums 和总 amunt_sold，并将 name 字段行转列，显示别名：

```
select * from (select quarter,name,nums,amunt_sold from demo)
PIVOT (sum(nums) as nums,sum(amunt_sold) as amunt_sold for name in('a','b'));
```

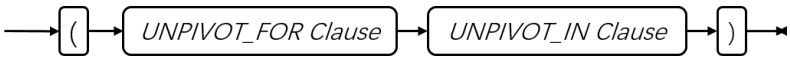
result:

| quarter | a_nums | a_amount_sold | b_nums | b_amount_sold |
|---|---|---|---|---|
| 1 | 1000 | 200 | 5000 | 234 |
| 2 | 2000 | 100.12 | (NULL) | (NULL) |
| 3 | 4000 | 156 | 3000 | 345 |

## UNPIVOT 子句

unpivot 子句用于查询中，将指定列的列名转换为查询结果集的列值，实现将列旋转为行。

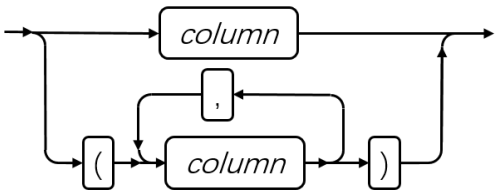
语法：


**用法**

- 仅支持对单表、视图、子查询进行 UNPIVOT 转换；
- unpivot 中自定义列名（投影列）不能为保留字；
- 包含虚拟列的表不支持 unpivot 转换；
- unpivot 的列不支持伪列。

### *unpivot_val_col_lst 子句*



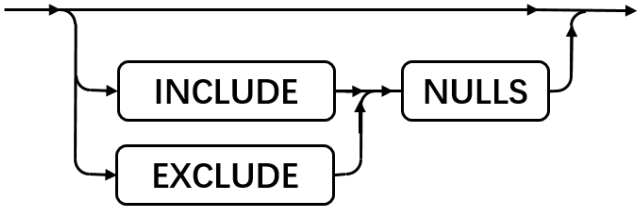
| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 定义在源表转换列的列值所对应的新列名 | 新表的列名，在新表中唯一 | 标识符 |

**用法**

- 表达式仅支持常量表达式；
- unpivot_val_col_lst 子句列名必须存在于投影列中；
- unpivot_val_col_lst 项个数与 unpivot_in 中待转换的列数目保持一致；
- unpivot_val_col_lst 子句列名不能与原表中列名重复。

### *include_null 子句*

include_null 子句定义 NULL 值的取舍。



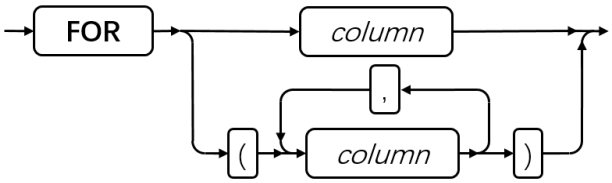

**用法**

- unpivot include NULL 定义包含 NULL 值，转换结果包含 NULL 值的行；
- unpivot exclude NULL 定义不包含 NULL 值，转换结果不包含 NULL 值；
- 缺省 include 和 exclude 关键字为不包含 NULL 值；

### *unpivot_for 子句*

unpivot_for 子句定义列转行标准，即依据哪个列进行列转行。

unpivot_for子句:



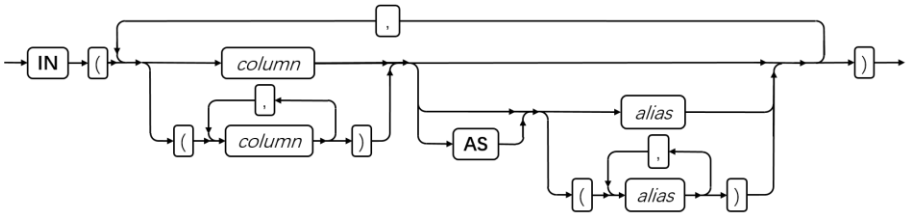

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 列转行作为列值时对应的列名 | 新表的列名，在新表中唯一 | 标识符 |

**用法**

- unpivot_for 子句不支持表达式、函数，但支持投影列的别名；
- unpivot_for 子句的项个数与 unpivot_in 子句中的 AS 项表达式数目保持一致

### *unpivot_in 子句*

unpivot_in 子句定义列转行的取值边界，即将原表哪些列转换为行数据。




| 元素 | 描述 | 限制 |
| --- | --- | --- |
| *column* | 转换为新表行数据的原表的列 | 原表中存在的列 |
| *alias* | 别名 | 定义显示在新表行数据中的列的别名，别名必须为常量表达式 |

**用法**

- unpivot_in 子句指定的列数据类型需保持一致；
- unpivot_in_clause 中指定的转换列数目最多 256 列；
- unpivot_in 子句的 as 项必须是常量表达式。

例如，以下代码创建了一个名为 fruit 的表，并向表中插入 5 条数据：

```
drop table fruit;

create table fruit(id int,name varchar(20),Q1 int,Q2 int,Q3 int,Q4 int);

insert into fruit values(1,'a',1000,2000,3300,5000);

insert into fruit values(2,'b',3000,3000,3200,1500);

insert into fruit values(3,'c',2500,3500,2200,2500);

insert into fruit values(4,'d',1500,2500,1200,3500);

insert into fruit values(5,'e',null,null,null,null);


select * from fruit;

result:

id    name    Q1    Q2    Q3    Q4

1      a     1000  2000  3300  5000

2      b     3000  3000  3200  1500

3      c     2500  3500  2200  2500

4      d     1500  2500  1200  3500

5      e
```

下列查询实现按 quarter 统计所有 fruit 的销售量（sold），并将 quarter 字段列转行显示：

```
SELECT id,name,quarter,sold FROM fruit
UNPIVOT (sold FOR quarter IN(q1,q2,q3,q4));
```

result:

| id | name | quarter | sold |
|---|---|---|---|
| 1 | a | Q1 | 1000 |
| 1 | a | Q2 | 2000 |
| 1 | a | Q3 | 3300 |
| 1 | a | Q4 | 5000 |
| 2 | b | Q1 | 3000 |
| 2 | b | Q2 | 3000 |
| 2 | b | Q3 | 3200 |
| 2 | b | Q4 | 1500 |
| 3 | c | Q1 | 2500 |
| 3 | c | Q2 | 3500 |
| 3 | c | Q3 | 2200 |
| 3 | c | Q4 | 2500 |
| 4 | d | Q1 | 1500 |
| 4 | d | Q2 | 2500 |
| 4 | d | Q3 | 1200 |
| 4 | d | Q4 | 3500 |

系统执行查询操作过程中， unpivot_in 子句参数 (Q1,Q2,Q3,Q4) 为原表的列名，转换为目标表（新表）之后作为 unpivot_for 子句参数（quarter）的列值出现，即 unpivot_for 子句参数（quarter）为新表一部分列名；unpivot_in 子句参数的原表列值（指代销售数据）作为新表 unpivot_val_col 子句参数（sold）的列值，即 unpivot_val_col 子句参数（sold）为新表一部分列名；select 语句投影列（id,name,Q1,Q2,Q3,Q4）除去 unpivot_in 子句参数（Q1,Q2,Q3,Q4）以后，其余列（id,name）为新表的一部分列。在转换过程中，将其余列每行数据分别与 unpivot_in 子句各参数的列值一一对应并组合，其组合值作为新表的行值出现，转换完成。

以下查询增加了 include nulls 以显示 NULL 值，如下所示：

```
SELECT id,name,quarter,sold FROM fruit
UNPIVOT INCLUDE NULLS (sold FOR quarter IN(q1,q2,q3,q4));
```

result:

| id | name | quarter | sold |
|---|---|---|---|
| 1 | a | q1 | 1000 |
| 1 | a | q2 | 2000 |
| 1 | a | q3 | 3300 |
| 1 | a | q4 | 5000 |
| 2 | b | q1 | 3000 |
| 2 | b | q2 | 3000 |
| 2 | b | q3 | 3200 |
| 2 | b | q4 | 1500 |
| 3 | c | q1 | 2500 |
| 3 | c | q2 | 3500 |
| 3 | c | q3 | 2200 |
| 3 | c | q4 | 2500 |
| 4 | d | q1 | 1500 |
| 4 | d | q2 | 2500 |
| 4 | d | q3 | 1200 |
| 4 | d | q4 | 3500 |
| 5 | e | q1 | |
| 5 | e | q2 | |
| 5 | e | q3 | |
| 5 | e | q4 | |

## 对连接和子查询中的外部表的限制

当您使用在连接和子查询中的外部表时，受到下列限制：

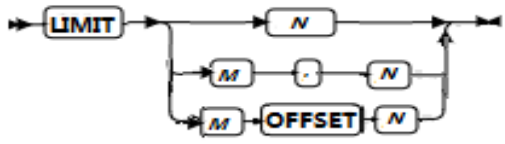
- 在查询中只有一个外部表是有效的。
- 该外部表不可为外连接中的外表。
- 对于不可转换成连接的子查询，您可使用在主查询中的外部表，但不可使用子查询中的。
- 您不可对外部表进行自连接。

要获取更多关于子查询的信息，请参阅您的 *GBase 8s 性能指南* 。

## LIMIT 子句

LIMIT 子句顺序获取结果集中某条记录开始的前 ***N*** 条记录。

语法




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***M*** | 返回的行数或结果集中的行计数 | 大于等于 0 的正整数 | 精确数值 |
| ***N*** | 返回的行数或结果集中的行计数 | 大于等于 0 的正整数 | 精确数值 |

注：M,N 元素含义随 LIMIT 子句的使用方式不同而不同。

用法

具有以下三种方式：

- LIMIT ***N*** ：指定取得结果集中前 ***N*** 条记录。
- LIMIT ***M,N*** ：指定取得结果集中第 ***M*** 条记录之后的 ***N*** 条记录。
- LIMIT ***M*** OFFSET ***N*** ：指定取得结果集中第 ***N*** 条记录之后的 ***M*** 条记录。

注意：LIMIT 不能与 TOP 同时出现在查询语句中。

例如，查询前 2 条记录：

```
SELECT emp_id, name FROM employee LIMIT 2;
```

查询第 3、4 条记录：

```
SELECT emp_id, name FROM employee LIMIT 2 OFFSET 2;
```

查询第 5、6、7 条记录：

```
SELECT emp_id, name FROM employee LIMIT 4,3;
```

以下查询返回符合员工编号（emp_id）大于 2 条件结果集中的第 3、4、5 条记录：

```
SELECT emp_id, name FROM employee where emp_id >'2'LIMIT 3 OFFSET 2;
```

## ONLY 关键字

如果 FROM 子句包括在类型表层级之内为超级表的永久表，则该查询从超级表及其缺省情况下的子表都返回符合条件的行，除非您指定 ONLY 关键字。

仅对于从超级表返回行的 SELECT 语句，您必须在 FROM 子句中包括 ONLY 关键字，紧排在超级表名称之前，且您必须将超级表的标识符或同义词括在小括号之内，如此例所示：

```
SELECT * FROM ONLY(super_tab);
```

此查询的数据源不包括 **super_tab** 的子表。

# 从集合变量选择

与“集合派生的表”段结合的 SELECT 语句允许您从集合变量选择元素。

“集合派生的表”段标识从其选择元素的集合变量。（请参阅 集合派生表。）

## *使用带有 SELECT 的集合变量*

要修改集合数据类型的列的内容，您可以各种方式使用带有集合变量的 SELECT 语句：

- 您可将集合列的内容（如果有的话）选择到集合变量之内。

  您可将列的数据类型指定为类型 COLLECTION 的集合变量（即，非类型集合变量）。

- 您可从集合变量选择内容来决定您可能想要更新的数据。

- 您可从集合变量选择内容 INTO 另一变量，以便更新确定的集合元素。

  INTO 子句为从集合变量选择的元素值标识变量。在 INTO 子句中的主变量的数据类型必须与相应的集合元素的数据类型相兼容。

- 您可使用 Collection 游标来从 GBase 8s ESQL/C 集合变量选择一个或多个元素。

  要获取更多包括对 SELECT 语句的限制的信息，请参阅 将游标与准备好的语句相关联。

- 您可使用 Collection 游标来从 SPL 集合变量选择一个或多个元素。

  要获取更多包括对 SELECT 语句的限制的信息，请参阅 使用 SELECT ... INTO 语句。

当要连接的表之一是集合时，FROM 子句不可指定连接。当该集合变量拥有您的集合派生的表时，此限制有效。另请参阅 集合派生表，以及在本章中的 INSERT、UPDATE 和 DELETE 语句描述。

# 从 Row 变量（ESQL/C）选择

SELECT 语句可包括“集合派生的表”段来从 **row** 变量选择一个或多个字段。

“集合派生的表”段标识要从其选择字段标识 **row** 变量。要获取更多信息，请参阅 集合派生表。

要选择字段：

1. 在您的 GBase 8s ESQL/C 程序中创建 **row** 变量。
2. 可选地，以字段值填充 **row** 变量。

   您可以 SELECT 语句（无“集合派生的表”段）将 ROW 类型列创建到 **row** 变量内。或者，您可以 UPDATE 语句和“集合派生的表”段将字段值插入到 **row** 变量之内。
3. 以 SELECT 语句和“集合派生的表”段从 **row** 变量选择行字段。
4. 一旦 **row** 变量包含正确的字段值，您可在表或视图名称上使用 INSERT 或 UPDATE 语句来将 **row** 变量的内容保存在命名的或未命名的行列中。

INTO 子句可指定主变量来把保持从 **row** 变量选择的字段值。

主变量的类型必须与字段的类型相兼容。例如，此代码片断将 **width** 字段值放到 **rect_width** 主变量之内。

```
EXEC SQL BEGIN DECLARE SECTION;
    ROW (x INT, y INT, length FLOAT, width FLOAT) myrect;
    double rect_width;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT rect INTO :myrect FROM rectangles
    WHERE area = 200;
EXEC SQL SELECT width INTO :rect_width FROM table(:myrect);
```

对 **row** 变量的 SELECT 语句有下列限制：

- 在 Projection 子句的选择列表中不允许有表达式。
- ROW 列不可在 WHERE 子句比较条件中。
- 如果 row 类型包含 opaque、distinct 或内建的数据类型，则 Projection 子句必须为星号(*)。
- 罗列在 Projection 子句中的列必须仅有未限定的名称。它们不可使用语法 ***database@server*:*table*.*column***。
- 不允许有下列子句：GROUP BY、HAVING、INTO TEMP、ORDER BY 和 WHERE。
- FROM 子句对进行连接没有任何规定。

您可以 UPDATA 语句的“集合派生的表”段修改 **row** 变量。（INSERT 和 DELETE 语句不支持“集合派生的表”段中的 **row** 变量。）

**row** 变量存储行的字段。然而，它与数据库行没有内在的连接。一旦 **row** 变量包含正确的字段值，那么您必须以下列 SQL 语句之一将该变量保存到 ROW 列之内：

- 要以 **row** 变量来更新表中的 ROW 列，请对表或视图名称使用 UPDATE 语句，并在 SET 子句中指定 **row** 变量。要获取更多信息，请参阅 更新 ROW 类型列。
- 要将行插入到 ROW 列，请对表或视图使用 INSERT 语句并在 VALUES 子句中指定 **row** 变量。请参阅 将值插入到 ROW 类型列内。

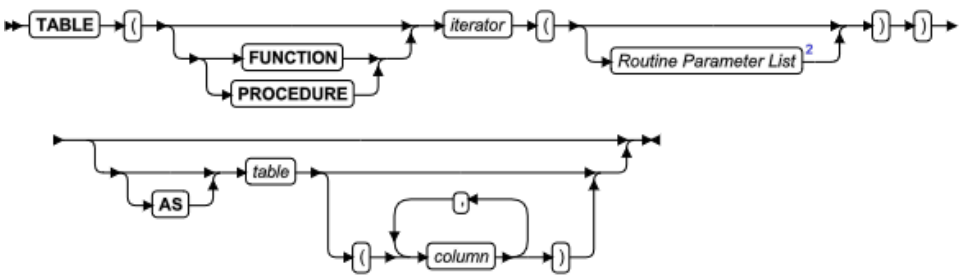
要获取如何使用 SPL row 变量的示例，请参阅 *GBase 8s SQL 教程指南*。要获取关于使用 GBase 8s ESQL/C **row** 变量的信息，请参阅在 *GBase 8s ESQL/C 程序员手册* 中对复合的数据类型的讨论。

## 迭代器函数

FROM 子句可包括对迭代器函数的调用，来为查询指定来源。迭代器***函数***是用户定义的功能，多次返回到它的调用 SQL 语句，每次返回至少一个值。

您可使用虚拟的表接口查询迭代器 UDR 的返回结果集。使用此语法来在 FROM 子句中调用迭代器函数：

迭代器




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 在此为 *table* 中的虚拟列声明的名称 | 在 *table* 中的 *column* 名称之中必须是唯一的，且不可包括限定符。 | 标识符 |
| *iterator* | 迭代器函数的名称 | 必须注册在该数据库中 | 标识符 |
| *table* | 在此为持有 *iterator* 结果集的虚拟表声明的名称 | 不可包括限定符 | 标识符 |

在早于 GBase 8s 10.5 的版本中，要求有关键字 FUNCTION（或 PROCEDURE）。在此版本中，这些对 SQL 的 ANSI/ISO 标准的关键字扩展是可选的，且无效。下列两个指定 fibGen() 作为迭代器函数的查询规范，是等同的：

```
SELECT * FROM TABLE FUNCTION ( fibGen(10));
SELECT * FROM TABLE ( fibGen(10));
```

仅可在此查询的上下文之内引用 ***table***。在 SELECT 语句终止之后，虚拟表不再存在。

列的数目必须与通过迭代器返回的值的数量相匹配。外部的函数可返回不超过一个值（但那可为集合数据类型）。SPL 例程可返回多个值。

然而，如果迭代器 ***table*** 函数的任何参数是聚集表达式，则数据库服务器发出错误 -595.

例如，要引用该 SELECT 语句的其他部分中的虚拟 *table*，在 WHERE 子句或 HAVING 子句中，您必须在 FROM 子句中声明它的名称和虚拟的列名称。如果您在 Projection 子句的 Select 列表中使用星号标记，则无需在 FROM 子句中声明 *table* 名称或 *column* 名称：

```
SELECT * FROM ...
```

要获取在查询中使用迭代器函数的更多信息和示例，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## 连接表的查询

如果 FROM 子句指定多个表引用，则该查询可从几个表或视图连接行。

***连接条件***指定来自要被连接的每一表的至少一列之间的关系。由于要对连接条件中的列进行比较，因此它们必须有可兼容的数据类型。

**注：** 在缺省情况下，数据库服务器连接表和视图所依的顺序不依赖于它们在 FROM 子句中被引用的顺序。要强制被连接的表的顺序与 FROM 子句顺序相匹配，则您可在 SELECT 关键字之后指定 ORDERED 优化器伪指令。要获取更多信息，请参阅 连接顺序伪指令 部分。

SELECT 语句的 FROM 子句可指定几种连接类型。

| FROM 子句关键字 | 相应的结果集 |
|---|---|
| CROSS JOIN | 笛卡尔积（所有可能的行的对） |
| INNER JOIN | 仅来自满足连接条件的 CROSS 的行 |
| LEFT OUTER JOIN | 一表的满足条件的行，和另一表的所有行 |
| RIGHT OUTER JOIN | 与 LEFT 相同，但两表的角色互换 |
| FULL OUTER JOIN | 来自两表的 INNER 连接的所有行的并集，以及在其他表中没有匹配的每一表的所有行（在其他的表的被选择的行中使用 NULL 值） |

在关系模型的文献中最后四个类别统称为“连接类型”；CROSS JOIN 忽略在被连接的表中特定的数据值，返回笛卡尔积作为它的结果集：每个可能的行的对，其中，每一对中的一行来自每一表。

在内（或简单的）连接中，结果仅包含满足连接条件的行的组合。外连接保留可能会被内连接废弃的那些行。在外连接中，结果包含满足连接条件的行的组合***以及***来自主表的可能会被废弃的那些行。在来自从表选择的列中，来自主表但在从表中没有相匹配的行的那些行包含 NULL 值。

GBase 8s 对于左外连接支持两种不同的语法：

- GBase 8s 扩展 OUTER 连接语法
- 符合 ANSI 的语法

对于外连接，数据库服务器的较早版本仅支持 GBase 8s 扩展语法。 GBase 8s 继续支持这种传统语法，但在 SQL 语言中对于连接查询使用符合 ISO/ANSI 标准的语法，提供更大的灵活性。然而，在视图定义中， GBase 8s 扩展语法不要求具体化的视图，因此它可能导致性能劣势。

如果您使用符合 ANSI 的语法来指定 FROM 子句中的连接，则对于同一查询块中的所有外连接，还必须使用符合 ANSI 的语法。因此，您不可仅以 OUTER 关键字开启另一外连接。例如，下列查询不是有效的：

```
SELECT * FROM customer, OUTER orders RIGHT JOIN cust_calls
        ON (customer.customer_num = orders.customer_num)
        WHERE customer.customer_num = 104);
```

这会返回错误，因为它尝试对外连接将 GBase 8s 扩展 OUTER 语法与符合 ANSI 的 RIGHT JOIN 语法组合在一起。

要了解 GBase 8s 扩展对 LEFT OUTER 外连接的语法，请参阅 GBase 8s 扩展外连接 部分。

# 符合 ANSI 的连接

对于连接的符合 ANSI 的语法支持这些连接规范：

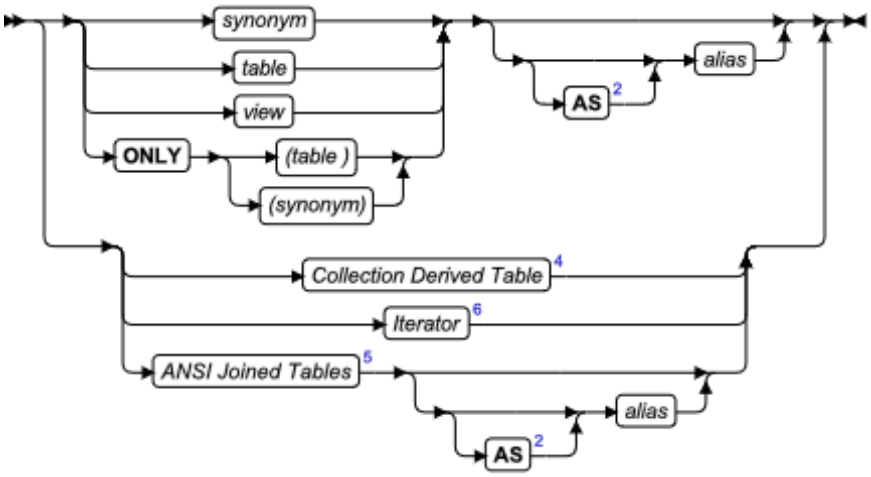
- 要使用 CROSS 连接、LEFT OUTER、RIGHT OUTER 或 FULL OUTER 连接，或 INNER（或简单的）连接，请参阅 ANSI INNER 连接。
- 要使用前连接过滤器，请参阅 使用 ON 子句。
- 要在 WEHERE 子句中使用一个或多个后连接过滤器，请参阅 指定后连接过滤器。
- 要使得外连接的主或从部分成为另一连接的结果集，请参阅 使用连接作为外连接的主部分或从部分。

**重要：** 当您在 GBase 8s 中创建新的查询时，请对连接使用符合 ANSI 的语法。

## *ANSI 表引用*

此图展示对表引用的符合 ANSI 的语法。

ANSI 表引用




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 在 SELECT 的作用域之内表或视图的临时名称 | 请参阅 AS 关键字。 | 标识符 |
| *synonym*、*table*、*view* | 从其检索数据的源 | 它指向的同义词和表或视图必须存在 | 数据库对象名 |

在此，ONLY 关键字的语义与在 GBase 8s 扩展“表引用”段的语义相同，如在 ONLY 关键字 中描述的那样。

当您为表引用声明别名（也称为***关联名称***）时，AS 关键字是可选的，如在 AS 关键字 中描述的那样，除非该别名与 SQL 关键字冲突。

在 Oracle 模式下，保持 GBase 8s 原有声明别名语法基础上，如果 SELECT 语句的 SQL 关键字包括 NAME、TEMP、ARRAY、LIST、REVERSE、CONTEXT、LENGTH、LOG 作为表或视图别名使用，可以不用关键字 AS 开始它的声明。

例如，在 SELECT 语句中查询使用关键字 temp 作为表 customer 别名：

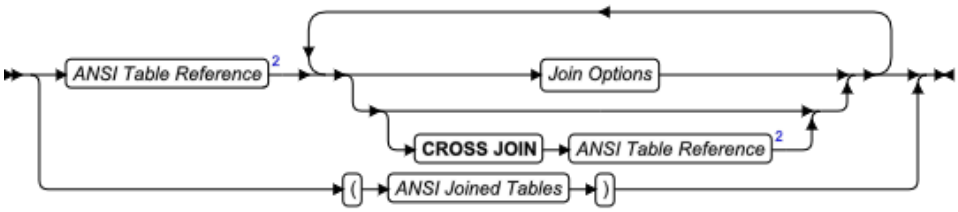
```
SELECT col FROM customer temp;
```

## *ANSI 连接的表*

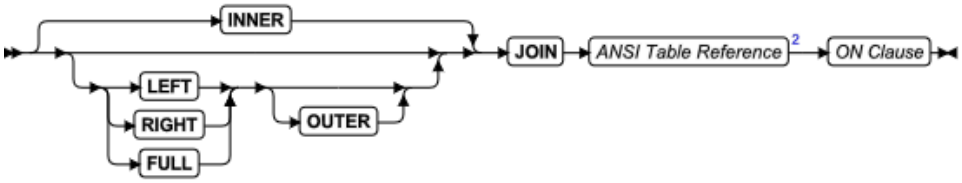
对连接表使用符合 ISO/ANSI 的语法，您可指定 INNER JOIN、CROSS JOIN、NATURAL JOIN、LEFT JOIN（或 LEFT OUTER JOIN）、RIGHT JOIN（或 RIGHT OUTER JOIN）和 FULL JOIN（FULL OUTER JOIN）关键字。在符合 ANSI 的外连接中，OUTER 关键字是可选的。

这是对于指定的内连接和外连接的符合 ANSI 的语法。

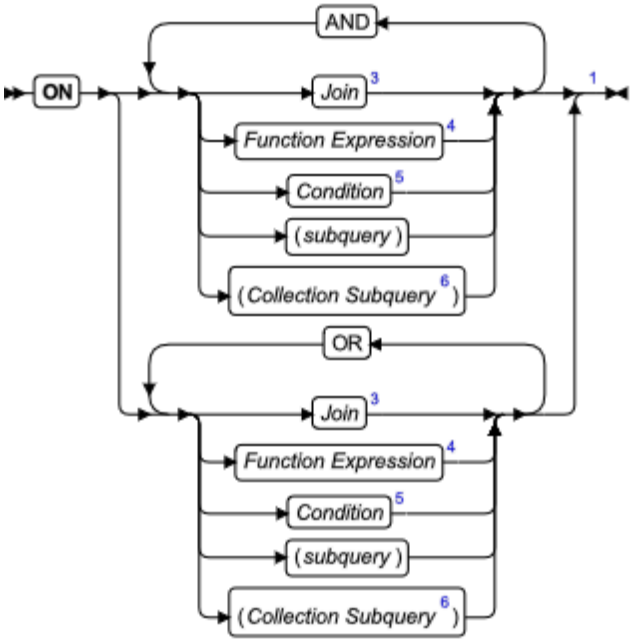
ANSI 连接的表




Join 选项




ON 子句



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *subquery* | 嵌入的查询 | 不可包含 FIRST 或 ORDER BY 子句 | SELECT 语句 |

在同一查询块中，对于所有外连接您必须使用相同的连接语法形式（或 GBase 8s 扩展，或符合 ANSI）。当您使用符合 ANSI 的连接语法时，您还必须在 ON 子句中指定连接条件。

如果“ANSI 连接的表”段紧跟在另一连接规范之后，则必须将它括在圆括号之间。 例如，下列两个查询的第一个返回错误；第二个查询是有效的：

```
SELECT * FROM (T1 LEFT JOIN T2) CROSS JOIN T3 ON (T1.c1 = T2.c5)
        WHERE (T1.c1 < 100);      -- Ambiguous order of operations;

SELECT * FROM (T1 LEFT JOIN T2 ON (T1.c1 = T2.c5)) CROSS JOIN T3
        WHERE (T1.c1 < 100);      -- Unambiguous order of operations;
```

下列有效的查询指定外部 SELECT 语句的 FROM 子句之内的表表达式的嵌套的 LEFT OUTER 连接：

```
SELECT * FROM
        ( (SELECT C1,C2 FROM T3) AS VT3(V31,V32)
        LEFT OUTER JOIN
        ( (SELECT C1,C2 FROM T1) AS VT1(VC1,VC2)
        LEFT OUTER JOIN
        (SELECT C1,C2 FROM T2) AS VT2(VC3,VC4)
        ON VT1.VC1 = VT2.VC3)
        ON VT3.V31 = VT2.VC3);
```

**ANSI CROSS 连接**

CROSS 关键字指定笛卡尔积，返回包括从每一被连接的表的一行的所有可能的成对组合。

**ANSI INNER 连接**

要使用符合 ANSI 的语法创建内（或简单的）连接，请以 JOIN 或 INNER JOIN 关键字指定该连接。如果您仅指定 JOIN 关键字，则数据库服务器缺省地创建隐式的内连接。内连接返回有一个或多个在其他一个表（或多个表）中的相匹配的行的表中的所有行。废弃不相匹配的行。

**ANSI LEFT OUTER 连接**

LEFT 关键字指定将第一个表引用处理作为该连接中的主表的连接。在左外连接中，外连接的从部分出现在起始该外连接规范的关键字的右边。结果集包括 INNER 连接返回的所有行，加上可能已从从表废弃的所有行。

**ANSI RIGHT OUTER 连接**

RIGHT 关键字指定连接，该连接处理第二个表引用作为连接中的从表。在右外连接中，外连接的从部分出现在起始该外连接规范的关键字的左边。结果集包括 INNER 连接返回的所有的行，加上可能已从从表废弃了的所有行。

对横向派生的表的相关联引用不是 ANSI RIGHT OUTER 连接中有效的表引用。例如，下列查询失败，因为在派生的表的 ON 子句中的相关联的引用 t1.c1 是不受支持的横向相关联：

```
SELECT * FROM t1 RIGHT JOIN LATERAL
        (SELECT * FROM t2 JOIN t3
            ON t2.c1 = t1.c1) AS X ON 1=1;
```

**ANSI FULL OUTER 连接**

FULL 关键字指定连接，在该连接的结果集中包括其连接条件为真的笛卡尔积的所有行，加上来自与连接条件不匹配的每一表的所有行。

在符合 ANSI 的连接中，该连接在 FROM 子句中指定 LEFT、RIGHT 或 FULL 关键字，OUTER 关键字是可选的。

对横向的派生表的相关联引用不是 ANSI FULL OUTER 连接中有效的表引用。例如，下列查询失败，因为在派生的表的 ON 子句中的相关联引用 t1.c1 是不受支持的横向的相关联：

```
SELECT * FROM t1 FULL JOIN LATERAL
        (SELECT * FROM t2 JOIN t3
        ON t2.c1 = t1.c1) AS X ON 1=1;
```

忽略您为符合 ANSI 的连接的查询指定的连接模式优化程序伪指令，但将伪指令罗列在解释输出文件中的 ***Directives Not Followed*** 之下。

## *使用 ON 子句*

使用 ON 子句来指定连接条件和任何表达式作为可选的连接过滤器。

下列来自 stores_demo 数据库的示例展示在 ON 子句中的连接条件如何组合 customer 与 orders 表：

```
SELECT c.customer_num, c.company, c.phone, o.order_date
        FROM customer c LEFT JOIN orders o
        ON c.customer_num = o.customer_num;
```

下表展示连接了的 customer 与 orders 表的一部分。

| customer_num | company | phone | order_date |
|---|---|---|---|
| 101 | All Sports Supplies | 408-789-8075 | 05/21/2008 |
| 102 | Sports Spot | 415-822-1289 | NULL |
| 103 | Phil’s Sports | 415-328-4543 | NULL |
| 104 | Play Ball! | 415-368-1100 | 05/20/2008 |
| — | — | — | — |

在外连接中，您在 ON 子句中指定的连接过滤器（表达式）决定将从表的哪些行连接到主（或外）表。根据定义，主表返回在连接了的表中的所有行。即，在 ON 子句中的连接过滤器对主表不起作用。

如果 ON 子句在主表上指定连接过滤器，则数据库服务器仅将那些满足连接过滤器条件的主表行连接到从表中的行。连接的结果包含来自主表的所有行。那些不满足连接过滤器条件的主表中的行，为从列以 NULL 值扩展。

下列来自 stores_demo 数据库的示例展示 ON 子句中连接过滤器的作用：

```
SELECT c.customer_num, c.company, c.phone, o.order_date
        FROM customer c LEFT JOIN orders o
        ON c.customer_num = o.customer_num
        AND c.company <> "All Sports Supplies";
```

包含 All Sports Supplies 的行保留在连接的结果中。

| customer_num | company | phone | order_date |
|---|---|---|---|
| 101 | All Sports Supplies | 408-789-8075 | NULL |
| 102 | Sports Spot | 415-822-1289 | NULL |
| 103 | Phil’s Sports | 415-328-4543 | NULL |
| 104 | Play Ball! | 415-368-1100 | 05/20/2008 |
| — | — | — | — |

在 **orders** 表中，即使客户编号 101 的订单日期是 05/21/2008，放置连接过滤器（`c.company <> "All Sports Supplies"`）的作用是阻止在主表 **customer** 中的此行被连接到从表 **orders**。相反，**order_date** 的 NULL 值会扩展到 All Sports Supplies 的行。

将连接过滤器应用到外连接的从部分中的基础表，可提升性能。要获取更多信息，请参阅您的 *GBase 8s 性能指南* 。

**指定后连接过滤器**

当您使用 ON 子句来指定连接时，您可使用 WHERE 子句作为后连接过滤器。数据库服务器将 WHERE 子句的后连接过滤器应用到外连接的结果。

下列示例展示后连接过滤器的使用。此查询从 stores_demo 数据库返回数据。假设您想要确定目录中的哪些项没有被订购。下一查询从 catalog 与 items 表创建该数据的外连接，然后从特定的制造商（HRO）确定哪个目录项尚未售出：

```
SELECT c.catalog_num, c.stock_num, c.manu_code, i.quantity
        FROM catalog c LEFT JOIN items i
        ON c.stock_num = i.stock_num AND c.manu_code = i.manu_code
        WHERE i.quantity IS NULL AND c.manu_code = "HRO";
```

WHERE 子句包含后连接过滤器，定位目录中尚未被出售的 HRO 项的那些行。

当您在对连接的主部分或从部分中的基础表应用后连接过滤器时，您可提升性能。要了解更多信息，请参阅您的 *GBase 8s 性能指南* 。

**使用连接作为外连接的主部分或从部分**

以 ANSI 连接语法，您可嵌套连接。您可使用连接作为外连接或内连接的主部分或从部分。

假设您想要修改先前的查询（后连接过滤器示例）来获得更多信息，帮助您确定是否继续保留该目录中的每一未售出项。您可修改该查询来包括来自 stock 表的信息，以便能看到带有其成本的每一未售出项的简短描述：

```
SELECT c.catalog_num, c.stock_num, s.description, s.unit_price,
        s.unit_descr, c.manu_code, i.quantity
        FROM (catalog c INNER JOIN stock s
        ON c.stock_num = s.stock_num
        AND c.manu_code = s.manu_code)
        LEFT JOIN items i
        ON c.stock_num = i.stock_num
        AND c.manu_code = i.manu_code
        WHERE i.quantity IS NULL
        AND c.manu_code = "HRO";
```

在此示例中，在 **catalog** 与 **stock** 表之间的内连接形成带有 **items** 表的外连接的主部分。

要了解外连接的附加示例，请参阅 *GBase 8s SQL 教程指南*。

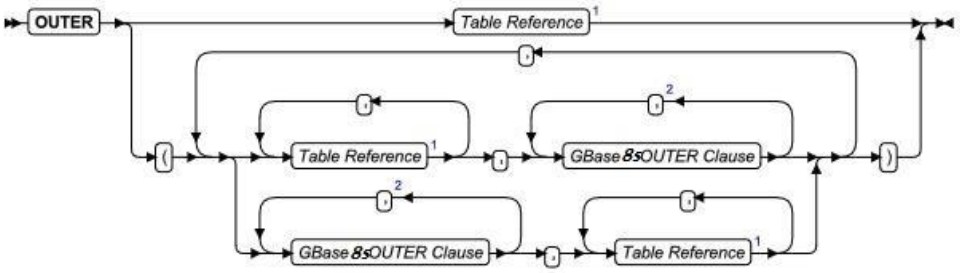
## GBase 8s 扩展外连接

### *OUTER 关键字*

对于外连接的 GBase 8s 扩展语法始于隐式的左外连接。即，您以 OUTER 关键字开始 GBase 8s 扩展外连接。

这是 GBase 8s 扩展 OUTER 子句的语法。

GBase 8s OUTER 子句




下列示例使用 OUTER 关键字来创建罗列所有客户及其订单的外连接，不论他们是否已下订单：

```
SELECT c.customer_num, c.lname, o.order_num FROM customer c,
        OUTER orders o WHERE c.customer_num = o.customer_num;
```

此带有在 **orders** 表中相匹配的行从 **customer** 表返回所有行。如果在 **orders** 表中没有客户的记录出现，则对那些有 NULL 值的客户返回的 **order_num** 列。

如果您有复合的外连接，即，该查询有多个外连接，则您必须或者嵌入附加的外连接或者在圆括号中连接，如语法图所示，或在 WHERE 子句中的主表与每一从表之间创建连接条件或关系。

当 WHERE 子句中的表达式或条件关系到两个从表时，您必须在 FROM 子句中使用圆括号将连接的表括起来，以强调主-从关系，如此例中所示：

```
SELECT c.company, o.order_date, i.total_price, m.manu_name
        FROM customer c,
        OUTER (orders o, OUTER (items i, OUTER manufact m))
        WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num
        AND i.manu_code = m.manu_code;
```

当您在 FROM 子句中省略括起从表的圆括号时，您必须在 WHERE 子句中的主表与每一从表之间建立连接条件。如果连接条件在两个从表之间，则查询失败。

然而，下列示例成功地返回结果

- 将主表 customer 与子表 orders 连接
- 并将主表 customer 与从表 cust_calls 连接：

```
SELECT c.company, o.order_date, c2.call_descr
        FROM customer c, OUTER orders o, OUTER cust_calls c2
        WHERE c.customer_num = o.customer_num
        AND c.customer_num = c2.customer_num;
```

*GBase 8s SQL 教程指南* 有复合的外连接的示例。

**对 GBase 8s 扩展外连接的限制**

如果您对于外连接使用此 GBase 8s 扩展语法，则对同一 SELECT 语句使用所有下列限制：

- 您必须对单个查询块中的所有外连接使用 GBase 8s 扩展语法。
- 您必须在 WHERE 子句中包括连接条件。
- 您不可以 LEFT JOIN 或 LEFT OUTER JOIN 关键字起始另一外连接。
- 您不可定义横向的表引用或包括 LATERAL 关键字。
- 在 GBase 8s 扩展外连接之内，“表引用”语法段不可包括在同一 SELECT 语句中声明的横向的表引用。

## *(+) 操作符*

本版本可以使用（+）形式表现外连接，即用 (+) 来表示两表的连接关系。可以在 WHERE 条件中使用“(+)”形式的外连接语法来实现多表连接。

例如，在以下示例中，使用 (+) 形式的外连接，连接了 **orders** 表与 **customer** 表。

```
SELECT c.customer_num, c.company, o.order_num,
FROM orders o,customer c
WHERE c.customer_num = o.customer_num(+);
```

此外连接罗列客户的公司名称、编号和所有相关联的订单编号，如果该客户已下了订单的话。如果没有，仍罗列公司名称，且为订单编号返回 NULL 值。

此外，WHERE 子句的多个连接条件中既可以包含左外连接又可以包含右外连接。

例如，在以下示例中，**orders** 表的使用不同的字段与 **customer** 表进行左外连接，与 **items** 表进行右外连接：

```
SELECT o.customer_num, c. company, i.order_num
FROM orders o,customer c, items i
WHERE o.customer_num = c.customer_num(+)
AND   o.order_num(+)=i.order_num;
```

其次， WHERE 条件句中可同时包含多个连接条件和多个过滤条件。例如:

```
SELECT o.customer_num, c.company, i.order_num,
FROM orders o,customer c, items i
WHERE c.customer_num = o.customer_num(+)
AND   o.order_num(+)=i.order_num
AND (o.customer_num='101'or i.stock_num >10 );
```

此语句在上一示例的基础上添加了(o.customer_num=’101’ or i.stock_num >10 ）的过滤条件。请注意，使用时，OR 关系两端的表达式必须用括号包围。

另外，使用 ”(+)” 操作符表示外连接具有以下限制：

1）『=』两边不能同时设置(+)。

2）多个连接条件之间必须使用 AND 关键字连接，不支持其它关键字。

3）(+) 操作符只适用于列，不能用于表达式，并且不能与 OR 或 IN 运算符一起使用。

4）不能使用 (+) 操作符连接同一个表，可以通过采用不同别名方式进行自连接。

5）如果在一个单独的查询块中使用了 (+) 操作符，则所有的外连接都必须使用 (+) 操作符的形式。

6）单次查询中任意两张表不能既左外连接后右连外接。

7）如果使用了 (+) 操作符的外连接形式，则在一个查询块中的其它外连接中不能使用 LEFT JOIN 或者 LEFT OUTER JOIN 形式。

## GRID 子句

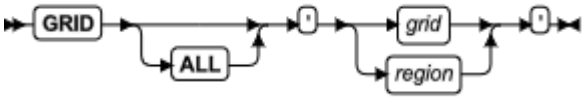
使用 GRID 子句来指定跨服务器查询的作用域，该查询的数据源是作为 GBase 8s 网格的节点的数据库服务器的表。

除非会话连接到现有的网格之内的数据库，GRID 子句才是有效的。可通过使用 Enterprise Replication 设施的适当的 cdr 命令和 ifx_grid 例程创建网格。

此语法是对 SQL 的 ANSI/ISO 标准的扩展。

SELECT 语句的 GRID 子句支持下列语法：

GRID 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *grid* | 此查询所在作用域之内的网格的名称 | 必须存在，且必须通过 cdr define grid 命令定义 | 标识符 |
| *region* | 此查询所在作用域中的区域的名称 | 必须存在且必须通过 cdr define region 命令定义 | 标识符 |

**用法**

任何显式地或隐式地包括 GRID 子句的 SELECT 语句都称为*网格查询*。网格查询的结果，是来自在每个网格服务器中的带有相同名称和相同模式的跨表的 FROM 子句中的每一表的逻辑 UNION 或 UNION ALL 的符合条件的行。此联合可包括该网格中跨所有节点的表，或跨这些网格节点的子集，称为*区域*。

**注：** 网格查询的 FROM 子句指定的表必须都有相同的模式，且必须满足此主题标识的其他要求。由于这些限制，所以不是所有的 GBase 8s 网格都可支持网格查询。

**可选的 ALL 关键字**

如果可选的 ALL 关键字紧跟在 GRID 关键字之后，则网格查询的结果是逻辑的 UNION ALL，意味着该网格查询的结果集可包括重复的行。否则，如果您省略 ALL 关键字，则仅从每一参与的服务器的结果的逻辑 UNION 返回 distinct 值。

**网格查询的 SET ENVIRONMENT 语句选项**

对 SET ENVIRONMENT 语句的两个选项可定义缺省的 GRID 子句，以便于任何不带有 GRID 子句的后续的 SELECT 语句被解释为包括该缺省的 GRID 子句的网格查询：

SET ENVIRONMENT SELECT_GRID

此语句可指定网格或区域作为后续的网格查询的缺省的作用域，这些网格查询返回唯一的符合条件的行的联合。GRID 子句可省略网格查询的网格或区域名称，为指定的缺省节点返回 UNION 结果。

SET ENVIRONMENT SELECT_GRID_ALL

此语句可指定网格或区域作为后续的网格查询的缺省的作用域，这些网格查询返回所有符合条件的行的联合，包括重复的行。GRID 子句可省略网格查询的网格或区域名称，为指定的缺省节点返回 UNION ALL 结果。

在启用 SET ENVIRONMENT 语句的这些选项之一时，SQL 解析器将当前的缺省 GRID 子句应用到不包括显式的 GRID 子句的会话中的每个 SELECT 语句。在同一时间上的同一会话期间，只有一个缺省的 GRID 子句可起作用。当使用 SET ENVIRONMENT 语句来设置其他关键字选项使选项生效时，或当为不同的网格或区域重置同一关键字选项时，会禁用先前设置的缺省值。

您还可通过发出这些 SQL 语句之一来禁用缺省的 GRID 子句：

```
SET ENVIRONMENT SELECT_GRID DEFAULT;_
SET ENVIRONMENT SELECT_GRID_ALL DEFAULT;
```

上述每一语句都阻止数据库服务器将当前的会话中的每个后续的查询都解释为网格查询。除非您在同一会话中定义新的缺省的 GRID 子句，否则，任何后续的 SELECT 语句必须包括显式的 GRID 子句来作为网格查询运行。

在 UNION 查询的 SELECT_GRID 会话环境选项或 UNION ALL 查询的 SELECT_GRID_ALL 会话环境选项已指定了缺省的网格或区域作为当前会话中的网格查询的作用域时， 您可省略那个网格或区域所在节点上的网格查询中的 GRID 子句，如下例所示。在此，tab1 和 tab2 是在网格的 region_03 子集之内的每个网格服务器上的有相同的模式、语言环境和代码集的表：

```
SET ENVIRONMENT SELECT_GRID 'region_03'
        SELECT * FROM tab1;
        SELECT * FROM tab2;
```

执行上述两个查询，就如同您已显式地指定了此 GRID 子句：

```
SELECT * FROM tab1 GRID 'region_03';
SELECT * FROM tab2 GRID 'region_03';
```

在同一时点，对于当前的用户会话，对于 SET ENVIRONMENT 语句仅可启用一个 SELECT_GRID 和 SELECT_GRID_ALL 会话环境选项。当其中一个选项生效时，使用 SET ENVIRONMENT 语句来设置其他的关键字选项，或重置不同的网格或区域的同一关键字选项，可禁用先前设置的缺省值。

下列 SQL 语句通过定义不同的缺省的 GRID 子句，组合来自不同区域中的参与的网格服务器的 UNION ALL 结果，将先前的缺省的 GRID 子句替换为网格的 **region_04** 子集：

```
SET ENVIRONMENT SELECT_GRID_ALL 'region_04'
        SELECT * FROM tab1;
        SELECT * FROM tab2;
```

将会执行那两个查询，就如同您已制定了此 GRID 子句：

```
SELECT * FROM tab1 GRID ALL 'region_04';
SELECT * FROM tab2 GRID ALL 'region_04';
```

在缺省情况下，如果发出网格查询的数据库服务器不可连接到网格或区域之内的一个或多个节点，显式的或缺省的 GRID 子句指定该网格或区域，则网格查询失败。即使指定的网格或区域中的有些网格服务器不可用，SET ENVIRONMENT 语句仍可启用的另一会话环境变量可从网格查询返回部分的结果：

```
SET ENVIRONMENT GRID_NODE_SKIP
```

当一个或多个网格服务器不可用时，此语句可使得网格查询的处理得以继续。

如果您发出 SQL 语句

```
SET ENVIRONMENT GRID_NODE_SKIP ON;
```

，则数据库服务器不论任何不可用的节点，并从参与的网格服务器返回符合条件的行。您可通过调用 ifx_gridquery_skipped_nodes() 函数来表示任何跳过的节点。

另一函数，ifx_gridquery_skipped_node_count()，可用于检测跳过了多少个节点。要获取更多关于这些函数的信息，请参阅 *GBase 8s Enterprise Replication 指南*。

**在网格查询的 FROM 子句中的表**

在网格查询的 FROM 子句中，仅永久数据库表是有效的。必须通过运行 `cdr change gridtable` 命令来将它们定义为网格表。

不支持下列表对象：

- 表上的同义词或视图，除了 **sysmaster** 数据库中的表之外
- CREATE EXTERNAL TABLE 语句定义的表对象
- 由数据库服务器或网格服务器的名称限定的表
- 正在其上执行并发的 ALTER TABLE、ALTER FRAGMENT 或 ALTER INDEX 操作的表
- 与参与的网格服务器的数据库中同一名字的其他表有不同的模式的表
- 不是以相同的数据库语言环境和代码集创建的数据库中的表
- 跨参与该网格查询的所有数据库，其 SQL_LOGICAL_CHAR 配置参数或 DELIMIDENT 或 GL_USEGLU 环境变量的设置不相同的数据库中的表。

此外，网格查询的 projection 列表不可包括在跨服务器查询中其数据类型不被支持的任何列或表达式。不被支持的数据类型包括所有复合的或大对象类型，以及一些用户定义的类型（UDT）和 opaque 类型。

适用于同一 GBase 8s 示例的跨数据库的分布式 DML 操作中的 DISTINCT 数据类型的相同限制，也适用于网格查询中的 DISTINCT 数据类型。要获取在分布式查询中有效的数据类型的讨论，请参阅主题 跨服务器事务中的数据类型 and 分布式操作中的 DISTINCT 类型。

**对网格查询的附加的限制**

执行网格查询的用户必须是该网格或区域之内所有节点上的有效用户。

网格查询不可为包含对其外查询引用的子查询。

在其规范之中，网格查询不可引用的任何下列之一：

- 子查询（但网格查询自身可为其不引用的外查询的子查询）
- 跨网格服务器的连接操作
- 导致跨服务器连接的连接请求
- 在所有参与的网格服务器上都不存在的过程或函数。

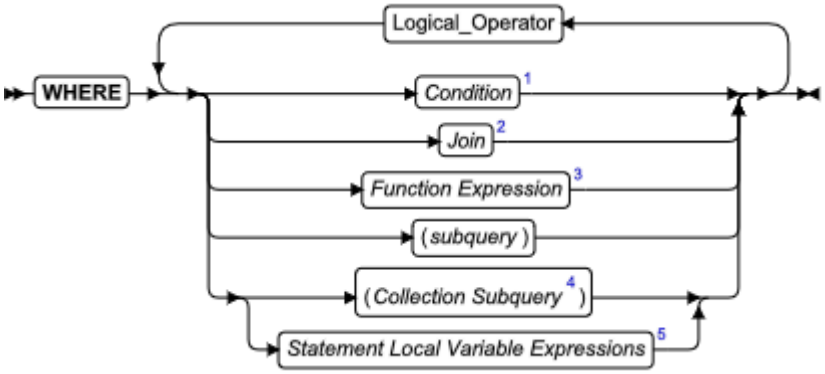
在网格查询块中，UNION 或 UNION ALL 集合运算符都不是有效的，INTERSECT、MINUS 或 EXCEPT 集合运算符也都不是有效的。

GRID 子句不应包括在网格上下文外部的 SELECT 语句中。要获取更多关于网格的信息，请参阅 SET ENVIRONMENT 语句 和 *GBase 8s Enterprise Replication 指南*。

## SELECT 的 WHERE 子句

WHERE 子句为 GBase 8s 扩展连接指定连接条件，为符合 ANSI 的连接指定后连接过滤器，和对数据指定搜索条件。

WHERE 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *Logical_Operator* | 两个条件的组合 | 有效的选项为***逻辑并***（= OR ***或*** OR NOT）或***逻辑与***（= AND ***或*** AND NOT） | 带有 AND 或 OR 的条件 |
| *subquery* | 内嵌的查询 | 不可包括 FIRST 或 ORDER BY 关键字 | SELECT 语句 |

## 在 WHERE 子句中使用条件

在 WHERE 子句中，您可使用这些简单的条件或比较：

- 关系运算符条件
- IN 或 BETWEEN . . . AND
- IS NULL 或 IS NOT NULL
- LIKE 或 MATCHES

您还可在 WHERE 子句中使用 SELECT 语句；这称为子查询。在子查询中，下列 WHERE 子句运算符是有效的：

- IN 或 EXISTS
- ALL、ANY 或 SOME

要获取更多信息，请参阅 条件。

在 WHERE 子句中，聚集函数不是有效的，除非它是子查询的一部分，或是来源自父查询的相关的列上，且 WHERE 子句在 HAVING 子句之内的子查询中。

### *关系运算符条件*

如果关系运算符的每一边的表达式满足该表达式指定的关系，则关系运算符条件是满足的。下列语句使用大于（>）和等于（=）关系运算符：

```
SELECT order_num FROM orders
        WHERE order_date > '6/04/08';
        SELECT fname, lname, company
        FROM customer
        WHERE city[1,3] = 'San';
```

'San' 需要加上单引号，因为该子字符串来自字符列。请参阅 关系运算符条件。

**WHERE 子句中的空格字符串和空字符串**

对于 LVARCHAR、NVARCHAR 或 VARCHAR 列，指定列值等于空字符串的带有 WHERE 子句的查询（

```
WHERE varlength_col = ''
```

）返回的结果集，与其中的 WHERE 子句指定等于空格（ASCII 32）字符的字符串的同一查询是一样的。

例如，如果 varlength_col 是类型 VARCHAR、NVARCHAR 或 LVARCHAR，则下列 WHERE 子句示例在功能上完全等同于指定等于空字符串的 WHERE 子句：

```
WHERE varlength_col = ' '
WHERE varlength_col = '  '
WHERE varlength_col = '   '
```

因此，对于内建的可变长度字符数据类型，在空字符串与全由一个或多个空白字符组成的字符串之间，没有区别。（然而，请注意，查询过滤器

```
WHERE varlength_col IS NULL
```

不等同于先前的 WHERE 子句示例，且如果 varlength_col 值为 NULL，则返回一个不同的结果集。

## *IN 条件*

当在该关键字右边的值的列表中包括 IN 关键字左边的表达式时，IN 条件是满足的。

下列示例展示 IN 条件：

```
SELECT lname, fname, company FROM customer
        WHERE state IN ('CA','WA', 'NJ');
        SELECT * FROM cust_calls
        WHERE user_id NOT IN (USER );
```

要获取更多信息，请参阅 IN 子查询。

## *BETWEEN 条件*

当 BETWEEN 左边的值在 BETWEEN 右边的两个值的范围之内时，BETWEEN 条件是满足的。下列示例中的前两个查询在 BETWEEN 关键字之后使用文字值。第三个查询使用内建的 CURRENT 函数和一个文字间隔来搜索当天与七天前之间的日期。

```
SELECT stock_num, manu_code FROM stock
        WHERE unit_price BETWEEN 125.00 AND 200.00;
        SELECT DISTINCT customer_num, stock_num, manu_code
        FROM orders, items
        WHERE order_date BETWEEN '6/1/07' AND '9/1/07';
        SELECT * FROM cust_calls WHERE call_dtime
        BETWEEN (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT;
```

要获取更多信息，请参阅 BETWEEN 条件。

## *使用 IS NULL 和 IS NOT NULL 条件*

如果指定的 ***column*** 包含 NULL 值，或如果指定的 ***expression*** 求值为 NULL，则 IS NULL 条件是满足的。

如果您使用 IS NOT NULL 谓词，则当 *column* 包含非 NULL 的值时，或当 *expression* 求值不为 NULL 时，该条件是满足的。下列示例选择尚未支付的订单的订单编号和客户编号：

```
SELECT order_num, customer_num FROM orders
        WHERE paid_date IS NULL;
```

要获取 IS NULL 和 IS NOT NULL 运算符的完整描述，请参阅 IS NULL 和 IS NOT NULL 条件。

## *LIKE 或 MATCHES 条件*

如果下列任一为真，即符合 LIKE 或 MATCHES 条件：

- LIKE 或 MATCHES 关键字前面的列的值与加引号的字符串指定的模式相匹配。您可在字符串中使用通配符。
- LIKE 或 MATCHES 关键字前面的列的值与由跟在 LIKE 或 MATCHES 关键字之后的列指定的模式相匹配。在条件中，右边的列值作为匹配模式。

在 LIKE 或 MATCHES 条件中指定的列可以为简单的字符数据类型（如 CHAR、LVARCHAR、NCHAR、NVARCHAR 或 VARCHAR）或数值型数据类型（如 INTEGER、SMALLINT、DECIMAL 或 NUMERIC(p,s)、FLOAT、SMALLFLOAT、BIGINT、INT8、BIGSERIAL、SERIAL、SERIAL8、MONEY ）。

下列 SELECT 语句返回 customer 表中 VARCHAR 类型的 lname 列以文字字符串 'Baxter' 开头的所有行。由于该字符串为文字字符串，因此该条件区分大小写。

```
SELECT * FROM customer WHERE lname LIKE 'Baxter%' ;
```

下列 SELECT 语句返回 SG_DEV_ACLINE 表中的 length 列包含数字 '261' 的所有行。length 列是该表的一数值型字段

```
SELECT * FROM SG_DEV_ACLINE WHERE length LIKE '%261%' ;
```

以下 SELECT 语句返回 customer 表中 lname 列的值与 fname 列的值相匹配的所有行：

```
SELECT * FROM customer WHERE lname LIKE fname;
```

下列示例使用反斜线（\）作为缺省的转义字符。通过 DEFAULTESCCHAR 配置参数或 DEFAULTESCCHAR 会话环境选项设置缺省的转义字符。

以下示例使用带有通配符的 LIKE 条件。第一条 SELECT 语句查找所有球类的库存商品。第二条 SELECT 语句查找所有包含百分号（%）的公司名称。反斜杠（\）用作百分号（%）通配符的缺省转义字符。第三条 SELECT 语句使用 ESCAPE 选项和 LIKE 条件从 customer 表中检索 company 列包含百分号（%）的行。z 用作百分号（%）的转义字符：

```
SELECT stock_num, manu_code FROM stock
        WHERE description LIKE '%ball';
SELECT * FROM customer WHERE company LIKE '%\%%';
SELECT * FROM customer WHERE company LIKE '%z%%' ESCAPE 'z';
```

下列示例在 SELECT 语句中使用带有通配符的 MATCHES。第一条 SELECT 语句查找所有球类的库存商品。第二条 SELECT 语句查找所有包含星号（*）的公司名称。反斜杠（\）用作文字星号（ *）字符的缺省转义字符。第三条语句使用 ESCAPE 选项和 MACHES 条件从 customer 表中检索 company 列包含星号（*）的行。指定 z 字符作为星号（*）字符的转义字符。：

```
SELECT stock_num, manu_code FROM stock
        WHERE description MATCHES '*ball';
SELECT * FROM customer WHERE company MATCHES '*\**';
SELECT * FROM customer WHERE company MATCHES '*z**' ESCAPE 'z';
```

要获取关于 LIKE 或 MATCHES 表达式中支持的运算对象的数据类型的信息，请参阅主题 LIKE 和 MATCHES 条件。

## *IN 子查询*

随同 IN 子查询，可返回多个满足 IN 或 NOT IN 条件的行，但仅可返回一列。

此示例展示在 SELECT 语句中 NOT IN 子查询的使用：

```
SELECT DISTINCT customer_num FROM orders
    WHERE order_num NOT IN
        (SELECT order_num FROM items
            WHERE stock_num = 1);
```

要获取附加的信息，请参阅 IN 条件。

## *EXISTS 子查询*

从 EXISTS 子查询，可返回那些在一个或多个列中的满足 EXISTS 条件的行。（类似地，NOT EXISTS 子查询可返回在一列或多列中满足 NOT EXISTS 条件的那些行。）

下列带有 NOT EXISTS 子查询的 SELECT 语句返回那些从未被订购的每项的库存编号和生产商代码（因而未罗列在 **items** 表中）。

在此 SELECT 语句中使用 NOT EXISTS 子查询是恰当的，因为您需要相关联的子查询来同时测试 items 表中的 stock_num 和 manu_code。

```
SELECT stock_num, manu_code FROM stock
        WHERE NOT EXISTS
        (SELECT stock_num, manu_code FROM items
        WHERE stock.stock_num = items.stock_num AND
        stock.manu_code = items.manu_code);
```

如果您在列名称的位置在子查询中使用 SELECT *，则前一示例同样奏效，因为您正在测试一行或多行是否存在。

要获取附加的信息，请参阅 EXISTS 子查询条件。

### *ALL、ANY、SOME 子查询*

下列示例返回所有包含一项的所有订单的订单编号，该项的总价大于订单编号 1023 中每项的总价。第一个 SELECT 使用 ALL 子查询，第二个 SELECT 通过使用 MAX 聚集函数产生相同的结果。

```
SELECT DISTINCT order_num FROM items
        WHERE total_price > ALL (SELECT total_price FROM items
        WHERE order_num = 1023);

        SELECT DISTINCT order_num FROM items
        WHERE total_price > SELECT MAX(total_price) FROM items
        WHERE order_num = 1023);
```

下列 SELECT 语句返回所有包含一项的所有订单的订单编号，该项的总价大于订单编号 1023 中至少一项的总价。第一个 SELECT 语句使用 ANY 关键字，而第二个 SELECT 语句使用 MIN 聚集函数：

```
SELECT DISTINCT order_num FROM items
        WHERE total_price > ANY (SELECT total_price FROM items
        WHERE order_num = 1023);

        SELECT DISTINCT order_num FROM items
        WHERE total_price > (SELECT MIN(total_price) FROM items
        WHERE order_num = 1023);
```

如果子查询恰好返回一个值，则您可在子查询中省略关键字 ANY、ALL 或 SOME。如果您省略 ANY、ALL 或 SOME，且子查询返回多个值，则您会收到错误。下一示例中的子查询仅返回一行，因为它使用聚集函数：

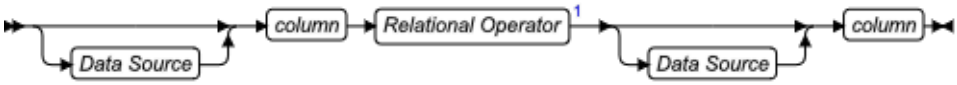
```
SELECT order_num FROM items
        WHERE stock_num = 9 AND quantity =
        (SELECT MAX(quantity) FROM items WHERE stock_num = 9);
```

另请参阅 ALL、ANY 和 SOME 子查询。

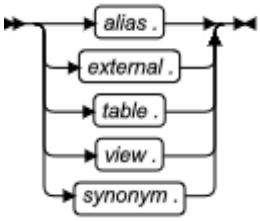
## 在 WHERE 子句中指定连接

您在 WHERE 子句中通过创建关系来连接两表，在来自一表的至少一列与来自另一表的至少一列之间连接。该连接创建临时的合成表，其中的满足连接条件的每一行对相连接形成单行。

连接




数据源




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *alias* | 为表或视图在 FROM 子句中声明的临时的可替代名称 | 请参阅 自连接；FROM 子句 | 标识符 |
| *column* | 要被连接的表或视图的列 | 必须在表或视图中存在 | 标识符 |
| *external* | 要从其检索数据的外部表 | 外部表必须存在 | 数据库对象名 |
| *synonym*、*table*、*view* | 在查询中要被连接的同义词、表或视图的名称 | 同义词和它指向的表或视图必须存在 | 数据库对象名 |

当指定的列的值之间相匹配时，来自该表或视图的列是***连接的***。当要被连接的列有相同的名称时，您必须以其数据源来限定每一列名称。

## *双表连接*

您可创建双表连接、多表连接、自连接和外连接（ GBase 8s 扩展语法）。下列样例展示双表连接：

```
SELECT order_num, lname, fname FROM customer, orders
        WHERE customer.customer_num = orders.customer_num;
```

## *多表连接*

多表连接是多于两个表的连接。它的结构类似于双表连接的结构，除了您在 WHERE 子句中有对于多于一对表的连接条件。当来自不同的表的列有相同的名称时，您必须以它的相关联的表或表别名来限定列名称，如在 ***table.column*** 中。要获取表名称的完整语法，请参阅 数据库对象名。

下列多表连接产生订购了一项的客户的公司名称及其库存编号和生产商代码：

```
SELECT DISTINCT company, stock_num, manu_code
    FROM customer c, orders o, items i
    WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num;
```

### *自连接*

您可将表连接到自身。要这样做，您必须在 FROM 子句中列出该表名称两次，并为它指定两个不同的表别名。在 WHERE 子句中使用别名来引用这*两个*表中的每一个。下一示例是 stock 表上的自连接。它找到库存项的对，其单价差一个大于 2.5 的系数。字母 x 和 y 分别为 stock 表的两个别名。

```
SELECT x.stock_num, x.manu_code, y.stock_num, y.manu_code
    FROM stock x, stock y WHERE x.unit_price > 2.5 * y.unit_price;
```

### *GBase 8s 扩展外连接*

下一外连接罗列客户的公司名称和所有相关联的订单编号，如果该客户已下了订单的话。如果没有，仍罗列公司名称，且为订单编号返回 NULL 值。

```
SELECT company, order_num FROM customer c, OUTER orders o
            WHERE c.customer_num = o.customer_num;
```

要获取更多关于外连接的信息，请参阅 *GBase 8s SQL 教程指南*。

## 层级查询子句

层级子句对表对象上的递归查询设置条件，在该表对象的行之中，存在父子依赖的层级。包括此子句的 SELECT 语句称为*层级查询*。

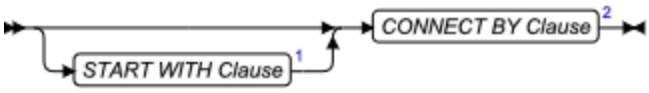
必须在 SELECT 语句的 FROM 子句中指定在其上进行层级查询操作的表对象。该表对象通常是自引用表，在其中一个或多个列作为同一表中另一列（或这些列的子集）的外键约束。

层级查询对若干行进行操作，在其中一个或多个列值对应于父子关系的逻辑结构之内的节点。如果父行有多个孩子，则在同一父母的孩子行之中存在兄弟关系。例如，这些关系可能反映一个组织的部门和管理级别之内的员工与管理者之中的报告结构。

此子句支持的语法是对 SQL 的 ANSI/ISO 标准的扩展。

#### 语法

层级子句




必须在 SELECT 语句的 FROM 子句中指定层级查询在其上操作的那些表对象。该表对象可为下列表对象中的任何一种：

- 表或可更新的视图
- 临时表
- 该会话连接到的同一 GBase 8s 实例的另一数据库中的表
- 作为查询的结果的派生的表
- 受到基于标签的访问控制（LBAC）安全策略保护的表

- 带有列级加密或行级加密的表
- 任何其他表对象的同义词。

在层级查询的 FROM 子句中，不支持下列表对象：

- 两个或多个表的连接
- 不可更新的视图
- 远程 GBase 8s 实例的数据库中的表
- CREATE EXTERNAL TABLE 语句定义了的外部表
- 序列对象。

GBase 8s 支持在层级查询的 projection 列表中的序列对象，在 WHERE 子句中，以及在表达式在 SELECT 语句中为有效的其他上下文中，但不在层级查询子句中。

在相关联的子查询和在不相关联的子查询中，层级子句是无效的。

层级查询可包括所有类型的优化程序伪指令，这些是例外：

- 连接顺序伪指令
- 连接方法伪指令

层级查询不支持 GBase 8s 的“并行数据库查询”（PDQ）特性。

层级子句可在表上指定递归的查询，该表的行描述父子关系的层级。

- 该层级可为简单的层级，诸如组织的报告结构，其中每个非根的节点向该层级之内的高级的单个节点报告。（在 GBase 8s 的 LBAC 安全特性中，TREE 类型的安全标签组件有简单的层级的逻辑结构。）
- 层级子句可查询更复杂拓扑的数据层级，其中的节点有多对多关系，且其中的孩子节点可为其父母的祖先。要了解关于使用层级子句来查询在数据层级之内有循环的信息，请参阅 CONNECT BY 子句。

**重要：** 层级查询对某些数据集合最为有效，其中的表内的父子依赖有简单图的逻辑拓扑。如果自引用表包括对相同列集合的多个独立的层级，或如果任何孩子行还是其父母的祖先，则请参阅 不是简单图的依赖样式。

**注：** 层级子句与表层级无关，在一系列类型表的模式之中存在父子关系的层级。类似地，全都来自通用基础类型的一系列 DISTINCT 数据类型的层级与数据层级类似，但与层级子句无关，层级存在于数据实体之间的父子依赖中，而不是数据类型之中的关系。

**特定于层级查询的 SQL 语法**

除了为包含层级数据的表的递归查询指定条件的 START WITH、CONNECT BY 和 CONNECT NOCYCLE BY 关键字之外，层级查询还支持那些仅在层级查询中才有效的语法令牌，以及在没有 CONNECT BY 子句的 SELECT 语句中不可使用的语法令牌。 特定于层级查询的语法令牌包括两个运算符、三个伪列和一个内建的函数：

- CONNECT_BY_ROOT 运算符

  此运算符可为其运算对象的根祖先返回一表达式。

- PRIOR 运算符

  此运算符可引用从前一递归步骤返回的值（此处的“步骤”是指该递归查询的一次迭代）。

- LEVEL 伪列

  此伪列返回一整数，指示该层级之内递归查询的哪一步骤返回了行。

- CONNECT_BY_ISCYCLE 伪列

  此伪列可指示一行是否有一个还是其祖先的孩子。

- CONNECT_BY_ISLEAF 伪列

  此伪列可指示一行在查询返回的行之中是否有任何的孩子。

- SYS_CONNECT_BY_PATH 函数

  此函数可构建和返回一字符串，该字符串表示从指定的行到层级的根的路径

- 在 ORDER BY 子句中的 SIBLINGS 关键字

  ORDER SIBLINGS BY 子句可对返回的每个级别的同一父母的兄弟行进行排序。

*伪列*是在特定的上下文中 SQL 解析器可识别的内建的标识符，共享同一命名空间作为列和变量。通常在 SELECT 语句的 Projection 子句中指定这些伪列和 SYS_CONNECT_BY_PATH 函数，但可在层级子句中指定 LEVEL 伪列和 PRIOR 运算符。

要获取仅支持层级查询的这些令牌的语法和语义的详细信息，请参阅 在 CONNECT BY 子句中的条件 和 ORDER SIBLINGS BY 子句。

**层级查询概述**

按下列次序处理包括层级子句的 SELECT 语句的子句：

1. FROM 子句（仅对于当前数据库中的单个表对象）
2. Hierarchical 子句
3. WHERE 子句（无连接断言）
4. GROUP BY 子句
5. HAVING 子句
6. Projection 子句
7. ORDER BY 子句

ORDER BY 子句的 ORDER BY SIBLING 选项可对同一父母的孩子行的集合进行排序。

包括层级子句的子查询按部分的顺序返回中间结果集，在此，特定层级的迭代（n+1）中产生的行紧跟在产生它们的迭代（n）中的行之后。然而，ORDER BY 子句、GROUP BY 或 HAVING 子句，或在 Projection 子句中指定的 DISTINCT 或 UNIQUE 关键字会销毁那部分的顺序。

层级子句跟在 SELECT 语句子句的词汇序列中的 WHERE 子句之后，但在该层级子句的结果上处理 WHERE 子句断言。如果 SELECT 语句包括层级子句，则 WHERE 子句不可指定连接子句，但在 FROM 子句中指定的表对象可作为连接一个或多个表的查询的结果集。

任何包括层级查询子句的 SELECT 语句都称为层级查询，在 FROM 子句指定的表上执行查询的递归序列：

1. 可选的 START WITH 子句可指定条件。返回任何满足此条件的行作为该层级查询的第一个中间结果集。
2. 下一步骤将在 CONNECT BY 子句中指定的条件应用到表。返回任何满足那个条件的行作为第二个中间结果集。
3. 下一步骤将 CONNECT BY 条件应用到表。返回的任何行构成第三个中间结果集。
4. CONNECT BY 子句递归地运行查询来产生连续的中间结果集，直到迭代产生空结果集为止。
5. 然后，层级 SELECT 语句组合前面的递归步骤的所有中间结果集，产生该层级子句的最终结果集。
6. 然后，将 WHERE 子句的断言应用到该层级子句检索了的这个行集合，然后按罗列的顺序应用 SELECT 语句的剩余的子句。

在 START WITH 和 CONNECT BY 子句返回所有中间的结果集之后，您可使用 ORDER SIBLINGS BY 子句来对该层级之内的每个级别的有相同的父母的兄弟行进行排序。要获取更多信息，请参阅 ORDER SIBLINGS BY 子句。

您可使用来自 SET EXPLAIN 语句的输出来查看层级查询的执行路径。

层级子句提供一种有效的替代机制，使用节点数据库扩展来从层级数据集检索信息

**层级数据集的示例**

在接下来的几个主题中，那些展示层级查询的 SQL 代码示例是基于下列 employee 表中的层级数据，其行包含关于在组织的层级之内的员工的信息。mgrid 列展示员工向其汇报的管理者的员工标识符（empid）：

```
CREATE TABLE employee(
        empid    INTEGER NOT NULL PRIMARY KEY,
        name     VARCHAR(10),
        salary DECIMAL(9, 2),
        mgrid    INTEGER
        );
```

**employee** 表中 17 行的数据值如下。

```
INSERT INTO employee VALUES   ( 1, 'Jones',      30000, 10);
INSERT INTO employee VALUES   ( 2, 'Hall',       35000, 10);
INSERT INTO employee VALUES   ( 3, 'Kim',        40000, 10);
```

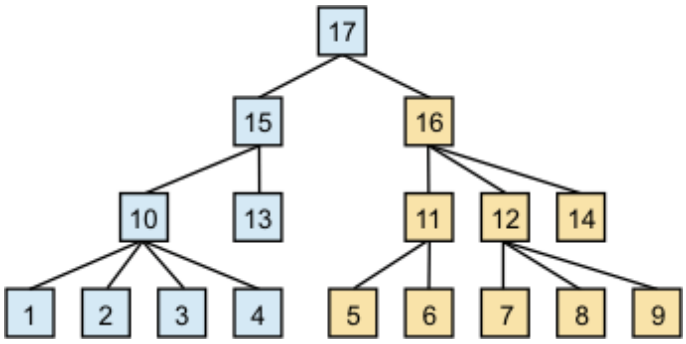
```
INSERT INTO employee VALUES  ( 4, 'Lindsay',   38000, 10);
INSERT INTO employee VALUES  ( 5, 'McKeough', 42000, 11);
INSERT INTO employee VALUES  ( 6, 'Barnes',    41000, 11);
INSERT INTO employee VALUES  ( 7, 'O''Neil',   36000, 12);
INSERT INTO employee VALUES  ( 8, 'Smith',      34000, 12);
INSERT INTO employee VALUES  ( 9, 'Shoeman',   33000, 12);
INSERT INTO employee VALUES  (10, 'Monroe',    50000, 15);
INSERT INTO employee VALUES  (11, 'Zander',    52000, 16);
INSERT INTO employee VALUES  (12, 'Henry',     51000, 16);
INSERT INTO employee VALUES  (13, 'Aaron',     54000, 15);
INSERT INTO employee VALUES  (14, 'Scott',     53000, 16);
INSERT INTO employee VALUES  (15, 'Mills',     70000, 17);
INSERT INTO employee VALUES  (16, 'Goyal',     80000, 17);
INSERT INTO employee VALUES  (17, 'Urbassek', 95000, NULL);
```

每一 **empid** 与 **mgrid** 值对表达引用的关系，带有适当的 CONNECT BY 条件的查询的递归迭代可正确地组装成层级。

在此，最后一行中 **mgrid** 列中的 NULL 值展示其 **empid** 值为 17 的员工 **Urbassek** 是此报告层级的根节点。

下图展示 **employee** 表数据的报告层级（以展示 **empid** 值的节点）的四个级别：

***图: 在报告层级中的元素的关系***

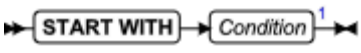



## START WITH 子句

可选的 START WITH 子句指定条件。满足此条件的行成为层级查询中开启 CONNECT BY 子句的递归操作的根。

START WITH 子句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




**用法**

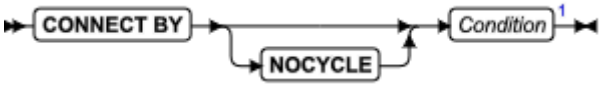
START WITH 子句指定 CONNECT BY 子句为其递归活动的第一迭代使用的搜索条件。如果您省略 START WITH 子句，则 CONNECT BY 子句对于中间结果的初始集将每行都作为层级的根处理。

## CONNECT BY 子句

CONNECT BY 子句为执行层级查询中的递归操作指定条件。

CONNECT BY 子句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




**用法**

如果您包括 START WITH 子句，则它指定的搜索条件应用于为层级查询生成第一个中间结果集。这由那些在 FROM 子句中指定的表中满足 START WITH 条件为真的行构成。

如果省略 START WITH 子句，则没有可用的 START WITH 条件作为过滤器，且第一个中间结果集是 FROM 子句指定的表中的行的全集。

通过应用 CONNECT BY 搜索条件，CONNECT BY 子句生成连续的中间结果集，直到当迭代生成空结果集时此递归过程终结为止。

**NOCYCLE 关键字**

由 CONNECT BY 子句的递归查询返回的行必须为简单的层级的一部分。 如果该查询返回一行，该行既是另一节点的祖先又是另一节点的后代，则包括该层级子句的 SELECT 语句失败并报错。此拓扑称为循环。

您可在 CONNECT BY 关键字与 CONNECT BY 子句的条件规范之间包括 NOCYCLE 关键字来过滤掉任何会导致层级查询失败并报错 -26079 的那些行，出错的原因是在中间的结果集中有循环。

例如，对于在主题 层级查询子句 中描述的 **employee** 表的层级数据集，下列 UPDATE 语句会引起其 **empid** 值为 5 和 17 的员工的循环：

```
UPDATE employee SET mgrid = 5 WHERE name = 'Urbassek';
```

在通过上述 UPDATE 语句修改了层级数据集之后，下列查询（省略 NOCYCLE 关键字）失败：

```
SELECT empid, name, mgrid , CONNECT_BY_ISLEAF leaf
        FROM employee
        START WITH name = 'Goyal'
        CONNECT BY PRIOR empid = mgrid;
```

当最后的 CONNECT BY 步骤检测到员工 **McKeough** 是循环的一部分时，发出错误 -26079：

```
    Empid     name                mgrid     leaf

      16      Goyal                17         0
      14      Scott                 16         1
```

```
12      Henry                16       0
9       Shoeman              12       1
8       Smith               12       1
7       O'Neil               12       1
11      Zander               16       0
6       Barnes               11       1
5       McKeough             11       0

26079: CONNECT BY query resulted in a loop/cycle.
Error in line 8
Near character position 28
```

您可在 CONNECT BY 关键字与 CONNECT BY 子句的*条件*规范之间包括 NOCYCLE 关键字来过滤掉会导致层级查询失败并报错 -26079 的任何行，该错误是由于在中间的结果集中有循环所致。下列查询在 CONNECT BY 子句中包括 NOCYCLE 关键字，这与在 Projection 子句中包括 CONNECT_BY_ISCYCLE 伪列而导致失败的查询不同。

```
SELECT empid, name, mgrid, CONNECT_BY_ISLEAF leaf, CONNECT_BY_ISCYCLE cycle
        FROM employee
        START WITH name = 'Goyal'
        CONNECT BY NOCYCLE PRIOR empid = mgrid;

empid name                mgrid     leaf    cycle

16      Goyal               17        0        0
14      Scott                16        1        0
12      Henry               16        0        0
9       Shoeman              12        1        0
8       Smith               12        1        0
7       O'Neil               12        1        0
11      Zander               16        0        0
6       Barnes              11        1        0
5       McKeough             11        0        0
17      Urbassek              5        0        1
15      Mills               17        0        0
13      Aaron               15        1        0
10      Monroe               15        0        0
4       Lindsay              10        1        0
3       Kim                 10        1        0
2       Hall                10        1        0
1     Jones                 10        1        0

17 row(s) retrieved.
```

由于 NOCYCLE 关键字使得在检测到了循环之后 CONNECT BY 子句能够继续处理，从在先前的示例中失败了的 CONNECT BY 步骤返回了 **Urbassek**，并继续处理直到已返回了该结果集中的所有行为止。在上述输出显示中，**leaf** 为 CONNECT_BY_ISLEAF 伪列的别名，且 **cycle** 为

CONNECT_BY_ISCYCLE 伪列的别名，在 Projection 子句中同时声明两个别名。在这些结果中，**Urbassek** 在 **cycle** 中被标记为循环的起因。

上述结果集表明，通过更改该行中的 **mgrid** 值，该行已识别了 **McKeough** 作为 **Urbassek** 的管理者，可从 **employee** 表移除循环：

```
UPDATE employee SET mgrid = NULL WHERE empid = 17;
```

## 在 CONNECT BY 子句中的条件

在布尔条件和在一般的 SQL 表达式中，除了表达式和运算符是有效的之外，在 CONNECT BY 子句中指定的 *condition* 支持另外两种语法结构，仅在包括层级子句的 SELECT 语句中，PRIOR 运算符和 LEVEL 伪列才是有效的。

### PRIOR 运算符

PRIOR 一元运算符可被包括在 CONNECT BY 子句中，以列名称作为它的运算对象。PRIOR 可用于将 CONNECT BY 子句的最近的先前递归步骤的结果的列引用，与对当前结果集的列引用区分开来。列名称紧跟在此右关联的运算符之后，如下列语法片断所示：

```
CONNECT BY mgrid = PRIOR empid
```

此处，在 mgrid 中指定管理者的那些行满足 CONNECT BY 条件，在先前的迭代中与员工值相匹配的列在 empid 列中。

PRIOR 运算符可应用于比列名称更复杂的表达式。下列条件使用算术表达式作为 PRIOR 的运算对象：

```
CONNECT BY PRIOR (salary - 10000) = salary
```

在同一 CONNECT BY 条件中，可以多次包括 PRIOR 运算符。另请参阅主题 层级查询子句，该主题提供一个在 CONNECT BY 子句的条件中使用 PRIOR 运算符的层级查询的示例。

### LEVEL 伪列

*伪列*是与列名称共享同一命名空间的 SQL 的关键字，在某些其中的列表达式为有效的上下文中那是有效的。

LEVEL 是返回在返回了行的层级子句中迭代步骤的序号的伪列。对于由 START WITH 子句返回的所有行，LEVEL 返回值 1。通过应用 CONNECT BY 子句的第一个迭代返回的行返回 2。通过 CONNECT BY 的连续的迭代返回的行有增量为 1 的 LEVEL 值，因此 LEVEL = (N + 1) 表明为第 N 次 CONNECT BY 迭代返回的值。LEVEL 列的数据类型是 INTEGER。

下列层级查询的样例在 Projection 子句的选择列表中指定 LEVEL：

```
SELECT name, LEVEL FROM employee START WITH name = 'Goyal'
   CONNECT BY PRIOR empid = mgrid;
```

该查询返回这些结果：

```
name            level
```

```
Goyal          1
Zander         2
McKeough       3
Barnes         3
Henry          2
O'Neil         3
Smith          3
Shoeman        3
Scott          2

9 row(s) retrieved.
```

在包括层级子句的 SELECT 语句的 Projection 子句中，以及在 CONNECT BY 子句的 *condition* 中，可包括 LEVEL。

然而，在下列上下文中，LEVEL 伪列不是有效的：

- 没有 CONNECT BY 子句的 SELECT 语句
- 层级子句的 START WITH *condition*
- CONNECT_BY_ROOT 运算符的运算对象
- SYS_CONNECT_BY_PATH 函数的参数。

**仅在层级查询中有效的附加的语法**

下列语法令牌支持层级查询，且仅在层级查询中是有效的。 然而，不同于 PRIOR 运算符和 LEVEL 伪列，它们在层级子句中不是有效的：

- CONNECT_BY_ISCYCLE 伪列
- CONNECT_BY_ISLEAF 伪列
- CONNECT_BY_ROOT 一元运算符
- SQL 的 SYS_CONNECT_BY_PATH( ) 函数。

**ROWNUM 伪列**

在 SELECT 语句中使用层次查询，CONNECT BY 子句支持使用 ROWNUM 伪列作为连接条件或筛选条件。

ROWNUM 伪列只能紧跟在 CONNECT BY 后使用，不支持使用在相关子查询的 GROUP BY、ORDER BY 中；

SELECT 语句中 CONNECT BY 子句使用 ROWNUM，不支持使用 INTERSECT、MINUS、UNION、UNION ALL 对结果集进行处理;

例如，查询 employee 表时，在层次查询 CONNECT BY 子句中指定条件 ROWNUM<3：

```
SELECT * FROM employee CONNECT BY ROWNUM <3;
```

该查询返回这些结果：

```
empid       name          salary       mgrid

17          Urbassek      95000.00
16          Goyal         80000.00      17
```

```
15          Mills          70000.00          17
14          Scott          53000.00          16
13          Aaron          54000.00          15
12          Henry          51000.00          16
11          Zander         52000.00          16
10          Monroe         50000.00          15
 9          Shoeman        33000.00          12
 8          Smith          34000.00          12
 7          O'Neil         36000.00          12
 6          Barnes         41000.00          11
 5          McKeough       42000.00          11
 4          Lindsay        38000.00          10
 3          Kim            40000.00          10
 2          Hall           35000.00          10
 1          Jones          30000.00          10
17          Urbassek       95000.00
16          Goyal          80000.00          17
15          Mills          70000.00          17
14          Scott          53000.00          16
13          Aaron          54000.00          15
12          Henry          51000.00          16
11          Zander         52000.00          16
10          Monroe         50000.00          15
 9          Shoeman        33000.00          12
 8          Smith          34000.00          12
 7          O'Neil         36000.00          12
 6          Barnes         41000.00          11
 5          McKeough       42000.00          11
 4          Lindsay        38000.00          10
 3          Kim            40000.00          10
 2          Hall           35000.00          10
 1          Jones          30000.00          10
```

**CONNECT_BY_ISCYCLE 伪列**

如果在层级中的下一级该行可能循环，则 CONNECT_BY_ISCYCLE 是返回 1 的伪列。即，该行有一个直接的孩子，该孩子又是在 CONNECT BY 子句中指定的搜索条件给出的祖先。如果该行不直接地导致循环，则该列返回 0。仅当在 CONNECT BY 子句中指定 NOCYCLE 时，才有可能为非 0 的值。此列的数据类型是 INTEGER。

下列 UPDATE 语句在 **employee** 表的数据层级中创建循环：

```
UPDATE employee SET mgrid = 5 WHERE empid = 17;
```

下列层级查询在 Projection 子句中包括 CONNECT_BY_ISCYCLE 伪列，但在它遇到 UPDATE 语句创建的循环的步骤中，CONNECT BY 子句抛出错误。

```
SELECT empid,
        name,
        mgrid,
        CONNECT_BY_ISLEAF leaf,
        CONNECT_BY_ISCYCLE cycle
FROM   employee
START WITH name = 'Goyal'
CONNECT BY   PRIOR empid = mgrid;

  665: Internal error on semantics -
  CONNECT_BY_ISCYCLE is used without NOCYCLE parameter..
Error in line 1
Near character position 72
```

在它遇到 UPDATE 语句创建的循环的步骤中抛出错误的 CONNECT BY 子句中，通过指定 NOCYCLE，此查询可避免 -655 错误。

```
SELECT empid, name, mgrid,
   CONNECT_BY_ISLEAF leaf, CONNECT_BY_ISCYCLE cycle
FROM employee
   START WITH name = 'Goyal'
   CONNECT BY NOCYCLE PRIOR empid = mgrid;
```

要获取此查询的结果，请参阅在主题 CONNECT BY 子句 中的对 NOCYCLE 关键字的描述的示例。

在下列上下文中，CONNECT_BY_ISCYCLE 伪列不是有效的：

- 没有 CONNECT BY 子句的 SELECT 语句
- START WITH 或 CONNECT BY 子句
- CONNECT_BY_ROOT 运算符的运算对象
- SYS_CONNECT_BY_PATH 函数的参数

CONNECT_BY_ISLEAF 伪列

如果该行如 CONNECT BY 所定义的那样是层级中的叶子，则 CONNECT_BY_ISLEAF 是返回 1 的伪列。如果节点在查询结果层级中（不是在实际的数据层级中）没有孩子，则该节点是*叶节点*。如果该行不是叶子，则该列返回 0。该列的数据类型是 INTEGER。

下列层级查询在 Projection 子句中指定 CONNECT_BY_ISLEAF 伪列，并声明 **leaf** 为那列的别名，在 DB-Access 中的结果集显示为：

```
SELECT empid, name, mgrid, CONNECT_BY_ISLEAF leaf
FROM emp1oyee
START WITH name = 'Goyal'
CONNECT BY PRIOR empid = mgrid;

      empid name                 mgrid     leaf

         16 Goyal                  17        0
```

```
14 Scott                16      1
12 Henry                 16      0
 9 Shoeman                12       1
 8 Smith                 12      1
 7 O'Neil                12      1
11 Zander                16      0
 6 Barnes                11      1
 5 McKeough               11       1

9 row(s) retrieved.
```

**CONNECT_BY_ROOT 运算符**

对于层级中的每行，CONNECT_BY_ROOT 一元运算符接受表达式作为它的运算对象，该表达式求值为层级的一个节点的行。CONNECT_BY_ROOT 返回它的运算对象的根祖先的表达式。


该 *expression* 运算对象可为任何 SQL 表达式，但它必须不包含任何层级查询令牌，包括下列令牌：

- CONNECT_BY_ROOT 或 PRIOR 一元运算符
- CONNECT_BY_ISCYCLE、CONNECT_BY_ISLEAF 或 LEVEL 伪列
- SYS_CONNECT_BY_PATH 函数。

此右关联的运算符的返回数据类型是指定的表达式的数据类型。

下列示例中的层级查询从 **employee** 表返回行，既包括员工要向其直接地报告的管理者的标识编号，也包括位于此查询的层级的根的管理者的名称。

```
SELECT empid, name, mgrid, CONNECT_BY_ROOT name AS topboss
FROM employee
START WITH name = 'Goyal'
CONNECT BY PRIOR empid = mgrid;

  empid name            mgrid topboss

     16 Goyal             17 Goyal
     14 Scott             16 Goyal
     12 Henry              16 Goyal
      9 Shoeman             12 Goyal
      8 Smith              12 Goyal
      7 O'Neil             12 Goyal
     11 Zander             16 Goyal
      6 Barnes             11 Goyal
      5 McKeough             11 Goyal

9 row(s) retrieved.
```

在下列上下文中，CONNECT_BY_ROOT 运算符不是有效的：

- 没有 CONNECT BY 子句的 SELECT 语句
- START WITH 或 CONNECT BY 子句
- SYS_CONNECT_BY_PATH 函数的参数

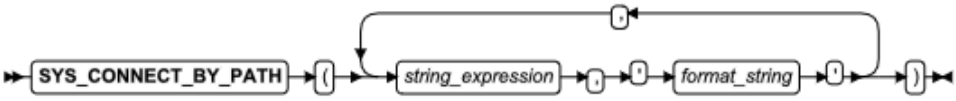
**SYS_CONNECT_BY_PATH 函数**

在包括层级子句的 SELECT 语句中调用 SYS_CONNECT_BY_PATH ( ) 函数是有效的，但不可从层级子句调用此函数。如果您尝试在 START WITH 或 CONNECT BY 子句的 *condition* 之内运行此函数，则 GBase 8s 返回错误。

在层级查询中，SYS_CONNECT_BY_PATH 函数可用来构建表示路径的字符串，该路径从对应于根节点的行至当前行。

这是 SYS_CONNECT_BY_PATH 的调用语法，返回在 LEVEL N 的指定行的字符串：

SYS_CONNECT_BY_PATH 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *format_string* | 通常是作为运算符的常量字符串 | 无 | 引用字符串 |
| *string_expression* | 标识行的表达式。 | 不可包括层级查询令牌 | 表达式 |

SYS_CONNECT_BY_PATH 构建表示路径的字符串，通过递归地串联连续的返回值，该路径从根到该层级的 LEVEL N 的指定的行：

- *path1* := *string_expression1*||*format_string* 表示从第一个中间的结果集到根行的路径，
- *path2* := *path1*||*string_expression2*||*format_string* 求值为从根到在第二个中间的结果集中的行的路径，
- . . .
- *pathN* := *path(N-1)*||*string_expressionN*||*format_string* 求值为从根到第 N 个中间的结果集的路径。

SYS_CONNECT_BY_PATH 的参数中的表达式必须不包括任何层级查询结构，包括下列结构：

- CONNECT_BY_ROOT 或 PRIOR 一元运算符
- CONNECT_BY_ISCYCLE、CONNECT_BY_ISLEAF 或 LEVEL 伪列
- SYS_CONNECT_BY_PATH 函数。

在参数列表中也无效的是聚集表达式。

从 SYS_CONNECT_BY_PATH ( ) 的返回值是 LVARCHAR(4000) 类型。

下列示例中的层级查询调用在 Projection 列表中的 SYS_CONNECT_BY_PATH 函数，以 **employee.name** 列和斜线（/）字符作为它的参数。

```
SELECT empid, name, mgrid, SYS_CONNECT_BY_PATH( name,'/') as hierarchy
    FROM employee
    START WITH name = 'Henry'
    CONNECT BY PRIOR empid = mgrid;
```

该查询返回数据层级的子集之内的行，其中在 START WITH 子句中指定 **Henry** 作为根，展现每一员工和员工的管理者的名称和 **empid** 编号，以及在该层级中到 **Henry** 的路径。CONNECT BY 子句使用相等断言 `PRIOR empid = mgrid` 来返回向管理者报告的员工（在此情况下，仅 **Henry**），通过先前的步骤返回了其 **empid**。该查询的结果集是：

```
empid        12
name          Henry
mgrid        16
hierarchy    /Henry

empid        9
name          Shoeman
mgrid        12
hierarchy    /Henry/Shoeman

empid        8
name          Smith
mgrid        12
hierarchy    /Henry/Smith

empid        7
name          O'Neil
mgrid        12
hierarchy    /Henry/O'Neil

4 row(s) retrieved.
```

这些行按检索它们的顺序罗列：

- START WITH 子句在此层级的根返回了 **Henry** 行。
- CONNECT BY 子句的第一个步骤返回了三行，对应于向 **Henry** 报告的三名员工。
- 下一 CONNECT BY 步骤未返回行，因为由先前的步骤返回的 **Shoeman**、**Smith** 和 **O'Neil** 都是此层级之内的叶节点，对其 `PRIOR empid = mgrid` 条件求值为假。

查询执行结束，展示返回的四行，此处，**hierarchy** 是 SYS_CONNECT_BY_PATH( name,'/') 为每一行返回到 **Henry** 的路径的别名。（在第一个返回的行中，字符串 /Henry 展示 **Henry** 的根状态。）

## 不是简单图的依赖样式

您可在包括复合的依赖的数据集上运行递归层级查询，诸如多根节点，或孩子节点的多父节点。然而，基于层级的数据树结构，可能发生循环，且可能返回更多的记录。

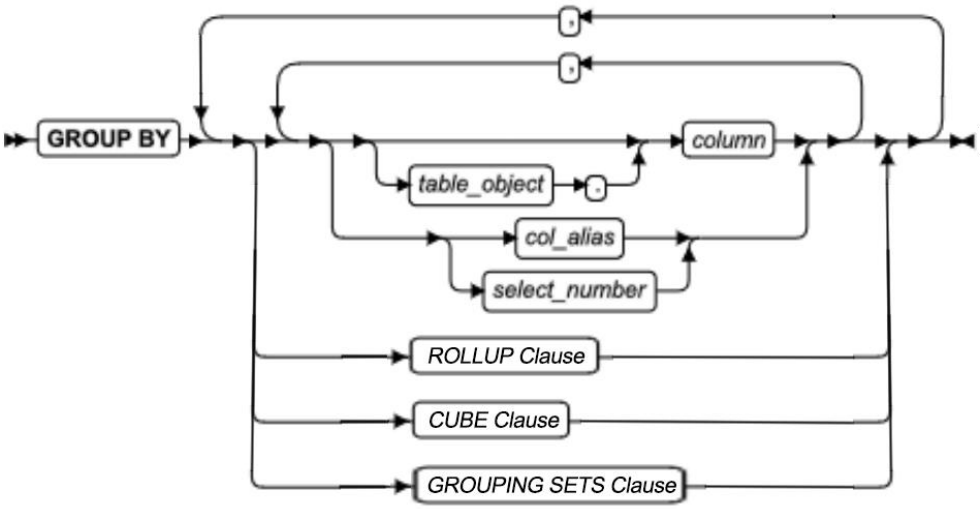
CONNECT BY 子句不可分析包括循环的数据集，其中有些孩子节点标识为它们的父节点的祖先。如果您的数据集包括循环，则该循环可能是包含无效数据的行的产物。

如果自引用的表描述的数据集包括多个层级，则包括带有条件的 START WITH 子句，该条件仅对于您想要该递归查询返回的层级的根为真。请对每一层级在表上运行分别的查询，使用不同的 START WITH 子句来指定每一查询中的根。

## GROUP BY 子句

使用 GROUP BY 子句来为每一组产生单行结果。*组*是在此子句中引用的每一列（或列表达式）的有相同值的行的集合。

GROUP BY 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *col_alias* | 列名称的别名 | 必须已在 Projection 子句中声明 | 标识符 |
| *column* | 通过此列（或此表达式）的值将行成组 | 请参阅 GROUP BY 与 Projection 子句之间的依赖。 | 标识符、表达式 |
| *select_number* | 指定在 Projection 子句的选择列表中的列或表达式的次序位置的整数 | 请参阅 使用选择编号。 | 精确数值 |
| *table_object* | 包含 *column* 的表或视图的名称、同义词或别名 | 必须存在且必须在 FROM 子句中指定 | 标识符 |

带有 GROUP BY 子句的 SELECT 语句为在 ***column*** 中有相同的值的，或在 ***col_alias*** 引用的列中有相同的值的，或在 ***select_number*** 指定的列或表达式中有相同的值的每一组行返回单一的结果行。

在 NLSCASE INSENSITIVE 数据库中，在 NCHAR 和 NVARCHAR 数据上的排序和字符串比较时，不管字母大小写的区别，因此，数据库服务器将由相同序列字母组成的字符串之中的大小写变量视作完全相同。对于在 NCHAR 或 NVARCHAR 列上的成组数据的查询，如果某些限定的行仅字母的大小写不同，则与在区分大小写的数据库中在同一数据集上的同一查询的组数相比，该组的数目较少。要获取更多关于在以 NLSCASE INSENSITIVE 属性创建了的数据库中的数据处理的更多信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## GROUP BY 与 Projection 子句之间的依赖

GROUP BY 子句限制 Projection 子句可指定的内容。如果您包括 GROUP BY 子句，则还必须在 GROUP BY 子句中引用 Projection 子句的选择列表中的每一 ***column***。

如果您在包括 GROUP BY 子句的查询的选择列表中指定聚集函数以及一个或多个列表达式，则必须包括在 GROUP 子句中用作聚集或时间表达式的一部分的所有列名称。

如果在 projection 子句中指定 OLAP 窗体函数，则在 GROUP BY 子句中还必须包括该 OLAP 窗体函数之内所有的列引用。

如果您在选择列表中为列声明别名，则可在 GROUP BY 子句中以那个别名代替该列名称，在 GROUP BY 子句中需要该列的名称或别名之一。

在 GROUP BY 列表中常量表达式和 BYTE 或 TEXT 列表达式不是有效的。

如果选择列表包括 BYTE 或 TEXT 列，则您不可使用 GROUP BY 子句。此外，您不可在 GROUP BY 子句中包括 ROWID。

如果选择列表包括用户定义的数据类型的列，则不可在 GROUP BY 子句中使用该列，除非该 UDT 可使用内建的 bit-hashing 函数。必须以 CANNOTHASH 修饰符创建不可使用该内建的 bit-hashing 函数的任何 UDT，其告诉数据库服务器不可在 GROUP BY 子句中使用该 UDT。

下列示例指定不在聚集表达式中的一列。 total_price 列不应在 GROUP BY 列表中，因为它是作为聚集函数的参数出现。COUNT 和 SUM 聚集被应用到每一组，而不是该查询的整个结果集。

```
SELECT order_num, COUNT(*), SUM(total_price)
        FROM items GROUP BY order_num;
```

如果选择列表中的列表达式仅是列名称，则您必须在 GROUP BY 子句中使用它的名称或它的别名。如果通过算术运算符将列与另一列组合，则您可选择以两种方式中的一种来成组该查询结果集：

- 通过单个列的名称或别名，
- 抑或，通过使用 ***select number*** 的组合的表达式，指定 Projection 子句的选择列表之内的表达式的次序位置的文字整数。

## GROUP BY 子句中的 NULL 值

在罗列在 GROUP BY 子句中的列中，包含 NULL 值的每一行属于单个组。也就是说，将所有的 NULL 值组在一起。

## 使用选择编号

您可在 GROUP BY 子句中使用一个或多个整数来代表列表达式。在下一示例中，第一个 SELECT 语句在 GROUP BY 子句中使用 **order_date** 和 **paid_date - order_date** 的选择编号。您仅可通过使用选择编号的组合的表达式来成组。

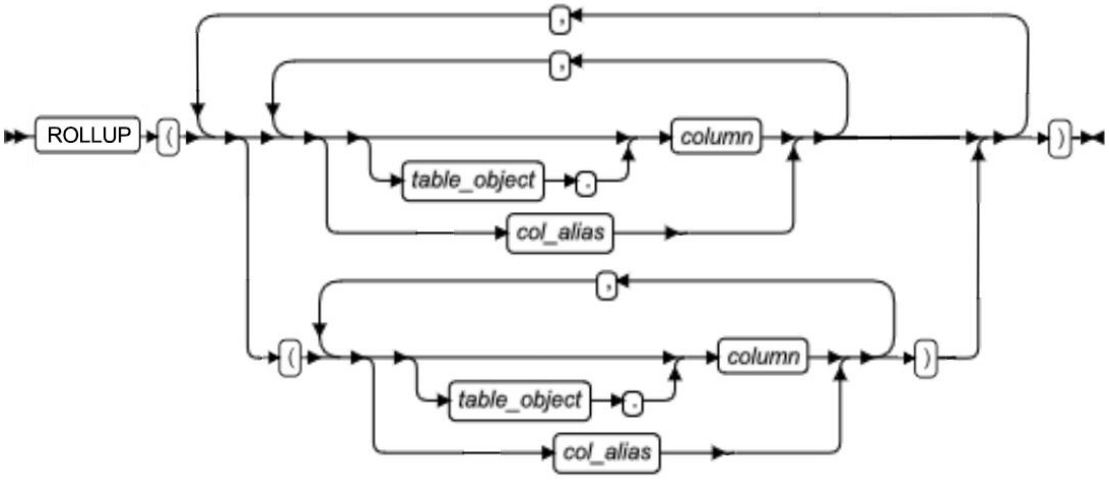
在第二个 SELECT 语句中，您不可以数学表达式 paid_date - order_date 来代替 2：

```
SELECT order_date, COUNT(*), paid_date - order_date
    FROM orders GROUP BY 1, 3;
SELECT order_date, paid_date - order_date
    FROM orders GROUP BY order_date, 2;
```

## ROLLUP 子句

GROUP BY ROLLUP 子句是一种分组统计用法。数据库首先按照指定的多列进行分组，然后将多组结果集进行 UNION ALL 并返回多组 GROUP BY 的结果。

ROLLUP 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *col_alias* | 列名称的别名 | 必须已在 Projection 子句中声明 | 标识符 |
| *column* | 通过此列（或此表达式）的值将行成组 | 请参阅 GROUP BY 与 Projection 子句之间的依赖。 | 标识符、表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *table_object* | 包含 *column* 的表或视图的名称、同义词或别名 | 必须存在且必须在 FROM 子句中指定 | 标识符 |

GROUP BY ROLLUP 子句分组规则如下：

- 分组个数：列个数 n + 1。
- 分组方式：先按照全部指定列分组，然后从右向左依次减少一列进行分组，直到最后不按照任何列分组，不参与分组的列对应结果集内容为 NULL，返回以上分组结果。

假如 ROLLUP 分组列为(A, B, C)，首先对(A,B,C)进行分组，然后对(A,B)进行分组，接着对(A)进行分组，无分组列进行分组，查询结果是把每种分组的结果集进行 UNION ALL 合并输出。分组统计的过程中，分组组合中未使用某分组列，则对应结果集设置为 NULL。例如，对(A,B)进行分组时，C 列为 NULL。如果分组列为 n 列，则共有 n+1 种组合方式。

### *ROLLUP 的示例*

例如，表 tab 1 表结构如下：

```
CREATE TABLE tab1 (
c1 int,
c2 int,
c3 varchar(20)
);
```

插入如下数据：

```
INSERT INTO tab1 values(1,1,'test1');
INSERT INTO tab1 values(1,2,'test2');
INSERT INTO tab1 values(2,1,'test3');
INSERT INTO tab1 values(2,2,'test4');
```

对表 tab1 的 c1、c2、c3 列进行 ROLLUP 分组，统计分组组合包括(c1,c2,c3),(c1,c2),(c1),全空 四种组合。

```
SELECT * FROM tab1 GROUP BY ROLLUP (c1,c2,c3);
```

返回结果如下：

```
c1 |c2 |c3     |
---|---|------|
2  |2  |test4 |
```

```
1  |2  |test2 |
2  |1  |test3 |
1  |1  |test1 |
1  |2  |      |
2  |1  |      |
2  |2  |      |
1  |1  |      |
2  |   |      |
1  |   |      |
   |   |      |
```

### *ROLLUP 的限制*

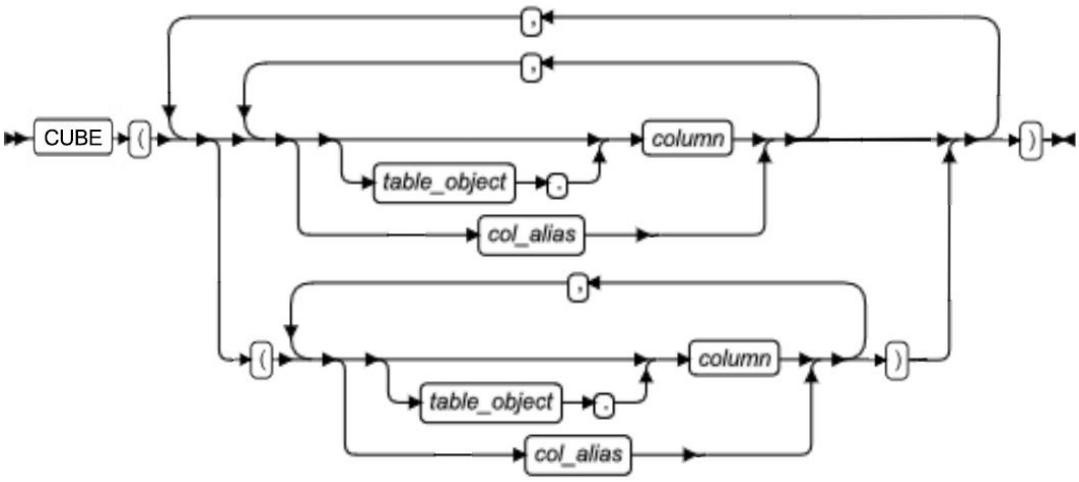
1. ROLLUP 关键字、参数不支持省略，参数支持字母、数字、中文字符、下划线及‘,’等字符。
2. ROLLUP 表达式内，支持 255 个参数。
3. ROLLUP 表达式内参数，支持单投影列的名称或别名，包括 列名、列别名、内建或用户自定义的非聚集函数表达式别名、算术表达式别名。
4. ROLLUP 表达式内参数，不支持常量表达式、大对象类型的列表达式及别名、函数表达式、聚集函数表达式及别名、NULL、虚拟列。
5. （表达式）用法可嵌套，不可超过两层。例如，GROUP BY ROLLUP((A,B),(C,D))，执行通过；GROUP BY ROLLUP(((A,B),(C,D)))，返回报错 201: A syntax error has occurred。
6. 支持多个 ROLLUP 子句，用逗号分隔。当使用多个 ROLLUP 子句时，多分组项之间做笛卡尔积分组。例如，GROUP BY ROLLUP(A,B),ROLLUP(A,B)，共计 9 种情况，(A,B,A,B)，(A,B,A)，(A,B)，(A,A,B)，(A,A)，(A)，(A,B)，(A)，全空。
7. ROLLUP 表达式内参数，不支持选择编号用法。例如，select a from t1 group by rollup(1)，报错 201: A syntax error has occurred。
8. 使用 ROLLUP 子句后，外部分组项不可使用选择编号用法。例如，select a from t1 group by rollup(a),1；返回报错 201: A syntax error has occurred。
9. 使用 ROLLUP 子句后，无法根据别名使用 HAVING 过滤，只能根据列名、表达式等进行过滤。
10. 使用 ROLLUP 子句后，查询语句不可使用 ROWID。
11. 使用 ROLLUP 子句后，查询语句不可使用 SEQUENCE 类型。
12. 使用 ROLLUP 子句后，查询语句不支持使用 OLAP 窗体函数。

13. 如果存在用户定义的数据类型的列，则不可在 GROUP BY 子句中使用该列，除非该 UDT 可使用内建的 bit-hashing 函数。

## CUBE 子句

GROUP BY CUBE 子句是一种分组统计用法。数据库首先按照指定的多列进行分组，然后将多组结果集进行 UNION ALL 并返回多组 GROUP BY 的结果。

CUBE 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *col_alias* | 列名称的别名 | 必须已在 Projection 子句中声明 | 标识符 |
| *column* | 通过此列（或此表达式）的值将行成组 | 请参阅 GROUP BY 与 Projection 子句之间的依赖。 | 标识符、表达式 |
| *table_object* | 包含 *column* 的表或视图的名称、同义词或别名 | 必须存在且必须在 FROM 子句中指定 | 标识符 |

GROUP BY CUBE 子句分组规则如下：

- 分组个数：2 的列个数次方个。
- 分组方式：按照所有排列组合的子集进行分组，不参与分组的列对应结果集内容为 NULL，返回以上分组结果。

假如，CUBE 分组列为(A, B, C)，则首先对(A,B,C)进行分组，然后依次对(A,B)、(A,C)、(A)、(B,C)、(B)、(C)、无分组列八种情况进行分组，最后进行查询。当某种分组组合未使用到某分组列，且投影列存在该列，则对应结果集设置为 NULL。输出为每种分组的结果集进行 UNION ALL。CUBE 分组共有 2n 种组合方式。

### *CUBE 的示例*

例如，表 tab 1 表结构如下：

```
CREATE TABLE tab1 (
c1 int,
c2 int,
c3 varchar(20)
);
```

插入如下数据：

```
INSERT INTO tab1 values(1,1,'test1');
INSERT INTO tab1 values(1,2,'test2');
INSERT INTO tab1 values(2,1,'test3');
INSERT INTO tab1 values(2,2,'test4');
```

对表 tab1 的 c1、c2、c3 列进行 CUBE 分组，统计分组组合包括(c1,c2,c3),(c1,c2),(c1,c3),(c2,c3),(c1),(c2),(c3),全空 八种组合。

```
SELECT * FROM tab1 GROUP BY CUBE(c1,c2,c3);
```

返回结果如下：

```
c1 |c2 |c3     |
---|---|------|
   |   |       |
2  |   |       |
1  |   |       |
   |1  |       |
   |2  |       |
1  |2  |       |
2  |1  |       |
2  |2  |       |
1  |1  |       |
   |   |test2  |
   |   |test4  |
   |   |test1  |
   |   |test3  |
2  |   |test4  |
```

```
1   |    |test1 |
1   |    |test2 |
2   |    |test3 |
    |2   |test2 |
    |1   |test1 |
    |2   |test4 |
    |1   |test3 |
2   |2   |test4 |
1   |2   |test2 |
2   |1   |test3 |
1   |1   |test1 |
```

### *CUBE 的限制*

1. CUBE 关键字、参数不支持省略，参数支持字母、数字、中文字符、下划线及‘,’等字符。
2. CUBE 表达式内，支持 255 个参数。
3. CUBE 表达式内参数，支持单投影列的名称或别名，包括 列名、列别名、内建或用户自定义的非聚集函数表达式别名、算术表达式别名。
4. CUBE 表达式内参数，不支持常量表达式、大对象类型的列表达式及别名、函数表达式、聚集函数表达式及别名、NULL、虚拟列。
5. （表达式）用法可嵌套，不可超过两层。例如，GROUP BY CUBE ((A,B),(C,D))，执行通过；GROUP BY CUBE (((A,B),(C,D)))，返回报错 201: A syntax error has occurred。
6. 支持多个 CUBE 子句，用逗号分隔。当使用多个 CUBE 子句时，多分组项之间做笛卡尔积。例如，GROUP BY CUBE (A,B), CUBE (A,B)，共计 16 种情况，(A,B,A,B),(A,B,A),(A,B,B),(A,B),(A,A,B),(A,A),(A,B),(A),(B,A,B),(B,A),(B,B),(B),(A,B),(A),(B),全空。
7. CUBE 表达式内参数，不支持选择编号用法。例如，select a from t1 group by cube(1)，报错 201: A syntax error has occurred。
8. 使用 CUBE 子句后，外部分组项不可使用选择编号用法。例如，select a from t1 group by cube(a),1 ; 返回报错 201: A syntax error has occurred。
9. 使用 CUBE 子句后，无法根据别名使用 HAVING 过滤，只能根据列名、表达式等进行过滤。
10. 使用 CUBE 子句后，查询语句不可使用 ROWID。
11. 使用 CUBE 子句后，查询语句不可使用 SEQUENCE 类型。

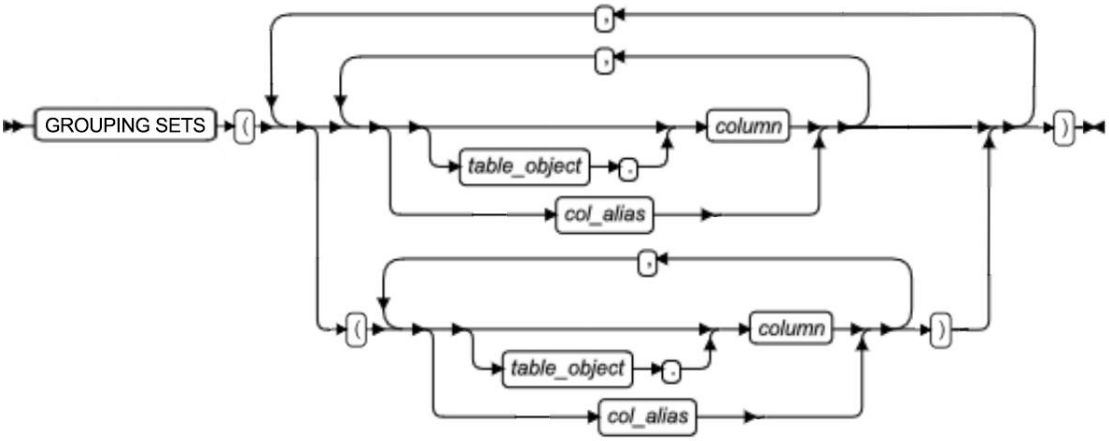
12. 使用 CUBE 子句后，查询语句不支持使用 OLAP 窗体函数。

13. 如果存在用户定义的数据类型的列，则不可在 GROUP BY 子句中使用该列，除非该 UDT 可使用内建的 bit-hashing 函数。

## GROUPING SETS 子句

GROUP BY GROUPING SETS 子句是一种分组统计用法，用于避免了 ROLLUP/CUBE 过多的分组情况。数据库首先按照指定的多列进行分组，然后将多组结果集进行 UNION ALL 并返回多组 GROUP BY 的结果。

GROUPING SETS 子句




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *col_alias* | 列名称的别名 | 必须已在 Projection 子句中声明 | 标识符 |
| *column* | 通过此列（或此表达式）的值将行成组 | 请参阅 GROUP BY 与 Projection 子句之间的依赖。 | 标识符、表达式 |
| *table_object* | 包含 *column* 的表或视图的名称、同义词或别名 | 必须存在且必须在 FROM 子句中指定 | 标识符 |

GROUP BY GROUPING SETS 子句分组规则如下：

- 分组个数：GROUPING SETS 中以逗号分隔的分组单元的个数。
- 分组方式：对每个分组单元依次 GROUP BY 分组并将结果集进行 UNION ALL，不参与分组的列对应结果集内容为 NULL。

假如，GROUPING SETS 分组列为(A, B, C)，则对 (A)、(B)、(C) 三种情况进行分组，最后进行查询。当某种分组组合未使用到某分组列，且投影列存在该列，则对应结果集设置为 NULL。

## *GROUPING SETS 的示例*

例如，表 tab 1 表结构如下：

```
CREATE TABLE tab1 (
c1 int,
c2 int,
c3 varchar(20)
);
```

插入如下数据：

```
INSERT INTO tab1 values(1,1,'test1');
INSERT INTO tab1 values(1,2,'test2');
INSERT INTO tab1 values(2,1,'test3');
INSERT INTO tab1 values(2,2,'test4');
```

对表 tab1 的 c1、c2、c3 列进行 GROUPING SETS 分组，统计分组组合包括 (c1) , (c2) , (c3) 三种组合。

```
SELECT * FROM tab1 GROUP BY GROUPING SETS(c1,c2,c3);
```

返回结果如下：

```
c1 |c2 |c3    |
---|---|------|
2  |    |      |
1  |    |      |
   |1   |      |
   |2   |      |
   |    |test2 |
   |    |test4 |
   |    |test1 |
   |    |test3 |
```

## *GROUPING SETS 的限制*

1. GROUPING SETS 关键字、参数不支持省略，参数支持字母、数字、中文字符、下划线及‘,’等字符。

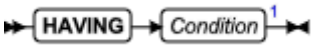
2. GROUPING SETS 表达式内，支持 255 个参数。

3. GROUPING SETS 表达式内参数，支持单投影列的名称或别名，包括 列名、列别名、内建或用户自定义的非聚集函数表达式别名、算术表达式别名。
4. GROUPING SETS 表达式内参数，不支持常量表达式、大对象类型的列表达式及别名、函数表达式、聚集函数表达式及别名、NULL、虚拟列。
5. （表达式）用法可嵌套，不可超过两层。例如，GROUP BY GROUPING SETS ((A,B),(C,D))，执行通过；GROUP BY GROUPING SETS (((A,B),(C,D)))，返回报错 201: A syntax error has occurred。
6. 支持多个 GROUPING SETS 子句，用逗号分隔。当使用多个 GROUPING SETS 子句时，多分组项之间做笛卡尔积。例如，GROUP BY GROUPING SETS (A,B), GROUPING SETS (A,B)，共计 4 种情况，(A,A),(A,B),(B,A),(B,A)。
7. GROUPING SETS 表达式内参数，不支持选择编号用法。例如，select a from t1 group by grouping sets(1)，报错 201: A syntax error has occurred。
8. 使用 GROUPING SETS 子句后，外部分组项不可使用选择编号用法。例如，select a from t1 group by grouping sets(a),1 ； 返回报错 201: A syntax error has occurred。
9. 使用 GROUPING SETS 子句后，无法根据别名使用 HAVING 过滤，只能根据列名、表达式等进行过滤。
10. 使用 GROUPING SETS 子句后，查询语句不可使用 ROWID。
11. 使用 GROUPING SETS 子句后，查询语句不可使用 SEQUENCE 类型。
12. 使用 GROUPING SETS 子句后，查询语句不支持使用 OLAP 窗体函数。
13. 如果存在用户定义的数据类型的列，则不可在 GROUP BY 子句中使用该列，除非该 UDT 可使用内建的 bit-hashing 函数。

## HAVING 子句

使用 HAVING 子句来将一个或多个限定的条件应用到组或应用到整个结果集。

HAVING 子句




在下列示例中，每一条件将该组的一个计算的属性与该组的另一计算的属性或常量进行比较。第一个 SELECT 语句使用 HAVING 子句，将计算的表达式 COUNT(*) 与常量 2 进行比较。该查询返回有两个以上项的所有订单上每项的平均合计价格。

第二个 SELECT 语句罗列那些在同一个月中呼叫两次或更多次的客户的客户及其呼叫月份：

```
SELECT order_num, AVG(total_price) FROM items
        GROUP BY order_num HAVING COUNT(*) > 2;
        SELECT customer_num, EXTEND (call_dtime, MONTH TO MONTH)
```

```
        FROM cust_calls GROUP BY 1, 2 HAVING COUNT(*) > 1;
```

您可使用 HAVING 子句来在 GROUP BY 列值上以及在计算的值上设置条件。 此示例返回 cust_code 和 customer_num、call_dtime，并从 customer_num 小于 120 的客户已收到的所有呼叫的 call_code 来将它们分组：

```
SELECT customer_num, EXTEND (call_dtime), call_code
        FROM cust_calls GROUP BY call_code, 2, 1
        HAVING customer_num < 120;
```

HAVING 子句通常是对 GROUP BY 子句的补充。 如果您省略 GROUP BY 子句，则 HAVING 子句应用于满足该查询的所有行，以及在组成单个组的表中的所有行。下列示例返回该表中所有值的平均价格，只要表中有十行以上：

```
SELECT AVG(total_price) FROM items HAVING COUNT(*) > 10;
```

由于在 WHERE 子句中的条件不可包括聚集表达式，所以您可使用 HAVING 子句来将带有聚集的条件应用于查询的整个结果集，如上例所示。

在 HAVING 子句中的条件不可包括 DISTINCT 或 UNIQUE 聚集表达式。例如，下列查询失败并报语法错误：

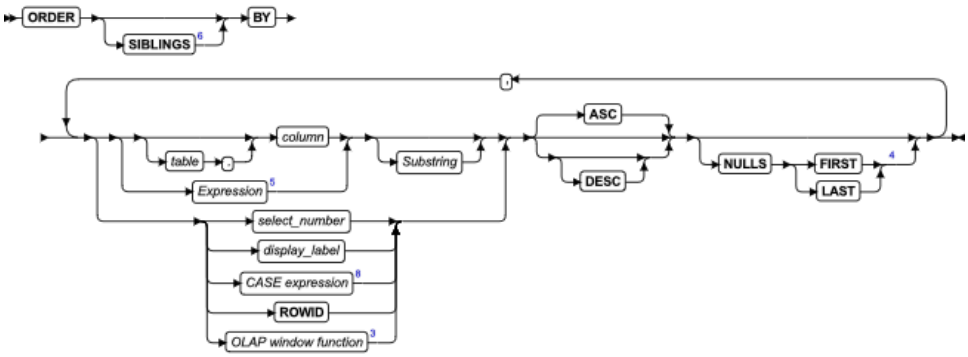
```
 SELECT order_num, COUNT(*) number, AVG (total_price) average
        FROM items
        GROUP BY order_num
        HAVING COUNT(DISTINCT *) > 2;
```

然而，如果从上述示例省略 DISTINCT 关键字，则不发出错误。

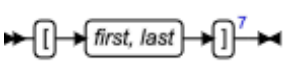
## ORDER BY 子句

ORDER BY 子句按指定的列或表达式对查询结果排序。

ORDER BY 子句




子字符串




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 按此列中的值对行排序 | 无 | 标识符 |
| *display_label* | 列或列表达式的临时名称 | 在 Projection 子句中声明的标签之中必须是唯一的 | 标识符 |
| *first*、*last* | 在要对结果集排序的列子串中的第一个和最后一个字节 | 整数；仅限于 BYTE、TEXT 和字符数据类型 | 精确数值 |
| *select_number* | 在 Projection 子句的选择列表中列的次序位置 | 请参阅 使用选择编号。 | 精确数值 |
| *table* | 包含 *column* 的表或视图的名称、同义词或别名 | 必须存在且必须在 FROM 子句中指定 | 标识符 |

ORDER BY 子句表明该查询返回多行。在 SPL 中，如果您指定 ORDER BY 子句而没有 FOREACH 循环来处理 SPL 例程之内单个地返回的行，则数据库服务器发出错误。

下列查询在 FROM 子句中指定派生的表，按照 col1 值对其行排序，并声明 vtab 作为派生的表的名称，且 vcol 作为其唯一列的名称：

```
SELECT vcol FROM
        (SELECT FIRST 5 col1 FROM tab1 ORDER BY col1) vtab(vcol);
```

**在 NLSCASE INSENSITIVE 数据库中的 ORDER BY**

在以 NLSCASE INSENSITIVE 属性创建的数据库中，对 NCHAR 或 NVARCHAR 数据类型的列和表达式的操作不区分大写字母和小写字母。因此，包括 ORDER BY 子句的查询可能以不管变量的字母大小写的序列返回行，如果该列或表达式为 NLS 数据类型，且该数据包括仅字母大小写不同的值。

如果数据集包括同一字符串的字母大小写变量，则按重复来处理这些，带有按它们的检索顺序罗列的大小写变量。例如，被处理为重复的一系列 NCHAR 或 NVARCHAR 字符串可能按此顺序出现：

```
gAMma
GAmma
GaMMa
gamma
GAMMA
```

要获取更多信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## 按列或按表达式排序

要按表达式对查询结果排序，您还必须为在 Projection 子句中的表达式声明显示标签，如下例所示，为两列之间的差异声明显示标签 span：

```
SELECT paid_date - ship_date span, customer_num FROM orders
        ORDER BY span;
```

GBase 8s 支持在 ORDER BY 子句中使用列和表达式，它们不会出现在 Projection 子句的选择列表中。可以忽略选择列表中的派生列的显示标签，并通过 ORDER BY 子句中的选择编号指定派生列。

然而，如果下列任一为真的话，那么 Projection 子句的选择列表必须包括所有 ORDER BY 子句指定的列或表达式：

- 该查询包括 DISTINCT、UNIQUE 或 UNION 运算符。
- 该查询包括 INTO TEMP ***table*** 子句。
- 分布式查询访问远程数据库，其服务器要求 ORDER BY 子句中的每个列或表达式还要出现在 Projection 子句的选择列表中。
- ORDER BY 子句中的表达式包括列子字符串的显示标签。（请参阅下一部分 按子字符串排序。）

下一查询从 orders 表选择一列，并按照另一列的值对结果排序。缺省情况下，按升序罗列这些行。

```
SELECT ship_date FROM orders ORDER BY order_date;
```

本版 **GBase 8s** 支持当 SELECT 投影列仅包含聚集表达式列时，无需 GROUP BY 子句即可按该表达式排序，例如：

```
SELECT MAX(ship_weight)
        FROM orders ORDER BY ship_weight;
```

但是，当SELECT 投影列不仅包含了聚集表达式列，还包含了其它列时，查询中必须具有 GROUP BY 子句才能按照聚集表达式排序。如下所示：

```
SELECT ship_charge, MAX(ship_weight)
        FROM orders GROUP BY ship_charge ORDER BY ship_weight;
```

如果当前的处理语言环境定义了本地化顺序，那么 NCHAR 和 NVARCHAR 列的值会以本地化的顺序进行排序。

在 GBase 8s 中，ORDER BY 子句中的任何 ***column*** 均不能为集合类型，但其结果集定义集合派生表的查询可包括 ORDER BY 子句。要了解示例，其参阅 集合派生表。

如果增加 DS_NONPDQ_QUERY_MEM 配置参数的设置，那么可以提升一些使用 ORDER BY 子句对很大的行的集合进行排序的非 PDQ 查询的性能。

## 按子字符串排序

您可按照子字符串排序，而不是按照字符、BYTE 或 TEXT 列的整个长度，或按照返回字符串的表达式排序。数据库服务器使用该子字符串来对结果集排序。通过指定整数下标（ ***first*** 和 ***last*** 参数）来定义该子字符串，表示在该列值之内子字符串的起始和终止字节位置。

下列 SELECT 语句查询 customer 表，并在 ORDER BY 列中指定列子字符串。这会指导数据库服务器通过包含在列值的第六至第九字节中的 lname 列的一部分来对查询结果排序。

```
SELECT * from customer ORDER BY lname[6,9];
```

假设在 **customer** 表的一行中的 **lname** 的值为 Greenburg。由于 ORDER BY 子句中的列子字符串，数据库服务器通过使用值 burg 来确定此行的排序位置，而不是通过整个列值 Greenburg。

当按照表达式排序时，您可仅为返回字符数据类型的表达式指定子字符串。如果您在 ORDER BY 子句中指定列子字符串，则该列必须有下列数据类型中的一种：BYTE、CHAR、NCHAR、NVARCHAR、TEXT 或 VARCHAR。

GBase 8s 还可支持 ORDER BY 子句中的 LVARCHAR 列子字符串，如果该列在本地数据库服务器的数据库中的话。

要获取关于使用在 ORDER BY 子句中的列子字符串的 GLS 方面的信息，请参阅 *GBase 8s GLS 用户指南*。

## 按 CASE 表达式排序

ORDER BY 子句可包括 CASE 表达式来指定排序键。

在下列示例中，表 tab_case 的列 a_col 为 INT 类型。对表 tab_case 的查询包括 Projection 列表中的列 a_col 和聚集表达式 SUM(a_col)，并通过 a_col 的值将结果分组。ORDER BY 子句指定两个排序键：

- 紧跟在 ORDER BY 关键字之后的 CASE 表达式
- AVG(a_col) 聚集表达式：

```
CREATE TABLE tab_case(a_col INT, b_col VARCHAR(32));

SELECT   a_col, SUM(a_col)
   FROM tab_case
   GROUP BY a_col
   ORDER BY CASE
                WHEN a_col IS NULL
                   THEN 1
                ELSE 0 END ASC,
            AVG(a_col);
```

在此，ASC 关键字显式地将 CASE 表达式的结果标识为升序排序键。在缺省情况下，AVG(a_col) 排序键也指定升序排序。

在下列类似的示例中，基于同一 tab_case 表上的查询，第二个 CASE 表达式返回或者 1 或者 0 作为返回的 AVG(a_col) 聚集值的排序键值。

```
SELECT   a_col, SUM(a_col)
   FROM tab_case GROUP BY a_col
```

```
ORDER BY CASE
            WHEN a_col IS NULL
               THEN 1
            ELSE 0 END ASC,
         AVG(a_col),
         CASE
            WHEN AVG(a_col) IS NULL
               THEN 1
            ELSE 0 END;
```

## 升序和降序

您可使用 ASC 和 DESC 关键字来指定升序（最小值在先）或降序（最大值在先）。

缺省的顺序为升序。对于 DATE 和 DATETIME 数据类型，***最小的***意思是时间上最早的，***最大的***意思是时间上最晚的。对于在缺省的语言环境中的字符数据类型，该顺序为 ASCII 顺序序列，如 U.S. English 数据的排序顺序 中所列。

对于 NCHAR 或 NVARCHAR 数据类型，使用当前会话的本地化顺序排序，如果那与代码集顺序不同的话。要获取更多关于排序的信息，请参阅 SET COLLATION 语句。

如果您指定 ORDER BY 子句，则在缺省情况下，NULL 值的排序小于非 NULL 值。使用 ASC 顺序，则 NULL 值排在任何非 NULL 值之前；使用 DESC 顺序，则 NULL 排在最后。

**指定 NULL 值的顺序**

ORDER BY 子句可包括 NULLS FIRST 关键字或 NULLS LAST 关键字来显式地（抑或覆盖）展示 NULL 值的缺省的排序顺序：

- NULLS FIRST 关键字指示数据库服务器将 NULL 值排在排序的查询结果的最前面。按降序排序，ASC NULLS FIRST 关键字请求缺省的顺序。在降序排序中，DESC NULLS FIRST 指定在排序键列中的带有 NULL 值的行排在排序的结果集中非 NULL 行的前面。
- NULLS LAST 关键字指示数据库服务器将 NULL 值排在排序的查询结果的最后面。在降序排序中，DESC NULLS LAST 关键字请求缺省的顺序。在降序排序中，ASC NULLS LAST 指定在排序键列中的带有 NULL 值的行跟在排序的结果集中非 NULL 行之后。

## 嵌套排序

如果您在 ORDER BY 子句中罗列多个列，则您的查询按照嵌套的排序排列。排序的第一级是基于第一列的；第二列决定排序的第二级。下列嵌套排序的示例选择 cust_calls 表中的所有行，通过 call_code 之内的 call_code 和 call_dtime 对它们进行排序：

```
SELECT * FROM cust_calls ORDER BY call_code, call_dtime;
```

## 使用选择编号

在列名称的位置，您可在 ORDER BY 子句中输入一个或多个整数，来引用罗列在 Projection 子句的选择列表中所罗列的项的位置。您还可使用选择编号来按表达式排序。

下列示例使用嵌套排序中的选择编号来按照表达式 paid_date - order_date 和 customer_num 排序：

```
SELECT order_num, customer_num, paid_date - order_date
        FROM orders
        ORDER BY 3, 2;
```

当通过 UNION 或 UNION ALL 关键字连接 SELECT 语句时，或当在同一位置中相兼容的列有不同的名称时，在 ORDER BY 子句中需要选择编号。

## 按 Rowid 排序

您可在 ORDER BY 子句中指定 ROWID 关键字。这指定 **rowid** 列，在为分片的表以及以 WITH ROWIDS 子句创建了的分片的表中的隐藏列。**rowid** 列包含与表中的行相关联的唯一的内部记录编号。（然而，建议您使用主键作为您的访问方法，而不是利用 **rowid** 列。）

如果您正在从其选择的表是一个没有 **rowid** 列的分片的表，则 ORDER BY 子句不可指定 **rowid** 列。

当您在 ORDER BY 子句中指定 ROWID 时，您不需要在 Projection 子句中包括 ROWID 关键字。

要获取关于 **rowid** 值以及如何在列表达式中使用 **rowid** 列的进一步的信息，请参阅 WITH ROWIDS 选项 和 使用 Rowid。

## 带有 DECLARE 的 ORDER BY 子句

在 GBase 8s ESQL/C 中，您不可使用带有 FOR UPDATE 子句的 DECLARE 语句来将游标与没有 ORDER BY 子句的 SELECT 语句相关联。

## 在 ORDER BY 列上设置索引

当您在 SELECT 语句中包括 ORDER BY 子句时，您可通过在 ORDER BY 子句指定的一列或多列上创建索引来提升查询的性能。数据库服务器使用您在 ORDER BY 列上设置的索引来以最高效的方式对查询结果排序。要获取更多关于如何创建与 ORDER BY 子句的列相对应的索引的信息，请参阅 使用 ASC 和 DESC 排序顺序选项。

## ORDER SIBLINGS BY 子句

ORDER SIBLINGS BY 子句仅在层级查询中是有效的。 可选的 SIBLINGS 关键字指定首先对父行排序，以及然后对该层级之内每个级别的每一父行的孩子行进行排序的顺序。

有些行有 SIBLINGS BY 关键字之后指定的列中的值的重复列表，这些行在带有相同的值列表和相同的父行的那些行之中是任意排序的。如果层级查询包括不带有 SIBLINGS 关键字的 ORDER BY 子句，则根据那些跟在 ORDER BY 关键字之后的排序规范来排列行的顺序。在层级查询中，既不需要 ORDER BY 子句，也不需要 ORDER BY 子句的 ORDER SIBLINGS BY 选项。

在下列示例中的层级查询返回层级数据集中的行的子集，其根为 **Goyal**，如主题 层级查询子句 中罗列的那样。 此查询包括 ORDER SIBLINGS BY 子句来按照 **name** 对那些报告给同一管理者的员工进行排序：

```
SELECT empid, name, mgrid, LEVEL
        FROM employee
        START WITH name = 'Goyal'
        CONNECT BY PRIOR empid = mgrid
        ORDER SIBLINGS BY name;
```

以下列顺序对此查询返回的行进行排序：

```
 empid name            mgrid       level

    16 Goyal             17          1
    12 Henry             16          2
     7 O'Neil            12          3
     9 Shoeman           12          3
     8 Smith             12          3
    14 Scott             16          2
    11 Zander            16          2
     6 Barnes            11          3
     5 McKeough          11          3

    9 row(s) retrieved.
```

在此，START WITH 子句返回了在此层级的根部的 Goyal 行。 两个后续的 CONNECT BY 步骤（在 level 伪列中标记为 2 和 3）返回三个兄弟行的集合：

- **Henry**、**Scott** 和 **Zander** 是其父母为 **Goyal** 的兄弟；
- **O'Neil**、**Shoeman** 和 **Smith** 是其父母为 **Henry** 的兄弟；
- **Barnes** 和 **McKeough** 使其父母为 **Zander** 的兄弟。

下一 CONNECT BY 步骤未返回行，因为符合 level = 3 的那些行是此层级中的叶节点。在该查询的此执行点上，将 ORDER SIBLINGS BY 应用于结果集，按上述顺序对这些行排序。

由于该排序键 **name** 为 VARCHAR 列，因此在每一兄弟的集合之内的返回的行都按照它们的 **employee.name** 值的 ASCII 顺序排列。仅在返回的行的层级中为叶节点的那些兄弟的集合在排序的结果集中连续地出现，因为管理者紧跟在向他们报告的员工之后，而不是他们的兄弟。此示例中的例外是 **Scott**，其孩子节点形成空集。

ORDER BY 子句中的 SIBLINGS 关键字是对 SQL 语言的 ISO 标准语法的扩展。如果您在不包括有效的 CONNECT BY 子句的查询或子查询的 ORDER BY 子句中包括 SIBLINGS 关键字，则 SELECT 语句失败并报错。

要获取更多关于层级查询和 CONNECT BY 子句的信息，请参阅 层级查询子句。

## 在 SELECT 语句的 ORDER BY 子句中的 OLAP window 函数

您可在不包括 CONNECT BY 子句的那些 SELECT 子句的最终的 ORDER BY 子句中包括 OLAP window 函数。

如果在 ORDER BY 子句中出现 OLAP 函数子句，则在 ORDER BY 求值之前，先求值 OLAP 函数。

一般地说，对于包括一个或多个 OLAP window 函数的简单的 SELECT 语句，数据库服务器遵循下列处理次序：

- 将任何连接、过滤器、GROUP BY 或 HAVING 规范应用于获取符合条件的行的集合，来作为查询集合返回。
- 创建符合条件的行的 window 分区，并将指定的 OLAP 函数应用于每一分区结果集（或应用于这个查询结果集，如果未定义分区的话）中的每一行。
- 将 SELECT 语句的 ORDER BY 子句应用于最终的查询结果。

对于嵌套查询，每一子查询都遵循上述顺序，但将 OLAP window 分区及其 OLAP 函数应用于最里面的子查询的结果集，在其中定义该 OLAP window。

如果 OLAP window 包括 window ORDER 子句，则那个子句，而不是 SELECT 语句的 ORDER BY 子句，定义 window ROW_NUMBER 函数在同一 OLAP window 的分区中分配给这些行的行编号。然而，window ORDER 子句不定义查询结果集的排序，由 SELECT 语句的 ORDER BY 子句定义。

如果 OLAP window 不包括 window ORDER 子句，则以任意的顺序排列 window ROW_NUMBER 函数分配给这些行的行编号，如查询或子查询返回的那样，而不是根据 SELECT 语句的任何 ORDER BY 子句。

## 对 STANDARD 或 RAW 结果表排序

当 SELECT INTO Table 子句定义要存储查询的结果的永久表时，那个子句中的任何非平凡的列表达式还必须为新创建的结果表中相应的列声明别名。要指定那一列作为结果表的排序键，ORDER BY 子句还必须引用相同的别名，而不是指定非平凡的列表达式。

例如，在下列嵌套查询中，**tab56** 是结果表的标识符，且 **tab56_col0** 是子查询在 Projection 子句中定义的非平凡的列表达式的别名。ORDER BY 子句指定相同的子查询作为排序键，而不是通过它的别名引用那个非平凡的列：

```
SELECT ( SELECT tab54.tab54_col7 tab56_col0
      FROM tab54
      WHERE (tab54.tab54_col7 = -1423023 )
      ) tab56_col0,
   "" tab56_col1
FROM tab57
WHERE tab57.tab57_col1 == -6296233
ORDER BY   (
      SELECT tab54.tab54_col7 tab56_col0
      FROM tab54
```

```
    WHERE (tab54.tab54_col7 = -1423023 )
    ) NULLS FIRST,2 NULLS FIRST
INTO tab56;
```

在正常的 SELECT 语句中，在 ORDER BY 子句中指定非平凡的列表达式是可接受的，但在对一由 SELECT INTO Table 子句创建了的结果表进行排序的 ORDER BY 子句中是不可接受的。在上述示例中，数据库服务器返回 SQL 错误 -19828。

要避免此错误，必须修改上述示例，将非平凡的列表达式从 ORDER BY 子句移除，以它的别名取代那个表达式：

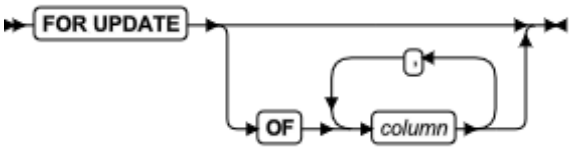
```
SELECT ( SELECT tab54.tab54_col7 tab56_col0
    FROM tab54
    WHERE (tab54.tab54_col7 = -1423023 )
    ) tab56_col0,
  "" tab56_col1
FROM tab57
WHERE tab57.tab57_col1 == -6296233
ORDER BY
    tab56_col0        -- Substituted alias for column expression in result table)
NULLS FIRST,2 NULLS FIRST
INTO tab56;
```

## FOR UPDATE 子句

当您打算更新由准备好的 SELECT 语句返回的值，当存取这些值时，请在 ESQL/C 应用中和在 DB-Access 中使用 FOR UPDATE 子句。

准备包含 FRO UPDATE 子句的 SELECT 语句，等同于准备不带有 FOR UPDATE 子句的 SELECT 语句，然后为准备好的语句声明 FOR UPDATE 游标。

FOR UPDATE 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 在 FETCH 之后可被更新的列的名称 | 必须在 FROM 子句 *table* 中存在，但不需要在 Projection 列表中。所有的列必须都来自同一表。 | 标识符 |

FOR UPDATE 关键字通知数据库服务器可能会有更新，导致它使用比随同 Select 游标更严格的锁。不带有此子句，您不可通过游标修改数据。您可指定哪些列可被更新。

在您为 SELECT . . . FOR UPDATE 语句声明游标之后，您可使用带有 WHERE CURRENT OF 子句的 UPDATE 或 DELETE 语句更新或删除当前选择了的行。关键字 CURRENT OF 引用最近存取了的行；它

们替代在 WHERE 子句中的通常的条件表达式。要以特定的值更新行，您的程序可能包含诸如下列示例中的语句：

```
EXEC SQL BEGIN DECLARE SECTION;
      char fname[ 16];
      char lname[ 16];
      EXEC SQL END DECLARE SECTION;
. . .

 EXEC SQL connect to 'stores_demo';
 /* select statement being prepared contains a for update clause */
 EXEC SQL prepare x from 'select fname, lname from customer for update';
 EXEC SQL declare xc cursor for x;

 for (;;)
    {
    EXEC SQL fetch xc into $fname, $lname;
    if (strncmp(SQLSTATE, '00', 2) != 0) break;
    printf("%d %s %s\n",cnum, fname, lname );
    if (cnum == 999)                    --update rows with 999 customer_num
          EXEC SQL update customer set fname = 'rosey' where current of xc;
    }

 EXEC SQL close xc;
 EXEC SQL disconnect current;
```

SELECT . . . FOR UPDATE 语句，像 Update 游标一样，允许您执行那些单独使用 UPDATE 语句不可能执行的更新，因为对更新的决定以及新的数据项的值都可基于该行的原始内容。UPDATE 语句不可查询正在被更新的表。

**注：** 在游标的 FETCH 循环内部的正常的更新不可确保在 UPDATE 之后再次存取更新了的行。WHERE CURRENT OF 规范将 UPDATE 联系到 Update 游标，并确保每一行仅更新一次，通过在内部保持一个已被更新了的行的列表。这些行将不被 Update 游标再次存取。

## 与 FOR UPDATE 子句不兼容的语法

包括 FOR UPDATE 子句的 SELECT 语句必须符合下列限制：

- 该语句可仅从一个表选择数据。
- 该语句不可包括任何聚集函数。
- 该语句不可包括任何下列子句或关键字：DISTINCT、EXCEPT、FOR READ ONLY、GROUP BY、INTO TEMP、INTERSECT、INTO EXTERNAL、MINUS、ORDER BY、UNION、UNIQUE。
- 将游标与该语句关联的 DECLARE 语句还不可包括 FOR UPDATE 关键字。
- 该语句仅在 ESQL/C 例程中和（在事务之内）在 DB-Access 实用程序中是有效的。例如，不可在 SPL 例程之内发出它。

要获取关于如何为一不包括 FOR UPDATE 子句的 SELECT 语句声明 update 游标的信息，请参阅使用 FOR UPDATE 选项。

**在 SPL 例程中更新游标**

您不可在 SPL 的 FOREACH 语句的 SELECT . . . INTO 段中包括 FOR UPDATE 关键字。然而，SPL 例程可提供 FOR UPDATE 游标的功能

- 通过在 FOREACH 语句中声明 ***cursor*** 名称，
- 然后使用 UPDATE 或 DELETE 语句中的 WHERE CURRENT OF ***cursor*** 子句，对同一 FOREACH 循环之内的那个 ***cursor*** 的当前行进行操作。

## FOR READ ONLY 子句

请使用 FOR READ ONLY 关键字来指定为 SELECT 语句声明了的 Select 游标是只读游标。 只读游标是不可修改数据的游标。此部分提供关于 FOR READ ONLY 子句的下列信息：

- 您何时必须使用 FOR READ ONLY 子句
- 对于使用 FOR READ ONLY 子句的 SELECT 语句的语法限制

## 以只读方式使用 FOR READ ONLY 子句

通常，您无需在 SELECT 语句中包括 FOR READ ONLY 子句。根据定义，SELECT 是只读操作，因此 FOR READ ONLY 子句通常是没有必要的。然而，在某些环境下，您必须在 SELECT 语句中包括 FOR READ ONLY 关键字。

在符合 ANSI 的数据库中，在缺省情况下，Select 游标是 update 游标。update 游标是可用来修改数据的游标。这些 update 游标与数据库的只读方式是不兼容的。例如，针对 customer_ansi 表的此 SELECT 语句失败：

```
EXEC SQL declare ansi_curs cursor for
        select * from customer_ansi;
```

解决方法是在您的 Select 游标中包括 FOR READ ONLY 子句。此子句指定的只读游标与数据库的只读模式相兼容。例如，下列针对 customer_ansi 表的 SELECT FOR READ ONLY 语句成功：

```
EXEC SQL declare ansi_read cursor for
        select * from customer_ansi for read only;
```

DB-Access 以 Select 游标执行所有的 SELECT 语句，因此，您必须在所有访问只读的符合 ANSI 的数据库中数据的查询中指定 FOR READ ONLY。FOR READ ONLY 子句导致 DB-Access 将 SELECT 语句的游标声明为只读游标。

要获取更多关于 0 级备份的信息，请参阅 *GBase 8s 备份与恢复指南*。要获取关于 Select 游标、只读游标和 update 游标的信息，请参阅 DECLARE 语句。

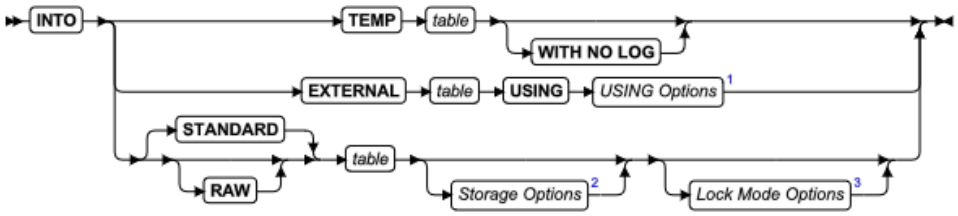
## 与 FOR READ ONLY 子句不兼容的语法

如果您尝试在同一 SELECT 语句中包括 FOR READ ONLY 子句和 FOR UPDATE 子句，则该 SELECT 语句失败。要获取关于为不包括 FOR READ ONLY 子句的 SELECT 语句声明只读游标的信息，请参阅 DECLARE 语句。

## INTO table 子句

使用 INTO Table 子句来创建新的临时的、永久的或外部的表来接收 SELECT 语句检索的数据。

INTO table




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *table* | 要接收查询结果的表在此声明的名称 | 在当前数据库中您拥有的表、视图、同义词和序列对象之中，必须是唯一的 | 标识符 |

您必须对要在其上创建临时的、永久的或外部表的数据库有 Connect 权限。在其他用户会话中的临时表的标识符之中，临时表的名称无需是唯一的。

在 Projection 子句中，必须指定在永久的、临时的或外部表中的列名称，在此，您必须为不是简单的列表达式的所有表达式提供显示标签。该显示标签成为在永久的、临时的或外部表中的列名称。如果您没有为简单的列表达式声明显示标签，则产生的新表使用 Projection 子句的选择列表中的列名称。

下列 INTO TEMP 示例创建 pushdate 表，带有两个列 customer_num 和 slowdate：

```
SELECT customer_num, call_dtime + 5 UNITS DAY slowdate
        FROM cust_calls INTO TEMP pushdate;
```

下列 INTO STANDARD 示例创建 **stab1** 表，带有两列 **fcol1** 和 **col2**：

```
SELECT col1::FLOAT fcol1, col2
        FROM tab1 INTO STANDARD stab1;
```

在此，**col1** 是该查询从其检索数据的 **tab1** 表中的 INTEGER 列，但 **fcol1** 值在产生的 **stab1** 表中被强制转型为 FLOAT。省略 STANDARD 关键字的查询会创建相同的结果表，因为 STANDARD 是缺省的表类型。

### 当没有返回行时的结果

当您使用与 WHERE 子句组合的 INTO Table 子句时，且没有返回行，则 SQLNOTFOUND 值在符合 ANSI 的数据库中是 100，在不符合 ANSI 的数据库中是 0。如果 SELECT INTO TEMP…WHERE… 语句是多语句 PREPARE 的一部分，且没有返回行，则对于符合 ANSI 的数据库和不符合 ANSI 的数据库，SQLNOTFOUND 值都是 100。

此 GBase 8s 版本在遇到 SQLNOTFOUND 值 100 之后，继续处理多语句准备好的对象的剩余的语句。然而，您可维持传统的行为，通过将 IFX_MULTIPREPSTMT 环境变量设置为 1，不执行剩余的准备好的语句。

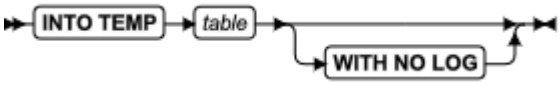
**对 ESQL/C 中的 INTO table 子句的限制**

在 GBase 8s ESQL/C 中，请不要在同一查询中同时使用 INTO ***table*** 子句与 INTO ***variable*** 子句。如果您同时使用，则不会向程序变量返回结果，且将 **SQLCODE** 变量设置为负值。要获取更多关于 INTO ***variable*** 子句的信息，请参阅 INTO 子句。

# INTO TEMP 子句

INTO TEMP 子句创建临时表来保存查询结果。

INTO TEMP 子句




INTO TEMP 子句创建的临时表的缺省的初始的 extent 和下一 extent 为每个 8 页。通过数据库服务器的内建的 RSAM 访问方式，该临时表必须是可访问的；您不可指定另一访问方式。

如果您使用同一查询结果一次以上，则使用临时表可节省时间。此外，使用 INTO TEMP 子句常常可以使 SELECT 语句更清晰和易于理解。

临时表中的数据值是静态的；当我们用来构建临时表的表发生更改时，临时表中的数据并不更新。您可使用 CREATE INDEX 语句来在临时表上创建索引。

日志记录的临时表一直存在，直到发生下列事件之一为止：

- 应用程序从数据库断开连接。
- 在临时表上发出 DROP TABLE 语句。
- 数据库关闭。

无日志记录的临时表存在，直到发生下列事件之一为止：

- 应用程序从数据库断开连接。
- 在临时表上发出 DROP TABLE 语句。

如果您的 GBase 8s 数据库没有事务日志记录，则临时表采取的行为与以 WITH NO LOG 选项创建的表的行为相同。

如果您在 **DBSPACETEMP** 环境变量中指定多个临时 dbspace（或如果未设置，在 DBSPACETEMP 配置参数中）， 则 INTO TEMP 子句将查询的结果集的行以轮询方式加载到这些 dbspace 内。要获取更多关于带有 INTO TEMP 子句的查询创建的临时表的存储位置的信息，请参阅 临时表的存储位置。

由于在无日志记录的临时表上的操作不做日志记录，所以使用 WITH NO LOG 选项会减轻事务日志记录的负荷。

由于当数据库被关闭时无日志记录的临时表不消失，所以您可使用无日志记录的临时表来在应用程序保持连接时将数据从一个数据库转移到另一个。您以 INTO TEMP 子句的 WITH NO LOG 选项创建的临时表的行为与 RAW 表的行为相似。

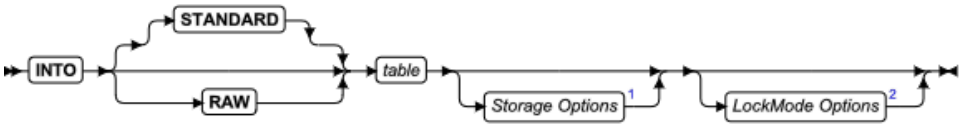
要获取更多关于临时表的信息，请参阅 CREATE TEMP TABLE 语句。

## INTO STANDARD 和 INTO RAW 子句

您可使用 INTO STANDARD 和 INTO RAW 子句来创建一个新的存储 SELECT 语句的结果集的永久表。

此语法提供单一机制来指定查询，来接收符合条件的记录，并将那些查询结果插入到永久的数据库表内。

INTO STANDARD 和 INTO RAW 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *table* | 为结果表在此声明的名称 | 在本地数据库中必须尚未存在 | 标识符 |

当使用 SELECT INTO 来创建新的永久表时，您可指定它的类型为 STANDARD 或 RAW。缺省的类型是 STANDARD。您可可选地指定新表的存储位置、extent 大小和锁模式选项。

新的永久表的列名称是在 Projection 子句的选择列表中指定的那些名称。如果星号（*）作为 Projection 子句的选择列表出现，则该星号扩展到 SELECT 语句的 FROM 子句中相应的表或视图的所有列名称。任何影子列都不会通过星号规范来扩展。

下列示例创建新的名为 **ptabl** 的 raw 表来存储连接查询的结果：

```
SELECT t1col1, t1col2, t2col1
        FROM tab1, tab2
        WHERE t1col1 < 100 and t2col1 > 5
        INTO RAW ptab1;
```

在上例中，新的 **ptabl** 表可能包含列 **t1col1**、**t1col2** 和 **t2col1**。

除了简单的列表达式之外的所有表达式都必须有一个定义在 Projection 子句中的显示标签。此显示标签被用作新表中的列的名称。如果简单的（或平凡的）列表达式没有显示标签，则该表使用列名称。如果在选择列表中有重复的显示标签或列名称，则返回错误。

下一示例失败并返回错误 -249，因为它没有为 **col1+5** 表达式声明显示标签：

```
SELECT col1+5, col2
        FROM tab1
        INTO ptab1;
```

下列修订的查询避免在前一示例中的 -249 错误：

```
SELECT col1+5 pcol1, col2
        FROM tab1
        INTO ptab1;
```

上述修正的示例创建标准 **ptabl** 表来在它的列 **pcol1** 和 **col2** 中存储查询结果。

**对结果表的限制**

与大部分 DDL 语句一样，使用完全符合条件的表名称在另一数据库中创建新的结果表的尝试失败并返回语法错误。

要以与同一数据库中现有的表相同的名称创建结果表，也会发生类似的错误。

SELECT INTO . . . TABLE 语句不可被用作子查询的一部分。

然而，您可使用不是 SELECT 子句的 projection 列表的一部分的列作为 ORDER BY 子句中的排序键。

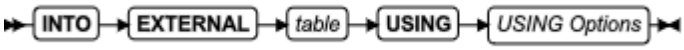
要获取对于可类似地创建查询结果表并通过插入符合条件的行来填充那表的 CREATE TABLE 语句语法的描述，请参阅 AS SELECT 子句。

## INTO EXTERNAL 子句

INTO EXTERNAL 子句将查询结果卸载到外部表内，创建缺省的外部表描述，当您之后重新加载这些文件时可以使用。

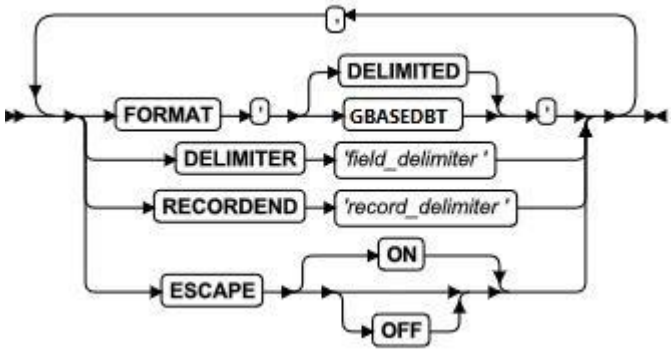
使用 SELECT INTO EXTERNAL 语句的 Table Options 子句来指定在外部表中卸载的数据的格式。

INTO table 子句




USING Options


Table Options




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *field_delimiter* | 分隔字段的字符。缺 | 如果您未设置 | 引用字 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 省为管道（\|）字符 | RECORDEND 环境变量，则 *record_delimiter* 的缺省值为换行字符（CTRL-J）。<br>如果您使用一个不可打印的字符作为定界符，则将它编码作为该 ASCII 字符的八进制表示。例如，'\006' 可表示 CTRL-F。 | 符串 |
| *record_delimiter* | 分隔记录的字符 | | 引用字符串 |
| *table* | 在此声明的要接收查询结果的表的名称 | 在当前数据库中您拥有的表、视图、同义词和序列对象的名称之中必须是唯一的 | 数据库对象名 |

INTO EXTERNAL 子句将 CREATE EXTERNAL TABLE . . . SAMEAS 与 INSERT INTO . . . SELECT 语句的功能组合在一起。

INTO EXTERNAL 子句重写在 EXTERNAL 表中任何先前存在的行。

下表描述应用于卸载数据的关键字。如果您想在外部表描述中为之后重新加载该表指定附加的表选项，则请参阅 Table 选项。

在 SELECT ... INTO EXTERNAL 语句中，您可指定除了固定的格式选项之外的在 CREATE EXTERNAL TABLE 语句中讨论的任何表选项。

当创建的数据文件的格式类型或为定界的文本（如果您使用 DELIMITED 关键字的话）或 GBase 8s 内部的数据格式的文本（如果您使用 GBASEDBT 关键字的话）时，您可使用 INTO EXTERNAL 子句。您不可将它用于固定的格式卸载。

| 关键字 | 作用 |
| --- | --- |
| DELIMITER | 指定在定界的文本文件中分隔字段的字符 |
| ESCAPE | 指示数据库服务器识别在基于 ASCII 文本的数据文件中字段之间作为分隔符嵌入的 ASCII 特殊字符。 紧接在 DELIMITER 指定的 ***field_delimiter*** 分隔符的任何实例之前的缺省的转义字符，此处的那个字符是该数据中的字母值。 或者您包括或者您省略 ESCAPE 关键字，在缺省情况下，此功能是被启用的，或者您可指定 ESCAPE ON 关键字来使得您的 SQL 代码的读者更清楚地了解启用 |

了此特性。要防止在该数据中的字母的 ***field_delimiter*** 分隔符字符被转义，您必须指定 ESCAPE OFF 关键字。

在缺省情况下，ESCAPE 关键字在字母的 ***field_delimiter*** 字符之前插入的转义字符是反斜杠（\）字符。但如果将 DEFAULTESCCHAR 配置参数设置为单个字符值，则以那个字符取代定界符字符的反斜杠（\）用作字母，当指定 ESCAPE 或 ESCAPE ON 时。

FORMAT　　指定在该数据文件中的数据的格式

RECORDEND　　指定在定界的文本文件中分隔记录的字符

要获取更多关于外部表的信息，请参阅 CREATE EXTERNAL TABLE 语句。

## 在组合查询中的集合运算符

集合运算符 UNION、UNION ALL、INTERSECT 和 MINUS 可操作指定 Projection 子句中的相同数目的列的两个查询的结果集，且在两个查询的相应的列中有可兼容的数据类型。

（MINUS 集合运算符有 EXCEPT 作为它的关键字同义词。MINUS 和 EXCEPT 运算符从相同的运算对象返回的结果通常是相同的。）

这些运算符对两个查询的结果集执行基本的集合操作***并集***、***交集***和***差集***，这两个结果集是这些集合运算符的左运算对象和右运算对象：

- UNION 集合运算符将来自两个查询的符合条件的行组合到单个结果集内，该结果集由一个查询返回的或两个查询都返回的不重复的行组成。（如果您还包括 ALL 关键字，则 UNION ALL 结果集可包括重复的行。）
- INTERSECT 集合运算符比较来自两个查询的结果集，但仅返回同时在两个查询的结果集中的不重复的行。
- MINUS 集合运算符比较来自两个查询的结果集，但仅返回在左边的查询的结果集中但不在右边的查询的结果集中的不重复的行。

在业务分析上下文中，集合运算符是有用的。它们还可用在 SELECT 语句中，在已执行了诸如 UPDATE、INSERT、DELETE 或 MERGE 这样的 DML 操作之后，来检查数据库的完整性。当您将数据转移到历史表时，可类似地使用集合运算符，例如，当您需要在从原始的表删除行之前确认在历史表中有正确的数据的时候。

所有集合运算符有相同的优先顺序。在包括多个集合运算符的复杂的查询中，运算符的优先顺序为从左至右。请使用括号来将集合运算符及其运算对象分组，如果您需要覆盖集合运算符的缺省的从左至右的优先顺序的话。

仅 UNION 集合运算符支持 ALL 关键字。带有 INTERSECT、MINUS 或 EXCEPT 集合运算符的 ALL 关键字是无效的，仅从其返回不重复的行。

当比较行来计算集合并集或差集时，在 INTERSECT 和 MINUS 操作中的两个 NULL 值被认为是相等的。

# 对组合 SELECT 的限制

对于您通过 UNION、INTERSECT、MINUS 或 EXCEPT 集合运算符进行连接的查询，会受到以下限制：

- 每条查询的 Projection 子句中的项数必须相同，且每条 Projection 子句中的相应项必须有兼容的数据类型。
- 每条查询的 Projection 子句不能指定 BYTE 或 TEXT 对象（此限制不适用于 UNION ALL 操作。）
- 如果使用 ORDER BY 子句，它必须紧跟在最后一条 Projection 子句之后，而且必须引用整数 ***select_number*** 排序（而不是按 SQL 标识符排序）的项。设置操作完成后，就开始排序。
- 可以在临时表中存储任何集合运算符的组合结果，但 INTO TEMP 子句只能出现在最后一条 SELECT 语句中。
- 在 GBase 8s ESQL/C 中，除非返回的行只有一行，而且您没有使用游标，否则不能将 INTO 子句用于复合查询。在这种情况下，INTO 子句必须在第一条 SELECT 语句中。

***UNION 子查询***是在子查询中包括 UNION 运算符的查询。在定义视图的 CREATE VIEW 语句中，不能指定 UNION 子查询。

在组合查询中，本版 GBase 8s 支持在分布式查询中引用 UNION 子查询。即，可以在本地服务器的不同实例间，或跨服务器的实例间包含 UNION 子查询。

下列限制影响所有组合的查询，包括 UNION 和 UNION ALL 子查询，以及包括 INTERSECT、MINUS 或 EXCEPT 集合运算符的查询：

- UNION 子查询不能是触发器事件。如果有效 UNION 子查询指定已在其中定义 Select 触发器，那么该查询成功，但是忽略该触发器（或视图上的 INSTEAD OF 触发器）。
- 在包含 UNION 子查询或任何其他集合运算符的查询中，在 ALL、ANY、IN、NOT IN 和 SOME 运算符的左边，包括主变量的通用的表达式是无效的。但是只由单个主变量组成的表达式在此上下文中有效。

例如，在上述限制之下，下列查询是有效的：

```
SELECT col1 FROM tab1 WHERE ?  <= ALL
        (SELECT col2 FROM tab2 UNION SELECT col3 FROM tab3);
```

在此示例中，ALL 左边的表达式是单个主变量(?)，这是包含 UNION 子查询的查询中 ALL、ANY、IN、NOT IN 或 SOME 运算符之前受支持的唯一涉及主变量的表达式。

相反，下列示例显示了无效查询：

```
SELECT col1 FROM tab1 WHERE (? + 8) <= ALL
        (SELECT col2 FROM tab2 UNION SELECT col3 FROM tab3);
```

该查询失败是因为在 ALL 运算符左边的 <= 关系运算符的操作数是(? + 8)（包含主变量的算术表达式）。这在 UNION 子查询中是无效语法，在通过任何其他集合运算符组合查询中也无效。

不包含主变量的表达式不遵守此限制。因此，下列（包括相同的 UNION 子查询的）查询有效：

```
SELECT col1 FROM tab1 WHERE (col1 + 8) <= ALL
        (SELECT col2 FROM tab2 UNION SELECT col3 FROM tab3);
```

## UNION 运算符

将 UNION 运算符放在两条 SELECT 语句之间来将查询组合到单个查询内。

可以使用 UNION 运算符将几个 SELECT 语句串在一起。相应的项无需具有相同的名称。可通过省略 ALL 关键字来排除重复的行。

在本版本 GBase 8s 中，在包含 UNION 运算符的查询中支持数值型字段和数值字符串的组合查询。其返回结果的为 DECIMAL(16) 或 DECIMAL(32) （取决于返回的数值的长度）数据类型。例如：

```
SELECT 123 FROM tab1 UNION SELECT '456' FROM tab2;
```

下一示例假定 tab1 表中 colid 列为 INT 类型：

```
SELECT col1id FROM tab1 UNION SELECT '111' FROM tab2;
```

使用 UNION 时还可以直接将 NULL 值与其它类型的列值进行组合查询，而无需强制转换 NULL 的数据类型。其返回结果的类型与 NULL 值对应的其它类型值的数据类型一致。

例如，在以下示例中假定 tab2 表中 colid 的类型为 VARCHAR：

```
SELECT NULL as a1 FROM tab1
UNION SELECT colname FROM tab2
INTO tab3;
```

数据库会自动将上面示例中 NULL 的类型转换为 colname 列的数据类型 VARCHAR。查看 tab3 表的信息：

```
info columns for tab3;
```

返回如下：

```
Column name          Type              Nulls
A1                   varchar(10)
```

### *UNION ALL 运算符*

如果您使用 UNION ALL 运算符，则从两个查询返回所有符合条件的行，而不排除任何重复的行。（如果您使用 UNION 运算符而不带 ALL 关键字来组合两个查询，则从符合条件的行的组合的集合移除任何重复的行。也就是说，如果有多行，它们每一列包含的值均完全相同，那么只保留一行。）

下一示例使用 UNION ALL 来组合两个 SELECT 语句的结果，而不移除重复行。该查询返回在 2007 年第一季度与 2008 年第一季度期间接收的所有电话的列表。

```
SELECT customer_num, call_code FROM cust_calls
        WHERE call_dtime BETWEEN
        DATETIME (2007-1-1) YEAR TO DAY
        AND DATETIME (2007-3-31) YEAR TO DAY
  UNION ALL
  SELECT customer_num, call_code FROM cust_calls
```

```
    WHERE call_dtime BETWEEN
    DATETIME (2008-1-1)YEAR TO DAY
    AND DATETIME (2008-3-31) YEAR TO DAY;
```

如果想要从结果集移除重复行，请使用不带关键字 ALL 的 UNION 作为查询之间的集合运算符。在前一示例中，如果两个 SELECT 语句都返回了组合 `101 B`，则 UNION 运算符会导致该组合只罗列一次。（如果您想要移除每一 SELECT 语句中的重复行，则请紧接在 Projection 子句的 Select 列表之前使用 DISTINCT 或 UNIQUE 关键字，如同 允许重复 中描述的那样。）

对于指定仅带有 UNION 运算符的集合操作，ALL 关键字是有效的。如果 ALL 紧跟在 INTERSECT、MINUS 或 EXCEPT 集合运算符之后，这些集合运算符排除重复的部分，则数据库服务器发出错误。

要获取关于数据库服务器如何在有 NLCASE INSENSITIVE 属性的数据库中标识重复的 NCHAR 或 NVARCHAR 值的信息，请参阅主题 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

### *子查询中的 UNION*

您可在 WHERE 子句、FROM 子句之内的 SELECT 语句的子查询中，以及在集合子查询中使用 UNION 和 UNION ALL 运算符。然而，在此 GBase 8s 版本中，在下列上下文中不支持包含 UNION 或 UNION ALL 的子查询：

- 在视图的定义中
- 在触发器的事件或 Action 子句中
- 使用 FOR UPDATE 子句或使用 Update 游标

有关集合子查询的信息，请参阅 集合子查询。关于 FOR UPDATE 子句的更多信息，请参阅 FOR UPDATE 子句。

特别地是，在本版本数据库中支持在分布式查询中包含 UNION 的子查询。

在组合的子查询中，数据库服务器只能在列的限定表引用的作用域中解析出列名。例如，下列查询返回错误：

```
SELECT * FROM t1 WHERE EXISTS
    (SELECT a FROM t2
UNION
SELECT b FROM t3 WHERE t3.c IN
        (SELECT t4.x FROM t4 WHERE t4.4 = t2.z));
```

在此，不可解析最内部的子查询 **t2.z**，因为 **z** 发生在表引用 **t2** 的引用范围之外。在最内部的子查询中，仅可解析属于 **t4**、**t3** 或 **t1** 的列引用。表引用的作用域通过子查询向下扩展，但不越过 UNION 运算符扩展到兄弟 SELECT 语句。

## INTERSECT 运算符

当通过此集合运算符组合两个查询时，INTERSECT 计算通过为其运算对象的两个查询返回的行的交集。

INTERSECT 返回的行展现在左边和右边的 SELECT 语句的结果集中。INTERSECT 结果通常是不同的或唯一的行，因为 INTERSECT 消除任何重复的行。

请考虑下列示例，其中表 **t1** 有下列行：

```
create table t1 (col1 int);
insert into t1 values (1);
insert into t1 values (2);
insert into t1 values (2);
insert into t1 values (2);
insert into t1 values (3);
insert into t1 values (4);
insert into t1 values (4);
insert into t1 values (NULL);
insert into t1 values (NULL);
insert into t1 values (NULL);
```

在同一示例中，表 **t2** 有这些行：

```
create table t2 (col1 int);
insert into t2 values (1);
insert into t2 values (3);
insert into t2 values (4);
insert into t2 values (4);
insert into t2 values (NULL);
```

下列查询从 INTERSECT 操作对象的左边与右边的两个查询返回不同的行。在此要注意的重要问题是该结果有 NULL 值。因为当将表 **t2** 与表 **t1** 进行比较时，考虑到表 **t2** 中的 NULL 值是相等的，因此来自该交集的 NULL 值返回在组合的结果集中：

```
SELECT col1 FROM t1 INTERSECT SELECT col1 FROM t2;

      col1
         1
         3
         4
4 row(s) retrieved.
```

INTERSECT 运算符有一些（但不是所有）与 UNION 运算符相同的限制，但 INTERSECT 不支持使得 UNION 能够返回重复的值的 ALL 关键字。另请参阅主题 对组合的 SELECT 的限制。

## MINUS 运算符

当通过此集合运算符来组合两个查询时，MINUS 运算符计算通过左边的 SELECT 语句返回的行与通过右边的 SELECT 语句返回的行之间的差集。

MINUS 仅返回出现在第一个结果集但不在第二个集合中的那些行。MINUS 结果通常是不同的或唯一的行，因为 MINUS 消除任何重复的行。

对于罗列在 INTERSECT 运算符 主题中的同一数据集，下列查询从 MINUS 运算符左边的查询的结果集返回不在右边的查询的结果集中的所有不同的行：

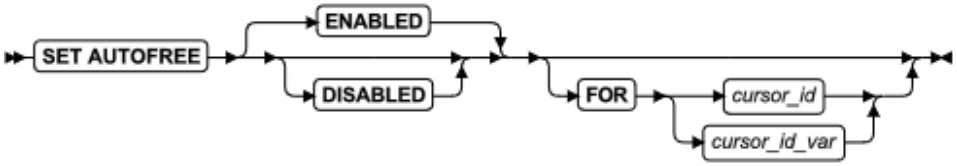
```
SELECT col1 FROM t1 MINUS SELECT col1 FROM t2;

    col1
    2
1 row(s) retrieved.
```

**MINUS 运算符有一些（但不是所有）与 UNION 运算符相同的限制，但 MINUS 不支持使得 UNION 能返回重复的值的 ALL 关键字。另请参阅主题 对组合的 SELECT 的限制。**

## 2.124 SET AUTOFREE 语句

使用 SET AUTOFREE 语句来指示数据库服务器启用或禁用内存管理特性，一旦游标关闭，该特性可自动地释放为该游标分配的内存。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *cursor_id* | Autofree 要为其重设的游标的名称 | 必须已在程序中声明了 | 标识符 |
| *cursor_id_var* | 持有 *cursor_id* 的值的主变量 | 必须存储在程序中已声明了的 cursor_id | 必须符合名称的特定语言的规则。 |

**用法**

此语句是对 SQL 的 ANSI/ISO 标准的扩展。您仅可随同 GBase 8s ESQL/C 使用此语句。

当为游标启用 Autofree 特性且该游标随后关闭时，您不需要显式地使用 FREE 语句来释放数据库服务器为该游标分配的内存。如果您发出 SET AUTOFREE 但未指定选项，则缺省为 ENABLED。

启用 Autofree 特性的 SET AUTOFREE 语句必须出现在打开游标的 OPEN 语句之前。SET AUTOFREE 语句不影响分配给已经打开的游标的内存。在启用游标的 Autofree 之后，您不可第二次打开那个游标。

## 以 SET AUTOFREE 全局地影响游标

如果您未包括 FOR cursor_id 或 FOR cursor_id_var 子句，则 SET AUTOFREE 的范围是该程序中所有后续声明的游标（或更准确地说，在不带有 FOR 子句的后续的 SET AUTOFREE 语句之前声明的所有游标全局地重置 Autofree 特性）。此示例为程序中所有后续的游标启用 Autofree 特性：

```
EXEC SQL set autofree;
```

下一示例为所有后续的游标禁用 Autofree 特性：

```
EXEC SQL set autofree disabled;
```

## 使用 FOR 子句来指定特定的游标

如果您指定 FOR ***cursor _id*** 或 FOR ***cursor_id_var***，则 SET AUTOFREE 仅影响您在 FOR 关键字之后指定的游标。

此选项允许您覆盖对所有游标的全局的设置。例如，如果您在程序中为所有游标发出 SET AUTOFREE ENABLED 语句，则可发出后续的 SET AUTOFREE DISABLED FOR 语句来为特定的游标禁用 Autofree 特性。

在下列示例中，第一个语句为所有游标启用 Autofree 特性，而第二个语句为名为 **x1** 的游标禁用 Autofree 特性：

```
EXEC SQL set autofree enabled;
EXEC SQL set autofree disabled for x1;
```

在此，必须已声明了但尚未打开 **x1** 游标。

## 关联的和拆离的语句

当自动地释放游标时，也释放它的关联的准备好的语句（或关联的语句）。

在 Autofree 特性的上下文中，术语***关联的语句***有特殊的含意。如果游标是您以准备好了的语句声明的第一个游标，或如果它是您在该语句拆离之后以该语句声明的第一个游标，则以准备好了的语句关联该游标。

在 Autofree 特性的上下文中，术语***拆离的语句***有特殊的含意。如果您不以准备好了的语句声明游标，或如果释放了以其关联语句的游标，则拆离该准备好了的语句。

如果对游标启用 Autofree 特性，该有表有相关联的准备好了的语句，且那个游标关闭，则数据库服务器释放分配给该准备好了的语句的内存以及分配给该游标的内存。假设您为下列游标启用 Autofree 特性：

```
/*Cursor associated with a prepared statement */
        EXEC SQL prepare sel_stmt 'select * from customer';
        EXEC SQL declare sel_curs2 cursor for sel_stmt;
```

当数据库服务器关闭 sel_curs2 游标时，它等同于自动地执行下列 FREE 语句：

```
FREE sel_curs2;
FREE sel_stmt;
```

由于自动地释放了 **sel_stmt** 语句的内存，因此您不可在其上声明新的游标，除非您再次准备该语句。

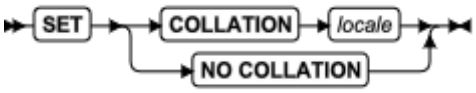
### 隐式地关闭游标

启用了特性的游标存在潜在的问题。在不符合 ANSI 的数据库中，如果您不显式地关闭游标然后再打开它，则隐式地关闭该游标。对游标的这种隐式的关闭触发 Autofree 特性。第二次打开该游标时，数据库服务器会生成错误消息（`cursor not found`），因为该游标已被释放。

## 2.125 SET COLLATION 语句

使用 SET COLLATION 语句来指定该会话的新的对照顺序，取代通过 DB_LOCALE 环境变量设置暗示的对照。SET NO COLLATION 恢复缺省的对照。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *locale* | 要在此会话中使用其对照顺序的语言环境的名称 | 必须为数据库服务器可访问的语言环境的名称 | 引用字符串 |

**用法**

SET COLLATION 语句是对 SQL 的 ANSI/ISO 标准的扩展。您可随同 GBase 8s ESQL/C 使用此语句。

如同 *GBase 8s GLS 用户指南* 所解释的那样，数据库服务器使用语言环境文件来指定字符集、对照顺序和显示与操作字符串级其他数据值的一些自然语言的其他约定。数据库语言环境的对照顺序是数据库服务器据其对字符串排序的次序顺序。

如果您未设定 DB_LOCALE 的值，则基于 United States English，对于 UNIX™ 的缺省的语言环境为 **en_us.8859-1** ，对于 Windows™ 系统的语言环境为 Code Page 1252 。否则，数据库服务器使用 DB_LOCALE 设置作为它的语言环境。在运行时，对于在同一会话中先前访问的所有数据库服务器，SET COLLATION 语句覆盖 DB_LOCALE 的对照顺序。

对于余下的会话，该新对照顺序保持有效，或直到您发出另一 SET COLLATION 语句为止。不影响其他的会话，但您以非缺省的对照创建的数据库对象使用在他们创建时有效的任何对照顺序。

在缺省情况下，对照顺序是代码集顺序，但有些语言环境还支持特定的语言环境顺序。在大多数上下文中，仅 NCHAR 和 NVARCHAR 数据值可根据特定于语言环境的对照顺序存储。

### 以 SET COLLATION 指定对照顺序

对当前会话中先前访问的所有数据库服务器，SET COLLATION 以 *locale* 指定的对照顺序替代当前的对照顺序。 例如，此示例指定中文的对照顺序：

```
EXEC SQL set collation "zh_cn.gb18030-2000";
```

如果在此会话中的下一数据库服务器的活动是对 NCHAR 或 NVARCHAR 值排序，则遵循中文的对照顺序。

在同一会话中，假设下列 SET NO COLLATION 语句恢复对对照顺序的 DB_LOCALE 设置：

```
EXEC SQL set no collation;
```

在 SET NO COLLATION 执行之后，在同一会话中的后续的对照基于该 **DB_LOCALE** 设置。然而，您使用中文对照顺序创建的任何数据库对象，诸如检查约束、索引、准备好的对象、触发器或 UDR，会继续将中文对照应用于 NCHAR 和 NVARCHAR 数据类型。

**在 NLSCASE INSENSITIVE 数据库中的对照**

在 NLSCASE INSENSITIVE 数据库中，在 NCHAR 和 NVARCHAR 数据上的对照操作不区分字母大小写，以便于数据库服务器将有相同序列的字母组成的字符串中的大小写变量按重复的字符串处理。对照的列表按照其检索的顺序来对这些区分大小写的重复的内容排列，因此，带有字符串 alpha 的大小写变量的对照的列表可能以任何顺序出现，比如下列顺序，不管变量的大小写：

```
alpha
Alpha
alpha
ALPHA
Alpha
```

要获取更多信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## 对 SET COLLATION 的限制

虽然 SET COLLATION 使得您能在会话内动态地更改数据库服务器的对照顺序，但您应了解对 SET COLLATION 语句可发挥作用的范围的几个限制。

- 仅数据库服务器执行的对照受影响。对数据排序的客户端进程不受 SET COLLATION 影响。
- 仅当前会话受影响。其他会话不受您的 SET COLLATION 语句的直接影响，但数据库服务器会使用它们的创建时对照顺序来设置任何数据库对象，这些数据库对象是您在已成功地运行了 SET COLLATION 之后创建的那些。
- 对对照顺序的更改不更改代码集。数据库服务器通常使用由 **DB_LOCALE** 指定的代码集。
- 仅按照特定于语言环境的顺序对 NCHAR 和 NVARCHAR 值进行排序。

处理来自不同的代码集的字符

由于 SET COLLATION 仅更改对照顺序，而不是当前的语言环境或代码集，因此您通常不可使用此语句来将来自不同的语言环境的字符数据插入到同一数据库之内。如果数据库需要存储来自两种或多种语言的字符，这些语言内在地需要不同的代码集或代码页，则您必须改为使用支持 Unicode 的语言环境。对于 GBase 8s ESQL/C 应用，以及对于使用 GBase 8s GLS 库的其他客户端应用，其语言环境支持 UTF-8 字符编码的数据库可存储相应于来自多种自然语言的不同的字符集的代码点的字符，但仅当下列条件全都满足时：

- 当数据库服务器实例启动时，设置 GL_USEGLU 环境变量为 1。
- 当创建数据库时，设置 DB_LOCALE 环境变量为有效的 Unicode 语言环境。
- 设置 CLIENT_LOCALE 环境变量为数据库服务器的 DB_LOCALE 设置支持的有效的 Unicode 语言环境。

对于 GBase 8s 要使用“Unicode 的国际组件”（ICU）4.8.1 库来支持最高达 6.0 的 Unicode 的版本，在启动服务器之前，在服务器环境中必须将 GL_USEGLU 环境变量设置为值 1（一）。在使用 **UTF-8** 字符编码的数据库中，此设置初始化启用 Unicode 对照和 SQL 操作的那些转换规则，包括 Chinese **GB18030-2000** 代码集。此转换仅适用于以已设置的 GL_USEGLU=1 创建了的数据库。

**注意：** 然而，GL_USEGLU 环境变量对 JDBC 客户端应用不起作用 ，包括 GBase 8s JSON 兼容性线协议 listener 的那些应用。要在 Unicode 语言环境中正确地支持 JDBC 应用，没有在客户端或在服务器环境中将 GL_USEGLU 设置为 1 的要求。

### 由数据库对象执行的对照

虽然在会话结束之后（或您执行 SET NO COLLATION 之后）数据库服务器恢复到 **DB_LOCALE** 对照顺序，但您使用非缺省的对照创建的对象仍留在数据库中。例如，您可在相同的列的集合上，使用 SET COLLATION 指定的不同的对照顺序来创建多索引，称为***多语言索引***。

然而，在给定的列的集合上，仅可存在一个集群的索引。

在给定的列的集合上，仅可存在一个唯一的约束或主键，但您可在同样的列的集合上创建多个唯一的索引，如果每一索引有不同的对照顺序的话。

当计算查询的成本时，对于那些将任何非当前会话对照的任何对照应用于 NCHAR 或 NVARCHAR 列，查询优化器不顾及这些索引。

附加的索引的对照顺序必须与它的表的相同，且此必须为由 **DB_LOCALE** 指定的缺省的对照顺序。

ALTER INDEX 语句不可更改索引的对照。当 ALTER INDEX 执行时，不管任何先前的 SET COLLATION 语句。

当您将来自 CHAR 列的值与 NCHAR 列比较时， GBase 8s 强制将 CHAR 值转型为 NCHAR，然后再应用当前的对照。类似地，在比较 VARCHAR 与 NVARCHAR 值之前， GBase 8s 首先将 VARCHAR 值强制转型为 NVARCHAR。

当为远程表或视图创建同义词时，参与的数据库必须有相同的对照顺序。然而，现有的同义词可用在支持 SET COLLATION 的其他数据库中，且该同义词的对照顺序不顾及 **DB_LOCALE** 的设置。

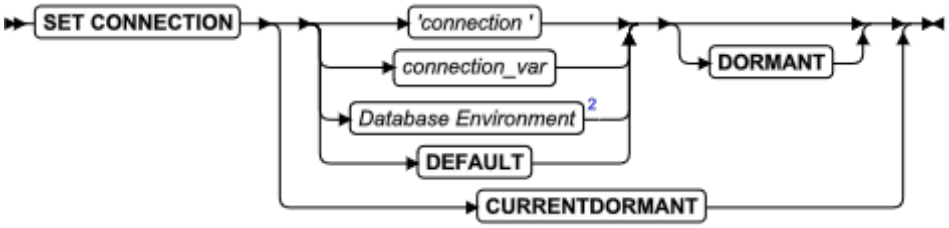
对 NCHAR 或 NVARCHAR 值排序的检查约束、游标、准备好的对象、触发器和 SPL 例程使用在它们创建时刻生效的对照，如果这与 **DB_LOCALE** 设置不同的话。

当创建按本地化的顺序排序的数据库对象时，有多少不同的对照会对性能有灵敏的影响。

## 2.126 SET CONNECTION 语句

使用 SET CONNECTION 语句来重新建立在应用于数据库环境之间的连接，并使该连接成为当前连接。您还可随同 DORMANT 选项使用此语句来将当前的连接置于休眠状态。请以 GBase 8s ESQL/C 使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *connection* | CONNECT 语句建立的初始连接的名称 | 数据库必须已存在 | 引用字符串 |
| *connection_var* | 包含 *connection* 值的主变量 | 必须为字符数据类型 | 特定于语言 |

用法

您可使用 SET CONNECTION 来使休眠连接成为当前连接，或使当前连接休眠。

SET CONNECTION 不是有效的准备好的语句。

## 使休眠连接成为当前的连接

如果您使用不带 DORMANT 选项的 SET CONNECTION 语句，则 ***connection*** 必须表示休眠连接。***休眠连接***是已建立但不是当前连接的连接。

不带 DOMANT 选项的 SET CONNECTION 语句使指定的休眠连接成为当前的连接。应用指定的连接必须是休眠的。当该语句执行时，当前的连接成为休眠的。

在下列示例中的 SET CONNECTION 语句使连接 con1 成为当前的连接，并使 con2 成为休眠的连接：

```
CONNECT TO 'stores_demo' AS 'con1';
        ...
CONNECT TO 'demo' AS 'con2';
        ...
SET CONNECTION 'con1';
```

休眠的连接有一***连接上下文***与它相关联。当应用使休眠的连接成为当前的时，它建立到数据库环境的那个连接，并恢复它的连接上下文。（要获取更多关于连接上下文的信息，请参阅在 CONNECT 语句 页上的 CONNECT 语句 语句。）重新建立连接与建立初始的连接是可比的，除了它典型地为用户避免再次认证许可，且它避免重新分配与初始的连接相关联的资源之外。例如，应用不需要准备在该连接中先前已准备好了的任何语句，也不需要重新声明任何游标。

## 使当前的连接成为休眠的连接

在 SET CONNECTION ***connection*** DORMANT 语句中，***connection*** 必须表示当前的连接。带有 DORMANT 选项的 SET CONNECTION 语句使指定的当前连接成为休眠的连接。

例如，下列 SET CONNECTION 语句使连接 con1 休眠：

```
SET CONNECTION 'con1' DORMANT;
```

如果您指定一个已经是休眠的连接，则带有 DORMANT 选项的 SET CONNECTION 语句生成错误。例如，如果连接 con1 是当前的，而连接 con2 是休眠的，则下列 SET CONNECTION 语句返回错误消息：

```
SET CONNECTION 'con2' DORMANT;
```

下列 SET CONNECTION 语句成功地执行：

```
SET CONNECTION 'con1' DORMANT;
```

## 单线程环境中的休眠连接

在单线程 GBase 8s ESQL/C 应用（不使用线程的）中，DORMANT 选项使当前的连接成为休眠的。使用此选项使单线程的 GBase 8s ESQL/C 应用与线程安全的 GBase 8s ESQL/C 应用向上兼容。然而，在程序执行时，单线程的环境可仅有一个活动的连接。

## 在线程安全环境中的休眠连接

在线程安全的 GBase 8s ESQL/C 应用中，DORMANT 选项使活动的连接成为休眠的。现在另一线程可通过发出不带 DORMANT 选项的 SET CONNECTION 语句来使用该连接。在一个 GBase 8s ESQL/C 应用中，线程安全的环境可有许多线程（执行特别任务的工作的并发部分），且每一线程可有一个活动的连接。

活动的连接与特定的线程相关联。两个线程不可分享相同的活动的连接。一旦线程使活动的连接成为休眠的，其他线程就可使用那个连接。仍然建立休眠的连接，但当前休眠的连接不与任何线程相关联。例如，如果在名为 thread_1 的线程中名为 con1 的连接是活动的，则名为 thread_2 的线程不可使连接 con1 成为它的活动连接，直到 thread_1 已使连接 con1 成为了休眠的为止。

下列来自线程安全的 GBase 8s ESQL/C 程序的代码片段展示在线程安全的应用之内，特定的线程是如何使连接成为活动的，通过此连接在表上执行工作，然后再使该连接成为休眠的，以便其他线程可使用该连接：

```
thread_2()
        {   /* 使 con2 成为活动的连接 */
        EXEC SQL connect to 'db2' as 'con2';
        /*Do insert on table t2 in db2*/
        EXEC SQL insert into table t2 values(10);
        /* 使其他线程可使用 con2 */
        EXEC SQL set connection 'con2' dormant;
        }
```

如果使用 CONNECT . . . WITH CONCURRENT TRANSACTION 语句初始化了到数据库环境的连接，则随后连接到那个数据库环境的任何线程都可使用正在进行的事务。此外，如果开放的游标与这样的连接相关联，则当使该链接成为休眠的时，该游标保持打开。

在线程安全的 GBase 8s ESQL/C 应用之内的线程可通过使相关联的连接成为当前的来使用同一游标，即使在任何给定的时间仅一个线程可使用该连接。

## 标识连接

如果应用未在初始的 CONNECT 语句中指定连接名称，则您必须使用数据库环境（诸如数据库名称或数据库路径名称）作为连接名称。例如，下列 SET CONNECTION 语句为连接名称使用数据库环境，因为 CONNECT 语句不使用连接名称。要获取更多关于指定数据库环境的加引号的字符串的信息，请参阅 数据库环境。

```
CONNECT TO 'stores_demo';
        ...
CONNECT TO 'demo';
        ...
SET CONNECTION 'stores_demo';
```

然而，如果为到数据库服务器的连接指定连接名称，则您必须使用该连接名称来重新连接到数据库服务器。当连接名称存在时，如果您使用数据库环境而不是连接名称，则返回错误。

## DEFAULT 选项

DEFAULT 选项指定 SET CONNECTION 语句的缺省的连接。缺省的连接是下列连接之一：

- 显式的缺省连接（以 CONNECT TO DEFAULT 语句建立的连接）
- 隐式的缺省连接（以 DATABASE 或 CREATE DATABASE 语句建立的任何连接）

使用不带 DOMANT 选项的 SET CONNECTION 来重新建立缺省的连接，或以那个选项来使缺省的连接成为休眠的。

要获取更多信息，请参阅 缺省连接规范 和 使用 DATABASE 语句的隐式连接。

## CURRENT 关键字

使用 SET CONNECTION 语句的带有 DORMANT 选项的 CURRENT 关键字作为标识当前连接的简写形式。CURRENT 关键字替代当前的连接名称。如果当前的连接是 con1，则下列两个语句是等同的。：

```
SET CONNECTION 'con1' DORMANT;

SET CONNECTION CURRENT DORMANT;
```

## 当事务是活动的时

不带有 DORMANT 关键字，SET CONNECTION 隐式地将当前的连接置于休眠状态。

当您发出带有 DORMANT 关键字的 SET CONNECTION 语句时，SET CONNECTION 语句显式地将当前的连接置于休眠状态。在两种情况下，如果成为休眠的连接有一未提交的事务，则该语句可失败。如果成为休眠的连接有一未提交的事务，则适用下列条件：

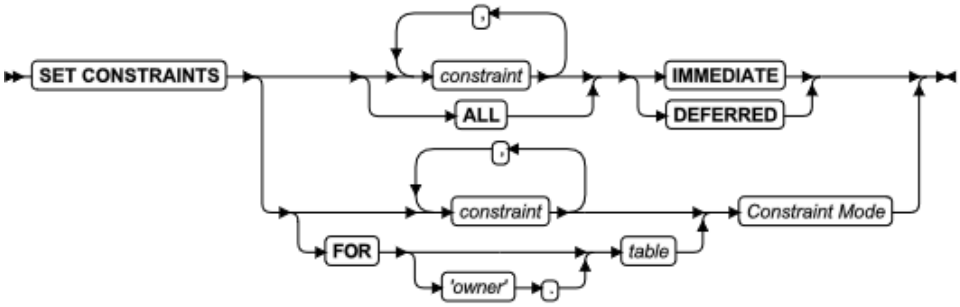
- 如果使用 CONNECT 语句的 WITH CONCURRENT TRANSACTION 子句建立了连接，则 SET CONNECTION 成功并将该连接置于休眠状态。
- 如果该连接不是通过 CONNECT 语句的 WITH CONCURRENT TRANSACTION 子句建立了的，则 SET CONNECTION 失败且不可将该连接设置为休眠状态，且当前的连接中的事务继续为活动的。该语句生成错误且应用必须决定是提交还是回滚活动的事务。

## 2.127 SET CONSTRAINTS 语句

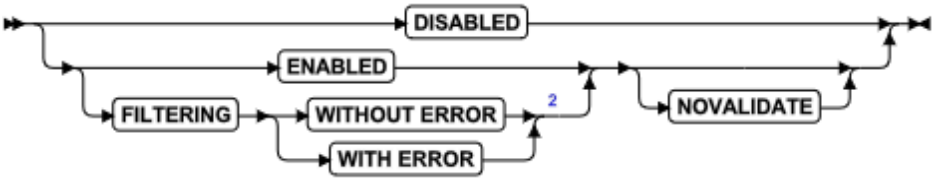
使用 SET CONSTRAINTS 语句来更改处理表上的某些或所有现有的约束的方式。

**语法**

仅 SQL 的 CREATE TABLE、CREATE TEMP TABLE 和 ALTER TABLE ADD CONSTRAINT 语句可创建新的约束。SET CONSTRAINTS 语句支持下列语法，用于修改数据库服务器强制（或不理会）单个表上的一个或多个现有的约束的方式：




约束模式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constraint* | 要重置其模式的约束 | 必须存在，且必须都在同一表上定义 | 标识符 |
| *owner* | *table* 的所有者 | 必须拥有表 | 所有者名称 |
| *table* | 对于所有约束，要重置其约束模式的表 | 必须在数据库中存在 | 标识符 |

**用法**

SET CONSTRAINTS 语句的 Constraint-mode 关键字选项包括这些：

- 是在语句级（IMMEDIATE）还是在事务级（DEFERRED）检查约束
- 是启用（ENABLED）还是禁用（DISABLED）约束
- 带有违反表的表上的约束的过滤模式应为 FILTERING WITH ERROR 还是 FILTERING WITHOUT ERROR
- 是否要启用引用约束，而不验证（NOVALIDATE）每行中的外键值是否与被引用的表中的主键值相匹配。

SET Transaction Mode 语句可以 SET CONSTRAINTS 关键字开始，在 SET Transaction Mode 语句中对此描述。

SET Database Object Mode 语句的特殊情况也可以 SET CONSTRAINTS 关键字开始，这是对 SQL 的 ANSI/ISO 标准的扩展。除了约束之外，SET Database Object Mode 语句还可启用或禁用触发器或索引，或更改唯一索引的过滤模式。要获取那个语句的完整语法和语义，请参阅 SET Database Object Mode 语句。

要获取关于使用 SET CONSTRAINTS 语句来启用或禁用通过 PRIMARY KEY 和 FOREIGN KEY 约束定义隐式地创建的系统定义的索引的信息，请参阅主题 SET INDEXES 语句。

**约束模式的保持**

您对约束的模式的任何更改都保持，直到再次修改那个约束模式的设置，或直到删除那个约束或它的表为止。

然而，引用约束的 NOVALIDATE 模式是例外，因为这些模式在指定 NOVALIDATE 模式的 SET CONSTRAINTS 语句之外（或 ALTER TABLE ADD CONSTRAINT 语句之外）不保持。

也就是说，当指定 NOVALIDATE 模式的 DDL 语句完成时，该约束模式转化为 sysobjstate 系统目录表为这三种可能的模式之中的外键约束记录的任何一种模式：

- ENABLED NOVALIDATE 成为 ENABLED
- FILTERING WITH ERROR NOVALIDATE 成为 FILTERING WITH ERROR
- FILTERING WITHOUT ERROR NOVALIDATE 成为 FILTERING WITHOUT ERROR.

在所有对该表的后续 DML 操作中,诸如 SQL 的 DELETE、INSERT、MERGE 或 UPDATE 语句，数据库服务器在通过它的 IMMEDIATE 或 DEFERRED 设置确定的时间点，强制启用的外键约束，但不理会任何先前的 NOVALIDATE 模式。

**在辅助服务器上的约束**

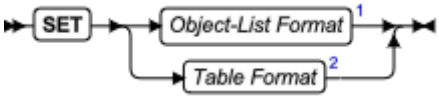
在集群环境中，在可更新的辅助服务器上，不支持 SET CONSTRAINTS ENABLED 和 SET CONSTRAINTS DISABLED 语句。（更一般地，SET Database Object Mode 语句指定的会话级索引、触发器和约束模式不会为辅助服务器的数据库中表对象上的 UPDATE 操作被重定向。）

## 2.128 SET Database Object Mode 语句

使用 SET Database Object Mode 语句来更改约束和唯一索引的过滤模式，或启用或禁用约束、索引和触发器，或在此语句正在重置它们的约束模式时绕过外键约束的引用完整性检查。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。要指定是在语句级还是在事务级检查约束，请参阅 SET Transaction Mode 语句。

**语法**




**用法**

在此语句的上下文中，***database object*** 有***索引***、***触发器***或***约束***的受限的含义，而不是在 其它语法段 中定义的 数据库对象名 段描述此术语的那种更通用的含义。

SET Database Object Mode 语句的作用域限定在当前连接的会话的本地数据库中的约束、索引或触发器。在您更改对象的模式之后，新模式对那个数据库的所有会话生效，并保持生效，直到另一 SET Database Object Mode 语句再次更改它为止，或直到从该数据库删除该对象为止。

重要：

此语句可将外键约束重置到的 NOVALIDATE 模式对上述通用语句是一例外，如本主题下列部分所说明的那样。

触发器、索引和约束的对象模式

允许重复的值的触发器和索引仅可用两种对象模式：

- 启用的（通过 ENABLED 关键字）
- DISABLED 禁用的（通过 DISABLED 关键字）

对于约束和唯一索引，您还可指定两种附加的模式：

- 不带有违反完整性错误的过滤（通过 FILTERING WITHOUT ERROR 关键字）
- 带有违反完整性错误的过滤（通过 FILTERING WITH ERROR 关键字）

对于外键约束，您还可指定三种附加的模式：

- 启用的，但不检查违反完整性错误（通过 ENABLED NOVALIDATE 关键字）
- 带有违反完整性错误的过滤，但不检查违反完整性错误（通过 FILTERING WITH ERROR NOVALIDATE 关键字）
- 不带有违反完整性错误的过滤，但不检查违反完整性错误（通过 FILTERING WITHOUT ERROR NOVALIDATE 关键字）。

在运行 SET Database Object Mode 语句时，仅最后三种约束模式保持，之后，该约束模式转换为相应的启用或过滤模式，且在后续的 DML 操作过程中强制要求引用的完整性。但对于那些被认为不违反引用的约束的大型表，这些绕过违反外键约束的模式可显著地减少迁移或导入大型数据集所需要的时间。

在任何给定的时刻，对象必须恰好处于这些模式中的一种之中。这些模式，有时称为***对象状态***，在 数据库对象模式的定义 部分描述。

**sysobjstate** 系统目录表描述数据库中的所有约束、索引和触发器对象，以及每一对象的当前模式。由于仅在指定那种模式的 SET CONSTRAINTS 语句或 ALTER TABLE ADD CONSTRAINT 语句期间保持该 NOVALIDATE 模式，**sysobjstate** 表不理会 NOVALIDATE 模式，其仅在那些 DDL 语句之内阻止违反检查。要获取关于 **sysobjstate** 表的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

在集群环境中，在可更新的辅助服务器上，不支持 SET Database Object Mode 语句。（更为通用地，对于辅助服务器的数据库中的表上的 UPDATA 操作，该语句指定的任何会话级索引、触发器或约束模式不会重定向。）

## 更改数据库对象模式所需要的权限

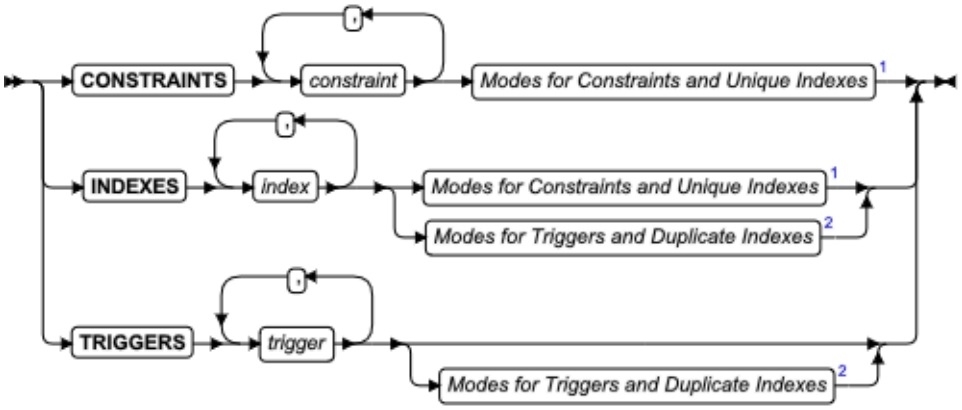
要更改约束、索引或触发器的模式，您必须有必要的访问权限。你必须至少满足这些要求之一：

- 您必须有对数据库的 DBA 权限。
- 您必须是在其上定义该数据库对象的表的所有者，且您还必须有对该数据库的 Resource 权限。
- 您必须有对在其上定义该数据库的表的 Alter 权限，且您必须还有对该数据库的 Resource 权限。

## 对象列表格式

使用对象列表格式来更改一个或多个约束、索引或触发器的模式。

对象列表格式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constraint* | 要设置其模式的约束的名称 | 必须是本地的约束，且该列表中的所有约束必须定义在同一表上 | 标识符 |
| *index* | 要设置其模式的索引的名称 | 必须是本地的索引，且该列表中的所有索引必须定义在同一表上 | 标识符 |
| *trigger* | 要设置其模式的触发器的名称 | 必须是本地的触发器，且该列表中的所有触发器必须定义在同一表 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 或视图上 | |

例如，要将 cust_subset 表上的唯一索引 unq_ssn 的模式更改为过滤的，请输入下列语句：

```
SET INDEXES unq_ssn FILTERING;
```

您还可使用对象列表格式来更改定义在同一表上的约束、索引或触发器的列表的模式。假设在 cust_subset 表上定义四个触发器：insert_trig、update_trig、delete_trig 和 execute_trig。还假设启用全部四个触发器。要禁用除了 execute_trig 之外的所有触发器，请输入此语句：

```
SET TRIGGERS insert_trig, update_trig, delete_trig DISABLED;
```

如果 my_trig 是在视图上的禁用的 INSTEAD OF 触发器，则下列语句启用那个触发器：

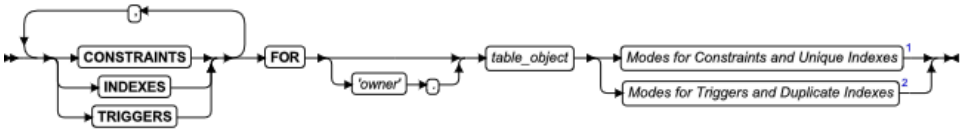
```
SET TRIGGERS my_trig ENABLED;
```

在集群环境中，在可更新的辅助服务器上不支持 SET TRIGGERS 语句。更为通用地，SET Database Object Mode 语句指定的会话级索引、触发器和约束模式，对于辅助服务器的数据库中的表对象上的 UPDATA 操作不重定向。

## 表格式

使用表格式来更改已在同一表或视图上定义了的指定的类型的所有数据库对象的模式。

表格式




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *owner* | *table* 的所有者 | 必须拥有 *table* | 所有者名称 |
| *table_object* | 在其上定义对象的表或视图 | 必须为本地的表或视图。在临时表上定义的对象不可设置为禁用的或过滤的模式。 | 标识符 |

此示例禁用 cust_subset 表上定义的所有约束：

```
SET CONSTRAINTS FOR cust_subset DISABLED;
```

在表格式中，您可以单个语句更改多个数据库对象类型的模式。例如，此示例启用在 cust_subset 表上定义的所有约束、索引和触发器：

```
SET CONSTRAINTS, INDEXES, TRIGGERS FOR cust_subset ENABLED;
```

在 GBase 8s 10.00 以及更早的版本中，您不可使用 SET Database Object Mode 语句的 SET TRIGGERS 选项来选择性地禁用表层级之内的继承的触发器。然而，在此版本中，在层级之内的表

上禁用触发器不影响继承的触发器。例如，下列语句禁用在指定的 *subtable* 上的所有触发器，但该语句不影响在表层级之内在 *subtable* 之上或之下的表对象上的触发器：

SET TRIGGERS FOR ***subtable*** DISABLED;

然而，在集群环境中，在可更新的辅助服务器上不支持 SET TRIGGERS、SET INDEXES 和 SET CONSTRAINTS 语句。对于辅助服务器的数据库中表对象上的 UPDATE 操作，SET Database Object Mode 语句指定的会话级索引、触发器和约束模式不会重定向。

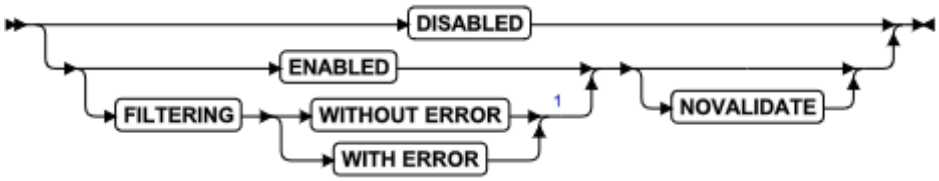
## 约束和唯一索引的模式

您可为约束或为唯一索引指定启用的或禁用的模式。对于 START VIOLATIONS TABLE 语句已将违反表与诊断表相关联的表，您还可使用 FILTERING 关键字来为处理不遵守约束或不遵守唯一索引要求的那些行指定 ERROR 模式。

当您将外键约束的模式更改为 ENABLED 或 FILTERING 时，您可可选地包括 NOVALIDATE 关键字。这会挂起对那些在 SET CONSTRAINTS 语句执行期间违反该约束的行的引用完整性检查。

这是为了更改在 SET CONSTRAINTS 或 SET INDEXES 语句中的约束或唯一索引的数据库对象模式的语法：

约束和唯一索引的模式




### 用法

如果您在创建约束的 ALTER TABLE 或 CREATE TABLE 语句中未指定模式，则缺省地启用该约束。

类似地，如果您在创建索引的 CREATE INDEX 语句中未指定模式，则缺省地启用该索引。

然而，对于在 SET Database Object Mode 语句中的数据库对象没有缺省的模式。如果您在 SET Database Object Mode 语句的 SET CONSTRAINTS 或 SET INDEXES 选项中未指定模式，则该语句失败并报错 -201，且不更改该约束模式或索引模式。

WITHOUT ERROR 和 WITH ERROR 过滤选项支持 DML 操作，在其中数据库服务器检测新的或修改了的行是否违反索引或目标表上的唯一索引。在过滤模式中，数据库服务器如何处理不符合的行还依赖于这些因素：

- 违反表和诊断表是否与在其上定义该约束或唯一索引的表相关联。
- 当前是启用还是禁用到相关联的违反表和诊断表的输入。

要获取更多信息，请参阅 START VIOLATIONS TABLE 语句 和 STOP VIOLATIONS TABLE 语句。

**更改约束模式和唯一索引模式的示例**

下列语句禁用约束 u100_1，以便它仍然注册在系统目录中，但不起作用：

```
SET CONSTRAINTS u100_1 DISABLED;
```

如果 u100_1 是启用的唯一索引，而不是约束，则下列语句有类似的作用：

```
SET INDEXES u100_1 DISABLED;
```

下列语句启用引用的约束 u100_1，而不验证每一行的外键关系：

```
SET CONSTRAINTS u100_2 ENABLED NOVALIDATE;
```

**警告：**

您可将外键约束的新模式指定为 ENABLED NOVALIDATE 或 FILTERING WITH ERROR NOVALIDATE 或 FILTERING WITHOUT ERROR NOVALIDATE。这样可提升加载操作的性能，例如，如果知道数据集会对在外键约束的作用域之内的每行都有相应的主键的话。然而，避免在后续的 DML 操作中发生数据库冲突是用户的职责。如果您不确信数据行是否符合，则

- 您应禁用外键约束，
- 将数据加载到新的数据库内，
- 然后在它的表已成功地加载了之后再启用外键约束，以便于数据库服务器可验证数据的引用完整性。

当 SET CONSTRAINTS 语句执行完成时，数据库服务器自动地删除 NOVALIDATE 属性。下列语句启用相同的外键约束并恢复该约束的自动验证：

```
SET CONSTRAINTS u100_2 ENABLE;
```

当您使用 FILTERING WITHOUT ERROR 关键字来定义过滤模式时，后续的那个约束的违反或那个索引的唯一性违反不会导致 INSERT、DELETE、MERGE 或 UPDATE 操作失败，如果有些行违反该约束或该唯一索引的话。在此过滤模式中，DML 语句成功，但数据库服务器通过将不符合的行写到违反表来强制满足该约束或该唯一索引的要求。

下列语句指导数据库服务器将违反 r104_11 约束的任何行写到违反表，假如违反表与该目标表相关联的话。

```
SET CONSTRAINTS r104_11 FILTERING WITHOUT ERROR;
```

要获取关于过滤模式的更多信息，请参阅主题 过滤模式。

下列语句启用在 orders 表上定义的所有约束：

```
SET CONSTRAINTS FOR orders DISABLED;
```

那个表上的后续的 DML 操作不理会违反 orders 表上的约束的那些行，不在它的违反表或诊断表中创建条目，如果这些表存在的话。然而，如果在 orders 表上存在任何唯一索引，则根据索引的当前模式处理违反唯一性要求的那些行，如罗列在 sysobjstate 系统目录表中的那样。

## 当在引用的表上存在索引时启用外键约束

在缺省情况下，当它们的模式更改为 ENABLED 时，数据库服务器自动地验证引用的约束。当 SET CONSTRAINTS 语句启用外键约束时，您可能节省时间，如果该引用的表在对应于外键约束的键的列上（或列的集合上）已有唯一索引或主键约束的话。

关于如何验证启用的外键约束，数据库服务器作出基于成本的决策。在许多上下文中，索引键算法可能更快，因为它通过仅扫描索引值，而不是索引表中所有的行来验证该约束。

数据库服务器可考虑使用索引键算法来验证它启用的外键约束，但当 SET CONSTRAINTS ENABLED 语句重置该约束模式时，仅当满足所有下列条件时才行：

- SET CONSTRAINTS 语句仅正在启用一个外键约束。

  如果是这种情况，则数据库服务器仅需要检查在其上正在启用外键约束的列上的个别值。同时验证两个外键约束可能会需要在同一扫描上使用两个索引，这是不支持的。

- 同一语句没有启用 CHECK 约束。

  如果 SET CONSTRAINTS 语句正在启用多个约束，则验证 CHECK 约束会要求检查每行，而不是个别的值。在那种情况下，不可为了验证外键约束而使用索引键算法。

- 外键列不包括用户定义的数据类型（UDT）或内建的 opaque 数据类型。

  要使得快速的索引键算法尽可能高效，它消除与用户定义的或内建的 opaque 数据类型相关联的所有执行例程的低效率，诸如 BOOLEAN 和 LVARCHAR 内建 opaque 类型。

- 外键约束的新模式不是 DISABLED。

  如果它是启用的，则不需要约束检查算法，因为不会发生对引用的完整性违反的检查。

- 该表不与活动的违反表相关联。

  在检查的时刻，违反表要求必须将不满足新约束的每行插入到违反表内。对每行进行违反扫描会防止数据库服务器使用跳过重复的行的更快的索引键算法。

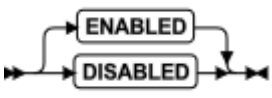
除了在一个或多个违反行的情况之外，当不满足这些要求时，SET CONSTRAINTS 语句可启用并验证外键约束，但数据库服务器不会考虑使用索引键算法来验证外键约束。扫描整个表导致的附加验证成本通常与表的大小是成比例的。对于非常大的表，这些成本可非常可观。

当您启用自引用外键约束时，其 REFERENCING 子句指定在其上定义约束的同一表，数据库服务器可考虑索引键算法来验证引用的完整性，如果满足以上罗列的所有条件的话。

## 触发器和重复的索引的模式

您可指定触发器或重复的索引的模式。

触发器和重复的索引的模式




当您创建索引或触发器时，或在后续的 SET Database Object Mode 语句中未指定它的模式，在缺省情况下启用该对象。

## 数据库对象模式的定义

您可使用数据库对象模式来控制 INSERT、DELETE 和 UPDATE 语句的作用。您对模式的选择会影响您正在操作其数据的表、在那些表上定义的数据库对象的行为，以及数据操纵语句自身的行为。

# 启用的模式

在表上的 DML 操作期间，处于启用的模式的数据库对象充当约束、索引或触发器。

如果当创建约束、索引或触发器时，您未指定数据库对象模式，则缺省地启用它们。数据定义语句 CREATE TABLE、ALTER TABLE、CREATE INDEX 和 CREATE TRIGGER 都以启用的模式创建数据库对象，除非您显式地指定一种不同的模式。

在创建时刻哪种非缺省的对象模式是可用的，依赖于对象的类型：

- 当创建触发器或非唯一索引时，可替代启用的模式的唯一关键字是 DISABLED。
- 当创建约束或唯一索引时，替代缺省的或显式的 ENABLED 关键字的包括 DISABLED、FILTERING WITH ERROR 和 FILTERING WITHOUT ERROR。（但如果您仅指定 FILTERING，则对于 FILTERING 对象，FILTERING WITHOUT ERROR 是缺省的错误模式。）
- 然而，在 ALTER TABLE ADD CONSTRAINT 语句正在创建外键约束时，任何这三种模式都可被替代地指定作为对启用的模式的附加的替代：
    - ENABLED NOVALIDATE
    - FILTERING WITH ERROR NOVALIDATE
    - FILTERING WITHOUT ERROR NOVALIDATE。

然而，当 SET Database Object Mode 语句更改现有的约束、索引或触发器的模式时，没有缺省的模式。如果您未指定对象模式，则 SET Database Object Mode 语句失败并报警 -201。如果您想要将约束、索引或触发器的模式从某些其他模式重置为启用的，则必须显式地指定 ENABLED 关键字。

当成功地启用数据库对象时，数据库服务器在系统目录的 **sysobjstate** 表中注册那个对象状态，并当它的表是后续的 INSERT、DELETE、MERGE 或 UPDATE 语句（或对于 Select 触发器，SELECT 语句）的目标时会考虑那个数据库对象。因此，启动的约束是强制的，启动的索引被更新，且当触发器事件发生时，执行表上的启用的触发器。

例如，在您将外键约束和唯一索引设置为启用的模式之后，当 INSERT、DELETE、MERGE 或 UPDATE 操作尝试违反该表的引用完整性时，数据操纵操作失败，表中的行不被更改，且数据库服务器返回错误消息。

外键约束的 ENABLED NOVALIDATE 模式

在 SET Database Object Mode 语句正在将外键约束的模式更改为 ENABLED 时，数据库服务器通过检测受约束的表中的每一行验证该约束，来核实带有相应的值的行存在于被引用的表的主键列中。此验证可需要大量的时间和资源。您可在 SET Database Object 模式操作期间替代地绕过对违反行的

搜索，通过包括 NOVALIDATE 关键字来将外键约束模式更改为 ENABLED NOVALIDATE。对于大型表，指定 ENABLED NOVALIDATE 可显著地减少启用外键约束所需要的时间。

在 SET Database Object Mode 语句的 SET CONSTRAINTS 选项成功地启用外键约束之后，

- 在 **sysobjstate** 系统目录表中，该约束模式成功地注册为启用的（E），
- 在 SET CONSTRAINTS 语句运行时，NOVALIDATE 关键字已阻止了对引用完整性违反的检查，但在系统目录内无处对该关键字编码，且它对外键约束的对象模式或行为没有进一步的影响。

直到删除或禁用那个约束之前，在对它的表执行后续的 DML 操作期间，它都是强制的，以便于维护数据库的引用完整性。

## 禁用的模式

当数据库对象是禁用的时，数据库服务器在 INSERT、DELETE、MERGE、SELECT 或 UPDATE 语句的执行期间不理会它。禁用的约束不是强制的，不更新禁用的索引，且当触发器事件发生时不执行禁用的触发器。

当请您禁用约束和唯一索引时，任何违反约束或唯一索引的限制的数据操纵语句都成功（也就是说，会更改目标行），且数据库服务器不返回错误消息。

您可使用禁用的模式来将新的约束或新的唯一索引添加到现有的表，即使表中的有些行不满足新的完整性规范。在 LOAD 操作中，禁用还可提高效率。

要获取关于添加约束的信息，请参阅 ALTER TABLE 语句中的 当现有行违法约束时添加约束。要获取关于添加唯一索引的信息，请参阅 CREATE INDEX 语句中的 当复制值存在于列中时添加唯一索引。

## 过滤模式

处于过滤模式的约束或唯一索引可在 DML 操作期间将任何不符合该约束或索引的任何行插入到相关的违反表内。此模式还支持 WITH ERROR 和 WITHOUT ERROR 选项，为了处理来自 INSERT、DELETE、MERGE 和 UPDATE 语句的引用完整性违反。

当约束或唯一索引处于 FILTERING WITH ERROR 模式时，在 INSERT、DELETE、MERGE 或 UPDATE 语句导致一个或多个行不符合唯一索引或约束之后，数据库服务器返回引用完整性违反错误消息。

在缺省情况下，不带有错误选项的 FILTERING 关键字指定 FILTERING WITHOUT ERROR 对象模式。

当约束或唯一索引处于 FILTERING WITHOUT ERROR 模式时，INSERT、DELETE、MERGE 或 UPDATE 语句成功，但数据库服务器通过将任何失败的行写到与目标表相关联的违反表，来强制该索引或唯一索引要求。将关于约束违反或唯一索引违反的诊断信息写到与目标表相关联的诊断表。

在数据操纵操作中，过滤模式对 INSERT、UPDATE 和 DELETE 语句有下列特定的作用：

- 在 INSERT 语句期间的约束违反会导致数据库服务器制成不符合的记录的拷贝，并将它写到违反表。数据库服务器不将不符合的记录写到目标表。

  如果 INSERT 语句不是单 INSERT，则以下一记录继续剩余的插入操作。
- 在 UPDATE 语句期间的约束违反或唯一索引违反会导致数据库服务器制作要被更新的现有记录的拷贝，并将它写到违反表。数据库服务器还制作新记录的拷贝，并将它写到违反表，但在目标表中不更新实际的记录。如果该 UPDATE 语句不是单 update，则以下一记录继续剩余的更新操作。
- 在 DELETE 语句期间的约束违反或唯一索引违反会导致数据库服务器制作要被删除的记录的拷贝，并将它写到违反表。数据库服务器在目标表中不删除实际的记录。如果该 DELETE 语句不是单 delete，则以下一记录继续剩余的删除操作。
- 在 MERGE 语句中，分别按照以上所述来处理组件 INSERT、DELETE 或 UPDATE 操作。

在所有这些情况下，数据库服务器将关于每一约束违反或唯一索引违反的诊断信息发送到与目标表相关联的诊断表。

要获取关于数据库服务器写到违反表和诊断表的记录的结构信息，请参阅 违反表的结构 和 诊断表的结构。

**外键约束的 FILTERING NOVALIDATE 模式**

在 SET Database Object Mode 语句正在将外键约束的模式更改为 FILTERING WITHOUT ERROR 或 FILTERING WITH ERROR 时，数据库服务器通过检测受约束的表中的每一行验证该约束，来确保带有相应的值的行在被引用的表的主键列中存在。此验证可需要大量时间和资源。在 SET Database Object 模式操作期间您可替代地绕过对违反行的搜索，通过包括 NOVALIDATE 关键字来将外键约束模式更改为 FILTERING WITHOUT ERROR NOVALIDATE 或 FILTERING WITH ERROR NOVALIDATE。对于大型表，指定 ENABLED NOVALIDATE 可显著地减少将外键约束的模式更改为过滤模式所需的时间。

在 SET Database Object Mode 语句成功地启用外键约束之后，

- 在 **sysobjstate** 系统目录表中，该约束被注册为 FILTERING WITHOUT ERROR 模式（**F**）或 FILTERING WITH ERROR 模式（**G**），
- 但 NOVALIDATE 关键字没有编码且无后续的影响。

在后续的 DML 操作期间，数据库服务器将外键约束强制作为指定的 SET CONSTRAINTS 语句，带有或不带有完整性违反错误，来维护数据库的引用的完整性。

## *启动和停止违反表和诊断表*

您必须为在其上定义数据库对象的目标表使用 START VIOLATIONS TABLE 语句来启动违反表和诊断表，或在您将定义的任何数据库对象设置为过滤模式之前，或在您将数据库对象设置为过滤之后，但要在任何用户发出 INSERT、DELETE 或 UPDATE 语句之前。

如果您想要停止数据库服务器将坏记录过滤到违反表并将每一坏记录的诊断信息发送到诊断表，则必须发出 STOP VIOLATIONS TABLE 语句。

要获取关于这些语句的进一步信息，请参阅 START VIOLATIONS TABLE 语句 和 STOP VIOLATIONS TABLE 语句。

### *过滤模式的错误选项*

当您将约束或唯一索引的模式设置为过滤时，您必须指定两个错误选项之一。在数据操纵语句执行期间，当遇到坏记录时，这些错误选项控制数据库服务器是否显示完整性违反错误消息：

- 在执行 INSERT、DELETE 或 UPDATE 语句之后，其中一个或多个目标行导致约束违反或唯一约束违反，WITH ERROR 选项指导数据库服务器返回引用的完整性违反错误消息。
- WITHOUT ERROR 选项是缺省的。在 INSERT、DELETE 或 UPDATE 语句导致约束违反或唯一索引违反之后，此选项阻止数据库服务器向用户发出引用的完整性违反错误消息。

### *过滤模式对数据库的影响*

过滤模式的净影响就是目标表的内容总是满足表上的所有约束以及表上的任何唯一索引要求。

此外，数据库服务器不丢失违反约束或唯一索引要求的任何数据值，因为将不符合的记录发送到违反表，并将关于那些记录的诊断信息发送到诊断表。

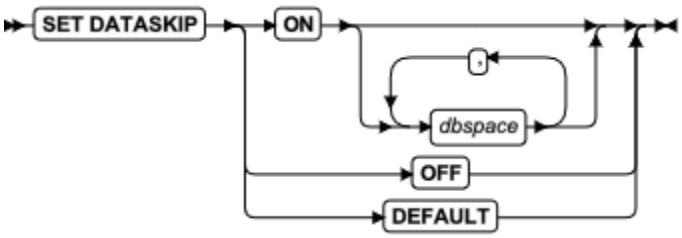
而且，当过滤模式生效时，当数据库服务器遇到坏记录时，对目标表的插入、删除和更新操作不会失败。这些操作成功地将所有好记录添加到目标表。因此，过滤模式适合于表的大规模批量更新。用户可在该情况发生之后，修理违反约束和唯一索引要求的那些记录。在批量更新之前，用户不需要修理坏记录，以避免在批量更新期间丢失坏记录。

## 2.129 SET DATASKIP 语句

在事务处理期间，使用 SET DATASKIP 语句来控制数据库服务器是否跳过不可用的 dbspace。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbspace* | 跳过的 dbspace 的名称 | 在执行时刻必须存在 | 标识符 |

用法

SET DATASKIP 允许您在运行时重置 Dataskip 特性，在处理事务的过程中，其控制数据库服务器是否跳过不可用的 dbspace（例如，由于介质失败）。

在 GBase 8s ESQL/C 中，如果跳过 dbspace，则将警告标志 **sqlca.sqlwarn.sqlwarn6** 设置为 W。另请参阅 *GBase 8s ESQL/C 程序员手册*。

在 GBase 8s 中，此语句仅适用于那些跨 dbspace 或分区分片的表。它既不适用于 blobspace 也不适用于 sbspac。

指定不包括 ***dbspace*** 的 SET DATASKIP ON，指导数据库服务器跳过在不可用的分片列表中的任何 dbspace。您可使用 **gstat -d** 或 **-D** 选项来确定 dbspace 是否关闭。

当您指定 SET DATASKIP ON ***dbspace*** 时，你正在指导数据库服务器跳过指定的 ***dbspace***，如果它是不可用的话。

如果您指定 SET DATASKIP OFF，则禁用 Dataskip 特性。如果您指定 SET DATASKIP DEFAULT，则数据库服务器使用在 ONCONFIG 文件中 DATASKIP 配置参数中指定的设置。

示例

下例跳过当前会话的 dbsp1：

```
SET DATASKIP ON dbsp1;
```

下例将 DATASKIP 的值设置为在 onconfig 中指定的值：

```
SET DATASKIP DEFAULT;
```

下例关闭 DATASKIP 以便于使用所有 dbspace。

```
SET DATASKIP OFF;
```

## 当不可跳过 dbspace 时的情况

在某些条件下，数据库服务器不可跳过 dbspace。下列列表概述这些条件：

- 引用的约束检查

  当您想要删除父行时，子行也必须可用于删除，且必须在可用的分片中存在。

  当您想要插入新的子行时，必须在可用的分片中找到父行。

- 更新

  当您执行一个将记录从一个分片移到另一分片的更新时，两个分片必须都是可用的。

- 插入

  当您试图在基于表达式的分片策略中插入记录且该 dbspace 不可用时，返回错误。

  当您试图在基于轮转法分片策略中插入记录时，且 dbspace 关闭，数据库服务器将这些行插入到任何可用的 dbspace 内。

  当没有 dbspace 可用时，返回错误。

- 索引

  当您执行影响索引的更新时，比如当您插入或删除行，或更新索引了的列，该索引必须是可用的。

  当您试图创建索引时，您想要使用的 dbspace 必须是可用的。

- 序列键

使用第一个分片内部地存储当前的序列键。除了当第一个分片不再可用并需要新的序列键值时之外，这是对您不可见的，其可发生在 INSERT 语句期间。

## 2.130 SET DEBUG FILE 语句

使用 SET DEBUG FILE 语句来标识接收 SPL 例程的运行时跟踪输出的文件。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 返回 filename 的表达式 | 必须为有效的 filename | 表达式 |
| *filename* | 包含 TRACE 语句的输出的文件的 pathname | 请参阅 使用 WITH APPEND 选项 | 引用字符串. |
| *filename_var* | 存储 *filename* 字符串的主变量 | 必须是字符类型 | 特定于语言 |

用法

此语句指定数据库服务器将来自 SPL 例程中 TRACE 语句的输出写到其中的那个文件。每次执行 TRACE 语句，都将跟踪信息添加到此输出文件。

### 使用 WITH APPEND 选项

您在 SET DEBUG FILE 语句中指定的输出文件可为新的文件或现有的文件。如果您指定现有的文件，当您发出 SET DEBUG FILE TO 语句时，删除它的当前的内容。TRACE 语句的第一次执行将跟踪输出发送到该文件的开头。

如果您包括 WITH APPEND 选项，则当您发出 SET DEBUG FILE 语句时，保留该文件的当前的内容。TRACE 语句的第一次执行将新的跟踪输出添加到该文件的末尾。

如果您在 SET DEBUG FILE TO 语句中指定新的文件，则不论您是否包括 WITH APPEND 选项都没什么不同。不论您是包括还是省略 WITH APPEND 选项，TRACE 语句的第一次执行都会将跟踪输出发送到新文件的开头。

### 关闭输出文件

要关闭 SET DEBUG FILE TO 语句打开了的文件，请发送另一带有另一 filename 的 SET DEBUG FILE TO 语句。然后，您可读或编辑第一个文件的内容。

## 重定向跟踪输出

您可使用 SPL 例程外部的 SET DEBUG FILE TO 语句来将 SPL 例程的跟踪输出定向到一文件。您还可在 SPL 例程内使用此语句来重定向它自己的输出。

## 输出文件的位置

如果您在本地数据库上执行带有简单的 filename 的 SET DEBUG FILE 语句，则输出文件位于您当前的目录中。如果您当前的数据库在远程数据库服务器上，则该输出文件位于远程数据库服务器上您的 home 目录中。如果您为调试文件提供完全的 pathname，则该文件置于远程数据库服务器上您指定的目录中。如果您在该目录中没有写权限，则会收到错误。

下列示例将 SET DEBUG FILE TO 语句的输出发送到名为 debug.out 的文件：

```
SET DEBUG FILE TO 'debug' || '.out';
```

# 2.131 SET DEFERRED_PREPARE 语句

使用 SET DEFERRED_PREPARE 语句来控制客户端处理是否推迟将 PREPARE 语句发送到数据库服务器，直到发送 OPEN 或 EXECUTE 语句为止。

仅 GBase 8s 支持此语句，这是对 SQL 的 ANSI/ISO 标准的扩展。您仅可随同 GBase 8s ESQL/C 使用此语句。

**语法**


用法

在缺省情况下，SET DEFERRED_PREPARE 语句导致应用程序推迟将 PREPARE 语句发送到数据库服务器，直到执行 OPEN 或 EXECUTE 语句。实际上，PREPARE 语句与其他语句捆绑在一起，以便在客户端和服务器之间发送消息的一个来回，而不是两个。此 Deferred-Prepare 特性可影响下列动态 SQL 语句的系列：

- 随同 FETCH 或 PUT 语句操作的 PREPARE、DECLARE、OPEN 语句块
- EXECUTE 或 EXECUTE IMMEDIATE 语句跟随的 PREPARE

您可为 SET DEFERRED_PREPARE 指定 ENABLED 或 DISABLED 选项。

如果您未指定选项，则缺省的是 ENABLED。下列示例缺省地启用 Deferred-Prepare 特性：

```
EXEC SQL set deferred_prepare;
```

ENABLED 选项在应用之内启用 Deferred-Prepare 特性。下列示例显式地指定 ENABLED 选项：

```
EXEC SQL set deferred_prepare enabled;
```

在应用发出 SET DEFERRED_PREPARE ENABLED 之后，在该应用中为后续的 PREPARE 语句启用 Deferred-Prepare 特性。然后，该应用表现下列行为：

- 序列 PREPARE、DECLARE、OPEN 以 OPEN 语句将 PREPARE 语句发送到数据库服务器。如果准备好的语句有语法错误，则数据库服务器不将错误消息返回给应用，直到应用为准备好的语句声明游标并打开该游标。
- 序列 PREPARE、EXECUTE 随同 EXECUTE 语句将 PREPARE 语句发送到数据库服务器。如果准备好的语句包含语法错误，则数据库服务器不将错误消息返回到应用，直到应用尝试执行该准备好的语句为止。

如果在包含 DESCRIBE 语句的 PREPARE、DECLARE、OPEN 语句块中启用 Deferred-Prepare，则 DESCRIBE 语句必须跟在 OPEN 语句而不是 PREPARE 语句之后。如果 DESCRIBE 跟着 PREPARE，则 DESCRIBE 语句导致错误。

使用 DISABLED 选项来在该应用之内禁用 Deferred-Prepare 特性。下列示例指定 DISABLED 选项：

```
EXEC SQL set deferred_prepare disabled;
```

如果您指定 DISABLED 选项，则当执行 PREPARE 语句时，应用将每一 PREPARE 语句发送到数据库服务器。

## SET DEFERRED_PREPARE 的示例

下列代码段展示带有 PREPARE、EXECUTE 语句块的 SET DEFERRED_PREPARE 语句。在此情况下，数据库服务器立即执行 PREPARE 和 EXECUTE 语句：

```
EXEC SQL BEGIN DECLARE SECTION;
    int a;
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate descriptor 'desc';
EXEC SQL create database test;
EXEC SQL create table x (a int);


  /* 启用 Deferred-Prepare 特性 */
EXEC SQL set deferred_prepare enabled;

  /* 准备 INSERT 语句 */
EXEC SQL prepare ins_stmt from 'insert into x values(?)';
a = 2;
EXEC SQL EXECUTE ins_stmt using :a;
if (SQLCODE)
    printf("EXECUTE : SQLCODE is %d\n", SQLCODE);
```

## 随同 OPTOFC 使用 Deferred-Prepare

您可组合地使用 Deferred-Prepare 与“Open-Fetch-Close 优化”（OPTOFC）。OPTOFC 特性推迟将 OPEN 消息发送到数据库服务器，直到发送 FETCH 消息为止。如果您同时启用 Deferred-Prepare 和 OPTOFC，则发生下列情况：

- 如果准备好的语句的文本包含语法错误，则不将错误消息返回到应用，直到执行第一个 FETCH 语句为止。

- 不可执行 DESCRIBE 语句，直到 FETCH 语句之后为止。
- 在可执行 DESCRIBE 或 GET DESCRIPTOR 语句之前，您必须发出 ALLOCATE DESCRIPTOR 语句。

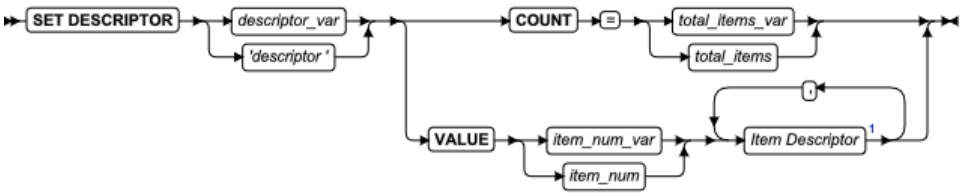
数据库服务器执行 SET DESCRIPTOR 语句的内部执行，设置系统描述符区域中的 TYPE、LENGTH、DATA 和其他字段。您在 FETCH 语句之后指定 GET DESCRIPTOR 语句来查看返回的数据。

## 2.132 SET DESCRIPTOR 语句

使用 SET DESCRIPTOR 来在系统描述符区域（SDA）中设置值。

随同 GBase 8s ESQL/C 使用此语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *descriptor* | 标识指定其值的 SDA 的字符串 | 先前必须分配了系统描述符区域（SDA） | 引用字符串 |
| *descriptor_var* | 存储 *descriptor* 的主变量 | 与 *descriptor* 相同的限制 | 特定于语言 |
| *item_num* | 指定 SDA 中项描述符的次序位置的无符号整数 | 0 < *item_num* ≤（当分配 SDA 时指定的项描述符的数目） | 精确数值 |
| *item_num_var* | 存储 *item_num* 的主变量 | 与 *item_num* 相同的限制 | 特定于语言 |
| *total_items* | 指定 SDA 描述的项的数量的无符号整数 | 与 *item_num* 相同的限制 | 精确数值 |
| *total_items_var* | 存储 *total_items* 的主变量 | 与 *total_items* 的限制相同 | 特定于语言 |

**用法**

在您以 DESCRIBE ... USING SQL DESCRIPTOR 语句已描述了 SELECT、EXECUTE FUNCTION、EXECUTE PROCEDURE、ALLOCATE DESCRIPTOR 或 INSERT 语句之后，可使用 SET DESCRIPTOR 语句。

SET DESCRIPTOR 可在这些情况下给系统描述符区域指定值：

- 设置系统描述符区域的 **COUNT** 字段来匹配您正在系统描述符区域中提供描述的项的数目。
- 为您正在系统描述符区域中提供描述的每一值设置项描述符
- 修改项描述符字段的内容

如果在给任何标识的系统描述符字段赋值期间发生错误，则将所有标识了的字段的内容设置为 0 或 NULL，这依赖于该变量的数据类型。

## 使用 COUNT 子句

使用 COUNT 子句来设置要在系统描述符区域中被使用的项的数目。如果您分配给系统描述符区域的项比您正在使用的项更多，则您需要将 COUNT 字段设置为您实际正在使用的项的数目。下列示例展示 GBase 8s ESQL/C 程序的片段：

```
EXEC SQL BEGIN DECLARE SECTION;
    int count;

EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc_100'; /*为 100 项分配*/
    count = 2;
EXEC SQL set descriptor 'desc_100' count = :count;
```

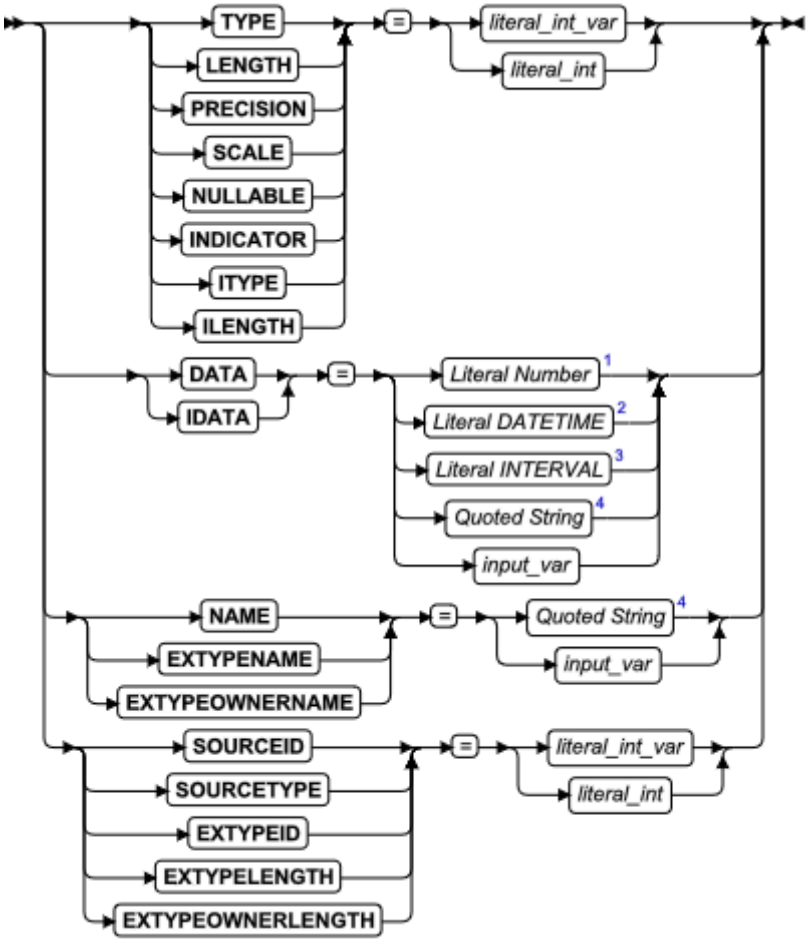
## 使用 VALUE 子句

使用 VALUE 子句来从主变量赋值到系统描述符区域的字段内。您可为您正在为其提供描述的项赋值（诸如在 WHERE 子句中的参数），或您可在为 UPDATE 或 INSERT 语句使用 DESCRIBE 语句来填充字段之后修改值。

## 项描述符

使用 SET DESCRIPTOR 语句的“项描述符”部分来为系统描述符中单个字段设置值。

项描述符




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *input_var* | 为指定的项描述符字段存储数据的主变量 | 必须适合于指定的字段 | 特定于语言 |
| *literal_int* | 赋值给指定的项描述符字段的整数值（> 0） | 依赖于 = 符号左边的关键字的限制 | 精确数值 |
| *literal_int_var* | 有 *literal_int* 值的变量 | 与 *literal_int* 的相同 | 特定于语言 |

要获取更多关于 TYPE 或 ITYPE 字段的有效代码以及它们的含义的信息，请参阅 设置 TYPE 或 ITYPE 字段。

要获取适用于其他字段类型的限制，请参阅 使用 VALUE 子句 之下单个字段类型的的标题。

## 设置 TYPE 或 ITYPE 字段

使用整数值来为每一项设置 TYPE 或 ITYPE 的值。

| SQL 数据类型 | 整数值 | X-Open 整数值 | SQL 数据类型 | 整数值 | X-Open 整数值 |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| CHAR | 0 | 1 | MONEY | 8 | - |
| SMALLINT | 1 | 4 | DATETIME | 10 | - |
| INTEGER | 2 | 5 | BYTE | 11 | - |
| FLOAT | 3 | 6 | TEXT | 12 | - |
| SMALLFLOAT | 4 | - | VARCHAR | 13 | - |
| DECIMAL | 5 | 3 | INTERVAL | 14 | - |
| SERIAL | 6 | - | NCHAR | 15 | - |
| DATE | 7 | - | NVARCHAR | 16 | - |

下表罗列以 GBase 8s 表示可用的附加的数据类型的整数值。

| SQL 数据类型 | 整数值 | SQL 数据类型 | 整数值 |
|---|---|---|---|
| INT8 | 17 | 固定长度 OPAQUE 类型 | 41 |
| SERIAL8 | 18 | LVARCHAR（仅限于客户端侧） | 43 |
| SET | 19 | BOOLEAN | 45 |
| MULTISET | 20 | BIGINT | 52 |
| LIST | 21 | BIGSERIAL | 53 |
| ROW（未命名的） | 22 | IDSSECURITYLABEL | 2061 |
| COLLECTION | 23 | ROW（命名的） | 4118 |
| 可变长 OPAQUE 类型 | 40 | | |

在系统目录中，相同的 **TYPE** 常量还可出现在 **syscolumns.coltype** 列中。请参阅 *《GBase 8s SQL 指南：参考》*。

对于更易于维护的代码，请使用为这些 SQL 数据类型预定义的常量，而不是它们的实际整数值。在 **$GBASEDBT/incl/public/sqltypes.h** 头文件中定义这些常量。然而，您不可在 SET DESCRIPTOR 语句中使用实际的常量名称。相反，将该常量赋值给整数主变量，并在 SET DESCRIPTOR 语句文件中指定该主变量。

下列示例展示您可在 GBase 8s ESQL/C 中如何设置 TYPE 字段：

```
main()
        {
        EXEC SQL BEGIN DECLARE SECTION;
        int itemno, type;
        EXEC SQL END DECLARE SECTION;
        ...
```

```
EXEC SQL allocate descriptor 'desc1' with max 5;
...
type = SQLINT; itemno = 3;
EXEC SQL set descriptor 'desc1' value :itemno type = :type;
}
```

对于 ITYPE，此信息时唯一的。当您创建不符合 X/Open 标准的动态的程序时，请使用 ITYPE。

### *不带 -xopen 选项的编译*

如果您不带 **-xopen** 选项编译，则将正常的 GBase 8s SQL 代码赋值为 TYPE。您必须小心莫将正常的模式与 X/Open 模式搞混，因为可导致错误。例如，如果数据类型未在 X/Open 模式之下定义，但在正常的模式之下定义，则执行 SET DESCRIPTOR 语句可导致错误。

### *在 X/Open 程序中设置 TYPE 字段*

在 X/Open 模式中，您必须在 **TYPE** 字段中使用该数据类型的整数代码的 X/Open 集。

如果您在 SET DESCRIPTOR 语句中使用 **ILENGTH**、**IDATA** 或 **ITYPE** 字段，则出现警告消息。该警告表示这些字段不是系统描述符区域的标准 X/Open 字段。

对于比较容易维护的代码，请使用这些 X/Open SQL 数据类型的预定义的常量，而不是它们的实际整数值。在 **$GBASEDBT/incl/public/sqlxtype.h** 头文件中定义这些常量。

### *使用 DECIMAL 或 MONEY 数据类型*

如果您为 DECIMAL 或 MONEY 数据类型设置 **TYPE** 字段，且您想要使用标度或精度而不是缺省的值，则请设置 **SCALE** 和 **PRECISION** 字段。 您不需要为 DECIMAL 或 MONEY 项设置 **LENGTH** 字段；相应地从 **SCALE** 和 **PRECISION** 字段设置 **LENGTH** 字段。

### *使用 DATETIME 或 INTERVAL 数据类型*

如果您为 DATETIME 或 INTERVAL 值设置 **TYPE** 字段，则 **DATA** 字段可为 DATETIME 或 INTERVAL 文字或字符串。如果您使用字符串，则必须把 **LENGTH** 字段编码为限定符值。

要为 DATETIME 或 INTERVAL 字符串确定编码的限定符，请在 **datetime.h** 头文件中使用 datetime 和 interval 宏。

如果您将 **DATA** 设置为 DATETIME 或 INTERVAL 的主变量，则不需要显式地将 **LENGTH** 设置为编码的限定符整数。

## 设置 DATA 或 IDATA 字段

当您设置 **DATA** 或 **IDATA** 字段时，请使用数据的适当的类型（对于 CHAR 或 VARCHAR 使用字符串，对于 INTEGER 使用整数，等等）。

如果设置任何 **DATA** 之外的内容，则不定义 **DATA** 的值。您不可为项设置 **DATA** 或 **IDATA** 字段而不为那个项设置 **TYPE**。 如果您为一项将 **TYPE** 字段设置为字符类型，则您还必须设置 **LENGTH** 字段。如果您未为字符项设置 **LENGTH** 字段，则会收到错误。

## 设置 LENGTH 或 ILENGTH 字段

如果您的 **DATA** 或 **IDATA** 字段包含字符串，则您必须指定 **LENGTH** 的值。如果您指定 LENGTH=0，则自动地将 **LENGTH** 设置为字符串的最大长度。 **DATA** 或 **IDATA** 字段可包含最多 368 字节的文字字符串，或从 CHAR 或 VARCHAR 数据类型的字符变量派生的字符串。这提供了一种自动地确定 **DATA** 或 **IDATA** 字段中字符串长度的方法。

如果 DESCRIBE 语句在 SET DESCRIPTOR 语句之前，则自动地将 **LENGTH** 设置为在您的表中指定的字符字段的最大长度。

此信息对于 **ILENGTH** 是相同的。当您创建不符合 X/Open 标准的动态的程序时，请使用 **ILENGTH**。

## 设置 INDICATOR 字段

如果您想要将 NULL 值放到系统描述符区域之内，请将 **INDICATOR** 字段设置为 -1，且不设置 **DATA** 字段。

如果您将 **INDICATOR** 字段设置 0 来表示该数据不为 NULL，则必须设置 **DATA** 字段。

## 设置 Opaque 类型字段

下列项描述符字段提供关于以 opaque 类型作为其数据类型的列的信息：

- **EXTYPEID** 字段存储 opaque 类型的扩展的标识符。此整数值必须对应于 **sysxtdtypes** 系统目录表的 **extended_id** 列中的值。
- **EXTYPENAME** 字段存储 opaque 类型的名称。此字符值必须对应于 **sysxtdtypes** 系统目录表中带有与 **extended_id** 值相匹配的行的 **name** 列中的值。
- **EXTYPELENGTH** 字段存储 opaque 类型名称的长度。此整数值是 **EXTYPENAME** 字段中字符串的长度，以字节为单位。
- **EXTYPEOWNERNAME** 字段存储 opaque 类型所有者的名称。此字符值必须对应于 **sysxtdtypes** 系统目录表中与 **extended_id** 值相匹配的行的 **owner** 列中的值。
- **EXTYPEOWNERLENGTH** 字段存储 **EXTTYPEOWNERNAME** 字段中值的长度。此整数是 **EXTYPEOWNERNAME** 字段中字符串的长度，以字节为单位。

要获取更多关于 **sysxtdtypes** 系统目录表的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 设置 Distinct 类型字段

下列项描述符字段提供关于以 distinct 类型为其数据类型的列的信息：

- **SOURCEID** 字段存储源数据类型的扩展的标识符。

  如果该 distinct 类型的源类型是 opaque 数据类型，则设置此字段。此整数值必须对应于 **sysxtdtypes** 系统目录表中其 **extended_id** 值与您正在设置的 distinct 类型相匹配的行的 **source** 列中的值。

- **SOURCETYPE** 字段存储源数据类型的数据类型常量。

此值是该 distinct 类型的源类型的内建的数据类型的数据类型常量。**SOURCETYPE** 字段的代码与 **TYPE** 字段的代码相同（设置 TYPE 或 ITYPE 字段页）。此整数值必须对应于 **sysxtdtypes** 系统目录表中其 **extended_id** 值与您正在设置的 distinct 类型相匹配的行的 **type** 列中的值。

要获取更多关于 **sysxtdtypes** 系统目录表的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

### 修改由 DESCRIBE 语句设置的值

您可使用 DESCRIBE 语句来修改系统描述符区域的内容，在它被设置之后。

您在 SELECT 或 INSERT 语句上使用 DESCRIBE 之后，必须检查以确定将 **TYPE** 字段设置为 11 还是 12 来表示 TEXT 或 BYTE 数据类型。如果 **TYPE** 包含 11 或 12，则您必须使用 SET DESCRIPTOR 语句来将 **TYPE** 重置为 116，来表示 FILE 类型。

## 2.133 SET ENCRYPTION PASSWORD 语句

使用 SET ENCRYPTION PASSWORD 语句设置加密密码和提示信息。

仅 GBase 8s 支持此语句，这是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *password* | 用于数据加密的密码（或多字词短句）。 | 必须是符型数据。并且，6 字节≤ *password* ≤128 字节。 | 引用字符串 |
| *hint* | 密码提示信息。 | 必须是字符型数据。并且，0 字节≤ *hint* ≤ 32 字节。 | 引用字符串 |

**用法**

SET ENCRYPTION PASSWORD 语句通过内置的函数声明密码，以支持数据保密性。这些函数使用 AES 或 TDES 算法进行加密和解密并使数据库能够以加密格式存储敏感数据，防止不能提供机密密码的任何人员查看、复制或修改加密数据。

该密码不以明文存储在数据库中，并且 DBA 不能访问。

***hint*** 应该为帮助您记忆 ***password*** 的词或短语，但最好不要包括 ***password***。

例如，使用 SET ENCRYPTION PASSWORD 语句设置密码和提示：

SET ENCRYPTION PASSWORD ‘abc123456’ WITH HINT ‘1234’;

有关加密和解密函数的详细信息，请参阅 加密和解密函数 一章。

## 加密的存储需求

使用 **ENCRYPT_AES** 或 **ENCRYPT_TDES** 内建的函数来加密数据。以 BASE64 格式（又称为 Radix-64）来存储字符数据类型的加密的值。对于字符数据，这需要比相应的未加密的数据大得多的存储。省略 ***hint*** 可对每一加密的值减少超过 50 字节的开销。留出充足的存储空间供加密的值使用是用户的职责。

下表罗列可加密的数据类型，以及您可用来对那些数据类型的值进行加密和解密的内建的函数：

| 原始的数据类型 | 加密的数据类型 | BASE64 格式 | 解密函数 |
|---|---|---|---|
| CHAR | CHAR | 是 | DECRYPT_CHAR |
| NCHAR | NCHAR | 是 | DECRYPT_CHAR |
| VARCHAR | VARCHAR | 是 | DECRYPT_CHAR |
| NVARCHAR | NVARCHAR | 是 | DECRYPT_CHAR |
| LVARCHAR | LVARCHAR | 是 | DECRYPT_CHAR |
| BLOB | BLOB | 否 | DECRYPT_BINARY |
| CLOB | BLOB | 否 | DECRYPT_CHAR |

您不可加密 IDSSECURITYLABEL 数据类型的列。

**DECRYPT_BINARY** 和 **DECRYPT_CHAR** 都从加密的 CHAR、NCHAR、VARCHAR、NVARCHAR 或 LVARCHAR 值返回相同的值。没有内建的加密或解密函数支持 BYTE 或 TEXT 数据类型，但您可使用 BLOB 数据类型来加密每个大型字符串。

**警告：** 如果您想要在其中存储加密的数据的数据库列的声明的大小小于加密的数据长度，则当您将加密的数据插入到该列内时会发生截断。被截断的数据不可在后续进行解密，因为在加密的字符串的头部内标明的数据长度与该列存储的不相匹配。要避免发生截断，请确保任何存储加密的字符串的列都有充足的长度。（要了解如何计算加密的字符串长度，请参阅下一段中的交叉引用。）

除了未加密的数据长度之外，对加密的数据需要的存储依赖于编码格式，依赖于您是否指定 ***hint***，以及依赖于加密函数的块大小。要了解估算加密的大小的公式，请参阅 计算加密的数据的存储需求 页上的“计算加密的数据的存储需求”。

## 指定会话口令和提示

所需的 ***password*** 规范可为带引号的字符串或其他求出其长度至少为 6 字节但不大于 128 字节的字符串的字符表达式。可选的 ***hint*** 可指定不长于 32 字节的字符串。

口令或 ***hint*** 可为单个词或几个词。***hint*** 应为帮助您记忆 ***password*** 的词或短语，但您不包括 ***password***。您可后续地执行内建的 **GETHINT** 函数（带有加密的值作为其参数）来返回 ***hint*** 的明文。

下列 ESQL/C 程序片段定义包括 SET ENCRYPTION PASSWORD 语句的例程，并执行 DML 语句：

```
process_ssn( )
{
EXEC SQL BEGIN DECLARE SECTION;
char password[128];
char myhint[33];
char myid[16], myssn[16];
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL SET ENCRYPTION PASSWORD :password WITH HINT :myhint;
...
EXEC SQL INSERT INTO tab1 VALUES (':abcd', ENCRYPT_AES("111-22-3333")) ;
EXEC SQL SELECT Pid, DECRYPT(ssn, :password) INTO :myid, :myssn;
...
EXEC SQL SELECT GETHINT(ssn) INTO :myhint, WHERE id = :myid;
}
```

## 加密的级别

您可使用带有加密和解密函数的 SET ENCRYPTION PASSWORD 来支持在数据库中的这些加密粒度。

- **列级加密：**使用相同的口令、相同的加密算法和相同的加密模式加密数据库表的给定列中的所有值。（在这种情况下，您可通过在加密的列的外部存储 ***hint***，而不是在每行中重复它，来节省磁盘空间。）
- **单元级加密：**使用不同的口令或不同的加密算法或不同的加密模式来加密同一数据库表的不同行中给定的列的值。有时有必要采用此技术来保护个人的数据。（对于单元级加密，***行-列级*** 加密与***集合-列级***加密都是同义词。）

单元级加密可导致极大的维护成本。如果您实施此级别的加密，则您的应用有责任确定哪些行包含加密的数据，并使用正确的编码来处理数据。如果将 GBase 8s 的内建的解密函数应用到未加密的数据，则它们会失败并报错 -26005。避免此错误的最简单的办法是使用列级加密而不是单元级加密。

如果您不使用加密函数，则人们可将未加密的数据输入到本应包含加密的数据的列内。要确保输入到字段中的数据总是加密了的，请使用视图和 INSTEAD OF 触发器。

## 保护口令

您以 SET ENCRYPTION PASSWORD 声明的口令和提示不会以明文存储在系统目录的任何表内，系统目标中也不维护包含加密的数据的列或表的记录。

然而，要防止其他用户访问加密的数据或口令的明文，您必须避免可能违反口令的保密性的活动：

- 不要使用解密函数创建函数的索引。（这可能会在数据库中存储明文，达不到加密的目的。）

- 在不安全的网络上，总是以加密的数据工作或使用会话加密，因为在客户端与服务器之间的 SQL 通信以明文发送要被加密的口令、提示和数据。
- 不要在将口令暴露给公众的触发器或 UDR 中存储口令。
- 不要在创建任何视图、触发器、过程或 UDR 之前设置会话口令。仅当您使用对象时才设置会话口令。否则，该口令可能在该模式中对其他用户可见，且通过其他用户执行的查询可能返回未加密的数据。下列示例展示包括加密的口令的过程：

```
-- reset session encryption password
        set encryption password null;

        -- create procedure without password
        create procedure p1 ();
        insert into tab2 select (decrypt_char (col1))
        from tab1;
        end procedure;

        -- set session encryption password
        set encryption password ("PASSWD2");

        -- insert data
        insert into tab1 values (encrypt_aes ('WXY'));

        -- call procedure
```

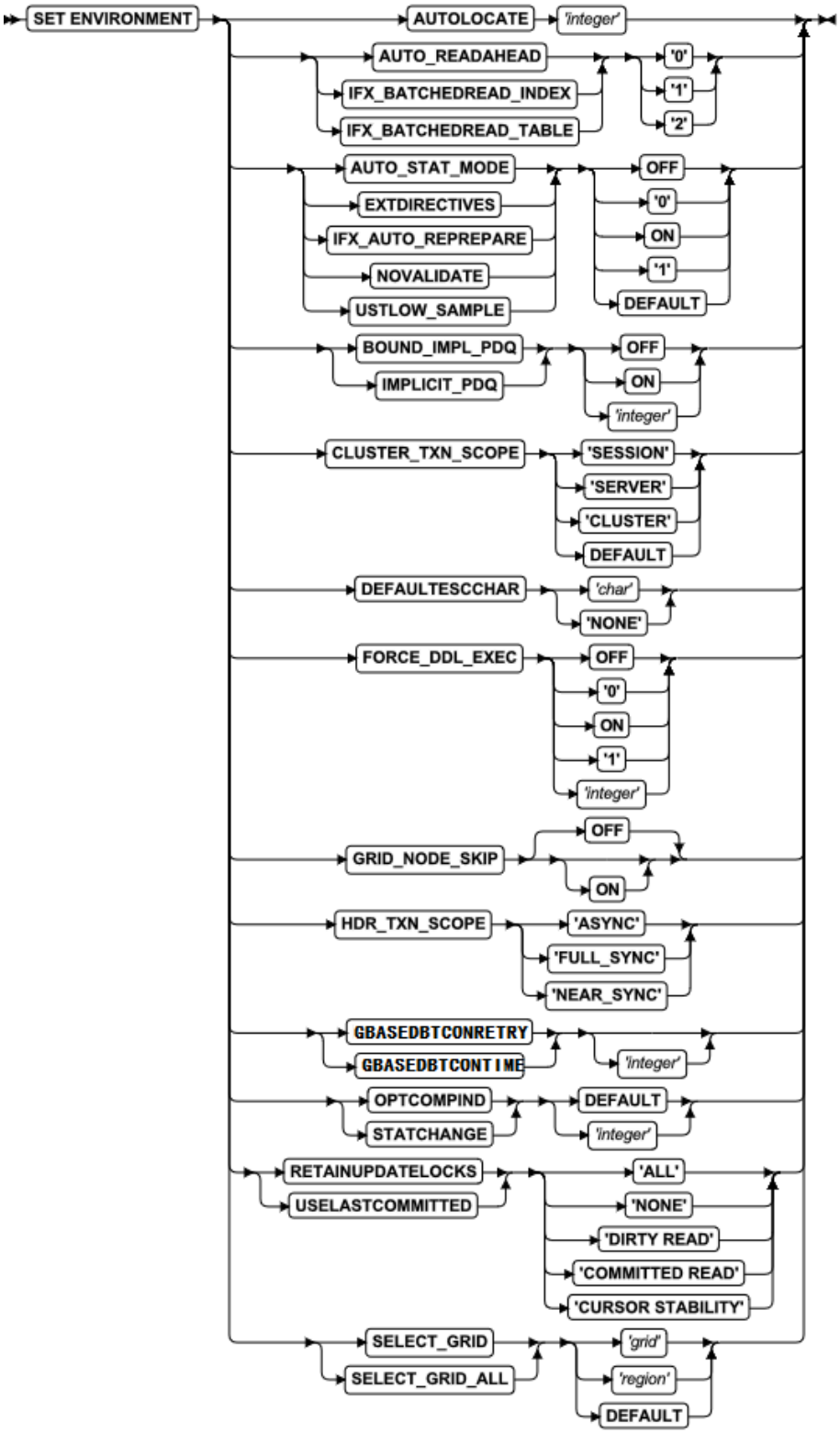
从 SET EXPLAIN 语句的输出总是显示 ***password*** 和 ***hint*** 参数为 XXXXX，而不显示实际的 ***password*** 或 ***hint*** 值。

## 2.134 SET ENVIRONMENT 语句

使用 SET ENVIRONMENT 语句来指定会话环境选项的设置，这些可影响同一例程之内提交的后续查询，或当前的用户会话的其他操作。对于有些选项，此语句覆盖那些通过配置参数或通过环境变量为数据库服务器实例设置的缺省行为。

这是对 SQL 的 ANSI/ISO 标准的扩展。 GBase 8s 支持 AUTOLOCATE、AUTO_READAHEAD、AUTO_STAT_MODE、BOUND_IMPL_PDQ、CLUSTER_TXN_SCOPE、DEFAULTESCCHAR、EXTDIRECTIVES、FORCE_DDL_EXEC、GRID_NODE_SKIP, HDR_TXN_SCOPE、IFX_AUTO_REPREPARE、IFX_BATCHEDREAD_INDEX、IFX_BATCHEDREAD_TABLE、IMPLICIT_PDQ、GBASEDBTCONRETRY、GBASEDBTCONTIME、NOVALIDATE、OPTCOMPIND、RETAINUPDATELOCKS、SELECT_GRID、SELECT_GRID_ALL、STATCHANGE、USELASTCOMMITTED 和 USTLOW_SAMPLE 会话环境选项。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *char* | 单个字符来设置作为该会话中 LIKE 或 MATCHES 运算对象中的缺省转义符 | 必须为单字节字符 | 引用字符串 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *grid* | 对于不带有显式的 GRID 子句的后续查询的缺省的现有网格 | 必须存在，且必须通过 cdr define grid 命令定义 | 引用字符串 |
| *integer* | 取值范围为 1-32 的 AUTOLOCATE 值启用自动的存储定位并扩展表的大小，且分配那个数目的初始轮转分片。0 设置禁用此存储自动。<br>取值范围为 1 ≤ *integer* ≤ 100 的 BOUND_IMPL_PDQ 或 IMPLICIT_PDQ 值通过那个百分比界定显式的 PDQPRIORITY 值。<br>取值 > 1 的 FORCE_DDL_EXEC 值设置超时限制，以秒为单位。<br>GBASEDBTCONRETRY 值设置在第一次失败之后附加的连接尝试的最大数目<br>取值 > 0 的 GBASEDBTCONTIME 值设置 CONNECT 语句尝试建立到数据库服务器的连接的秒数。0 设置是 GBASEDBTCONTIME 配置参数的缺省值。<br>OPTCOMPIND 代码 0、1 或 2 优先处理嵌套的循环连接查询优化器策略。<br>取值范围为 1 ≤ *integer* ≤ 100 的 STATCHANGE 值，作为百分率，为 UPDATE STATISTICS 操作设置数据交换阈值。 | 对于指定的会话环境选项必须为有效的 | 引用字符串 |
| *region* | 对于不带有显式的 GRID 子句的后续的查询，现有的网格之内缺省的区域 | 必须存在，且通过 cdr define region 命令定义 | 引用字符串 |

## 用法

SET ENVIRONMENT 指定可影响查询的环境选项，或通过该语句运行在其中的例程管理资源使用。在其中发出 SET ENVIRONMENT 的会话期间，有些选项可覆盖环境变量或配置参数的设置。 例如，SET ENVIRONMENT OPTCOMPIND '2' 语句指导查询优化器在会话期间使用成本作为后续连接

计划的基础，而不是偏爱嵌套的循环连接。即使此行为与 OPTCOMPIND 环境变量的当前的 0 或 1 设置冲突，也执行此指导。

对于几个会话环境选项，下列关键词有类似的作用：

- OFF 禁用指定的选项
- ON 启用该选项
- DEFAULT 将选项设置为它的缺省值

跟在选项名称之后的参数依赖于该选项的语法。选项名称和它的 ON、OFF 和 DEFAULT 关键字不要求引号定界符，且不区分大小写。所有其他的参数都必须括在单引号（'）或双引号（"）之间。如果对于会话环境选项带引号的字符串是有效的参数，则该参数区分大小写。前述的语法图是简化的，仅以单引号（'）围绕语法令牌来展示，双引号（"）也是有效的定界符。

如果您指定一个不被支持的环境选项名称，则返回错误 -19840。如果您指定一不被支持的 *integer* 或数字值作为有效的环境选项的设置，则返回特定于选项的错误（例如，错误 -19843，Invalid IFX_AUTO_REPREPARE value specified）。

## AUTOLOCATE 环境选项

使用 SET ENVIRONMENT 语句的 AUTOLOCATE 环境选项来启用或禁用数据库、索引和表的自动定位，以及在当前会话期间表的自动分片。

AUTOLOCATE 环境选项可有下列值：

- 0 = 禁用当前会话期间的自动定位和分片。
- 1 - 32 = 启用自动的定位和分片。该数目表明在当前的会话期间初始地分配轮转分片的数目。

AUTOLOCATE 环境选项覆盖在当前会话期间 AUTOLOCATE 配置参数的值。

例如，要启用当前会话的自动分片，并自动地为每一新表分配两个分片，请运行下列语句：

```
SET ENVIRONMENT AUTOLOCATE '2';
```

要禁用当前会话的自动分片，请运行下列语句：

```
SET ENVIRONMENT AUTOLOCATE '0';
```

## AUTO_READAHEAD 环境选项

使用 AUTO_READAHEAD 环境选项来更改自动的预读模式或禁用会话的自动的预读操作。

为会话指定自动的预读模式如下：

- 0 = 禁用自动的预读请求。
- 1 = 在标准模式中启用自动的预读请求。仅当查询等待 I/O 时，服务器将自动地处理预读请求。（缺省的）
- 2 = 在 aggressive 模式中启用自动的预读请求。服务器将在查询的启动时自动地处理预读请求，并在整个查询期间持续。

您可可选地指定预读计数（readahead_cnt）值。要这么做，请指定 4 至 4096 之间的一个数作为自动的预读请求要预先读取的页数。如果不设置此值，则缺省的是 128 页。在预读模式与预读计数值之间，请使用逗号作为分隔符。

您指定的值覆盖 AUTO_READAHEAD 配置参数为该会话的设置。

例如，要禁用会话的自动的预读操作，请指定：

```
SET ENVIRONMENT AUTO_READAHEAD '0';
```

通常，对大多数生产环境，缺省的设置（AUTO_READAHEAD = 1）都是恰当的，即使是高速缓存的环境。

## AUTO_STAT_MODE 环境选项

使用 AUTO_STAT_MODE 环境选项来启用或禁用在当前的模式期间 UPDATE STATISTICS 操作的自动模式。 在自动模式中，用户可定义最小的数据更改阈值作为表的属性。数据库服务器刷新该表上的统计、它的索引，仅当从上一次计算分发统计以来数据更改已超出阈值，才刷新表和索引分片上的统计。

查询优化器使用分布统计来标识 DML 操作的有效执行计划。然而，由于对大型表的计算统计是资源密集的操作，因此与更有效的系统资源的分配相比，对他们在系统目录中的当前值没有显著差异而重新计算分布，会降低数据库服务器的性能。

当启用自动的 UPDATE STATISTICS 模式时，UPDATE STATISTICS 语句仅有选择地刷新其标识为陈旧的或丢失的表、列和索引数据分发统计。当创建或修改表时，用户可指定最小的更改阈值作为表属性。此属性的值覆盖 STATCHANGE 配置参数的显式的设置或缺省的设置。SET ENVIRONMENT STATCHANGE 语句类似地覆盖当前会话的 STATCHANGE 配置参数设置。如果未显式地设置 STATCHANGE 阈值，则当启用自动的 UPDATE STATISTICS 模式时，系统缺省的阈值（自从上次计算统计以来，至少 10% 的行更改了）定义陈旧的数据分发。

当禁用自动模式时，当 UPDATE STATISTICS 语句重新计算分发统计信息时，数据库服务器不考虑任何用户定义的或缺省的陈旧统计的阈值。在非自动模式下（或当您在 UPDATE STATISTICS 语句中包括 FORCE 关键字时），数据库服务器对所有指定的表和索引删除并重新计算统计信息，不引用任何先前计算的数据分发。

SET ENVIRONMENT AUTO_STAT_MODE 语句指定的值可启用或禁用陈旧的统计的自动标识和重新计算：

- 如果指定的 AUTO_STAT_MODE 值为 'ON'，则启用自动的模式并自动地重新计算陈旧的统计信息。
- 如果指定的 AUTO_STAT_MODE 值为 'OFF'，则禁用自动的模式，且 UPDATE STATISTICS 操作重新计算陈旧的和当前的统计信息。

UPDATE STATISTICS 的自动模式要求所有的表的分片来维护在同一解析度的列的分发。这暗指以不同于在系统目录中创建当前列分发所使用的解析度的持续的 UPDATE STATISTICS 操作，会强制刷新所有分片的所有列分发。如果不指定解析度，则使用随同分发存储的解析读，而不是缺省的解析度 2.5。

自动模式仅影响永久表。AUTO_STAT_MODE 设置对临时表无影响。

**AUTO_STAT_MODE 和 STATCHANGE 配置参数**

AUTO_STAT_MODE 配置参数可为数据库服务器的所有会话的 UPDATE STATISTICS 操作的自动模式指定 1 或 0 全局的值。 然而，您可使用 SQL 的 SET ENVIRONMENT AUTO_STAT_MODE 语句来覆盖当前会话的 AUTO_STAT_MODE 配置参数设置。

STATCHANGE 配置参数可指定一负整数作为更改阈值的全局的百分率来定义陈旧的数据分发。当通过 AUTO_STAT_MODE 配置参数来启用 UPDATE STATISTICS 的自动模式时，对于任何其 STATCHANGE 表属性指定为 AUTO 或缺省为 AUTO 的表，此设置作为缺省的更改阈值生效。

要获取更多关于 AUTO_STAT_MODE 和 STATCHANGE 配置参数的信息，请参阅您的 *GBase 8s 管理员参考*。

要获取更多 STATCHANGE 表属性的信息，请参阅主题 ALTER TABLE 语句的 Statistics 选项、CREATE TABLE 语句的 Statistics 选项 和 UPDATE STATISTICS 语句的性能因素。

**SET ENVIRONMENT AUTO_STAT_MODE 的示例**

下列语句启用当前会话的自动模式：

```
SET ENVIRONMENT AUTO_STAT_MODE 'ON';
```

如果它是 'OFF'，则这会覆盖 AUTO_STAT_MODE 配置参数的设置，对于当前会话的剩余部分，或直到您重置 AUTO_STAT_MODE 会话环境变量为止。

如果您对非自动模式下分发统计上的 UPDATE STATISTICS 操作的行为感到满意，则可禁用自动模式，如此例中所示：

```
SET ENVIRONMENT AUTO_STAT_MODE 'OFF';
```

## BOUND_IMPL_PDQ 环境选项

如果 IMPLICIT_PDQ 设置为 ON 或不大于 100 的正整数值，则使用 BOUND_IMPL_PDQ 环境选项来指定分配的内存，该内存应受到当前显式的 **PDQPRIORITY** 值或范围的限制。如果 IMPLICIT_PDQ 为 OFF，则不理会 BOUND_IMPL_PDQ。

例如，您可能执行下列语句来强制数据库服务器使用显式的 PDQPRIORITY 值作为分配内存的准则，如果为当前的会话启用 IMPLICIT_PDQ 环境选项的话：

```
SET ENVIRONMENT BOUND_IMPL_PDQ ON;
```

如果您改为指定取值范围从 1 至 100 的正整数，则在当前会话期间通过那个设置标度显式的 PDQPRIORITY 值。 必须通过引号定界指定的整数，如下例中所示，指定 75% 可用的 PDQ 内存作为上限：

```
SET ENVIRONMENT BOUND_IMPL_PDQ "75";
```

在缺省情况下，不启用 BOUND_IMPL_PDQ。当为当前的会话将 BOUND_IMPL_PDQ 会话环境选项设置为 ON 时，您要求数据库服务器使用显式的 **PDQPRIORITY** 设置作为可分配给查询的内存的上限。如果既设置 IMPLICIT_PDQ 又设置 BOUND_IMPL_PDQ，则显式的 **PDQPRIORITY** 值

决定可分配给查询的内存的上限。 如果指定 **PDQPRIORITY** 作为范围，则数据库服务器在指定的范围之内授予内存。

另请参阅 *GBase 8s 性能指南* 对并行数据库查询（PDQ）的讨论。

## CLUSTER_TXN_SCOPE 环境选项

运行 SET ENVIRONMENT CLUSTER_TXN_SCOPE 语句，以便于当高可用性集群服务器上的客户端会话发出提交时，服务器阻塞该会话，直到在辅助服务器上或跨集群地在那个会话中应用事务为止。

在集群环境中，此语句可为当前的用户会话覆盖 CLUSTER_TXN_SCOPE 配置参数的设置，或可在同一会话中先前的 SET ENVIRONMENT CLUSTER_TXN_SCOPE 语句覆盖了配置参数设置之后，恢复 onconfig 文件设置的作用。

要使用此会话协作特性，请紧跟在 SET ENVIRONMENT CLUSTER_TXN_SCOPE 关键字之后，执行下列选项之一。

- 'SESSION' 以便当客户端会话发出提交时，数据库服务器阻塞该会话，直到将该会话提交的影响返回到那个会话为止。在将控制返回到会话之后，在同一数据库服务器或在该集群中其他数据库服务器上的其他会话可能察觉不到该事务提交和该事务的影响。
- 'SERVER' 以便当客户端会话发出提交时，数据库服务器阻塞该会话，直到将该事务应用到客户端会话从其发出提交的数据库服务器为止。那个数据库服务器上的其他会话觉察到该事务提交和该事务的影响。该集群中的其他数据库服务器的会话可能觉察不到该事务的提交及其影响。对于高可用性集群服务器，此行为是缺省的。
- 'CLUSTER' 以便当客户端会话发出提交时，数据库服务器阻塞会话，直到将该事务引用到高可用性集群中所有的数据库服务器为止，除了正在使用 DELAY_APPLY 或 STOP_APPLY 的 RS 辅助服务器之外，在高可用性集群中的任何数据库服务器的其他会话都觉察到该事务提交和该事物的影响。
- DEFAULT 如果设定那个参数，则以便将集群事务作用域恢复到该数据库服务器实例的 onconfig 文件中的 CLUSTER_TXN_SCOPE 配置参数设置。

例如，要为集群启用事务协作，请运行下列语句：

```
SET ENVIRONMENT CLUSTER_TXN_SCOPE 'CLUSTER';
```

## DEFAULTESCCHAR 环境选项

在当前的会话期间，您可使用 SET ENVIRONMENT 语句的 DEFAULTESCCHAR 会话环境选项来覆盖 LIKE 或 MATCHES 表达式的字符串运算对象内的当前的缺省转义字符。

***转义字符***指导 SQL 解析器将那些可为通配符的字符（例如，LIKE 运算符的运算对象的 % 或 _，或 MATCHES 运算符的运算对象的 * 或 ^）翻译为文字字符。在 LIKE 或 MATCHES 表达式中，转义字符必须紧接在要忽略其特殊含意的字符之前。

您为 DEFAULTESCCHAR 选项指定的设置可覆盖 ONCONFIG 文件中的 DEFAULTESCCHAR 配置参数的设置，来定义当前会话中 LIKE 或 MATCHES 表达式的字符串运算对象内的缺省的转义字符。其他用户会话不受 SET ENVIRONMENT DEFAULTESCCHAR 语句的影响。

要覆盖系统缺省的转移字符（\），或要覆盖通过 DEFAULTESCCHAR 配置参数设置的任何缺省的转义字符，或要覆盖在当前会话中您先前以 SET ENVIRONMENT DEFAULTESCCHAR 设置的任何缺省的转义字符，您可指定 'NONE' 作为缺省的转义字符。当 'NONE' 设置生效时，在 LIKE 或 MATCHES 表达式的 ESCAPE 子句中必须定义您在 LIKE 或 MATCHES 表达式中用于将通配符号处理为文字字符的任何转义字符。

DEFAULTESCCHAR 会话环境设置仅影响那些没有 ESCAPE 子句的 LIKE 或 MATCHES 表达式。要获取更多关于在 LIKE 或 MATCHES 条件中的转义字符的信息，请参阅 ESCAPE 与 LIKE 一起使用 和 ESCAPE 与 MATCHES 一起使用。

## EXTDIRECTIVES 会话环境选项

在当前的会话期间，您可使用 SET ENVIRONMENT 语句的 EXTDIRECTIVES 环境选项来启用或禁用外部的优化器伪指令。

EXTDIRECTIVES 会话环境选项有此语法：

EXTDIRECTIVES 环境选项

用法

您为 EXTDIRECTIVES 会话环境选项指定的设置可覆盖在 ONCONFIG 文件中的 **IFX_EXTDIRECTIVES** 环境变量和 EXT_DIRECTIVES 配置参数的设置，用于启用或禁用当前会话中的外部优化器伪指令。不影响其他用户会话。

- 在当前的会话期间，要禁用外部的优化器伪指令，请指定 '0'、off 或 OFF 作为 EXTDIRECTIVES 的设置。在当前的会话期间，此设置覆盖在客户端侧 IFX_EXTDIRECTIVES 环境变量中或 EXT_DIRECTIVES 配置参数中设置的任何值。
- 在当前的会话期间，要启用外部的优化器伪指令，请指定 '1'、on 或 ON 作为 EXTDIRECTIVES 的设置。在当前的会话期间，此设置覆盖在客户端侧 **IFX_EXTDIRECTIVES** 环境变量中或在 EXT_DIRECTIVES 配置参数中设置的任何值。
- 在当前的会话期间，要启用在 EXT_DIRECTIVES 配置参数中和客户端侧 **IFX_EXTDIRECTIVES** 环境变量中指定的缺省值，请指定 DEFAULT 作为 SET ENVIRONMENT EXTDIRECTIVES 的值。

DEFAULT 设置

如果在同一会话中，您（或 **sysdbopen** 例程）稍早使用了 SET ENVIRONMENT EXTDIRECTIVES 语句来启用或禁用外部的优化器伪指令，则 SET ENVIRONMENT EXTDIRECTIVES DEFAULT 语句将数据库服务器的外部优化器伪指令行为恢复到它的原始状态。

在会话的 DEFAULT 设置生效之后，外部的伪指令是否影响查询执行优化器依赖于 EXT_DIRECTIVES 配置参数和客户端侧 IFX_EXTDIRECTIVES 环境变量的设置。

- 当 SET ENVIRONMENT 的 EXTDIRECTIVES 选项设置为 DEFAULT，且同时启用 EXT_DIRECTIVES 与 IFX_EXTDIRECTIVES 时，在当前的会话期间启用外部的优化器伪指令。
- 当 EXTDIRECTIVES 设置为 DEFAULT，且同时禁用 EXT_DIRECTIVES 和 **IFX_EXTDIRECTIVES** 时，在当前的会话期间禁用外部的优化器伪指令。

  如果 EXTDIRECTIVES 设置为 DEFAULT，但将 EXT_DIRECTIVES 与 IFX_EXTDIRECTIVES 设置为冲突的值，比如启用一个，但禁用另一个，则数据库服务器的行为依赖于 EXT_DIRECTIVE 配置参数的设置：

  - 如果 EXTDIRECTIVES 设置为 0（或如果它没有设置），则在缺省情况下，即使启用 **IFX_EXTDIRECTIVES**，也禁用外部的伪指令。
  - 如果 EXTDIRECTIVES 设置为 1，则可启用外部的指令，如果设置启用 **IFX_EXTDIRECTIVES** 的话。（但如果 **IFX_EXTDIRECTIVES** 设置为禁用的，则禁用外部的伪指令。）
  - 如果 EXTDIRECTIVES 设置为 2，则即使 **IFX_EXTDIRECTIVES** 设置为禁用的，也启用外部的伪指令。

如果 SET ENVIRONMENT 的 EXTDIRECTIVES 选项设置为 DEFAULT，但 EXT_DIRECTIVES 配置参数与客户端侧 **IFX_EXTDIRECTIVES** 环境变量设置为冲突的值，比如启用一个，而禁用另一个，则数据库服务器的行为依赖于 EXT_DIRECTIVE 设置：

**启用或禁用外部的伪指令的示例**

下列语句启用在当前会话中执行的后续查询中的外部伪指令：

```
SET ENVIRONMENT EXTDIRECTIVES "1";
```

下列语句有相同的作用：

```
SET ENVIRONMENT EXTDIRECTIVES ON;
```

在两种情况下，数据库服务器都不理会 EXT_DIRECTIVES 和 IFX_EXTDIRECTIVES 的设置，但那些设置会影响其他并发会话中的查询。

下列两个语句都禁用在当前的会话中后续查询的外部伪指令：

```
SET ENVIRONMENT EXTDIRECTIVES OFF;
SET ENVIRONMENT EXTDIRECTIVES "0";
```

下列语句允许通过 EXT_DIRECTIVES 和 IFX_EXTDIRECTIVES 的设置来决定对当前会话的后续查询中外部伪指令的处理，如本主题中稍早描述的那样：

```
SET ENVIRONMENT EXTDIRECTIVES DEFAULT;
```

要获取关于如何定义外部的优化器伪指令以及如何将它们保存在系统目录的 **sysdirectives** 表中的信息，请参阅 SAVE EXTERNAL DIRECTIVES 语句 的描述。要获取更多关于 EXT_DIRECTIVES 配置参数以及它的设置的作用，请参阅“GBase 8s 管理员参考”。要获取更多关于 **IFX_EXTDIRECTIVES** 环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》*，其中还描述

EXT_DIRECTIVES 配置参数与 **IFX_EXTDIRECTIVES** 环境变量二者的设置如何能确定对于查询优化器是启用还是禁用对外部伪指令的访问。

## FORCE_DDL_EXEC 环境选项

使用 SET ENVIRONMENT 语句的 FORCE_DDL_EXEC 环境选项来防止已经在表上已打开或已锁定的其他事务参与 ALTER FRAGMENT ON TABLE 操作。

当启用 FORCE_DDL_EXEC 环境选项时，服务器还通过执行 ALTER FRAGMENT ON TABLE 操作的会话来关闭回滚期间的持有游标。

FORCE_DDL_EXEC 选项可有任何下列值：

- 'ON'、'on' 或 '1' 使得当发出 ALTER FRAGMENT ON TABLE 语句时服务器能够封杀那些打开的或在该表上有锁的事务，直到服务器获得锁和对该表的排他访问为止。
- 'OFF'、'off' 或 '0' 防止当发出 ALTER FRAGMENT ON TABLE 语句时服务器封杀那些打开的或在该表上有锁的事务。（缺省值是 off。）
- 以秒为单位表示时间量的正整数。该数值使得服务器能够封杀事务，直到服务器得到对该表的排他访问和排他锁为止，或直到达到指定的时限为止。如果服务器不可通过指定的时间量来封杀事务，则服务器停止封杀事务的尝试。

例如，当发出 ALTER FRAGMENT ON TABLE 语句时，要启用 FORCE_DDL_EXEC 环境选项来操作 100 秒，请指定：

```
SET ENVIRONMENT FORCE_DDL_EXEC '100';
```

**重要：**当您使用 FORCE_DDL_EXEC 环境选项时，为了获得排他访问和锁，还使用 SET LOCK MODE TO WAIT 语句来为服务器封杀任何事务指定一个时间的期间。如果您运行 SET LOCK MODE TO WAIT 而未指定时间量，则 FORCE_DDL_EXEC 将对更改分片操作不起作用。要获取更多信息，请参阅 SET LOCK MODE 语句。

当您启用 FORCE_DDL_EXEC 环境选项时，服务器支持多个会话执行 ALTER FRAGMENT ON TABLE 操作。当启用 FORCE_DDL_EXEC 选项时，如果两个会话在一个共有的表上执行 ALTER FRAGMENT ON TABLE，则第二个会话会出错。如果在表上正在发生另一 ALTER 操作，则带有启用的 FORCE_DDL_EXEC 环境选项的 ALTER FRAGMENT ON TABLE 操作会出错。

启用 FORCE_DDL_EXEC 选项的前提是：

- 您必须是用户 **gbasedbt** 或有对数据库的 DBA 权限。
- 该数据库必须是日志记录的数据库。

在您完成带有启用的 FORCE_DDL_EXEC 环境选项的 ALTER FRAGMENT ON TABLE 操作之后，您可关闭 FORCE_DDL_EXEC 环境选项。

**gshowaudit** 实用程序显示改变分片事件代码（ALFR），标识当启用了 FORCE_DDL_EXEC 环境选项时的改变分片事件。

## GRID_NODE_SKIP 环境选项

当查询指定的网格或区域内的网格服务器不可用时，使用 SET ENVIRONMENT 语句的 GRID_NODE_SKIP 选项来定义数据库服务器的行为。

GRID_NODE_SKIP 会话环境选项可设置为三个值中的任一个：

DEFAULT

此设置指定缺省的行为。当网格查询尝试从那个节点的数据库检索符合条件的行时，如果指定的网格或区域内的网格服务器不可用，则返回错误到发出该网格查询的服务器，且该查询失败。

OFF

如果未设置 GRID_NODE_SKIP，则这是缺省的设置。它与显式的 DEFAULT 设置有同样的作用，如上所述。

ON

当网格查询尝试从那个节点检索符合条件的行时，如果指定的网格或区域内的网格服务器不可用，则返回错误到发出该网格查询的服务器。那个服务器通过尝试连接该网格或区域中的下一服务器来继续处理该查询，除非其结果被跳过了的服务器是参与的网格服务器之中的最后一个。在那种情况下，组合来自可用的节点的结果，且通过发出该网格查询的服务器来计算来自可用的网格服务器的结果的逻辑 UNION 或 UNION。

对于当前的会话，当 SET ENVIRONMENT GRID_NODE_SKIP ON 语句生效时，当多个网格节点不可用时，网格查询可返回结果。通过执行 ifx_gridquery_skipped_nodes() 函数可返回跳过的网格服务器的标识。可调用另一函数 ifx_gridquery_skipped_node_count() 来检测跳过了多少个网格或区域的节点。要获取更多关于这些函数的信息，请参阅 *GBase 8s Enterprise Replication 指南*。

由于网格查询是动态的 UNION 或 UNION ALL 组合的查询，因此在语句准备的开始时作出跳过网格服务器的决定，而不是在执行语句时决定。这是因为如果不可从远程数据字典获取信息，则不可准备 SELECT 语句。因此，在语句执行之前，发出网格查询的数据库服务器作出跳过还是不跳过不可用的网格服务器的决定，如果 GRID_NODE_SKIP 会话环境选项设置为 ON 的话。

## HDR_TXN_SCOPE 环境选项

运行 SET ENVIRONMENT HDR_TXN_SCOPE 语句来控制何时将事务提交返回到集群环境中的客户端应用。

在集群环境中，此语句可执行下列活动：

- 为当前的用户会话覆盖 HDR_TXN_SCOPE 配置参数的当前设置。
- 在同一会话中先前的 SET ENVIRONMENT HDR_TXN_SCOPE 语句覆盖了配置参数设置之后，恢复 onconfig 文件设置的作用。

要使用此事务同步特性，请设置 DRINTERVAL 配置参数为 0，然后运行带有下列选项之一的 SET ENVIRONMENT HDR_TXN_SCOPE 语句：

- 'ASYNC' 代表异步模式，在事务可完成之前，它们需要在 HDR 辅助服务器上被接收或完成的确认。当复制对使用异步模式时，系统性能最佳，但如果有一个服务器失败，则可丢失数据。
- 'FULL_SYNC' 代表完全同步模式，在事务完成之前，它们需要在 HDR 辅助服务器上的完成的确认。当复制对使用完全同步模式时，数据完整性最高，但如果客户端应用使用未缓冲的日志记录且有许多小型事务，则系统性能可受到负面影响。
- 'NEAR_SYNC' 代表部分同步模式，在事务可完成之前，它们需要在 HDR 辅助服务器上被接收的确认。部分同步模式比完全同步模式有较好的性能，比异步模式有较好的数据完整性。当 DRINTERVAL 设置为 -1 时，开启 SYNC 模式，如果使用未缓冲的日志记录，则与部分同步模式相同。

**示例**

要保持对数据丢失的防备，但避免通过执行带有未缓冲的日志记录的许多小型事务的客户端应用导致的性能问题，您可通过运行下列语句来启用部分同步模式：

```
SET ENVIRONMENT HDR_TXN_SCOPE 'NEAR_SYNC';
```

## IFX_AUTO_REPREPARE 环境选项

使用 IFX_AUTO_REPREPARE 环境选项来减少在动态的 SQL 应用访问的数据库中 SQL 错误 -710 的发生率。

当游标尝试执行准备好的对象时，或当 SPL 例程执行查询时，在 DLL 操作已更改了准备好的对象或 SPL 例程引用的表的模式之后，可能发出错误 -710。

在启用 IFX_AUTO_REPREPARE 选项时，在对数据库表的模式的有些更改之后，您可避免 -710 错误，诸如添加启用的索引。此特性可减少显式地发出 PREPARE 语句来重新编译准备好的对象的需要，或显式地发出 UPDATE STATISTICS 语句来重新优化 SPL 例程的需要。在不需要重新发出 DESCRIBE 语句的表模式更改期间，如果启用 IFX_AUTO_REPREPARE，则数据库服务器自动地标识并重新编译准备好的语句和引用修改的表的 SPL 例程。

SET ENVIRONMENT IFX_AUTO_REPREPARE 语句指定的值可启用或禁用此自动重新编译特性：

- 如果指定的 IFX_AUTO_REPREPARE 值为 '1' 或 'ON' 或 'on'，则启用自动重新编译。
- 如果指定的 IFX_AUTO_REPREPARE 值为 '0' 或 'OFF' 或 'off'，则禁用自动重新编译。

在对准备好的对象或 SPL 例程引用的表的 DDL 操作之后，下列语句启用自动重新编译：

```
SET ENVIRONMENT IFX_AUTO_REPREPARE '1';
```

这覆盖 AUTO_REPREPARE 配置参数的设置，如果它是 0 或 'None' 的话，对于当前会话的剩余部分，或直到您重置 IFX_AUTO_REPREPARE 为止。

数据库服务器可能不检测有些使得准备好的对象或 SPL 例程无效的表模式的更改，即使当启用 IFX_AUTO_REPREPARE 时。例如，当您尝试在获得共享锁之后读同一表时，通过一个会话对表模式的更改导致并发的会话收到错误 -710。

启用 IFX_AUTO_REPREPARE 可能不影响引用表的准备好的语句和 SPL 例程，其中的 DDL 操作更改表中列的数目，或更改列的数据类型。要避免在这些模式更改之后发生错误 -710，对于引用其模式已被修改的表的任何例程，您通常必须重新发出 DESCRIBE 语句、PREPARE 语句以及（对于与例程相关联的游标）UPDATE STATISTICS 语句。

如果您对您的应用当前处理由于模式更改导致的错误的方式满意，则可禁用自动重新编译，如此例中所示：

```
SET ENVIRONMENT IFX_AUTO_REPREPARE 'OFF';
```

如果启用 IFX_AUTO_REPREPARE 会话环境变量导致运行时错误，则将那个错误传回应用。

要获取更多关于 AUTO_REPREPARE 配置参数的信息，请参阅您的 *GBase 8s 管理员参考手册*。要了解对游标和对查询的 IFX_AUTO_REPREPARE 和 AUTO_REPREPARE 设置的作用的讨论，请参阅您的 *GBase 8s 性能指南* 。

## IFX_BATCHEDREAD_INDEX 环境选项

在当前会话期间，使用 SQL 的 SET ENVIRONMENT 语句的 IFX_BATCHEDREAD_INDEX 环境选项来启用或禁用对来自索引缓冲区的键集合的自动存取。

指定：

- '1' 来启用优化器自动地存取来自索引缓冲区的键集合
- '0' 来禁用对来自索引缓冲区的键的自动存取

例如，要启用优化器来自动地存取来自会话的索引缓冲区的键集合，请指定：

```
SET ENVIRONMENT IFX_BATCHEDREAD_INDEX '1';
```

## IFX_BATCHEDREAD_TABLE 环境选项

在当前的会话期间，使用 SQL 的 SET ENVIRONMENT 语句的 IFX_BATCHEDREAD_TABLE 环境选项来启用或禁用对压缩的表、带有大于一页的行的表和带有 VARCHAR、LVARCHAR 和 NVARCHAR 数据的表的轻量扫描。

指定：

- '1' 来为会话启用对压缩的表、带有大于一页的行的表和带有 VARCHAR、LVARCHAR 和 NVARCHAR 数据的表的轻量扫描
- '0' 来为会话禁用这些轻量扫描

例如，要启用对带有 VARCHAR 数据的大型表的轻量扫描，请指定：

```
SET ENVIRONMENT IFX_BATCHEDREAD_TABLE '1';
```

## IMPLICIT_PDQ 环境选项

使用 IMPLICIT_PDQ 会话环境选项来允许数据库服务器决定分配给查询的内存数量。除非也设置 BOUND_IMPL_PDQ，当 IMPLICIT_PDQ 设置为 ON 或 100 时，数据库服务器不理会 **PDQPRIORITY** 环境变量的当前显式设置。

然而，当 PDQPRIORITY 设置为 100 时，数据库服务器不会分配多于可用的内存。数据库服务器可分配的内存的最大数量受您的系统可用的物理内存的限制，且受这些参数的设置的限制：

- PDQPRIORITY 环境变量
- 最近的 SQL 的 SET PDQPRIORITY 语句
- MAX_PDQPRIORITY 配置参数
- DS_TOTAL_MEMORY 配置参数
- BOUND_IMPL_PDQ 会话环境变量

当正在运行并行查询时，DS_MAX_QUERIES 配置参数设置还可限制新的查询可用的 PDQ 内存的数量。

在缺省情况下，IMPLICIT_PDQ 是禁用的。当 IMPLICIT_PDQ 设置为 OFF 时，不论是显式地还是在缺省情况下，当为查询分配资源时，数据库服务器不覆盖当前的 **PDQPRIORITY** 设置。

仅在支持 **PDQPRIORITY** 的系统上，IMPLICIT_PDQ 会话环境选项是可用的。

如果您设置 ***value*** 介于 1 与 100 之间，则数据库服务器根据指定的值调节它的估计值。如果您设置低值，则减少分配给查询的内存数量，这可能增加查询运算符溢出的风险。

要请求数据库服务器确定查询的内存分配并根据它们的需要在查询运算符之中分配内存，请输入下列语句：

SET ENVIRONMENT IMPLICIT_PDQ ON;

要要求数据库服务器来使用显式的 **PDQPRIORITY** 设置作为上限和它授权给查询的可选的较低限，请设置 BOUND_IMPL_PDQ 会话环境选项。

星型连接查询执行计划需要设置 PDQ 优先级。将 IMPLICIT_PDQ 会话环境选项设置为启用隐式的 PDQ 提供一种替代。如果对于该会话将 IMPLICIT_PDQ 设置为 ON，则星型连接执行计划会考虑不带有显式的设置 PDQPRIORITY。可通过 **sysdbopen** 过程来发出 SET ENVIRONMENT IMPLICIT_PDQ ON 语句，以便当用户打开数据库时自动地启用隐式的 PDQ。在此情况下，查询优化器自动地考虑不带有由用户设置的显式的 PDQPRIORITY 的星型连接。

查询的 IMPLICIT_PDQ 功能在该查询中的所有表要求至少 LOW 级别统计。如果在该查询中分发统计遗失一个或多个表，则 IMPLICIT_PDQ 设置不起作用。此限制还适用于星型连接查询，在遗失统计的情况下，不支持它。

要获取关于创建 **sysdbopen** 过程和关于指定其会话会受影响的用户的信息，请参阅主题 使用 SYSDBOPEN 和 SYSDBCLOSE 过程。要获取关于 **PDQPRIORITY** 环境变量的信息，请参阅 *《GBase 8s SQL 指南：参考》*。要获取关于 DS_TOTAL_MEMORY 和 MAX_PDQPRIORITY 配置参数的信息，请参阅 *GBase 8s 管理员参考手册*。

## GBASEDBTCONRETRY 环境选项

使用 SET ENVIRONMENT 语句的 GBASEDBTCONRETRY 环境选项来指定，在初次连接尝试失败之后，在当前的会话中可对每一数据库服务器进行的连接尝试的最大次数。在 GBASEDBTCONTIME 环境选项指定的时限之内进行这些尝试。

GBASEDBTCONRETRY 环境变量覆盖客户端的 GBASEDBTCONRETRY 环境变量以及在 onconfig 文件中的 GBASEDBTCONRETRY 配置参数设置的值。对于服务器到服务器连接，使用此选项来指定在会话期间可对每一数据库服务器进行的连接尝试的数目。

例如，如果初次的连接尝试失败，下列语句指定在数据库服务器发出错误之前，最多可进行三次附加的连接尝试。

SET ENVIRONMENT GBASEDBTCONRETRY '3';

GBASEDBTCONRETRY 环境选项的缺省值为 1，指定在初次的连接尝试失败之后进行一次尝试。

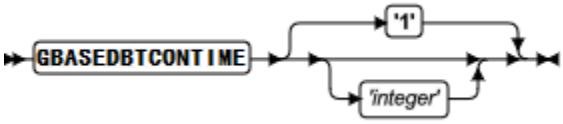
GBASEDBTCONTIME 设置优先于 GBASEDBTCONRETRY 设置。在超过 GBASEDBTCONTIME 值之后，但在达到 GBASEDBTCONRETRY 值之前，连接尝试可终止。

## GBASEDBTCONTIME 会话环境选项

在当前会话中每一 CONNECT 语句尝试建立到数据库服务器的连接时，使用 SET ENVIRONMENT 语句的 GBASEDBTCONTIME 会话环境选项来指定秒数的限定。

GBASEDBTCONTIME 会话环境选项有此语法：

GBASEDBTCONTIME 环境选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *integer* | 在 CONNECT 语句尝试建立到数据库服务器的连接时，无符号整数值 >0 设置最大秒数 | 必须定界在单引号（’）或双引号（″）之间。’0’ 设置是 GBASEDBTCONTIME 配置参数的缺省设置。 | 精确数值 和 引用字符串 |

### 用法

GBASEDBTCONTIME 会话环境设置可覆盖客户端 GBASEDBTCONTIME 环境变量的设置。通过设置 GBASEDBTCONTIME 和 GBASEDBTCONRETRY 会话环境变量来配置在当前的会话中您的服务器到服务器连接能力，您可最小化连接错误。要设置 GBASEDBTCONTIME 会话环境选项的优化值，请考虑节点之间的总距离、硬件速度、流量以及网络的并发水平。

为了设置 CONNECT 语句可用于尝试连接到数据库服务器实例的时间量的上限，这是按升序（从最低至最高）排列的该方法的优先权：

- GBASEDBTCONTIME 配置参数
- GBASEDBTCONTIME 客户端环境变量
- SQL 的 SET ENVIRONMENT GBASEDBTCONTIME 语句。

如果同一 CONNECT 语句的先前尝试未能建立连接，则 GBASEDBTCONTIME 会话环境选项的值除以 GBASEDBTCONRETRY 会话环境选项的值决定连续的连接尝试之间的最大秒数。

例如，下列语句设置 GBASEDBTCONTIME 为 60 秒，并设置 GBASEDBTCONRETRY 为重试一次：

```
SET ENVIRONMENT GBASEDBTCONTIME '60';
        SET ENVIRONMENT GBASEDBTCONRETRY '1';
```

在此示例中，CONNECT 语句尝试建立连接 60 秒。在 0 秒初次尝试连接到数据库服务器。如果 GBASEDBTCONTIME 会话环境选项设置为缺省值 ’0’，则在返回错误之前，在 60 秒进行附加的连接尝试，如果必要的话。

类似地，您可为多次重新尝试配置这些会话环境选项。以相同的 GBASEDBTCONTIME 设置，下列语句指定另外重新尝试两次：

```
SET ENVIRONMENT GBASEDBTCONRETRY '3';
```

如果 GBASEDBTCONRETRY 设置为 '3'，则在返回错误之前，最多进行三次附加的对数据库服务器的连接尝试（在 20、40 和 60 秒，如果必要的话）。

此 20 秒的间隔是 GBASEDBTCONTIME 值除以 GBASEDBTCONRETRY 值的结果。

如果您设置 GBASEDBTCONTIME 会话环境选项为 ’0’，

```
SET ENVIRONMENT GBASEDBTCONTIME '0';
```

数据库服务器自动地使用 GBASEDBTCONTIME 配置参数的设置。如果未设置 GBASEDBTCONTIME 参数，则在该会话中后续的 CONNECT 语句期间使用它的缺省值 60 秒。

如果 CONNECT 语句必须搜索 DBPATH，则 GBASEDBTCONRETRY 会话环境选项指定可对 DBPATH 中每一数据库服务器条目进行的附加连接尝试的次数。

- 即使超过 GBASEDBTCONTIME 值，也至少访问 DBPATH 设置中所有恰当的服务器一次。因此，CONNECT 语句可能花费比 GBASEDBTCONTIME 时限更长的时间来返回错误，指明连接失败，或找不到数据库。
- GBASEDBTCONTIME 值在 DBPATH 中指定的数据库服务器条目的数目之中分配。因此，如果 DBPATH 包含大量服务器，则相应地增加 GBASEDBTCONTIME 值。例如，如果 DBPATH 包含三个条目，要为每一连接尝试花费至少 30 秒，则请设置 GBASEDBTCONTIME 为 '90'。

  ```
  SET ENVIRONMENT GBASEDBTCONTIME '90';
  ```

## NOVALIDATE 环境选项

使用 NOVALIDATE 环境选项来指定，在缺省情况下，ALTER TABLE ADD CONSTRAINT 语句创建的（或通过 SET CONSTRAINTS 语句已重置了它的约束模式）外键约束是否处于 NOVALIDATE 模式，除非在 DISABLED 模式下创建它们（或更改为 DISABLED 模式）。

启用此会话环境变量可防止在随后的 ALTER TABLE ADD CONSTRAINT 或 SET CONSTRAINTS ENABLED 或 SET CONSTRAINTS FILTERING 操作期间检查外键约束的引用完整性。在没有理由要期待完整性验证的上下文中，或可推迟外键约束的验证直到将表重新定位到另一数据库为止的上下文中，这可提高这些 DDL 语句的性能。

- 如果您将该值设置为 ON 或 '1'，则不需要显式地包括 NOVALIDATE 关键字来绕过对 ENABLED 或 FILTERING 约束的验证，在正在运行那些 DDL 语句之一时。
- 如果您将该值设置为 OFF 或 '0'，则恢复那些数据定义语言（DDL）语句的缺省行为，以便于在创建或启用外键约束的 ALTER TABLE 或 SET CONSTRAINTS 操作期间，数据库服务器自动地检查表中是否存在违反该约束的行。

注：

不管是否启用 SET ENVIRONMENT NOVALIDATE 会话环境选项，在那个 DDL 语句执行完成之后，自动地删除 ALTER TABLE ADD CONSTRAINT 语句或 SET Database Object Mode 语句的 SET CONSTRAINTS 选项将其应用到外键约束的对象模式的任何 NOVALIDATE 属性。外键约束的模式成为任何注册在 sysobjstate 系统目录表中的 SET CONSTRAINTS 或 ALTER TABLE 语句，不理会 NOVALIDATE 属性。

在您正在以 ALTER TABLE ADD CONSTRAINT 语句创建 ENABLED 或 FILTERING 外键约束，或以 SET CONSTRAINTS 语句将外键约束的模式更改为 ENABLED 或 FILTERING 时，NOVALIDATE 选项防止在 ALTER TABLE 或 SET CONSTRAINTS 语句正运行时数据库服务器检查表的每行是否符合引用约束。那可节省大量用于移动其引用完整性无可置疑的大型表的时间。

例如，下列语句启用 NOVALIDATE 会话环境变量：

```
SET ENVIRONMENT NOVALIDATE '1';
```

在当前的会话被连接到的数据库中，在下列对外键约束的 DDL 操作期间，它有这些后续的影响：

- 对于外键约束，SET Database Object Node 语句的 SET CONSTRAINTS 选项将缺省的或显式的约束模式更改为包括 NOVALIDATE，除非指定 DISABLED 作为约束模式。
- 对于由不带有显式模式的 ALTER TABLE ADD CONSTRAINT 指定的外键约束，在缺省情况下，在 ENABLED NOVALIDATE 模式下创建它。
- 对于由 ALTER TABLE ADD CONSTRAINT 语句在 ENABLED 或 FILTERING 模式下指定的外键约束，在缺省情况下，也处于 NOVALIDATE 模式。

下列示例恢复缺省的约束模式行为，在其中 NOVALIDATE 不是外键的缺省约束模式的一部分，在 SET CONSTRAINTS 或 ALTER TABLE ADD CONSTRAINT 操作期间，不处于 DISABLED 模式下：

```
SET ENVIRONMENT NOVALIDATE OFF;
```

对于随后的 SET CONSTRAINTS 或 ALTER TABLE ADD CONSTRAINT 语句，数据库服务器以该外键约束执行全表扫描或表的索引扫描，以便于验证该表的引用完整性。对于有几百万行的表，此验证的成本巨大。

通过启用 NOVALIDATE 会话环境选项，在 SET CONSTRAINTS 或 ALTER TABLE ADD CONSTRAINT 语句期间挂起外键约束检查，对于那些通过强制相同的约束的 OLTP 操作已填写的表，可提高效率。在以他们的被删除或禁用的外键约束将那些表移至另一数据库或数据仓库之后，在恢复引用约束时，您可使用 SET ENVIRONMENT NOVALIDATE ON 语句来避免进行违反检查。

## OPTCOMPIND 环境选项

使用 SET ENVIRONMENT 的 OPTCOMPIND 环境选项来指定查询优化器在随后的连接查询的方法，以及当前正在执行的例程的 MERGE 语句。此语句覆盖 **OPTCOMPIND** 环境变量的系统缺省设置。

OPTCOMPIND 环境选项可提升用于决策支持和联机事务处理的数据库的性能。 使用此选项来指定查询优化器在随后的查询中使用的连接方法。

- 如果值为 '0'，则查询优化器在可能的地方使用嵌套的循环连接，而不是排序-合并连接或哈希连接。
- 如果值为 '1' 且事务隔离级别为 Repeatable Read，则优化器的表现与上述的设置 '0' 相同；对于任何其他隔离级别，它都表现得像设置 '2' 一样，如后所述。
- 如果值为 '2'，则查询优化器不是必然地优选嵌套的循环连接，而是完全基于它对成本的估计作出决定，不理会事务隔离模式。

例如，下列替换任何先前生效的完全基于成本的优化器策略的 OPTCOMPIND 设置：

```
SET ENVIRONMENT OPTCOMPIND '2';
```

使用 DEFAULT 关键字来恢复系统缺省的值，如 *《GBase 8s SQL 指南：参考》* 的 **OPTCOMPIND** 主题中所述的那样。

要了解 OPTCOMPIND 选项的性能影响，请参阅您的 *GBase 8s 性能指南* 。

SET ENVIRONMENT OPTCOMPIND 指定的 OPTCOMPIND 设置的作用域局限于发出该语句的例程，并持续到例程退出，或直到例程发出另一 SET ENVIRONMENT OPTCOMPIND 语句，而不是在整个会话中持续。在例程终止之后，设置恢复到 **OPTCOMPIND** 环境变量指定的系统缺省值。

没有其他的 SET ENVIRONMENT 选项有局限于该例程的作用域。所有其他的 SET ENVIRONMENT 选项都持续到会话终止为止，或直到另一 SQL 语句重置它们的值为止。

## RETAINUPDATELOCKS 环境选项

RETAINUPDATELOCKS 环境选项可提高在包括 SELECT ... FOR UPDATE 语句的“动态的 SQL”应用中的并发性。如果会话正在使用 Committed Read、Dirty Read 或 Cursor Stability 隔离级别来启用（或禁用）SET ISOLATION 语句的 RETAIN® UPDATE LOCKS 子句，则此选项可修改当前的事务隔离级别在运行时的行为。

当为当前的隔离级别启用 RETAINUPDATELOCKS 环境选项时，在缺省情况下，数据库服务器保持在行上的更新锁，直到该事务结束。保持任何更新锁，直到提交或回滚该事务，不管定义了该隔离级别的 SET ISOLATION 语句是否包含了 RETAIN UPDATE LOCKS 关键字。当此选项设置为 ALL 或为当前的 GBase 8s 隔离级别的名称时（如果此级别为 Committed Read、Dirty Read 或 Cursor Stability），则此设置防止其他事务中的并发用户删除或更新您已在其上放置了更新锁，而您还未更新的行。

通过指定 NONE 为 RETAINUPDATELOCKS 设置，您禁用此特性并恢复缺省的锁定行为。 当 NONE 为设置时，除非已经通过显式地包括了 RETAIN UPDATE LOCKS 关键词的 SET ISOLATION 语句设置了隔离级别，数据库服务器在下一 FETCH 操作时，或当游标关闭时，释放该更新锁。

SET ENVIRONMENT RETAINUPDATELOCKS 语句对更新游标不起作用，如果 GBase 8s 隔离级别为 REPEATABLE READ 的话。 类似地，在作用域之外是已通过 SET TRANSACTION 语句设置了其隔离级别的事务，其定义符合 ANSI/ISO 的隔离级别，而不是 GBase 8s 隔离级别。（要获取更多关于 GBase 8s 和 ISO 隔离级别的信息，请参阅主题 对比 SET TRANSACTION 与 SET ISOLATION。）

RETAINUPDATELOCKS 选项接受可影响当前的 GBase 8s 隔离级别的五种设置中的任何一种，以及在 SET ENVIRONMENT 语句之后通过 SET ISOLATION 语句建立的隔离级别。对于除了 'NONE' 之外的每个设置，该设置的作用就是隐式地在 SET ISOLATION 规范中包括 RETAIN UPDATE LOCKS 关键字：

- 如果值为 'COMMITTED READ'，则数据库服务器保持任何更新锁，直到使用 Committed Read 隔离级别的事务的结束为止。
- 如果值为 'CURSOR STABILITY'，则数据库服务器保持任何更新锁，直到使用 Cursor Stability 隔离级别的事务的结束为止。
- 如果值为 'DIRTY READ'，则数据库服务器保持任何更新锁，直到使用 Dirty Read 隔离级别的事务的结束为止。
- 如果值为 'ALL'，则数据库服务器保持任何更新锁，直到使用 Committed Read、Dirty Read 或 Cursor Stability 隔离级别的事务的结束为止。
- 如果值为 'NONE'，则禁用 RETAINUPDATELOCKS 特性，直到该会话结束，或直到另一 SET ISOLATION 或 SET ENVIRONMENT 语句重新启用更新锁的保持为止。在 NONE 设置之下，如果您的应用定义更新游标，则数据库服务器在下一 FETCH 操作时，或当更新游标关闭时，释放它的更新锁。即使在 SET ENVIRONMENT RETAINUPDATELOCKS 'NONE' 语句执行之前，Committed Read、Dirty Read 或 Cursor Stability 隔离级别已强制了 RETAIN UPDATE LOCKS 行为，更新锁依然不被保持。

这些设置不区分大小写。例如，'ALL' 和 'all' 作用相同。

当发出 SET ENVIRONMENT RETAINUPDATELOCKS 语句时，它就生效（通过重置该会话环境）。可在事务的外部发出它。如果当前事务的隔离级别与在 RETAINUPDATELOCKS 关键字之后指定

的设置相匹配，则当发出该语句时，新的设置可更改正在运行的事务的 RETAIN UPDATE LOCKS 行为。

例如，考虑下列 SET ENVIRONMENT 和 SET ISOLATION 语句：

```
BEGIN WORK; --开始第一个事务
        SET ENVIRONMENT RETAINUPDATELOCKS 'COMMITTED READ';
        SET ISOLATION TO COMMITTED READ LAST COMMITTED;
        SELECT ... FOR UPDATE ...
        ...
        COMMIT WORK;
        SET ENVIRONMENT RETAINUPDATELOCKS 'DIRTY READ';
        BEGIN WORK; --开始第二个事务
        SET ISOLATION TO COMMITTED READ LAST COMMITTED;
        SELECT ... FOR UPDATE ...
        ...
        COMMIT WORK;
```

在上述的第一个事务中，SET ENVIRONMENT 语句中的 RETAINUPDATELOCKS 设置使得更新锁的保持成为 Committed Read 隔离级别的缺省的行为。因此，数据库服务器翻译第一个 SET ISOLATION 语句，其指定 Committed Read 但没有 RETAIN UPDATE LOCKS 子句，就好像它已经包括了那个子句：

```
SET ISOLATION TO
        READ LAST COMMITTED RETAIN UPDATE LOCKS;
```

然而，由于在第二个事务中的 SET ENVIRONMENT RETAINUPDATELOCKS 语句指定 DIRTY READ 作为它的设置，它对第二个 SET ISOLATION 语句不起作用，其定义 Committed Read 隔离级别。对应于特定的 GBase 8s 隔离级别的每一设置仅影响使用相同隔离级别的事务中的更新锁。

但是在跨服务器 SELECT ... FOR UPDATE 分布式查询中，有些参与的服务器不支持更新锁保持，整个事务符合发出了该事务的会话的隔离级别。如果那个会话有启用的 RETAINUPDATELOCKS 选项生效，对于支持更新锁保持的服务器它也生效，但其他参与的服务器为了释放更新锁，遵循它们的缺省行为。

如果在其中发出语句的数据库不支持事务日志记录，则 SET ENVIRONMENT RETAINUPDATELOCKS 语句失败并报错 -26199。

**sysdbopen( ) 过程**

当您的会话连接到其中定义了 sysdbopen( ) 的数据库时，内建的 sysdbopen( ) 例程可发出 SET ENVIRONMENT RETAINUPDATELOCKS 语句，如下例中所示。

```
CREATE PROCEDURE PUBLIC.SYSDBOPEN()
        SET PDQPRIORITY 10;
        SET ENVIRONMENT RETAINUPDATELOCKS 'ALL';
        END PROCEDURE
```

在上述示例生效之后，它防止其他会话更改您已在其上放置了更新锁的行，以便于您可在当前事务中稍后再更新这些行。除非您在相同的会话之内发出另一 SQL 语句来禁用更新锁的保持，否则

**sysdbopen( )** 发出的 SET ENVIRONMENT RETAINUPDATELOCKS 语句的作用保持，直到该会话结束为止。然而， **sysdbopen( )** 指定的 RETAINUPDATELOCKS 值的会话内持续是特殊情况。任何其他 SPL 例程都可使用 SET ENVIRONMENT 语句来指定为 Committed Read、Dirty Read 或 Cursor Stability 事务隔离级别，或为 'ALL'，指定更新锁保持，但仅在例程执行时保持它们的作用，而不是在例程退出之后。

**重置缺省的更新锁行为**

在 GBase 8s 早于版本 11.50.xC6 的版本中，最近执行的 SET ISOLATION 语句为随后的事务指定缺省值。如果最近的 SET ISOLATION 语句包括了 RETAIN UPDATE LOCKS 子句，则有必要为相同的隔离级别执行 SET ISOLATION 语句（但不带有 RETAIN UPDATE LOCKS 子句）来禁用更新锁的保持。然而，目前，如果 SET ENVIRONMENT RETAINUPDATELOCKS 已启用了保持，则您必须显式地运行 SET ENVIRONMENT RETAINUPDATELOCKS 'NONE' 语句来恢复不保持作为缺省的行为，如下例中所示。

```
BEGIN WORK;  --开始第一个事务
        SET ISOLATION TO COMMITTED READ LAST COMMITTED;
        SET ENVIRONMENT RETAINUPDATELOCKS 'COMMITTED READ';
        SELECT ... FOR UPDATE ...
        ...
        COMMIT WORK;
        BEGIN WORK;  --开始第二个事务
        SET ENVIRONMENT RETAINUPDATELOCKS 'NONE';
        SET ISOLATION TO COMMITTED READ LAST COMMITTED;
        SELECT ... FOR UPDATE ...
        ...
```

在上述第一个事务中，第一个 SET ENVIRONMENT 语句将当前事务的 Committed Read 隔离级别的行为更改为保持更新锁，即使在该事务之内按照词典顺序，建立了那个隔离级别的 SET ISOLATION 语句仍在 SET ENVIRONMENT 语句之前。通过此 SET ENVIRONMENT 语句对此隔离级别的 LAST COMMITTED 规范不起作用。

然而，按字面翻译第二个事务中的 SET ISOLATION 语句，因为通过第二个 SET ENVIRONMENT 语句将缺省的行为重置为了 NONE。

**在高可用性集群中的更新锁保持**

在高可用性集群环境中，RETAINUPDATELOCKS 选项仅在主服务器上有效。需要更新锁的保持的应用必须在主服务器上运行，如果它们包括 SET ENVIRONMENT RETAINUPDATELOCKS 语句的话。当从辅助服务器发出它时，该语句对锁定行为没有影响，且服务器返回错误。

## SELECT_GRID 环境选项

使用 SET ENVIRONMENT 语句的 SELECT_GRID 选项来为网格查询定义缺省的 GRID 子句。此子句指定缺省的网格或区域，从其来返回等同于来自参与的网格服务器的符合条件的行的逻辑 UNION 的结果集。

SELECT_GRID 会话环境选项可设置为任何三个选项：

'grid'

这指定缺省的网格，不包括 GRID 子句的 SELECT 语句从其返回结果集，该结果集是来自指定的 ***grid*** 中所有参与的网格服务器的不同的符合条件的值的逻辑 UNION。在 SET ENVIRONMENT 的 SELECT_GRID 选项设置为网格的名称时，发出 SET ENVIRONMENT SELECT_GRID '*grid*' 语句的数据库服务器将所有查询翻译为 UNION 网格查询。

'region'

这指定缺省的区域，不包括 GRID 子句的 SELECT 语句可从其返回结果集，该结果集是来自指定的区域中所有参与的网格服务器的不同的符合条件的值的逻辑 UNION。在 SET ENVIRONMENT 语句的 SELECT_GRID 选项设置为区域的名称时，发出 SET ENVIRONMENT SELECT_GRID '*region*' 语句的数据库服务器将所有查询翻译为 UNION 网格查询。

DEFAULT

此设置指定缺省的行为。在 SET ENVIRONMENT 语句的 SELECT_GRID 选项设置为 DEFAULT 时，数据库服务器不处理每个没有 GRID 子句的查询作为网格查询。因此，DEFAULT 设置指定没有 SELECT 语句的缺省的 GRID 子句。使用此选项来禁用先前的 SET ENVIRONMENT SELECT_GRID 或 SET ENVIRONMENT SELECT_GRID_ALL 语句的作用，该语句为不包括 GRID 子句的所有查询定义了缺省的 GRID 子句。

SET ENVIRONMENT 语句的 SELECT_GRID 与 SELECT_GRID_ALL 选项是相互排他的。在您发出 SET ENVIRONMENT SELECT_GRID 或 SET ENVIRONMENT SELECT_GRID_ALL 语句之后，在相同的会话中发出任一个语句的不同的选项，会取代先前的 SET ENVIRONMENT SELECT_GRID 或 SET ENVIRONMENT SELECT_GRID_ALL 语句建立的作为网格查询的任何服务器行为。

在 SELECT_GRID 会话环境变量设置为 ON 时，请不要调用 ifx_gridquery_skipped_nodes() 函数。

在网格上下文之外，SELECT_GRID 会话环境选项不应设置为 '*grid*' 或 '*region*'。要获取更多关于网格的信息，请参阅 GRID 子句 和 *GBase 8s Enterprise Replication 指南*。

## SELECT_GRID_ALL 环境选项

使用 SET ENVIRONMENT 的 SELECT_GRID_ALL 选项来为网格查询定义缺省的 GRID 子句。此子句指定缺省的网格或区域，从其来返回等同于来自参与的网格服务器的符合条件的行的逻辑 UNION ALL 结果集。

SELECT_GRID_ALL 会话环境选项可设置为任意三个值：

'grid'

这指定缺省的网格，不包括 GRID 子句的 SELECT 语句可从其返回结果集，该结果集是来自指定的 ***grid*** 中的所有参与的网格服务器的符合条件值的逻辑 UNION ALL，包括重复的值。在 SET ENVIRONMENT 语句的 SELECT_GRID 选项设置为网格的名称时，发出 SET ENVIRONMENT SELECT_GRID_ALL '*grid*' 语句的数据库服务器将所有查询翻译为 UNION ALL 网格查询。

'region'

这指定缺省的区域，不包括 GRID 子句的 SELECT 语句从其返回结果集，该结果集是来自指定的区域中的所有参与的网格服务器的符合条件的值的逻辑 UNION ALL，包括重复的值。在 SET ENVIRONMENT 语句的 SELECT_GRID 选项设置为区域的名称时，发出 SET ENVIRONMENT SELECT_GRID_ALL '*region*' 语句的数据库服务器将所有查询翻译为 UNION ALL 网格查询。

DEFAULT

此设置指定缺省的行为。在 SET ENVIRONMENT 语句的 SELECT_GRID_ALL 选项设置为 DEFAULT 时，数据库服务器不将每个没有 GRID 子句的查询作为网格查询处理。因此，DEFAULT 设置指定没有 SELECT 语句的缺省的 GRID 子句。使用此选项来禁用先前的 SET ENVIRONMENT SELECT_GRID 或 SET ENVIRONMENT SELECT_GRID_ALL 语句的作用，该语句为不包括 GRID 子句的所有查询定义了缺省的 GRID 子句。

SET ENVIRONMENT 语句的 SELECT_GRID 与 SELECT_GRID_ALL 选项是相互排他的。在您发出 SET ENVIRONMENT SELECT_GRID_ALL 或 SET ENVIRONMENT SELECT_GRID 语句之后，在相同的会话中发出任一语句的不同选项，会取代为网格查询作为服务器行为建立的任何先前的 ENVIRONMENT SELECT_GRID_ALL 或 SET ENVIRONMENT SELECT_GRID 语句。

在 SELECT_GRID_ALL 会话环境变量设置为 ON 时，请不要调用 ifx_gridquery_skipped_nodes() 函数。

在网格上下文之外，SELECT_GRID_ALL 会话环境选项不应设置为 '*grid*' 或 '*region*'。要获取更多关于网格的信息，请参阅 GRID 子句 和 *GBase 8s Enterprise Replication 指南*。

## STATCHANGE 环境选项

当启用将 UPDATE STATISTICS 操作限制到陈旧的或丢失的分发的自动模式时，使用 STATCHANGE 环境选项来为 UPDATE STATISTICS 语句的更改阈值的全局百分率指定要使用的正整数。

当 AUTO_STAT_MODE 配置参数或 AUTO_STAT_MODE 环境选项已经为 UPDATE STATISTICS 语句启用了自动模式时，使用 STATCHANGE 环境选项的值，以便于它有选择地仅刷新陈旧的数据分发。

您为 STATCHANGE 设置的值指定更改阈值，来决定当 UPDATE STATISTICS 语句正在自动模式下操作时，分发统计是否符合更新的要求。

STATCHANGE 配置参数可指定正整数作为更改阈值的百分率来定义陈旧的数据分发。当启用 UPDATE STATISTICS 的自动模式时，此设置作为任何表的缺省的更改阈值生效，其 STATCHANGE 表属性指定为 AUTO，或在缺省情况下为 AUTO。STATCHANGE 配置参数的缺省值是 10。然而，您可使用 SET ENVIRONMENT STATCHANGE 语句来指定一整数值，该值为当前的会话覆盖显式的或缺省的 STATCHANGE 配置参数设置。

您可为 STATCHANGE 会话环境选项指定一个取值从 0 至 100 的整数百分率值。

**SET ENVIRONMENT STATCHANGE 的示例**

下列语句为服务器设置要使用的阈值，来确定分发统计是否符合更新到 50%：

```
SET ENVIRONMENT STATCHANGE '50';
```

## USELASTCOMMITTED 环境选项

当两个或多个会话尝试访问其锁定颗粒度为行级锁定的表中的相同行时，通过降低锁定冲突的风险，USELASTCOMMITTED 环境选项可提高使用 Committed Read、Dirty Read、Read Committed 或 Read Uncommitted 隔离级别的会话中的并发能力。

在更改数据值时，SET ENVIRONMENT USELASTCOMMITTED 语句可指定遇到其他会话持有的排他锁的查询和其他操作是否应使用数据的最近提交的版本，而不是等待锁被释放。

此语句可在当期会话期间覆盖 USELASTCOMMITTED 配置参数。您可使用 SET ISOLATION 语句来覆盖 USELASTCOMMITTED 会话环境设置。

USELASTCOMMITTED 选项可有四个值的任意一个：

- 如果该值为 'COMMITTED READ'，则在尝试读取 Committed Read 或 Read Committed 隔离级别的行时，当它遇到排他锁时，数据库服务器读取数据的最近提交的版本。
- 如果该值为 'DIRTY READ'，那么在尝试读取 Dirty Read 或 Read Uncommitted 隔离级别的行时，如果它遇到排他锁，则数据库服务器读取数据的最近提交的版本。
- 如果该值为 'ALL'，那么在尝试读取 Committed Read、Dirty Read、Read Committed 或 Read Uncommitted 隔离级别的行时，如果它遇到排他锁，则数据库服务器读取数据的最近提交的版本。
- 如果该值为 'NONE'，则此值禁用可访问被锁定的行中数据的最后提交的版本的 USELASTCOMMITTED 特性。在此设置之下，当尝试读取 Committed Read、Dirty Read、Read Committed 或 Read Uncommitted 隔离级别的行时，如果您的会话遇到排他锁，则您的会话不可读取那行，直到提交了或回滚了持有该排他锁的并发会话为止。

例如，下列语句指定 Committed Read 隔离模式，并将显式的或缺省的 USELASTCOMMITTED 配置参数设置替换为一设置，在并发读者持有排他锁的行上，该设置读取数据的最近提交的版本：

```
SET ISOLATION COMMITTED READ;
SET ENVIRONMENT USELASTCOMMITTED 'ALL';
```

在会话期间，任何 SPL 例程都可使用这些语句来指定 Committed Read Last Committed 事务隔离级别。在读取行的操作期间，当遇到排他锁时，这些语句使得读取数据的 SQL 操作能够使用最后提交的版本。当另一会话正尝试修改相同的行时，这可避免死锁状况或其他锁定错误。它不会减少与正在写表的其他会话，或与在用户表上或在系统目录表上持有隐式的或显式的锁的并发 DDL 事务，之间发生锁定冲突的风险。

例如，在 PUBLIC.sysdbopen 或 *user*.sysdbopen 过程内的下列语句指定在连接时刻的 Committed Read 隔离模式，并将显式的或缺省的 USELASTCOMMITTED 配置参数设置替代为一设置，在并发读者在其上持有排他锁的表中。该设置读取数据的最近提交的版本。：

```
SET ISOLATION COMMITTED READ;
SET ENVIRONMENT USELASTCOMMITTED 'ALL';
```

除了 sysdbopen( ) 之外，任何 SPL 例程都可在会话期间，使用这些语句来指定 Committed Read Last Committed 事务隔离级别。在读取表的操作期间，当遇到排他锁时，这些语句使得读取数据的 SQL 操作能够使用最后提交的版本。当另一会话正尝试修改同一行或表时，这可避免死锁状况或其他锁定错误。它不会降低与正在写表的其他会话，或与对用户表或对系统目录表持有隐式的或显式的锁的并发 DDL 事务，之间发生锁定冲突的风险。

在跨服务器分布式查询中，如果发出该查询的会话的隔离级别有 LAST COMMITTED 隔离级别选项在生效，但一个或多个参与的数据库不支持此 LAST COMMITTED 特性，则整个会话符合该会话的 Committed Read 或 Dirty Read 隔离级别，该会话发出了事务，未启用 LAST COMMITTED 选项。

在启用 USELASTCOMMITTED 时，可防止事务从被另一事务锁定的表读取最近提交的数据。要获取关于这方面的信息，请参阅 Committed Read 的 LAST COMMITTED 选项。

要获取更多关于 USELASTCOMMITTED 配置参数的信息，请参阅您的 *GBase 8s 管理员参考手册*。

### USTLOW_SAMPLE 环境选项

使用 USTLOW_SAMPLE 会话环境选项来为当前会话中的 UPDATE STATISTICS LOW 操作收集索引统计信息期间启用或禁用抽样。

在缺省情况下，当 UPDATE STATISTICS 语句为在其上定义一个或多个索引的表收集分发统计信息时，数据库服务器读取序列中所有索引叶子页来计算索引统计信息，诸如叶子页的数目、唯一的引导索引-键值的数目，以及集群信息。

对于带有多于 100 KB 叶子页的索引，从抽样来估算这些索引统计信息可提高 UPDATE STATISTIC LOW 操作的速度。

要设置 USTLOW_SAMPLE 会话环境变量，请指定：

- '0' 或 OFF 来禁用抽样
- '1' 或 ON 来启用抽样

您指定的值覆盖 USTLOW_SAMPLE 配置参数为该会话的设置。

例如，要在当前的会话中为索引统计启用抽样，请使用任一下列语句：

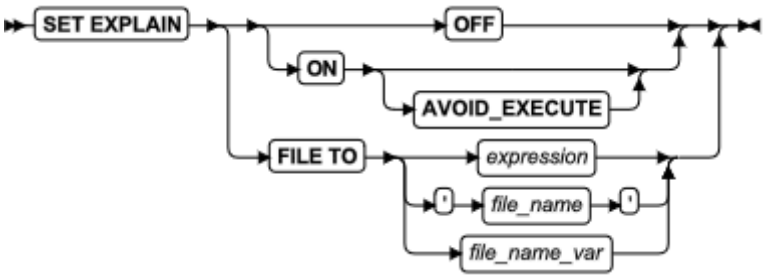
```
SET ENVIRONMENT USTLOW_SAMPLE '1';
SET ENVIRONMENT USTLOW_SAMPLE ON;
```

您不可控制在采样中的数据数量。

## 2.135 SET EXPLAIN 语句

使用 SET EXPLAIN 语句来启用或禁用在当前的会话中查询的记录评估，包括查询优化器的计划、对返回行的数目的估计，以及该查询的相对成本。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 返回文件名称规范的表达式 | 必须返回满足文件名称限制的字符串 | 表达式 |
| *file_name* | 说明输出文件名称。如果未包括文件的绝对路径，则会在默认的说明输出文件位置创建说明输出文件 | 必须符合操作系统规则。如果该文件已存在，则将说明输出追加在它后面。 | 引用字符串 |
| *file_name_var* | 存储文件名称的主变量 | 必须为字符数据类型 | 特定于语言 |

**用法**

将来自 SET EXPLAIN ON 语句的输出指向适合的文件，直到您发出 SET EXPLAIN OFF 语句或直到程序结束为止。如果您未输入 SET EXPLAIN 语句，则缺省的行为是 OFF，且数据库服务器不生成查询的评估。

在数据库服务器优化阶段期间，执行 SET EXPLAIN 语句，当您初始化查询时会发生。对于与游标相关联的查询，如果准备该查询且没有主变量，则当您准备它时发生优化。否则，当您打开游标时，发生优化。

SET EXPLAIN 语句提供参与执行查询的工作的多种评估。

| 选项 | 作用 |
|---|---|
| ON | 为每一随后的查询生成评估并将结果写到当前目录中的输出文件。如果给文件已存在，则将新的输出追加到现有的文件之后。 |
| AVOID_EXECUTE | 防止 SELECT、INSERT、MERGE、UPDATE 或 DELETE 语句执行。数据库服务器将查询计划打印到输出文件 |
| OFF | 终止 SET EXPLAIN 语句的活动，以便于不再生成随后的查询的评估或写到输出文件 |

FILE TO　　为每一随后的查询生成评估，并允许您指定说明输出文件的位置。

下列示例为当前会话中随后的查询将查询计划写到说明输出文件中：

```
SET EXPLAIN ON;
```

下列示例将查询计划追加写到当前会话中的文件：

```
SET EXPLAIN OFF;
```

## 使用 AVOID_EXECUTE 选项

AVOID_EXECUTE 关键字防止 DML 语句执行。相反，数据库服务器将查询计划打印到输出文件。

SET EXPLAIN ON AVOID_EXECUTE 语句为会话激活 Avoid Execute 选项，或直到下一不带 AVOID_EXECUTE 的 SET EXPLAIN OFF（或 ON）为止。如果您为包含远程表的查询激活 AVOID_EXECUTE，则该查询既不在本地也不在远程站点执行。

下列示例在指定的文件中存储输出。

```
SET EXPLAIN ON AVOID_EXECUTE;
SET EXPLAIN FILE TO '/tmp/explain.out';
```

当设置 AVOID_EXECUTE 时，数据库服务器发出警告消息。如果您正在使用 DB-Access，则它为任何选择、删除、更新或插入查询操作显示文本消息。

```
Warning! avoid_execute has been set
```

对于 ESQL，sqlwarn.sqlwarn7 字符设置为 'W'。

使用 SET EXPLAIN ON 或 SET EXPLAIN OFF 语句来关闭 AVOID_EXECUTE 选项。SET EXPLAIN ON 语句关闭 AVOID_EXECUTE 选项，但继续生成查询计划并将结果写到输出文件。

如果您在 SPL 例程中发出 SET EXPLAIN ON AVOID_EXECUTE 语句，则该 SPL 例程和任何 DDL 语句仍然执行，但在该 SPL 例程内部的 DML 语句不执行。数据库服务器将该 SPL 例程的查询计划打印到输出文件。要关闭此选项，您必须在该 SPL 例程的外部执行 SET EXPLAIN ON 或 SET EXPLAIN OFF 语句。如果您在执行 SPL 例程之前执行 SET EXPLAIN ON AVOID_EXECUTE 语句，则在该 SPL 例程内部的 DML 语句不执行，且数据库服务器不将该 SPL 例程的查询计划打印到输出文件。

当 AVOID_EXECUTE 生效时，仍然对查询中的恒定函数求值，因为数据库服务器在优化之前计算这些函数。

例如，即使不执行下列 SELECT 语句，也对 func( ) 函数求值：

```
SELECT * FROM orders WHERE func(10) > 5;
```

要了解 AVOID_EXECUTE 选项的其他性能影响，请参阅 *GBase 8s 性能指南* 。

如果您在 GBase 8s ESQL/C 程序中打开游标之前执行 SET EXPLAIN ON AVOID_EXECUTE 语句，则每一 FETCH 操作都返回找不到行的消息。然而，如果您在 GBase 8s ESQL/C 程序打开游标之后执行 SET EXPLAIN ON AVOID_EXECUTE，则此语句对游标不起作用，其继续返回行。

## 使用 FILE TO 选项

当您执行 SET EXPLAIN FILE TO 语句时，开启说明输出。SET EXPLAIN FILE TO 语句可更改说明输出的缺省的文件名称，直到会话结束为止，或直到发出另一 SET EXPLAIN 语句为止。

该 filename 可为任何路径与文件名称的任何组合。如果未指定路径，则将该未见值域缺省的说明输出位置。当前的用户拥有该文件的权限。

您在 SET EXPLAIN 语句中指定的输出文件可为新文件或现有的文件。如果 FILE TO 子句指定现有的文件，则将新的输出追加到那个文件的后面。

## 在 UNIX™ 上的说明输出文件的缺省名称和位置

当您发出 SET EXPLAIN ON 语句时，将优化器为每一随后的查询选择的计划写到说明输出文件。

当您发出 SET EXPLAIN ON 时，如果说明输出文件不存在，则数据库服务器创建该文件。如果当您发出 SET EXPLAIN ON 语句时，说明输出文件已存在，则将随后的输出追加到该文件后面。

### 说明输出文件的缺省名称

由 SET EXPLAIN 语句生成的说明输出文件与由 gadmin -Y 生成的说明文件有不同的名称。映射的用户的说明输出 filename 与 OS 用户的说明输出 filename 也不相同。下表展示缺省的名称：

表 1. 缺省的说明输出文件名称。

| 用户与生成类型 | 文件名称 |
| --- | --- |
| 常规用户与 SET EXPLAIN | sqexplain.out |
| 映射的用户与 SET EXPLAIN | *username*_sqexplain.out |
| 常规用户与 gadmin -Y | sqexplain.out.*session_id* |
| 映射的用户与 gadmin -Y | *username*_sqexplain.out.*session_id* |

### 说明输出文件的缺省位置

如果客户端应用与数据库服务器在同一台计算机上，则输出文件存储在您的当前名录中。如果您正在使用 Version 5.x 或更早的客户端应用且输出文件不出现在当前的目录中，则请检查您对于该文件的 home 目录。当当前的数据库在另一台计算机上时，该输出文件存储在远程主机上您的 home 目录中。

对于没有 home 目录的映射的用户，说明输出文件存储在 $GBASEDBTDIR/users/server_*svrnum*/uid_*uid* 中。

对于带有 home 目录的映射的用户，远程客户端的说明输出文件存储在该用户的 home 目录中，且本地客户的说明输出文件存储在用户的当前工作目录中。

## 在 Windows 上的输出文件的缺省名称和位置

在 Windows™ 上，SET EXPLAIN ON 将优化器为每一随后的查询选择的计划写到 %GBASEDBTDIR%\sqexpln 中的文件。

对于由 SET EXPLAIN 语句生成的说明输出文件，缺省的文件名称为 *user_name*.out，在此，*user_name* 是用户登录名。

对于由 gadmin -Y 生成的说明输出文件，缺省的文件名称为 sqexplain.out.*session_id*。

## SET EXPLAIN 输出

查看 SET EXPLAIN 输出文件来分析关于执行了的查询的信息，包括查询的伪指令集、对查询成本的估算、对返回行数的估计、服务器访问的表中的顺序、索引键、连接方式和查询统计信息。

下表罗列可出现在输出文件中的术语及其意义。

| 术语 | 意义 |
| --- | --- |
| 查询 | 显示执行了的查询，并指示将 SET OPTIMIZATION 设置为了 HIGH 还是 LOW。如果您 SET OPTIMIZATION 为 LOW，则输出显示下列大写字符串作为第一行：QUERY:{LOW}<br>如果您 SET OPTIMIZATION 为 HIGH，则 SET EXPLAIN 的输出显示下列大写字符串作为第一行：QUERY: |
| 后跟的伪指令 | 罗列查询的伪指令集<br>如果伪指令的语法不正确，则处理不带伪指令的查询。在那种情况下，输出在 DIRECTIVES FOLLOWED 之外还展示 DIRECTIVES NOT FOLLOWED。<br>要获取更多关于在此术语之后指定的伪指令的信息，请参阅 优化程序伪指令 或 SET OPTIMIZATION 语句。<br>如果 DELETE 或 UPDATE 语句在 WHERE 子句中指定不相关联的子查询，则由该子查询返回的符合条件的行的集合具体化为临时表，SET EXPLAIN 的输出显示在下列消息的括号之内：(Temp Table For Subquery) |
| 估计的成本 | 该查询的工作量的估计值<br>优化器使用估计值来比较一路径与另一路径的成本。估计值是优化器赋予被选择的访问方式的一个数。此数不直接地译为时间，且不用于比较不同的查询。然而，它可用于比较同一查询产生的更改。当使用数据分发时，带有较高估计值的查询通常比带有较小估计值的查询花费更长的运行时间。<br>在查询与子查询的情况下，返回两个估计的成本数字；查询数字还包括子查询成本。展示子查询成本以便于您可看到仅与子查询相关联的成本。 |
| 返回的行的 | 要返回的行的数目的估计值 |

| 术语 | 意义 |
| --- | --- |
| 估计数 | 此数值是基于系统目录表中的信息的。 |
| 编号的列表 | 访问表的次序，后跟所使用的访问方式（索引路径或顺序扫描）当查询涉及表继承时，罗列在超级表之下以访问它们的顺序为顺序的所有表。 |
| 索引名称 | 索引的名称<br>例如，idx1 是下列索引的名称：<br>Index Name: gbasedbt.idx1<br>索引名称中的 FOT 标识该索引为树型索引的森林：例如，下列索引是树型索引的森林：<br>Index Name: gbasedbt.fot_idx (FOT) |
| 索引键 | 用作过滤器或索引的列；指示用于索引路径或过滤器的列名称。符号 ***(Key Only)*** 指示所有期望的列都是索引键的一部分，因此索引的仅键读取可被实际表的读取所取代。在有 NLSCASE INSENSITIVE 属性的数据库中，所有索引扫描方式（仅键扫描除外）都允许查询执行计划将所有区分大小写的值映射到 NCHAR 和 NVARCHAR 列的单个值。要获取更多关于 NLSCASE INSENSITIVE 数据库的信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行。<br>***“下部索引过滤器”***在索引读取开始处展示键值；在索引读取停止处展示键值的***“上部索引过滤器”***。***索引键过滤器***展示将用在被检索的索引键值的过滤器。如果查询使用索引自连接路径，则***“索引自连接键”***展示用作自连接键的引导索引键，且***下限***和***上限***展示引导索引键列的边界。 |
| 连接方式 | 当查询涉及两表之间的连接时，优化器使用的（“嵌套的循环”或“动态哈希”）连接方式展示在那个查询的输出的底部。<br>当查询涉及两表的动态连接时，如果输出包含词 ***Build Outer***，则在罗列的第一个表（成为构建表）上构建哈希表。如果未出现词 ***Build Outer***，则在罗列的第二个表上构建哈希表。 |
| 查询统计信息 | 当 EXPLAIN_STAT 配置参数设置为 1 时，这部分展示返回的行数、在查询计划中估计的行数、需要的时间、对 iterator 函数的调用，以及对表对象的扫描和连接操作的估计成本。 |
| 时间 | 当输出显示查询执行计划或那个计划的组件消耗的时间时，该值的格式为 *minutes*:*seconds*.*fraction* 来显示分、表和秒的小数部分。 |

如果查询使用核对顺序而不是 **DB_LOCALE** 设置的缺省顺序，则输出文件包括 **DB_LOCALE** 设置以及作为在该查询中（通过 SET COLLATION 指定的）核对基础的其他语言环境的名称。类似地，如果由于它的核对而不使用索引，则输出文件作此指示。

# 完整连接级别设置和输出示例

SET EXPLAIN 语句支持***完整连接级别***设置。

SET EXPLAIN 语句支持***完整连接级别***设置。这意味着在连接时将本地会话环境中的值传播到所有下列类型的新的或恢复的事务：

- 本地数据库之内的事务
- 跨同一服务器实例的数据库的分布式事务
- 跨两个或多个数据库服务器实例的数据库的分布式事务
- 带有在本地数据库中注册的符合 XA 的数据源的全局事务

如果您更改事务之内的 SET EXPLAIN 设置，则将新的值传播回到本地环境以及所有随后的新的或恢复的事务。

**SET EXPLAIN 输出的示例**

下列 SQL 语句导致数据库服务器将 UPDATE 语句(及其子查询)的查询计划写到缺省的输出文件：

```
DATABASE stores_demo;
SET EXPLAIN ON;
UPDATE orders SET ship_charge = ship_charge + 2.00
    WHERE customer_num IN
        (SELECT orders.customer_num FROM orders
            WHERE orders.ship_weight < 50);
CLOSE DATABASE;
```

在结果的输出中显示下列信息：

```
QUERY:
------
update orders set ship_charge = ship_charge + 2.00
where customer_num in
(select orders.customer_num from orders where
            orders.ship_weight < 50)

Estimated Cost: 4
Estimated # of Rows Returned: 8

  1) gbasedbt.orders: INDEX PATH

    (1) Index Keys: customer_num     (Serial, fragments: ALL)
        Lower Index Filter: gbasedbt.orders.customer_num = ANY

    Subquery:
    ---------
    Estimated Cost: 2
    Estimated # of Rows Returned: 8
    (Temp Table For Subquery)
```

```
    1) gbasedbt.orders: SEQUENTIAL SCAN

            Filters: gbasedbt.orders.ship_weight < 50.00
```

下一示例基于下列 SQL 语句，其中包括 DELETE 操作：

```
DATABASE stores_demo;
SET EXPLAIN ON;
DELETE FROM catalog WHERE stock_num IN
 (SELECT stock.stock_num FROM stock, catalog WHERE
        stock.stock_num = catalog.stock_num
        AND stock.unit_price < 50);
CLOSE DATABASE;
```

以下是结果输出：

```
QUERY:
------
DELETE FROM catalog WHERE stock_num IN
 (SELECTstock.stock_num from stock, catalog
    WHERE stock.stock_num = catalog.stock_num
    AND stock.unit_price < 50);

Estimated Cost: 19
Estimated # of Rows Returned: 37

  1) ajay.catalog: INDEX PATH

    (1) Index Keys: stock_num manu_code     (Serial, fragments: ALL)
        Lower Index Filter: ajay.catalog.stock_num = ANY

    Subquery:
    ---------
    Estimated Cost: 12
    Estimated # of Rows Returned: 44
    (Temp Table For Subquery)

      1) ajay.stock: SEQUENTIAL SCAN

            Filters: ajay.stock.unit_price < $50.00

      2) ajay.catalog: INDEX PATH

        (1) Index Keys: stock_num manu_code
            (Key-Only)   (Serial, fragments: ALL)
            Lower Index Filter:
            ajay.stock.stock_num = ajay.catalog.stock_num
```

NESTED LOOP JOIN

## 在 SET EXPLAIN 输出中的外部表操作

SET EXPLAIN 的 Query Statistics 部分提供关于从外部表加载数据或将数据卸载到外部表的操作的信息。

SET EXPLAIN 输出文件的 Query Statistics 部分中的下列代码提供关于外部表的信息：

- `xlcnv` 标识从外部表加载数据并将数据插入基础表的操作。此处，`x =` 外部表，`l =` 加载，且 `cnv =` 转换器
- `xucnv` 标识从基础表读数据并写到外部表正指向的文件的操作。此处，`x =` 外部表，`u =` 卸载，且 `cnv =` 转换器

示例

下列示例展示一查询，其中的操作是从外部表加载数据并将数据插入到基础表内：

```
QUERY: (OPTIMIZATION TIMESTAMP: 11-11-2009 12:55:20)
------
insert into items select * from ext_items

Estimated Cost: 5
Estimated # of Rows Returned: 68

  1) gbasedbt.ext_items: SEQUENTIAL SCAN

Query statistics:
-----------------
  Table map :
  ----------------------------
  Internal name        Table name
  ----------------------------
  t1                   items

  type        it_count     time
  ----------------------------
  xlread      1            00:00.00

  type        it_count     time
  ----------------------------
  xlcnv       67           00:00.00

  type        table    rows_ins     time
  -----------------------------------
  insert      t1       67           00:00.00
```

下列示例展示一查询，其中的操作是从基础表读数据并写到外部表指向的文件：

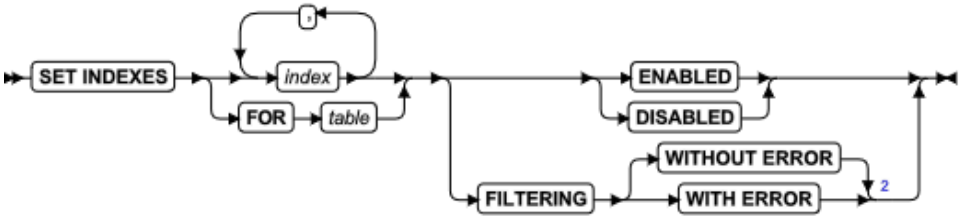
```
QUERY: (OPTIMIZATION TIMESTAMP: 11-11-2009 12:47:55)
------
select * from orders into external ord_ext
using (datafiles ('disk:/tmp/ord'))


Estimated Cost: 2
Estimated # of Rows Returned: 23

   1) gbasedbt.orders: SEQUENTIAL SCAN


Query statistics:
-----------------

   Table map :
   ----------------------------
   Internal name        Table name
   ----------------------------
   t1                    orders

   type       table   rows_prod   est_rows   rows_scan   time          est_cost
   ---------------------------------------------------------------------------
   scan       t1      23          23         23          00:00.00      3

   type       it_count    time
   ---------------------------
   xucnv      23          00:00.00

   type       it_count    time
   ---------------------------
   xuwrite    23          00:00.00
```

## 2.136 SET INDEXES 语句

使用 SET INDEXES 语句来启用或禁用用户定义的索引，或更改唯一索引的过滤模式。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *index* | 要启用、禁用或更改它的过滤模式的索引 | 必须存在 | 标识符 |
| *table* | 要全部启用、禁用或更改其索引的过滤模式的表 | 必须存在 | 标识符 |

**用法**

您可使用此语句来启用或禁用特定的索引或索引的列表。您还可使用 *table* 选项来在未指定其单个标识符的表上启用或禁用所有用户定义的索引。例如，以下两个示例分别地禁用和启用 cust_calls 表上的所有索引：

```
SET INDEXES FOR cust_calls DISABLED;
SET INDEXES FOR cust_calls ENABLED;
```

在您打算 LOAD 或 TRUNCATE 表中的所有数据，或合并表中的空闲空间的地方，此简单的语法可便于操作。

**显式定义的索引和隐式定义的索引**

SET INDEXES 语句对 CREATE INDEX 语句显式地创建的索引进行操作。然而，这对 PRIMARY KEY 或 FOREIGN KEY 约束定义显式地创建的系统定义的索引不起作用。SET INDEXES 语句不可指定以空格（ASCII 32）字符开头的系统生成的名称，即使您的数据库有 **DELIMIDENT** 环境变量设置来支持双引号作为数据库对象标识符的定界符。

要启用或禁用隐式定义的索引，请改为使用 SET CONSTRAINTS 语句，其 FOR *table* 选项可隐式地引用系统生成的约束，如下例中所示：

```
SET CONSTRAINTS FOR cust_calls DISABLED;
SET CONSTRAINTS FOR cust_calls ENABLED;
```

要禁用表的所有显式定义的索引和隐式定义的索引，请同时使用 SET INDEXES 和 SET CONSTRAINTS 语句的 FOR *table* 选项，如下例中所示：

```
SET INDEXES FOR cust_calls DISABLED;
SET CONSTRAINTS FOR cust_calls DISABLED;
```

通过将上例中的 DISABLED 替换为 ENABLED ，您可类似地启用表的所有显式定义的索引和隐式定义的索引，而不引用隐式定义的索引的系统生成的名称。

SET INDEXES 语句是 SET Database Object Mode 语句的一个特例。SET Database Object Mode 语句还可启用或禁用触发器或约束，或可更改约束和唯一索引的过滤模式。

要获取 SET INDEXES 语句的完整语法和语义，请参阅 SET Database Object Mode 语句。

请不要将 SET INDEXES 语句与 SET INDEX 语句混淆。在当前版本中， GBase 8s 数据库服务器不理会 SET INDEX 语句。

**辅助服务器上的限制**

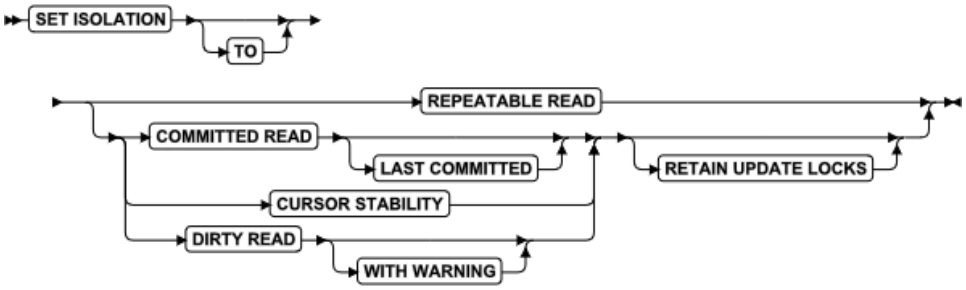
在集群环境中，在可更新的辅助服务器上不支持 SET INDEXES 语句。（一般地说，SET Database Object Mode 语句指定的会话级索引、触发器和约束模式不被重新指向辅助服务器的数据库中的表对象上的 UPDATE 操作。）

## 2.137 SET ISOLATION 语句

使用 SET ISOLATION 语句来定义在尝试同时地访问相同行的进程之中的并发程度。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




**用法**

SET ISOLATION 语句是对 ANSI SQL-92 标准的 GBase 8s 扩展。SET ISOLATION 语句可更改会话的持久的隔离级别。如果您想要通过符合 ANSI 的语句设置隔离级别，则请改为使用 SET TRANSACTION 语句。要获取这两个语句的对比信息，请参阅 SET TRANSACTION 语句。

TO 关键字是可选的，且不起作用。

对于 DIRTY READ（在 SET TRANSACTION 中称为 UNCOMMITTED）、COMMITTED READ 和 REPEATABLE READ（在 SET TRANSACTION 中称为 SERIALIZABLE in SET TRANSACTION）隔离级别，SET ISOLATION 提供与符合 ISO/ANSI 的 SET TRANSACTION 语句相同的功能。

当从数据库检索行时，数据库 *isolation_level* 影响读并发。隔离级别指定在并发 SQL 事务执行期间可发生的现象。可能发生下列现象：

- **Dirty Read**。SQL 事务 T1 修改一行。然后，SQL 事务 T2 在 T1 执行 COMMIT 之前读那行。如果 T1 然后执行 ROLLBACK，则 T2 会读了从未提交的，且可被认为从未存在的一行。
- **Non-Repeatable Read**。SQL 事务 T1 读一行。然后 SQL 事务 T2 修改或删除那行并执行 COMMIT。如果 T1 然后尝试重新读那行，则 T1 可能收到修改了的值或发现那行已被删除了。

- **Phantom Row**。SQL 事务 T1 读满足某搜索条件的多行 N 的集合。然后 SQL 事务 T2 执行 SQL 语句，该语句生成满足 SQL 事务 T1 所使用的搜索条件的一个或多个新行。如果 T1 然后以相同的搜索条件重复原来的读，则 T1 收到不同的行的集合。

数据库服务器使用分享的锁来支持尝试访问数据的进程中不同的隔离级别。

更新或删除进程在正被修改的行上总是要求排他锁。隔离级别不干扰您正在更新或删除的行。如果另一进程尝试更新或删除您正在以 Repeatable Read 隔离级别读取的行，则拒绝那个进程访问那些行。

在 GBase 8s ESQL/C 中，当 SET ISOLATION 执行时打开的那些游标检索行时，可能会或可能不会使用新的隔离级别。从打开了游标直到应用存取行这段时间，设置了的任何隔离级别都可能生效。数据库服务器使用在那个时刻生效的隔离级别，可能已经将行读到内部的缓冲区和内部的临时表之内。要确保一致性和可复现的结果，在您执行 SET ISOLATION 语句之前，请关闭任何打开的游标。

仅在打开数据库之后，您才可从客户端计算机发出 SET ISOLATION 语句。

### 完整连接级别设置

SET ISOLATION 语句支持***完整连接级别***设置。这意味着将在连接时刻本地会话环境中的值传播到所有新的或恢复的事务。这些可包括下列事务类型：

- 本地数据库内的事务，
- 跨同一服务器实例的数据库的分布式事务，
- 跨两个或多个数据库服务器实例的数据库的分布式事务，
- 带有注册在本地数据库中的符合 XA 的数据源的全局事务。

如果您更改事务之内的隔离级别，则将新的值传播回本地环境，也传播到所有随后的新的或恢复的事务。

## GBase 8s 隔离级别

下列定义说明每一隔离级别的关键特征，从最低隔离级别到最高。

# 使用 Dirty Read 隔离级别

不论在它们之上是否有锁，都使用 Dirty Read 选项来从数据库复制行。获取行的程序不放置锁且不予考虑。Dirty Read 是不实现会话日志记录的数据库的唯一可用的隔离级别。

此隔离级别最适合于用于数据未被修改的表上的查询的静态表，因为它不提供隔离。通过 Dirty Read，程序可能返回在随后已回滚了的事务之内插入或修改了的未提交的行，或当您首次读查询集时不可见的***幻像行***，但会在同一事务内随后的读之前在查询集中具体化。（仅 Repeatable Read 隔离级别防止访问幻像行。仅 Dirty Read 提供从并发事务访问未提交的行，这些事务可能随后被回滚。）

当使用 Dirty Read 隔离级别的 DML 操作可能返回未提交的行或幻像行时，可选的 WITH WARNING 关键字指导数据库服务器发出警告。下列示例中的事务使用此隔离级别：

```
BEGIN WORK;
        SET ISOLATION TO DIRTY READ WITH WARNING;
        ...
```

```
COMMIT WORK;
```

Dirty Read 隔离级别对 USELASTCOMMITTED 配置参数和 **USELASTCOMMITTED** 会话环境变量的当前设置非常敏感。要获取更多当将隔离级别设置为 DIRTY READ 或 ALL 时 Dirty Read 隔离级别的行为的信息，请参阅 Committed Read 的 LAST COMMITTED 选项。

当您使用“高可用性数据复制”时，数据库服务器有效地使用“HDR 辅助服务器”上的 Dirty Read 隔离，不理会指定的 SET ISOLATION 或 SET TRANSACTION 隔离级别，除非启用 UPDATABLE_SECONDARY 配置参数。要获取关于此主题的更多信息，请参阅 辅助数据复制服务器的隔离级别。

## 使用 Committed Read 隔离级别

使用 Committed Read 选项来保证在检索行的时刻，每个被检索的行都在表中提交了。此选项不在获取的行上放置锁。Committed Read 是带有不符合 ANSI 的日志记录的数据库中缺省的隔离级别。

当每一行作为独立的单元处理，而不引用同一表中或其他表中的其他行时，Committed Read 是适合的。

### *Committed Read 的 LAST COMMITTED 选项*

使用 Committed Read 隔离级别的 LAST COMMITTED 关键字选项来减小其他会话持有排他的行级锁的风险，该风险会导致应用的锁定错误失败，或直到提交或回滚并发的事务之后应用才能读锁定的行。

在应用尝试读取另一会话在其上持有排他锁的行的上下文中，这些关键字指导数据库服务器返回该行的最近提交的版本，而不是等待该锁被释放。

在下列任何环境之下，此特性隐式地在使用 SET ISOLATION 语句的 Committed Read 隔离级别的，或使用符合 ANSI/ISO 的 SET TRANSACTION 语句的 Read Committed 隔离级别的所有用户会话中生效：

- 如果 USELASTCOMMITTED 配置参数设置为 'COMMITTED READ' 或 'ALL'
- 如果 SET ENVIRONMENT 语句将 USELASTCOMMITTED 会话环境变量设置为 'COMMITTED READ' 或 'ALL'。

在下列任何环境之下，此特性还隐式地在使用 SET ISOLATION 语句的 Dirty Read 隔离级别，或使用符合 ANSI/ISO 的 SET TRANSACTION 隔离级别的的所有用户会话中生效：

- 如果 USELASTCOMMITTED 配置参数设置为 'DIRTY READ' 或 'ALL'
- 如果 SET ENVIRONMENT 语句将 USELASTCOMMITTED 会话环境变量设置为 'DIRTY READ' 或 'ALL'。

启用此特性不可消除锁定冲突的可能性，但它们减少其他会话读取同一行的场景可导致错误的数目。LAST COMMITTED 关键字仅对并发读操作有效。当并发的会话尝试写到同一行时，它们不可防止可发生的锁定冲突或错误。

在表的“最后提交的”版本不可用的上下文中，此特性对 Committed Read 或 Dirty Read 行为无效，包括：

- 数据库不支持事务日志记录
- 以 LOCK MODE PAGE 关键字创建了的表，或已更改为有一 **IFX_DEF_TABLE_LOCKMODE** 环境变量的锁模式设置为了 'PAGE'
- DEF_TABLE_LOCKMODE 配置参数设置为 'PAGE'
- LOCK TABLE 语句已显式地在表上设置了排他锁
- 未提交的 DDL 语句已隐式地在表上设置了排他锁
- 该表是在其上未提交的 DDL 语句已隐式地设置排他锁的系统目录表
- 该表有复合的数据类型的列或用户定义的数据类型的列
- 该表为 RAW 表
- DataBlade 模块正在访问该表
- 使用“虚拟表接口”创建了的表。

不需要用户定义的访问方式来支持 LAST COMMITTED 特性。

LAST COMMITTED 语义的作用域既不是基于语句的，也不是基于事务的。此隔离级别与不带 LAST COMMITTED 选项的 Committed Read 隔离级别有相同的瞬间作用域。例如，当在生效的带有 LAST COMMITTED 的单个事务内执行查询两次时，可能通过相同的查询返回不同的结果，如果在查询的两次提交的间隔之间提交正在相同的数据上操作的其他 DML 事务的话。Committed Read 和 Committed Read Last Committed 语义的此瞬时特性恰好实现 ANSI/ISO Read Committed 隔离级别。

LAST COMMITTED 特性不支持通过表级锁的读取。如果使用 LAST COMMITTED 特性的查询的访问计划遇到它需要访问的表或索引中的表级锁，则该查询会返回下列错误代码：

SQL 错误代码：

```
252: Cannot get system information for table.
```

ISAM 错误代码：

```
113: ISAM error: the file is locked.
```

## 使用 Cursor Stability 隔离级别

使用 Cursor Stability 选项来在获取的行上放置共享锁，当您获取另一行或关闭该游标时，将其释放。另一进程还可在同一行上放置共享锁，但没有进程可获得排他锁来更改该行中的数据。当程序基于它从该行读取的数据来更新另一表时，这样的行稳定性很重要。

如果您将隔离级别设置为 Cursor Stability，但您未正在使用事务，则 Cursor Stability 的作用就像 Committed Read 隔离级别一样。

## 使用 Repeatable Read 隔离级别

使用 Repeatable Read 选项来在会话期间选择的每行上放置共享锁。另一进程也可在被选择的行上放置共享锁，但没有其他进程可在您的事务期间修改任何被选择的行，或在您的事务期间插入满足您的查询的搜索条件的行。如果您在事务期间重复该查询，则您重新读取相同的信息。仅当事务提交或回滚时，才释放共享锁。Repeatable Read 是符合 ANSI 的数据库中缺省的隔离级别。

Repeatable Read 隔离级别放置的锁数目最大，持有锁的时间最长。因此，它是最能减少并发的级别。

## 缺省的隔离级别

当您根据数据库类型创建数据库时，建立特定数据库的缺省的隔离级别。下列列表描述每一数据库类型的缺省的隔离级别。

| 隔离级别 | 数据库类型 |
| --- | --- |
| Dirty Read | 在没有日志记录的数据库中的缺省级别 |
| Committed Read | 在不符合 ANSI 的日志记录的数据库中的缺省级别 |
| Repeatable Read | 在符合 ANSI 的数据库中的缺省级别 |

直到您发出 SET ISOLATION 语句之前，缺省的级别保持有效。在执行 SET ISOLATION 语句之后，直到下列事件发生之前，新的隔离级别保持有效：

- 您输入另一 SET ISOLATION 语句。
- 您打开另一数据库，该数据库的缺省隔离级别不同于您最后的 SET ISOLATION 语句指定的级别。
- 程序结束。

对于不符合 ANSI 的 GBase 8s 数据库，除非您显式地设置 USELASTCOMMITTED 配置参数，否则，对于缺省的隔离级别 LAST COMMITTED 特性无效。SET ENVIRONMENT 语句或 SET ISOLATION 语句可覆盖此缺省值，并为当前的会话启用 LAST COMMITTED。

## 使用 RETAIN UPDATE LOCKS 选项

当数据库服务器处理 SELECT ... FOR UPDATE 语句时，使用 RETAIN® UPDATE LOCKS 选项来影响它的行为。

在隔离级别设置为 Dirty Read、Committed Read 或 Cursor Stability 的数据库中，数据库服务器在 SELECT ... FOR UPDATE 语句获取的行上放置更新锁。当您开启 RETAIN UPDATE LOCKS 选项时，数据库服务器保持更新锁，直到事务结束为止，而不是在下一随后的 FETCH 时或当关闭游标时才释放它。此选项防止在当前的用户到达事务的结束之前其他用户在更新了的行上放置排他锁。

您可使用此选项来获得相同的锁定效果，但避免 dummy 更新或可重复读隔离级别的开销。

在当前的会话期间的任何时刻，您都可开启或关闭此选项。

您可通过重置隔离级别而不使用 RETAIN UPDATE LOCKS 关键字来关闭该选项，如下例中所示。

```
BEGIN WORK;
        SET ISOLATION TO
        COMMITTED READ LAST COMMITTED RETAIN UPDATE LOCKS;
        ...
        COMMIT WORK;
        BEGIN WORK;
        SET ISOLATION TO COMMITTED READ LAST COMMITTED ;
        ...
        COMMIT WORK;
```

**通过会话环境控制更新锁**

禁用 RETAIN UPDATE LOCKS 行为的另一种方法是执行此 SQL 语句：

```
SET ENVIRONMENT RETAINUPDATELOCKS 'NONE';
```

通过重置 RETAINUPDATELOCKS 会话环境变量，这为当前的事务，或为同一会话的任意随后的事务禁用 RETAIN UPDATE LOCKS 子句。

SET ENVIRONMENT RETAINUPDATELOCKS 语句还可使得更新锁的保持成为 Committed Read、Cursor Stability 或 Dirty Read 隔离级别，或对于所有这些隔离级别的缺省行为，不论 SET ISOLATION 语句是否包括 RETAIN UPDATE LOCKS 子句。

要获取更多关于更新锁的信息，请参阅 RETAINUPDATELOCKS 环境选项 和 锁定注意事项。

### *在事务期间关闭选项*

在事务已经开始之后，但在提交或回滚事务之前，如果您将 RETAIN® UPDATE LOCKS 选项设置为 OFF，则可能仍存在几个更新锁。

切换到 OFF，该特性不直接地释放任何更新锁。当您关闭此选项时，数据库服务器恢复到三个隔离级别的正常行为。也就是说，通过紧接在前面的 FETCH 语句，FETCH 语句释放放置在行上的更新锁，且关闭了的游标释放在当前行上的更新锁。

不释放稍早的 FETCH 语句放置的更新锁，除非在同一事务之内出现多个更新游标。在此情况下，随后的 FETCH 还可能释放其他游标的较旧的更新锁。

## 隔离级别的影响

您不可在没有日志记录的数据库中设置事务隔离级别。在这样的数据库中发生的每次检索都作为 Dirty Read。

从 BYTE 或 TEXT 列检索的数据可有所不同，这依赖于事务隔离级别。在 Dirty Read 或 Committed Read 隔离级别之下，进程可读取或被删除的（如果该删除尚未提交的话）或在正被删除的进程之中的 BYTE 或 TEXT 列。在这些隔离级别之下，在某些情况下，被删除的数据是可读的。要获取更过关于这些情况的信息，请参阅 *GBase 8s 管理员指南*。

当您使用 DB-Access 时，由于您使用更高的隔离级别，所以锁冲突发生得更频繁。例如，如果您使用 Cursor Stability，则与您使用 Committed Read 相比，会发生更多的锁冲突。

在 GBase 8s ESQL/C 事务中使用滚动游标，或通过将级别设置为 Repeatable Read，或通过在事务期间锁定整个表，您可在您的临时表与数据库表之间强制一致性。

如果您在事务中使用滚动游标 WITH HOLD，则您不可在您的临时表与数据库表之间强制一致性。当事务完成时，释放表级锁或通过 Repeatable Read 设置的锁，但处于保持状态的滚动游标在事务结束之外仍保持打开。事务一结束，您就可修改释放了的行，但在临时表中的被检索的数据可能与实际数据不一致。

注意： 请不要在事务内使用无日志记录的表。如果您需要在事务内使用无日志记录的表，则或者将隔离级别设置为 Repeatable Read，或者以 Exclusive 模式锁定该表来防止并发问题。

### 辅助数据复制服务器的隔离级别

如果禁用 UPDATABLE_SECONDARY 配置参数（通过未设置或通过设置为零），则辅助数据复制服务器为只读。在此情况下，在“高可用性数据复制”（HDR）和远程独立辅助（RSS）服务器上仅 Dirty Read 或 Read Uncommitted 事务隔离级别是可用的。

如果启用 UPDATABLE_SECONDARY 参数（通过设置为一个大于零的有效的连接数），则辅助数据复制服务器可支持 Read Committed、Committed Read 或 Committed Read Last Committed 事务隔离级别，带有或不带 SET ENVIRONMENT 语句的 USELASTCOMMITTED 会话环境变量。仅 SQL 的 DELETE、INSERT、UPDATE 和 MERGE 语句（以及 **dbexport** 实用程序，如果设置 STOP_APPLY、USELASTCOMMITTED 和 UPDATABLE_SECONDARY 配置参数的话）可支持在可更新的辅助服务器上的写操作。

然而，“共享磁盘辅助”（SDS）服务器可支持 Read Committed、Committed Read、Committed Read Last Committed 隔离级别，不管他们的 UPDATABLE_SECONDARY 设置。要获取更多关于 UPDATABLE_SECONDARY 配置参数的信息，请参阅 GBase 8s “管理员参考手册”。

## 2.138 SET LOCK MODE 语句

使用 SET LOCK MODE 语句来定义数据库服务器如何处理一个试图访问锁定了的行或表的进程。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *seconds* | 在发出错误之前，进程等待释放锁的最大秒数 | 仅当比系统缺省值更短时，才 | 精确数值 |

| | | | |
|---|---|---|---|
| | | 是有效的 | |

**用法**

当进程试图访问锁定了的行或表时，此语句可以下列方式定向数据库服务器的响应。

| 锁定方式 | 作用 |
|---|---|
| NOT WAIT | 数据库服务器立即结束操作并返回错误代码。这是缺省的情况。 |
| WAIT | 数据库服务器挂起进程，直到释放锁为止。 |
| WAIT ***seconds*** | 数据库服务器挂起进程，直到释放锁或直到等待期间结束为止。如果在等待期间之后该锁定保持，则操作结束并返回错误代码。 |

要获取本文档中术语***锁模式***的两种不同含意的描述，请参阅相关的概念部分中的“锁定颗粒度”。

要避免在尝试读取并发会话对其持有排他行级锁的操作中等待，您还可使用 LAST COMMITTED 特性，或者通过在 SET ISOLATION COMMITTED READ 语句中显式地设置它，或者通过设置 USELASTCOMMITTED 配置参数或 USELASTCOMMITTED 会话环境选项。

**示例**

在下例中，用户指定如果进程请求锁定了的行，则操作应立即结束并应返回错误代码：

```
SET LOCK MODE TO NOT WAIT;
```

在下例中，用户指定应挂起进程直到释放该锁为止：

```
SET LOCK MODE TO WAIT;
```

下一示例在任何等待的长度上设置上限 17 秒：

```
SET LOCK MODE TO WAIT 17;
```

## WAIT 子句

WAIT 子句导致数据库服务器挂起进程，直到释放锁或直到未释放锁但已超过了指定的秒数为止。

当您请求 WAIT 选项时，数据库服务器为防止死锁的可能性提供保护。在数据库服务器挂起进程之前，它检查挂起进程是否会导致死锁。如果数据库服务器发现可能发生死锁，则终止该操作（否决您的等待指令）并返回错误代码。在疑似死锁或实际死锁的情况下，数据库服务器都返回错误。

请谨慎地使用无限制的等待期间，当您指定不带 ***seconds*** 的 WAIT 选项时会产生这种情况。如果您不指定上限，则放置了锁的进程会以某种方式不释放它，被挂起的进程可能无限地等待。由于不存在真的死锁状况，所以数据库服务器不采取纠正活动。

在网络环境中，DBA 使用 ONCONFIG 参数 DEADLOCK_TIMEOUT 来建立 ***seconds*** 的缺省值。如果您使用 SET LOCK MODE 语句来设置上限，则仅当您的等待期间比系统缺省值更短时，才应用您的值。

## 完整连接级别设置

SET LOCK MODE 语句支持***完整连接级别***设置。这意味着将在连接时刻本地会话环境中的值传播到所有新的或恢复的事务。这些可包括下列事务类型：

- 本地数据库内的事务，
- 跨同一服务器实例的数据库的分布式事务，
- 跨两个或多个数据库服务器实例的数据库的分布式事务，

符合 XA 的注册在本地数据库中的数据源的全局事务。

如果您在事务内更改锁模式设置，则将新的值传播回本地环境以及所有随后的新的或恢复的事务。

# 2.139 SET LOG 语句

使用 SET LOG 语句来将您的数据库日志记录模式从缓冲的事务日志记录更改为未缓冲的事务日志记录，反之亦然。

此语句是对 SQL 的 ANSI/ISO 标准的扩展。与大多数扩展不一样，SET LOG 语句在符合 ANSI 的数据库中是无效的。

**语法**


用法

当您创建数据库或向现有的数据库添加日志记录时，您激活事务的日志记录。这些事务日志可为缓冲的或未缓冲的。

缓冲的日志记录是在内存缓冲区中保持事务直到该缓冲区满为止的一类日志记录，不管提交或回滚事务的时间。数据库服务器提供此选项来通过减少磁盘写的次数提高操作速度。

Attention： 以缓冲的日志记录您只可获得微小的效率提升，但会招致一些风险。在系统故障的时候，数据库服务器不可恢复内存缓冲区中任何已完成的事务，这些尚未写到磁盘。

下列示例中的 SET LOG 语句将事务日志记录模式更改为缓冲的日志记录：

```
SET BUFFERED LOG;
```

未缓冲的日志记录是不在内存缓冲区中保持事务的日志记录。事务一结束，数据库服务器就将事务写到磁盘。当您正在使用未缓冲的日志记录时，如果发生系统故障，则您恢复所有已完成的事务，但不包括仍在缓冲区中的那些。事务日志的缺省条件是未缓冲的日志记录。

下列示例中的 SET LOG 语句将事务日志记录模式更改为未缓冲的日志记录：

```
SET LOG;
```

SET LOG 语句仅定义当前会话的模式。数据库管理员以 **gdblogmode** 实用程序设置的缺省模式保持不变。

缓冲的选项不影响从外部表的检索。对于分布式查询，带有日志记录的数据库仅可从带有日志记录的数据库检索，但与数据库使用缓冲的还是非缓冲的日志记录无关。

符合 ANSI 的数据库不可使用缓冲的日志记录。

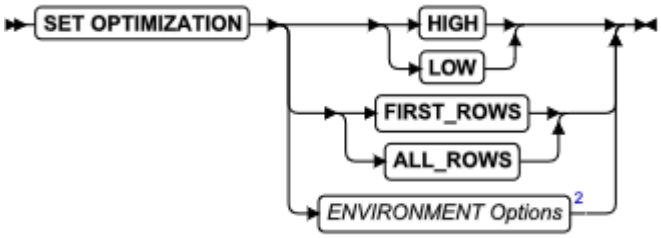
您不可更改符合 ANSI 的数据库的日志记录模式。如果您以 WITH LOG MODE ANSI 关键字创建了数据库，则在后期您不可使用 SET LOG 语句将日志记录模式更改为缓冲的或未缓冲的事务日志记录。

## 2.140 SET OPTIMIZATION 语句

使用 SET OPTIMIZATION 语句来指定查询执行优化器花费多长时间来制定查询计划或指定优化目标。SET OPTIMIZATION 语句是对 SQL 的 ANSI/ISO 标准的扩展。

当您随同 GBase 8s 使用 **DB-Access** 时，SET OPTIMIZATION 语句的 ENVIRONMENT 选项可为当前会话中的所有语句定义通用的优化环境。

**语法**




用法

您可在任何时刻执行 SET OPTIMIZATION 语句。在当前的数据库服务器上跨数据库地支持指定的优化级别。您指定的选项保持有效，直到您发出另一 SET OPTIMIZATION 语句或直到程序结束为止。对于查询优化器为确定查询计划而花费的时间量，缺省的数据库服务器优化级别为 HIGH。

在 GBase 8s 上，缺省的优化目标为 ALL_ROWS。虽然在某一时刻您仅可设置一个选项，但您可发出两个 SET OPTIMIZATION 语句：一个指定优化器为确定查询计划所花费的时间，一个指定查询的优化目标。

类似地，您可发出包括 ENVIRONMENT 选项的多个 SET OPTIMIZATION 语句来指定用于优化查询的会话环境。在数据仓库应用中，适当的优化器环境可提升在星型模式中表的连接查询的性能。保持优化器环境设置，直到另一 SET OPTIMIZATION ENVIRONMENT 语句覆盖它们为止，或直到会话结束为止。要获取更多信息，请参阅 ENVIRONMENT 选项 主题。

**示例**

下列示例展示跨网络的优化。`central` 数据库（在 `midstate` 数据库服务器上）将有 LOW 优化；`western` 数据库（在 `rockies` 数据库服务器上）将有 HIGH 优化。

```
CONNECT TO 'central@midstate';
        SET OPTIMIZATION LOW;
        SELECT * FROM customer;
        CLOSE DATABASE;
        CONNECT TO 'western@rockies';
        SET OPTIMIZATION HIGH;
        SELECT * FROM customer;
        CLOSE DATABASE;
        CONNECT TO 'wyoming@rockies';
        SELECT * FROM customer;
```

在此，**wyoming** 数据库将有 HIGH 优化，因为它驻留在与 **western** 数据库相同的数据库服务器上。该代码不需要为 **wyoming** 数据库重新指定优化级别，因为 **wyoming** 数据库像 **western** 数据库一样，驻留在 **rockies** 数据库服务器上。

下列示例指导 GBase 8s 优化器使用大部分时间来确定查询计划，然后尽可能返回结果的前几行：

```
SET OPTIMIZATION LOW;
        SET OPTIMIZATION FIRST_ROWS;
        SELECT lname, fname, bonus
        FROM sales_emp, sales
        WHERE sales.empid = sales_emp.empid AND bonus > 5,000
        ORDER BY bonus DESC;
```

## HIGH 和 LOW 选项

HIGH 和 LOW 选项确定查询优化器花费多长时间来确定查询计划：

- HIGH

  此选项指导优化器使用复杂的基于成本的算法，测试所有合理的查询计划选择并选择总体上最佳的选项。

  对于大型连接，此算法可能招致超出预期的代价。在极端情况下，您可用尽内存。

- LOW

  此选项指导优化器使用不很复杂的但更快速地设计优化算法，在每一阶段基于最低成本路径。此算法在优化的早期阶段期间消除不大可能的连接策略，并减少优化期间的时间和资源消耗。

  当您指定优化的 LOW 级别时，数据库服务器可能不选择最佳的策略，因为在该算法的早期阶段期间就消除了那种策略。

## FIRST_ROWS 和 ALL_ROWS 选项

FIRST_ROWS 与 ALL_ROWS 关键字选项标识两个不同的查询优化目标：

- FIRST_ROWS

  此选项指导优化器尽快选择返回第一个符合条件的记录的查询计划，而忽略可能对记录排序或创建哈希表的计划。

- ALL_ROWS

  此选项指导优化器尽快选择返回所有符合条件的记录的查询计划。

**Inline 优化器伪指令**

不选择这些 SET OPTIMIZATION 语句选项中的一种，您可以改为为单个查询指定优化-目标伪指令作为紧跟在开启该查询或子查询的 SELECT 关键字之后的注解。对于 DELETE、SELECT 或 UPDATE 语句中的查询，要获取更多关于 inline 优化器伪指令的语法和选项的信息，请参阅 优化程序伪指令。

**外部的优化器伪指令**

除了 SET OPTIMIZATION 语句可为当前会话中的查询指定的查询优化器伪指令，或可跟在 SELECT 关键字之后的 inline 优化器伪指令之外，数据库服务器还支持通过查询优化器影响执行路径的选择的第三种格式。

在 sysdirectives 系统目录表中，DBA 或用户 **gbasedbt** 可执行 SAVE EXTERNAL DIRECTIVES 语句来注册*外部的优化器伪指令*，也称为*外部的伪指令*。如果已启用了对外部伪指令的支持，则 GBase 8s 数据库服务器自动地将这些伪指令应用到与指定的 SELECT 语句相匹配的查询。SAVE EXTERNAL DIRECTIVES 语句的 ACTIVE、INACTIVE 或 TEST ONLY 关键字选项分别启用、禁用或限制外部的伪指令的作用域。

在 ONCONFIG 文件中设置 EXT_DIRECTIVES 配置参数为 1 或 2，且设置 IFX_EXTDIRECTIVES 客户端侧环境变量为 1，启用对外部的伪指令的支持。

独立于 EXT_DIRECTIVES 配置参数或 IFX_EXTDIRECTIVES 客户端侧设置，为会话环境设置 SET ENVIRONMENT EXTDIRECTIVES 为 '1'、on 或 ON，在当前的用户会话中启用外部的优化器伪指令，如果在 sysdirectives 表中注册任何活动的外部伪指令的话。

SET EXPLAIN 输出文件显示外部的伪指令对查询是否生效。

要获取更多关于外部的优化器伪指令的信息，请参阅 为会话启用或禁用外部伪指令。

## 优化 SPL 例程

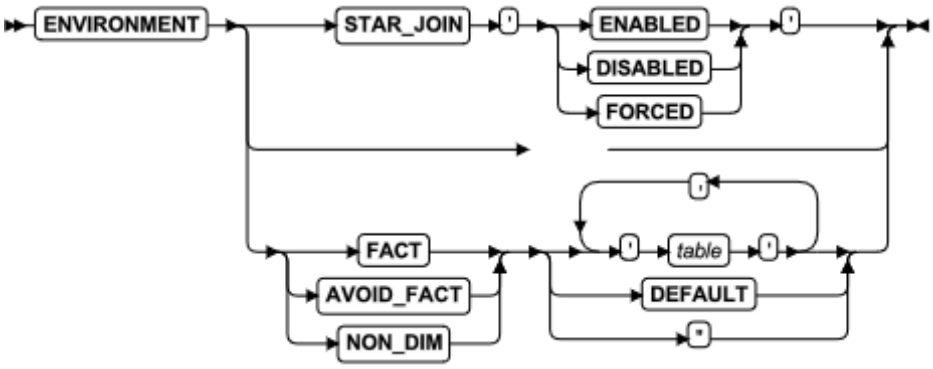
对于保持不变或仅有轻微改变的 SPL 例程，当您创建该 SPL 例程时，您可能想要设置 SET OPTIMIZATION 语句为 HIGH。此步骤为 SPL 例程存储最佳查询计划。然后，在您执行 SPL 例程之前执行 SET OPTIMIZATION LOW 语句。然后，SPL 例程使用最佳的查询计划并以更高的成本-效率比运行。

## ENVIRONMENT 选项

使用 SET OPTIMIZATION 语句的 ENVIRONMENT 选项子句来为当前会话中的所有查询定义通用的优化环境。对于有些数据仓库应用，在每一维度表的主键对应于事实表的外键的数据库中，您在此子句中指定的会话环境设置可提升将事实表与维度表连接的查询的性能。

GBase 8s 的 **DB-Access** 实用程序支持 SET OPTIMIZATION 语句的 ENVIRONMENT 选项子句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *table* | 表、视图或同义词 | 在数据库中必须存在 | 标识符 |

**用法**

ENVIRONMENT 选项子句可指定当前会话的优化环境的属性。保持这些属性直到会话结束为止，或直到另一 SET OPTIMIZATION ENVIRONMENT 语句重置优化属性为止。

下表描述每一星型连接伪指令并指明它如何影响优化器的查询计划。

| 关键字 | 作用 | 优化器行动 |
|---|---|---|
| STAR_JOIN | 'ENABLED'开启（与 'DISABLED' 关闭）对当前会话的星型连接支持。当可能的时候，对于所有查询，'FORCED' 设置偏爱星型执行路径。 | 对于 'ENABLED'，优化器考虑星型连接执行计划的可能性。对于 'FORCED'，如果可用的话，会选择星型计划。对于 'DISABLED'，不考虑星型连接。 |
| FACT | 标识在星型模式中对应于事实表的表。如果 AVOID_FACT 表也列为 FACT，则 FACT 优先。DEFAULT（或空字符串）为会话关闭此环境设置。 | 仅将在 FACT 列表中的表考虑作为星型连接优化中的事实表。可罗列多个表作为 FACT。 |
| AVOID_FACT | 请不要使用该表（或在表的列表中的任何表）作为星型连接优化中的事实表。DEFAULT（或空字符串）为会话关闭此环境设置。 | 将 AVOID_FACT 列表中的表考虑作为星型连接优化中的事实表。可罗列多个表作为 AVOID_FACT。 |
| NON_DIM | 标识在星型模式中不对应于维度表的表。DEFAULT（或空字符串）为会话关闭此环境设置。 | 将在 NON_DIM 列表中的表不考虑作为星型连接优化中的维度表。可罗列多个表作为 NON_DIM。 |

跟在 STAR_JOIN 伪指令之后的任何 'ENABLED'、'DISABLED' 或 'FORCED' 关键字，或以逗号分隔的一个或多个指定优化器环境属性的设置的 ***table*** 标识符，必须通过单引号(')或双引号(")定界。如果以逗号分隔的多个 ***table*** 标识符的列表跟在 FACT、AVOID_FACT 或 NON_DIM 关键字之后，则请不要在列表中的任何项之间包括空格。

当用户连接到数据库时，DBA 可使用 sysdbopen( ) 例程来定义生效的优化环境。例如，要指定总是偏爱星型连接执行计划的优化器环境，sysdbopen( ) 例程应包括这些 SQL 语句：

```
SET OPTIMIZATION ENVIRONMENT STAR_JOIN 'FORCED';
SET OPTIMIZATION ENVIRONMENT FACT 'table1,table2, ... tableN';
```

在此，其名称罗列在 FACT 关键字之后的表都应是事实表。

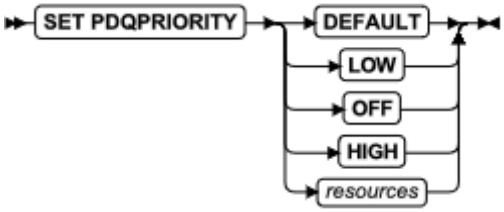
要获取更多关于可偏爱或避免星型连接执行计划的查询优化器伪指令的信息，请参阅 星型连接伪指令。

要获取更多关于如何使用内建的 **sysdbopen( )** 例程来为指定的用户、为 PUBLIC 组或为角色，在连接时刻定义会话环境的信息，请参阅 会话配置过程。

## 2.141 SET PDQPRIORITY 语句

SET PDQPRIORITY 语句启用一应用来在例程之内动态地设置查询优先级别。SET PDQPRIORITY 语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *resources* | 指定查询优先级别以及处理该查询的资源的百分率的整数 | 可取值从 -1 到 100。另请参阅 分配数据库服务器资源。 | 精确数值 |

**用法**

SET PDQPRIORITY 语句覆盖 PDQPRIORITY 环境变量（但优先权低于 MAX_PDQPRIORITY 配置参数）。SET PDQPRIORITY 对例程的作用域是本地，且不影响同一会话内的其他例程。当发出此语句的例程终止时，该设置恢复为系统缺省值。

将 PDQ 优先级设置为小于 100 除以准备好的语句的最大数目之商的值。例如，如果两个准备好的语句是活动的，则您应设置 PDQ 优先级小于 50。

例如，假设 DBA 设置 MAX_PDQPRIORITY 参数为 50。那么用户输入下列 SET PDQPRIORITY 语句来将查询优先级别设置为资源的 80%：

SET PDQPRIORITY 80;

当它处理该查询时，数据库服务器使用 MAX_PDQPRIORITY 值来将用户设置的查询优先级别作为因子。数据库服务器静静地以优先级别 40 处理该查询。此优先级别表示用户指定的资源的 80% 的 50%。

SET PDQPRIORITY 语句支持下列关键字。

| 关键字 | 作用 |
| --- | --- |
| DEFAULT | 使用 PDQPRIORITY 环境变量的设置 |
| LOW | 并行地从分片的表获取数据值。（在 GBase 8s 中，当您指定 LOW 时，数据库服务器不适用于并行机制的其他形式。） |
| OFF | 关闭 PDQ（仅限于 GBase 8s ）。数据库服务器不适用于并行机制。如果您既不使用 PDQPRIORITY 环境变量，也不使用 SET PDQPRIORITY 语句，则 OFF 为缺省值。 |
| HIGH | 基于包括可用的处理器数目、正在查询的表的分片、查询的复杂度以及其他因素，数据库服务器确定一恰当的 PDQPRIORITY 值。当在未来的版本中指定 HIGH 时，GBase 保留更改查询的性能行为的权利。 |

**对于分配 PDQ 优先级的方法的优先级**

对于受 PDQPRIORITY 影响的操作中的可用内存，数据库服务器可分配的最大内存量受您的系统可用物理内存的限制，也受这些参数设置（按升序排列）的限制：

- PDQPRIORITY 环境变量
- 最近的 SQL 的 SET PDQPRIORITY 语句
- MAX_PDQPRIORITY 配置参数
- DS_TOTAL_MEMORY 配置参数
- BOUND_IMPL_PDQ 会话环境变量

当并发查询正在运行时，DS_MAX_QUERIES 配置参数设置还可限制新的查询的可用 PDQ 内存量。

## 分配数据库服务器资源

您可指定取值范围从 -1 到 100 的整数来表明查询优先级别作为处理该查询的数据库服务器资源的百分率。资源包括内存量和处理器的数目。您指定的数目越大，数据库服务器使用的资源就越多。

使用更多的资源通常表明给定的查询的更好的性能。然而，使用过多的资源可导致对资源的争夺，以及从其他查询移除资源，从而降低性能结果。随同 ***resources*** 选项，下列值与标识查询优先级别的关键字等值。

| 值 | 同等的关键字优先级别 |
|---|---|
| -1 | DEFAULT |
| 0 | OFF |
| 1 | LOW |

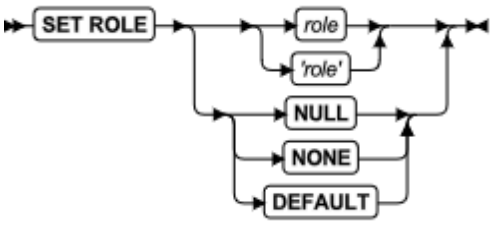
对于 GBase 8s ，下列语句是等同的。第一个语句使用关键字 LOW 来建立低查询优先级别。第二个在 *resources* 参数中使用值 1 来建立低查询优先级别。

```
SET PDQPRIORITY LOW;
SET PDQPRIORITY 1;
```

## 2.142 SET ROLE 语句

使用 SET ROLE 语句来启用用户定义的角色的权限。此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *role* | 要启用的角色的名称 | 必须在数据库中已存在，且必须已被授权给用户，但不可为内建的角色。如果括在引号之间，则 *role* 区分大小写。 | 所有者名称; |

**用法**

被授予角色的任何用户都可通过使用 SET ROLE 语句来启用该角色。您一次仅可启用一个角色。如果您在已设置角色之后执行 SET ROLE 语句，则新的角色取代旧的角色作为当前的角色。

如果用户当前不持有该角色，或如果该角色为内建的角色，则 SET ROLE 语句返回错误。（由内建的角色持有的访问权限，诸如 EXTEND 角色或 DBSECADM 角色，总是生效，且如果用户持有那个角色，则不要求通过 SET ROLE 语句激活。）

当 DBA 发出 GRANT DEFAULT ROLE 语句时，用户可被授予一个数据库实例的缺省角色。如果对于当前数据库中的角色不存在缺省的角色，则缺省地分配角色 NULL 或 NONE。在此上下文中，NULL 与 NONE 是同义词。角色 NULL 和 NONE 可没有权限。要将您的角色设置为 NULL 或 NONE，会禁用您的当前角色。

当请您使用 SET ROLE 来启用角色时，您获得该角色的权限，除了 PUBLIC 以及您自己的权限之外。如果将一角色授予已分配给您的另一角色，则您获得两个角色的权限，除了 PUBLIC 的任何权限和您自己的权限之外。

在 SET ROLE 成功地执行之后，指定的角色保持有效，直到关闭当前数据库或用户执行另一 SET ROLE 语句为止。然而，仅用户，不是角色，保持在会话期间创建了的任何数据库对象的拥有权，比如表。

仅在当前数据库之内，角色才在作用域中。您不可使用您从角色获得的权限来访问另一数据库中的数据。例如，如果您有来自名为 **acctg** 的数据库中的角色的权限，且您在名为 **acctg** 和 **inventory** 的数据库之上执行分布式查询，则您的查询不可访问 **inventory** 数据库中的数据，除非您还被授予了 **inventory** 数据库中的适当的权限。作为安全预防措施，用户仅从角色持有的自主访问权限不可通过视图或通过触发器的活动来提供对当前数据库外部的表的访问。

如果您的数据库支持显式的事务，您必须在事务的外部发出 SET ROLE 语句。如果您的数据库符合 ANSI，则 SET ROLE 必须是新的事务的第一个语句。如果在事务是活动的时候执行 SET ROLE 语句，则发生错误。要获取更多关于初始隐式的事务的 SQL 语句的信息，请参阅 SET SESSION AUTHORIZATION 和事务。

如果执行 SET ROLE 语句作为触发器或 SPL 例程的一部分，且随同 WITH GRANT OPTION 将该角色授予了触发器或 SPL 例程的所有者，则启用该角色，即使未授予您该角色。例如，此代码片段设置角色，然后在查询之后放弃它：

```
EXEC SQL set role engineer;
        EXEC SQL select fname, lname, project
        INTO :efname, :elname, :eproject FROM projects
        WHERE project_num > 100 AND lname = 'Larkin';
        printf ("%s is working on %s\n", efname, eproject);
        EXEC SQL set role NULL;
```

## 设置缺省的角色

DBA 或数据库的所有者可发出 GRANT DEFAULT ROLE 语句来对指定的用户列表或对 PUBLIC 指定一现有的角色为***缺省的角色***。不像非缺省的角色那样，缺省的角色不要求 SET ROLE 语句来启用它。当为用户分配缺省的角色时，将一对数据库的隐式的连接授予该用户。

在下一示例中的三个语句中的每一个分别对角色执行下列操作之一：

- 声明名为 **Engineer** 的角色
- 将对 **locomotives** 表的 Select 权限分配给 **Engineer** 角色
- 定义 **Engineer** 作为用户 **jgould** 的缺省的角色。

```
EXEC SQL CREATE ROLE 'Engineer';
EXEC SQL GRANT SELECT ON locomotives TO 'Engineer';
EXEC SQL GRANT DEFAULT ROLE 'Engineer' TO jgould;
```

如果 jgould 随后使用 SET ROLE 语句来启用一些其他角色，则通过执行下列语句，jgould 以 Engineer 替代那个角色作为缺省的角色：

```
SET ROLE DEFAULT;
```

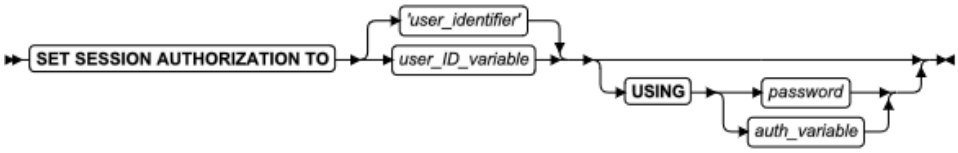
如果您没有缺省的角色，则 SET ROLE DEFAULT 使得 NONE 成为您的缺省角色，仅留下已经显式地授予了您的 ***username*** 或 PUBLIC 的那些权限。在 GRANT DEFAULT ROLE 将您的缺省角色更改为新的缺省角色之后，执行 SET ROLE DEFAULT 恢复您最近被授予的缺省角色，即使当您连接到了数据库时此角色不是您的缺省角色。

如果未授予 PUBLIC 一个缺省角色，但授予不同的角色作为一单个用户的缺省角色，则单独授予的缺省角色优先，如果那个用户发出 SET ROLE DEFAULT 或连接到数据库的话。

## 2.143 SET SESSION AUTHORIZATION 语句

SET SESSION AUTHORIZATION 语句让您在当前的会话中执行的数据库操作之下更改用户名称。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *auth_variable* | 对于在 *user_identifier* 或 *user_ID_variable* 中指定的登录名称持有有效的口令的主变量 | 变量必须为定长的字符数据类型。它的值与 *password* 有相同的限制。 | 对于变量名称必须符合特定于语言的规则。 |
| *password* | 指定用户的口令的用引号括起来的字符串。 | 必须为那个用户的口令，且不多于 32 字节 | 引用字符串 |
| *user_identifier* | 用引号括起来的应用的有效的登录名称。引号定界符保持字母大小写。 | 不超过 32 字节的授权标识符 | 引用字符串 |
| *user_ID_variable* | 持有用户标识符的值的 ESQL/C 主变量的名称。 | 变量必须为定长字符数据类型。它的值与 *user_identifier* 有相同的限制。 | 必须符合变量名称的特定于语言的规则。 |

## 用法

此语句允许您采用另一用户的身份，包括自主访问控制（DAC）和基于标签的访问控制（LBAC）凭证。您还可在支持 GBase 8s 受信的上下文的 API 中使用此语句，来切换在受信的连接上的用户 ID。

同时需要 DBA 和 SETSESSIONAUTH 访问权限来执行此语句。除非当您启动您已持有对 PUBLIC（或对于您在 SET SESSION AUTHORIZATION 语句中指定其名称的用户）的 SETSESSIONAUTH 权限的会话，且您还持有 DBA 权限时，此语句失败并报错。

如果数据库服务器已经从不支持基于标签的访问控制的旧版本转换了，则在迁移进程中，自动地授予持有 DBA 权限的用户对于 PUBLIC 的 SETSESSIONAUTH 访问权限。如果数据库服务器已经被初始化作为支持 LBAC 安全策略的版本，则持有 **DBSECADM** 角色的用户可将 SETSESSIONAUTH 权限授予其他用户。由于每一用户的安全凭证决定在受保护的表中可访问那些数据行，因此 **DBSECADM** 在授予 SETSESSIONAUTH 权限和指定它的作用域时应小心行事。

新的身份在当前数据库中保持有效，直到您再次执行 SET SESSION AUTHORIZATION 为止，或直到您关闭当前数据库为止。当您使用此语句时，指定的 *user* 必须有对当前数据库的 Connect 权限。此外，DBA 不可将新的授权标识符设置到 PUBLIC 组，也不可设置到当前数据库中现有的角色。

将会话设置到另一用户会导致在当前活动的数据库服务器中用户名称的更改。就此数据库服务器进程而言，在指定的 *user* 通过一些管理实用程序访问数据库服务器时，完全失去任何权限。此外，新的会话 *user* 不能以获得的身份开启器任何管理操作（例如，执行实用程序）。

在 SET SESSION AUTHORIZATION 语句成功地执行之后，放弃通过先前的用户启用的任何角色。如果您希望采用已被授予指定的 *user* 的角色，则您必须使用 SET ROLE 语句。数据库服务器不自动地启用 *user* 的缺省角色。

在 SET SESSION AUTHORIZATION 成功地执行之后，在数据库服务器在 RESTRICTED 模式下使用新的授权标识符时，放入任何 DBA 创建了的所有者权限的 UDR，这可影响在远程数据库中对象之上 的 UDR 的操作期间的访问权限。要获取更多关于 RESTRICTED 模式的信息，请参阅在 *《GBase 8s SQL 指南：参考》* 中的 **sysprocedures** 系统目录表。

当您通过执行 SET SESSION AUTHORIZATION 语句来采用另一用户的身份时，您仅可在当前数据库中执行操作。您不可在当前数据库之外的数据库对象上执行操作，诸如远程表。此外，您不可执行 DROP DATABASE 或 RENAME DATABASE 语句，即使真实的用户或实际的用户拥有该数据库。

您可使用此语句或者来直接获取对数据的访问，或者来授予数据库操作处理所需的数据库级权限或表级权限。下列示例展示如何使用 SET SESSION AUTHORIZATION 语句来获取表级权限：

```
SET SESSION AUTHORIZATION TO 'cathl';
        GRANT ALL ON customer TO 'mary';
SET SESSION AUTHORIZATION TO 'mary';
        UPDATE customer SET fname = 'Carl' WHERE lname = 'Pauli';
```

如果您用引号括起 *user*，则该名称区分大小写，且完全按输入形式存储该名称。在符合 ANSI 的数据库中，如果您不使用引号作为定界符，则以大写字母存储授权标识符，除非设置 **ANSIOWNER** 环境变量来防止从小写字母转化为大写。

下列“开放数据库连接”（ODBC）API 示例在与授权要求的受信的连接上启用用户 ID 切换：

```
SQLExecDirect(hstmt,"SET SESSION AUTHORIZATION TO 'zurbie' USING
'pass01'",SQL_NTS);
```

在上述函数调用中，

- `'zurbie'` 为在此会话中的随后的操作指定授权标识符
- `pass01` 必须为用户 **zurbie** 的当前口令。

注：

除了在非敌对的环境中，`'pass01'` 不是登录口令的推荐的示例，因为在某些语言环境中，它容易被猜出。

### SET SESSION AUTHORIZATION 和事务

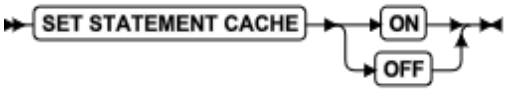
如果您的数据库不符合 ANSI，则您必须在事务之外发出 SET SESSION AUTHORIZATION 语句。如果您在事务之内发出该语句，则会收到错误消息。

在符合 ANSI 的数据库中，仅当尚未执行开启隐式的事务的语句（例如，CREATE TABLE 或 SELECT）时，您可执行 SET SESSION AUTHORIZATION 语句。不开启隐式的事务的语句是那些不要求锁或日志数据的语句（例如，SET EXPLAIN 和 SET ISOLATION）。在 DATABASE 语句或 COMMIT WORK 语句之后，您可立即执行 SET SESSION AUTHORIZATION 语句。

## 2.144 SET STATEMENT CACHE 语句

使用 SET STATEMENT CACHE 语句来为当前的会话开启高速缓存或关闭高速缓存。此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




**用法**

您可使用 SET STATEMENT CACHE 语句来为当前的会话在 SQL 语句高速缓存中开启或关闭高速缓存。该语句高速缓存存储在会话中反复地运行的缓冲区相同的语句中。仅数据操纵语言（DML）语句（DELETE、INSERT、UPDATE 或 SELECT）可被存储在语句高速缓存中。

此机制允许符合条件的语句绕过优化阶段，并避免重新编译，这可降低内存消耗，并可改善查询处理时间。

**示例**

下列示例为当前会话开启语句高速缓存：

```
SET STATEMENT CACHE ON;
```

下例为当前会话关闭语句高速缓存：

```
SET STATEMENT CACHE OFF;
```

### 优先级和缺省的行为

SET STATEMENT CACHE 优先于 **STMT_CACHE** 环境变量和 STMT_CACHE 配置参数。然而，在 SET STATEMENT CACHE 语句可成功地执行之前，您必须或通过设置 STMT_CACHE 配置参数或通过使用 **gadmin** 实用程序来启用 SQL 语句高速缓存。

当您发出 SET STATEMENT CACHE ON 语句时，SQL 语句高速缓存保持生效，直到您发出 SET STATEMENT CACHE OFF 语句为止，或直到程序结束为止。如果您不使用 SET STATEMENT CACHE，则缺省的行为依赖于 **STMT_CACHE** 环境变量或 STMT_CACHE 配置参数的设置。

### 开启高速缓存

使用 ON 选项来启用 SQL 语句高速缓存。当启用 SQL 语句高速缓存时，您通过 SQL 语句高速缓存执行的每一语句决定相匹配的高速缓存条目是否出现。如果如此，则数据库服务器使用高速缓存了的条目来执行该语句。

如果该语句没有相匹配的条目，则数据库服务器测试来查看它是否有资格进入高速缓存。要了解一语句必须满足哪些条件才能进入高速缓存，请参阅 SQL 语句高速缓存具备资格的标准。

## 对在 SQL 语句高速缓存中的相匹配的条目的限制

当数据库服务器考虑一语句是否与 SQL 语句高速缓存中的语句一致时，下列项必须相匹配：

- 字母大小写
- 注释
- 空格
- 优化设置
  - SET OPTIMIZATION 语句选项
  - 优化器伪指令
  - SET ENVIRONMENT OPTCOMPIND 语句选项或 **OPTCOMPIND** 环境变量的设置，或 ONCONFIG 文件中的 OPTCOMPIND 配置参数的设置。（如果对同一查询存在冲突的设置，则这是优先级的降序排列。）
- 并行性设置
  - SET PDQPRIORITY 语句选项或 **PDQPRIORITY** 环境变量的设置
- 查询文本字符串
- 文字

如果 SQL 语句在语义上等同于在 SQL 语句高速缓存中的语句，但文字不同，则不认为是一致的语句且具备条目进入高速缓存的资格。例如，下列 SELECT 语句是不相同的：

```
SELECT col1, col2 FROM tab1 WHERE col1=3;
SELECT col1, col2 FROM tab1 WHERE col1=5;
```

在此示例中，两个语句都进入到 SQL 语句高速缓存内。

然而，主变量名称并不重要。例如，下列选择语句认定为相同：

```
SELECT * FROM tab1 WHERE x = :x AND y = :y;
SELECT * FROM tab1 WHERE x = :p AND y = :q;
```

在先前的示例中，虽然主名称不同，但语句够资格，因为大小写、查询文本字符串和空格都匹配。然而，性能没有改进，因为通过 PREPARE 语句已解析和优化了每一语句。

## 关闭高速缓存

OFF 选项禁用 SQL 语句高速缓存。当您为您的会话关闭高速缓存时，不为那个会话执行 SQL 语句高速缓存代码。

在反复地执行相同的查询以及模式更改不频繁的环境中，设计 SQL 语句高速缓存旨在节省内存。如果不是这种情况，您可能想要关闭 SQL 语句高速缓存来避免高速缓存的开销。例如，如果你您几乎没有高速缓存结合，也就是说，当相对少量的匹配但存在许多新的条目进入高速缓存时，高速缓存管理的开销很高。在此情况下，请关闭 SQL 语句高速缓存。

如果您知道您正在执行许多不具备 SQL 语句高速缓存资格的语句，则您可能想要禁用它，并避免因查看是否每一 DML 语句具备插入到高速缓存内的资格而导致的测试开销。

## SQL 语句高速缓存具备资格的标准

可高速缓存在 SQL 语句高速缓存中的语句（于是，可与已在 SQL 语句高速缓存中出现的语句相匹配）必须满足特定的条件。

要具备高速缓存的资格，该语句必须满足所有下列条件：

- 它必须是 SELECT、INSERT、UPDATE 或 DELETE 语句。
- 它必须仅包含非 opaque 的内建的数据类型（不包括 BLOB、BOOLEAN、BYTE、CLOB、LVARCHAR 和 TEXT）。
- 它必须仅包含内建的运算符。
- 它不可包含用户定义的例程。
- 它不可包含临时表或远程表。
- 它不可包含在 Projection 列表中的子查询。
- 它不可是多语句 PREPARE 的一部分。
- 它不可有目标列上的用户权限限制。
- 在符合 ANSI 的数据库中，它必须包含完全具备资格的对象名称。

- 它不可需要重新优化。

## 在高速缓存插入之前需要重新执行

仅在数据库服务器对引用（有时称之为“命中”）具备资格的 SQL 语句的可配置数目计数之后，才将那个语句完全地插入到 SQL 语句高速缓存内。对于缺省值 0，在被高速缓存之前，不需要重新执行具备资格的 DML 语句。

然而，使用 STMT_CACHE_HITS 配置参数，数据库管理员（DBA）可指定在将具备资格的 DML 语句插入到语句高速缓存内之前，必须执行该语句的最少次数。通过将此设置为 1（或更大的值），DBA 拒绝将一次性的*即席*查询完全插入到 SQL 语句高速缓存内，从而降低高速缓存管理的开销。

## 在大小超出配置的限度之后启用或禁用插入

当高速缓存大小达到它的配置的大小（如通过 STMT_CACHE_SIZE 配置参数指定的那样）时，DBA 可通过将配置参数 STMT_CACHE_NOLIMIT 设置为 0 来防止将附加的具备资格的 SQL 语句插入到语句高速缓存。

## 准备好的语句和语句高速缓存

准备好的语句本来是对单个会话进行高速缓存。也就是说，如果执行准备好的语句多次（或如果打开单个游标多次），则那个会话反复地使用相同的准备好的查询计划。

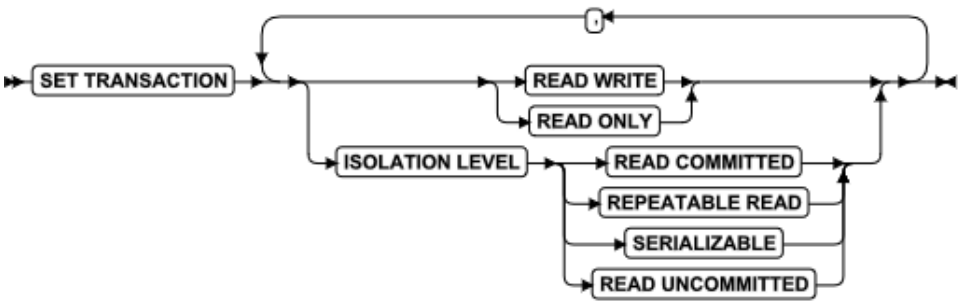
如果会话准备语句，然后执行它多次，则使用 SQL 语句高速缓存基本上不会对它的性能产生影响，因为在 PREPARE 语句期间，该语句只优化一次。

然而，如果其他会话也准备那个相同的语句，或如果第一个会话准备该语句几次，则语句高速缓存通常会带来直接的性能收益，因为数据库服务器仅计算该查询计划一次。当然，原始的会话可能从语句高速缓存获得小的收益，即便它仅准备该语句一次，因为其他会话使用较少的内存，且数据库服务器为其他会话执行较少的操作。

# 2.145 SET TRANSACTION 语句

使用 SET TRANSACTION 语句来定义隔离级别和指定事务的访问模式是只读还是读写。

**语法**




**用法**

SET TRANSACTION 仅在带有事务日志记录的数据库中是有效的。在打开数据库之后，您可从客户端计算机发出此语句。在尝试同时从数据库访问相同的行的进程之中，事务隔离级别影响并发性。在正在尝试读数据的进程之中，数据库服务器使用共享锁来支持不同的隔离级别，如下列列表所示：

- Read Uncommitted
- Read Committed
- (ANSI) Repeatable Read
- Serializable

更新或删除进程总是在正被修改的行上获得排他锁。隔离的级别不干涉这样的行，但访问模式不影响您可更新行还是可删除行。

如果另一进程尝试更新或删除您正在以 Serializable 或(ANSI)Repeatable Read 隔离级别读取的行，则会拒绝那个进程访问那些行。

## 对比 SET TRANSACTION 与 SET ISOLATION

SET TRANSACTION 语句符合 ANSI SQL-92。此语句类似于 GBase 8s SET ISOLATION 语句；然而，SET ISOLATION 语句不符合 ANSI 且不提供访问模式。实际上，您可以 SET TRANSACTION 语句设置的隔离级别与您可以 SET ISOLATION 语句设置的隔离级别极为相似，如下表所示。

| SET TRANSACTION 隔离级别 | SET ISOLATION 隔离级别 |
|---|---|
| Read Uncommitted | Dirty Read |
| Read Committed | Committed Read |
| [*不支持*] | Cursor Stability |
| (ANSI) Repeatable Read | ( GBase 8s ) Repeatable Read |
| Serializable | ( GBase 8s ) Repeatable Read |

SET TRANSACTION 与 SET ISOLATION 之间的另一差异是在事务内的隔离级别的行为。对于事务您仅可发出 SET TRANSACTION 一次。在那个事务期间打开的任何游标都要保证那个隔离级别（或访问模式，如果您正在定义访问模式的话）。在启动事务之后，你可以 SET ISOLATION 在该事务内多次更改隔离级别。

下列示例展示 SET ISOLATION 与 SET TRANSACTION 语句在行为方面的这种差异：

```
EXEC SQL BEGIN WORK;
        EXEC SQL SET ISOLATION TO DIRTY READ;
        EXEC SQL SELECT ... ;
        EXEC SQL SET ISOLATION TO REPEATABLE READ;
        EXEC SQL INSERT ... ;
        EXEC SQL COMMIT WORK;       -- Executes without error
```

请将前一示例与这些 SET TRANSACTION 语句对比：

```
EXEC SQL BEGIN WORK;
        EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- Produces error 876: Cannot issue SET TRANSACTION
-- in an active transaction.
```

SET ISOLATION 与 SET TRANSACTION 之间另一差异是隔离级别的持续时间。由于 SET ISOLATION 支持完整连接级别设置，因此通过 SET ISOLATION 指定的隔离级别保持生效，直到发出另一 SET ISOLATION 语句为止。通过 SET TRANSACTION 设置的隔离级别仅在事务终止之前保持生效。然后，隔离级别重置为数据库类型的缺省值。

## GBase 8s 隔离级别

下列定义说明每一隔离级别的关键特性，从最低隔离级别到最高的。

# 使用 Read Uncommitted 选项

使用 Read Uncommitted 选项来从数据库服复制行，不管它们之上是否有锁。获取行的程序不放置锁且不予考虑。Read Uncommitted 是没有事务的数据库唯一可用的隔离级别。

此隔离级别最适合于对其数据未在修改的静态表的查询，因为它不提供隔离。以 Read Uncommitted，程序可能返回在随后回滚了的事务之内插入或修改了的未提交的行。

SET TRANSACTION 的 Uncommitted Read 隔离级别不直接支持 SET ISOLATION 语句的 Committed Read 隔离级别的 LAST COMMITTED 特性。当应用尝试读取另一会话在修改数据时在其上持有排他锁的行时，LAST COMMITTED 特性可降低锁定冲突的风险。当启用此特性时，数据库服务器返回最近提交的数据的版本，而不是等待释放该锁。

然而，在下列环境之一中，此特性在使用 SET TRANSACTION 语句的 Uncommitted 隔离级别的所有用户会话中隐式地生效：

- 如果 USELASTCOMMITTED 配置参数设置为 'DIRTY READ' 或 'ALL'
- 如果 SET ENVIRONMENT 语句设置 USELASTCOMMITTED 会话环境选项为 'DIRTY READ' 或 'ALL'。

要获取关于 LAST COMMITTED 特性及其限制的信息，请参阅 Committed Read 的 LAST COMMITTED 选项 部分。

# 使用 Read Committed 选项

使用 Read Committed 选项来保证在检索行的时刻在表中提交每一被检索的行。此选项不在获取的行上放置锁。Read Committed 是不符合 ANSI 的带有日志记录的数据库中的缺省隔离级别。

当将每一行数据作为独立的单元来处理，而不引用同一表或其他表中的其他行时，Read Committed 是适合的。

SET TRANSACTION 的 Read Committed 隔离级别不直接支持 SET ISOLATION 语句的 Committed Read 隔离级别的 LAST COMMITTED 特性，当应用尝试读取另一会话在其上持有排他

的行级锁的行中的数据时，这可降低锁定冲突的风险。当启用此特性时，数据库服务器返回最近提交的数据的版本，而不是等待释放该锁。

然而，在下列环境之一之下，此特征在使用 SET TRANSACTION 语句的 Read Committed 隔离级别的所有用户会话中隐式地生效：

- 如果 USELASTCOMMITTED 配置参数设置为 'COMMITTED READ' 或 'ALL'
- 如果 SET ENVIRONMENT 语句设置 USELASTCOMMITTED 会话环境变量为 'COMMITTED READ' 或 'ALL'。

要获取更多关于 LAST COMMITTED 特性及其限制的信息，请参阅 Committed Read 的 LAST COMMITTED 选项 部分。

## 使用 Repeatable Read 和 Serializable 选项

Repeatable Read 与 Serializable 的 GBase 8s 实现是相同的。Serializable（Repeatable Read）选项在事务期间选择的每行上放置共享锁。

另一进程还可在选择的行上放置共享锁，但其他进程不可在您的事务期间修改任何选择的行，或在您的事务期间插入满足您的搜索条件的行。

*幻像行*是当您首次读取查询集时不可见的行，但在同一事务中查询集的随后读取中形成的行。仅此隔离级别防止对幻像行的访问。

如果您在事务期间重复该查询，则您重新读取相同的数据。仅当提交或回滚该事务时，才释放共享锁。Serializable 是符合 ANSI 的数据库中的缺省的隔离级别。Serializable 隔离放置的锁最多，持有它们的时间最长。因此它是最降低并发性的级别。

### 缺省的隔离级别

当您创建数据库时，建立缺省的隔离级别。

| GBase 8s 名称 | ANSI 名称 | 何时为缺省的隔离级别 |
|---|---|---|
| Dirty Read | Read Uncommitted | 不带有事务日志记录的数据库 |
| Committed Read | Read Committed | 不符合 ANSI 的带有日志记录的数据库 |
| Repeatable Read | Serializable | 符合 ANSI 的数据库 |

对于不符合 ANSI 的 GBase 8s 数据库，除非您显式地设置 USELASTCOMMITTED 配置参数，否则，LAST COMMITTED 特性对于缺省的隔离级别不生效。SET ENVIRONMENT 语句或 SET ISOLATION 语句可覆盖此缺省值并为当前的会话启用 LAST COMMITTED。

缺省的隔离级别保持生效，直到您在会话内发出 SET TRANSACTION 语句为止。在 COMMIT WORK 语句完成该事务或 ROLLBACK WORK 语句取消整个事务之后，隔离级别重置为缺省值。

当您使用“高可用性数据复制”时，数据库服务器在“HDR 辅助服务器”上有效地使用 Dirty Read 隔离，不理会指定的 **SET ISOLATION** 或 **SET TRANSACTION** 隔离级别，除非启用 UPDATABLE_SECONDARY 配置参数。要获取关于此主题的更多信息，请参阅 辅助数据复制服务器的隔离级别。

### 访问模式

访问模式影响事务内行的读和写并发性。使用访问模式来控制数据修改。SET TRANSACTION 可指定事务是只读的或读写的。在缺省情况下，事务是读写的。当您指定只读事务时，某些限制适用。只读事务不可执行下列活动：

- 插入、删除或更新表的行。
- 创建、改变或删除任何数据库对象，诸如模式、表、临时表、索引或 SPL 例程。
- 授予或取消访问权限。
- 更新统计信息。
- 重命名列或表。

您可在只读事务中执行 SPL 例程，只要该 SPL 例程不试图执行任何受限制的语句。

### 隔离级别的作用

您不可在没有日志记录的数据库中设置事务隔离级别。在无日志的数据库中的每一检索都作为 Read Uncommitted 发生。

在 BYTE 或 TEXT 数据的检索期间获取的数据可有所不同，这依赖于事务隔离级别。在 Read Uncommitted 或 Read Committed 隔离级别之下，允许进程读取 BYTE 或 TEXT 列，该列或是删除了的（如果该删除尚未提交的话）或正在删除过程中。在这些隔离级别之下，当存在某些条件时，应用可读取删除的 BYTE 或 TEXT 列。要了解关于这些条件的信息，请参阅 *GBase 8s 管理员指南*。

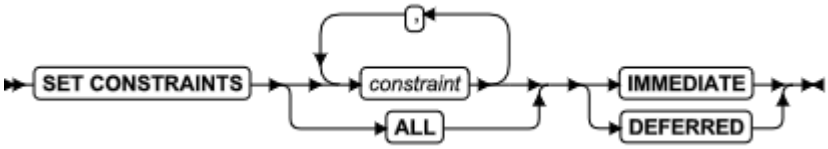
在 GBase 8s ESQL/C 中，如果您在事务中使用滚动游标，则可或通过设置隔离级别为 Serializable 或通过锁定整个表来强制您的临时表与数据库表之间的一致性。然而，持有滚动游标不可保证两表之间有相同的一致性。当事务完成时，释放通过 Serializable 设置的表级锁，在该事务的结束之外持有的滚动游标保持打开。一旦事务结束，您就可修改释放的行，因此在临时表中检索的数据可能与实际数据不一致。

**警告：** 请不要在事务内使用无日志记录的表。如果您需要在事务内使用无日志记录的表，请或设置隔离级别为 Repeatable Read，或在排他模式下锁定该表来防止并发问题。

## 2.146 SET Transaction Mode 语句

使用 SET Transaction Mode 语句来指定在当前事务期间，是在语句级还是在事务级检查约束。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constraint* | 要更改其事务模式的约束 | 所有约束必须在同一数据库中存在，该数据库必须支持日志记录 | 标识符 |

**用法**

要启用或禁用约束，或要更改它们的过滤模式，请参阅 SET Database Object Mode 语句。

此语句仅在带有事务日志记录的数据库中是有效的，且它的作用限于它在其中执行的事务。

使用 IMMEDIATE 关键字将约束的事务模式设置为语句级检查。当创建约束时，IMMEDIATE 是它们的缺省事务模式。

使用 DEFERRED 关键字来将事务模式设置为事务级检查。您不可将约束的事务模式更改为 DEFERRED，除非当前启用该约束。

## 语句级检查

当您将事务模式设置为 IMMEDIATE 时，开启语句级检查，且在每一 INSERT、UPDATE 或 DELETE 语句的结束时检查所有指定的约束。如果发生约束违反，则不执行该语句。

## 事务级检查

当您将约束的事务模式设置为 DEFERRED 时，关闭语句级检查，直到提交该事务时才检查所有的（或指定的）约束。在提交事务时，如果发生约束违反，则回滚该事务。

提示： 如果您推迟检查主键约束，则也推迟对那列或列的集合的非 NULL 约束检查。

## 事务模式的持续时间

SET Transaction Mode 语句指定的事务模式的持续时间是 SET Transaction Mode 语句在其中执行的事务。您不可在事务的外部执行此语句。一旦 COMMIT WORK 或 ROLLBACK WORK 语句执行成功，所有约束的事务模式都恢复到 IMMEDIATE。

要从事务级检查切换到语句级检查，您可使用 SET Transaction Mode 语句来设置事务模式为 IMMEDIATE，或您可使用 COMMIT WORK 或 ROLLBACK WORK 语句来终止您的事务。

## 指定所有约束或约束的列表

您可在 SET Transaction Mode 语句中指定数据库中的所有约束，或您可指定单个约束或约束的列表。

如果您指定 ALL 关键字，则 SET Transaction Mode 语句为数据库中的所有约束设置事务模式。如果事务中的任何语句需要检查数据库中任何表上的任何约束，则数据库服务器是在语句级还是在事务级执行检查，依赖于您在 SET Transaction Mode 语句中指定的设置。

如果您指定单个约束名称或约束的列表，则 SET Transaction Mode 语句仅为指定的约束设置事务模式。如果事务中的任何语句需要对您未在 SET Transaction Mode 语句中指定的约束进行检查，则在语句级检查那个约束，而不理会您为其他约束在 SET Transaction Mode 语句中指定的设置。

当您指定约束的列表时，不需要在同一表上定义这些约束，但它们必须存在于同一数据库中。

### 指定远程约束

您可设置本地约束或远程约束的事务模式。也就是说，在 SET Transaction Mode 语句中指定的约束可为在本地表上定义的约束，或在远程表上定义的约束。

### 设置约束的事务模式的示例

下列示例展示如何延迟事务内的约束检查，直到该事务完成为止。该示例中的 SET Transaction Mode 语句指定不检查数据库中任何表上的任何约束，直到遇到 COMMIT WORK 语句为止。

```
BEGIN WORK;
    SET CONSTRAINTS ALL DEFERRED;
        ...
    COMMIT WORK;
```

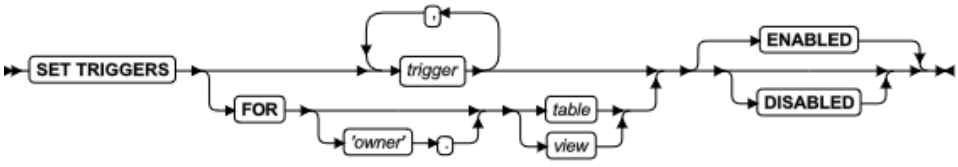
下列示例指定不检查约束的列表，直到该事务完成为止：

```
BEGIN WORK;
SET CONSTRAINTS update_const, insert_const DEFERRED;
...
COMMIT WORK;
```

## 2.147 SET TRIGGERS 语句

使用 SET TRIGGERS 语句来启用或禁用表上的所有或某些触发器，或视图上的所有或某些 INSTEAD OF 触发器。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *owner* | *table* 或 *view* 的所有者 | 必须拥有该表或视图 | 所有者名称 |

| | | | |
|---|---|---|---|
| *table* | 要启用或禁用其所有触发器的表 | 必须存在 | 标识符 |
| *trigger* | 要启用或禁用的触发器 | 必须存在 | 标识符 |
| *view* | 要启用或禁用其全部 INSTEAD OF 触发器的视图 | 必须存在 | 标识符 |

**用法**

SET TRIGGERS 语句是 SET Database Object Mode 语句的特例。SET Database Object Mode 语句还可启用或禁用索引或约束，或更改唯一索引或约束的过滤器模式。

要获取 SET TRIGGERS 语句的完整的语法和语义，请参阅 SET Database Object Mode 语句。

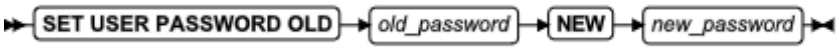
**在辅助服务器上的限制**

在集群环境中，在可更新的辅助服务器上不支持 SET TRIGGERS 语句。（更为一般地，SET Database Object Mode 语句指定的会话级索引、触发器和约束模式不会重新指向辅助服务器的数据库中表对象上的 UPDATE 操作。）

## 2.148 SET USER PASSWORD 语句 (UNIX™、Linux™)

使用 SET USER PASSWORD 语句来更改您的数据库服务器访问口令，如果您是内部认证的用户的话。此语句是对 SQL 语言的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_password* | 内部认证的用户的新口令。 | 长度必须在 6 至 32 字节之间。 | 引用字符串 |
| *old_password* | 内部认证的用户的现有口令。 | 长度必须在 6 至 32 字节之间。 | 引用字符串 |

**用法**

DBSA 不可使用此语句来更改另一用户的口令。要更改其他用户的口令，DBSA 可使用 ALTER USER 语句。

可以 PWUR 审计代码来审计 SET USER PASSWORD 语句的执行。

**示例**

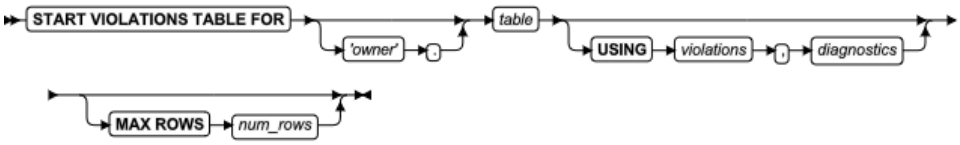
下列语句将口令从 **joebar** 更改为 **joefoo**：

```
SET USER PASSWORD OLD 'joebar' NEW 'joefoo';
```

# 2.149 START VIOLATIONS TABLE 语句

使用 START VIOLATIONS TABLE 语句来为指定的目标表创建违反表和诊断表。START VIOLATIONS TABLE 语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *diagnostics* | 声明要与目标 *table* 相关联的诊断表的名称。缺省名称为 table_dia。 | 在表、视图、序列和同义词的名称中必须是唯一的 | 标识符 |
| *num_rows* | 当在 *table* 上执行单个语句时，数据库服务器可插入到 *violations* 内的行的最大数目 | 必须为从 1 至 INTEGER 数据类型的最大值之间的整数 | 精确数值 |
| *owner* | *table* 的所有者 | 必须拥有该表 | 所有者名称 |
| *table* | 要为其创建 *violations* 表和 *diagnostics* 表的目标表 | 如果忽略 USING 子句，则不超过 124 字节 | 标识符 |
| *violations* | 要与 *table* 相关联的违反表。缺省的名称为 table_vio。 | 与 *diagnostics* 的约束相同 | 标识符 |

**用法**

数据库服务器将 *violations* 表和 *diagnostics* 表与您在 FOR 关键字之后指定的目标表相关联，通过在 **sysviolations** 系统目录表中记录三表之间的关系。

目标表必须满足这些要求：

- 不可为不是当前的数据库中的表。
- 它不可为 CREATE EXTERNAL TABLE 语句定义了的对象。
- 它不可已经与违反表或诊断表相关联。
- 它不可为系统目录表。

START VIOLATIONS TABLE 语句创建特定的违反表，该表持有在目标表上执行插入、更新和删除操作期间不能满足约束和唯一索引的不符合的那些行。此语句还创建特定的诊断表，该表包含关于在违反行中的每一行导致的完整性违反的信息。

## 与 SET Database Object Mode 语句的关系

START VIOLATIONS TABLE 语句与 SET Database Object Mode 语句关系密切。如果您使用 SET Database Object Mode 来将定义在表上的约束或唯一索引设置为 FILTERING 模式，还不使用 START VIOLATIONS TABLE，则不将在数据操作操作中违反约束或唯一索引要求的任何行过滤到违反表。相反，您会收到错误消息，表明您必须为目标表启动违反表。

类似地，如果您使用 SET Database Object Mode 语句来将禁用的约束或禁用的唯一索引设置为 ENABLED 或 FILTERING 模式，但您没有为在其上定义数据库对象的表使用 START VIOLATIONS TABLE，则不将不满足该约束或唯一索引要求的任何行过滤到违反表。

在这些情况下，要标识不满足约束或唯一索引要求的那些行，请发出 START VIOLATIONS TABLE 语句来启动违反表和诊断表。请在您使用 SET Database Object Mode 语句来将数据库对象设置为 ENABLED 或 FILTERING 数据库对象模式之前执行此操作。

## 对并发事务的影响

如果数据库有事务日志记录，则您必须单独发出 START VIOLATIONS TABLE。也就是说，当您在事务内对目标表发出 START VIOLATIONS TABLE 时，在目标表上不可有任何其他事务正在处理。在第一个事务已发出 START VIOLATIONS TABLE 语句之后，在目标表上启动的任何事务有关违反表和诊断表的行为都会与第一个事务的方式相同。也就是说，由这些随后的事务引起的任何约束或唯一索引违反都会记录在违反表和诊断表中。

例如，如果事务 A 在表 **tab1** 上操作，并在表 **tab1** 上发出 START VIOLATIONS TABLE 语句，则数据库服务器启动名为 **tab1_vio** 的违反表，并将由事务 A 在表 **tab1** 上引起的任何约束或唯一索引违反都过滤到表 **tab1_vio**。如果在事务 A 已发出了 START VIOLATIONS TABLE 语句之后，在表 **tab1** 启动事务 B 和 C，则还会将由事务 B 和 C 引起的任何约束和唯一索引违反过滤到表 **tab1_vio**。

结果就是，所有这三个事务都不会收到关于约束和唯一索引违反的错误消息，即使事务 B 和 C 并不期望该行为。例如，如果事务 B 在表 **tab1** 上发出一违反检查约束的 INSERT 或 UPDATE 语句，数据库服务器不会向事务 B 发出约束违反错误。数据库服务器反而会将不符合的行（也称为“坏行”）过滤到违反表，而不通知发生了数据完整性违反的事务 B。

当您在 SET Database Object Mode、CREATE TABLE、ALTER TABLE 或 CREATE INDEX 语句中指定 FILTERING 模式时，您可通过指定 WITH ERRORS 来防止在 GBase 8s 中发生这种情况。当多个事务在表上操作且 WITH ERRORS 选项生效时，在目标表上违反约束或唯一索引要求的任何事务都会收到数据完整性错误消息。

## 停止违反表和诊断表

在您使用 START VIOLATIONS TABLE 来在目标表与违反表和诊断表之间创建关联之后，删除在目标表与违反表和诊断表之间的关联的唯一方法就是为目标表发出 STOP VIOLATIONS TABLE 语句。要获取更多信息，请参阅 STOP VIOLATIONS TABLE 语句。

## USING 子句

您可使用 USING 子句来为违反表和为诊断表声明显式的名称。

如果您省略 USING 子句，则数据库服务器为违反表和诊断表指定名称。系统指定的违反表的名称由目标表的名称后跟字符串 _vio 构成。数据库服务器指定给诊断表的名称由目标表的名称后跟字符串 _dia 构成。

如果您省略 USING 子句，则目标表的名称的最大长度为 124 字节。

## 使用 MAX ROWS 子句

当在目标表上执行单个语句时，MAX ROWS 子句指定数据库服务器可插入到诊断表内的最大行数。如果您省略 MAX ROWS 子句，则当在目标表上执行单个语句时，对可插入到诊断表中的行的数目不设上限。

## 指定诊断表中行的最大数目

下列语句为名为 orders 的目标表启动违反表和诊断表。当在目标表上执行诸如 INSERT 之类的单个语句时，MAX ROWS 子句指定可插入到诊断表内的最大行数。

```
START VIOLATIONS TABLE FOR orders MAX ROWS 50000;
```

## 启动违反表和诊断表所需要的权限

要为目标表启动违反表或诊断表，您必须满足下列要求之一：

- 您必须有对数据库的 DBA 权限。
- 您必须是目标表的所有者且还有对数据库的 Resource 权限。
- 您必须有对目标表的 Alter 权限且还有对数据库的 Resource 权限。

## 违反表的结构

当您为目标表发出 START VIOLATIONS TABLE 时，该语句创建的违反表有预先定义的结构。此结构由目标表的列和三个附加列组成。

下表展示违反表的模式。

| 列名称 | 数据类型 | 列描述 |
| --- | --- | --- |
| 出现在目标表中的相同的列（以相同的顺序） | 与目标表中对应的列相同的类型。 | 违反表与目标表有相同的模式，因此可将在插入、更新和删除操作期间违反约束或唯一索引的那些行过滤到违反表。 |
| gbasedbt_tupleid | SERIAL | 对于不符合的行的唯一的序列代码 |
| gbasedbt_optype | CHAR(1) | 导致此坏行的操作的类型。此列可有下列值： |

| 列名称 | 数据类型 | 列描述 |
| --- | --- | --- |
| | | • I = Insert<br>• D = Delete<br>• O = Update（在此行中带有原始的值）<br>• N = Update（在此行中带有新值）<br>• S = SET Database Object Mode |
| gbasedbt_recowner | CHAR(32) | 发出了创建此不符合的行的语句的用户 |

如果 START VIOLATIONS TABLE 语句的目标表受到安全策略的保护，则数据库服务器以同样的安全策略保护该违反表。在此情况下，违反表的模式包括一 IDSSECURITYLABEL 列，其名称和在其他列之中的位置对应于目标表的 IDSSECURITYLABEL 列。当创建违反表时，在目标表中保护列的任何 SECURED WITH ***label*** 规范还保护对应的违反表列。

将目标表中的连续的列转换成违反表中的整数数据类型。

用户可检测在违反表中的这些不符合的行，分析在诊断表中包含诊断信息的相关的行，并采取纠正行动。

## START VIOLATIONS TABLE 语句的示例

下列示例展示执行 START VIOLATIONS TABLE 语句的不同方式。

### 带有缺省名称的违反表和诊断表

下列语句为名为 cust_subset 的目标表启动违反表和诊断表。在默认情况下，违反表名为 cust_subset_vio，诊断表名为 cust_subset_dia。

```
START VIOLATIONS TABLE FOR cust_subset;
```

### 带有显式的名称的违反表和诊断表

下列语句为名为 items 的目标表启动违反表和诊断表。USING 子句指定违反表和诊断表的显式的名称。违反表命名为 exceptions，诊断表命名为 reasons。

```
START VIOLATIONS TABLE FOR items USING exceptions, reasons;
```

## 目标表、违反表和诊断表之间的关系

用户可利用目标表、违反表和诊断表之间的关系来获得关于在 INSERT、DELETE 和 UPDATE 期间导致数据完整性违反的行的诊断信息。违反表的每一行至少在诊断表中有一对应的行。

- 违反表中的一行是在目标表中检测到了其数据完整性违反的任何行的一个副本。诊断表中的行包含由违反表中不符合的行导致的数据完整性违反的本质的信息。

- 违反表中的一行在 **gbasedbt_tupleid** 列中有唯一的序列标识符。诊断表中的行在它的 **gbasedbt_tupleid** 列中有相同的序列标识符。

违反表中的给定的行可在诊断表中有多个对应的行。诊断表中的多个行在它们的 **gbasedbt_tupleid** 列都有相同的序列标识符，以便于它们都链接到违反表中的同一行。对于违反表中的同一行，诊断表中可存在多行，因为违反表中的不符合的行可导致多次数据完整性违反。

例如，同一不符合的行可违反一列的唯一索引，违反另一列的非 NULL 约束，并违反第三列的检查约束。在此情况下，对于违反表中的单个不符合的行，诊断表包含三行。每一诊断行标识违反表不符合的行导致的一种不同的数据完整性违反。

通过连接违反表与诊断表，DBA 或目标表的所有者可获得关于违反表中任何或所有不符合的行的诊断信息。SELECT 语句可交互地执行这些连接，或您可写程序来在会话内执行它们。

## 对违反表的初始权限

当您发出 START VIOLATIONS TABLE 语句来创建违反表时，数据库服务器使用在目标表上授予的权限集作为对违反表授予权限的基础。然而，当数据库服务器授予每一权限类别时，它遵循不同的规则。

下表汇总数据库服务器在其下对违反表授予的每一类权限的环境。

| 权限 | 授予该权限的条件 |
| --- | --- |
| Alter | 未在违反表上授予 Alter 权限。（用户不可更改违反表。） |
| Index | 如果用户在目标表上有 Index 权限，则用户在违反表上有 Index 权限。 |
| Insert | 如果用户在目标表的任何列上有 Insert、Delete 或 Update 权限，则用户在违反表上有 Insert 权限。 |
| Delete | 如果用户在目标表的任何列上有 Insert、Delete 或 Update 权限，则用户在违反表上有 Delete 权限。 |
| Select | 如果用户在目标表的任何列上有 Select 权限，则用户在违反表的 **gbasedbt_tupleid**、**gbasedbt_optype** 和 **gbasedbt_recowner** 列上有 Select 权限。<br>如果用户在目标表中同一列上有 Select 权限，则用户在违反表的任何其他列上有 Select 权限。 |
| Update | 如果用户在目标表的任何列上有 Update 权限，则用户在违反表的 **gbasedbt_tupleid**、**gbasedbt_optype** 和 **gbasedbt_recowner** 列上有 Update 权限。<br>（然而，即使在 **gbasedbt_tupleid** 列上带有 Update 权限，用户也不可更新此 SERIAL 列。）<br>如果用户在目标表中的同一列上有 Update 权限，则用户在任何其他的违反表列上有 Update 权限。 |
| References | 不在违反表上授予 References 权限。（用户不可将引用约束添加到违反表。） |

下列规则适用于违反表的所有者和违反表上的权限：

- 当创建违反表时，目标表的所有者成为违反表的所有者。
- 违反表的所有者自动地收到违反表上的所有表级权限，包括 Alter 和 References 权限。然而，数据库服务器防止违反表的所有者更改违反表或将引用约束添加到违反表。
- 您可使用 GRANT 和 REVOKE 语句来更改违反表上的权限的初始集。
- 当您在其上定义了过滤模式唯一索引或约束的目标表上发出 INSERT、DELETE 或 UPDATE 语句时，您必须在违反表和诊断表上有 Insert 权限。

  如果您在违反表和诊断表上没有 Insert 权限，则数据库服务器在目标表上执行 INSERT、DELETE 或 UPDATE 语句，假设您在目标表上有必要的权限。数据库服务器不返回关于在违反表和诊断表上缺少 Insert 权限的错误，除非在执行 INSERT、DELETE 或 UPDATE 语句期间检测到完整性违反。

  类似地，当您发出 SET Database Object Mode 语句来将禁用的约束或禁用的唯一索引设置为启用的或过滤器模式，且对于目标表存在违反表和诊断表，则您必须在违反表和诊断表上有 Insert 权限。

  如果您在违反表和诊断表上没有 Insert 权限，则数据库服务器执行 SET Database Object Mode 语句，如果您在在目标表上有必要的权限的话。 数据库服务器不返回有关在违反表和诊断表上缺少 Insert 权限的错误，除非在执行 SET Database Object Mode 语句期间检测到完整性违反。
- 在违反表上的权限的初始集的授予者与在目标表上的权限的授予者相同。

  例如，如果用户 **jill** 和用户 **albert** 授予了用户 **henry** 在目标表上的 Insert 权限，则 **jill** 和 **albert** 将违反表上的 Insert 权限授予 **henry**。
- 在启动违反表之后，从一用户撤销目标表上的权限不会自动地从那个用户撤销在违反表上的同一权限。您反而必须显式地从该用户撤销在违反表上的该权限。
- 如果您在目标表上有分片级权限，则您在违反表上有对应的分片级权限。

## 违反表上权限的示例

下列示例展示如何从目标表上的权限的当前集推导出违反表上的权限的初始集。 假设名为 **cust_subset** 的表由下列列组成：**ssn** （客户“社会保险”号）、**fname**（客户的名）、**lname**（客户的姓）和 **city**（客户生活的城市）。

在 cust_subset 表上存在下列权限集：

- 用户 **barbara** 在该表上有 Insert 和 Index 权限。她还在 **ssn** 和 **lname** 列上有 Select 权限。
- 用户 **carrie** 在 **city** 列上有 Update 权限。她还在 **ssn** 列上有 Select 权限。
- 用户 **danny** 在该表上有 Alter 权限。

现在，用户 alvin 为 cust_subset 表启动名为 cust_subset_viols 的违反表和名为 cust_subset_diags 的诊断表：

```
START VIOLATIONS TABLE FOR cust_subset
        USING cust_subset_viols, cust_subset_diags;
```

数据库服务器在 cust_subset_viols 违反表上授予下列初始权限集：

- 用户 **alvin** 是违反表的所有者，因此他在表上有所有表级权限。
- 用户 **barbara** 在表上有 Insert、Delete 和 Index 权限。

  用户 **barbara** 在违反表的五列上有 Select 权限：**ssn**、the **lname**、**gbasedbt_tupleid**、**gbasedbt_optype** 和 **gbasedbt_recowner** 列。
- 用户 carrie 在违反表上有 Insert 和 Delete 权限。

  用户 **carrie** 在违反表的四列上有 Update 权限：**city**、**gbasedbt_tupleid**、**gbasedbt_optype** 和 **gbasedbt_recowner** 列。然而，她不可更新 **gbasedbt_tupleid** 列（因为这是 SERIAL 列）。

  用户 **carrie** 在违反表的四列上有 Select 权限：**ssn** 列、**gbasedbt_tupleid** 列、**gbasedbt_optype** 列和 **gbasedbt_recowner** 列。
- 用户 **danny** 在违反表上没有权限。

## 使用违反表

下列规则涉及违反表的结构和使用：

- 违反表中每对更新行都在 **gbasedbt_tupleid** 列中有相同的值，表明两行都引用目标表中的同一行。
- 如果目标表有名为 **gbasedbt_tupleid**、**gbasedbt_optype** 或 **gbasedbt_recowner** 的列，则数据库服务器尝试通过在列名称尾部附加数字，在违反表中为这些列生成可替代的名称（例如 **gbasedbt_tupleid1**）。如果这样做失败，则返回错误，且不为目标表启动违反表。
- 当某个表作为违反表时，它不可在其上定义触发器或约束。
- 当某个表作为违反表时，用户可在它之上创建索引，即使存在索引会影响性能。不可将违反表上的唯一索引设置为 FILTERING 数据库对象模式。
- 如果目标表有违反表和诊断表与它相关联，则以级联模式（缺省的模式）删除该目标表导致违反表和诊断表也被删除。如果以受限模式删除该目标表，则 DROP TABLE 操作失败（因为存在违反表和诊断表）。
- 在为目标表启动违反表之后，ALTER TABLE 不可添加、修改或删除违反表、诊断表或目标表的列。在您可更改任何这些表之前，您必须为目标表发出 STOP VIOLATIONS TABLE 语句。
- 在 INSERT、UPDATE、DELETE 或 SET Database Object Mode 操作期间，在数据库服务器使用违反表之前或之后，它不清除违反表的内容。

- 如果目标表在它之上定义有过滤器模式约束或唯一索引，且违反表与它相关联，则用户不可通过从违反表选择来插入到目标表内。 在您通过从违反表选择来将行插入到目标表内之前，您必须采用下列步骤之一：
    - 您可将约束或唯一索引设置为 DISABLED 模式。
    - 您可为目标表发出 STOP VIOLATIONS TABLE。

  如果不便于采取这些步骤之一，但您仍想将记录从违反表复制到目标表内，则第三个选项就是从违反表选择到临时表内，然后再将临时表的内容插入到目标表内。
- 如果在 START VIOLATIONS TABLE 语句中指定的目标表是分片的，则违反表与目标表有相同的分片策略。违反表的每一分片与目标表的对应分片存储在相同的 dbspace 分区中。
- 一旦为目标表启动违反表，您不可使用 ALTER FRAGMENT 语句来更改目标表或违反表的分片策略。
- 如果在 START VIOLATIONS TABLE 语句中指定的目标表未分片，则数据库服务器将违反表放置在与目标表相同的 dbspace 中。
- 如果目标表有 BYTE 或 TEXT 列，则在存储目标表中的 BYTE 或 TEXT 数据的同一 blobspace 中创建违反表中的 BYTE 或 TEXT 数据值。

## 违反表的示例

要在演示数据库中为名为 customer 的目标表启动违反表和诊断表，请输入下列语句：

```
START VIOLATIONS TABLE FOR customer;
```

由于您未包括 USING 子句，所以将违反表缺省地命名为 **customer_vio**。**customer_vio** 表包括这五列：

| 列一 | 列二 | 列三 | 列四 | 列五 |
|---|---|---|---|---|
| customer_num<br>fname lname | company<br>address1<br>address2 | city<br>state | zipcode<br>phone | gbasedbt_tupleid<br>gbasedbt_optype<br>gbasedbt_recowner |

**customer_vio** 表与 **customer** 表有相同的表定义，除了 **customer_vio** 表有三个附加的列之外，这些列包含关于导致不符合行的操作的信息。

## 诊断表的结构

当您为目标表发出 START VIOLATIONS TABLE 语句时，该语句创建的诊断表有预先定义的结构。此结构与目标表的结构无关。

下表展示诊断表的模式。

| 列名称 | 数据类型 | 描述 |
|---|---|---|

| 列名称 | 数据类型 | 描述 |
| --- | --- | --- |
| gbasedbt_tupleid | INTEGER | 隐式地引用违反表中的 gbasedbt_tupleid 列值。然而，不将此关系声明为外键对主键的关系。 |
| objtype | CHAR(1) | 标识违反的类型。此列可有下列值：<br>C = 约束违反<br>I = 唯一索引违反 |
| objowner | CHAR(32) | 表示为其检测到了完整性违反的约束或索引的所有者 |
| objname | VARCHAR(128, 0) | 包含为其检测到了完整性违反的约束或索引的名称 |

## 诊断表上的初始权限

当 START VIOLATIONS TABLE 语句创建诊断表时，在目标表上授予的访问权限集是在诊断表上授予权限的基础。然而，当数据库服务器授予每一类权限时，它遵循下列规则：

下表说明数据库服务器在诊断表上授予每一权限所处的环境。

**权限　　授予权限的条件**

Insert　　如果用户在目标表的任何列上有 Insert、Delete 或 Update 权限，则用户有在诊断表上的 Insert 权限。

Delete　　如果用户在目标表的任何列上有 Insert、Delete 或 Update 权限，则用户在诊断表上有 Delete 权限。

Select　　如果用户在目标表的任何列上有 Select 权限，则用户在诊断表上有 Select 权限。

Update　　如果用户在目标表中的任何列上有 Update 权限，则用户在诊断表上有 Update 权限。

Index　　如果用户在目标表上有 Index 权限，则用户在诊断表上有 Index 权限。

Alter　　不在诊断表上授予 Alter 权限。

（用户不可更改诊断表的模式。）

References 不在诊断表上授予 References 权限。

（用户不可在诊断表上定义引用的约束。）

下列规则涉及在诊断表上的访问权限：

- 当创建诊断表时，目标表的所有者成为诊断表的所有者。
- 诊断表的所有者自动地收到诊断表上的所有表级权限，包括 Alter 和 References 权限。然而，数据库服务器防止诊断表的所有者更改诊断表或給诊断表添加引用的约束。

- 您可使用 GRANT 和 REVOKE 语句来修改在诊断表上的初始权限集。
- 对于在其上定义有过滤器模式唯一索引或约束的目标表上的 INSERT、DELETE 或 UPDATE 操作，您必须在违反表和诊断表上有 Insert 权限。

  如果您在违反表和诊断表上没有 Insert 权限，则数据库服务器在目标表上执行 INSERT、DELETE 或 UPDATE 语句，假如您在目标表上有必要的权限的话。数据库服务器不返回有关在违反表和诊断表上缺少 Insert 权限的错误，除非在执行 INSERT、DELETE 或 UPDATE 语句期间检测到完整性违反。

  类似地，当您发出 SET Database Object Mode 语句来将禁用的约束或禁用的唯一索引设置为启用的或过滤器模式，且对于目标表存在违反表和诊断表时，您必须在违反表和诊断表上有 Insert 权限。

  如果您在违反表和诊断表上没有 Insert 权限，则数据库服务器执行 SET Database Object Mode 语句，假如您在目标表上有必要的权限的话。数据库服务器不返回有关在违反表和诊断表上缺少 Insert 权限的错误，除非在执行 SET Database Object Mode 语句期间检测到完整性违反。

- 在诊断表上的权限的初始集的授予者与在目标表上的权限的授予者相同。例如，如果通过用户 **wayne** 和用户 **laurie** 在目标表上授予了 用户 **jenny** Insert 权限，则用户 **wayne** 和用户 **laurie** 都将在诊断表上的 Insert 权限授予用户 **jenny**。
- 一旦为目标表启动诊断表，从一用户取消在目标表上的权限不会自动地从该用户取消在诊断表上的同一权限。您反而必须显式地在诊断表上从该用户取消该权限。
- 如果您在目标表上有分片级权限，则您在诊断表上有相应的表级权限。

下一示例展示如何从目标表上的当前权限推导出诊断表上的权限的初始集。 假设您有持有 customer 数据的名为 cust_subset 的表。此表由下列列组成：ssn （社保编号）、fname（名）、lname（姓）和 city（客户生活的城市）。在 cust_subset 表上存在下列访问权限集：

- 用户 **alvin** 是表的所有者。
- 用户 **barbara** 有对表的 Insert 和 Index 权限。她还有在 **ssn** 和 **lname** 列上的 Select 权限。
- 用户 **danny** 有表上的 Alter 权限。
- 用户 **carrie** 在 **city** 列上有 Update 权限。她还在 **ssn** 列上有 Select 权限。

现在，用户 alvin 为 cust_subset 表启动名为 cust_subset_viols 的违反表和名为 cust_subset_diags 的诊断表：

```
START VIOLATIONS TABLE FOR cust_subset
USING cust_subset_viols, cust_subset_diags;
```

数据库服务器在 cust_subset_diags 诊断表上授予下列初始的权限集：

- 用户 **alvin** 是诊断表的所有者，因此他有表上的所有表级权限。
- 用户 **barbara** 在诊断表上有 Insert、Delete、Select 和 Index 权限。
- 用户 **carrie** 在诊断表上有 Insert、Delete、Select 和 Update 权限。
- 用户 **danny** 在诊断表上没有权限。

## 使用诊断表

下列问题涉及诊断表的结构和使用。

- 当您在目标表上执行诸如 INSERT 或 SET Database Object Mode 语句这样的单个语句时，START VIOLATIONS TABLE 语句的 MAX ROWS 子句对可插入到诊断表内的行数设置限制。
- MAX ROWS 子句仅对该表作为诊断表的操作限制行数。
- 当某表作为诊断表时，它不可在它之上定义触发器或约束。
- 当某表作为诊断表时，用户可在该表上创建索引，但索引的存在会影响性能。您不可将诊断表上的唯一索引设置为 FILTERING 数据库对象模式。
- 如果对象表有违反表和诊断表与它相关联，则以级联模式（缺省的模式）删除对象表会导致违反表和诊断表也被删除。
- 如果以受限的模式删除目标表，则 DROP TABLE 操作失败（因为存在违反表和诊断表）。
- 一旦为目标表启动违反表，您不可使用 ALTER TABLE 语句来添加、修改或删除目标表、违反表或诊断表中的列。在您可修改任何这些表之前，您必须为目标表发出 STOP VIOLATIONS TABLE 语句。
- 在 Insert、Update、Delete、Merge、SET CONSTRAINTS 或 SET INDEXES 操作期间，在数据库服务器使用诊断表之前或之后，它不清除诊断表的内容。
- 如果在 START VIOLATIONS TABLE 语句中指定的目标表是分片的，则诊断表以轮转法策略在目标表被分片的同一 dbspace 之上分片。

要在演示数据库中为名为 stock 的目标表启动违反表和诊断表，请输入下列语句：

```
START VIOLATIONS TABLE FOR stock;
```

由于您的 START VIOLATIONS TABLE 语句不包括 USING 子句，因此缺省地命名诊断表为 stock_dia。stock_dia 表包括下列两列：

| 列一 | 列二 |
| --- | --- |
| gbasedbt_tupleid objtype | objowner objname |

对于目标表，此列的列表展示诊断表与违反表之间的差异。尽管违反表有与目标表中每列相匹配的列，而诊断表的列不与目标表中的任何列相匹配。通过任何 START VIOLATIONS TABLE 语句创建的诊断表都有相同的列，有相同的列名称和数据类型。

要获取更多关于诊断表与违反表之间关系的信息，请参阅 目标表、违反表和诊断表之间的关系。

# 2.150 STOP VIOLATIONS TABLE 语句

使用 STOP VIOLATIONS TABLE 语句来删除目标表、它的违反表与它的诊断表之间的关联。此语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *owner* | *table* 的所有者 | 必须拥有该表 | 所有者名称 |
| *table* | 与要删除的违反表和诊断表相关联的目标表的名称。不存在缺省值。 | 必须是与违反表和诊断表相关联的本地表 | 标识符 |

**用法**

STOP VIOLATIONS TABLE 语句删除目标表、违反表与诊断表之间的关联。在您发出此语句之后，以前的违反表和诊断表继续存在，但不再作为该目标表的违反表和诊断表。现在，它们有常规的数据库表的状态，而不是目标表的违反表和诊断表。您必须发出 DROP TABLE 语句来显式地删除这两个表。

当 DML 操作（INSERT、DELETE 或 UPDATE）对目标表的行造成数据完整性违反时，不再将不符合的行过滤到以前的违反表，且不将关于数据完整性违反的诊断信息放置到以前的诊断表中。

## 停止违反表和诊断表的示例

假设名为 cust_subset 的目标表与名为 cust_subset_vio 的违反表相关联，且与名为 cust_subset_dia 的诊断表相关联。要删除目标表与违反表和诊断表之间的关联，请输入下列语句：

```
STOP VIOLATIONS TABLE FOR cust_subset;
```

这删除已注册了以前的相关联的 **sysviolations** 系统目录表中的行。对在目标 **cust_subset** 表上的随后的 DML 操作将不再导致数据库服务器将关于不符合的行的信息插入到它以前的违反表和诊断表内。

## 删除违反表和诊断表的示例

在您在前面的示例中执行 STOP VIOLATIONS TABLE 语句之后，cust_subset_vio 和 cust_subset_dia 表继续存在，但不再与 cust_subset 表相关联。相反现在它们有常规的数据库表的状态。要删除这两个表，请输入下列语句：

```
DROP TABLE cust_subset_vio;
DROP TABLE cust_subset_dia;
```

如果您先前已经发出了的不带有 RESTRICT 关键字的 DROP TABLE 语句成功地删除 **cust_subset** 表，则上面的语句可能并不必要，因为以级联模式删除目标表会隐式地删除任何相关联的违反表和诊断表。

## 停止违反表所需要的权限

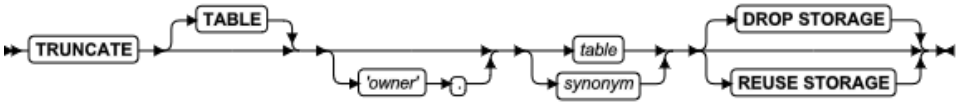
要为给定的目标表停止违反表或诊断表，您必须满足下列要求之一：

- 您在该数据库上必须有 DBA 权限。
- 您必须是目标表的所有者，且在数据库上有 Resource 权限。
- 您在目标表上必须有 Alter 权限，且在数据库上有 Resource 权限。

# 2.151 TRUNCATE 语句

使用 TRUNCATE 语句来快速地从本地表删除所有行，并释放相关联的存储空间。您可可选地为同一表及其索引保留该空间。仅 GBase 8s 支持 TRUNCATE 语句的此实现，这是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | *table* 或 *synonym* 的所有者 | 请参阅**用法**说明。 | 所有者名称 |
| *synonym* | 要从其移除所有数据的表的同义词 | 必须存在，且必须未设置 USETABLENAME | 标识符 |
| *table* | 从其移除所有数据和它的索引的所有 B 树结构的表名称 | 必须在数据库中存在 | 标识符 |

**用法**

TRUNCATE 语句快速地从本地表删除所有活动的数据行和表上索引的 B 树结构。您有释放由行和索引 extent 占据的存储空间的选项，或当随后以新的行重新植入该表时重用相同的空间。

要执行 TRUNCATE 语句，必须满足下列条件中至少一条：

- 您是该表的所有者。
- 您在该表上持有 Delete 访问权限。
- 您在当前的数据库上持有 DBA 访问权限。

如果在该表上定义启用的 Delete 触发器，则您还必须在该表上持有 Alter 权限，即使 TRUNCATE 语句不激活触发器。

对于不拥有该表的非 DBA 用户，虽然要求有 Delete 权限，但 TRUNCATE 是数据定义语言（DDL）语句。像其他 DDL 语句一样，TRUNCATE 不可在您连接到的数据库之外的任何表上操作，也不可在并发会话正在以 Dirty Read 隔离模式读取的表上操作。

GBase 8s 总是记录 TRUNCATE 操作日志，即使是对于无日志记录的表。在支持会话日志记录的数据库中，在同一事务之内的 TRUNCATE 语句之后，仅 SQL 的 COMMIT WORK 或 ROLLBACK WORK 语句是有效的。在此，ROLLBACK 语句必须取消包括 TRUNCATE 语句的整个事务。如果 ROLLBACK TO SAVEPOINT（或除了不带有 TO SAVEPOINT 子句的 COMMIT WORK 或 ROLLBACK WORK 之外的任何其他 SQL 语句）紧跟在 TRUNCATE 语句之后，则 GBase 8s 发出错误。

当您成功地回滚 TRUNCATE 语句时，不从该表移除任何行，且持有这些行和索引分区的存储 extent 会被分配给该表。仅带有事务日志记录的数据库可支持 ROLLBACK WORK 语句。

在 TRUNCATE 语句成功地执行之后， GBase 8s 自动地为该表及其在系统目录中的索引更新统计信息和分发，来展示在该表中没有行，在它的 dbspace 分区中也没有行。不必在您提交 TRUNCATE 语句之后立即运行 UPDATE STATISTICS 语句。

如果 TRUNCATE 语句指定的表是 typed 表，则成功的 TRUNCATE 操作从那个表和从表层级内所有的它的子表移除所有行和 B 树结构。

TRUNCATE 语句不重置 SERIAL、BIGSERIAL 或 SERIAL8 列的序列值。要重置序列列的计数器，请使用 ALTER TABLE 语句的 MODIFY 子句，在您执行 TRUNCATE 语句之前或之后都可以。

## TABLE 关键字

TABLE 关键字对此语句没有影响，但可包含它来使您的代码更加清晰易读。下列两个语句有相同的作用，都是从 customer 表删除所有行和任何相关的索引数据：

```
TRUNCATE TABLE customer;

TRUNCATE customer;
```

## 表规范

您必须指定您当前连接到的本地数据库中表的名称或同义词。如果设置 **USETABLENAME** 环境变量，则您必须使用表的名称，而不是同义词。该表可为 STANDARD、RAW 或 TEMP 类型，但您不可指定视图的名称或同义词，或 CREATE EXTERNAL TABLE 语句定义了的对象。（在 对 TRUNCATE 语句的限制 部分中罗列对 TRUNCATE 无效的表的类别。）

在符合 ANSI 的数据库中，如果您不是 ***table*** 或 ***synonym*** 的所有者，则您必须指定 ***owner*** 限定符。

在 TRUNCATE 语句开始执行之后， GBase 8s 尝试在指定的表上放置排他锁，以防止其他会话锁定该表，直到提交或回滚 TRUNCATE 语句为止。排他锁还适用于表层级之内的截断表的任何依赖表。

然而，在使用 Dirty Read 隔离级别的并发会话正在读表时，TRUNCATE 语句失败并报 RSAM -106 错误。要降低此风险，您可设置 **IFX_DIRTY_WAIT** 环境变量来指定 TRUNCATE 操作等待 Dirty Read 事务提交或回滚的指定的秒数。

## STORAGE 规范

当 TRUNCATE 操作开始时，可选的 DROP STORAGE 或 REUSE STORAGE 关键字指定数据库服务器对表的存储 extent 采取什么行动。如果您省略此规范，则 DROP STORAGE 选项是缺省的。

使用缺省的或显式的 DROP STORAGE 选项，成功的 TRUNCATE 语句释放当前分配给表及其索引的 extent 之中除了第一个 extent 之外的所有 extent。您可以 `gcheck -pT` ***table*** 命令显示当前的 extent 列表。如果您仅有一个 extent，则不会释放空间。

例如，对于下表，将 4 个缺省的 8 页 extent 合并到当前的第一个 extent 内。还有第二个更大的 extent：

```
Extents
        Logical Page    Physical Page       Size Physical Pages
        0               1:104455            32                  32
        32              1:104495            4576                4576
```

在 TRUNCATE 语句运行之后，从同一 gcheck 命令的输出显示为：

```
Extents
        Logical Page    Physical Page       Size Physical Pages
        0               1:104455            32                  32
```

或者，如果您想要为随后加载的数据保持分配给同一表的相同的存储空间，请指定 REUSE STORAGE 关键字来防止该空间被分配。TRUNCATE 的 REUSE STORAGE 选项可使得引用中的存储管理更有效，在这些应用中，定期地清空同一表并以新的行重新加载。

下列示例删除 state 表，并释放它的除了第一 extent 之外的所有 extent：

```
TRUNCATE TABLE state DROP STORAGE;
```

下列示例删除同一表，但仅移除实际的数据。所有 extent 保持不变。

```
TRUNCATE TABLE state REUSE STORAGE;
```

不论您指定 DROP STORAGE 还是 REUSE STORAGE，当提交 TRUNCATE 事务时，对于表的所有行，释放任何行外数据值。还释放在 TRUNCATE 事务中不再被引用的任何 BLOB 或 CLOB 值所占据的存储。

## AM_TRUNCATE 目的函数

GBase 8s 为支持在永久表和临时表的列上的 TRUNCATE 操作的主访问方式提供内建的 **am_truncate** 目的函数。它还为 B 树索引上的 TRUNCATE 操作的辅助访问方式提供内建的 **am_truncate** 目的函数。

为了 TRUNCATE 语句在虚拟表接口（VTI）中正确地运行，在主访问方式中为 VTI 表的数据类型需要有效的 am_truncate 目的函数。要在数据库中注册新的主访问方式，请使用 SQL 的 CREATE PRIMARY ACCESS_METHOD 语句：

```
CREATE PRIMARY ACCESS_METHOD vti(
        AM_GETNEXT    = vti_getnext
        AM_TRUNCATE = vti_truncate
        ...);
```

您还可使用 ALTER ACCESS_METHOD 语句来将有效的 am_truncate 目的函数添加到一没有 am_truncate 目的函数的现有的访问方式：

```
ALTER ACCESS_METHOD abc (ADD AM_TRUNCATE = abc_truncate);
```

在这些示例中，**vti_truncate** 和 **abc_truncate** 函数必须是支持 AM_TRUNCATE 目标选项关键字的功能的例程，且通过 CREATE FUNCTION 或 CREATE ROUTINE FROM 语句预先在数据库中注册了。

## TRUNCATE 的性能优势

TRUNCATE 语句不等同于 DROP TABLE。在 TRUNCATE 成功地执行之后，指定的 ***table*** 及其所有列和索引仍注册在数据库中，但没有数据行。在一定的时间间隔之后要求替换表中所有记录的信息管理应用中，TRUNCATE 比同等的 DROP TABLE、CREATE TABLE，以及要定义任何同义词、视图、约束、触发器、权限、分片方案和其他属性以及表的相关联的数据库对象，需要对系统目录的较少更新。

在不需要表的现有行的上下文中，与使用不带有 WHERE 子句的 DELETE 语句来清空表相比，TRUNCATE 语句通常更有效率，因为 TRUNCATE 比 DELETE 需要更少的资源和更低的日志记录开销：

- DELETE FROM ***table*** 删除每一行作为分别的日志记录操作。如果在表上存在索引，则当删除行时，必须更新每一索引，且此更新还为每一行做日志记录。如果在表上定义一启用的 Delete 触发器，则还执行它的触发的活动并记录日志。
- TRUNCATE ***table*** 执行对表上所有行和每一索引的 B 树结构的移除，作为单个操作，并当提交或回滚包括 TRUNCATE 的事务时，在逻辑日志中写单个条目。忽略任何启用的触发器的被触发的活动。

当表有带有下列属性的一个或多个列时，TRUNCATE 胜于 DELETE 的这些性能优势会降低：

- 存储在 blobspace 中的任何简单大对象数据类型
- 存储在 sbspace 中的任何 BLOB、CLOB、复杂的或用户定义的类型
- 为其定义 **destroy** 支持函数的任何 opaque 类型。

这些特性的每一个都需要数据库服务器读取表的每一行，大幅降低 TRUNCATE 的速度。

如果表包括一个或多个您已为其注册了 **am_truncate( )** 目的函数的 UDT，则 TRUNCATE 与 DELETE 之间的性能差异会在为 TRUNCATE 调用 **am_truncate** 接口一次与为每一行调用 **destroy( )** 支持函数的相对成本上反映出来。

像在下一部分中罗列的那样，某些条件导致 TRUNCATE 失败并报错。这些条件中的某些在 DELETE 操作上没有影响，因此在那些情况下，您以 DELETE 语句移除所有行会更有效，如在 **customer** 表上的下列操作所示：

```
DELETE customer;
```

仅当设置 DELIMIDENT 环境变量时，才可省略紧跟在 DELETE 之后的 FROM 关键字，如在此示例中那样。

## 对 TRUNCATE 语句的限制

如果存在下列任何条件，则 TRUNCATE 语句失败：

- 用户不持有在表上的 Delete 访问权限。
- 该表有启用的 Delete 触发器，但用户缺少 Alter 权限。
- 在本地数据库中不存在指定的表或同义词。
- 通过 CREATE EXTERNAL TABLE 语句定义了此表。
- 指定的同义词未引用本地服务器中的表。

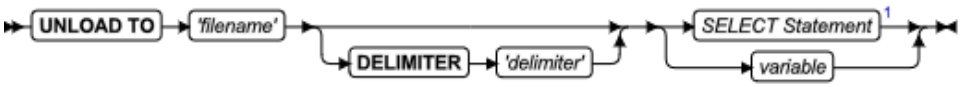
- 该语句为本地表指定同义词，但设置了 **USETABLENAME** 环境变量。
- 该语句为视图指定视图或同义词的名称。
- 该表为系统目录表或系统监视接口（SMI）表。
- 在该表上定义 R 树索引。
- 该表是在数据库中不存在对于其有效的 **am_truncate** 访问方式的虚拟表（或有虚拟索引接口）。
- Enterprise Replication 复制的不是表上定义的主复制。（要获取更多关于复制的信息，请参阅 *GBase 8s Enterprise Replication 指南*。）
- 在表上已存在共享锁或排他锁。
- 在表上打开一个或多个游标。
- 带有 Dirty Read 隔离级别的并发会话正在读表。
- 至少有一行的另一表在指定的表上有启用的外键约束。（然而，没有行的另一表的启用的外键约束对 TRUNCATE 操作不起作用。）

## 2.152 UNLOAD 语句

使用 UNLOAD 语句来将通过 SELECT 语句检索到的行写到操作系统文件。UNLOAD 语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**

仅 DB-Access 支持 UNLOAD 语句。




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *delimiter* | 指定在 *filename* 文件中的字段定界符字符的加引号的字符串 | 请参阅 DELIMITER 子句 | 引用字符串 |
| *filename* | 要接收这些行的操作系统文件。缺省的 *pathname* 是当前的目录。 | 请参阅 UNLOAD TO 文件。 | 引用字符串 |
| *variable* | 包含有效的 SELECT 语句的文本的主变量 | 必须已被声明为字符数据类型 | 特定于语言 |

**用法**

**重要：** 仅随同 DB-Access 使用 UNLOAD 语句。

UNLOAD 语句将通过查询检索到的行复制到文件。您必须在 SELECT 语句中指定的所有列上有 Select 权限。要获取关于数据库级和表级权限，请参阅 GRANT 语句。

您可指定字面的 SELECT 语句，或包含 SELECT 语句的文本的字符变量（请参阅 SELECT 语句。）

下列示例卸载其 customer.customer_num 的值大于或等于 138 的行，并将它们写到名为 cust_file 的文件：

```
UNLOAD TO 'cust_file' DELIMITER '!'
        SELECT * FROM customer WHERE customer_num> = 138;
```

结果输出文件 cust_file 包含两行数据值：

```
138!Jeffery!Padgett!Wheel Thrills!3450 El Camino!Suite 10!Palo Alto!CA!94306!!
        139!Linda!Lane!Palo Alto Bicycles!2344 University!!Palo Alto!CA!94301!
        (415)323-5400
```

## UNLOAD TO 文件

通过 ***filename*** 参数指定的 UNLOAD TO 文件接收检索到的行。

您可使用 UNLOAD TO 文件作为对 LOAD 语句的输入。

UNLOAD TO 在缺省的 U.S. English 语言环境中的数据格式

在缺省的语言环境中，在 UNLOAD TO 文件中，数据值有这些格式。

| 数据类型 | 输出格式 |
|---|---|
| BOOLEAN | BOOLEAN 值显示为 t 表示 TRUE，或显示为 f 表示 FALSE。 |
| 字符 | 如果字符字段包含定界符，则 GBase 8s 产品自动地用反斜杠（\）转义它，以防止翻译为特殊字符。在 UNLOAD TO 文件中，字面的反斜杠字符显示为两个连续的反斜杠字符（\\）。如果您使用 DB-Access 的 LOAD 语句来将行插入到表内，则自动地从那个表除去反斜杠。 |
| Collections | 卸载的集合的值括在大括号（{}）之间，每个元素之间有定界符。要获取更多信息，请参阅 卸载复杂类型。 |
| DATE | DATE 值表示为 *mm/dd/yyyy*（或数据库语言环境的缺省的格式），在此，*mm* 为月份（January = 1，依此类推），*dd* 为日期，*yyyy* 为年份。如果您已设置了 GL_DATE 或 DBDATE 环境变量，则 UNLOAD 语句为 DATE 值使用指定的数据格式。 |
| DATETIME、INTERVAL | 字面的 DATETIME 和 INTERVAL 值显示为数字和定界符，不带有关键字限定符，以缺省的格式 *yyyy-mm-dd hh:mi:ss.fff*。省略声明的精度之外的时间单位。如果设置 GL_DATETIME 或 DBTIME 环境变量，则 DATETIME 值以指定的格式显示。 |
| DECIMAL、MONEY | 不带有主要货币符号的卸载的值。在缺省的语言环境中，逗号（,）是千分隔符，而句号（.）是小数分隔符。如果设置 DBMONEY，则对于 MONEY 值，UNLOAD 使用它的指定的分隔符和货币格式。 |
| NULL | NULL 显示为在其间不带有字符的两个定界符。 |

| 数据类型 | 输出格式 |
| --- | --- |
| 数值 | 值显示为前面没有空格的文字。对于 BIGINT、INTEGER、INT8 和 SMALLINT 数据类型，零显示为 0，对于 MONEY、FLOAT、SMALLFLOAT 和 DECIMAL 数据类型，零显示为 0.0。 |
| ROW 类型（命名的和未命名的） | ROW 类型将它的值括在圆括号之间，且以字段定界符分隔每一元素。要获取更多信息，请参阅 卸载复杂类型。 |
| 简单大对象（TEXT、BYTE） | 直接将 TEXT 和 BYTE 列卸载到 UNLOAD TO 文件内。BYTE 值出现在 ASCII 十六进制表中，不带有添加的空白或换行符。要获取更多信息，请参阅 卸载简单大对象。 |
| 智能大对象（CLOB、BLOB） | 将 CLOB 和 BLOB 列卸载到客户端计算机上的单独的操作系统文件（对于每一列）内。UNLOAD TO 文件中的 CLOB 或 BLOB 字段包含此文件的名称。要获取更多信息，请参阅 卸载智能大对象。 |
| 用户定义的数据类型（opaque 类型） | Opaque 类型必须定义有 **export( )** 支持函数。它们需要特殊的处理来从 opaque 数据类型的内部格式复制到 UNLOAD TO 文件格式。对于二进制格式的数据，可能还需要 **exportbinary( )** 支持函数。UNLOAD TO 文件中的数据可能对应于 **export( )** 或 **exportbinary( )** 支持函数返回的格式。 |

**在非缺省的语言环境中的 UNLOAD TO 数据格式**

在非缺省的语言环境中，DATE、DATETIME、MONEY 和数值列值有语言环境对这些数据类型支持的格式。要获取更多信息，请参阅 *GBase 8s GLS 用户指南*。要获取更多关于 **DBDATE**、**DBMONEY** 和 **DBTIME** 环境变量的信息，请参考 *《GBase 8s SQL 指南：参考》*。

在使用非缺省的语言环境的数据库中，如果 **GL_DATETIME** 环境变量有非缺省的设置，则在 UNLOAD 语句可正确地将本地化的 DATETIME 值正确地从数据库表，或从视图，或从通过 EXTERNAL TABLE 语句定义的表对象卸载之前，必须将 **USE_DTENV** 环境变量设置为 1。要获取更多关于 **GL_DATETIME**、**GL_DATE** 和 **USE_DTENV** 环境变量的信息，请参考 *GBase 8s GLS 用户指南*。

## 卸载字符列

在包含 VARCHAR 或 NVARCHAR 列的卸载文件中，在 VARCHAR、LVARCHAR 或 NVARCHAR 字段中保留结尾空格。当卸载 CHAR 或 NCHAR 列时，丢弃这些结尾空格。

对于 CHAR、VARCHAR、NCHAR 和 NVARCHAR 列，空字符串（长度为零，不包含字符的数据字符串）出现在 UNLOAD TO 文件中，为四个字节 "|\ |"（定界符、反斜杠、空格、定界符）。

GBase 8s 数据库服务器的一些较早版本使用了 "||"（连续的定界符）来表示 LOAD 和 UNLOAD 操作中的空字符串。然而，在此版本中，"||" 仅表示 CHAR、VARCHAR、LVARCHAR、NCHAR 和 NVARCHAR 列中的 NULL 值。

## 卸载简单大对象

数据库服务器将 BYTE 和 TEXT 值直接写到 UNLOAD TO 文件内。以十六进制转储格式写 BYTE 值，无添加的空格和换行字符。因此，包含 BYTE 数据的 UNLOAD TO 文件的逻辑长度可能长且难以打印和编辑。

如果您正在卸载包含简单大对象数据类型的文件，请不要使用在 UNLOAD TO 文件中可能作为定界符出现在 BYTE 或 TEXT 值中的那些字符。另请参阅 DELIMITER 子句 部分。

数据库服务器为 TEXT 数据处理任何需要的代码集转换，请参阅 *GBase 8s GLS 用户指南*。

如果您正在卸载包含简单大对象数据类型的文件，则将小于 10 KB 的对象临时地存储在内存中。您可以 **DBBLOBBUF** 环境变量将 10 KB 设置调整到较大的设置。将大于缺省的或 **DBBLOBBUF** 设置的 BYTE 或 TEXT 值存储在临时文件中。要获取关于 **DBBLOBBUF** 的附加信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 卸载智能大对象

数据库服务器为每一列将智能大对象（从 BLOB 或 CLOB 列）卸载到单独的操作系统文件内，与 UNLOAD TO 文件在客户端计算机上的同一个目录中。 将同一列中所有智能 blob 存储在单个文件中。filename 有下列之一的格式：

- 对于 BLOB 值： `blob########`
- 对于 CLOB 值： `clob########`

在前面的格式中，符号（#）表示该文件中第一个智能大对象的唯一十六进制智能大对象的位数。智能大对象标识符的数字的最大数目为 17。然而，大多数智能大对象可能有更少位数的标识符。

当数据库服务器卸载第一个智能大对象时，它以智能大对象的十六进制标识符来创建适当的 BLOB 或 CLOB 客户端文件。如果在同一列中出现附加的智能大对象值，则数据库服务器将它们的值写到同一文件，并对于该文件中的每一 BLOB 或 CLOB 值，在 UNLOAD TO 文件（***.unl**）中罗列 sbspace、chunk 和页编号，以及智能大对象标识符。

下列示例展示来自同一列的两个智能大对象值的 UNLOAD TO 文件条目：

```
Object # 1
        Space Chunk Page = [5,6,3] Object ID = 1192071051

        Object #2
        Space Chunk Page = [5,6,4] Object ID = 1192071050

        both rows unloaded
```

在 UNLOAD TO 文件中，BLOB 或 CLOB 列值以此格式出现：

***start_off*, *length*, *client_path***

在此，*start_off* 是在客户端文件内智能大对象的开始偏移量(以十六机制格式)，*length* 是 BLOB 或 CLOB 值的长度（以十六进制），*client_path* 是客户端文件的 pathname。在这些值之间不可出现空格。 例如，如果 CLOB 值为 512 字节长，且在 /usr/apps/clob9ce7.318 文件中的偏移量为 256，则在 UNLOAD TO 文件中，CLOB 值显示如下：

|100,200,/usr/apps/clob9ce7.318|

如果 BLOB 或 CLOB 列值占据整个客户端文件，则在 UNLOAD TO 文件中，CLOB 或 BLOB 列值显示如下：

***client_path***

例如，如果 CLOB 值占据整个文件 /usr/apps/clob9ce7.318，则在 UNLOAD TO 文件中，CLOB 值显示如下：

|/usr/apps/clob9ce7.318|

对于支持多字节代码集的语言环境，请确信接收字符数据的任何列的声明的大小（以字节为单位）足够大，可以存储整个数据字符串。对于某些语言环境，这可需要高达数据字符串中逻辑字符数目的 4 倍。

数据库服务器为 CLOB 数据处理任何需要的代码集转换。要获取更多信息，请参阅 *GBase 8s GLS 用户指南*。

## 卸载复杂类型

在 UNLOAD TO 文件中，复杂数据类型的值显示如下：

- 以适当的构造函数（MULTISET、LIST、SET）引入集合，以逗号分隔的它们的元素括在大括号（{ }）中：

  ***constructor*{*val1* , *val2* , ... }**

  例如，要从 SET (INTEGER NOT NULL) 数据类型的列卸载 SET 值 {1, 3, 4}，UNLOAD TO 文件的相应的字段显示如下：

  |SET{1,3,4}|

- 通过 ROW 构造函数引入 ROW 类型（命名的和未命名的），且将它们的字段括在圆括号之间，并以逗号分隔：

  ROW(***val1* , *val2* , ...** )

  例如，要卸载 ROW 值 (1, 'abc')，UNLOAD TO 文件的对应的字段显示如下：

  |ROW(1,abc)|

## DELIMITER 子句

使用 DELIMITER 子句来指定分隔在输出文件中的行中每一列中包含的数据的定界符。

*如果您省略此子句，则 DB-Access 检查 DBDELIMITER 环境变量的设置。如果尚未设置 DBDELIMITER，则缺省的定界符是管道（|）符号。您可指定 TAB（CTRL-I）或空格（ASCII 32）作为定界符，但在任何语言环境中，下列字符都不是有效的定界符：*

- 反斜杠（\）
- 换行字符（CTRL-J）
- 十六进制数字（0 至 9，a 至 f，A 至 F）

反斜杠（\）不是有效的字段分隔符或记录定界符，因为它是缺省的转义字符，表明数据中的下一个字符是文字字符，而不是特殊字符。然而，如果您通过设置 DEFAULTESCCHAR 配置参数或 DEFAULTESCCHAR 会话环境选项来更改缺省的转义字符，则您可使用反斜杠作为字段分隔符。

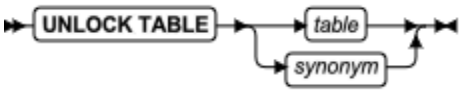
*下列 UNLOAD 语句指定分号（;）作为定界符：*

```
UNLOAD TO 'cust.out' DELIMITER ';'
        SELECT fname, lname, company, city FROM customer;
```

## 2.153 UNLOCK TABLE 语句

在不支持事务日志记录的数据库中，使用 UNLOCK TABLE 语句来解除您先前以 LOCK TABLE 语句锁定的表的锁定。UNLOCK TABLE 语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *synonym* | 要解除锁定的表的同义词 | 该同义词及它指向的表必须存在 | 数据库对象名 |
| *table* | 要解除锁定的表 | 必须在不带有事务日志记录的数据库中，且必须是您先前锁定的表 | 数据库对象名 |

用法

**限制：** UNLOCK TABLE 语句在事务内是无效的。

如果您拥有该表，或如果您在表上有 Select 权限，则您可从对您的用户的直接授权或从对 PUBLIC 的授权来锁定表。您仅可解除您锁定了的表的锁定。你不可解除另一进程锁定了的表的锁定。一次仅将一个锁应用于表。

您必须指定您正在解除锁定的表的名称或同义词。请不要指定视图或视图的同义词的名称。

要更改不带有事务日志记录而创建了的数据库中表的锁定模式，请使用 UNLOCK TABLE 语句来解除该表的锁定，然后发出新的 LOCK TABLE 语句。下列示例展示如何更改不带有事务日志记录而创建了的数据库中表的锁定模式：

```
LOCK TABLE items IN EXCLUSIVE MODE;
        ...
 UNLOCK TABLE items;
        ...
 LOCK TABLE items IN SHARE MODE;
```

如果在事务内发出 UNLOCK TABLE 语句，则它会失败。当事务完成时，自动地释放在事务内设置的表锁。

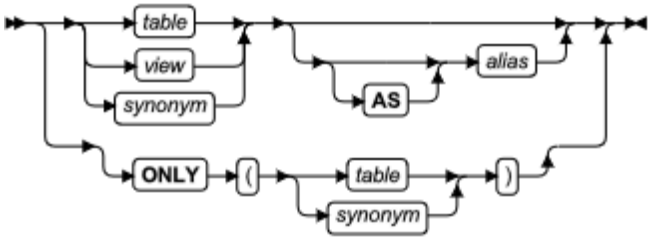
如果您正在使用符合 ANSI 的数据库，请不要发出 UNLOCK TABLE 语句。如果在事务内发出 UNLOCK TABLE 语句，且在符合 ANSI 的数据库中事务总是生效的，则该语句失败。

## 2.154 UPDATE 语句

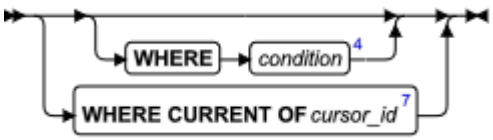
使用 UPDATE 语句来更改表或视图中一个或多个现有的行的一个或多个列中的值。

**语法**

Target




WHERE 选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 您在此为本地表或远程表声明的临时的名称 | 如果 SET 是 *alias* 的标识符，则 AS 关键字必须在 *alias* 之前 | 标识符 |
| *condition* | 被更新的行必须满足的逻辑标准 | 不可为 UDR 或相关联的子查询 | 条件 |
| *cursor* | 要更新其当前行的游标的名称 | 不可为主变量。您不可更新包括合计的行 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *synonym*、*table*、*view* | 包含要被更新的行的同义词、表或视图 | *synonym* 和表或它指向的视图必须存在 | 数据库对象名 |

**用法**

使用 UPDATE 语句来更新任何下列类型的数据库对象或程序对象：

- 表中的一行：单个行、行的组或表中的所有行
- 集合数据类型的列中的元素
- 在命名的或未命名的 ROW 数据类型的列中，一个字段或所有字段。

随同 GBase 8s ，您还可使用此语句来更改在 GBase 8s ESQL/C 或 SPL 集合变量或 ROW 变量中一个或多个元素的值。

要获取关于如何更新集合变量的元素的信息，请参阅 集合派生表。在此 UPDATE 语句的描述中接下来的部分描述如何更新表中的行。

您必须或拥有该表，或对该表有 Update 权限。请参阅 GRANT 语句。要更新视图中的数据，您必须有 Update 权限，且该视图必须满足 通过视图更新行 中说明的要求。

UPDATE 语句的目标不可为 CREATE EXTERNAL TABLE 语句定义了的表对象。

游标（如 DECLARE 语句的 SELECT ... FOR UPDATE 部分中定义的那样）可仅包含列名称。如果您省略 WHERE 子句，则更新目标表的所有行。

如果您正在使用有效的检查，且检查模式设置为 IMMEDIATE，则在每一 UPDATE 语句的结尾处检查所有启用的约束。如果检查模式设置为 DEFERRED，则直到提交事务才检查所有启用的约束。

在 DB-Access 中，如果您省略 WHERE 子句，且处于交互的模式中，则 DB-Access 不运行 UPDATE 语句，直到您确认您想要更改所有行为止。然而，如果该语句在命令文件中，且您在在命令行运行，则立即执行该语句。

**示例**

下列示例创建并更新视图。

```
CREATE VIEW cust_view AS SELECT * FROM customer;
UPDATE cust_view SET customer_num=10001 WHERE customer_num=101;
```

## 使用 ONLY 关键字

如果您使用 UPDATE 语句来更新超级表的行，则还可更新来自它的子表的行。要仅更新来自超级表的行，请在表名称之前使用 ONLY 关键字，如此示例所示：

```
UPDATE ONLY(am_studies_super)
        WHERE advisor = "johnson"
        SET advisor = "camarillo";
```

**注：** 如果您在超级表上使用不带有 ONLY 关键字且不带有 WHERE 子句的 UPDATE 语句，则更新超级表的所有行和它的子表。如果您计划使用 WHERE CURRENT OF 子句来更新游标的活动集的当前行，则您不可使用 ONLY 关键字。

## 通过视图更新行

如果您在视图上有 Update 权限，则您可通过*单表*视图更新数据（请参阅 GRANT 语句）。对于可更新的视图，定义该视图的查询必须不包含任何下列项：

- projection 列表中聚集值的列
- projection 列表中使用 UNIQUE 或 DISTINCT 关键字的列
- GROUP BY 子句
- UNION 运算符

此外，如果在有列的派生值的表上构建视图，则不可通过该视图更新那列。然而，可更新该视图中的其他列。在可更新的视图中，您可通过将值插入到该视图内来更新基础表中的值。

```
CREATE VIEW cust_view AS SELECT * FROM customer;
  UPDATE cust_view SET customer_num=10001 WHERE customer_cum=101;
```

下列语句定义包括 customer 表中所有行的视图，并将列值为 101 的任何行中的 customer_num 值更改为 10001：

```
CREATE VIEW cust_view AS SELECT * FROM customer;
 UPDATE cust_view SET customer_num=10001 WHERE customer_num=101;
```

当更新值不适合定义了该视图的 SELECT 语句时，您可使用数据完整性约束来防止用户更新基础表中的值。要获取更多信息，请参阅 WITH CHECK OPTION 关键字。

由于即使视图的基本表有唯一的行，在视图仍可发生重复的行，因此当通过视图更新表时请小心。例如，如果在 items 表上定义视图，且仅包含 order_num 和 total_price 列，且如果来自同一订单的两项有相同的总价，则该视图包含重复的行。在此情况下，如果您更新两个重复的 total_price 值中的一个，则您无法知道更新哪个项价格。

**重要：** 如果您正在使用带有检查点的视图，则您不可更新远程表中的行。

以 UPDATE 语句直接修改视图中的数据值的替代方法，是在该视图上创建 INSTEAD OF 触发器。要获取更多信息，请参阅 视图上的 INSTEAD OF 触发器。

## 在没有事务的数据库中更新行

如果您正在在没有事务的数据库中更新行，则您必须采取显式的活动来恢复更新了的行。例如，如果在更新某些行之后 UPDATE 语句失败，则成功地更新了的行保留在表中。您不可自动地从失败的更新中恢复。

## 在带有事务的数据库中更新行

如果您正在带有事务的数据库中更新行，且您正在使用事务，则您将可使用 ROLLBACK WORK 语句撤销更新。如果在更新之前您未执行 BEGIN WORK 语句，且更新失败，则数据库服务器自动地回滚自从更新操作开始以来对该表进行的任何数据库修改。

CREATE TEMP TABLE 语句可包括 WITH NO LOG 选项来创建不支持事务日志记录的临时表。这些表不进行日志记录且不可恢复。

请不要在事务内使用 RAW 表。您以 CREATE RAW TABLE 语句创建的表不进行日志记录。因此，RAW 表是不可恢复的，即使数据库使用日志恢复。RAW 表通常用作初始加载和验证数据。在您已加载了数据之后，请使用 ALTER TABLE 语句来将该表更改为类型 STANDARD，并在您在事务中使用该表之前执行 0 级备份。要获取更多关于 RAW 表的信息，请参考 *GBase 8s 管理员指南*。

在符合 ANSI 的数据库中，事务是隐式的，且所有数据库修改都发生在事务内。在此情况下，如果 UPDATE 语句失败，您可使用 ROLLBACK WORK 来撤销更新。

如果您在显式的事务内，且更新失败，则数据库服务器自动地撤销该更新的影响。

## 锁定注意事项

当带着更新的意图选择行时，该更新进程需要 update 锁。update 锁允许其他进程读或***共享***将要被更新的行，但它们不允许那些进程更新或删除它。

就在更新发生之前，更新进程将共享锁***提升***为排他锁。排他锁防止其他进程读取或修改行的内容，直到释放该锁为止。

更新进程可从另一进程在行上或在有共享锁的页上获得 update 锁，但您不可将 update 锁从共享的提升到排他的（且不可发生更新），直到其他进程释放它的锁为止。

如果单个更新影响的行的数目很大，则您可超过在同时发生的锁的最大数目上设置的限制。如果发生此情况，则您可减少每 UPDATE 语句的事务数，或您可在执行该语句之前锁定页或整个表。

## 为目标表声明别名

您可为目标表声明别名。该别名可引用本地或远程表、视图或同义词的完全符合条件的数据库对象。

别名是不注册在数据库的系统目录中的临时名称，仅在 UPDATE 语句正在运行时保持。

如果您声明作为别名的名称还是 UPDATE 语句的关键字，则您必须使用 AS 关键字来阐明语法：

```
UPDATE stock AS set
        SET unit_price = unit_price * 0.94;
```

下列 UPDATE 语句引用在目标子句中和在两个子查询中表的符合要求的名称：

```
UPDATE nmosdb@wnmserver1:test
        SET name=(SELECT name FROM test
        WHERE test.id = nmosdb@wnmserver1:test.id)
        WHERE EXISTS(
        SELECT 1 FROM test WHERE test.id = nmosdb@wnmserver1:test.id
        );
```

下一 UPDATE 语句在逻辑上等同于前一示例，但为符合条件的表名称声明 r_t 别名：

```
UPDATE nmosdb@wnmserver1:test r_t
        SET name=(SELECT name FROM test
```

```
WHERE test.id = r_t.id)
WHERE EXISTS(
SELECT 1 FROM test WHERE test.id = r_t.id
);
```

声明表别名会简化上述第二个示例的标记。

在 Oracle 模式下，保持 GBase 8s 原有声明别名语法基础上，如果 UPDATE 语句的 SQL 关键字包括 NAME、TEMP、ARRAY、LIST、REVERSE、CONTEXT、LENGTH、LOG 作为表或视图别名使用，可以不用关键字 AS 开始它的声明。

例如，在 UPDATE 语句中使用关键字 temp 作为表 customer 别名：

```
UPDATE customer temp SET temp.col = 1;
```

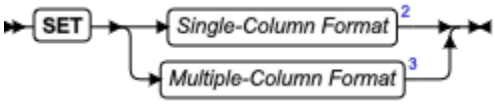
## SET 子句

使用 SET 子句来标识要更新的列并将值指定给每一列。

SET 子句支持下列语法格式：

- 单列格式，它将每一列与单个表达式配对
- 多列格式，它将多列的列表与通过一个或多个表达式返回的值相关联

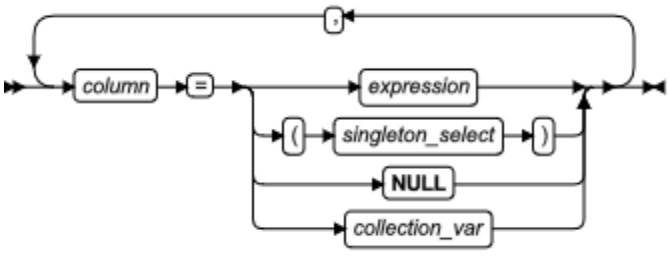
SET 子句




## 单列格式

使用单列格式来将一列与单个表达式配对。

单列格式




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 要被更新的列 | 不可为序列数据类型 | 标识符 |
| *collection_var* | 主变量或程序变量 | 必须声明作为集合数据类型 | 特定于语言 |
| *expression* | 为 *column* 返回一值 | 不可包含聚集函数 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *singleton _select* | 正好返回一行的子查询 | 返回的子查询值必须与 *column* 列表一一对应 | SELECT 语句 |

您可使用此语法来更新有 ROW 数据类型的列。

您可包括任意数量的 "single ***column*** = single ***expression***" 词语。***expression*** 可为返回单个行的 SQL 子查询（括在圆括号之间），假如对应的 ***column*** 为可从子查询返回的行存储该值（或值的集合）的数据类型。

要在 SET 子句中指定 ROW 类型列的值，请参阅 更新 ROW 类型列。下列示例说明 SET 子句的单列格式。

```
UPDATE customer
    SET address1 = '1111 Alder Court', city = 'Palo Alto',
        zipcode = '94301' WHERE customer_num = 103;

UPDATE stock
    SET unit_price = unit_price * 1.07;
```

## 使用子查询来更新单列

您可以子查询返回的值更新在 SET 子句中指定的列。

```
UPDATE orders
            SET ship_charge =
            (SELECT SUM(total_price) * .07 FROM items
            WHERE orders.order_num = items.order_num)
            WHERE orders.order_num = 1001;
```

如果您正在更新表层级中的超级表，则 SET 子句不可包括引用子表的子查询。如果您正在更新表层级中的子表，则 SET 子句中的子查询可引用超级表，如果它仅引用超级表的话。也就是说，子查询必须使用 SELECT … FROM ONLY (***supertable***) 语法。

## 使用相关子查询来更新单列

在 *singleton _select* 中使用相关子查询语法，SET 子句中支持被更新的表与自身做关联，将满足条件的值更新在 SET 子句中指定的列，其他使用方式、约束限制与“使用子查询更新单列”保持一致。

例如，modeinfo 表做数据更新，在 SET 子句中使用相关子查询时，modeinfo 表与自身做关联更新单列：

```
create table modeinfo (id int,modecode varchar(100));
insert into modeinfo values(1,NULL);
insert into modeinfo values(2,'xxxxxxx');
insert into modeinfo values(3,'AAAAAAA');
```

```
insert into modeinfo values(4,'BBBBBBBB');
insert into modeinfo values(5,' ');

update modeinfo t set t.modecode = (select sys_guid() from    modeinfo t1 where t1.id = t.id);

select * from modeinfo ;
ID          MODECODE
1           4A851E01A32649F49E3F574125B9506A
2             EABC52A208914F2A860A89A668E2D654
3             0FCBCF3A4AC349E79763D7F7E3A435F6
4           A322149A9350462ABD78473BA69C509D
5           A8A47810A44E44D8A829E69D024742F0
```

注：UPDATE SET 指定的修改列如果在相关子查询中发生自循环写，数据库将返回 360 错误信息。

## 将列更新为 NULL

当您使用 UPDATE 语句时，请使用 NULL 关键字来修改列值。例如，对于其先前的地址需要两个地址行但现在仅需要一个的客户，您可以使用下列条目：

```
UPDATE customer
        SET address1 = '123 New Street',
        SET address2 = null,
        city = 'Palo Alto',
        zipcode = '94303'
        WHERE customer_num = 134;
```

## 两次更新同一列

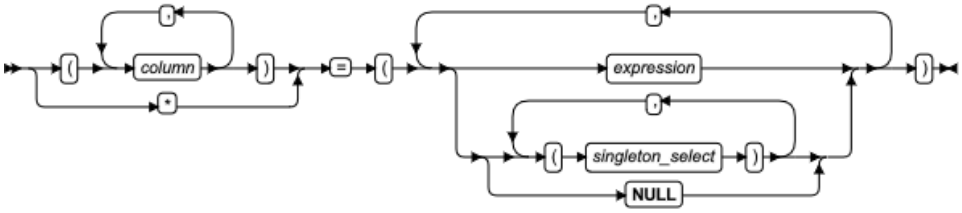
您可在 SET 子句中指定同一列一次以上。如果您这么做，则将该列设置为您为该列指定的最后的值。在下一示例中，fname 列在 SET 子句中出现两次。对于客户编号为 101 的行，用户先将 fname 设置为 gary 然后又设置为 harry。在该 UPDATE 语句执行之后，fname 的值为 harry。

```
UPDATE customer
        SET fname = "gary", fname = "harry"
        WHERE customer_num = 101;
```

## 多列格式

使用 SET 子句的多列格式来罗列多个列并将它们设置等于相应的表达式。

多列格式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 要被更新的列的名称 | 不可为序列类型或 ROW 类型。*column* 名称的数目必须等于返回到 = 号右边的值的数目。 | 标识符 |
| *expression* | 为 *column* 返回值的表达式 | 不可包括聚集函数 | 表达式 |
| *singleton_select* | 正好返回一行的子查询 | 子查询返回的值必须对应于 *column* 列表中的列 | SELECT 语句 |
| *SPL function* | 返回一个或多个值的 SPL 例程 | 返回的值必须与 *column* 列表中的列一一对应 | 标识符 |

SET 子句的多列格式为罗列您想要更新的列的集合提供下列选项：

- 显式地罗列每一列，在列之间放置逗号，将列的集合括在圆括号之间。
- 通过使用星号（*）隐式地罗列表中的所有列。

您必须显式地罗列每一表达式，在表达式之间放置逗号（,）分隔符，并将表达式的集合括在圆括号之间。列的数目必须等于表达式列表所返回的值的数目，除非该表达式列表包括一 SQL 子查询。

下列示例展示 SET 子句的多列格式：

```
UPDATE customer
    SET (fname, lname) = ('John', 'Doe') WHERE customer_num = 101;

UPDATE manufact
    SET * = ('HNT', 'Hunter') WHERE manu_code = 'ANZ';
```

## 使用子查询来更新多列值

该表达式列表可包括一个或多个子查询。每一必须返回包含一个或多个值的单个行。SET 子句显式地或隐式地指定的列的数目必须等于在多列 SET 子句中等号（=）之后的表达式（或表达式列表）所返回的值的数目。

必须将子查询括在圆括号之间。这些圆括号嵌套在紧跟在等号（=）之后的圆括号之内。如果表达式列表包括多个子查询，则必须将每一子查询括在圆括号之间，以逗号（,）分隔连续的子查询：

UPDATE ... SET ... = ((***subquery***A),(***subquery***B), ... (***subquery***N))

下列示例展示 SET 子句中子查询的使用：

```
UPDATE items
    SET (stock_num, manu_code, quantity) =
    ( (SELECT stock_num, manu_code FROM stock
    WHERE description = 'baseball'), 2)
    WHERE item_num = 1 AND order_num = 1001;

UPDATE table1
  SET (col1, col2, col3) =
   ((SELECT MIN (ship_charge), MAX (ship_charge) FROM orders), '07/01/2007')
   WHERE col4 = 1001;
```

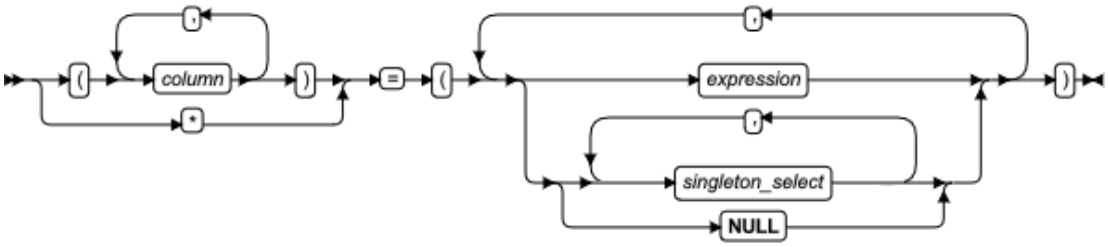
如果您正在更新表层级中的超级表，则 SET 子句不可包括引用它的子表之一的子查询。如果您正在更新表层级中的子表，则 SET 子句中的子查询可引用超级表，如果它仅引用超级表的话。也就是说，子查询必须使用 SELECT… FROM ONLY (***supertable***) 语法。

## 使用相关子查询来更新多列值

在 gbase 模式下，在 *singleton _select* 中使用相关子查询语法，SET 子句中支持被更新的表与自身做关联，将满足条件的值更新在 SET 子句中指定的列，其他使用方式、约束限制与“使用子查询更新多列”保持一致。。

在 oracle 模式下，与 gbase 模式相比，语法使用区别为去掉 *singleton _select* 外层“()”，其他使用方式、约束限制与 gbase 模式保持一致。

Multiple-Column Format::=




例如，modeinfo 表做数据更新，在 SET 子句中使用相关子查询时，modeinfo 表与自身做关联更新多列：

```
create table modeinfo (id int,modecode varchar(100),curtime DATETIME YEAR TO
FRACTION(5));

insert into modeinfo values(1,NULL,SYSDATE);

insert into modeinfo values(2,'xxxxxxx',SYSDATE);

insert into modeinfo values(3,'AAAAAAA',SYSDATE);

insert into modeinfo values(4,'BBBBBBBB',SYSDATE);

insert into modeinfo values(5,' ',SYSDATE);
```

```
update modeinfo t set (t.modecode,curtime) = (select sys_guid(),SYSDATE from    modeinfo
t1 where t1.id = t.id);


select * from modeinfo ;

ID          MODECODE

1           61F45F9B522744528B2EDFB4F6F36D09
       2023-03-13 10:45:20.00000

2             10321386DA304E57BAEBDD7CC4E7A4CE
       2023-03-13 10:45:20.00000

3             98DA4D553828472395D6B9EABA3F50A0                                    2023-03-13 10

4           EF3F8CF8008C451B9DB5E4067BBCEB90                    2023-03-13
10:45:20.00000

5           4520C874930243AE843CA0EEE2CDA591                    2023-03-13 10:45:20.00000
```

注：UPDATE SET 指定的修改列如果在相关子查询中发生自循环写，数据库将返回 360 错误信息。

## 更新 ROW 类型列

使用 SET 子句来更新命名的或未命名的 ROW 类型列。例如，假设您定义下列命名的 ROW 类型和包含命名的和未命名的 ROW 类型的列的表：

```
CREATE ROW TYPE address_t
        (
        street CHAR(20), city CHAR(15), state CHAR(2)
        );
        CREATE TABLE empinfo
        (
        emp_id INT
        name ROW ( fname CHAR(20), lname CHAR(20)),
        address address_t
        );
```

要更新未命名的 ROW 类型，请在圆括号括起的字段值的列表之前指定 ROW 构造函数。

下列语句更新 empinfo 表的 name 列（未命名的 ROW 类型）：

```
UPDATE empinfo SET name = ROW('John','Williams') WHERE emp_id =455;
```

要更新命名的 ROW 类型，请在字段值的列表（在圆括号中）之前指定 ROW 构造函数，并使用强制转型（::）运算符来将 ROW 值强制转型为命名的 ROW 类型。下列语句更新 empinfo 表的 address 列（命名的 ROW 类型）：

```
UPDATE empinfo
        SET address = ROW('103 Baker St','Tracy','CA')::address_t
        WHERE emp_id = 3568;
```

要获取更多关于 ROW 构造函数的语法，请参阅 构造函数表达式。另请参阅 Literal Row。

ROW 列 SET 子句仅可支持字段的文字值。要使用 ESQL/C 变量来指定字段值，您必须将 ROW 数据选择到 **row** 变量内，使用单独的字段值的主变量，然后以 **row** 变量更新 ROW 列。要获取更多信息，请参阅 更新 Row 变量（ESQL/C）。

您可使用 GBase 8s ESQL/C 主变量来插入***非文字***值作为：

- 整个 row 类型插入列内

  使用 **row** 变量作为 SET 子句中的变量名称来一次更新 ROW 列中的所有字段。

- ROW 类型的单独字段

  要将非文字值插入 ROW 类型列内，您可首先更新 **row** 变量中的元素，然后指定在 UPDATE 语句的 SET 子句中的 **collection** 变量。

当您使用 SET 子句中的 **row** 变量时，**row** 变量必须包含每一字段值的值。要获取关于如何将值插入到 **row** 变量内的信息，请参阅 更新 Row 变量（ESQL/C）。

您可使用 UPDATE 语句来只修改行中的某些字段：

- 以字段 projection 指定其值保持不变的所有字段的字段名称

  例如，下列 UPDATE 语句仅更改 **empinfo** 表的 **address** 列的 **street** 和 **city** 字段：

  ```
  UPDATE empinfo
  SET address = ROW('23 Elm St', 'Sacramento', address.state)
  WHERE emp_id = 433;
  ```

  **address.state** 字段保持不变。

- 将该行选择到 ESQL/C **row** 变量内并更新期望的字段。

  要获取更多信息，请参阅 更新 Row 变量（ESQL/C）。

## 更新集合列

您可使用 SET 子句来更新集合列中的值。要获取更多信息，请参阅 集合构造函数。

集合变量可更新集合类型列。以集合变量，您可插入一个或多个集合的单独元素。要获取更多信息，请参阅 集合派生表。

例如，假设您定义 tab1 表如下：

```
CREATE TABLE tab1
        (
        int1 INTEGER,
        list1 LIST(ROW(a INTEGER, b CHAR(5)) NOT NULL),
        dec1 DECIMAL(5,2)
        );
```

下列 UPDATE 语句更新 tab1 中的一行：

```
UPDATE tab1
```

```
SET list1 = LIST{ROW(2, 'zyxwv'),
ROW(POW(2,6), '=64'),
ROW(ROUND(ROOT(146)), '=12')},
WHERE int1 = 10;
```

在此示例中的集合列 **list1** 有三个元素。每一元素是带有 INTEGER 字段和 CHAR(5) 字段的未命名的 ROW 类型。第一个元素包括两个文字值：一个整数（2 ）和一个引号括起的字符串（'zyxwv'）。

第二个和第三个元素还使引用字符串来表明第二个字段的值。然而，它们以表达式，而不是以文字值，来指定第一个字段的值。

## 更新 Opaque 类型列中的值

当更新某些 opaque 数据类型时，它们需要特殊的处理。例如，如果 opaque 数据类型包含空间的数据或多重表示的数据，则它可能提供如何存储该数据的选择：在内部的结构之内，还是在智能大对象之中（对于大对象）。

通过调用名为 **assign( )** 的用户定义的支持函数来完成此处理。当您在其行包含这些 opaque 类型之一的表上执行 UPDATE 时，数据库服务器自动地调用该类型的 **assign( )** 函数。此函数可作出如何存储该数据的决定。要获取更多关于 assign( ) 支持函数的信息，请参阅 GBase 8s 用户定义的例程和数据类型开发者指南 。

## 分布式 UPDATE 操作中的数据类型

访问另一 GBase 8s 实例的数据库的 UPDATE 语句（或任何其他 SQL 数据操纵语言语句）仅可引用下列数据类型：

- 内建的非 opaque 的数据类型
- BOOLEAN
- LVARCHAR
- 非 opaque 的内建的数据类型的 DISTINCT
- BOOLEAN 的 DISTINCT
- LVARCHAR 的 DISTINCT
- 出现在此列表中的任何 DISTINCT 数据类型的 DISTINCT。

跨服务器分布式 UPDATE 操作可支持这些 DISTINCT 类型，仅当将该 DISTINCT 类型显式地强制转型为内建的类型，且所有的 DISTINCT 类型、 它们的数据类型层级以及它们的强制转型都正好以相同的方式定义在每一参与的服务器中。 要获取关于在跨服务器 DML 操作中 GBase 8s 支持的数据类型的信息，请参阅 跨服务器事务中的数据类型。

然而，访问本地 GBase 8s 实例的其他数据库的跨数据库分布式 UPDATE 操作可访问在前面的列表中的跨服务器数据类型，还有下列数据类型：

- 大多数***内建的 opaque 数据类型***，如 跨数据库事务中的数据类型 中罗列的那样
- 在前面的行中引用的内建的类型的 DISTINCT
- 在前面的两行中的一行中罗列的任何数据类型的 DISTINCT

- 显式地强制转型到内建的数据类型的 opaque 用户定义的数据类型（UDT）。

跨数据库 UPDATE 操作可支持这些 DISTINCT 类型和 opaque UDT，仅当将所有 opaque UDT 和 DISTINCT 类型强制转型到内建的类型，且所有的 opaque UDT、DISTINCT 类型、数据类型层级和强制转型都正好以同样的方式定义在每一参与的数据库中。

分布式 UPDATE 事务不可访问另一 GBase 8s 实例的数据库，除非两个服务器都在它们的 DBSERVERNAME 或 DBSERVERALIASES 配置参数中以及在 **sqlhosts** 文件或 SQLHOSTS 注册子键中定义 TCP/IP 或 IPCSTR 连接。两参与的服务器都要支持相同的连接类型（或 TCP/IP 或 IPCSTR），该要求也适用于 GBase 8s 实例之间的任何通信，即使双方位于同一台计算机上。

## UPDATE 的 WHERE 子句

WHERE 子句让您指定搜索标准来限制要被更新的行。如果您省略 WHERE 子句，则更新表中的每行。要获取更多信息，请参阅 SELECT 的 WHERE 子句。

## 当更新符合 ANSI 的数据库时的 SQLSTATE 值

如果您在符合 ANSI 的数据库中以 UPDATE 语句更新表，其中包含 WHERE 子句且未找到行，则数据库服务器发出警告。

您可以下列方式之一检测此警告条件：

- GET DIAGNOSTICS 语句将 **RETURNED_SQLSTATE** 字段设置为值 `02000`。 在 SQL API 应用中，**SQLSTATE** 变量包含与此相同的值。
- 在 SQL API 应用中，**sqlca.sqlcode** 和 **SQLCODE** 变量包含值 `100`。

数据库服务器还将 **SQLSTATE** 和 **SQLCODE** 设置为这些值，如果 UPDATE ... WHERE 语句是多语句 PREPARE 的一部分且数据库服务器不返回行的话。

## 当更新非 ANSI 数据库时的 SQLSTATE 的值

在不符合 ANSI 的数据库中，当数据库服务器发现没有与 UPDATE 语句的 WHERE 子句相匹配的行时，它不返回警告。**SQLSTATE** 代码为 `00000` 且 **SQLCODE** 代码为零（`0`）。然而，如果 UPDATE ... WHERE 语句是多语句 PREPARE 的一部分，且未返回行，则数据库服务器发出警告，并设置 **SQLSTATE** 为 `02000`，设置 **SQLCODE** 为 `100`。

GBase 8s 的客户端-服服务器通信协议，诸如 SQLI 和 DRDA®，支持 **SQLSTATE** 代码值。要了解这些代码的列表，以及要了解关于如何获得该消息文本的信息，请参阅 使用 SQLSTATE 错误状态代码。

## UPDATE 的 WHERE 子句中的子查询

UPDATE 语句的 WHERE 子句中的子查询的 FROM 子句可指定与 UPDATE 语句的 Table Options 子句指定的同一表或视图作为数据源。仅当下列条件都为真时，才支持带有引用同一表对象的子查询的 UPDATE 操作：

- 该子查询或者返回单个行，或者有不相关的列引用。

- 该子查询在 UPDATE 语句 WHERE 子句中，以 Subquery 语法使用 Condition。
- 在该子查询中没有 SPL 例程可引用 UPDATE 正在修改的同一表。

除非满足所有这些条件，否则包括引用 UPDATE 语句修改的同一表或视图的子查询的 UPDATE 语句返回错误 -360。

下列示例通过将更新 unit_price 值减去价格子集的 5% 来更新 stock 表。WHERE 子句通过将 IN 运算符应用于由子查询返回的行来指定减去哪个价格，仅选择 unit_price 值大于 50 的 stock 表的行：

```
UPDATE stock SET unit_price = unit_price * 0.95
        WHERE unit_price IN
        (SELECT unit_price FROM stock WHERE unit_price > 50);
```

此子查询仅包括不相关的列引用，因为它的唯一的引用的列是在它的 FROM 子句指定的表中。以上罗列的要求生效，因为该子查询的数据源与外部的 UPDATE 语句的 Table Options 子句指定的是相同的 **stock** 表。

前面的示例与发出两个单独的 DML 语句产生相同的结果：

- SELECT 语句，从 **stock** 表返回临时表，**tmp1** 包含的行与子查询返回的行相同。
- UPDATE 语句，发出临时表的子查询作为在它的 WHERE 子句中的断言来修改其 **unit_price** 与临时表中的执行匹配的 **stock** 表的每一行：

```
SELECT unit_price FROM stock WHERE unit_price > 50 INTO TEMP tmp1;
        UPDATE stock SET unit_price = unit_price * 0.95
        WHERE unit_price IN ( SELECT * FROM tmp1 );
```

这里是一个在它的 WHERE 子句中包括多个不相关的子查询的更复杂的 UPDATE 语句的示例：

```
UPDATE t1 SET a = a + 10
        WHERE a > ALL (SELECT a FROM t1   WHERE a > 1) AND
        a > ANY (SELECT a FROM t1   WHERE a > 10) AND
        EXISTS (SELECT a FROM t1   WHERE a > 5);;
```

如果在表上定义启用的 Select 触发器，该表是修改同一表的 UPDATE 语句的 WHERE 子句中的子查询的数据源，则在该 UPDATE 语句之内执行那个子查询不激活该 Select 触发器。请考虑下列程序片断：

```
CREATE TRIGGER selt11 SELECT ON t1 BEFORE
        (UPDATE d1
        SET (c1, c2, c3, c4, c5) =
        (c1 + 1, c2 + 1, c3 + 1, c4 + 1, c5 + 1));

        UPDATE t2 SET c1 = c1 +1
        WHERE c1 IN
        (SELECT t1.c1 from t1 WHERE t1.c1 > 10 );
```

在上述示例中，不作为 **t2** 上 UPDATE 操作的一部分激活表触发器 **selt11**。

UPDATE 语句的 WHERE 子句中的子查询可包括 UNION 或 UNION ALL 运算符，如下例所示。

```
UPDATE t1 SET a = a + 10 WHERE a in (SELECT a FROM t1 WHERE a > 1
        UNION SELECT a FROM t1, t2 WHERE a < b);
```

如果外部的 UPDATE 语句修改的表是表层级之内的类型表，则 GBase 8s 支持使用 UPDATE 的 WHERE 子句中有效子查询的所有下列操作：

- UPDATE 在目标父表上以子查询（SELECT from parent table）
- UPDATE 在目标父表上以子查询（SELECT from child table）
- UPDATE 在目标孩子表上以子查询（SELECT from parent table）
- UPDATE 在目标孩子表上以子查询（SELECT from child table）。

下列程序片断说明以类型表上的子查询的 UPDATE 操作：

```
CREATE ROW TYPE r1 (c1 INT, c2 INT);
        CREATE ROW TYPE r2 UNDER r1;
        CREATE TABLE t1 OF TYPE r1; -- parent table
        CREATE TABLE t2 OF TYPE r2 UNDER t1; -- child   table

        UPDATE t1 SET c1 = c1 + 1 WHERE c1 IN
        ( SELECT t1.c1 FROM t1 WHERE t1.c1 > 10);

        UPDATE t1 SET c1 = c1 + 1 WHERE c1 IN
        ( SELECT t2.c1 FROM t2 WHERE t2.c1 > 10);

        UPDATE t2 SET c1 = c1 + 1 WHERE c1 IN
        ( SELECT t2.c1 FROM t2 WHERE t2.c1 > 10);

        UPDATE t2 SET c1 = c1 + 1 WHERE c1 IN
        ( SELECT t1.c1 FROM t1 WHERE t1.c1 > 10);
```

要获取更多关于如何使用返回多行作为 UPDATE 语句的 WHERE 子句中的断言的信息，请参阅带有子查询的条件 主题。

## UPDATE 的 WHERE 子句中的相关子查询

在 *condition* 中支持使用相关子查询语法，在 WHERE 子句中支持被更新的表与自身做关联，将满足条件的值更新在 SET 子句中指定的列，其他使用方式、约束限制与“UPDATE 的 WHERE 子句中的子查询”保持一致。

例如，modeinfo 表做数据更新，在 WHERE 子句中使用相关子查询时，modeinfo 表与自身做关联更新数据：

```
create table modeinfo (id int,modecode varchar(100),curtime DATETIME YEAR TO
FRACTION(5));

insert into modeinfo values(1,NULL,SYSDATE);

insert into modeinfo values(2,'xxxxxxx',SYSDATE);

insert into modeinfo values(3,'AAAAAAA',SYSDATE);
```

```
insert into modeinfo values(4,'BBBBBBBB',SYSDATE);

insert into modeinfo values(5,' ',SYSDATE);


update modeinfo t set (t.modecode,curtime) = (select sys_guid(),SYSDATE from    modeinfo
t1 where t1.id = t.id) where

exists (select 1 from modeinfo t2 where t2.id = t.id)

and (modecode is null or modecode = ' ');


select * from modeinfo ;

ID          MODECODE                                                                CUR

1           61F45F9B522744528B2EDFB4F6F36D09                        2023-03-13
10:45:20.00000

2             xxxxxxx

3             AAAAAAA

4           BBBBBBBB

5           4520C874930243AE843CA0EEE2CDA591                        2023-03-13
10:45:20.00000
```

注：UPDATE SET 指定的修改列如果在相关子查询中发生自循环写，数据库将返回 360 错误信息。

## 使用 WHERE CURRENT OF 子句（ESQL/C、SPL）

使用 WHERE CURRENT OF 子句来更新 FOR UPDATE 声明了的游标的当前行，或更新 Collection 游标的当前元素。

在此，不可指定***游标***名称做为主变量。

当前行是最近获取的行。由于 UPDATE 语句不会将游标前进到下一行，因此通过此操作不更改在游标的活动集合内当前行的位置。

对于 GBase 8s 的表层级，您不可使用此子句，如果您正在仅从表层级中一个表选择的话。也就是说，如果您使用 ONLY 关键字，则您不可使用此选项。

在 ESQL/C 例程中，要包括 WHERE CURRENT OF 关键字，您必须提前已经使用了 DECLARE 语句来定义带有 FOR UPDATE 选项的***游标***。如果创建了该游标的 DECLARE 语句指定了 FOR UPDATE 中的一个或多个列，则限制您仅可更新随后的 UPDATE ... WHERE CURRENT OF 语句中的那些行。在 DECLARE 语句的 FOR UPDATE 子句中指定列的优势在于速度。如果在 DECLARE 语句中指定列，数据库服务器通常可更快地执行更新。

在 SPL 例程中，您可在 UPDATE 语句中的 WHERE CURRENT OF 关键字之后指定游标，仅当您在 SPL 的 FOREACH 语句中声明了 ***cursor_id***。您不可在 SPL 例程中使用 DECLARE 语句来声

明动态游标的名称，以及将那个游标与 PREPARE 语句已在同一 SPL 例程中声明了的准备好的对象的语句标识符相关联。

**说明：** Update 游标可执行以 UPDATE 语句不可能执行的更新。

下列 GBase 8s ESQL/C 示例说明 WHERE 子句的 CURRENT OF 形式。在此示例中，在收到 10% 折扣的客户的范围内（假设将新列 **discount** 添加到 **customer** 表）执行更新。在 WHILE 循环的外部准备 UPDATE 语句来确保仅执行一次解析。

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
    char fname[32],lname[32];
    int low,high;
EXEC SQL END DECLARE SECTION;
main()
{
    EXEC SQL connect to 'stores_demo';
    EXEC SQL prepare sel_stmt from
        'select fname, lname from customer
         where cust_num between ? and ? for update';
EXEC SQL declare x cursor for sel_stmt;
    printf("\nEnter lower limit customer number: ");
    scanf("%d", &low);
    printf("\nEnter upper limit customer number: ");
    scanf("%d", &high);
    EXEC SQL open x using :low, :high;
    EXEC SQL prepare u from
        'update customer set discount = 0.1    where current of x';
    while (1)
        {
        EXEC SQL fetch x into :fname, :lname;
         if ( SQLCODE == SQLNOTFOUND) break;
         }
    printf("\nUpdate %.10s %.10s (y/n)?", fname, lname);
    if (answer = getch() == 'y')
        EXEC SQL execute u;
    EXEC SQL close x;
}
```

## 更新 Row 变量（ESQL/C）

带有“集合派生的表”段的 UPDATE 语句允许您在 **row** 变量中更新字段。“集合派生的表”段标识要在其中更新字段的 **row** 变量。要获取更多信息，请参阅 集合派生表。

要更新字段

1. 在您的 GBase 8s ESQL/C 程序中创建 **row** 变量。

2. 可选地，以 SELECT 语句（不带有“集合派生的表”段）将 ROW 类型列选择到 **row** 变量内。

3. 以 UPDATE 语句和“集合派生的表”段更新 **row** 变量的字段。

4. 在 **row** 变量包含正确的字段之后，您使用表或视图名称上的 UPDATE 或 INSERT 语句来将 **row** 变量保存在 ROW 列（命名的或未命名的）之中。

UPDATE 语句和“集合派生的表”段允许您在 row 变量中更新字段或字段的组。 请在 SET 子句中指定新的字段值。例如，下列 UPDATE 更改 myrect GBase 8s ESQL/C row 变量中的 x 和 y 字段：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (x int, y int, length float, width float) myrect;
        EXEC SQL END DECLARE SECTION;
        . . .
        EXEC SQL select into :myrect from rectangles where area = 64;
        EXEC SQL update table(:myrect) set x=3, y=4;
```

假设在 SELECT 语句之后，**myrect2** 变量有值 x=0、y=0、length=8 以及 width=8。 在 UPDATE 语句之后，**myrect2** 变量有字段值 x=3、y=4、length=8，以及 width=8。 您不可使用 INSERT 语句的“集合派生的表”段中的 **row** 变量。

然而，您可使用 UPDATE 语句和“集合派生的表”段来将新字段插入到 **row** 主变量之内，如果您为该行中的每个字段都指定值的话。

例如，下列代码片段将新的字段值插入到 row 变量 myrect 内，然后将此 row 变量插入到数据库之内：

```
EXEC SQL update table(:myrect)
                set x=3, y=4, length=12, width=6;
                EXEC SQL insert into rectangles
                  values (72, :myrect);
```

如果该 **row** 变量是非类型的变量，则您必须在 UPDATE ***之前***使用 SELECT，以便 GBase 8s ESQL/C 可确定这些字段的数据类型。**row** 变量中字段的 UPDATE 不可包括 WHERE 子句。

row 变量可存储该行的字段值，但它没有与数据库列内在的连接。一旦 row 变量包含正确的字段值，您必须以下列 SQL 语句之一将该变量保存到 ROW 列内：

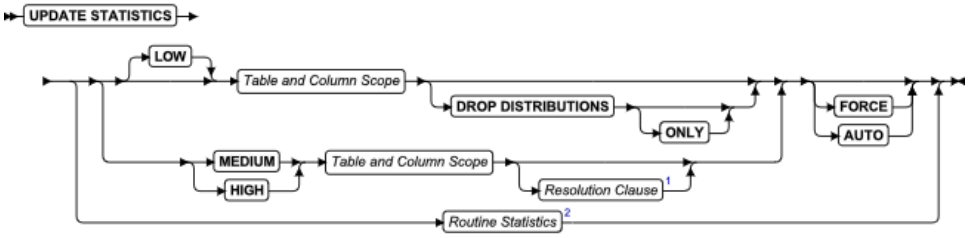
- 要以 **row** 变量的内容更新表中的 ROW 列，请在表或视图名称上使用 UPDATE 语句并在 SET 子句中指定该 **row** 变量。（要获取更多信息，请参阅 更新 ROW 类型列。）
- 要将 **row** 插入到列内，请在表或视图名称上使用 INSERT 语句并在 VALUES 子句中指定 **row** 变量。（要获取更多信息，请参阅 将值插入到 ROW 类型列内。）

要获取 SPL ROW 变量的示例，请参阅 *GBase 8s SQL 教程指南*。要获取关于使用 GBase 8s ESQL/C **row** 变量的更多信息，请参阅 *GBase 8s ESQL/C 程序员手册* 中对复杂数据类型的讨论。

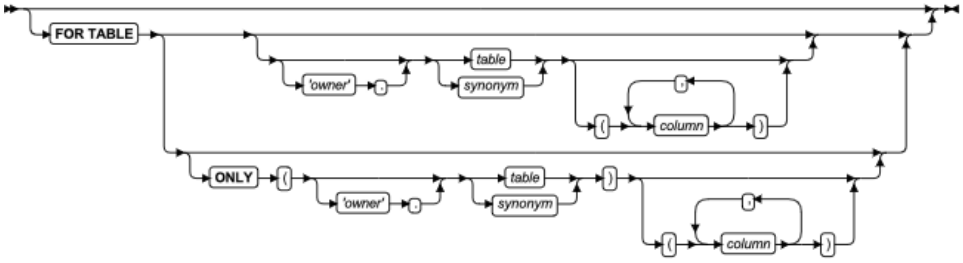
## 2.155 UPDATE STATISTICS 语句

使用 UPDATE STATISTICS 语句来更新系统目录信息，查询优化器用于对本地数据库中对象上的操作。UPDATE STATISTICS 语句是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**




表和列作用域




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 表或同义词中的列 | 必须存在。带有 MEDIUM 或 HIGH 关键字，该列不可为 BYTE、LVARCHAR 或 TEXT 数据类型。 | 标识符 |
| *owner* | 表或同义词的所有者 | 必须是表或同义词的所有者 | 所有者名称 |
| *synonym* | 要更新其统计信息的表的同义词 | 在当前的数据库中，该同义词以及它指向的表必须存在 | 标识符 |
| *table* | 要为其更新统计信息的表 | 必须在当前的数据库中存在，或为在当前会话中创建的临时表 | 标识符 |

**用法**

使用 UPDATE STATISTICS 语句来执行任何下列任务：

- 为表和表分片计算列值的分发。
- 更新数据库服务器用来优化查询的系统目录表。
- 强制重新优化 SPL 例程。
- 当您升级数据库服务器时，转换现有的索引。

请在不包含任何其他语句的事务中运行 UPDATE STATISTICS 语句。

如果您未指定表、例程以及 Resolution 子句，则 UPDATE STATISTICS 语句的缺省作用域是当前数据库中所有永久表。（另请参阅主题 UPDATE STATISTICS 的作用域。）

在高可用性集群中的辅助服务器上，不支持 UPDATE STATISTICS 语句。

**限制：** 在除了当前数据库之外的任何数据库中，您都不可更新表的或 UDR 的查询计划的统计信息。也就是说，当执行 UPDATE STATISTICS 语句时，数据库服务器忽略数据库对象。

## UPDATE STATISTICS 的作用域

跟在 FOR TABLE 关键字或 FOR PROCEDURE 关键字之后的任何表、列或 SPL 例程都限制 UPDATE STATISTICS 的作用域。

- 如果 UPDATE STATISTICS 语句
    - 未包括 Table 和 Column Scope 子句，且
    - 无 Resolution 子句，且
    - 无 FOR FUNCTION 规范，且
    - 无 FOR PROCEDURE 规范，且
    - 无 FOR ROUTINE 规范，且
    - 无 FOR SPECIFIC 规范，

  则在缺省情况下，对于当前数据库中的每个永久表，在 LOW 模式下更新列分发统计信息，包括系统目录表。
- 如果您使用 FOR TABLE 关键字，但未指定表的名称或同义词，则数据库服务器重新计算当前数据中所有表上的分发，以及您的会话中所有临时表上的分发。（然而，UPDATE STATISTICS 对通过 CREATE EXTERNAL TABLE 语句定义的对象不起作用。）
- 如果您在 FOR TABLE 关键字之后指定表，但没有指定列的列表，则数据库服务器重新计算指定的表的所有列上的统计分发。
- 如果您包括 FOR PROCEDURE 关键字，但未指定任何 SPL 例程的名称，则数据库服务器重新优化当前数据库中所有 SPL 例程的查询执行计划。

## 更新表的统计信息

虽然对数据库的更改可能使得 **systables**、**syscolumns**、**sysindices**、**sysfragments**、**sysdistrib** 和 **sysfragdist** 系统目录表中的信息过时，但在大多数 SQL 语句之后，数据库服务器不自动地更新那些表。

*在下列情况下，发出恰当的 UPDATE STATISTICS 语句来确保系统目录表中的列分发信息反映数据库的当前状态：*

- 您对表执行扩展的修改。
- 应用更改列值的分发。

  *UPDATE STATISTICS 语句刷新数据库服务器用于优化修改了的对象上的查询的数据分发统计信息。*

- 您以较新的数据库服务器为用户升级数据库。

  *UPDATE STATISTICS 语句将旧的索引转换为符合较新的数据库服务器索引格式，并隐式地删除旧的索引。*

  *您可逐个表地转换索引，也可一次转换整个数据库的索引。请遵循 GBase 8s 迁移指南 中的转换指南。*

如果您的引用导致特定的表中数据的许多修改，则以 UPDATE STATISTICS 常规地为那个表更新系统目录来提升查询效率。术语***许多修改***是相对于分发的分解。如果数据修改对列值的分发几乎没有影响，则您不需要执行 UPDATE STATISTICS。

**NLSCASE INSENSITIVE 数据库中的分发统计信息**

在以 NLSCASE INSENSITIVE 属性创建的数据库中，在列和 NCHAR 或 NVARCHAR 数据类型的表达式上的数据库服务器操作，在大写字母与小写字母之间没有区别。在包括相同的字母序列但大小写不同的字符串的数据集中，与包含相同的记录的区分大小写的数据库相比，生成 NCHAR 和 NVARCHAR 列的数据分发需要较少的 bin。数据库服务器仅将所有大小写不同的值标识为单个的不同值，并当它生成该列、索引或分片级统计信息时使用此结果。

要获取更多关于 NLSCASE INSENSITIVE 数据库的信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行、指定 NLSCASE 区分大小写 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## 自动化表统计信息维护

要简化 DBA 在维护当前的表统计信息中复杂的和重复的任务（查询优化器可从其设计高效的查询计划） GBase 8s 提供表统计信息维护系统，称为“自动更新统计信息”（AUS）。这可自动化对其统计信息陈旧的表的标识，并可自动化相应的 UPDATE STATISTICS 语句的构造和执行，以重新计算它们的列分发。当应该更新表统计信息时，以内建的准则提供 AUS 系统，但 DBA 可修改这些准则来反映当前的需要和工作负荷。

要获取更多信息，请参阅 *GBase 8s 性能指南* 中对“自动更新统计信息”维护系统的描述。另请参阅 *GBase 8s 管理员指南* 中对 Scheduler 的描述，DBA 可用于指定 AUS 系统以其重新计算表统计信息的政策、资源和频率。

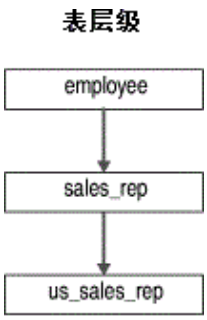
在 GBase OpenAdmin Tool (OAT) for GBase 8s 中也可都得到对表统计信息的 AUS 维护系统。请参考 OAT 在线帮助，以获取关于如何配置 AUS 维护系统来自动地提供当前表的统计信息的详细信息。

## 使用 FOR TABLE ONLY 关键字

使用 FOR TABLE ONLY 关键字来为 typed 表的层级之内单个表收集数据。当指定的 ***table*** 有子表时，如果您没有紧跟在 FOR TABLE 之后包括 ONLY 关键字，则 GBase 8s 为那个表以及为层级中它之下的每个子表创建分发。

例如，假设您的数据库有出现在 图 1 中的 typed 表层级， 展示名为 employee 的超级表有名为 sales_rep 的子表。反过来，sales_rep 表有名为 us_sales_rep 的子表。

*图: typed 表层级的示例*




当执行下列语句时，数据库服务器生成关于 sales_rep 和 us_sales_rep 表的统计信息：

```
UPDATE STATISTICS FOR TABLE sales_rep;
```

相反，下列示例为 sales_rep 中的每一列生成统计数据，但不处理表 employee 或 us_sales_rep：

```
UPDATE STATISTICS FOR TABLE ONLY (sales_rep);
```

如果你像在此示例中那样指定 FOR TABLE ONLY，则必须将表的标识符（或 ***owner.table***）括在圆括号之间。

由于前面的示例都未指定更新统计数据的级别，因此在缺省情况下，数据库服务器使用 LOW 模式。

## 更新列的统计信息

Table 和 Column Scope 规范还可包括一个或多个您想要为其计算统计分发的列的名称。

在下列示例中，此语句计算 orders 表的三列的分发：

```
UPDATE STATISTICS FOR TABLE orders (order_num, customer_num, ship_date);
```

如果您在 FOR TABLE 子句中未包括 ***column*** 名称，则使用 LOW、MEDIUM 或 HIGH 模式为指定的 ***table*** 的所有列计算分发，通过您请求的指定的或缺省的 Resolution 子句 ***percentage***（对于 MEDIUM 或 HIGH 模式）表明 bin 的数目。

要了解对存储 UDT 的列上的 UPDATE STATISTICS 限制，另请参阅 更新用户定义的类型的列的统计信息。

## 测试索引页

在 GBase 8s 中，当您在任何模式下执行 UPDATE STATISTICS 语句时，数据库服务器通读索引页来：

- 为查询优化器计算统计信息
- 定位有标记为 1 的删除标记的页

如果找到带有标记为 1 的删除标记的页，则从 B 树清除程序列表移除相应的键。

如果系统失败导致 B 树清除程序列表（存在于共享内存中）丢失，则此操作特别有用。要移除那些已标记为删除了但还未从 B 树移除的 B 树项，请运行 UPDATE STATISTICS 语句。要获取关于 B 树清除程序列表的信息，请参阅您的 *GBase 8s 管理员指南*。

## 更新用户定义的类型的列的统计信息

要计算更新用户定义的数据类型（UDT）的列的统计信息，您必须使用 UPDATE STATISTICS MEDIUM FOR TABLE 语句或 UPDATE STATISTICS HIGH FOR TABLE 语句。

不带有 MEDIUM 或 HIGH 关键字，UPDATE STATISTICS FOR TABLE 语句不理会 UDT 列。

**对 UDT 统计信息的限制**

对于持有用户定义的数据类型的列，UPDATE STATISTICS 语句不收集 **syscolumns** 系统目录表的 **colmin** 和 **colmax** 列的值。

要删除存储用户定义的数据类型的值的列的分发统计信息，您必须在 LOW 模式下执行 UPDATE STATISTICS，并包括 DROP DISTRIBUTIONS 关键字。

当您运行 UPDATE STATISTICS LOW FOR TABLE DROP DISTRIBUTIONS 语句时，数据库服务器删除对应于该列的 **tableid** 和 **colno** 值的 **sysdistrib** 系统目录表中的行。此外，数据库服务器移除可能为了指定的 opaque 列已创建了来存储分发统计信息的任何大对象。

# 对 Opaque 列上统计信息的要求

UPDATE STATISTICS 可控制用户定义的 opaque 数据类型的列的统计信息，仅当支持 **statcollect( )**、**statprint( )** 的例程以及为该 UDT 定义的选择性函数。你必须还持有对这些例程的 Usage 权限。

在有些情况下，如 SYSSBSPACENAME 配置参数指定的那样，UPDATE STATISTICS 还需要 sbspace。要获取关于如何为其数据类型为 UDT 的列提供统计数据的信息，请参考 *GBase 8s DataBlade API 程序员指南*。要获取关于 SYSSBSPACENAME 的信息，请参考您的 *GBase 8s 管理员参考手册*。

## 使用 FORCE 和 AUTO 关键字

您可可选地使用 FORCE 关键字或 AUTO 关键字来控制 UPDATE STATISTICS 语句的模式，当它更新系统目录中表和列的当前分发统计信息时。这些关键字仅影响表和分片统计信息，在对例程统计信息的操作中是无效的。

如果您既省略 FORCE 关键字又省略 AUTO 关键字，则由 AUTO_STAT_MODE 配置参数的显式设置或缺省设置确定 UPDATE STATISTICS 对表和分片分发统计信息的影响，除非设置 AUTO_STAT_MODE 会话环境变量来覆盖当前会话的那个配置参数。

指定这些关键字之一仅影响当前的 UPDATE STATISTICS 操作。如果您尝试在同一 UPDATE STATISTICS 语句中同时包括 FORCE 和 AUTO 关键字，则数据库服务器发出异常。

**重要：** 数据库服务器收集的统计信息可能需要用于存储的 sbspace。通过运行 gspaces -c -S 命令创建 sbspace，并将配置参数 SYSSBSPACENAME 设置为该 sbspace 名称。如果未设置 SYSSBSPACENAME 配置参数，则数据库服务器可能不能够存储指定的统计信息，因而 UPDATE STATISTICS 语句失败并报错 -9814, "Invalid default sbspace name"。

**FORCE 关键字**

FORCE 关键字刷新指定范围内所有表和列的统计信息。如果启用 UPDATE STATISTICS 语句的自动模式，则 FORCE 关键字覆盖自动模式，从而忽略 FOR TABLE 规范的范围内的表和分片的 STATCHANGE 属性的值，就好像当前 UPDATE STATISTICS FORCE 操作的 AUTO_STAT_MODE 设置是 OFF 一样。

下列语句指定 FORCE 关键字：

```
UPDATE STATISTICS FORCE;
```

此语句指导数据库服务器采取下列活动：

- 重新计算数据库中每个表的分发统计信息
- 重新优化每个用户定义的例程
- 将结果存储在系统目录中

包括 FORCE 关键字模仿 GBase 8s 数据库服务器 11.70 之前版本的先前的 UPDATE STATISTICS 行为。

**AUTO 关键字**

AUTO 关键字导致数据库服务器在自动模式下运行 UPDATE STATISTICS 语句，但仅对于其统计信息丢失或陈旧的表和分片。不为其 STATCHANGE 值低于指定阈值的表或分片刷新分发统计信息。

下列语句指定 AUTO 关键字：

```
UPDATE STATISTICS AUTO;
```

此语句指导数据库服务器采取下列行动：

- 重新为数据库中的每个表重新计算丢失的或陈旧的数据分发统计信息
- 重新优化每个用户定义的例程
- 将结果存储在系统目录中

对于某些表和表分片，当查询优化器已得到足够精确的现有的统计信息时，此选项避免不必要的重新计算。在那种情况下，与对应的 UPDATE STATISTICS FORCE 操作相比，UPDATE STATISTICS AUTO 操作需要更少的时间，又不损害查询性能。

## 使用 LOW 模式选项

使用 UPDATE STATISTICS 语句的 LOW 选项来生成和更新关于 **systables** 系统目录表中表、视图和页数统计信息的某些相关统计数据。如果您未指定任何模式，则 LOW 模式是缺省的。

在 GBase 8s 中，对于 **syscolumns** 和 **sysindexes** 系统目录表中指定的列，LOW 模式还生成和更新某些索引和列统计信息。

LOW 模式生成关于该列的最少量的信息。如果您想要 UPDATE STATISTICS 语句做极少的工作，则指定不是索引的一部分的列。不更新 **syscolumns** 中的 **colmax** 和 **colmin** 值，除非在该列上有索引。

下列示例更新 customer 表的 customer_num 列上的统计信息：

```
UPDATE STATISTICS LOW FOR TABLE customer (customer_num);
```

由于 LOW 模式选项不更新 **sysdistrib** 系统目录表中的数据，因此所有与 **customer** 表相关联的分发保持不动，即使那些在 **customer_num** 列上已存在的。

在 LOW 模式下采集 UPDATE STATISTICS 操作的索引统计信息期间，您可设置 USTLOW_SAMPLE 配置参数或 SET ENVIRONMENT 语句的 USTLOW_SAMPLE 选项来启用采样。

## 使用 DROP DISTRIBUTIONS 选项

使用 DROP DISTRIBUTIONS 选项来强制从 **sysdistrib** 系统目录表移除分发信息。

当您指定 DROP DISTRIBUTIONS 选项时，数据库服务器从您指定的一列或多列移除现有的分发数据。如果您未指定任何列，则数据库服务器为那个表移除所有分发数据。

您必须有 DBA 权限或是要使用此选项的表的所有者。

下列示例展示如何移除 customer 表中 customer_num 列的分发：

```
UPDATE STATISTICS LOW
        FOR TABLE customer (customer_num) DROP DISTRIBUTIONS;
```

如该示例所示，您在更新 LOW 模式选项生成的统计信息的同时，删除该分发数据。

## 使用 DROP DISTRIBUTIONS ONLY 选项

使用 DROP DISTRIBUTIONS ONLY 选项来从 **sysdistrib** 表移除分发信息，并为其分发已被删除的那些表更新该系统目录中的 **systables.version** 列，而不收集任何 LOW 模式表和索引统计信息。

如果您同时指定 DROP DISTRIBUTIONS ONLY 选项和 FOR TABLE 子句，则 GBase 8s 为 FOR TABLE 子句指定的 ***table*** 的列的集合（或为所有列，如果您未提供 ***column*** 规范的话）移除现有的分发数据，但不收集任何 LOW 模式表和索引统计信息。

您必须有 DBA 权限或是要使用此选项的表的所有者。

下列示例移除 customer 表中 customer_num 列的分发：

```
UPDATE STATISTICS LOW
    FOR TABLE customer (customer_num) DROP DISTRIBUTIONS ONLY;
```

当 ONLY 关键字不跟在 DROP DISTRIBUTIONS 关键字之后时，这会删除 **customer.customer_num** 分发信息，而不更新 LOW 模式选项生成的统计信息。此示例从系统目录删除任何描述 **sysdistrib** 表的 **customer.customer_num** 的行，并更新 **systables** 表中 **customer** 的 **version** 数目。不在 **systables** 上执行任何其他 LOW 模式更新，因此通过此示例不更改 **nrow** 和

**npused** 列值，且不更新系统目录的 **syscolumns**、**sysfragments** 和 **sysindexes** 表。在此示例中，LOW 关键字不起作用，但在 MEDIUM 或 HIGH 模式中 DROP DISTRIBUTIONS ONLY 选项不可用。

由于它未指定 FOR TABLE 子句，因此下一示例从 sysdistrib 表删除所有行，并为数据库中所有表更新系统目录中的 systables.version 列。

```
UPDATE STATISTICS DROP DISTRIBUTIONS ONLY;
```

## 使用 MEDIUM 模式选项

使用 MEDIUM 模式选项来更新与您可以 LOW 模式执行的相同的统计信息，并还生成关于每一指定的列的数据值的分发的统计信息。

UPDATE STATISTICS MEDIUM 在表上已运行之后，查询优化器通常选择更为高效的执行计划，这是与当对于该表仅获得 LOW 模式列分发统计信息时的同一 SELECT 语句相比。

数据库服务器将分发信息放置在 **sysdistrib** 系统目录表中，对于使用分布式存储的分片的表，还放置在其他系统目录表中。

如果您使用 MEDIUM 模式选项，则数据库服务器至少扫描表一次，并在给定的表上花费比 LOW 模式选项更长的执行时间。

当您使用 MEDIUM 模式选项时，通过采样数据行的百分率来获取数据分发，使用您指定的统计信任级别，或缺省的信任级别 95%。您还可在 Resolution 子句中指定一个显式的最小采样大小。由于 MEDIUM 采样大小通常远远小于实际的行数，因此此模式比 HIGH 模式执行得更快。

在通过采样获取的分发中，结果可多种多样，因为行的不同样例可有不同的采样错误。如果结果差异很大，则您可使用 Resolution 子句来增加采样大小，或降低 ***percent***，或增加 ***confidence*** 级别来获取更多一致的结果。

如果 Resolution 子句未指定每 bin 的采样的行的 ***percent***，则在每一 bin 中样例的缺省的平均百分率是 2.5，其将范围分为大约 40 个间隔。如果您未指定 ***confidence*** 级别的值，则缺省的级别是 0.95。此值可粗略地解释为 100 次中抽取 95 次，在 MEDIUM 估计的值与来自 HIGH 分发的精确值之间在统计上没有显著的差异。

然而，不为 LVARCHAR、BYTE 或 TEXT 列计算分发。

您必须有 DBA 权限或是该表的所有者来创建 MEDIUM 模式分发。要获取更多关于 MEDIUM 和 HIGH 模式选项的信息，请参阅 Resolution 子句。

## 使用 HIGH 模式选项

使用 HIGH 模式选项来更新您可以 MEDIUM 模式选项计算的相同的统计信息。UPDATE STATISTICS HIGH 与 UPDATE STATISTICS MEDIUM 之间的差异是采样行的数目。

在 UPDATE STATISTICS MEDIUM 仅采样行的子集时，UPDATE STATISTICS HIGH 基于通过 UPDATE STATISTICS 语句使用的信任和分辨率来扫描整个表。

对于已经获得每个列的 MEDIUM 模式分发统计信息的索引了的表，您在其为索引键的一部分的每个列上运行 UPDATE STATISTICS HIGH 之后，查询优化器通常选择更为高效的执行计划。

数据库服务器将分发信息放置在 **sysdistrib** 系统目录表中，对于使用分布式存储的分片的表，还放置在其他系统目录表中。

如果您未指定 Resolution 子句，则分发给每个 bin 的数据的缺省百分率为 0.5，这个值将每一列的值的范围分成大约 200 个间隔。

构造的分发是准确的。因为收集更多的信息，此模式比 LOW 或 MEDIUM 模式执行得更慢。如果您使用 UPDATE STATISTICS 的 HIGH 模式选项，则数据库服务器可花费相当多的时间来收集跨数据库的信息，特别是带有大型表的数据库。HIGH 模式可能扫描每一表的每一列多次。要最小化处理时间，请指定表名称和那个表内的列名称，而不是接受所有表的缺省的范围。

然而，对于 LVARCHAR、BYTE 或 TEXT 列，不计算分发。

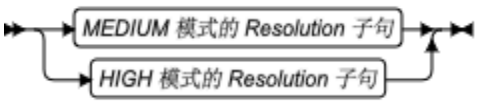
你必须有 DBA 权限或是该表的所有者来创建 HIGH 模式分发。要获取更多关于 MEDIUM 和 HIGH 模式选项的信息，请参阅主题 Resolution 子句。

## Resolution 子句

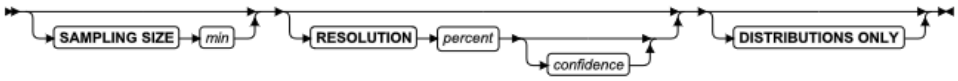
在 MEDIUM 或 HIGH 模式下，使用 Resolution 子句来调整分发 bin 的大小，并避免在索引上计算数据。

仅在 MEDIUM 模式下，您还可使用 Resolution 子句来指定对采样大小的较低限定，并调整信任级别。

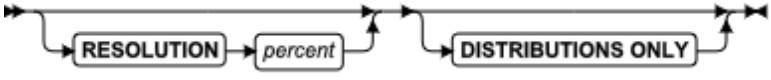
Resolution 子句




MEDIUM 模式的 Resolution 子句




HIGH 模式的 Resolution 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *confidence* | 在 MEDIUM 模式下采样的估计的可能性产生的结果正好等同于 HIGH 模式。缺省的级别为 0.95。 | 必须在从 0.80（最小）至 0.99（最大）的范围内 | 精确数值 |

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *percent* | 在每一分发 bin 中采样的平均百分率。MEDIUM 的缺省值为 2.5，HIGH 的缺省值为 0.5。 | 对于表中的行数 *nrows*，最小值为 1/*nrows* | 精确数值 |
| *min* | 在其上生成数据分发的随机选择的行的最小整数数目 | 必须大于 0，但不可超过 *nrows* | 精确数值 |

***分发***是列中的数据到列值的集合内的映射，按照量或对照排序。将这些样例值分成分离的间隔，称为 ***bin***，每一个都包含列值的样例的大约相等的部分。例如，如果一个 bin 持有数据的 2%，则大约 50 个这样的间隔持有整个样例。

有些统计文本将这些 bin 称为 ***equivalence categories***。每一个都包含从该列采样的数据值的范围的分离的子集。

如果您包括 RESOLUTION 关键字，则必须有一个文字数跟在它之后，指定每一 bin 中值的 *percent*。在 MEDIUM 模式下，一个或两个文字数可跟在它之后，以可选的第二个数目指定 *confidence* 级别，如此例中所示：

```
UPDATE STATISTICS MEDIUM FOR TABLE orders
        RESOLUTION 4 0.90 DISTRIBUTIONS ONLY;
```

这指定每 bin 数据的 4%，暗指大约 25 个 bin，以及信任级别 90%，且不检测索引数据。如果省略了 0.90 值，则缺省的信任级别会已生效。如果省略了 RESOLUTION 关键字和两个数值，则会使用缺省的 *percent* 值（2.5%）和 *confidence* 值（0.95）。

对于包括在 WHERE 子句中的每一列，查询优化器通过测试包含在该列中的数据值的发生比例，来估计 WHERE 子句的选择性。

您不可为 BYTE 或 TEXT 列创建分发。如果您在指定 MEDIUM 或 HIGH 分发的 UPDATE STATISTICS 语句中包括 BYTE 或 TEXT 列，则不为那些列创建分发。然而，为列表中的其他列创建分发，且该语句不返回错误。

VARCHAR 数据类型的列不使用溢出 bin，即使正在为重复的值使用多个 bin 时。

当 UPDATE STATISTICS 语句构建列分布时，您可使用 **DBUPSPACE** 环境变量的前两个参数来限制 UPDATE STATISTICS 语句可用于数据排序的磁盘空间和内存资源。这些设置影响性能，因为它们决定数据库服务器要扫描指定的表多少次来构建每一分发。（当计算列分布时，第三个 **DBUPSPACE** 参数可控制 UPDATE STATISTICS 是否以索引排序，以及 explain 输出文件是否通过计算的列分发来存储计划。）

## 指定 SAMPLING SIZE

对于计算列分发统计信息，在 MEDIUM 模式下，您可可选地使用 SAMPLING SIZE 关键字来指定采样的行的最小数。如果 Resolution 子句省略 RESOLUTION 关键字且未指定 confidence 级别，也未指定 percent 值，则 GBase 8s 采样的行数将会是下列两个值中的较大者：

- 您紧跟在 SAMPLING SIZE 关键字之后指定的 ***min*** 值
- 在每一 bin 中行的缺省 ***percent***（2.5%）以及最小的 ***confidence*** 级别（0.80）所需要的采样大小。

如果在 Resolution 子句中指定采样大小，包括为每 bin 的采样的行的平均 *percent* 以及为 *confidence* 级别的显式的值，则采样的行的数目将会是这两个值中的较大者：

- 您紧跟在 SAMPLING SIZE 之后指定的 ***min*** 值
- 指定的行的 ***percent*** 和指定的 ***confidence*** 级别所需要的采样大小。

如果在 Resolution 子句中指定采样大小，包括平均的 ***percentage*** 值，但未设置 ***confidence*** 级别，则使用最小的 ***confidence*** 值 0.80 来计算 GBase 8s 要使用的实际采样大小，如果指定的 ***size*** 较小的话。

例如，下列语句计算 customer 表的两列的统计信息，而不更新索引信息。将至少采样 200 行，但样例的实际大小可能大于 200， 如果对于使用大约 50 equivalence categories 的样例分发，以每一 bin 中采样值的平均百分率 2%，需要更多的行来提供缺省的 0.80 信任级别的话。

```
UPDATE STATISTICS MEDIUM FOR TABLE customer (city, state)
        SAMPLING SIZE 200 RESOLUTION 2 DISTRIBUTIONS ONLY;
```

不管您是否在 Resolution 子句中包括显式的 SAMPLING SIZE 规范，在 MEDIUM 模式 UPDATE STATISTICS 创建的时刻， GBase 8s 都在系统目录中记录实际的采样大小（作为该表中行的总数的百分率）。

## 使用 DISTRIBUTIONS ONLY 选项来阻止索引信息

在 GBase 8s 中，当您指定 DISTRIBUTIONS ONLY 选项时，您不更新索引信息。此选项不影响现有的索引信息。

使用此选项来避免可消耗大量处理时间的索引信息的测试。

此选项不影响表上信息的重新计算，诸如使用的页数、行数以及分片信息。UPDATE STATISTICS 需要此信息来指导精确的列分发，并需要很少的时间和系统资源来收集它。

请不要将此 DISTRIBUTIONS ONLY 选项与 LOW 模式的 DROP DISTRIBUTIONS ONLY 选项混淆，在 MEDIUM 或 HIGH 模式下，不支持其语法和语义。要获取如何阻止列分发的收集的信息，请参阅 使用 DROP DISTRIBUTIONS ONLY 选项。

## 使用 DBUPSPACE 设置来阻止索引信息

通过将 DBUPSPACE 环境变量的第三个参数设置为值 1，您还可防止由 UPDATE STATISTICS 操作在对行排序中使用索引。要获取关于 DBSPACETEMP 和 DBUPSPACE 环境变量的信息，请参考 《GBase 8s SQL 指南：参考》 的第 3 章，其可限制 UPDATE STATISTICS 操作可用的系统资源。（仅当您使用 UPDATE STATISTICS 的 HIGH 选项时，数据库服务器才使用 DBSPACETEMP 指定的存储位置。）

# 来自 SET EXPLAIN 语句的 UPDATE STATISTICS 的输出

SET EXPLAIN 语句可显示 UPDATE STATISTICS 用来生成列分发的计划。下列输出是基于排序内存的缺省的 DBUPSPACE 值 15 MB，在此示例中其需要两道检验来对 21.9 MB 数据进行排序：

```
UPDATE STATISTICS:
==================

Table: zelaine.t1
Mode: HIGH
Number of Bins: 267 Bin size 2505
Sort data 21.9 MB Sort memory granted 15.0 MB
Estimated number of table scans 2
PASS #1 b
PASS #2 a
Scan 9 Sort 1 Build 2 Insert 0 Close 0 Total 12
Completed pass 1 in 0 minutes 12 seconds
Scan 5 Sort 2 Build 1 Insert 0 Close 0 Total 8
Completed pass 2 in 0 minutes 8 seconds
```

## 例程统计信息

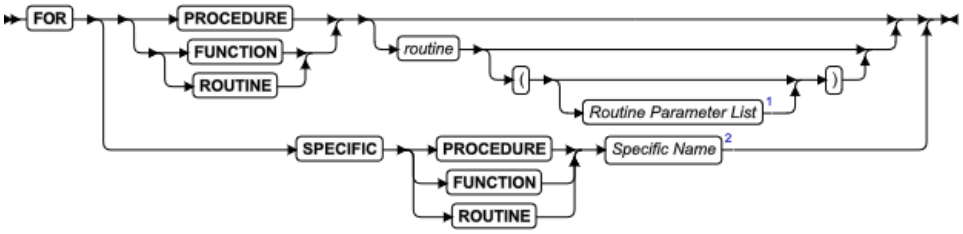
在首次执行新的 SPL 例程之前，数据库服务器优化 SPL 例程中的 DML 语句。然而，在您调用那个 SPL 例程之前，如果您使用 UPDATE STATISTICS 的 Routine Statistics 语法来更新它的查询执行计划，则可降低出错的风险，其中一些可能引用已通过并发会话的 DDL 操作修改了其模式的表。

优化使得该代码依赖于该例程引用的表的结构。在优化该例程之后，但在执行它之前，如果 DDL 操作修改引用的表的模式，则例程失败并报错。

然而，通常不会发生此失败，如果为引用表的例程启用自动的重新编译，而 ALTER TABLE、CREATE INDEX 或 DROP INDEX 操作已修改了这些表的话。这是 GBase 8s 的缺省行为。要获取更多关于在更改表的模式之后启用或禁用自动的重新优化的信息，请参阅对 SET ENVIRONMENT 语句的 IFX_AUTO_REPREPARE 选项的描述。

然而，当将 AUTO_REPREPARE 配置参数和 IFX_AUTO_REPREPARE 会话环境变量设置为禁用 SPL 例程的重新编译，这些例程引用已修改了其模式的表时，直接地向 SPL 例程引用的表添加或删除索引可导致该例程返回错误 -710。要避免在 DDL 操作之后发生此错误，或要在已通过 DML 操作修改了表分发之后重新优化 SPL 例程，请使用 UPDATE STATISTICS 的 Routine Statistics 段来更新引用该表的任何 SPL 例程的执行计划。

Routine Statistics




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *routine* | CREATE FUNCTION 或 CREATE PROCEDURE 语句为 SPL 例程声明了的名称 | 必须在数据库中存在。在符合 ANSI 的数据库中，如果您不是 *owner*，则以 *owner* 限定 *routine*。 | 标识符 |

下表说明 Routine Statistics 段的关键字。

**关键字**

**优化哪一执行计划**

FUNCTION

带有指定名称（以及带有与 ***routine 参数列表*** 相匹配的参数类型，如果提供的话）的 SPL 函数的计划。 如果您指定 FUNCTION 关键字，则 UPDATE STATISTICS 语句失败并报错，除非指定的例程返回一个值或多个值，带有或不带有 WITH RESUME 选项。

PROCEDURE

带有指定的名称（以及与 ***routine 参数列表*** 相匹配的参数类型，如果提供的话）的 SPL 过程的计划

ROUTINE

带有指定的名称（以及与***例程参数列表***相匹配的参数类型，如果提供的话）的 SPL 函数和过程的计划

SPECIFIC

名为 ***specific name*** 的 SPL 例程的计划。如果您包括 SPECIFIC 关键字，则紧跟在关键字之后的必须是 FUNCTION、PROCEDURE 或 ROUTINE。

如果您省略 SPECIFIC 关键字且未包括参数列表，则圆括号符号是可选的。

如果您未紧跟在 FOR FUNCTION、FOR PROCEDURE 或 FOR ROUTINE 关键字之后指定 ***routine*** 名称，则为当前数据库中所有 SPL 例程优化执行计划。

数据库服务器保持一个该 SPL 例程显式地引用的表的列表。无论何时修改显式地引用了的表，数据库服务器都在下次执行该过程时重新优化过程。

sysprocplan 系统目录表存储 SPL 例程的执行计划。两个活动可更新 sysprocplan 系统目录表：

执行使用修改了的表的 SPL 例程

UPDATE STATISTICS FOR ROUTINE、FUNCTION 或 PROCEDURE 语句。

如果您更改 SPL 例程引用的表，则可运行 UPDATE STATISTICS 来重新优化引用该表的过程，而不是一直等到下次执行使用该表的 SPL 例程为止。然而，如果 SPL 例程引用的表被删除，则运行 UPDATE STATISTICS 不可防止该 SPL 例程失败并报错。

**更新特定的例程的统计信息的示例**

下列 UPDATE STATISTICS FOR SPECIFIC 语句指导数据库服务器更新现有的返回一个或多个值的名为 Perform_work 的函数的统计信息：

```
UPDATE STATISTICS FOR SPECIFIC FUNCTION Perform_work;
```

对于同一 Perform_work 函数，下列示例的作用与前一示例相同：

```
UPDATE STATISTICS FOR SPECIFIC ROUTINE Perform_work;
```

类似地，使用关键字 SPECIFIC PROCEDURE 或 SPECIFIC ROUTINE 来更新不返回值的 SPECIFIC 过程的统计信息。

请不要在 SPECIFIC 例程的名称之后包括圆括号或参数列表。由于跟在名为 Perform_work 的函数之后的圆括号，下列语句失败并报错：

```
UPDATE STATISTICS FOR SPECIFIC ROUTINE Perform_work();
```

如果以圆括号括起 SPECIFIC 例程、函数或过程的参数，则数据库服务器也发出错误。

## 在 SPL 例程中间接地引用的更改了的表

然而，如果 SPL 例程依赖于仅被间接地引用的表，则在修改那个表之后，数据库服务器不可检测到重新优化该过程的需要。例如，如果 SPL 调用触发器，则可间接地引用表。如果更改由触发器引用的表的模式（但不是由 SPL 例程直接地引用），则在运行它之前，数据库服务器不知道它应重新优化该 SPL 例程。当在已更改了该表之后运行该过程时，可发生错误 -710。

在首次运行每一 SPL 例程时（不是当创建它时），优化它。此行为意味着 SPL 例程可以首次运行成功，但在虚拟相同的环境之下，后来会失败，如果已更改了间接引用了的表的模式的话。SPL 例程的失败还可是间歇的，因为一次执行期间的失败会强制内部的警告，来在下次执行之前重新优化该过程。

您可以两种方法之一来从此错误恢复：

- 发出 UPDATE STATISTICS 来强制执行该例程的重新优化。
- 重新运行该例程。

要防止此错误，您可强制重新优化该 SPL 例程。要强制重新优化，请执行下列语句：

```
UPDATE STATISTICS FOR PROCEDURE routine;
```

您可以下列方式之一将此语句添加到您的程序：

- 在更改对象的模式的每一语句之后，发出 UPDATE STATISTICS。
- 在每一 SPL 例程的调用之前，发出 UPDATE STATISTICS。

为了提高效率，您可将 UPDATE STATISTICS 语句与在程序中较少发生的行动（对象模式或过程的执行的更改）放在一起。在大多数情况下，在程序中较少发生的行为是对象模式的更改。

当您遵循此方法从此错误恢复时，必须为间接引用更改了的表的每一过程执行 UPDATE STATISTICS，除非该过程也显式地引用了这些表。

在只是简单地通过重新执行 SPL 例程来修改被间接地引用了的表之后，您也可从错误 -710 恢复。存储了的过程首次失败时，数据库服务器将该过程标记为需要重新优化。您下一次运行该过程时，数据库服务器在运行该过程之前优化它。然而，运行 SPL 例程两次可能既不实际也不安全。较为安全的选择是使用 UPDATE STATISTICS 语句来强制该过程的重新优化。

## 当您升级数据库服务器时更新统计信息

当您升级数据库来使用更新的数据库服务器时，您可使用 UPDATE STATISTICS 语句来将索引转换为更新的数据库服务器使用的形式。

您可选择一次转换一个表上的索引，或一次处理整个数据库。请遵循 *GBase 8s 迁移指南* 中概述的转换指南。

UPDATE STATISTICS 语句在其中导致隐式地删除和重新创建表索引，是迁移到更新的数据库服务器的仅有的上下文。

## UPDATE STATISTICS 语句的性能因素

您制作的 UPDATE STATISTICS 语句检测的对象的列表越明确，它完成执行就越快。限定列分发的数目，会提高更新速度。类似地，精度影响更新的速度。如果所有其他关键字一样，则 LOW 运行最快，但 HIGH 检测的数据最多。

对于 GBase 8s 数据库服务器， AUTO_STAT_MODE 设置可提高刷新数据分发统计信息的 UPDATE STATISTICS 操作的效率。 这使得数据库服务器能够有选择地仅重新计算表或分片分发，自从上一次计算统计信息以来，作为 DML 操作的结果，这些分发已变得陈旧了。 如同由一个显式的或缺省的 STATCHANGE 表属性定义的阈值更改所决定的那样。 要获取关于如何设置 STATCHANGE 以及如何为仅刷新陈旧的分发统计信息而启用 UPDATE STATISTICS 的自动模式的信息，请参阅 这些主题：

- 使用 FORCE 和 AUTO 关键字
- AUTO_STAT_MODE 环境选项
- STATCHANGE 环境选项
- CREATE TABLE 语句的 Statistics 选项
- ALTER TABLE 语句的 Statistics 选项

在 LOW 模式下，在收集 UPDATE STATISTICS 操作的索引统计信息期间，USTLOW_SAMPLE 环境选项 启用采样。对于多于 100 K 叶页的索引，使用采样的统计信息的收集可提高 UPDATE STATISTICS 操作的速度。

**UPDATE STATISTICS 语句的示例**

```
UPDATE STATISTICS MEDIUM;
```

```
UPDATE STATISTICS MEDIUM RESOLUTION 10;
UPDATE STATISTICS MEDIUM RESOLUTION 10 .95;
        { RESOLUTION 10, CONFIDENCE .95}
UPDATE STATISTICS MEDIUM RESOLUTION 10 DISTRIBUTIONS ONLY;
UPDATE STATISTICS MEDIUM RESOLUTION 10 .95 DISTRIBUTIONS ONLY;

UPDATE STATISTICS HIGH;
UPDATE STATISTICS HIGH RESOLUTION 10;
UPDATE STATISTICS HIGH RESOLUTION 10 DISTRIBUTIONS ONLY;
```

解析度必须大于 0.005 并小于或等于 10.0。信任水平必须在 [0.80, 0.99]（包括 0.80 和 0.99）范围中。

下列示例是基于 company_proc 过程和 square_w_default 函数，定义如下：

```
CREATE PROCEDURE company_proc ( no_of_items INT,
            itm_quantity SMALLINT, sale_amount MONEY,
            customer VARCHAR(50), sales_person VARCHAR(30) )
            SPECIFIC spec_cmpy

            DEFINE salesperson_proc VARCHAR(60);

            -- Update the company table
            INSERT INTO company_tbl VALUES (no_of_items, itm_quantity,
            sale_amount, customer, sales_person);

            -- Generate the procedure name for the variable salesperson_proc
            LET salesperson_proc = sales_person || "." || "tbl" ||
            month(current) || "_" || year(current) || "_proc" ;

            -- Execute the SPL procedure that the salesperson_proc
            -- variable specifies
            EXECUTE PROCEDURE salesperson_proc (no_of_items,
            itm_quantity, sale_amount, customer);
            END PROCEDURE;


            CREATE FUNCTION square_w_default
            (i INT DEFAULT 0) {Specifies default value of i}
            RETURNING INT {Specifies return of INT value}
            SPECIFIC spec_square
            DEFINE j INT; {Defines routine variable j}
            LET j = i * i; {Finds square of i and assigns it to j}
            RETURN j; {Returns value of j to calling module}
            END FUNCTION;
```

下列 UPDATE STATISTICS 示例引用 company_proc 过程和 square_w_default 函数：

```
UPDATE STATISTICS    FOR PROCEDURE;
UPDATE STATISTICS    FOR PROCEDURE company_proc1;
UPDATE STATISTICS    FOR PROCEDURE
            company_proc1(INT,SMALLINT,MONEY,VARCHAR(50), VARCHAR(30));
UPDATE STATISTICS FOR SPECIFIC PROCEDURE spec_cmpy;

UPDATE STATISTICS FOR FUNCTION;
UPDATE STATISTICS FOR FUNCTION square_w_default;
UPDATE STATISTICS FOR FUNCTION square_w_default(INT);
UPDATE STATISTICS FOR SPECIFIC FUNCTION spec_square;
```

要获取 UPDATE STATISTICS 的性能含义的讨论，请参阅您的 *GBase 8s 性能指南* 。

要获取如何使用 **dbschema** 实用程序来查看以 UPDATE STATISTICS 创建的分发的讨论，请参阅 *GBase 8s 迁移指南* 。

## 2.156 WITH AS 语句

使用 WITH AS 语句可定义子查询别名和列别名，并可在其他子查询和 SELECT 查询块引用该别名。定义的别名作用域仅限于紧跟在 WITH AS 之后的 SELECT 查询块的单次查询，查询后失效。在 SELECT 查询块未定义关联条件下，查询语句的结果集为各子查询结果的笛卡尔积。

WITH AS 的语法如下：

```
WITH <子查询别名  [(<列名>{,<列名>})] AS(子查询)> [,<子查询别名  [(<列名>{,<列名>})] AS(子查询)>]
SELECT 查询块
```

用法如下：子查询别名支持的字符与现有系统表名保持一致；支持 WITH 子句的子查询数量为一个或多个；支持 WITH 子句的子查询可不罗列列别名，不罗列列别名时，列名与子查询投影列列名保持一致；支持 WITH 子句的子查询名称可与当前数据库下基础表或基础视图重名，重名时，均被当做子查询别名处理。

WITH 子句定义的名称，其生命周期为紧邻 WITH AS 语句之后的 SELECT 语句查询块查询后结束，且查询一次后就失效，WITH 子句中无法访问到其他数据库的基础表或基础视图；支持 WITH 子句的子查询为多个使用集合运算符联结（UNION ALL、UNION、EXCEPT 或 INTERSECT）的 SELECT 查询语句，且集合运算符的查询语句不能使用 WITH 定义的子查询别名。子查询别名没有指定列，子查询包含 UNION ALL 等词语时，结果集的列名字与现有 UNION ALL 等语句保持一致；WITH 子句定义的子查询别名可以在查询块未使用；支持 WITH 子句的多个子查询可引用已定义的子查询别名，且只能先定义后引用。

WITH AS 可以支持 SELECT 查询块嵌套，支持 WITH 子句在批处理的语句中使用，支持在 DELETE、INSERT…SELECT、UPDATE 子句中使用。使用 WITH AS 定义子查询别名时，支持使用 FUNCTION 定义函数。支持 WITH 语句中有多条 FUNCTION 语句和多条 AS 语句，FUNCTION 语句必须放在 AS 语句之前，AS 语句可以使用 FUNCTION 定义的函数，两类语句不允许穿插。AS 定义子查询别名支持与 FUNCTION 语句重名。

WITH AS 语句中使用 ***引用字符串*** ，当前版本不支持字符串内包含 ？。引用字符串如包含 ？，返回语法错误。

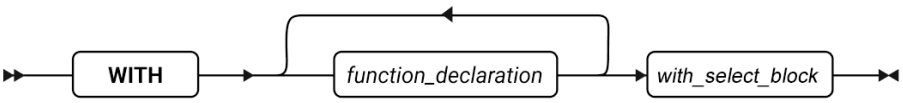
例如，可以通过如下代码在 with as 定义别名后进行查询：

```
WITH myEmployees AS
(
   SELECT * FROM employees WHERE department_id = '90'
)
SELECT * FROM myEmployees;
```

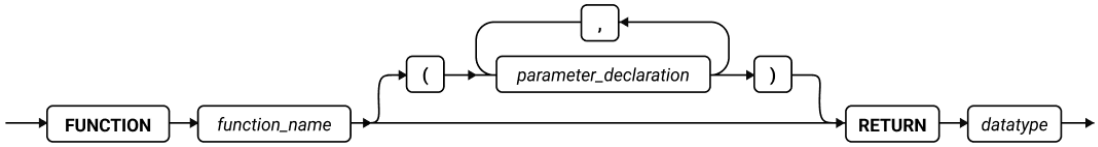
## 2.157 WITH FUNCTION 语句

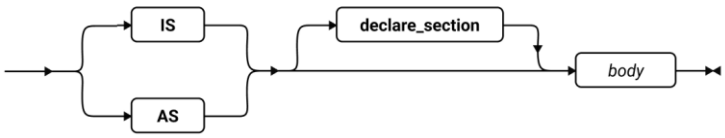
该功能仅在 GBase 8s 的 ORACLE 模式下支持。WITH FUNCTION 语句用于临时声明并定义存储函数。WITH FUNCTION 定义的函数对象不会存储到系统表中，且只在当前 SQL 语句内有效。。

语法：




function_declaration::=




说明及限制：

- <WITH FUNCTION> 语句定义函数 <function_declaration> 作用域为 <with_select_block > 所在的 SELECT 查询块内；
- < with_select_block > SELECT 查询块与 8s 现有语法规则保持一致。
- 在 SELECT 查询语句的 SELECT 关键字前允许使用<WITH FUNCTION>定义一个或多个自定义函数，并在后续 SELECT 语句中可以使用这些自定义函数，语法、行为与 8s 保持一致
- 数据库中存在同名自定义函数，优先使用通过 WITH FUNCTION 定义的存储函数。

例如，声明两个临时存储函数 add_string、doesnt_it 嵌套使用：

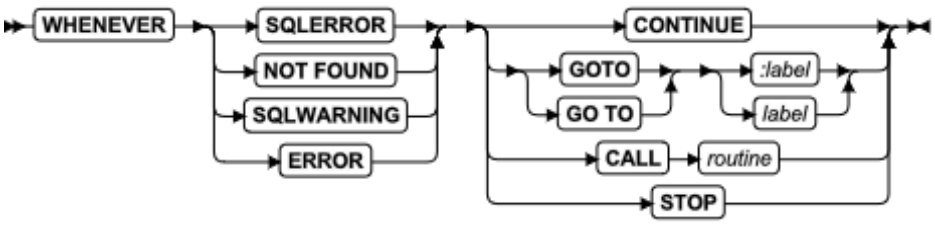
```
WITH
  FUNCTION add_string(p_string IN VARCHAR2) RETURN VARCHAR2
  IS
```

```
    l_buffer VARCHAR2(100);
  BEGIN
    l_buffer := p_string || ' works!';
    RETURN l_buffer;
  END;
  FUNCTION doesnt_it(p_string IN VARCHAR2) RETURN VARCHAR2
  IS
    l_buffer VARCHAR2(100);
  BEGIN
    l_buffer := p_string || ' Doesnt it?';
    RETURN l_buffer;
  END;
SELECT doesnt_it(add_string('Yes, it')) as outVal FROM DUAL;
/
RESULT:
OUTVAL
--------
Yes, it works! Doesnt it?
```

## 2.158 WHENEVER 语句

使用 WHENEVER 语句来捕获在执行 SQL 语句期间发生的异常。WHENEVER 语句等同于在每个 SQL 语句之后放置一异常检查例程。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 当发生异常时程序控制传送到的语句标签 | 必须在同一源代码模块中存在。 | 特定于语言 |
| *routine* | 当发生异常时，要调用的用户 | 无参数；在编译时刻 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 定义的例程（UDR）的名称 | UDR 必须存在。 | |

**用法**

**重要：** 仅随同 GBase 8s ESQL/C 使用此语句。

下表总结您可以 WHENEVER 语句检查的异常的类型。

| 异常的类型 | WHENEVER 关键字 | 要获取更多信息 |
| --- | --- | --- |
| 错误 | SQLERROR 或 ERROR | SQLERROR 关键字 |
| 警告 | SQLWARNING 关键字 | |
| Not Found 或 End of Data | NOT FOUND 关键字 | |

当发生异常时，不使用 WHENEVER 语句的程序不会自动地异常终止。这样的程序必须显式地检查异常并采取它们的逻辑指定的任何更正活动。如果您不检查异常，则程序只是继续运行。然而，如果发生错误，则程序可能不会达到它想要达到的目的。

跟在 WHENEVER 之后的第一个关键字指定一些异常的条件的类型；该语句的最后一部分指定当遇到异常时采取的行动（或者不采取行动，如果指定 CONTINUE 的话）。下表总结 WHENEVER 可指定的可能的行动。

| 行动的类型 | WHENEVER 关键字 | 要获取更多信息 |
| --- | --- | --- |
| 继续程序执行 | CONTINUE 关键字 | |
| 停止程序执行 | STOP 关键字 | |
| 将控制传送到指定的标签 | GOTO GO TO | GOTO 关键字 |
| 将控制传送到 UDR | CALL 子句 | |

## WHENEVER 的作用域

WHENEVER 是预处理器伪指令，而不是可执行的语句。GBase 8s ESQL/C 预处理器，不是数据库服务器，负责解释 WHENEVER 语句。当预处理器在 GBase 8s ESQL/C 源文件中遇到 WHENEVER 语句时，基于异常和 WHENEVER 指定的行动，它将适当的代码插入到每一 SQL 语句之后的预处理代码内。WHENEVER 语句的作用域起始于该语句在源模块中出现的地方，并保持有效，直到在继续处理该源模块时预处理器遇到下列情况中的一种时为止：

- 在同一源模块中带有相同的条件（SQLERROR、SQLWARNING 或 NOT FOUND）的下一 WHENEVER 语句
- 源模块的结尾

下列 GBase 8s ESQL/C 示例程序有三个 WHENEVER 语句，其中两个是 WHENEVER SQLERROR 语句。第 4 行使用带有 SQLERROR 的 STOP 来覆盖错误的缺省的 CONTINUE 活动。

第 8 行指定 CONTINUE 关键字来将错误的处理返回到缺省的行为。对于第 4 行与第 8 行之间的所有 SQL 语句，预处理器插入检查错误的代码，如果发生错误，则终止程序执行。因此，第 6 行 INSERT 语句产生的任何错误都会导致程序停止。

在第 8 行之后，预处理器不在 SQL 语句之后插入检查错误的代码。因此，忽略 INSERT 语句（第 10 行）、SELECT 语句（第 11 行）和 DISCONNECT 语句（第 12 行）产生的任何错误。然而，如果 SELECT 语句未定位到任何行，则它不停止程序执行；如果发生这样的异常，则第 7 行上的 WHENEVER 语句告诉程序继续：

```
1    main()
2    {
3    EXEC SQL connect to 'test';
4    EXEC SQL WHENEVER SQLERROR STOP;
5    printf("\n\nGoing to try first insert\n\n");
6    EXEC SQL insert into test_color values ('green');
7    EXEC SQL WHENEVER NOT FOUND CONTINUE;
8    EXEC SQL WHENEVER SQLERROR CONTINUE;
9    printf("\n\nGoing to try second insert\n\n");
10     EXEC SQL insert into test_color values ('blue');
11     EXEC SQL select paint_type from paint where color='red';
12     EXEC SQL disconnect all;
13     printf("\n\nProgram over\n\n");
14     }
```

## SQLERROR 关键字

如果您使用 SQLERROR 关键字，则遇到错误的任何 SQL 语句都按照 WHENEVER SQLERROR 语句的指示处理。如果发生错误，则将 **sqlcode** 变量（**sqlca.sqlcode**、**SQLCODE**）设置为小于零（0）的值，并将 SQLSTATE 变量设置为值大于 02 的类代码。

如果检测到 SQL 错误，则下一示例终止程序执行：

```
WHENEVER SQLERROR STOP
```

如果您在程序中未包括任何 WHENEVER SQLERROR 语句，则 WHENEVER SQLERROR 的缺省的活动是 CONTINUE。

## ERROR 关键字

在 WHENEVER 语句内（且仅在此上下文中），关键字 ERROR 是 SQLERROR 关键字的同义词。

## SQLWARNING 关键字

如果您使用 SQLWARNING 关键字，则生成警告的任何 SQL 语句都按 WHENEVER SQLWARNING 语句的指示处理。如果发生警告，则将 SQLCA 中警告结构的第一个字段（**sqlca.sqlwarn.sqlwarn0**）设置为 W，并将 SQLSTATE 变量设置为类代码 01。

除了警告结构的第一个字段之外，警号还将附加的字段设置为 W。设置的该字段指示发生的警告的类型。

如果存在警告的条件，则下一语句导致执行停止：

```
WHENEVER SQLWARNING STOP
```

如果您在程序中不使用任何 WHENEVER SQLWARNING 语句，则 WHENEVER SQLWARNING 的缺省的活动是 CONTINUE。

## NOT FOUND 关键字

如果使用 NOT FOUND 关键字，则 SELECT 和 FETCH 语句（包括 FOREACH 和 UNLOAD 语句中的隐式的 SELECT 和 FETCH 语句）的异常处理将以不同于其他 SQL 语句的方式进行。NOT FOUND 关键字检查下列情况：

- **End of Data** 条件：尝试获取活动集中第一行或最后一行之外的行的 FETCH 语句
- **Not Found** 条件：不返回行的 SELECT 语句

在每一情况下，**sqlcode** 变量设置为 100，且 SQLSTATE 变量有类代码 02。要了解在每一 GBase 8s 产品中 **sqlcode** 变量的名称，请参阅 SQLERROR 关键字 中的表。

在每一次存在 NOT FOUND 条件时，下列语句调用 no_rows( ) 函数：

```
WHENEVER NOT FOUND CALL no_rows
```

如果您在程序中不使用任何 WHENEVER NOT FOUND 语句，则 WHENEVER NOT FOUND 的缺省的活动是 CONTINUE。

## CONTINUE 关键字

使用 CONTINUE 关键字来指导程序忽略异常并在 SQL 语句之后继续在下一语句执行。所有异常的缺省的活动是 CONTINUE。您可使用此关键字来关闭先前为异常条件指定的活动。

## STOP 关键字

当发生指定的异常时，使用 STOP 关键字来指导程序停止执行。在每一次 SQL 语句生成警告时，下列语句终止 GBase 8s ESQL/C 程序的执行：

```
EXEC SQL WHENEVER SQLWARNING STOP;
```

## GOTO 关键字

当发生指定的异常时，使用 GOTO 子句来将控制传送到该标签标识的语句。 对于像 ESQL/C 这样的嵌入的 SQL 语言的此特性，GOTO 和 GO TO 关键字是符合 ANSI 的语法。 下列 GBase 8s

ESQL/C 代码片断展示在每一次发生 NOT FOUND 条件时，WHENEVER 语句将控制传送到标签 missing：

```
query_data()
        ...
    EXEC SQL WHENEVER NOT FOUND GO TO missing;
        ...
    EXEC SQL fetch lname into :lname;
        ...
        missing:
        printf("No Customers Found\n");
```

在 WHENEVER GOTO 语句的作用域内，您必须在包含 SQL 语句的每一例程中定义打了标签的语句。如果您的程序包含多个用户定义的函数，则您可能需要包括打了标签的语句及其在每一函数中的代码。

如果预处理器在 WHENEVER ... GOTO 语句的作用域内遇到 SQL 语句，但在没有指定的标签的例程内，则预处理器试图插入与该打了标签的语句相关联的代码，但当它不可找到该标签时，会生成错误。

要纠正此错误，或在每一 UDR 中放置一个带有相同的标签名称的打了标签的语句，或发出另一 WHENEVER 语句来重置该错误条件，或使用 CALL 子句来调用单独的函数。

## CALL 子句

当发生指定类型的异常时，使用 CALL 子句来将程序控制传送到指定的 UDR。请不要在 UDR 名称之后包括圆括号。如果程序检测到错误，则下列 WHENEVER 语句导致程序调用 error_recovery() 函数：

```
EXEC SQL WHENEVER SQLERROR CALL error_recovery;
```

当 UDR 返回时，在正导致该错误的行之后的下一语句处恢复执行。当发生错误时，如果您想要终止执行，包括终止该程序作为指定的 UDR 的一部分的语句。

请观察指定的例程上的下列限制：

- UDR 不可接受参数，也不可返回值。如果它需要外部的信息，请使用全局变量或 WHENEVER ... GOTO 选项来将程序控制传送到调用该 UDR 的标签。
- 您不可在 CALL 子句中指定 SPL 例程的名称。要调用 SPL 例程，请使用 CALL 子句来调用包含 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句的 UDR。
- 请确保在 WHENEVER ... CALL 语句的作用域内的所有函数都可发现指定的函数的声明。

**相关的语句**

相关的语句：EXECUTE FUNCTION 语句、EXECUTE PROCEDURE 语句 和 FETCH 语句

要了解关于异常处理和错误检查的讨论，请参阅 *GBase 8s ESQL/C 程序员手册*。

# 3. SPL 语句

该主题描述存储过程语言（SPL）语句，用于编写 SPL 例程。您可以将这些例程作为用户定义的例程（UDR）存储在数据库中。

SPL 例程（之前称为*存储过程*）是用于控制 SQL 活动的有效工具。本章包含 SPL 语句的描述。每个语句的描述包含以下信息： 信息

- 说明语句的作用的简单介绍
- 显示如何正确输入语句的语法图表
- 说明语法图表中每个输入参数的语法表
- 使用规则，包括带有说明这些规则的示例

如果语句由多个子句组成，则该语句描述为每个子句提供相同的信息集合。

有关创建和使用 SPL 例程的 SPL 语言和面向任务的信息的概述，请参阅 *GBase 8s SQL 教程指南*。

有关如果创建和使用 SPL 例程中预备对象和动态 SQL 的详细示例的概述，请参阅 GBase developerWorks 中的 GBase 8s 数据库服务器存储过程语言的动态 SQL 支持。

GBase 8s 可以使用 CREATE PROCEDURE 或 CREATE PROCEDURE FROM 语句创建 SPL 函数，但是它对与外部函数则需要使用 CREATE FUNCTION 或 CREATE FUNCTION FROM 语句。但是，建议您使用 CREATE FUNCTION 或 CREATE FUNCTION FROM 语句创建新的用户定义的函数。

## 3.1 调试 SPL 例程

可以使用 Routine Debugger 客户端应用程序标识并分析 SPL 中的逻辑错误。

您可以在 SPL 例程中包含 TRACE 语句来生成跟踪输出。有关如何生成及检查 TRACE 语句的输出，请参阅 TRACE。

**调试 SPL 例程中的当前限制**

以下软件产品可支持当前用于调试 GBase 8s SPL 例程的客户端环境：

- Optim™ Development Studio (ODS)
- GBase Database Add-Ins for Visual Studio (IDAIVS)

有关哪些 GBase 8s 数据类型是只读的以及哪些是可更新的信息，请参阅数据类型支持文档。

以下限制适用于 Optim Development Studio (ODS) 和 GBase Database Add-Ins for Visual Studio (IDAIVS) 调试环境：

**未日志记录的数据库**

GBase 8s 非事务型数据库不支持 SPL 调试。无法对 ODS 或 IDAIVS 调试环境使用省略 WITH LOG 关键字的 CREATE DATABASE 语句的 GBase 8s 数据库。

**辅助服务器实例**

集群环境中的辅助服务器不支持 Insert 、Delete 或 Update 触发器的 'Step into' 触发过程操作。

**用引号（"）分隔的数据字符串**

ODS 或 IDAIVS 调试客户端当前对引号标记分隔符的解释可能与某些 SPL 例程的预期行为冲突。

- 目前，在 ODS 调试会话中，缺省情况下，GBase Data Server 驱动程序 JDBC 和 SQLJ 连接字符串中的 **DELIMIDENT** 环境变量设置为'y'。JDBC 连接的 **DELIMIDENT** 缺省值为'n'。
- 在当前 IDAIVS 调试会话中，缺省情况下， GBase 8s .NET provider 连接字符串中的 **DELIMIDENT** 环境变量设置为 'y' 。

**DELIMIDENT** 环境变量的设置会影响数据库服务器如何解释带引号的字符串：

- 'y' 指定由双引号（"）标记包括的字符串是分隔的 SQL 标识符。客户端应用程序必须使用单引号（'）标记分隔字符串，并且只能在分隔的 SQL 标识符周围使用双引号（"），可以支持比在未限定标识符中有效的更大的字符集。在支持区分大小写的语言环境中，分隔字符串中或分隔标识符内的字母都区分大小写。这是 .NET 的缺省值。
- 'n' 指定客户端应用程序可以使用双引号（"）或单引号（'）标记分隔字符串，但是不能分隔 SQL 标识符。如果数据库服务器遇到在需要 SQL 标识符的上下文中由双引号或单引号分隔的字符串，则它会发出错误。限定 SQL 标识符的所有者名称可以用单引号（'）标记分隔。您必须使用一对相同的引号符号来分隔字符串。
- 如果客户端系统上没有指定 **DELIMIDENT** 值，则使用缺省设置。如上所述，对于 OCD 该缺省值为 'n'（来自 GBase Data Server 驱动程序 JDBC 和 SQLJ 连接字符串）。对于 IDAIVS ，该缺省值为 'y'（来自 GBase 8s .NET provider 连接字符串）。

**只影响 ODS 调试会话的限制**

缺省情况下，JDBC 和 SQLJ 的 GBase Data Server 驱动程序的 'AUTOCOMMIT' 事务方式设置为 'TRUE' 。它会在所有的日志记录的数据库的 ODS 调试会话中启用，包括创建的 WITH LOG MODE ANSI 的数据库，以及不兼容 ANSI 的数据库。

目前，GBase 8s 服务器在 Optim Development Studio 调试会话中完成每个 SQL 语句后执行隐式提交操作。如果在 GBase 8s SPL 例程中启动了显式事务，则数据库服务器将忽略 SQL ERROR -535 并继续调试会话。

**只影响 IDAIVS 调试会话的限制**

目前，GBase Database Add-Ins for Visual Studio 不支持 SPL 函数调试。您只能将 IDAIVS 调试环境用于不向调用上下文返回任何值的 GBase 8s SPL 过程。

## 使用 Optim Development Studio 开始 SPL 调试会话

要使用 Optim™ Development Studio (ODS) 调试 SPL 例程，您必须在 GBase 8s 数据库服务器和 ODS 客户端之间建立连接。

您可以在客户端调试环境中检查 SPL 例程的运行时的行为，并用 Optim Development Studio（ODS）的标准调试器接口控制调试会话的流程。以下步骤也适用于 Optim Data Studio。

**注：** GBase 8s 数据库服务器的 SPL Routine Debugger 支持在 ODS 2.2.1.0 及以后的版本中有效。早期的 ODS 版本（如 2.2.0）支持 GBase 8s 数据库服务器，但是不提供 GBase 8s SPL 支持和 SPL Routine Debugger 支持。

要在 ODS 中启用 SPL Routine 调试会话，请遵循以下步骤：

**GBase 8s 服务器实例：启动和配置**

要配置 GBase 8s 数据库服务器实例以致于启用 SPL 例程调试，请按照下列步骤操作：

1. 安装 GBase 8s 数据库服务器产品。
2. 配置 **onconfig** 和 **sqlhosts** 文件中的条目并启动 GBase 8s 服务器，以支持 DRDA 通信协议：

   - onconfig:

     DBSERVERNAME ids_spldb
   - sqlhosts:

     ids_spldb drsoctcp ***ids_server_machine_name*** ***port_number***

   **注：** GBase 8s 数据库服务器的 SPL Routine Debugger 支持通过 JDBC 和 SQL 的 GBase Data Server 驱动连接到客户端，并且需要 GBase 8s DRDA 协议连接。
3. 您必须提供一个 sbspace ，以便数据库服务器存储 ODS 发送给服务器的 XML 消息。SBSPACENAME 配置参数的设置指定了系统缺省 sbspace 的名称。可以通过带 `-Df "LOGGING=ON"` 选项的 gspaces 实用程序在数据库上创建缺省 sbspace。此 sbspace 在 ODS 上调试 SPL 的必要条件。
4. 如果这是新的服务器实例，使用 CREATE DATABASE 语句创建新的数据库。稍后将使用此数据库在 ODS 中创建数据库连接，以部署和调试 SPL 例程。

**启动例程调试器会话管理器（可选的）**

在大多数情况下，您可以使用 ODS 提供的内置的会话管理其调试 GBase 8s SPL 例程。但是在某些情况中（例如，如果您的服务器机器在防火墙之后），您可能需要在 TCP/IP 端口上启动会话管理器，该端口在服务器计算机或其它计算机上具有出站访问权限。

要手动启动会话管理器，请按照下列步骤操作：

1. 要手动启动会话管理器，使用以下的命令输出 **CLASSPATH** 环境变量设置：

   ```
   export CLASSPATH=${GBASEDBTDIR}/bin/db2dbgm.jar:$CLASSPATH
   ```
2. 使用 Java™ 1.5.0 或更高版本并确保它在 PATH 环境变量中运行下列命令，指定会话管理器将会使用的 ***port number*** 以及会话管理器日志文件的 ***pathname*** ：

java com.gbase.db2.psmd.mgr.Daemon -port ***port_num*** -log ***sess_mgr_log_path***

**在 ODS 中部署和调试 SPL 例程的步骤**

要配置 ODS 以部署和调试 GBase 8s SPL 例程，请按照下列步骤操作：

1. 如果不使用内置会话管理器并启动单个的会话管理器，请使用菜单命令在 ODS 中配置该选项（**Window > Preferences > Run/Debug > Routine Debugger >** DB2 **screen**）方法是选择 "Use an already running Session manager" 单项按钮，并提供端口号和机器名称。
2. 要配置 ODS 以调试 GBase 8s SPL 例程，您需要在 ODS 的 **Data Source Explorer** 中创建一个 Database 连接。有关如何创建 Database 连接的详细信息请参阅 Optim Development Studio 文档。
3. 在 **Database Connection > New Connection** 对话框中右击。选择 GBase 8s ，并通过从驱动程序列表中选择对 GBase 8s 适当版本的 JDBC 和 SQLJ GBase Data Server 驱动程序来创建新的数据库连接。
4. 在 **Edit Driver Definitions** 对话框中，打开 **Jar List** 选项卡并验证它是否包含 **db2jcc4.jar**。如果没有，在 **Driver files:** 列表中使用 **db2jcc4.jar** 替换 **db2jcc.jar** 。
5. 有关如何 **Create** 、**Deploy** 和 **Debug** SPL 例程的详细信息请参阅 Optim Development Studio 文档。

现在您可以准备在 Optim Development Studio 中创建、部署并调试 GBase 8s SPL 例程。

## 使用 GBase Database Add-Ins for Visual Studio 调试 SPL 过程

可以使用 GBase Database Add-Ins for Visual Studio 调试 GBase 8s SPL 过程。

您可以调试运行在 GBase 8s 上的任何 SPL 过程。您可以遍历您的代码，设置行或变量断点，查看变量值，更改变量值，查看嵌套过程的调用堆栈信息，以及在调用堆栈的不同过程之间切换。通过在调试模式下运行过程并查看结果时单步执行代码，可以发现过程的问题并进行必要的更改。

要在 GBase Database Add-Ins for Visual Studio（IDAIVS）中启动 SPL 例程调试，请按照以下步骤操作：

**启动并配置 GBase 8s 数据库服务器实例**

要配置 GBase 8s 数据库服务器实例以致于启用 SPL 例程调试，请按照下列步骤操作：

1. 安装 GBase 8s 数据库服务器产品。
2. 配置 **onconfig** 和 **sqlhosts** 文件中的条目并启动 GBase 8s 服务器，以支持 DRDA 通信协议：

   - onconfig:

     DBSERVERNAME ids_spldb
   - sqlhosts:

ids_spldb drsoctcp ***ids_server_machine_name*** ***port_number***

**注：** GBase 8s 的 SPL Routine Debugger 支持通过 GBase 8s .Net provider 连接到客户端，并且需要 GBase 8s DRDA 协议连接。

3. 您必须提供一个 sbspace ，以便数据库服务器存储 GBase Database Add-Ins for Visual Studio 发送给服务器的 XML 消息。SBSPACENAME 配置参数的设置指定了系统缺省 sbspace 的名称。可以通过带 -Df "LOGGING=ON" 选项的 **gspaces** 实用程序在数据库上创建缺省 sbspace。此 sbspace 在 GBase Database Add-Ins for Visual Studio 上调试 SPL 的必要条件。
4. 如果这是新的服务器实例，使用 CREATE DATABASE 语句创建新的数据库。稍后将使用此数据库在 GBase Database Add-Ins for Visual Studio 中创建数据库连接，以部署和调试 SPL 例程。

**启动例程调试会话管理器**

要启动会话管理器，请按照下列步骤操作：

1. 要手动启动会话管理器，使用以下的命令输出 **CLASSPATH** 环境变量设置：

   export CLASSPATH=${GBASEDBTDIR}/bin/db2dbgm.jar:$CLASSPATH

2. 使用 Java™ 1.5.0 或更高版本并确保它在 PATH 环境变量中运行下列命令，指定会话管理器将会使用的 ***port number*** 以及会话管理器日志文件的 ***pathname*** ：

   java com.gbase.db2.psmd.mgr.Daemon -port ***port_num*** -log ***sess_mgr_log_path***

**注：** 该会话管理器必须在运行 GBase 8s 实例的机器上启动，这也是 GBase Database Add-Ins for Visual Studio 连接到会话管理器的必要条件。

**建立 GBase 8s 数据库服务器连接**

通过选择 "Add Connection…" 菜单项创建新连接，可以从服务器资源管理器中 "Data Connections" 节点上的操作（右键单击）弹出菜单访问 Add Connection 对话框。**Data source** 字段会显示 " GBase 8s and U2 Servers (GBase 8s and U2 Data Provider)。"

1. 在下面 "**1. Select or enter server name:**"字段中，输入 GBase 8s 服务器实例的名称。
2. 在 "**2. Enter information to log on to the server:**" 旁边的字段中输入您的用户 ID 作为 **User ID:** ，在 **Password:** 字段中输入服务器上 **gbasedbt** 账户的密码。
3. 在"**3. Select or enter a database name:**" 下面的字段中，输入存储要调试的 SPL 过程的数据库的名称。
4. 点击 **Test Connection** 按钮以测试 GBase 8s 服务器实例的连接。
5. 如果出现 **Test connection succeeded** 对话框，您可以点击 **OK** 按钮。可以忽略此 Add Connection 对话框，因为您已经在 GBase 8s 服务器和 Visual Studio 之间创建了连接。

**设置会话管理器**

在您开始调试会话之前，需要指定要调试的 SPL 例程，和数据库服务器连接以及会话管理器的端口号。

1. 从 Server Explorer 中的 "Data Connections" 节点中，单击所有连接的 GBase 8s 服务器。
2. 展开 Procedures 标题以显示数据库中所有存储过程的名称。
3. 单击您要调试的 SPL 例程的名称。
4. 从 Tools 选项卡，选择 Tools 列表底部附近的 "Options . . . " 菜单。
5. 从 Options 对话框中，展开 GBase 8s Database Tools 项，然后单击 **General** 项显示类别的列表。
6. 使用 **General** 选项列表右侧的滑块控件显示 **Session Manager Connection** 和 **Session Manager Port** 号。
7. 单击 **OK** 接受会话管理器配置的这些值。

有关如何使用 Visual Studio 的调试 SPL 例程的详细信息，请参阅 GBase Database Add-Ins for Visual Studio。

**调试 GBase 8s SPL 过程**

这是使用 GBase Database Add-Ins for Visual Studio 调试 SPL 过程的步骤：

1. 启用该过程的调试：
   a. 在数据连接的服务器资源管理器中，右键点击要调试的过程，然后单击快捷菜单上的 **Open Definition** 。
   b. 在 GBase Procedure Designer 的过程视图中，在 Debug mode 文件中选择 ALLOW。
   c. 保存此过程，但是保持 Designer 打开。
2. 在 GBase Procedure Designer 中设置行断点。
   a. 如果过程在 Designer 中未打开，请在服务器资源管理器中的数据连接下，右键单击该过程，然后单击快捷菜单上的 Open Definition 。
   b. 在 Designer 的“过程”视图的 SQL Body 部分中，设置行断点。
   c. 要设置断点的属性，请右键单击左边距中的断点，选择快捷菜单上的 Location 、Filter 或 When Hit ，并在打开的窗口中指定必要的信息。
3. 在调试模式中开始运行此过程。
   - 如果该过程在 GBase Procedure Designer 中打开，请单击 GBase Procedure Designer 选项卡上的 Step Into。
   - 如果未在 Designer 中打开此过程，在资源管理器中右击此过程，然后单击快捷菜单上的 Step Into。
4. 在调试模式下运行每个过程，并使用以下方法之一：

- 设置变量断点。在 SQL 主体中，右击变量名称，在快捷菜单上点击 Breakpoints ，并选择 Insert Variable Breakpoints。
- 修改断点的值。

5. 继续调试 GBase 8s SPL 过程直到该过程返回期望的结果。

**在 CLR 应用程序中调试 GBase 8s SPL 过程**

当您以 C# 和 Visual Basic 语言开发 Windows™ 或 ASP.NET 应用程序，您可以使用 GBase Database Add-Ins for Visual Studio 调试公共语言运行库（CLR）应用程序。 如果该应用程序访问 GBase 8s 数据库服务器实例，则您可以在调试此应用程序时调试从此应用程序调用的 SPL 过程。

GBase Unified Debugger 通过 GBase Procedure Designer 调试 SPL 过程。如果当您调试应用程序时一个过程定义在 Designer 中打开，随着调试器进入该过程，将激活该 Designer 的实例。如果未在 Designer 中打开该过程定义，则随着调试器进入该过程，此调试器会在 Designer 的新实例中打开过程定义。

**先决条件：** 要调试 CLR 应用程序的 SPL 过程，在包含此应用程序中的项目中启用 GBase 8s SQL debugging 。

要调试 CLR 应用程序的 SPL 过程，请按照以下步骤操作：

1. 对于应用程序中的每个 SPL 过程：
   a. 启用此过程的调试：
      a) 在数据连接的服务器资源管理器中，右键点击要调试的过程，然后单击快捷菜单上的 **Open Definition** 。
      b) 在 GBase Procedure Designer 的过程视图中，在 **Debug mode** 文件中选择 **ALLOW**。
      c) 保存此过程，并且可以选择打开 Designer 以设置行断点。
   b. 可选：在 GBase Procedure Designer 中设置行断点。
      1. 在 Designer 的“过程”视图的 **SQL Body** 部分中，设置行断点。
      2. 要设置断点的属性，请右键单击左边距中的断点，选择快捷菜单上的 **Location** 、**Filter** 或 **When Hit** ，并在打开的窗口中指定必要的信息。
      3. 保持 Designer 打开。
2. 开始调试应用程序。

   在解决方案资源管理器中，右击此应用程序，选择快捷菜单上的 **Debug** ，然后选择 **Start new instance**。

   调试开始而且 Debugger Task Status 窗口打开。

   通用调试的每个过程的调用堆栈与调用该过程的线程的调用堆栈合并。您可以从 C# 或 Visual Basic 代码中查看调用过程的位置。

3. 在调试模式运行每个调用的过程。

   - 如果在步骤 1a.iii 中关闭了 GBase Procedure Designer，在 Designer 的新实例中，在 SQL Body 部分为该过程设置行断点和断点属性。
   - 设置变量断点。在 SQL 主体中，右键单击变量名称，单击快捷菜单上的 Breakpoints 然后选择 Insert Variable Breakpoints。
   - 修改断点的值。

4. 要取消长时间运行任务，请在 Debugger Task Status 窗口中单击 Cancel 。

   如果在调试应用程序时关闭了窗口，请从 **Tools** 菜单，选择 Show GBase 8s Debugger Task Status 来重新打开此窗口。

5. 继续调试 GBase 8s SPL 过程直到该过程返回期望的结果。

在 CLR 应用程序中启用 SQL 调试

如果要在 C# 或 Visual Basic 公共语言运行时（CLR）应用程序中调试 GBase 8s SPL 过程，则必须首先为 GBase 数据库项目启用 GBase SQL 调试。您只需执行一次此过程，为项目启用 SQL 调试会在 Visual Studio 中设置一个会在会话之间持续存在的项目属性。

当您在应用程序中启用 SQL 调试，则 GBase Unified Debugger 必须关闭并重新打开包含此应用程序的 GBase 数据库项目。项目重新打开后，必须指定数据连接以及可选的运行调试会话管理器的端口。您还必须更改某些标准 Visual Studio 项目调试属性的设置。

加载项（ 32 位应用程序）不支持控制台应用程序的 64 位调试过程。在 64 位操作系统中，缺省的 debug 平台是 **Any CPU** ，这是一个 64 位调试过程。您必须在项目属性中指定 32 位调试平台。

**先决条件:**要在应用程序中启用 GBase SQL 调试，在包含此应用程序的 GBase 数据库项目必须在解决方资源案管理中打开。

要在应用程序中启用 GBase SQL 调试，请按照以下步骤操作：

1. 在解决方资源案管理中，右键单击 GBase 数据库项目节点，然后在快捷菜单中选择 Enable GBase 8s SQL Debugging。

   将显示一条消息，说明必须关闭并重新打开项目。

2. 在消息窗口中单击 Yes 以确认项目的关闭和重新打开。

   保存所有未保存的修改，关闭项目，然后在解决方案资源管理器中重新打开它。

3. 在解决方案资源管理器中，右键单击 GBase 数据库项目节点，然后在快捷菜单中选择 Properties 。

   将打开 Project Designer 。

4. 在 Designer 的左侧框，单击 GBase 8s Unified Debugger。

   将显示 GBase Unified Debugger 页面。

5. 执行以下一组操作：

- 如果您有以下连接之一用于调试会话管理器：
  - GBase 8s 12.10
  - Linux™ 、UNIX™ 和 Windows 的 DB2
  - DB2 z/OS® 9.1 或 10 版本
  - DB2 i V5R4 或 V6R1
    1. 选择 **Use an existing connection for session manager**。
    2. 在列表中，选择要在其上运行调试会话管理器的数据连接。
- 如果您正在手动运行此调试会话管理器：
    1. 选择 Use a new host name for session manager。
    2. 为此调试会话管理器指定主机名。

6. 如果您在 64 位的操作系统上调试控制台应用程序，请在 Platform 列表中选择 x86。
7. 可选的：在 Session manager port 字段中，键入要在其上运行调试会话管理器的端口。
8. 在项目设计器左侧框中，单击 Debug。

   将显示项目设计器的 Debug 页面。
9. 在 Debug 页面修改以下属性：
   - 在 Start Options，清除 Use remote machine 复选框。
   - 在 Enable Debuggers，清除以下三个复选框：
     - Enable unmanaged code debugging
     - Enable SQL Server debugging
     - Enable the Visual Studio hosting process
10. 关闭 Project Designer。

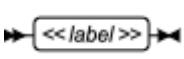
## 3.2 << Label >> 语句

使用 SPL 的 <<***label***>> 语句声明语句标签或循环标签。

- ***statement label*** 是 SQL 标识符，由双尖括号分隔，紧接语句块中语句之前，SPL 的 GO TO 语句可以传递程序执行的控制。
- ***loop label*** 是 SQL 标识符，由紧靠 SPL 循环语句之前的尖括号分隔。相同的标签，没有双尖括号分隔符，可以跟随 END LOOP 关键字或 END FOR 关键字或 END WHILE 关键字终止标签循环。EXIT ***label*** 语句可以将程序执行的控制传递到紧随未定界循环标签之后的任何语句。

请注意 *label* 并不是 <<*label*>> 语句的关键字，它是 <<*label*>> 语句声明的语句标签或循环标签某些特定用户定义的标识符的占位符。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *label* | 语句标签或 | 在 SPL 例程的语句标签和循环标签的中 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 循环标签的名称 | 必须是唯一的 | |

**用法**

您可以这两中方法使用 <<*Label*>> 语句：

- 在 SPL 的 GO TO 语句可以传递执行控制的可执行语句之前声明 ***statement label*** 。紧跟在语句标签声明之后的 SPL 语句称为 ***labeled statement***。
- 在 SPL 的 LOOP 、FOR 或 WHILE 语句之前声明 ***loop label*** 。紧跟在循环标签声明之后的 LOOP 、FOR 或 WHILE 语句称为 ***labeled loop***。

EXIT ***label*** 或 EXIT ***label*** WHEN （***条件***）语句可以从带标签的循环中跳出，将执行控制传递到 END LOOP ***label*** 标签语句之后的语句，在 EXIT 语句中指定的 ***label*** 可以匹配 EXIT 语句的标签循环的标签标识符，或者如果循环是嵌套的，此 ***label*** 必须符合输出的带标签的循环的标签。在任一情况下，EXIT ***label*** 语句将控制传递到跟随在指定相同循环标签的 END LOOP ***label*** 语句之后的语句。此 EXIT ***label*** 行为与 GOTO ***label*** 语句的行为不同，GOTO ***label*** 语句将控制传递到指定语句标签的声明之后的语句。

以下限制应用于 SPL 例程中的标签：

- 语句标签的名称必须在 GOTO 语句引用的作用域内。
- SQL 的 WHENEVER 语句的 GOTO 选项不能引用 SPL 语句标签，WHENEVER 语句只在 ESQL/C 应用程序中有效。
- SPL 的 GOTO 语句不能引用循环标签。
- GOTO 语句不能引用 ON EXCEPTION 语句块中的语句标签。
- 不能在 ON EXCEPTION 语句块中声明语句标签。
- 标签名称在 SPL 例程的语句标签和循环标签中必须是唯一的。

## 标签的示例

以下示例说明了 SPL 例程中名为 increment_x 的语句标签：

```
DEFINE x INT;
LET x = 0;
BEGIN
        <<increment_x>>
        BEGIN
                LET x = x + 1;
        END;
        IF x < 10 THEN
                GOTO increment_x;
        END IF;
END;
END   PROCEDURE;
```

以下程序片段显示了 FOR 循环标签的示例：

```
<<lb_for>>
FOR i IN 1..5
    i := i +1 ;
END FOR lb_for;
```

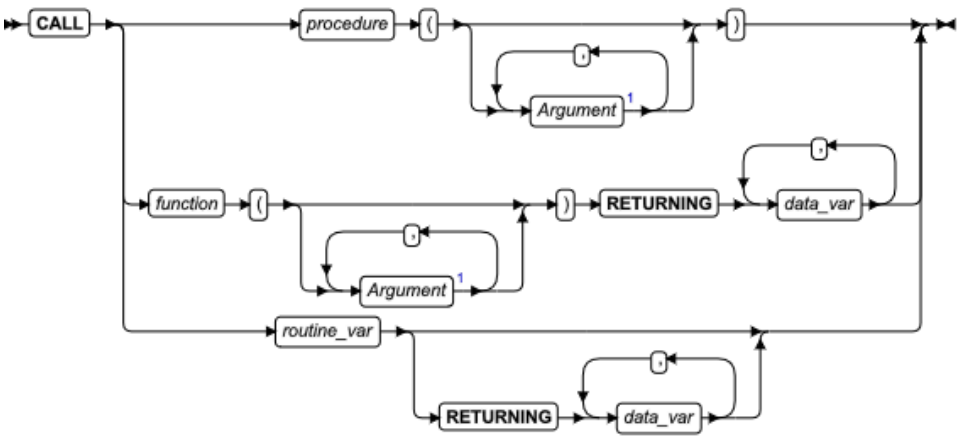
以下程序段说明了 EXIT *label* 语句可以退出的标签循环：

```
<<outer>>
LOOP
...
LOOP
...
EXIT outer WHEN ... -- exit from both loops
END LOOP;
...
END LOOP outer;
```

## 3.3 CALL

使用 CALL 在 SPL 例程中执行用户定义的例程（UDR）。

**语法**




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *data_var* | 用于接收 *function* 返回的值的变量 | *data_var* 的数据类型必须适合于已返回的值 | 标识符 |
| *function, procedure* | 用户定义的函数或过程 | 函数或过程必须存在 | 标识符 |
| *routine_var* | 包含 UDR 名称的变量 | 必须是包含现有 UDR 的非 NULL 名称的字符数据类型 | 标识符 |

**用法**

CALL 语句调用 UDR。CALL 语句对 EXECUTE PROCEDURE 和 EXECUTE FUNCTION 语句的行为是一致的，但仅可在 SPL 例程中使用 CALL 。

可以将 CALL 用在 GBase 8s ESQL/C 程序中或与 DB-Access 一起使用，但仅当该语句在执行了该程序或 DB-Access 的 SPL 例程中时。

当您使用 CALL 调用由其函数标识符指定的用户定义的函数或用于存储函数标识符的 ***routine_var*** 时，CALL 语句必须包含 RETURNING 子句。

CALL 语句不能从 SELECT 语句的 FROM 子句中的子查询调用迭代器 TABLE 函数。有关迭代器 TABLE 函数的语法，请参阅迭代器函数。

## 指定参数

CALL 语句的参数列表（由括号分隔）紧跟在 UDR 的名称后面。如果不包含执行参数，则空括号必须跟在 UDR 的名称后面。如果列表包含的参数超过 UDR 的参数，则会收到错误。

如果 CALL 指定的参数比 UDR 预期的少，则会指出缺少参数。数据库服务器将缺少的参数初始化为它们相应的缺省值。（请参阅 CREATE PROCEDURE 和 CREATE FUNCTION。）此初始化出现在 UDR 主体中的第一个可执行语句之前，如果缺少的参数没有缺省值，则它们被初始化为 UNDEFINED 的值。尝试使用 UNDEFINED 值的任何变量会导致错误。

在每个 UDR 调用中，您有指定传递给 UDR 的参数的参数名称的选择。下面每一个示例对于期望以该顺序命名为 t 、n 和 d 的字符参数是有效的：

```
CALL add_col (t='customer', n = 'newint', d ='integer');
CALL add_col('customer','newint','integer');
```

以上的 CALL 语句作用相同。

有关参数列表语法的详细信息，请参阅参数。

## 从被调用的 UDR 接收输入

RETURNING 子句指定变量，该变量接收已调用的函数返回的值。

以下示例显示两个 UDR 调用：

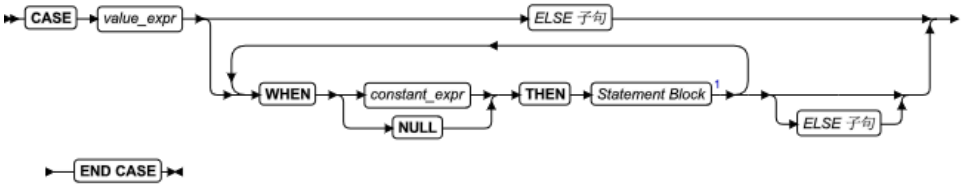
```
CREATE PROCEDURE not_much()
DEFINE i, j, k INT;
CALL no_args (10,20);
CALL yes_args (5) RETURNING i, j, k;
END PROCEDURE;
```

第一个例程调用（**no_args**）不期望有返回值。第二个例程调用是针对函数（**yes_args**）的，它期望三个返回值。**not_much()** 过程声明三个整数变量（**i** 、**j** 和 **k**）以接收来自 **yes_args** 的返回值。

# 3.4 CASE

当需要根据 SPL 变量或单一表达式的值从许多分支中取用一个时，使用 CASE 语句。CASE 语句是对 IF 语句的快速替换。

**语法**




ELSE 子句


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *constant_expr* | 指定精确值的表达式 | 必须是精确数值、引用字符串、精确日期时间或精确时间间隔。数据类型必须与数据 *value_expr* 相一致 | 常量表达式 |
| *value_expr* | 返回值的表达式 | SPL 变量或者返回值或 NULL 的其它类型的表达式。该数据类型不能是大对象（BLOB 、BYTE 、CLOB 、TEXT）、集合或者用户定义的 OPAQUE 或 DISTINCT 类型。在内置 OPAQUE 类型中，只有 BOOLEAN 和 LVARCHAR 是有效的。 | 表达式 |

**用法**

可以使用 CASE 语句在 SPL 例程中创建一组条件分支。WHEN 和 ELSE 子句都是可选的，但您必须提供其中一个或另一个。如果既不指定 WHEN 子句也不指定 ELSE 子句。则会收到语法错误消息。

- 如果不包含 WHEN 子句也不包含 ELSE 子句，则 CASE 语句由于语法错误而失败。
- 如果不包含 ELSE 子句，但是 WHEN 子句没有指定一个与 ***value_expr*** 相匹配的 ***constant expr*** t，则 CASE 语句在例程执行时产生错误 -26062 并失败。

不要将 CASE 语句和 SQL 的 CASE 表达式混淆。（CASE 表达式支持作为 CASE 语句的同一关键字。但是使用不同的语法和语义计算指定的 ***conditions*** 。CASE 表达式会返回一个值或 NULL，如 CASE 表达式 中描述。）

**数据库服务器如何执行 CASE 语句**

数据库服务器按一下操作的顺序的执行 CASE 语句：

- 数据库服务器计算 ***value_expr*** 表达式。

- 如果得到的值与在 WHEN 子句的 ***constant_expr*** 参数中指定的精确值相匹配，则数据库服务器执行跟随该 WHEN 子句中的 THEN 关键字的语句块。
- 如果计算 ***value_expr*** 参数得到的值与多个 WHEN 子句中的 ***constant_expr*** 参数相匹配，则数据库服务器执行跟随 CASE 语句中第一个匹配的 WHEN 子句中 THEN 关键字的语句块。（在这种情况中，WHEN 子句的词法顺序可以确定 CASE 语句的结果。）如果数据库服务器执行了跟随在 THEN 关键字之后的 GOTO 语句，则数据库服务器将程序控制传递给指定的语句标签。否则，数据库服务器执行 SPL 例程中标识当前 CASE 语句结束的 END CASE 关键字后面的下一个 SPL 语句或 SQL 语句。
- 如果 ***value_expr*** 参数计算的值与任何 WHEN 子句的 ***constant_expr*** 参数中指定的精确值不匹配，并且如果 CASE 语句包含 ELSE 子句，则数据库服务器执行跟随在 ELSE 关键字的语句块。如果数据库服务器执行了跟随在 ELSE 关键字之后的 GOTO 语句，则数据库服务器将程序控制传递给指定的语句标签。否则，数据库服务器执行 SPL 例程中标识当前 CASE 语句结束的 END CASE 关键字后面的下一个 SPL 语句或 SQL 语句。
- 如果 ***value_expr*** 参数计算的值与任何 WHEN 子句的 ***constant_expr*** 参数中指定的精确值不匹配，并且如果 CASE 语句不包含 ELSE 子句，则数据库服务器发出异常，并且 CASE 语句失败，错误 -26062。SPL 例程是否终止或继续执行取决于其异常处理逻辑。

CASE 语句的此实现非常类似于 GBase 8s Parallel Server 的实现，除了当未指定 ELSE 子句且没有 WHEN 子句与 ***value_expr*** 参数相匹配时，GBase 8s Parallel Server 不会发生错误。在这种情况下，程序执行将在紧跟 CASE 语句之后的 SPL 或 SQL 语句中继续。

更随 THEN 或 ELSE 关键字的语句块可以包括在 SPL 例程的语句块中有效的任何 SQL 语句或 SPL 语句。有关更多信息，请参阅语句块。

**在 CASE 语句中值表达式的计算**

数据库服务器仅计算一次 ***value_expr*** 参数的值。它在开始执行 CASE 语句时计算此值。如果该参数中的指定表达式包含一个或多个 SPL 变量。并且这些变量中的任何变量的值随后在 CASE 语句中的一个语句块中修改，则数据库服务器不会重新计算 ***value_expr*** 参数的值。因此，在 ***value_expr*** 参数中指定的变量值的任何更改都不好影响 CASE 语句采用的分支。

**CASE 语句的示例**

在以下示例中，CASE 语句根据另一个命名为 **i** 的 SPL 变量的值将 SPL 变量集合（命名为 **j** 、**k** 、**l** 和 **m**）之一初始化为命名为 **x** 的 SPL 变量的值：

```
CASE i
WHEN 1 THEN LET j = x;
WHEN 2 THEN LET k = x;
WHEN 3 THEN LET l = x;
WHEN 4 THEN LET m = x;
ELSE
RAISE EXCEPTION 100; --invalid value
END CASE;
```

此处每个 WHEN 子句指定一个整数作为它的常量表达式，假设值表达式具有数据类型。（如果这些精确值已用引号分隔，则数据库服务器将它们视为字符值。）

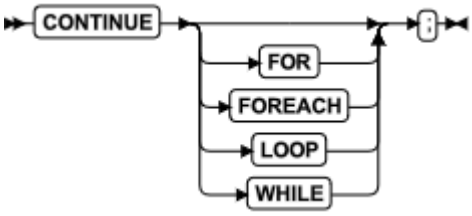
下面的示例包含 WHEN 子句中对 NULL 的测试，其表达式的值和常量表达式的数据类型为 CHAR(1)：

```
CREATE PROCEDURE case_proc( )
        RETURNING CHAR(1);
        DEFINE grade CHAR(1);
        LET grade = 'D';
        CASE grade
        WHEN 'A' THEN LET grade = 'a';
        WHEN 'B' THEN LET grade = 'b';
        WHEN 'C' THEN LET grade = 'c';
        WHEN NULL THEN LET grade = 'z';
        ELSE LET grade = 'd';
        END CASE;
        RETURN grade;
END PROCEDURE;
```

## 3.5 CONTINUE

使用 CONTINUE 语句启动最里面的 FOR 、LOOP 、WHILE 或 FOREACH 循环的下一个迭代。

**语法**




**用法**

当执行控制传递到 CONTINUE 语句时，SPL 例程跳过指定类型的最里面循环中的其余语句。执行在顶层循环继续下一个迭代。

在以下示例中，loop_skip 函数将值 3 到 15 插入到表 testtable 中。该函数还在此过程中返回 3 到 9 和 13 到 15 。该函数不返回值 11 因为它遇到 CONTINUE FOR 语句。CONTINUE FOR 语句使函数跳过 RETURN WITH RESUME 语句：

```
CREATE FUNCTION loop_skip()
        RETURNING INT;
        DEFINE i INT;
        ...
        FOR i IN (3 TO 15 STEP 2)
        INSERT INTO testtable values(i, null, null);
        IF i = 11
        CONTINUE FOR;
```

```
        END IF;
        RETURN i WITH RESUME;
        END FOR;

END FUNCTION;
```

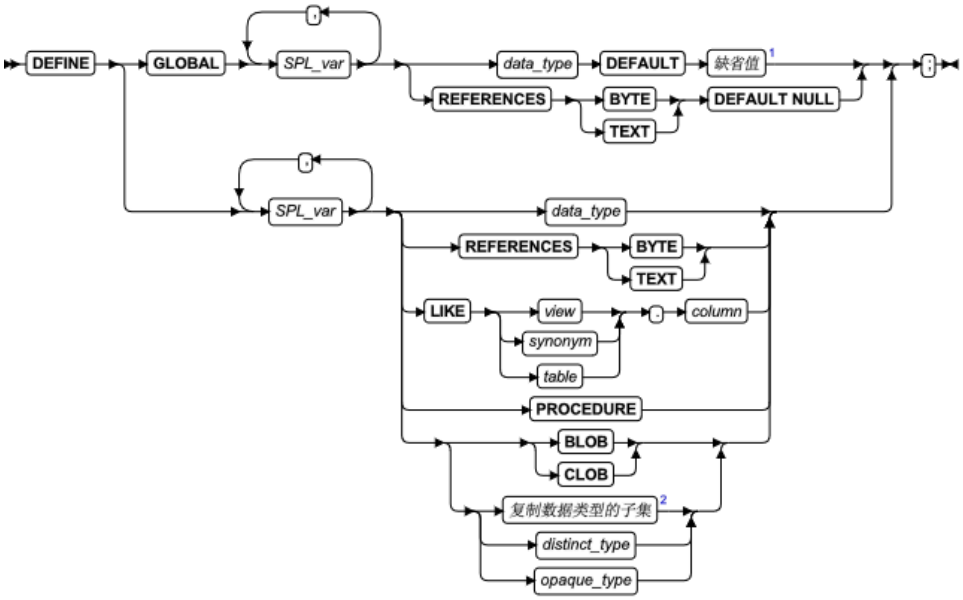
就像 EXIT 语句一样（EXIT），FOREACH 语句和 FOR 或 WHILE 语句不包含 LOOP 关键字，FOR 、WHILE 或 FOREACH 关键字必须紧跟在 CONTINUE 关键字之后来指定循环的类型。如果指定的循环类型与 CONTINUE 语句发出的上下文不匹配，则生成错误。

在 LOOP 、FOR LOOP 和 WHILE LOOP 语句中，不管标签的还是未标记的，关键字指示 CONTINUE 关键字之后的循环的类型是可选的，但是，如果您指定与循环类型不对应的关键字，则 GBase 8s 发出错误。

## 3.6 DEFINE

使用 DEFINE 语句声明 SPL 例程使用的本地变量，或声明可由几个 SPL 例程共享的全局变量。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 列名 | 必须已经存在于 *table* 或 *view* 中 | 标识符 |
| *data_type* | *SPL_var* 的类型 | 请参阅声明全局变量 | 数据类型 |
| *distinct_type* | Distinct 类型 | 必须已经在数据库中定义 | 数据类型 |
| *opaque_type* | Opaque 类型 | 必须已经在数据库中定义 | 数据类型 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *SPL_var* | 新 SPL 变量 | 在语句块中必须是唯一的 | 标识符 |
| *synonym, table, view* | 表、视图或同义词的名称 | 当发出 DEFINE 时它所指向的同义词和表或视图必须存在 | 标识符 |

**用法**

DEFINE 语句不是可执行语句。DEFINE 语句必须出现在例程头之后，任何其它语句之前。如果声明一个局部变量（通过使用不带 GLOBAL 关键字的 DEFINE），则它的引用作用域是定义了此局部变量的语句块。您可以在该语句块中使用该变量。在语句块之外具有不同定义的另一个变量可有相同的名称。

具有 GLOBAL 关键字的变量在作用域中是全局的并且在语句块之外且对于其它 SPL 例程是可用的。全局变量可以是除 BIGSERIAL 、BLOB 、BYTE 、CLOB 、SERIAL 、SERIAL8 或 TEXT 之外的任何内置数据类型。本地变量可以是除 BIGSERIAL 、BYTE 、SERIAL 、SERIAL8 或 TEXT 之外的任何内置数据类型。如果 ***column*** 是 BIGSERIAL 、SERIAL 或 SERIAL8 数据类型，则声明 BIGINT 、INT 或 INT8 变量（分别地）以存储其值。

将 SQL 关键字的名称或其它数据库对象的标识符声明为 SPL 变量会在某些上下文中产生错误或意外结果。有关涉及 SPL 变量的名称冲突的一些潜在问题的讨论，请参阅下面的相关概念。

## 引用 TEXT 和 BYTE 变量

REFERENCES 关键字使您能够使用 BYTE 和 TEXT 变量。这些不包括实际数据，但是是实际数据的指针。REFERENCES 关键字表示 SPL 变量仅仅是一个指针。您可以使用 BYTE 和 TEXT 变量就像您使用 SPL 中的任何其它变量一样。

## 重新声明或重新定义

如果您在同一语句块中将相同变量定义两次，则会收到错误消息。您可以在嵌套块中重新定义变量，在此情况下它临时隐藏外层声明。以下示例会产生错误：

```
CREATE PROCEDURE example1()
DEFINE n INT; DEFINE j INT;
DEFINE n CHAR (1);      -- 重定义产生错误
```

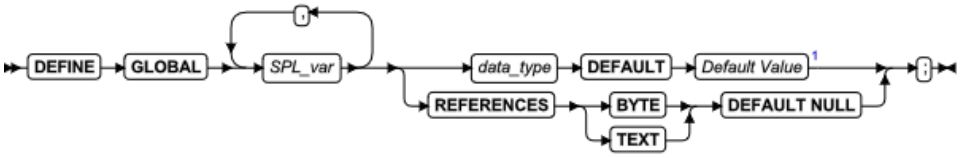
重新声明在以下示例中是有效的。在嵌套语句块中，n 是一个字符变量。在块之外，n 是一个整型变量。

```
CREATE PROCEDURE example2()
DEFINE n INT; DEFINE j INT;
...
BEGIN
DEFINE n CHAR (1);      -- character n masks global integer variable
...
```

END;

## 声明全局变量

**语法：**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | *SPL_var* 的类型 | 请参阅 声明全局变量. | 数据类型 |
| *SPL_var* | 新的 SPL 变量 | 在语句块中必须是唯一的 | 标识符 |

GLOBAL 关键字表示跟随的变量的引用作用域包含所有在给定的 DB-Access 或 SQL 管理 API 会话中运行的 SPL 例程。这些变量的数据类型必须与***全局环境***中的变量数据类型相匹配。全局变量是由所有 SPL 例程使用的内存。这些例程在给定的 DB-Access 或 SQL 管理 API 会话中运行。全局变量的值存储在内存中。

在当前会话中运行的 SPL 例程共享全局变量。因为数据库服务器不将全局变量保存在数据库中，所有当前会话关闭时不保留全局变量。

全局变量的第一个声明建立全局环境中的变量；后续全局声明只是将该变量绑定到全局环境并在此时建立变量的值。

以下示例显示两个 SPL 过程，**proc1** 和 **proc2**；每个过程都已定义了全局变量 **gl_out**：

- SPL procedure **proc1**

```
CREATE PROCEDURE proc1()
...
DEFINE GLOBAL gl_out INT DEFAULT 13;
...
 LET gl_out = gl_out + 1;
 END PROCEDURE;
```

- SPL procedure **proc2**

```
CREATE PROCEDURE proc2()
...
DEFINE GLOBAL gl_out INT DEFAULT 23;
DEFINE tmp INT;
   ...
LET tmp = gl_out
END PROCEDURE;
```

如果首先调用 **proc1**，则 **gl_out** 被设置为 13，然后增加为 14。如果接下来调用 **proc2**，它会发现 **gl_out** 已被定义，这样就不应用缺省值 23 。然后，**proc2** 将现有值 14 分配给 **tmp**。如果首

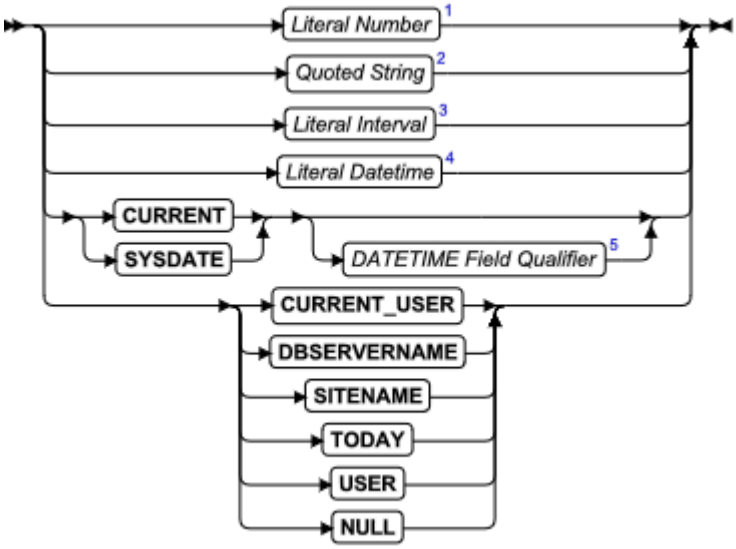
先调用了 **proc2** ，则 **gl_out** 会被设置为 23，并且 23 可能被分配给 **tmp**。以后对 **proc1** 的调用将不应用缺省值 13。

不同数据库服务器实例的数据库不共享全局变量，但是在单个会话中，同一个数据库服务器实例的所有数据库可以共享全局 SPL 变量。然而，数据库服务器和任何应用程序开发工具都不共享全局变量。

## 缺省值

全局变量可以拥有文字值、NULL 值 或者系统常量缺省值。

缺省值




如果您指定缺省值，则全局变量使用指定的值初始化。

## CURRENT

CURRENT 是仅用于 DATETIME 变量的有效缺省值。如果 YEAR TO FRACTION(3) 是其声明的精度，则不需要限定符。否则，当 CURRENT 是缺省值时必须指定相同的 DATETIME 限定符，如以下 DATETIME 变量的示例所示：

```
DEFINE GLOBAL d_var DATETIME YEAR TO MONTH
DEFAULT CURRENT YEAR TO MONTH;
```

## SYSDATE

SYSDATE 是仅用于 DATETIME 变量的有效缺省值。如果 YEAR TO FRACTION(5) 是其声明的精度，则不需要限定符。否则，当 SYSDATE 是缺省值时必须指定相同的 DATETIME 限定符，如以下 DATETIME 变量的示例所示：

```
DEFINE GLOBAL dt_var DATETIME YEAR TO DAY
DEFAULT SYSDATE YEAR TO DAY;
```

## USER

如果使用 **USER** 的值，或者它的同义词 **CURRENT_USER** 返回的值作为缺省值，则变量必须定义为 CHAR 、VARCHAR 、NCHAR 或 NVARCHAR 数据类型。建议变量的长度最少为 32 个字节。如果变量长度太短而无法存储缺省值，则在 INSERT 和 ALTER TABLE 操作期间会有得到错误消息的风险。

## TODAY

*如果您使用 **TODAY** 作为缺省值，则该变量必须是 DATE 值。（有关 **TODAY** 和可出现在缺省值子句中的其它系统常量的描述，请参阅*常量表达式。）

## BYTE 和 TEXT

对于 BYTE 或 TEXT 变量，唯一有效的缺省值是 NULL。以下示例定义称为 l_blob 的 TEXT 全局变量：

```
CREATE PROCEDURE use_text()
DEFINE i INT;
DEFINE GLOBAL l_blob REFERENCES TEXT DEFAULT NULL;
...
END PROCEDURE
```

这里需要 REFERENCES 关键字，因为 DEFINE 语句无法直接声明 BYTE 或 TEXT 数据类型；**l_blob** 变量是指向存储在全局环境中的 TEXT 值的指针。

## SITENAME 或 DBSERVERNAME

如果您声明 SITENAME 或 DBSERVERNAME 关键字作为缺省值，则变量必须是 CHAR 、VARCHAR 、NCHAR 、NVARCHAR 或 LVARCHAR 数据类型。它的缺省值是数据库服务器在运行时的名称。建议该变量的大小至少为 128 个字节。如果该变量的长度太短而无法存储缺省值，则在 INSERT 和 ALTER TABLE 操作期间会有得到错误消息的风险。

以下示例使用 SITENAME 关键字指定缺省值。该示例还将 BYTE 全局变量初始化为 NULL：

```
CREATE PROCEDURE gl_def()
DEFINE GLOBAL gl_site CHAR(200) DEFAULT SITENAME;
DEFINE GLOBAL gl_byte REFERENCES BYTE DEFAULT NULL;
...
END PROCEDURE
```

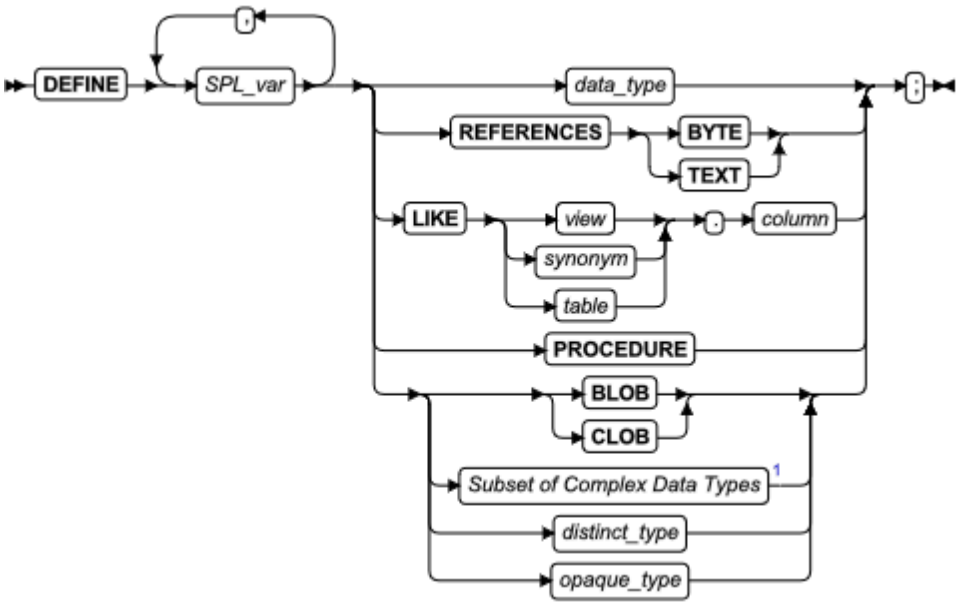
## 声明本地变量

***本地变量***有其引用作用域，在该作用域中声明例程。如果省略 GLOBAL 关键字，则在 DEFINE 语句中声明的任何变量都是本地变量，并且在其它 SPL 例程中不可见。

出于此原因，声明相同名称的本地变量的不同 SPL 例程可在同一 DB-Access 或 SQL 管理 API 会话中运行而无冲突。

如果本地变量和全局变量具有相同的名称，那么全局变量在声明了本地变量的 SPL 例程中不可见。（在所有其它 SPL 例程中，只有全局变量在作用域中。）

**语法：**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 列名 | 必须已经存在于 *table* 或 *view* 中 | 标识符; |
| *data_type* | *SPL_var* 的类型 | 不能是 BIGSERIAL 、BYTE 、SERIAL 、SERIAL8 或 TEXT | 数据类型 |
| *distinct_type* | Distinct 类型 | 必须已经在数据库中定义 | 标识符 |
| *opaque_type* | Opaque 类型 | 必须已经在数据库中定义 | 标识符 |
| *SPL_var* | 新的 SPL 变量 | 在语句块中必须是唯一的 | 标识符; |
| *synonym*, *table*, *view* | 表、视图或同义词的名称 | 当发出此语句时它所指向的同义词和表或视图必须存在 | 数据库对象名 |

本地变量不支持缺省值。以下示例显示本地变量的典型定义：

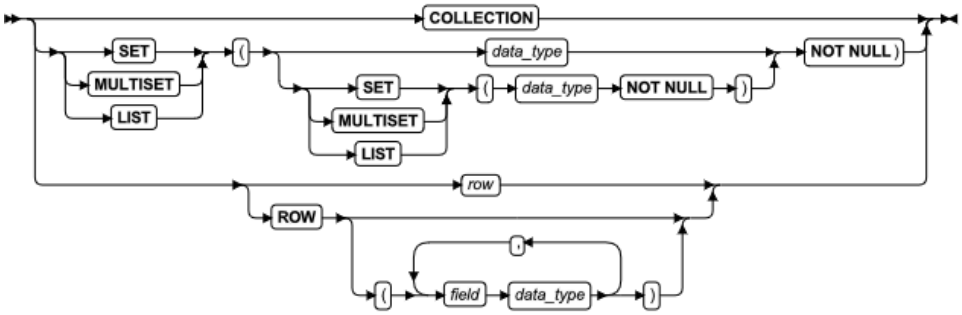
```
CREATE PROCEDURE def_ex()
        DEFINE i INT;
        DEFINE word CHAR(15);
        DEFINE b_day DATE;
        DEFINE c_name LIKE customer.fname;
```

```
        DEFINE b_text REFERENCES TEXT;
END PROCEDURE
```

## 复杂数据类型的子集

可以使用以下语法将 SPL 变量声明为已归类的或类属集合，或者已命名的、未命名的或类属 ROW 数据类型。

复杂数据类型（子集）




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | 集合元素的类型或未命名的 ROW 类型字段的类型 | 必须与变量将存储的值的数据类型相匹配。不能是 BIGSERIAL、 BLOB 、BYTE 、CLOB 、SERIAL 、SERIAL8 或 TEXT。 | 数据类型 |
| *field* | 未命名的 ROW 字段 | 必须在数据库中存在 | 标识符 |
| *row* | 已命名的 ROW 数据类型 | 必须在数据库中存在 | 标识符 |

## 声明集合变量

类型为 COLLECTION 、SET 、MULTISET 或 LIST 的本地变量可保留取自数据库的值的集合，您不能（使用 GLOBAL 关键字）或使用缺省值将集合变量定义为全局（用于关键字）。

使用 COLLECTION 关键字声明的变量是可保留任何数据类型集合的未归类的(或类属)集合变量。

已声明为 SET 、MULTISET 或 LIST 类型的变量是***已归类的集合变量***。它只能保留指定的数据类型的集合。

当您定义已归类的集合变量的元素时，必须使用 NOT NULL 关键字，如下例所示：

```
DEFINE a SET ( INT NOT NULL );
DEFINE b MULTISET ( ROW ( b1 INT,
        b2 CHAR(50)
        ) NOT NULL );
```

```
DEFINE c LIST( SET( INTEGER NOT NULL ) NOT NULL );
```

通过变量 **c**，SET 中的 INTEGER 值和 LIST 中的 SET 值都被定义为 NOT NULL。

您可以使用嵌套的复杂类型定义集合变量以保留匹配的嵌套复杂类型数据。允许任何类型或深度的嵌套。您可以在集合类型中嵌套 ROW 类型，在 ROW 类型中嵌套集合类型，在集合类型中嵌套集合类型，在集合和 ROW 类型中嵌套 ROW 类型，等等。

如果您将某个变量声明为 COLLECTION 类型，并且在相同的语句块中被重新指定，则该变量获取各不相同的数据类型声明，如下例所示：

```
DEFINE a COLLECTION;
LET a = setB;
...
LET a = listC;
```

在此示例中，**varA** 是一个类属集合变量，它将其数据类型更改为当前指定的集合的数据类型。第一个 LET 语句使 **varA** 成为 SET 变量。第二个 LET 语句使 **varA** 成为 LIST 变量。

## 声明 ROW 变量

ROW 变量保留从命名的或未命名的 ROW 类型的数据。您可以定义类属 ROW 变量，已命名的 ROW 变量或未命名的 ROW 变量。

使用 ROW 关键字定义的类属 ROW 变量可以保留来自任何 ROW 类型的数据。已命名的 ROW 变量保留来自变量声明中指定的已命名的 ROW 类型的数据。

以下语句显示了类属 ROW 变量和已命名的 ROW 变量的示例：

```
DEFINE d ROW;                    -- generic ROW variable
DEFINE rectv rectangle_t;        -- named ROW variable
```

已命名的 ROW 变量保留在变量声明中相同类型的已命名的 ROW 类型。

要定义一个变量（该变量将保留存储在未命名的 ROW 类型中的数据），请使用后面跟有 ROW 类型的字段的 ROW 关键字，如：

```
DEFINE area ROW ( x int, y char(10) );
```

未命名的 ROW 类型是仅由等同结构检查的类型。如果两个未命名的 ROW 类型具有相同的字段数量，并且具有相同的类型定义，那么这两个 ROW 类型被认为是等同的。因此，您可以将以下 ROW 类型中任意一个访存到上面定义的变量 area 中：

```
ROW ( a int, b char(10) )
ROW ( area int, name char(10) )
```

就像 ROW 类型可以有字段一样，ROW 变量也可以有字段。要给 ROW 变量的字段指定值，请使用限定符表示法 *variableName.fieldName* ，其后跟随表达式，如以下示例中所示：

```
CREATE ROW TYPE rectangle_t (start point_t, length real, width real);
DEFINE r rectangle_t; -- Define a variable of a named ROW type
LET r.length = 45.5;     -- Assign a value to a field of the variable
```

当您给 ROW 变量指定值时，可以使用任何有效的表达式。

## 声明 Opaque 类型变量

Opaque 类型变量保留从不透明数据类型中检索出的数据，这些不透明数据类型是用 CREATE OPAQUE TYPE 语句创建的。不透明类型变量只能保留定义它所依据的同一不透明类型的数据。以下示例定义 opaque 类型的变量 point，它保留二维点的 x 和 y 坐标：

```
DEFINE b point;
```

## 声明变量 LIKE 列

如果使用 LIKE 子句，则数据库服务器指定变量的数据类型和表、同义词或视图中指定列的数据类型相同。

定义为数据库列的变量的数据类型在运行时解析，因此在编译时不需要存在***列***和***表***。

可以使用 LIKE 关键字声明某个变量类似于 serial 列，它声明：

- 如果列是 SERIAL 数据类型，就声明一个 INTEGER 变量
- 如果列是 SERIAL8 数据类型，就声明一个 INT8 变量
- 如果列是 BIGSERIAL 数据类型，就声明一个 BIGINT 变量

例如，如果 mytab 表中的 serialcol 列有 SERIAL 数据类型，则可以创建以下的 SPL 函数：

```
CREATE FUNCTION func1()
DEFINE local_var LIKE mytab.serialcol;
RETURN;
END FUNCTION;
```

变量 **local_var** 被视为 INTEGER 变量。

## 使用逻辑字符语义定义变量

当为当前会话指定了 SQL_LOGICAL_CHAR 配置参数，设置为 'ON'或比 1 大的值时，GBase 8s 在以下数据类型的 SPL 变量的声明中将大小声明解释为逻辑字符，而不是字节：

- CHAR 或 CHARACTER
- CHARACTER VARYING 或 VARCHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- 基于内置字符数据类型的 DISTINCT 类型
- 基于之前列出的数据类型的 DISTINCT 类型
- 任何先前列出的数据类型的 ROW 数据类型字段
- LIST 、MULTISET 或 SET 集合数据类型中的数据类型的元素

为数据库语言环境启用逻辑字符语义可确保有足够的存储空间可供数据类型存储指定数量的逻辑字符。SPL 变量的结构字节大小是数据类型的声明大小乘以 SQL_LOGICAL_CHAR 值，如果为 2 、

3 或 4 或者（如果 SQL_LOGICAL_CHAR 设置为 'ON'）乘以数字的乘积的数据库语言环境的代码集中最大的逻辑字符所需的存储字节数。

如果客户端会话连接到在创建数据库时启用了 SQL_LOGICAL_CHAR 配置参数的数据库，则该设置将在连接时生效。

在数据类型声明中使用 LIKE 关键字的 DEFINE 语句创建其数据类型与 LIKE 规范引用的列的模式匹配的 SPL 变量。如果已定义，SQL_LOGICAL_CHAR 设置对 DEFINE 使用 LIKE 关键字声明的变量的内存大小没有影响。

有关使用多字节代码集（如 **UTF-8**）的语言环境中 SQL_LOGICAL_CHAR 设置的影响的详细信息，其中单个逻辑字符可能需要多个字节的存储，请参阅 *GBase 8s 管理员参考手册*中有关 SQL_LOGICAL_CHAR 配置参数的描述。有关多字节语言环境和逻辑字符的其它信息，请参阅 *GBase 8s GLS 用户指南*。

## 声明变量为 PROCEDURE 类型

PROCEDURE 关键字表示在当前作用域中，变量是对 UDR 的调用。

DEFINE 语句不支持 FUNCTION 关键字。不管您在调用用户定义的过程还有用户定义的函数，都使用 PROCEDURE 关键字。

将变量声明为 PROCEDURE 类型表示在当前语句作用域中，该变量不是对内置函数的调用。例如，以下语句定义 length 作为 SPL 例程，而不是作为内置 LENGTH 函数：

```
DEFINE length PROCEDURE;
...
LET x = length (a,b,c)
```

此定义在语句块的作用域中禁用内置 LENGTH 函数。如果您已使用名称 **length** 创建了用户定义的例程，则将使用这样的定义。

如果您使用与聚集函数（SUM 、MAX 、MIN 、AVG 、COUNT）相同的名称或使用名称 **extend** 创建 SPL 例程，则必须将例程名称限制为所有者名称。

## 为 BYTE 和 TEXT 数据声明变量

REFERENCES 关键字表示该变量不包含 BYTE 或 TEXT 值，但是是指向 BYTE 或 TEXT 值的指针。可以如同它保存了该值一样使用此变量。

以下示例定义本地 BYTE 变量：

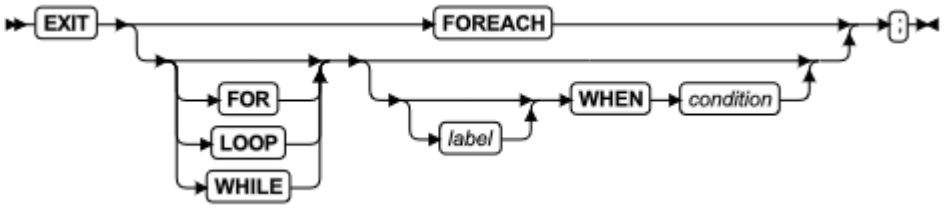
```
CREATE PROCEDURE use_byte()
DEFINE i INT;
DEFINE l_byte REFERENCES BYTE;
END PROCEDURE --use_byte
```

如果将 BYTE 或 TEXT 数据类型的变量传递给 SPL 例程，则该数据会传递到数据库服务器并存储在根数据库空间或者 **DBSPACETEMP** 环境变量指定的数据库空间（如果已设置）。您不需要指定保留数据的文件的位置或名称。仅需要 BYTE 或 TEXT 变量在例程中被定义时的名称。

## 3.7 EXIT

EXIT 语句可以终止 FOR 、FOREACH 、LOOP 或 WHILE 语句。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *condition* | 当此计算为 TRUE 时循环终止 | 如果 *condition* 计算为 FALSE，则循环继续。 | 条件 |
| *label* | 要退出的循环的标签 | 必须是包含 EXIT 语句的循环语句的标签 | 标识符 |

**用法**

EXIT 语句从迭代传输执行的控制，使封闭语句类型（FOR 、FOREACH 、LOOP 或 WHILE）的最内层循环终止。如果未指定循环标签或 WHEN 添加，则在当前 FOR 、FOREACH 、LOOP 或 WHILE 语句之后的第一个语句处恢复执行。

### 从 FOREACH 语句 EXIT

如果 EXIT 语句将 FOREACH 语句作为其最内层的结束语句，则 FOREACH 关键字必须跟在 EXIT 关键字之后。如果 FOREACH 语句没有以 EXIT FOREACH 语句结束，则 EXIT FOREACH 语句会无条件地终止 FOREACH 语句。

以下程序段包含 EXIT FOREACH 语句：

```
FOREACH cursor1 FOR
 SELECT * INTO a FROM TABLE(b);
        IF a = 4 THEN
        DELETE FROM TABLE(b)
        WHERE CURRENT OF cursor1;4
        EXIT FOREACH;
        END IF;
END FOREACH;
```

# 从 FOR 、LOOP 或 WHILE 循环 EXIT

如果 EXIT 语句在 FOREACH 语句外发出，则它返回一个错误，除非它是从作为其最内层的 FOR 、FOR LOOP 、LOOP 、WHILE LOOP 或 WHILE 语句发出。在不包含 LOOP 关键字的 FOR 或 WHILE 语句中，EXIT 关键字之后需要相应的 FOR 或 WHILE 关键字。执行从发出 EXIT 语句的最内层循环之后的第一个可执行语句处恢复。

当 EXIT 语句从 FOR LOOP 、LOOP 或 WHILE LOOP 语句发出时，不需要其它关键字，带有或不带有循环标签，但如果 EXIT 关键字之后包含 FOR 、LOOP 或 WHILE 关键字，则该关键字必须对应于从其发出 EXIT 语句的循环的类型。

如果 EXIT 关键字后面跟着循环标签的标识符，并且没有指定 ***condition***，那么在指定了标签的 FOR 、FOR LOOP 、LOOP 、WHILE LOOP 或 WHILE 语句之后的第一个可执行语句将继续执行，这使得 EXIT 语句能够从嵌套循环中退出（如果标记了外层循环）。

如果 WHEN ***condition*** 后跟着 EXIT 或 EXIT ***label*** 规范，则 EXIT 不会生效直到 ***condition*** 为真。如果条件为真，则如果没有指定标签，则在已标记循环之后或最内循环之后继续执行。

如果数据库服务器没有找到指定循环或循环标签，则 EXIT 语句失败。如果从 FOR 、FOREACH 、LOOP 或 WHILE 语句外发出 EXIT ，它将产生错误。

以下示例使用 EXIT FOR 语句。在 FOR 循环中，当 j 变为 6 时， WHILE 循环中的 IF 条件 i = 5 为真。FOR 循环停止执行，SPL 过程继续 FOR 循环之外的下一个语句（在这种情况下，是 END PROCEDURE 语句）。在该示例中，当 j 等于 6 时，该过程结束：

```
CREATE PROCEDURE ex_cont_ex()
        DEFINE i,s,j, INT;
        FOR j = 1 TO 20
        IF j > 10 THEN
        CONTINUE FOR;
        END IF
        LET i,s = j,0;
        WHILE i > 0
        LET i = i -1;
        IF i = 5 THEN
        EXIT FOR;
        END IF
        END WHILE
        END FOR
        END PROCEDURE;
```

以下程序片段显示了在标记的 WHILE LOOP 语句中的两个条件 EXIT 语句，它们嵌套在另一个标记为 LOOP 的语句中：

```
        <<outer>>
        LOOP
        LET x = x+1;
        <<inner>>
```

```
WHILE ( i >10 ) LOOP
LET x = x+1;
EXIT inner WHEN x = 2;
EXIT outer WHEN x > 3;
END LOOP inner;
LET x = x+1;
END LOOP outer;
```

当 x=2 条件为真时，EXIT inner 语句将控制传递给 **inner** 标签的循环后的 LET 语句。当 x>3 条件为真时，EXIT outer 语句终止 **outer** 循环的执行。

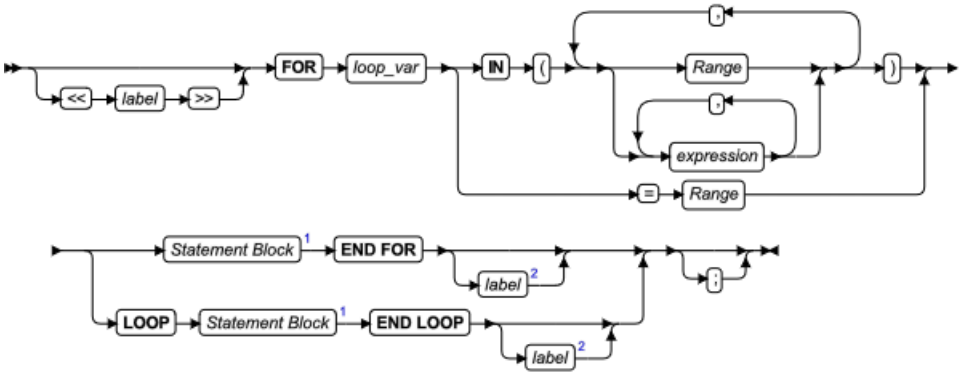
**相关语句**

<< Label >> 语句 、FOR 、FOREACH 、LOOP 、WHILE

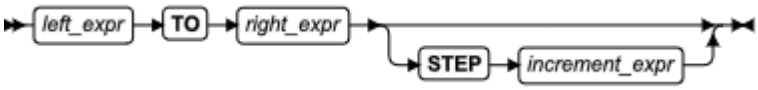
## 3.8 FOR

当您希望保证循环的终止时，请使用 FOR 语句来初始化受控的（限定的）循环。FOR 语句使用表达式或范围运算符来为循环指定有限数量的迭代。

**语法**




Range




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 与 *loop_var* 相比较的值 | 必须与 *loop_var* 数据类型相匹配 | 表达式 |
| *increment_expr* | *loop_var* 增加的正值或负值。缺省为 1（如果 *left_expr* < *right_expr*），或者 -1 （如果 | 必须返回整数。不能返回 0。 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | *left_expr* > *right_expr*) | | |
| *label* | 此循环的循环标签的名称 | 必须存在且在 SPL 例程中的标签名中必须是唯一的 | 标识符 |
| *left_expr* | 开始范围的表达式 | 值必须与 *loop_var* 的 SMALLINT 或 INT 数据类型相匹配，但是 *left_expr* 不能等于 *right_expr* | 表达式 |
| *loop_var* | 确定指定循环次数的变量 | 必须在该语句块的作用域内定义 | 标识符 |
| *right_expr* | 范围中的结束表达式 | 同 *left_expr* | 表达式 |

**用法**

数据库服务器在 FOR 语句执行之前计算所有表达式的值。如果其中一个或多个表达式是其值在循环期间更改的变量，则该更改对循环的迭代没有影响。

您可以将来自 SELECT 语句的输出用作***表达式***。

当 ***loop_var*** 等于表达式列表或连接的范围中的每个元素的值，或当它遇到 EXIT FOR 语句时，FOR 循环终止。但如果在 FOR 语句的主体中的赋值尝试修改 ***loop_var*** 值，会发生错误。

相对于 *left_expr* 的 *right_expr* 大小确定该范围是否按照正增量或负增量单步遍历：

- 如果 ***left_expr*** < ***right_expr*** ，则增量为正。
- 如果 ***left_expr*** > ***right_expr*** ，则增量为负。

如果未指定 ***increment_expr***，则每一步的缺省大小为 1，由以上的规则决定是正或负。

## 使用 TO 关键字定义范围

TO 关键字表示范围运算符。范围由 ***left_expression*** 和 ***right_expression*** 定义，并且 STEP ***increment_expr*** 选项隐式地设置增量数。如果使用 TO 关键字，***loop_var*** 必须是 INT 或 SMALLINT 数据类型。

以下示例显示了两个等同的 FOR 语句。每个语句使用 TO 关键字定义范围。第一个语句使用 IN 关键字，第二个语句使用等号（ = ）。每个语句使循环执行五次：

```
FOR index_var IN (12 TO 21 STEP 2)
-- statement block
END FOR;
FOR index_var = 12 TO 21 STEP 2
```

```
-- statement block
END FOR;
```

如果省略 STEP 选项，则数据库服务器给予 ***increment_expr*** 的值为 -1 （如果 ***right_expression*** 小于 ***left_expression***），或 +1 （如果 ***right_expression*** 大于 ***left_expression***）。如果指定了 ***increment_expr*** ，则它必须为负数（如果 ***right_expression*** 小于 ***left_expression***）或者正数（如果 ***right expression*** 大于 ***left_expression***）。

以下示例中的两个语句是等价的。在第一个语句中，STEP 增量是显式的。在第二个语句中，STEP 增量隐式地为 1：

```
FOR index IN (12 TO 21 STEP 1)
-- statement block
END FOR;
FOR index = 12 TO 21
-- statement block
END FOR;
```

数据库服务器将 ***loop_var*** 的值参数为 ***left_expression*** 的值。在后续迭代中，服务器将 ***increment_expr*** 添加到 ***loop_var*** 的值并检查 ***increment_expr*** 以确定 ***loop_var*** 值是否仍在 ***left_expression*** 和 ***right_expression*** 之间。如果这样，会发生下一迭代。否则，会发生循环退出。或者，如果指定另一范围，变量会具有下一范围中的第一个元素的值。

## 在单个 FOR 语句中指定两个或更多的范围

以下示例显示向前或向后遍历循环并在每一方向使用不同增量值的语句：

```
FOR index_var IN (15 to 21 STEP 2, 21 to 15 STEP -3)
-- statement body
END FOR;
```

### 将表达式列表作为范围使用

数据库服务器将 ***loop_var*** 的值初始化为指定的第一个表达式的值。在后续的迭代中，***loop_var*** 值取决于下一个表达式的值。当数据库服务器已计算了列表中的最后一个表达式的值并使用了它时，循环停止。

IN 列表中的表达式无须是数值，只要不在 IN 列表中使用范围运算符。以下示例使用字符表达式列表：

```
FOR c IN ('hello', (SELECT name FROM t), 'world', v1, v2)
INSERT INTO t VALUES (c);
END FOR;
```

以下 FOR 语句显示了数值表达式列表的使用：

```
FOR index IN (15,16,17,18,19,20,21)
-- statement block
END FOR;
```

## 在同一 FOR 语句中混合范围和表达式列表

如果 *loop_var* 是 INT 或 SMALLINT 值，您可以在同一 FOR 语句中混合范围和表达式列表。以下示例显示使用整数变量的混合。表达式列表中的值包括从 SELECT 语句返回的值、整数变量和常量的和、从名为 p_get_int 的 SPL 函数返回的值和整数变量：

```
CREATE PROCEDURE for_ex ()
DEFINE i, j INT;
LET j = 10;
FOR i IN (1 TO 20, (SELECT c1 FROM tab WHERE id = 1),
          j+20 to j-20, p_get_int(99),98,90 to 80 step -2)
INSERT INTO tab VALUES (i);
END FOR;
END PROCEDURE;
```

## 指定已标记的 FOR 循环

要创建已标记的 FOR 循环，在最初的 FOR 关键字之前声明循环标签，在 END FOR 关键字之后重复此标签，如下所示：

```
CREATE PROCEDURE ex_cont_ex()
        DEFINE i,s,j, INT;
        <<for_lab>>
        FOR j = 1 TO 20
        IF j > 10 THEN
        CONTINUE FOR;
        END IF
        LET i,s = j,0;
        WHILE i > 0
        LET i = i -1;
        IF i = 5 THEN
        EXIT for_lab;
        END IF
        END WHILE
        END FOR for_lab
```

```
END PROCEDURE;
```

这里 EXIT for_lab 语句具有与 EXIT 或 EXIT FOR 关键字相同的作用，都终止了 FOR 循环和例程。在此示例中，包含 EXIT for_lab 语句的语句具有与 EXIT for_lab WHEN i = 5 相同的效果。

您还可以标记紧跟在初始 FOR 关键字之前的循环 <<***label***>> 规范开头的 LOOP 语句。在这种类型的循环中，CONTINUE LOOP 、EXIT LOOP 和 END LOOP 关键字会替换 CONTINUE FOR 、EXIT FOR 和 END FOR 关键字。在 CONTINUE 和 EXIT 关键字之后的 LOOP 和 FOR 关键字都是可选的，但是在包含 LOOP 关键字的 SPL 循环语句中需要 END LOOP 关键字。

您可以使用类似的语法创建一个未标记的循环，省略紧跟在初始 FOR 关键字之前的 <<***label***>> 规范。在这种情况下，还必须省略 END LOOP 关键字后面的未定义循环标签标识符。有关这些形式的带标签和为标记循环语句的描述和示例，请参阅 LOOP 语句，这些语句使您能够将 FOR 语句语法与有限数量的循环迭代结合在一起，并与 LOOP 语句的“永久循环”语法相结合使用。

相关语句

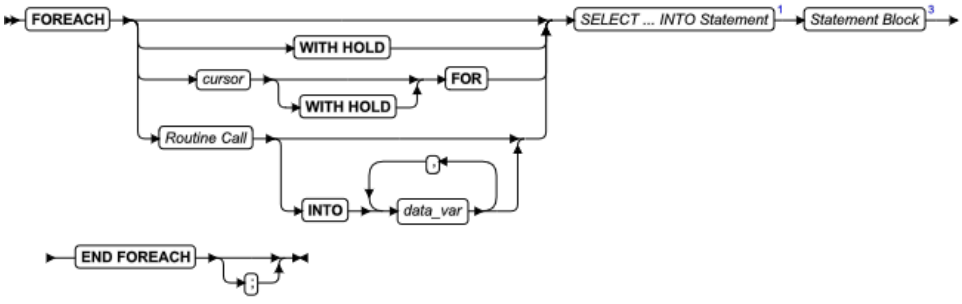
<< Label >> 语句 、CONTINUE 、EXIT 、LOOP 、FOREACH 、WHILE

## 3.9 FOREACH

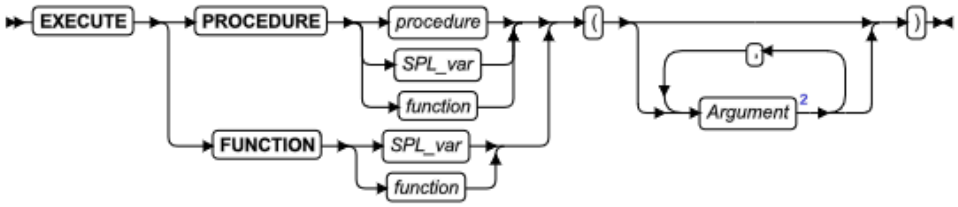
使用 FOREACH 语句声明一个直接游标，该游标可以查询和操纵查询返回的结果集的多个的行或者来自集合的多个元素。

**语法**

SPL 的 FOREACH 语句可以创建的直接顺序游标与 SQL 的 DECLARE 语句可以在 SPL 例程中创建的动态游标不同。（有关 SPL 例程中动态游标的语法和用法，请参阅在 SPL 例程中声明动态游标。）




例程调用



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *cursor* | 作为直接游标的名称而声明的标识符 | 在例程的游标名称、准备好的语句以及 SPL 变量中必须是唯一的 | 标识符 |
| *data_var* | 接收返回值的调用例程中的 SPL 变量 | *data_var* 的数据类型必须与返回的值相对应 | 标识符 |
| *function*, *procedure* | 要执行的 SPL 函数或过程 | 函数或过程必须存在 | 数据库对象名 |
| *SPL_var* | 包含要执行例程的名称的 SPL 变量 | 必须是 CHAR 、VARCHAR 、NCHAR 或 NVARCHAR 类型 | 标识符 |

**用法**

要执行 FOREACH 语句，数据库服务器应采用以下操作：

1. 它声明并隐式地打开游标。
2. 它从包含在 FOREACH 循环中的查询获取第一行，否则从调用的例程获取第一组值。
3. 它对变量列表中的每个变量赋值，所赋的值是来自 SELECT 语句或调用的例程创建的活动集合的相应值。
4. 它执行语句块。
5. 它从 SELECT 语句获取下一行或对每个迭代调用例程，并且重复步骤 3 和 4。
6. 当它发现没有其它行满足 SELECT 语句或调用的例程时，它会终止循环。当循环终止时，它关闭隐式游标。

因为语句块包含其它 FOREACH 语句，所以可以嵌套游标。对于嵌套的游标数量没有限制。

返回多个行、集合元素或值集合的 SPL 例程被称为***游标函数***。只返回一行或一个值的 SPL 例程是***非游标函数***。

该 SPL 过程使用 SELECT ... INTO 子句、显式地已命名游标和过程调用来说明 FOREACH 语句：

```
CREATE PROCEDURE foreach_ex()
DEFINE i, j INT;
FOREACH SELECT c1 INTO i FROM tab ORDER BY 1
INSERT INTO tab2 VALUES (i);
END FOREACH
FOREACH cur1 FOR SELECT c2, c3 INTO i, j FROM tab
IF j > 100 THEN
DELETE FROM tab WHERE CURRENT OF cur1;
CONTINUE FOREACH;
END IF
UPDATE tab SET c2 = c2 + 10 WHERE CURRENT OF cur1;
```

```
END FOREACH
FOREACH EXECUTE PROCEDURE bar(10,20) INTO i
INSERT INTO tab2 VALUES (i);
END FOREACH
END PROCEDURE; -- foreach_ex
```

当出现以下任意情形时，会关闭选择游标：

- 游标不再返回行。
- 游标是一个没有 HOLD 指定选择游标，并且使用 COMMIT 或 ROLLBACK 语句完成某项事务。
- 执行 EXIT 语句，它转移对 FOREACH 语句的控制。
- 在 FOREACH 语句的主体之内发生未俘获的异常。（请参阅 ON EXCEPTION。）
- 正执行此游标例程的调用例程中的游标（在 FOREACH 循环中）出于任何原因而关闭。

**注：**

FOREACH 语句不能定义 SCROLL 游标。每个 FOREACH 游标都是一个顺序游标，它只能获取来自活动集顺序中的下一行。FOREACH 定义的游标每次打开时只能读取活动集。

## 使用 SELECT ... INTO 语句

如在 FOREACH 的图表中所指出的，并非 SELECT 语句的所有子句和选项都可用在 FOREACH 语句中。FOREACH 语句中的 SELECT 语句必须包含 INTO 子句。它还可以包括 UNION 和 ORDER BY 子句，但不能使用 INTO TEMP 子句。关于 SELECT 语法和用法的完整描述，请参阅 SELECT 语句。变量列表中的每个变量的数据类型和计数必须与 SELECT ... INTO 语句返回的每个值相匹配。

如果您在 FOREACH 语句中包含分号（; ）以终止 SELECT ... INTO 规范，则该语句发出错误。例如，以下示例程序片段因产生语法错误而失败：

```
CREATE DBA PROCEDURE IF NOT EXISTS shapes()
DEFINE vertexes SET( point NOT NULL );
DEFINE pnt point;
SELECT definition INTO vertexes FROM polygons
WHERE id = 207;

FOREACH cursor1 FOR
SELECT * INTO pnt FROM TABLE(vertexes); -- Semicolon not valid
. . .
END FOREACH
. . .
END PROCEDURE;
```

在上面的示例中，您可以通过删除紧跟在 TABLE(vertexes) 规范之后的分号来避免此错误。

## 使用 SELECT 语句的 ORDER BY 子句

使用 SELECT 语句的 ORDER BY 子句表示查询返回多行。除非您使用 SQL 的 DECLARE 语句定义 Select 游标或 Function 游标，否则如果您在 FOREACH 循环的上下文之外指定 ORDER BY 子句以在 SPL 例程中单独处理返回的行，数据库服务器将发出错误。

有关 SPL 例程中 DECLARE 语句的语法和用法，请参阅在 SPL 例程中声明动态游标。

## 使用 Hold 游标

WITH HOLD 关键字指定（通过提交或回滚）关闭事务时游标仍保持打开。

## 更新或删除由游标名称标识的行

如果打算在 UPDATE 或 DELETE 语句（这些语句在 FOREACH 循环中 ***cursor*** 的当前行上操作）中使用 WHERE CURRENT OF ***cursor*** 子句，请在 FOREACH 循环中指定 ***cursor*** 名称。虽然不能在 FOREACH 语句的 SELECT ... INTO 段中包含 FOR UPDATE 关键字，但该游标的表现类似 FOR UPDATE 游标。

有关锁定的讨论。请参阅使用 Update 游标进行锁定部分。有关隔离级别的讨论，请参阅 SET ISOLATION 语句的描述。

## 使用集合变量

FOREACH 语句允许您为 SPL 集合变量声明游标。 这种游标称为***集合游标***。使用集合变量访问集合（SET、 MULTISET 、LIST）列的元素。当要访问集合变量中的一个或多个元素时，请使用游标。

以下来自 SPL 例程的摘抄显示了如果填充集合游标然后如何使用游标访问个别元素：

```
DEFINE a SMALLINT;
DEFINE b SET(SMALLINT NOT NULL);
SELECT numbers INTO b FROM table1 WHERE id = 207;
FOREACH cursor1 FOR
SELECT * INTO a FROM TABLE(b);
...
END FOREACH;
```

在此示例中，SELECT 语句从集合变量 **b** 中一次选择一个元素插入到元素变量 **a** 中。该投影列表是星号，因为集合变量 **b** 包含内置类型的集合。变量 **b** 与 TABLE 关键字作为 Collection-Derived Table 一起使用。有关更多信息，请参阅集合派生表。

下一个示例还显示了如果填充集合变量以及如果使用变量存取个别元素。但是，此示例在它的投影列表中使用 ROW 类型字段的列表：

```
DEFINE employees employee_t;
DEFINE n VARCHAR(30);
DEFINE s INTEGER;
SELECT emp_list into employees FROM dept_table
WHERE dept_no = 1057;
```

```
FOREACH cursor1 FOR
SELECT name,salary
INTO n,s FROM TABLE( employees ) AS e;
...
END FOREACH;
```

这里的集合变量 **employees** 包含一个 ROW 类型的集合。每个 ROW 类型包含 **name** 和 **salary** 字段。该集合查询每次查询一个名字和薪水，并将 **name** 放到 **n** 中将 **salary** 放到 **s** 中。AS 关键字声明 **e** 作为集合派生表 **employees** 的别名。只要 SELECT 语句执行，该别名就存在。

## 集合游标上的限制

当使用集合游标从集合变量获取个别元素时，FOREACH 语句具有以下限制：

- 它不能包含 WITH HOLD 关键字。
- 它必须在 FOREACH 循环中包含受限制的 SELECT 语句。

此外，与集合游标相关联的 SELECT 语句具有以下限制：

- 它的一般结构为 SELECT… INTO … FROM TABLE。该语句从按照 TABLE 关键字指定的集合变量中一次选择一个元素放入另一个被称为 ***element variable*** 的变量。
- 它不能在 Projection 列表中包含表达式。
- 它不能包含以下子句或选项：WHERE 、GROUP BY 、ORDER BY 、HAVING 、INTO TEMP 和 WITH REOPTIMIZATION。
- 该元素变量的数据类型必须与集合的元素类型相同。
- 元素变量的数据类型可以是任何 Opaque 、Distinct 或集合数据类型，或者任何除 BIGSERIAL 、BLOB 、BYTE 、CLOB 、SERIAL 、SERIAL8 或 TEXT 外的内置数据类型。
- 如果集合变量包含 Opaque 类型、Distinct 类型、内置类型或集合数据类型，则投影列表必须是星号（*）。
- 如果集合包含 ROW 类型，则投影列表可以是一个或多个字段名称的列表。

## 修改集合变量中的元素

要修改 SPL 例程中的集合的元素，必须首先使用 FOREACH 语句声明游标。

然后在 FOREACH 循环内，将集合变量用作 SELECT 查询中的集合派生的表，从集合变量一次选择一个元素。

当游标放置到要更新的元素上时，可以如下使用 WHERE CURRENT OF 子句：

- 带有 WHERE CURRENT OF 子句的 UPDATE 语句更新集合变量的当前元素中的值。
- 带有 WHERE CURRENT OF 子句的 DELETE 语句删除集合变量的当前元素中的值。

## 同 FOREACH 一起使用 Select 游标

当使用 FOREACH 语句时，如果来自查询的结果集要被更改，则不要使用此结果集作为 FOREACH 循环的退出条件。例如，如果 FOREACH 语句声明了一个期望返回 30 行的 Select 游标，但是 FOREACH 循环中的 DELETE 、INSERT 或 UPDATE 操作修改了此查询的结果集，这可能导致

意外行为。要确保 FOREACH 循环按照预期工作，请确保 FOREACH 语句中的任何 Select 游标在您开始修改结果集之前完成执行。

避免从对查询结果返回的行执行 DML 操作的 FOREACH 循环产生意外结果的一种方法是在 SELECT 语句中使用 ORDER BY 子句来实现结果集。

## 在 FOREACH 循环中调用 UDR

通常，使用以下准则用于从 SPL 例程调用另一个 UDR：

- 要调用用户定义的过程，请使用 EXECUTE PROCEDURE ***procedure name***。
- 要调用用户定义的函数，请使用 EXECUTE FUNCTION ***function name*** （如果用户定义的函数是用 CREATE PROCEDURE 语句创建的，则使用 EXECUTE PROCEDURE ***function name*** ）。

如果使用 EXECUTE PROCEDURE，则数据库服务器首先寻找具有您指定名称的用户定义的过程。如果它发现该过程，则数据库服务器执行它。如果它找不到该过程，则它寻找具有相同名称的用户定义的函数来执行。如果数据库服务器既找不到函数也找不到过程，则发出错误消息。如果使用 EXECUTE FUNCTION ，则数据库服务器寻找具有您指定名称的用户定义的函数。如果找不到该名称的函数，则数据库服务器发出错误消息。

SPL 函数可以返回零（0）个或多个值或行。

变量列表中的每个变量的数据类型和计数必须与函数返回的每个值相匹配。

相关的语句

CONTINUE 、EXIT 、FOR 、LOOP 、WHILE

# 3.10 GOTO

使用 GOTO 语句将程序执行的控制转移到具有指定语句标签的语句。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 此循环的循环标签的名称 | 在此 SPL 例程的标签中必须是唯一的 | 标识符 |

**用法**

GOTO 语句无条件地分支到语句标签。语句标签在其业务范围内必须是唯一的，并且必须在可执行语句之前。成功执行后，GOTO 语句将控制转移到标记的语句或语句块。

以下程序片段中，如果 j 变量的值大于 100 ，则 jump_back 函数将控制传递给有语句标签 back 的 LET 语句。

```
CREATE FUNCTION jump_back()
        RETURNING INT;
        DEFINE i,j INT;
        ...
        <<back>>
        LET j = j + i
        FOR i IN (1 TO 52 STEP 5)
        IF i < 11 THEN
        LET j = j + 3
        CONTINUE FOR;
        END IF;
        IF j > 100 THEN
        GOTO back
        END IF;
        RETURN j WITH RESUME;
        END FOR;
END FUNCTION;
```

GOTO 语句在 ON EXCEPTION 语句块中无效。

GOTO 语句引用的语句标签的标识符必须在数据库中存在，并在此 SPL 例程的语句标签和循环标签中必须是唯一的，而且必须在 GOTO 语句可以到达的作用域内。

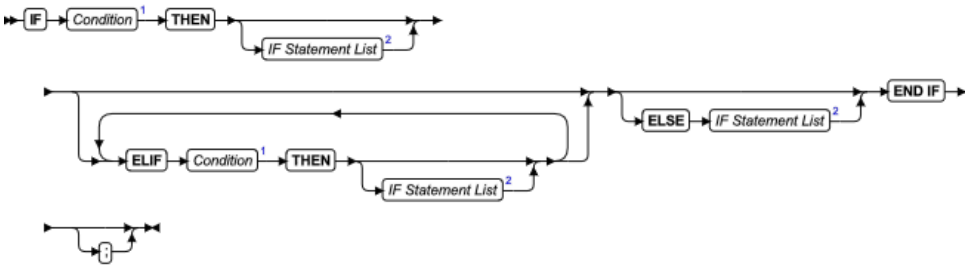
**相关语句**

<< Label >> 语句

## 3.11 IF

使用 IF 语句在 SPL 例程中创建逻辑分支。

**语法**




**用法**

数据库服务器按下列步骤处理 IF 语句：

1. 如果 IF 关键字之后的条件为真，则执行 IF 语句的第一个 THEN 关键字之后的任何语句，并且 IF 语句终止。
2. 如果初始 IF 语句条件的结果为假，当存在 ELIF 子句，则数据库服务器将计算 ELIF 关键字之后的条件。

3. 如果 ELIF 条件的结果为真，则执行 ELIF 语句的 THEN 关键字后的任何语句，并且 IF 语句终止。
4. 如果第一个 ELIF 子句中的条件的结果也为假，则数据库服务器会计算下一个 ELIF 子句的条件，如果它为真，则继续执行上一个步骤。如果为假，数据库服务器将计算连续 ELIF 子句中的条件，直到找到一个条件为真，在这种情况下，它会执行该 ELIF 子句中的 THEN 关键字后面的语句列表，然后 IF 语句终止。
5. 如果 IF 语句中没有条件为真，但存在 ELSE 子句，则执行 ELSE 关键字之后的语句，并且 IF 语句终止。
6. 如果 IF 语句中的条件都不为真，且不存在 ELSE 子句，则 IF 语句终止，而不执行任何语句列表。

## ELIF 子句

使用 ELIF 子句指定一个或多个附加条件以计算值。如果 IF 条件为假，则计算 ELIF 条件的值。如果 ELIF 条件为真，则执行 ELIF 子句中跟随 THEN 关键字之后的子句。

如果当 ELIF 条件为真时 ELIF 子句的 THEN 关键字之后没有语句，则程序将控制从 IF 语句传递给下一条语句。

## ELSE 子句

如果没有为真的上一级统计存在于 IF 子句或任何 ELIF 子句，则执行 ELSE 子句。

在以下示例中，SPL 函数使用具有 ELIF 子句和 ELSE 子句的 IF 语句。IF 语句比较两个字符串。

函数显示 1 来指示第一个字符串按字母顺序出现在第二个字符串前面，或则当前第一个字符串按字母顺序出现在第二个字符串后面时，则显示 -1 。如果字符串都相同，则返回量（0）。

```
CREATE FUNCTION str_compare (str1 CHAR(20), str2 CHAR(20))
        RETURNING INT;
        DEFINE result INT;
        IF str1 > str2 THEN LET result =1;
        ELIF str2 > str1 THEN LET result = -1;
        ELSE LET result = 0;
        END IF
        RETURN result;
END FUNCTION -- str_compare
```

## IF 语句中的条件

如同在 WHILE 语句中，如果 *condition* 中的任何表达式求值为 NULL，则条件不能为真，除非您正在使用 IS NULL 运算符对 NULL 显式地进行测试。以下规则总结了条件中的 NULL 值：

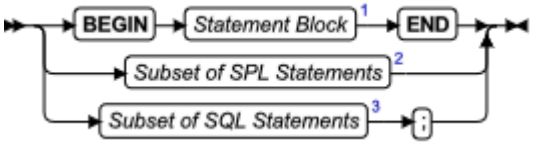
1. 如果表达式 **x** 求值为 NULL，则按照定义，**x** 不为真。而且，NOT (**x**) 也不为真。
2. IS NULL 是可使 **x** 恢复为真的唯一运算符。即 **x** IS NULL 为真，**x** IS NOT NULL 不为真。

如果条件中的表达式含有来自未初始化的 SPL 变量的 UNKNOWN 值，则语句终止并出现异常。

可以仅在触发器例程中的 IF 语句中将触发器类型 Boolean 运算符（DELETING 、INSERTING 、SELECTING 或 UPDATING）作为条件。

**IF 语句列表**

IF 语句列表




## IF 语句列表中允许的 SPL 语句的子集

您可以在 IF 语句列表中使用以下任何 SPL 语句：

- <<***Label*** >>
- CALL
- CASE
- CONTINUE
- EXIT
- FOR
- FOREACH
- GOTO
- IF
- LET
- LOOP
- RAISE EXCEPTION
- RETURN
- SYSTEM
- TRACE
- WHILE

IF 语句列表 的“SPL 语句的子集”语法图请参阅以上列出的 SPL 语句。

## IF 语句中无效的 SQL 语句

在 IF 语句列表的语法图中的“SQL 语句的子集”元素引用了所有 SQL 语句，除以下 SQL 语句，这些语句在 IF 语句列表中无效：

- ALLOCATE DESCRIPTOR
- CLOSE DATABASE
- CONNECT
- CREATE DATABASE

- CREATE PROCEDURE
- DATABASE
- DEALLOCATE DESCRIPTOR
- DESCRIBE
- DISCONNECT
- EXECUTE
- FLUSH
- GET DESCRIPTOR
- GET DIAGNOSTICS
- INFO
- LOAD
- OUTPUT
- PUT
- SET AUTOFREE
- SET CONNECTION
- SET DESCRIPTOR
- UNLOAD
- WHENEVER

仅当使用 INTO TEMP 子句将 SELECT 语句的结果集存储在临时表中时，您可以使用 SELECT 语句。

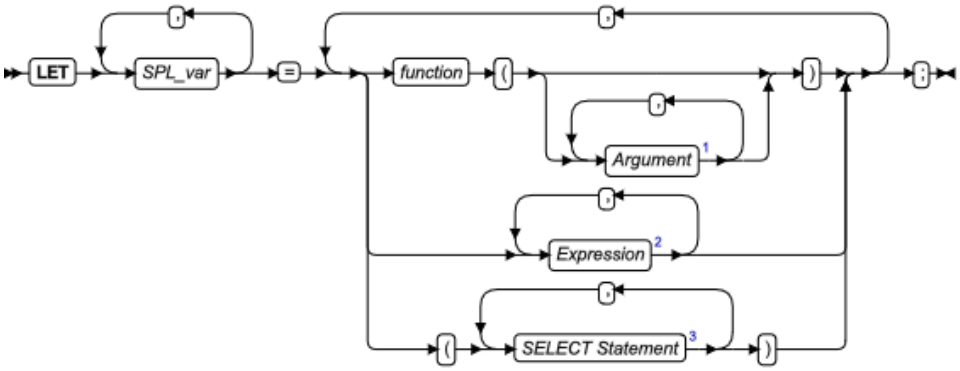
**相关语句**

WHILE

## 3.12 LET

使用 LET 语句对变量赋值或调用用户定义的 SPL 例程，并指定返回值或对 SPL 变量赋值。

**语法**



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *function* | 要调用的 SPL 函数 | 必须存在于数据库中 | 标识符 |
| *SPL_var* | 接收函数、表达式或查询返回的值的 SPL 变量 | 必须被定义并在语句块的作用域中 | 标识符; |

**用法**

LET 语句可将由表达式、函数或查询返回的值指定给 SPL 变量。在运行时，首先计算要被指定的值。如果可能，得到的值被强制转型为 ***SPL_var*** 数据类型并且进行赋值。如果不可能转换，则出现错误，并且变量的值保持为未定义状态。（将单一值指定给单一 SPL 变量的 LET 操作称为***简单赋值***。）

***复合赋值***给多个 SPL 变量指定多个表达式。表达式列表中的表达式的数据类型不需要与变量列表中的对应变量的数据类型相匹配。因为数据库服务器可自动转换数据类型。（对于强制转型的详细讨论，请参阅 *《GBase 8s SQL 指南：参考》*。）

在多赋值操作中，等号（=）左边的变量数必须与等号（=）右边列出的由函数、表达式和查询返回的值的数量相匹配。以下示例显示了对 SPL 变量赋值的几个 LET 语句：

```
LET a     = c + d ;
LET a,b = c,d ;
LET expire_dt = end_dt + 7 UNITS DAY;
LET name = 'Brunhilda';
LET sname = DBSERVERNAME;
LET this_day = TODAY;
```

不能使用等号（=）右边的多个值对其它值进行操作。例如，以下语句是无效的：

```
LET a,b = (c,d) + (10,15); -- INVALID EXPRESSION
```

## 在 LET 语句中使用 SELECT 语句

本部分中的示例在 LET 语句中使用 SELECT 语句。您可以使用 SELECT 语句对等号（=）运算符左边的一个或多个变量赋值，如以下示例中所示：

```
LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
LET a,b,c = (SELECT c1,c2 FROM t WHERE id = 1), 15;
```

您不能使用 SELECT 语句使多个值对其它值进行运算。以下示例是无效的：

```
LET a,b = (SELECT c1,c2 FROM t) + (10,15); -- INVALID CODE
```

因为 LET 语句等价于 SELECT ... INTO 语句，所以以下示例中的两个语句有相同结果：a=c 和 b=d：

```
CREATE PROCEDURE proof()
DEFINE a, b, c, d INT;
LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
SELECT c1, c2 INTO c, d FROM t WHERE id = 1
END PROCEDURE
```

如果 SELECT 语句返回多行，您必须将 SELECT 语句包括在 FOREACH 循环中。

有关 SELECT 语法和用法的描述，请参阅 SELECT 语句。

## 在 LET 语句中调用函数

您可以在 LET 语句中调用用户定义的函数，并将返回值指定给接收函数返回的值的 SPL 变量。

SPL 函数可将多个值（即，来自同一行中多个列的值）返回到变量名称列表中。换句话说，该函数可在其 RETURN 语句中有多个值，并且 LET 语句可有多个变量以接收返回值。

当调用函数时，必须对函数指定所有必要的参数，除非函数的参数有缺省值。如果您使用语法（如 **name = 'smith'**）给被调用的函数中的多个参数中的一个指定了名称，则必须命名所有参数。

选择和返回多个行的 SPL 函数必须包括在 FOREACH 循环中。

以下两个示例显示了有效的 LET 语句：

```
LET a, b, c = func1(name = 'grok', age = 17);
LET a, b, c = 7, func2('orange', 'green');
```

以下 LET 语句无效，因为它试图添加两个函数的输出，然后将总和指定给两个变量 a 和 b。

```
LET a, b = func1() + func2(); -- 无效代码
```

您可以轻松地将此 LET 语句分为两个有效的 LET 语句：

```
LET a = (func1() + func2());
LET b = a;                    -- 有效代码
```

在 LET 语句中调用的函数可以有 COLLECTION 、SET 、MULTISET 或 LIST 参数。您可以将函数返回的值指定给变量，例如：

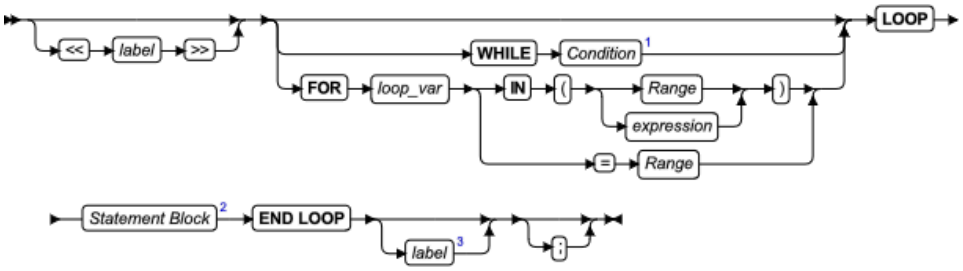
```
LET d = function1(collection1);
LET a = function2(set1);
```

在第一个语句中，SPL 函数 **function1** 接受 **collection1** （即，任何集合数据类型）作为参数并将其值返回给变量 **d**。第二个语句中，SPL 函数 **function2** 接受 s **set1** 作为参数并将值返回给变量 **a**。

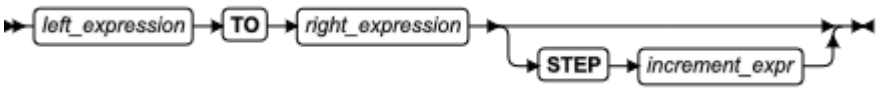
# 3.13 LOOP

使用 LOOP 语句定义具有不确定迭代次数的循环。

**语法**



Range




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 与 *loop_var* 相比的值 | 必须与 *loop_var* 数据类型相匹配 | 表达式 |
| *increment_expr* | *loop_var* 增加的正或负值 | 必须返回整数。不能返回 0。 | 表达式 |
| *label* | 此循环的循环标签的名称 | 在此 SPL 例程的标签中必须是唯一的 | 标识符 |
| *left_expression* | 范围开始的表达式 | 值必须符合 *loop_var* 的 SMALLINT 或 INT 数据类型 | 表达式 |
| *loop_var* | 确定循环执行的次数的变量 | 必须已定义并且在此语句块中的作用域中 | 标识符 |
| *right_expression* | 范围的结束表达式 | 同 *left_expression* | 表达式 |

**用法**

LOOP 语句是类似于 FOR 和 WHILE 语句的迭代语句。就像 FOR 和 WHILE，LOOP 语句具有可选的循环标签。它可以包含 CONTINUE 语句以指定另一个迭代，并且 EXIT 语句会终止循环的执行。

除了在其功能中类似于 FOR 和 WHILE 之外，LOOP 语句可以使用语句块之前的 FOR 或 WHILE 语法。下面的部分描述了 LOOP 语句的几种形式，包括：

- 无限期迭代语句循环的简单 LOOP 语句
- FOR LOOP 语句，使用 FOR 语句语法指定有限数量的迭代
- WHILE LOOP 语句，在指定条件为真时迭代
- 每个 LOOP 语句的标签版本，可以终止深层嵌套循环。

## 简单 LOOP 语句

以下程序片段说明了 LOOP 语句的样本：

```
LOOP
LET i = i + 1;
IF i = 5 THEN EXIT;
ELSE
CONTINUE;
```

```
END IF
END LOOP;
```

在此示例中，IF 语句限制了迭代的次数。这里 CONTINUE 和 EXIT 语句省略了可选的 LOOP 关键字，但是在语句循环的结束时需要 END LOOP 语句。类似的 FOR 或 WHILE 关键字在 CONTINUE 和 EXIT 语句中分别需要 FOR 或 WHILE 关键字。

下一个示例使用条件 EXIT 语句终止循环：

```
LOOP
LET i = i + 1;
EXIT WHEN i = 4;
END LOOP;
```

在 EXIT 语句之后不需要标识循环语句类型的关键字，如 FOR 、WHILE 或 FOREACH 语句中的 EXIT 语句的情况。当条件 i = 4 为真时，程序控制从 LOOP 语句传递到 END LOOP 关键字之后的任何语句。

## FOR LOOP 语句

FOR LOOP 语句使用 FOR 语句语法指定变量和变量可以采用的值的范围。循环迭代，直到达到对这些值的指定限制，或者直到控制被传递到循环之外，如以下示例中的无条件 EXIT 语句所示：

```
FOR i IN (1 TO 5) LOOP
IF i = 5 THEN EXIT;
ELSE
CONTINUE;
END LOOP;
```

在此 FOR LOOP 语句中，FOR 关键字可以后跟 EXIT 或 CONTINUE 关键字 ，但是不需要 FOR 关键字，因为它在一个普通的 FOR 语句中。

以下示例用同等功能的条件 EXIT 语句替换 IF 语句：

```
FOR i IN (1 TO 5) LOOP
EXIT WHEN i = 5;
END LOOP;
```

## WHILE LOOP 语句

要创建 WHILE LOOP 语句，循环，可以立即使用 LOOP 语句跟随 WHILE *condition* 规范。在条件变为假之后，或者当一些其他语句从循环中传递程序控制时，结果循环终止。在以下 WHILE LOOP 语句中，条件指定循环在循环变量 i 增加到值 6 之后终止：

```
WHILE (i < 6) LOOP
LET i = i + 1;
IF i = 5 THEN EXIT;
ELSE
CONTINUE;
END IF
END LOOP;
```

与在 FOR LOOP 语句中一样，EXIT 和 CONTINUE 关键字不需要指定循环语句类型，但是如果 XIT WHILE 和 CONTINUE WHILE 替换了 EXIT 和 CONTINUE 关键字，则该示例不会受到影响。但是，END LOOP 关键字是必需的，因为 GBase 8s 将 WHILE LOOP （和 FOR LOOP）语句视为 LOOP 语句，尽管它们的初始 FOR 和 WHILE 规范。

## Labeled LOOP 语句

所有形式的 LOOP 语句，包括 FOR LOOP 、WHILE LOOP 和简单 LOOP 语句可以具有语句标签。您可以按以下步骤创建带标签的 LOOP 语句：

1. 写一个有效的 LOOP 、FOR LOOP 或 WHILE LOOP 语句。
2. 通过紧接在 LOOP 、FOR LOOP 或 WHILE LOOP 语句的第一行之前的尖括号（<<loop_label>>）之间包含一个 SQL 标识符（它不是同一 SPL 例程中的标签的名称）来创建语句标签。
3. 输入相同的 SQL 标识符，但是不带尖括号分隔符，紧接在终止语句的 END LOOP 关键字之后，现在这是一个带标签的循环语句。

标记的 LOOP 语句的一个优点是它们可以在 EXIT 语句中引用。当执行 EXIT *label* 语句时，程序控制从 EXIT 语句传递到跟在指定循环标记之后的语句。

在以下示例中，标记的 WHILE LOOP 循环（其循环标记标识符是 endo）是标记的 LOOP 语句的语句块的一部分，其循环标签标识符是 voort。如果条件 EXIT 语句 EXIT endo WHEN x = 7: 检测到的条件为真，则程序控制传递到 END LOOP endo 内部生成之后的 LET x = x + 1 语句。如果条件语句 EXIT voort WHEN x > 9 检测到其条件为真，则程序控制传递到 END LOOP voort 语句之后的 LET x = x + 1 语句，并且 x 的值不能被 LET 语句递增。

```
<<voort>>
  LOOP
  LET x = x+1;
<<endo>>
WHILE ( i < 10 ) LOOP
LET x = x+1;
EXIT endo WHEN x = 7;
EXIT voort WHEN x > 9;
END LOOP endo;
LET x = x+1;
END LOOP voort;
```

使用 FOR 语句语法指定变量和变量可以采用的值的范围。循环迭代，直到达到对这些值的指定限制，或者直到控制被传递到循环之外，如以下示例中的无条件 EXIT 语句所示：

```
FOR i IN (1 TO 5) LOOP
IF i = 5 THEN EXIT;
ELSE
CONTINUE;
END LOOP;
```

在 FOR LOOP 语句中，FOR 关键字可以后跟 EXIT 或 CONTINUE 关键字，但是不需要 FOR 关键字，因为它在一个普通的 FOR 语句中。

以下示例用同等功能的条件 EXIT 语句替换 IF 语句：

```
FOR i IN (1 TO 5) LOOP
EXIT WHEN i = 5;
END LOOP;
```

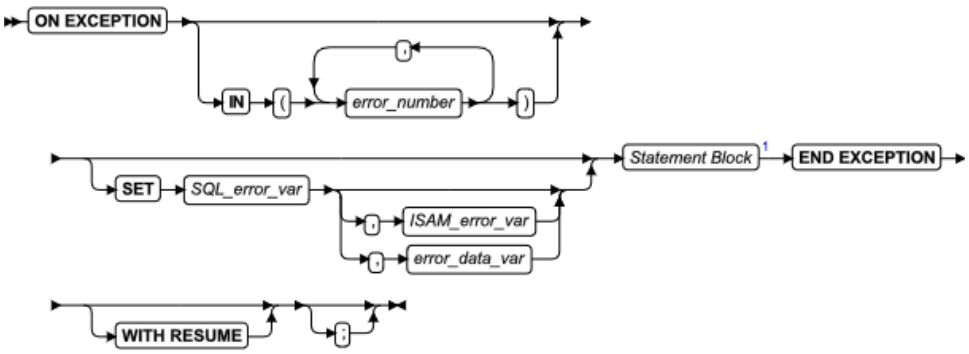
相关的语句

<< Label >> 语句 、FOR 、WHILE

## 3.14 ON EXCEPTION

使用 ON EXCEPTION 语句指定操作，这些操作是在语句块的执行期间对任何错误或对一个或多个指定错误的列表采取的。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *error_data_var* | SPL 变量，用于接收由 SQL 错误或由用户定义的异常返回的字符串 | 必须是字符类型以接收错误消息。必须在当前的语句块中有效。 | 标识符 |
| *error_number* | SQL 错误号或由将被捕获的 RAISE EXCEPTION 语句定义的号码 | 必须是整数类型。在当前的语句块中必须有效。 | 精确数值 |
| *ISAM_error_var* | SPL 变量，接收出现的异常的 ISAM 错误号 | 同 *error_number* | 标识符 |
| *SQL_error_var* | SPL 变量，接收出现的异常的 SQL 错误号 | 同 *ISAM_error_var* | 标识符 |

**用法**

ON EXCEPTION 语句与 RAISE EXCEPTION 语句一起提供 SPL 的错误俘获和错误恢复机制。ON EXCEPTION 可以在 SPL 例程执行时指定您想要捕获的错误，并指定当前语句块中发生错误时要采

取的操作。ON EXCEPTION 语句可以指定 IN 子句中的错误号列表，或可以不包括 IN 子句。如果省略 IN 子句，则俘获所有错误。

语句块可以包括多个 ON EXCEPTION 语句。被俘获的异常可以是系统定义的也可以是用户定义的。

ON EXCEPTION 语句的作用域是包含它的语句块，以及嵌套在语句块中的其它语句块，除非其中之一嵌套的语句块提供了覆盖外部语句的 ON EXCEPTION 语句。

当设陷阱捕捉到异常时，会清除错误状态。

如果您指定某个变量来接收 ISAM 错误，但不存在跟随的 ISAM 错误，则将零（0）指定给该变量。如果指定某个变量来接收错误文本，但不存在错误文本，则该变量存储空字符串。

**触发器操作中不支持 ON EXCEPTION**

当从以下调用上下文中的 SPL 例程发出时，ON EXCEPTION 语句无效：

- 在触发例程中
- 在表的触发器的 Action 子句或 Correlated Action 子句中
- 在视图上的 INSTEAD OF 触发器的 Action 子句中。

当 UDR 在这些上下文中包含 ON EXCEPTION ，数据库服务器忽略 ON EXCEPTION 语句。

## ON EXCEPTION 语句的放置

ON EXCEPTION 语句是一个声明性语句，不是可执行语句。出于此原因，ON EXCEPTION 必须在任何可执行语句之前并且在 SPL 中必须跟随任何 DEFINE 语句。

因为 SPL 例程的主体是语句块，所以 ON EXCEPTION 语句进程在例程的开头出现，并应用于例程的整个代码中。

语句示例显示了 ON EXCEPTION 语句的正确放置，以致于 FOREACH 语句在发生错误可以继续处理行。过程 X( ) 从表 A 读取客户编号，并将其插入到表 B。因为 INSERT 语句在 ON EXCEPTION 语句的作用域内，所以 INSERT 操作期间的错误会导致执行控制转移到 FOREACH 游标的下一行，而不用终止此 FOREACH 循环。

```
CREATE PROCEDURE X()
DEFINE v_cust_num CHAR(20);
FOREACH cs_insert FOR SELECT cust_num INTO v_cust_num FROM A
BEGIN
ON EXCEPTION
END EXCEPTION WITH RESUME;
INSERT INTO B(cust_num) VALUES(v_cust_num);
END
END FOREACH
END PROCEDURE
```

下一示例中，函数 add_salesperson( ) 向表中插入一组值。如果表不存在，则先创建该表再插入值。该函数还返回表中的总行数：

```
CREATE FUNCTION add_salesperson(last CHAR(15), first CHAR(15))
    RETURNING INT;
DEFINE x INT;
   N EXCEPTION IN (-206) -- If no table was found, create one
      CREATE TABLE emp_list
            (lname CHAR(15),fname CHAR(15), tele CHAR(12));
   INSERT INTO emp_list VALUES -- and insert values
                  (last, first, '800-555-1234');
END EXCEPTION WITH RESUME;
INSERT INTO emp_list VALUES (last, first, '800-555-1234');
SELECT count(*) INTO x FROM emp_list;
RETURN x;
END FUNCTION;
```

当出现错误时，数据库服务器搜索捕获错误代码的最后一个 ON EXCEPTION 语句。如果数据库服务器找不到相关 ON EXCEPTION 语句，则错误代码传回调用上下文（SPL 例程、应用程序或交互用户），执行终止。

在前面的示例中，在指定错误 -206 的 IN 子句中需要减号（-），大多数错误代码都是负整数。

下一示例使用了两个具有相同错误号的 ON EXCEPTION 语句，因而可以在两个嵌套级别中捕获错误代码 691 。除了标有 { 6 } 的 DELETE 语句，所有其它 DELETE 语句都在第一个 ON EXCEPTION 语句的作用域中。标有 { 1 } 和 { 2 } 的 DELETE 语句都在内部 ON EXCEPTION 语句的作用域中：

```
CREATE PROCEDURE delete_cust (cnum INT)
         ON EXCEPTION    IN (-691)       -- children exist
    BEGIN -- Begin-end so no other DELETEs get caught in here.
        ON EXCEPTION IN (-691)
        DELETE FROM another_child WHERE num = cnum;     { 1 }
        DELETE FROM orders WHERE customer_num = cnum; { 2 }
        END EXCEPTION -- for error -691
       DELETE FROM orders WHERE customer_num = cnum;       { 3 }
        END
DELETE FROM cust_calls WHERE customer_num = cnum;     { 4 }
DELETE FROM customer WHERE customer_num = cnum;        { 5 }
    END EXCEPTION
DELETE FROM customer WHERE customer_num = cnum;             { 6 }
END PROCEDURE
```

## 使用 IN 子句捕获特定的异常

如果 SQL 错误代码或 ISAM 错误代码与错误号列表中的异常代码相匹配，则会俘获错误。在错误列表中的搜索重左边开始，并在搜索到第一个匹配时停止。可以使用没有 IN 子句的 ON EXCEPTION 语句和具有 IN 子句的一个或多个 ON EXCEPTION 语句的组合。当出现错误时，数据库服务器搜索捕获特殊错误代码的 ON EXCEPTION 语句的最后一个声明。

```
CREATE PROCEDURE ex_test()
         DEFINE error_num INT;
```

```
        ...
        ON EXCEPTION SET error_num
        -- action C
        END EXCEPTION
        ON EXCEPTION IN (-300)
        -- action B
        END EXCEPTION
        ON EXCEPTION IN (-210, -211, -212) SET error_num
        -- action A
END EXCEPTION
```

在上一示例中的语句序列总结为：

1. 测试是否有错误。
2. 如果出现错误 -210 、-211 或 -212 ，则采取操作 A。
3. 如果出现错误 -300 ，则采取操作 B。
4. 如果出现任何其它错误，则采取操作 C。

### 接收 SET 子句中的错误信息

如果使用 SET 子句，但出现异常时，SQL 错误代码和（可选地）ISAM 错误代码被插入到在 SET 子句中指定的变量中，如果提供 ***error_data_var***，则数据库服务器返回的任何错误文本被放入到 ***error_data_var***。错误文本包括类似损坏的表或列名称的信息。

### 强制例程继续

ON EXCEPTION 语句的放置中的第一个示例包括 WITH RESUME 关键字，用于指定如果 ON EXCEPTION 语句捕获错误，则 FOREACH 循环的执行在 **cs_insert** 游标的发出错误的行的下一行继续。如果活动集的最后一行发出了错误，则该过程退出。在 **X** 过程执行完毕后，表 **B** 会包含表 **A** 中每个客户编号的副本，在 INSERT 期间不会发生错误。

ON EXCEPTION 语句的放置中的第二个示例使用 WITH RESUME 关键字表示在 ON EXCEPTION 语句中的语句块执行后，在 SELECT COUNT(*) FROM emp_list 语句继续执行，该语句是跟随在出现错误的行之后的行。对于此函数，其结果是即使出现错误也计数销售人员姓名。

## 在出现异常后继续执行

如果省略 WITH RESUME 关键字，则在出现异常后执行的下一个语句取决于 ON EXCEPTION 语句的位置，如以下应用场合所描述的：

- 如果 ON EXCEPTION 语句在具有 BEGIN 和 END 关键字的语句块中，则在对该 BEGIN ... END 块后的第一个语句（如果有）恢复执行，即，在 ON EXCEPTION 语句的作用域后恢复。
- 如果 ON EXCEPTION 语句在循环（FOR 、WHILE 、FOREACH）内部，则跳过循环的其余部分，并在循环的下一迭代时恢复执行。
- 如果没有语句块，而仅有 SPL 例程包含 ON EXCEPTION 语句，那么该例程执行不带参数的 RETURN 语句，返回成功状态而不返回值。

如果在语句块执行期间出现错误，要防止无限循环，则搜索另一个 ON EXCEPTION 语句以俘获错误不包括搜索当前的 ON EXCEPTION 语句。

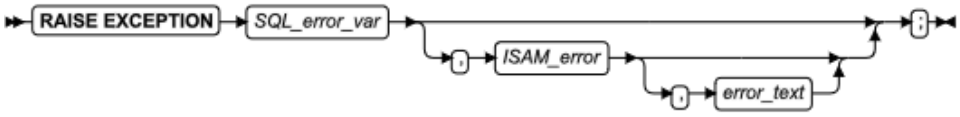
相关语句

RAISE EXCEPTION

## 3.15 RAISE EXCEPTION

使用 RAISE EXCEPTION 语句模拟错误的生成。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *error_text* | SPL 变量或表达式，包含错误文本 -746 的错误消息 | 必须是字符数据类型并且在语句块中有效 | 标识符；表达式 |
| *ISAM_error* | SPL 变量或代表 ISAM 错误号的表达式 SPL。缺省值为 0。 | 必须返回 SMALLINT 范围中的值。可以在错误号前指定一个一元减号 | 表达式 |
| *SQL_error* | SPL 变量或表达式，代表 SQL 错误号 | 同 *ISAM_error* | 表达式 |

**用法**

使用 RAISE EXCEPTION 语句模拟错误或通过定制消息生成错误。ON EXCEPTION 语句可捕捉生成的异常。

如果省略 ***ISAM_error***，则当出现异常时，数据库服务器将 ISAM 错误代码设置为零（0）。如果希望指定 ***error_text*** ，但不指定 ***ISAM_error*** 值，则指定零（0）为 ***ISAM_error*** 的值。

RAISE EXCEPTION 语句可以产生系统生成的异常或用户生成的异常。例如，以下语句产生错误号 -208：

```
RAISE EXCEPTION -208, 0;
```

此处在 EXCEPTION 关键字后需要减号（- ）用于错误 -208；大多数错误代码是负整数。

### 特殊的错误号 -746

特殊的错误号 -746 允许您生成定制的消息。例如，以下语句产生错误号 -746 并返回引号中的文本：

```
RAISE EXCEPTION -746, 0, 'You broke the rules';
```

在下列示例中，alpha 的负值产生异常 -746 并提供描述该问题的特定消息。代码中应包含 ON EXCEPTION 语句，它俘获异常 -746。

```
FOREACH SELECT c1 INTO alpha FROM sometable
IF alpha < 0 THEN
RAISE EXCEPTION -746, 0, 'a < 0 found' -- emergency exit
END IF
END FOREACH
```

当执行 SPL 例程并满足 IF 条件时，数据库服务器返回以下错误：

-746: a < 0 found.

有关作用域和异常兼容性的更多信息，请参阅 ON EXCEPTION。

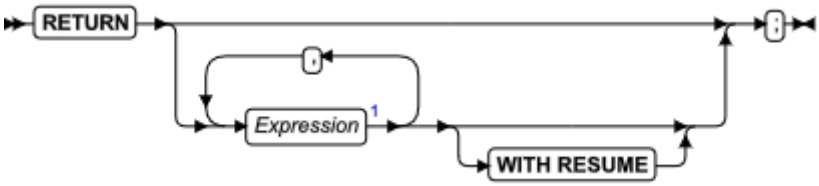
**相关语句**

ON EXCEPTION

## 3.16 RETURN

使用 RETURN 语句来指定 SPL 函数将哪些值（如果有的话）返回给调用上下文。

**语法**




**用法**

在 GBase 8s 中，对于向后兼容性，您可以在 CREATE PROCEDURE 语句中使用 RETURN 语句创建 SPL 函数。然而，通过在 CREATE FUNCTION 语句中只使用 RETURN ，您可以保留使用 CREATE FUNCTION 来定义返回值的例程的约定，并为其它例程保留 CREATE PROCEDURE。

SPL 函数中的所有 RETURN 语句必须和定义该函数的 CREATE FUNCTION （或 CREATE PROCEDURE）语句的 RETURNING 子句相一致。表达式的任何 RETURN 列表在基数上必须与函数定义的 RETURNING 子句的数据类型的有序列表相匹配，并且在数据类型方面必须与其兼容。

或者，即使 RETURNING 子句列出了一个或多个数据类型，RETURN 语句也不能指定表达式。在这种情况下，没有指定表达式的 RETURN 语句等于将预期的 NULL 值的数目返回给调用上下文。只有将 SPL 函数声明为不返回任何值时，不带有任何表达式的 RETURN 语句才可以存在。否则，它返回 NULL 值。

以下 SPL 函数有两个有效的 RETURN 语句：

```
CREATE FUNCTION two_returns (stockno INT)  RETURNING CHAR (15);
DEFINE des CHAR(15);
ON EXCEPTION (-272)        -- if user does not have select privilege
```

```
RETURN;                    -- return no values.
END EXCEPTION;
SELECT DISTINCT descript INTO des FROM stock
WHERE stock_num = stockno;
RETURN des;
END FUNCTION;
```

在前面的示例中调用函数的程序应测试是否没有返回值并进行相应的操作。

### WITH RESUME 关键字

如果使用 WITH RESUME 关键字，则在执行 RETURN 语句后，SPL 函数的下一个调用（对下一个 FETCH 或 FOREACH 语句）从跟随 RETURN 语句的语句开始。任何执行 RETURN WITH RESUME 语句的函数必须在 FOREACH 循环或 SELECT 的 FROM 子句中被调用。如果 SPL 例程执行 RETURN WITH RESUME 语句，则 GBase 8s ESQL/C 应用程序中的 FETCH 语句可以调用 SPL 例程。

以下示例显示另一个 UDR 可调用的游标函数。在 RETURN WITH RESUME 语句将每个值返回到调用的 UDR 或程序后，在下一次调用 series 时执行 series 的下一行。如果变量 backwards 等于零(0)，则没有值返回给调用的 UDR 或程序，并停止 series 的执行：

```
CREATE FUNCTION series (limit INT, backwards INT) RETURNING INT;
        DEFINE i INT;
        FOR i IN (1 TO limit)
        RETURN i WITH RESUME;
        END FOR;
        IF backwards = 0 THEN
        RETURN;
        END IF;
        FOR i IN (limit TO 1 STEP -1)
        RETURN i WITH RESUME;
        END FOR;
END FUNCTION; -- series
```

## 从另一个数据库返回值

如果 SPL 函数使用 Return 子句从本地 GBase 8s 实例的另一个数据库返回值，则支持将以下数据类型作为返回的数据类型：

- 不透明的内置数据类型
- 大多数***内置透明数据类型***，在跨数据库事务中的数据类型 中列出
- 以上两行中引用的内置类型的 DISTINCT
- 此列表中任何 DISTINCT 数据类型的 DISTINCT
- 显式转换为此列表中某个内置数据类型的任何不透明用户定义类型（UDT）。

UDF 以及类型层次结构、转型、DISTINCT 类型和 UDT 的定义在每个参与数据库中必须完全相同。相同的数据类型限制适用于外部函数从本地 GBase 8s 实例的另一个数据库返回的值。有关跨同一数

据库服务器的两个或多个数据库的分布式操作中支持的数据类型的详细信息，请参阅跨数据库事务中的数据类型。

UDR 从另一个数据库服务器的表中只能返回以下数据类型：

- 任何不透明的内置数据类型
- BOOLEAN
- LVARCHAR
- 不透明数据类型 DISTINCT
- BOOLEAN DISTINCT
- LVARCHAR DISTINCT
- 此列表中出现的任何 DISTINCT 类型的 DISTINCT

只有在 DISTINCT 类型显式转换为内置类型时，UDR 才能从其它 GBase 8s 实例的数据库返回这些 DISTINCT 类型。DISTINCT 数据类型的定义，它们的类型层次结构，以及它们的强制转型在参与分布式操作的数据库中必须完全相同。对于使用上一列表中的数据类型作为参数或返回数据类型的跨服务器 UDR 中的查询或其它 DML 操作，必须在每个参与数据库中定义 UDR ，参与的 GBase 8s 实例必须支持数据类型为跨服务器操作中的返回值。

有关 GBase 8s 可在分布式操作中访问的数据类型的其它信息，请参阅分布式查询中的数据类型。

### *外部函数和迭代器函数*

在 SPL 程序中，如果外部函数不是迭代器函数，则可以使用 C 或 Java™ 语言外部函数作为 RETURN 语句的表达式。***迭代器函数***是返回一行或多行数据的外部函数（因而需要游标来执行）。

SPL 迭代函数必须包含 RETURN WITH RESUME 语句。有关在查询的 FROM 子句中通过虚拟表接口使用迭代器函数的信息，请参阅迭代器函数。

## 3.17 SYSTEM

使用 SYSTEM 语句从 SPL 例程中发出操作系统命令。

**语法**


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 对用户可执行操作系统命令求值 | 不能指定该命令在后台运行 | 操作系统从属 |
| *SPL_var* | 包含命令的 SPL 变量 | 必须是字符数据类型 | 标识符; |

**用法**

如果指定的 ***expression*** 不是字符表达式，则它被转换为字符表达式并传递到操作系统以进行执行。

SYSTEM 指定的命令不能在后台运行。数据库服务器在继续 SPL 例程中的下一语句之前等待操作系统完成该命令的执行。SPL 例程不能使用任何从该命令返回的值。

如果操作系统命令失败（即，返回命令的非零状态），则出现异常。该异常包含返回的操作系统状态作为 ISAM 错误代码和相应的 SQL 错误代码。

回滚不终止系统调用，这样暂挂的事务可以无限等待调用以返回。有关在长事务回滚期间从死锁恢复的指示信息，请参阅 *GBase 8s 管理员指南*。

GBase 8s 的动态日志功能自动添加日志文件，直到长事务成功完成或回滚。

在包含 SYSTEM 语句的 DBA 和所有者特权的 SPL 例程中，该命令经执行例程的用户的许可方可运行。

## 在 UNIX 上执行 SYSTEM 语句

在 UNIX™ 平台的 SPL 过程中，评估为有效 UNIX 操作系统命令的规范必须紧跟在 SYSTEM 关键字之后。

这两个程序片段使用 SPL 的 SYSTEM 语句向系统管理员发送消息：

- 在第一个示例中，**sensitive_update** 例程定义名为 **mailcall** 的 SPL 变量存储指定 **mail** 实用程序名称，邮件的收件人的用户 ID 和邮件文本的字符串。
- 在第二个示例中，**sensitive_update2** 例程类似地使用 SYSTEM 语句调用 **mail** 实用程序。表达式通过连接三个带引号的字符串和 SPL 变量 **user1** 和 **user2** 来构造有效的命令行，以向系统管理员发送一个名为 **violations_file** 的文件。

### 使用 SYSTEM 语句发送邮件

在 SPL 例程的以下示例中的 SYSTEM 语句可使 UNIX 操作系统将邮件消息发送给系统管理员，他的用户 ID 是 headhoncho：

```
CREATE PROCEDURE sensitive_update()
    ...
    LET mailcall = 'mail headhoncho < alert';
    -- code to execute if user tries to execute a specified
    -- command, then sends email to system administrator
    SYSTEM mailcall;
    ...
END PROCEDURE; -- sensitive_update
```

可以使用双竖线符号（||）将表达式与 SYSTEM 语句连接起来，如以下示例中所示：

```
CREATE PROCEDURE sensitive_update2()
        DEFINE user1 char(15);
        DEFINE user2 char(15);
        LET user1 = 'joe';
        LET user2 = 'mary';
        ...
```

```
    -- code to execute if user tries to execute a specified
    -- command, then sends email to system administrator
    SYSTEM 'mail -s violation' || user1 || ' ' || user2|| '< violation_file';
    ...
END PROCEDURE; --sensitive_update2
```

在以上两个示例中，空格分隔命令行的元素，因此 SYSTEM 关键字后面的表达式计算为符合操作系统 mail 实用程序的语法要求的字符串。

## 在 Windows 上执行 SYSTEM 语句

在 Windows™ 系统中，只有当正在执行 SPL 例程的当前用户已经用密码登录后，才能在 SPL 例程中执行任何 SYSTEM 语句。

数据库服务器必须拥有用户的密码和登录名以代表该用户的执行命令。

以下 SPL 例程示例中的第一个 SYSTEM 语句可使 Windows 将错误消息发送给临时文件并将消息放入按字母排序的系统日志中。第二个 SYSTEM 语句导致操作系统删除临时文件：

```
CREATE PROCEDURE test_proc()
        ...
SYSTEM 'type errormess101 > %tmp%tmpfile.txt |
        sort >> %SystemRoot%systemlog.txt';
SYSTEM 'del %tmp%tmpfile.txt';
        ...
END PROCEDURE; --test_proc
```

在此示例中跟随 SYSTEM 语句的表达式包含由 Windows 定义的变量 **%tmp%** 和 **%SystemRoot%**。

## 在 SYSTEM 命令中设置环境变量

当执行 SYSTEM 指定的操作系统命令时，不存在任何用户应用程序设置的环境变量被传递到操作系统的保证。如果在 SYSTEM 命令中设置环境变量，则该设置仅在该 SYSTEM 命令期间有效。

要避免此潜在问题，建议使用以下方法确保用户应用程序要求的任何环境变量被转发到操作系统。

更改操作系统命令的环境设置：

1. 创建将会设置希望环境的 shell 脚本(在 UNIX™ 系统上)或 batch 文件(在 Windows™ 平台上），然后执行操作系统命令。
2. 使用 SYSTEM 命令执行 shell 脚本或 batch 文件。

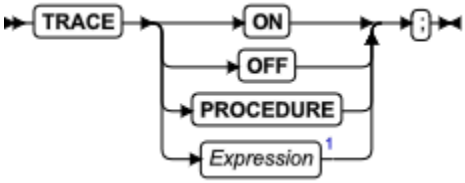
此解决方案还有其它优势：如果您以后需要更改环境，可以修改 shell 脚本或 batch 文件，而不需重新编译 SPL 例程。

有关设置环境变量的操作系统命令的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

# 3.18 TRACE

使用 TRACE 语句来控制调试输出的生成。

**语法**




**用法**

TRACE 语句生成输出，该输出会发送给 SET DEBUG FILE TO 语句指定的文件。跟踪以下程序对象的当前值写入到调试文件：

- SPL 变量
- 例程参数
- 返回值
- SQL 错误代码
- ISAM 错误代码

每个执行的 TRACE 语句的输出以独立行显示。

如果在指定 DEBUG 文件以包含输出前使用 TRACE 语句，则会生成错误。

SPL 例程调用的任何例程继承跟踪状态。即，调用的例程（在相同的数据库服务器上）假设相同的跟踪状态（ON 、OFF 或 PROCEDURE）作为调用例程。调用的例程可设置其子句的跟踪状态，但不将该状态传递回调用的例程。

在远程数据库服务器上执行的例程不继承跟踪状态。

## TRACE ON

如果指定关键字 ON，则跟踪所有语句。在使用变量（在表达式中或其它）的值之前打印它们。将跟踪设置为 ON 表示例程主体中的例程调用和语句都被跟踪。

## TRACE OFF

如果您指定关键字 OFF，则所有的跟踪都被关闭。

## TRACE PROCEDURE

如果指定关键字 PROCEDURE，则仅跟踪例程调用和返回值，而非例程主体。

TRACE 语句没有 ROUTINE 或 FUNCTION 关键字。因为即使想要跟踪的 SPL 例程是一个函数，也使用 TRACE PROCEDURE 关键字。

## 显示表达式

可以使用具有引号引起的字符串或表达式的 TRACE 语句显示输出文件中的值或注释。如果表达式不是字符表达式，则在写入输出文件之前计算表达式的值。

即使在例程中较早地使用了 TRACE OFF 语句，也可以使用具有表达式的 TRACE 语句。但必须首先使用 SET DEBUG 语句建立一个跟踪输出文件。

下一个示例在使用 TRACE OFF 语句之后使用具有表达式的 TRACE 语句。该示例使用了 UNIX™ 文件命名约定：

```
CREATE PROCEDURE tracing ()
    DEFINE i INT;
  BEGIN
      ON EXCEPTION IN (1)
      END EXCEPTION; -- do nothing
    SET DEBUG FILE TO '/tmp/foo.trace';
          TRACE OFF;
          TRACE 'Forloop starts';
          FOR i IN (1 TO 1000)
    BEGIN
      TRACE 'FOREACH starts';
      FOREACH SELECT...INTO a FROM t
          IF <some condition> THEN
          RAISE EXCEPTION 1           -- emergency 退出
          END IF
      END FOREACH                     -- 返回值
    END
 END FOR
 END;
END PROCEDURE
```

## 显示不同格式的 TRACE 的示例

以下示例显示了几种不同格式的 TRACE 语句。该示例使用了 Windows™ 文件命名约定：

```
CREATE PROCEDURE testproc()
    DEFINE i INT;
    SET DEBUG FILE TO 'C:\tmp\test.trace';
    TRACE OFF;
    TRACE 'Entering foo';
    TRACE PROCEDURE;
    LET i = test2();

    TRACE ON;
    LET i = i + 1;

    TRACE OFF;
    TRACE 'i+1 = ' || i+1;
    TRACE 'Exiting testproc';

    SET DEBUG FILE TO 'C:\tmp\test2.trace';
```

END PROCEDURE

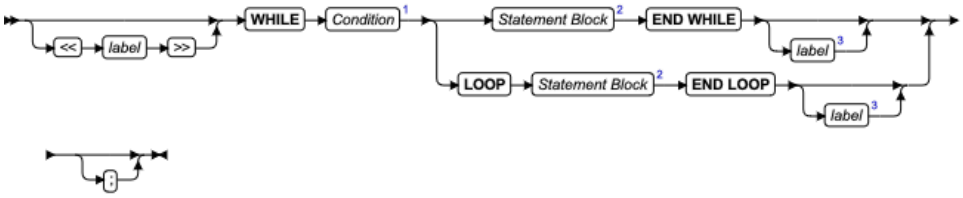
## 查看跟踪输出

要查看跟踪输出，请使用文本编辑器或类似的实用程序显示或阅读文件内容。

# 3.19 WHILE

使用 WHILE 语句建立具有变量结束条件的循环。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *label* | 此循环的循环标签的名称 | 在此 SPL 例程的标签中必须是唯一的 | 标识符 |

**用法**

在 ***statement block*** 第一次运行和每个后续迭代之前计算 ***condition*** 的值。只要 ***condition*** 保留为 true ，迭代继续。当 ***condition*** 的值不为 true 时，循环终止。

如果 ***condition*** 中的任何表达式求值为 NULL，则 ***condition*** 不为真，除非您正在用 IS NULL 运算符对 NULL 显式地进行测试。

如果 ***condition*** 中的表达式因为引用了未初始化的 SPL 变量而具有 UNKNOWN 关键字，则会导致错误。在这种情况下，循环终止，出现异常。

## SPL 例程中的 WHILE 循环的示例

以下示例说明了在 SPL 例程中的 WHILE 循环的使用。在 SPL 例程中，simp_while（第一个 WHILE 循环）执行 DELETE 语句。第二个 WHILE 循环执行 INSERT 语句并增加 SPL 变量的值。

```
CREATE PROCEDURE simp_while()
    DEFINE i INT;
    WHILE EXISTS (SELECT fname FROM customer
          WHERE customer_num > 400)
        DELETE FROM customer WHERE id_2 = 2;
    END WHILE;
    LET i = 1;
    WHILE i < 10
        INSERT INTO tab_2 VALUES (i);
```

```
        LET i = i + 1;
    END WHILE;
END PROCEDURE;
```

## Labeled WHILE 循环

要创建 Labeled WHILE 循环，您可以在初始的 WHILE 关键字之前声明循环标签，并在 END WHILE 关键字之后重复此标签，如以下两个 WHILE 循环的示例：

```
CREATE PROCEDURE ex_cont_ex()
      DEFINE i,s,j, INT;
      <<while_jlab>>
       WHILE j < 20
            IF j > 10 THEN
            CONTINUE WHILE;
            END IF
            LET i,s = j,0;
        <<while_slab>>
            WHILE i > 0
            LET i = i -1;
            IF i = 5 THEN
            EXIT while_jlab;
            END IF
      END WHILE while_slab
      END WHILE while_jlab
END PROCEDURE;
```

此处的 `EXIT while_jlab` 语句与 `EXIT` 或 `EXIT FOR` 关键字具有相同的效果，终止外部 WHILE 循环和例程。在这个例子中，包括 `EXIT while_jlab` 语句的语句具有与 `EXIT while_jlab WHEN i = 5` 相同的效果。

您还可以标记以紧跟在初始 WHILE 关键字和条件之前的循环 <<***label***>> 规范开头的 LOOP 语句。在这种类型的循环中，CONTINUE LOOP 、EXIT LOOP 和 END LOOP 关键字将替换 CONTINUE WHILE 、EXIT WHILE 和 END WHILE 关键字。LOOP 和 WHILE 关键字在 CONTINUE 和 EXIT 关键字之后都是可选的，但是在包含 LOOP 关键字的 SPL 循环语句中需要 END LOOP 关键字。

您可以使用类似的语法创建一个未标记的循环，省略紧跟在 WHILE 条件规范之前的 <<***label***>>声明。在这种情况下，还必须省略 END LOOP 关键字后面的未定义循环标签标识符。有关这些形式的标记和未标记的循环语句的描述和示例，请参阅 LOOP 语句，这些语句使您能够将 WHILE 语句语法及其基于条件的循环迭代数与 LOOP 语句的“循环永远”语法相结合。

相关语句

<< Label >> 语句 、CONTINUE 、EXIT 、LOOP

# 4. 数据类型和表达式

这些主题描述 GBase 8s 支持的数据类型和表达式。

这些基本的语法段可出现在数据定义语言（DDL）和数据操纵语言（DML）语句中，以及在其他类型 SQL 语句中。某些 SPL 语句也可指定数据类型或表达式。您可使用在不同的上下文中的关系型数据库或对象关系型数据库的这些特性，比如，来定义表的模式、来指定例程的签名和参数，或来表示或计算特定的数据值。

## 4.1 段描述的范围

每一段的描述包括下列信息：

- 说明该段的作用的简短介绍
- 展现如何正确地输入段的语法图
- 说明语法图中数据的表，你必须为其替换名称、值或其他特定的信息
- 用法规则，通常包括说明这些规则的示例

如果一段由多个部分组成，则段描述提供关于每一部分的类似的信息。有些描述以对此文档中和其他文档中相关信息的引用来结束。

## 4.2 段描述的使用

每一段描述内的语法图不是孤立的图。更准确地说，它是可包括该段的 SQL 语句（在 SQL 语句 中）或 SPL 语句（在 SPL 语句 中）的语法的子图。

SQL 或 SPL 语法描述可以两种方式引用段描述：

- 在语法图中的 ***subdiagram reference*** 可罗列段名称和此文档在该段描述开始处的页。
- 紧跟在语法图之后的表的**语法**列可罗列段名称以及该段描述起始处的页。

如果语句的语法图包括对段的引用，请转至那个段描述来查看该段的完整语法。

例如，如果您想要写包括 *view* 名称的 *database* 和 *database server* 限定符的 CREATE VIEW 语句，则首先查找 CREATE VIEW 语句 的语法图。图下的表引用 *view* 的语法的 Database Object Name 段。然后，使用 Database Object Name 段语法来输入有效的 CREATE VIEW 语句，还为该视图指定 *database* 和 *database server* 名称。在下例中， CREATE VIEW 语句在 boston 数据库服务器上的 sales 数据库中定义名为 name_only 的视图：

```
CREATE VIEW sales@boston:name_only AS
SELECT customer_num, fname, lname FROM customer;
```

除了本章记录的“数据类型”和“表达式”语法段之外，其它语法段 还提供在此文档的语法图中引用的附加的语法段。

## 4.3 数据类型和表达式段

数据类型和表达式段可出现在 SQL 语句中。

数据类型和表达式段可包括下列项：

- 数据类型
- DATETIME 字段限定符
- INTERVAL 字段限定符
- 表达式
- 聚集表达式
- AVG、COUNT、MAX、MIN、SUM、RANGE、STDDEV、VARIANCE *和*用户定义的聚集
- 算术表达式
- 二进制（+、-、*、/）运算符、运算符函数*和*一元（+、-）运算符
- 强制转型表达式
- CAST 函数*和* Cast（::）运算符
- 集合子查询
- 列表达式
- 列名称、ROWID *和*子串（[ ... ]）运算符
- CONCAT 函数*和*串联（||）运算符
- 条件段*和*条件表达式
- 比较条件：AND、OR、NOT、BETWEEN、IS NULL、LIKE、MATCHES *和*关系运算符
- 带有子查询的条件：IN、EXISTS、ALL、ANY *和* SOME 运算符
- 布尔 UDF
- CASE 表达式
- ISNULL 函数
- NVL 函数
- DECODE 函数
- 常量表达式：CURRENT、SYSDATE、TODAY、DBSERVERNAME、SITENAME、UNITS、CURRENT_USER *和* USER
- 文字值
- 文字集合
- 文字 DATETIME
- 文字 INTERVAL
- 精确数值
- 文字 Row
- 引用的字符串
- 构造函数表达式
- 集合构造函数
- ROW 构造函数表达式
- 代数函数：ABS、MOD、POW、POWER® ROOT、ROUND、SQRT *和* TRUNC 函数
- CARDINALITY 函数

- DBINFO 函数
- 加密和解密函数：DECRYPT_BINARY、DECRYPT_CHAR、ENCRYPT_AES、ENCRYPT_TDES *和* GETHINT 函数
- 指数和对数函数：EXP、LOGN *和* LOG10 函数
- HEX 函数
- 层级查询运算符和函数：CONNECT_BY_ROOT、PRIOR *和* SQL_CONNECT_BY_PATH
- IFX_ALLOW_NEWLINE 函数
- 长度函数：CHARACTER_LENGTH、CHAR_LENGTH、LENGTH *和* OCTET_LENGTH 函数
- 序列运算符：CURRVAL、NEXTVAL
- 智能大对象函数：FILETOBLOB、FILETOCLOB、LOCOPY *和* LOTOFILE 函数
- 字符串操纵函数：LPAD、RPAD、TRIM、REPLACE、SUBSTR、SUBSTRING、INITCAP、LOWER *和* UPPER 函数
- 时间函数：DATE、DAY、EXTEND、MDY、MONTH、TO_CHAR、GBase_TO_DATE、TO_DATE、WEEKDAY *和* YEAR 函数
- 触发器类型布尔运算符：DELETING、INSERTING、SELECTING *和* UPDATING
- 三角函数：ACOS、ASIN、ATAN、ATAN2、COS、SIN *和* TAN 函数
- 汉字转拼音函数：GetHzFullPY、GetHzPYCAP *和* GetHzFullPYsubstr 函数
- SYS_GUID 函数
- 列转行函数：WM_CONCAT、WM_CONCAT_TEXT 函数
- 用户定义的函数
- 语句-本地变量表达式

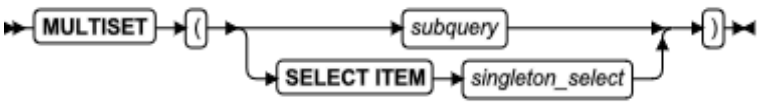
您还可使用主变量或 SPL 变量作为表达式。要获取带有页引用的字母列表，请参阅 表达式的列表。

## 4.4 集合子查询

您可使用“集合子查询”来从子查询的结果创建 MULTISET 集合。此语法是对 SQL 的 ANSI/ISO 标准的扩展。

**语法**

集合子查询




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *singleton_select* | 返回正好一行的子查询 | 子查询不可重复 SELECT 关键字，也不可包括 ORDER BY 子句 | SELECT 语句 |

| | | | |
|---|---|---|---|
| *subquery* | 嵌入的查询 | 不可包含 ORDER BY 子句 | SELECT 语句 |

**用法**

MULTISET 和 SELECT ITEM 关键字有下列重要意义：

- MULTISET 指定可包括重复值的元素，但没有特定元素的顺序的集合。
- SELECT ITEM 仅支持 projection 列表中的一个表达式。您不可在单个子查询中重复 SELECT 关键字。

您可在下列上下文中使用集合子查询：

- SELECT 语句的 Projection 子句和 WHERE 子句
- INSERT 语句的 VALUES 子句
- UPDATE 语句的 SET 子句
- 在您可使用集合表达式的任何地方（即，计算得到单个集合的任何表达式）
- 作为传递给用户定义的例程的一个参数

下列限制适用于集合子查询：

- Projection 子句不可包含重复的列（字段）名称。
- 它不可包含表名称的别名。（但它可使用列（字段）名称的别名，如下列一些示例中那样。）
- 它是只读的。
- 不可打开它两次。
- 它不可包含 NULL 值。
- 它不可包含尝试在子查询内搜索的语法。

集合子查询返回未命名的 ROW 数据类型的多重集。此 ROW 类型的字段是子查询的 projection 列表中的元素。下列示例访问表和这些语句定义的 ROW 类型：

```
CREATE ROW TYPE rt1 (a INT);
CREATE ROW TYPE rt2 (x int, y rt1);
CREATE TABLE tab1 (col1 rt1, col2 rt2);
CREATE TABLE tab2 OF TYPE rt1;
CREATE TABLE tab3 (a ROW(x INT));
```

下列结合子查询的示例返回罗列在该子查询右边的 MULTISET 集合。

| 集合子查询 | 结果集合 |
|---|---|
| MULTISET (SELECT * FROM tab1)... | MULTISET(ROW(col1 rt1, col2 rt2)) |
| MULTISET (SELECT col2.y FROM tab1)... | MULTISET(ROW(y rt1)) |
| MULTISET (SELECT * FROM tab2)... | MULTISET(ROW(a int)) |
| MULTISET(SELECT p FROM tab2 p)... | MULTISET(ROW(p rt1)) |
| MULTISET (SELECT * FROM tab3)... | MULTISET(ROW(a ROW(x int))) |

下列是另一个集合子查询：

```
SELECT f(MULTISET(SELECT * FROM tab1 WHERE tab1.x = t.y))
                                FROM t WHERE t.name = 'john doe';
```

下列集合子查询包括 UNION 运算符：

```
SELECT f(MULTISET(SELECT id FROM tab1
  UNION
SELECT id FROM tab2 WHERE tab2.id2 = tab3.id3)) FROM tab3;
```

### FROM 子句中的表表达式

GBase 8s 支持在 SELECT 查询和子查询的 FROM 子句中表表达式的 ANSI/ISO 标准语法，替代 GBase 8s 扩展集合子查询语法。在 10.00 和更早的版本中需要关键字 TABLE 和 MULTISET。支持对 SQL 的 ANSI/ISO 标准的这些扩展，但在 SELECT 语句的 FROM 子句中不再需要集合子查询。

下列两个查询返回相同的结果集，但仅第二个查询符合 ANSI/ISO 标准：

```
SELECT * FROM TABLE(MULTISET(SELECT col1 FROM tab1
        WHERE col1 = 100))
         AS vtab(c1),
  (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1) ORDER BY c1;

  SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
         (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1)
         ORDER BY c1;
```

相同的 SELECT 语句可为集合子查询组合 GBase 8s 扩展与 ANSI/ISO 语法二者的实例：

```
SELECT * FROM (select col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 10)) AS vtab1(vc1)
         ORDER BY c1;
```

集合子查询必须通过两种格式的圆括号定界，但紧跟在 TABLE 关键字之后并括在 MULTISET 集合子查询规范的圆括号(())的外部集是对 ANSI/ISO 语法的扩展。此 ANSI/ISO 语法仅在 SELECT 语句的 FROM 子句中是有效的。在任何其他上下文中，您不可省略来自集合子查询规范的这些关键字和圆括号。

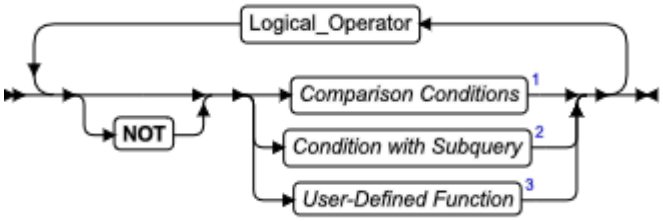
**重要：** FROM 子句中的集合子查询不可包括相关的表引用，也不可包括 LATERAL 关键字。

## 4.5 条件

使用条件来测试数据是否满足某些限定条件。在语法图中您看到对条件的引用的任何地方，请使用此段。

**语法**

条件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *Logical_Operator* | 组合两个条件 | 有效的选项是 OR ( = *logical union*) *或* AND ( = *logical intersection*) | 带有 AND 或 OR 的条件 |

**用法**

***条件***是搜索标准，通过逻辑运算符 AND 或 OR 可选地连接起来。可将条件划分为下列几类：

- 比较条件（也称为过滤器或布尔表达式）
- 带有子查询的条件
- 用户定义的函数（仅限于 GBase 8s ）

条件可包含聚集函数，仅当它用在 SELECT 语句的 HAVING 子句中，或在子查询的 HAVING 子句中。

在 DELETE、SELECT 或 UPDATE 语句的 WHERE 子句中的条件中不可出现聚集函数，除非下列二者都是 TRUE：

- 起源于父查询的相关列上的聚集。
- WHERE 子句出现在 HAVING 子句内的子查询中。

在 GBase 8s 中，在下列上下文中，用户定义的函数作为条件是无效的：

- 在 SELECT 语句的 HAVING 子句中
- 在检查约束的定义中

在下列上下文中，SPL 例程作为条件是无效的：

- 在检查约束的定义中
- 在 SELECT 语句的 ON 子句中
- 在 DELETE、SELECT 或 UPDATE 语句的 WHERE 子句中

在下列上下文中，外部的例程作为条件是无效的：

- 在检查约束的定义中
- 在 SELECT 语句的 ON 子句中
- 在 DELETE、SELECT 或 UPDATE 语句的 WHERE 子句中
- 在 CREATE TRIGGER 的 WHEN 子句中
- 在 SPL 的 IF、CASE 或 WHILE 语句中

## 比较条件（布尔表达式）

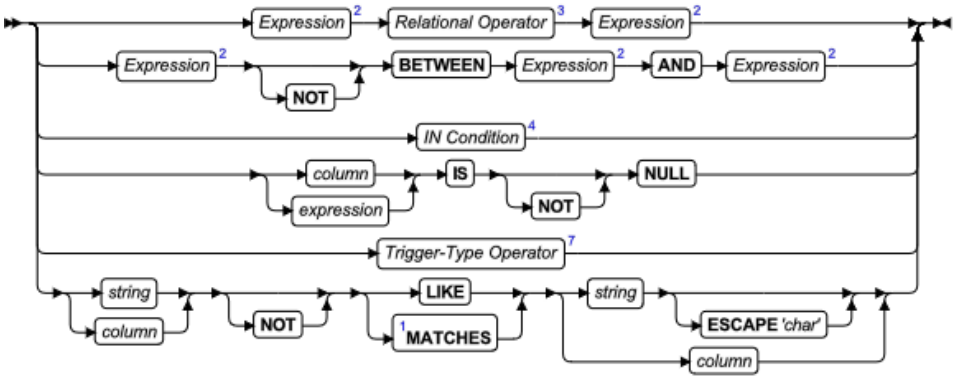
比较表达式常被称为***布尔表达式***，因为它们返回 TRUE 或 FALSE 结果。

六种布尔运算符可指定比较条件：

- 关系运算符
- [NOT] BETWEEN … AND 运算符
- [NOT] IN 运算符
- IS [NOT] NULL 运算符
- 触发器类型运算符
- [NOT] LIKE 或 MATCHES 运算符

在此图中总结它们的语法并在后面的部分说明。

比较条件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *char* | 在括起来的字符串中要作为转义字符的 ASCII 字符。单引号（’）和双引号（″）作为 *char* 是无效的。 | 请参阅 ESCAPE 与 LIKE 一起使用 和 ESCAPE 与 MATCHES 一起使用 | 引用字符串 |
| *column* | 列名称（或 ROW 类型列的字段），以其数据值与 NULL、与 *string* 或与另一 *column* 做比较 | 可通过标识符、同义词或表或视图的别名来限定。 | 请参阅列名称 |
| *expression* | 返回单个值的 SQL 表达式 | 必须返回单个值 | 表达式 |
| *string* | 通过单引号（’）或双引号（″）定界的字符串 | 两个定界符必须是相同的 | 请参阅引用字符串 |

下列部分描述比较条件的不同类型：

- 关系运算符条件
- BETWEEN 条件
- IN 条件
- IS NULL 和 IS NOT NULL 条件

- LIKE 和 MATCHES 条件。

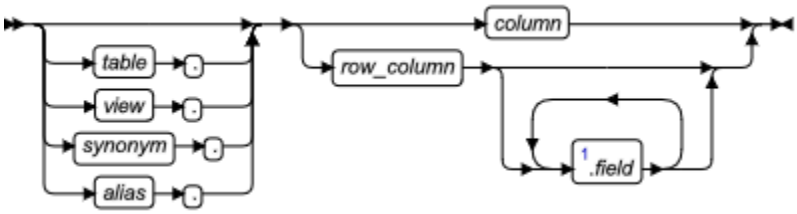
要获取在 SELECT 语句的上下文中比较条件的讨论，请参阅 在 WHERE 子句中使用条件。

**警告：** 比较条件中的字面 DATE 或 DATETIME 值应为年份指定 4 为数字。当您指定 4 位字符年份时，DBCENTURY 环境变量对结果不起作用。当您指定 2 位数字年份时，DBCENTURY 可影响数据库服务器解释比较条件的方式，这可能产生您不希望的结果。要获取更多关于 DBCENTURY 的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 列名称

Column Name 段可为比较条件中的一个元素。列的名称（或 ROW 数据类型的列内一个或多个字段）不是比较的主体，但数据库服务器使用此 SQL 标识符来访问数据库表或视图中指定列的或行字段的数据值。

列名称




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 表或视图的临时的替换名称 | 必须在 SELECT 语句的 FROM 子句中定义 | 标识符 |
| *column* | 列的名称 | 在指定的表中必须存在 | 标识符 |
| *field* | 在 ROW 类型列中要比较的字段 | 必须是 *row-column name* 或 *field name*（对于嵌套的行）的组件 | 标识符 |
| *row_column* | 类型 ROW 的列 | 必须是现有的命名的 ROW 类型或未命名的 ROW 类型 | 标识符 |
| *synonym*、*table*、*view* | 同义词、表或视图的名称 | *synonym* 和它指向的表或视图必须在数据库中存在 | 标识符 |

要获取更多关于在这些条件中 ***column*** 名称的含义的信息，请参阅 IS NULL 和 IS NOT NULL 条件 和 LIKE 和 MATCHES 条件。

## 条件中的引号

当您将列表达式与任何比较条件中的常量表达式做比较时，请遵守下列规则：

- 如果该列有数值数据类型，则请不要在引号之间括起常量表达式。
- 如果该列有字符数据类型，则请在引号之间括起常量表达式。

- 如果该列有时间数据类型，则请在引号之间括起常量表达式。

否则，您可能得不到期望的结果。

下列示例展示在比较条件中引号的正确用法。在此，ship_instruct 列有字符数据类型，order_date 列有日期数据类型，而 ship_weight 列有数值数据类型。

```
SELECT * FROM orders
WHERE ship_instruct = 'express'
AND order_date > '05/01/98'
AND ship_weight < 30;
```

## 关系运算符条件

关系运算符定量地比较两个表达式。

要获取受到支持的关系运算符及其描述的列表，请参阅 关系运算符。

下列示例展示一些关系运算符条件：

```
city[1,3] = 'San'
o.order_date > '6/12/98'
WEEKDAY(paid_date) = WEEKDAY(CURRENT- (31 UNITS DAY))
YEAR(ship_date) < YEAR (TODAY)
quantity <= 3
customer_num <> 105
customer_num != 105
```

关系运算符条件中的运算对象不可有 UNKNOWN 或 NULL 值。如果 ***condition*** 内的表达式有 UNKNOWN 值，则由于它引用未初始化的变量，数据库服务器会产生异常。

**NULL 值的条件测试**

如果 *condition* 内的任何表达式求值为 NULL，则 *condition* 不可为真，除非您正在使用 IS NULL 运算符显式地进行测试。例如，如果 paid_date 列有 NULL 值，则下列查询都不可检索那一行：

```
SELECT customer_num, order_date FROM orders
        WHERE paid_date = '';
SELECT customer_num, order_date FROM orders
        WHERE NOT (paid_date !='');
```

您必须使用 IS NULL 运算符来测试 NULL 值，如下一示例所示。

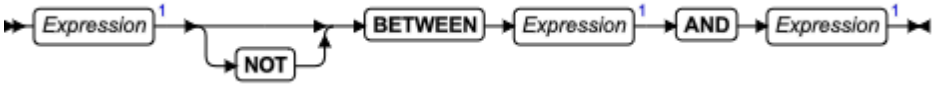
```
SELECT customer_num, order_date FROM orders
        WHERE paid_date IS NULL;
```

在 IS NULL 和 IS NOT NULL 条件 中描述 IS NULL 运算符及其逻辑反、IS NOT NULL 运算符。

## BETWEEN 条件

使用 BETWEEN 条件来测试数字表达式、字符表达式或时间表达式的值是否在指定的范围内。

BETWEEN 条件




用法

NULL 值不可满足该条件。定义的范围可求值为 NULL 的表达式也不可满足。

BETWEEN 条件中的三个表达式必须满足这些限制：

- 所有三个表达式都必须求值为相互可比的数值、时间或字符数据类型。
- 紧跟在 BETWEEN 关键字之后的表达式的值必须小于跟在 AND 关键字之后的表达式的值。

**BETWEEN 条件中的数值和时间表达式**

对于数值表达式，*小于*意味着在数轴的左边。

对于 DATE 和 DATETIME 表达式，*小于*意味着时间较早。

对于 INTERVAL 表达式，*小于*意味着更短的时间跨度。

**BETWEEN 条件中的字符表达式**

对于 CHAR、VARCHAR 和 LVARCHAR 表达式，*小于*意味着在代码集顺序*之前*。

对于 NCHAR 和 NVARCHAR 表达式，*小于* 意味着在本地化的排序顺序*之前*，如果存在一个的话；否则*小于*意味着在代码集合顺序*之前*。

如果该语言环境定义了排序顺序的话，则将基于语言环境的排序顺序用于 NCHAR 和 NVARCHAR 表达式。因此，对于 NCHAR 和 NVARCHAR 表达式，*小于*意味着在基于语言环境的排序顺序*之前*。要获取更多关于基于语言环境的排序顺序以及 NCHAR 和 NVARCHAR 数据类型的信息，请参阅 *GBase 8s GLS 用户指南*。

要获取关于在有 NLCASE INSENSITIVE 属性的数据库中带有 NCHAR 和 NVARCHAR 运算对象的关系运算符表达式如何不同于在区分大小写的数据库中它们的行为的信息，请参阅主题 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

**BETWEEN 条件中的 NOT 关键字**

对于要为 TRUE 的 BETWEEN 条件，依赖于您是否包括 NOT 关键字。

- 如果您省略 NOT 关键字，则仅当 BETWEEN 关键字左边的表达式的值在 BETWEEN 关键字右边的两个表达式的包括范围之中时，BETWEEN 条件才为 TRUE。
- 如果 NOT 关键字在 BETWEEN 关键字的紧前边，则仅当 BETWEEN 关键字左边的表达式的值不在 BETWEEN 关键字的右边的两个表达式的值的包括范围之中时，BETWEEN 条件才为 TRUE。

否则，BETWEEN 条件为 FALSE。

BETWEEN 条件的示例

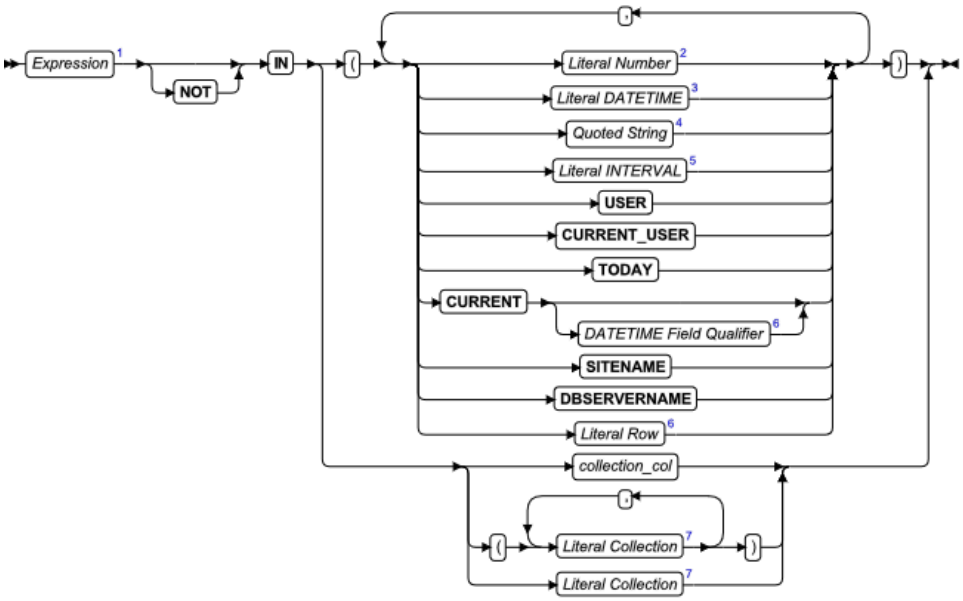
下列示例说明 BETWEEN 条件：

```
order_date BETWEEN '6/1/97' and '9/7/97'
zipcode NOT BETWEEN '94100' and '94199'
EXTEND(call_dtime, DAY TO DAY) BETWEEN
      (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT
lead_time BETWEEN INTERVAL (1) DAY TO DAY
      AND INTERVAL (4) DAY TO DAY
unit_price BETWEEN loprice AND hiprice
```

## IN 条件

当项列表中包括该关键字左边的表达式时，满足 IN 条件。

IN 条件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *collection_col* | 在 IN 条件中使用的集合列的名称 | 该列必须在指定的表中存在 | 标识符 |

如果您指定 NOT 运算符，则当该表达式不在项的列表中时，该 IN 条件为 TRUE。NULL 值不满足 IN 条件。

下列示例展示一些 IN 条件：

```
WHERE state IN ('CA', 'WA', 'OR')
WHERE manu_code IN ('HRO', 'HSK')
WHERE user_id NOT IN (USER)
WHERE order_date NOT IN (TODAY)
```

在 GBase 8s ESQL/C 中，在执行时刻对内建的 TODAY 函数求值。当打开游标或当执行查询时，对内建的 CURRENT 函数求值，如果它是单个 SELECT 语句的话。

内建的 USER 函数区分大小写；例如，它将 **minnie** 与 **Minnie** 解释为不同的值。

# 使用带有集合数据类型的 IN 运算符

您可使用 IN 运算符来确定集合中是否包含某个元素。

集合可为简单的集合或嵌套的集合。（在***嵌套的***集合类型中，集合的元素类型也是集合类型。）当您使用 IN 来搜索集合中的元素时，IN 左边或右边的表达式不可包含 BYTE 或 TEXT 数据类型。

假设您创建包含两个集合列的下列表：

```
CREATE TABLE tab_coll
        (
        set_num SET(INT NOT NULL),
        list_name LIST(SET(CHAR(10) NOT NULL) NOT NULL)
        );
```

下列语句片段展示您可能对 tab_coll 表的集合列上的搜索条件使用 IN 运算符的方式：

```
WHERE 5 IN set_num
WHERE 5.0::INT IN set_num
WHERE "5" NOT IN set_num
WHERE set_num IN ("SET{1,2,3}", "SET{7,8,9}")
WHERE "SET{'john', 'sally', 'bill'}" IN list_name
WHERE list_name IN ("LIST{""SET{'bill','usha'}"",
""SET{'ann' 'moshi'}""}",
"LIST{""SET{'bob','ramesh'}"",
""SET{'bomani' 'ann'}""}")
```

通常，当您在集合数据类型上使用 IN 运算符时，数据库服务器检查 IN 运算符左边的值是否是 IN 运算符右边值的集合中的一个元素。

## IS NULL 和 IS NOT NULL 条件

如果紧接在 IS 关键字之前的术语指定下列未定义的值之一，则满足该 IS NULL 条件：

- 包含空值的 ***column*** 的名称。
- 求值为空的 ***expression***。

反之，如果您使用 IS NOT NULL 运算符，则当 ***column*** 包含一个非空的值时，或当紧接在 IS NOT NULL 关键字之前的 ***expression*** 求值不为空时，满足该条件。

假设您希望在可包含 NULL 值的列上执行算术计算。您可创建表、将值插入到表内，然后为了数据计算执行一使用将空值转换为 0 的通用的 CASE 表达式的查询:

```
CREATE TABLE employee (emp_id INT, savings_in_401k INT, total_salary INT);

    INSERT INTO employee VALUES(1, 5000, 40000);
    INSERT INTO employee VALUES(2, 0, 40000);
    INSERT INTO employee VALUES(3, NULL, 100000);

    SELECT emp_id, savings_in_401k AS employer_match FROM employee WHERE
    CASE WHEN(savings_in_401k IS NULL) THEN 0
    ELSE savings_in_401k END * 0.06 > 0;
```

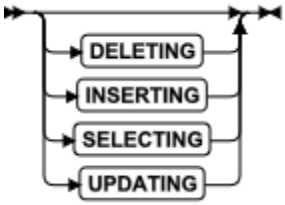
此示例展示通过使用 CASE 表达式中的 IS NULL，您可为不可计算的条目提供值，因为空不是有效的数值。

如果该列包含空值，或如果该表达式由于包含一个或多个空值而不可求值，则满足 IS NULL 条件。如果您使用 IS NOT NULL 运算符，则当运算对象是不为空的列值，或是求值不为空的表达式时，满足该条件。

## 触发器类型的布尔运算符

GBase 8s 的触发器类型的布尔运算符可在运行时测试当前正在执行的触发器活动是否是通过 DML 事件的指定的类型触发了的。这些运算符不使用操作对象。

触发器类型的布尔运算符




如果当前正在执行的触发器的触发事件是对应于操作符的名称的 DML 操作，则这些操作符返回 TRUE ('t')，否则它们返回 FALSE ('f')。在 IF 语句中，在 CASE 表达式中，以及在布尔条件为有效的 SPL 触发器例程内的其他上下文中，这些操作符是有效的。

例如，在下列语句片断中，仅当通过 INSERT 事件激活的当前正在执行的触发器时，才执行第一个 THEN 子句中的 LET 语句，且仅当通过 DELETE 事件激活了该触发器时，才执行第二个 THEN 子句中的 LET 语句：

```
IF (INSERTING = 't') THEN
LET square = NEW.X * NEW.X
ELIF (DELETING = 't') THEN
LET square = 0
```

仅在表上触发器或（对于 DELETING、INSERTING 和 UPDATING 运算符）在视图上 INSTEAD OF 触发器的 FOR EACH ROW 触发的活动中调用的触发器 UDR 中，SELECTING、DELETING、INSERTING 和 UPDATING 运算符才是有效的。如果您尝试在任何其他的上下文中使用触发器类型的布尔运算符，则发出错误。

如果通过 MERGE 语句已激活了的 Delete、Insert 或 Update 触发器调用触发器例程，则

- 在 MERGE 正在从目标表删除行时，DELETING 返回 TRUE。
- 在 MERGE 正在将行插入到目标表内时，INSERTING 返回 TRUE。
- 在 MERGE 正在更新目标表的行时，UPDATING 返回 TRUE。

## LIKE 和 MATCHES 条件

LIKE 或 MATCHES 条件测试字符串的匹配。

当下列测试为 TRUE 时，条件为 TRUE，或满足：

- 左边的列的值与括起来的字符串指定的模式相匹配。您可在字符串中使用通配符。NULL 值不满足该条件。
- 左边的列的值与右边指定的列的模式相匹配。右边的列的值用作该条件中的匹配模式。

如果括起来的字符串包括字面字符，这些字符与 LIKE 或 MATCHES 运算符识别的任何通配符相匹配，则 ESCAPE 子句可定义您可在括起来的字符串中包括的 ASCII 字符。当将左边的列值与括起来的字符串相比较时，将紧跟在***转义字符***之后的下一字符解释为字面字符，而不解释为通配符，且忽略该转义字符。LIKE 和 MATCHES 运算符识别不同的通配符。要获取更多关于 LIKE 和 MATCHES 转义字符的信息，请参阅 ESCAPE 与 LIKE 一起使用 和 ESCAPE 与 MATCHES 一起使用 主题。

您仅可使用带有括起来的字符串的单引号（'）来匹配字面的单引号；您不可使用 ESCAPE 子句。您可使用单引号字符作为与任何其他模式相匹配的转义字符，如果您将它写作 '''' 这样的话。

**重要：**您在 LIKE 或 MATCHES 条件中指定的列应为简单的字符数据类型，像 CHAR、LVARCHAR、NCHAR、NVARCHAR 或 VARCHAR。例如，您不可在 LIKE 或 MATCHES 条件中指定复合的数据类型，诸如 ROW 类型列。（ROW 类型列是声明为命名的或未命名的 ROW 类型的列。）类似地，数据库服务器不可对使用带有简单大对象或智能大对象列（诸如 CLOB 列）的 LIKE 或 MATCHES 的条件求值；包括此条件的查询失败并报错 -640。

## NOT 运算符

当左边的列有一非 NULL 的值，且与括起来的字符串指定的模式不匹配时，NOT 运算符使得该搜索条件成功。

例如，下列条件排除 lname 列中以字符 Baxter 开头的所有行：

```
WHERE lname NOT LIKE 'Baxter%'
WHERE lname NOT MATCHES 'Baxter*'
```

## LIKE 运算符

LIKE 是用于将列值与另一列值或括起来的字符串相比较的 ANSI/ISO 标准运算符。

LIKE 运算符支持括起来的字符串中的这些通配符。

| 通配符 | 作用 |
|---|---|
| % | 与零个或多个字符相匹配 |
| _ | 与任何单个字符相匹配 |

除了 % 和 _ 之外，当 DEFAULTESCCHAR 配置参数和 DEFAULTESCCHAR 会话环境变量都未设置时，LIKE 支持第三个通配符：

| 通配符 | 作用 |
|---|---|
| \ | 移除下一字符的特殊意义（通过指定 \% 或 _ 或 \\ 来匹配字面的 % 或 _ 或 \） |

使用反斜杠（\）符号作为缺省的转义字符（当未设置 DEFAULTESCCHAR 时） 是对 SQL 的 ANSI/ISO 标准的 GBase 8s 扩展。通过将 DEFAULTESCCHAR 值设置为那个字符，您可指定反斜

杠（\）符号或某些其他 ASCII 字符作为缺省的转义字符。要获取更多信息，请参阅 DEFAULTESCCHAR 环境选项。

在符合 ANSI 的数据库中，您仅可使用 LIKE 转义字符来转义百分号（%）、下划线（_）或转义字符自身。

下列条件单独或在更长的字符串中测试字符串 tennis 的 description 列，诸如 tennis ball 或 table tennis paddle：

```
WHERE description LIKE '%tennis%' ESCAPE '\'
```

下一个示例测试包含一下划线字符的行的 description。在此，反斜杠（\）转义字符是必要的，因为下划线（_）是通配符。

```
WHERE description LIKE '%\_%' ESCAPE '\'
```

LIKE 运算符有一相关联的名为 **like( )** 的运算符函数。您可定义 **like( )** 函数来处理您自己的用户定义的数据类型。另请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## MATCHES 运算符

MATCHES 运算符是用于将列值与另一列值或括起来的字符串相比较的 GBase 8s 扩展。

MATCHES 运算符支持括起来的字符串中的这些通配符。

| 通配符 | 作用 |
|---|---|
| * | 与零个或多个字符的任何字符串相匹配 |
| ? | 与任何单个字符相匹配 |
| [...] | 与包括范围的任何括起来的字符相匹配，如在 [a-z] 中那样。不可转义方括号内的字符。 |
| ^ | 作为方括号内的第一个字符，与未罗列的任何字符相匹配。因此，[^abc] 与除了 a、b 或 c 之外的任何字符相匹配。 |
| \ | 移除下一字符的特殊意义（通过指定 \\ 或 * 或 \?，等等，来与字面的 \ 或任何其他通配符相匹配） |

下列条件单独或在更长的字符串内测试字符串 tennis，诸如 tennis ball 或 table tennis paddle：

```
WHERE description MATCHES '*tennis*'
```

对于名称 Frank 和 frank，下列条件为 TRUE：

```
WHERE fname MATCHES '[Ff]rank'
```

对于以 F 或 f 开头的任何名称，下列条件为 TRUE：

```
WHERE fname MATCHES '[Ff]*'
```

对于任何以字母 a、b、c 或 d 结尾的任何名称，下一条件为 TRUE：

```
WHERE fname MATCHES '*[a-d]'
```

MATCHES 有一相关联的 **matches( )** 运算符函数。您可为您自己的用户定义的数据类型定义 **matches( )** 函数。要获取更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

如果 **DB_LOCALE** 或 SET COLLATION 指定支持本地化排序的非缺省的语言环境，且您使用方括号（[...]）符号为 MATCHES 运算符指定范围，则数据库服务器使用本地化的排序顺序，而不是代码集顺序，来解释该范围并比较那些有 CHAR、CHARACTER VARYING、LVARCHAR、NCHAR、NVARCHAR 和 VARCHAR 数据类型的值。

通常的规则是，仅可在本地化的排序顺序中比较 NCHAR 和 NVARCHAR 数据类型，此行为是该规则的例外。要获取更多关于包括 MATCHES 或 LIKE 运算符的条件的 GLS 方面的信息，请参阅 *GBase 8s GLS 用户指南*。

在 NLSCASE INSENSITIVE 数据库中，对 NCHAR 和 NVARCHAR 数据的比较操作不理会大小写的差异，因此数据库服务器将包含相同序列字符的字符串之中的大写变量作为重复处理。作为 MATCH 运算符的运算对象，下列字符串的所有对都返回 TRUE：

'beta' 'Beta' 'BETA' 'bETa' 'betA' 'BetA'

要获取更多信息，请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行 和 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## ESCAPE 与 LIKE 一起使用

ESCAPE 子句可指定与缺省的转义字符不同的一个转义字符。通过 DEFAULTESCCHAR 配置参数或 DEFAULTESCCHAR 会话环境选项设置缺省的转义字符。

例如，如果您在 ESCAPE 子句中指定 z，则将包含了 z_ 的括起来的字符串运算对象解释为包括字面的下划线（_）字符，而不将 _ 作为通配符。 类似地，将 z% 解释作为字面的百分号（% ），而不将 % 当做通配符。最后，会将字符串中的字符 zz 解释为单个字面的 z。 下列语句从 customer 表检索行，其中的 company 列包括字面的下划线字符：

```
SELECT * FROM customer WHERE company LIKE '%z_%' ESCAPE 'z';
```

您还可使用包含单个字符的主变量。下一语句使用主变量来指定一转义字符：

```
EXEC SQL BEGIN DECLARE SECTION;
        char escp='z';
        char fname[20];
        EXEC SQL END DECLARE SECTION;
        EXEC SQL select fname from customer
        into :fname where company like '%z_%' escape :escp;
```

## ESCAPE 与 MATCHES 一起使用

ESCAPE 子句可指定与缺省的转义字符不同的转义字符。通过 DEFAULTESCCHAR 配置参数或 DEFAULTESCCHAR 会话环境选项设置缺省的转义字符。

使用此作为您想要的缺省的转义字符，反斜杠，来在括起来的字符串中包括问号（?）、星号（*）、插入符（^） 或左方括号（[）或右方括号（]）作为字面的字符，以防止将它们解释为特殊字符。如果您选择使用 z 作为该转义字符，则字符串中的字符 z? 代表字面的问号（?）。类似地，字符 z* 代表字面的星号（*）。最后，字符串中的字符 zz 代表单个字符 z。

下列示例从 customer 表检索行，其中的 company 列的值包括问号（?）：

```
SELECT * FROM customer WHERE company MATCHES '*z?*' ESCAPE 'z';
```

## 独立条件

独立条件可为没有显式地罗列在比较条件的语法中的任何表达式。仅当表达式返回 BOOLEAN 值时，它作为条件才是有效的。例如，下列示例返回 BOOLEAN 数据类型的值：

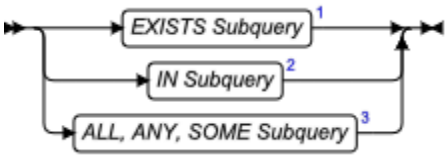
```
funcname(x)
```

## 带有子查询的条件

在条件内包括 SELECT 语句，指定带有子查询的条件。您可在 SELECT、INSERT、DELETE 或 UPDATE 语句中使用子查询来执行下列这样的任务：

- 将表达式与查询的结果作比较。
- 确定查询的结果中是否包括表达式。
- 询问查询是否选择任何行。

带有子查询的条件




子查询可依赖于外部 SELECT 语句正在求值的当前行；在此情况下，该子查询称为***相关的子查询***。（要获取相关的子查询及其对性能的影响的讨论，请参阅 *GBase 8s SQL 教程指南*。）

下列部分描述子查询条件及其语法。

- 要获取在 SELECT 语句的上下文中子查询条件的类型的讨论，请参阅 在 WHERE 子句中使用条件。
- 要获取在 INSERT 语句的上下文中子查询条件的类型的讨论，请参阅 SELECT 语句的子集。
- 要获取在 DELETE 语句的上下文中子查询条件的类型的讨论，请参阅 DELETE 的 WHERE 子句中的子查询。
- 要获取在 UPDATE 语句的上下文中子查询条件的类型的讨论，请参阅 UPDATE 的 WHERE 子句中的子查询。

依赖于子查询的上下文，子查询可返回单个值、无值或值集。如果子查询返回值，它必须仅选择单个列。如果子查询简单地检查一行（或多行）是否存在，则它可选择任何数目的行和列。

子查询不可引用 BYTE 或 TEXT 列，也不可包含 ORDER BY 子句。然而，在 FROM 子句中指定表表达式的子查询可包括 ORDER BY 子句。

如果子查询的 FROM 子句指定外部语句在这些子句之一中引用的同一表或视图，则子查询及其外部 DML 语句在同一表对象上操作：

- 在 DELETE 或 SELECT 语句的 FROM 子句中
- 在 INSERT 语句的 INTO 子句中

- 在 UPDATE 语句的“表选项”或“集合派生的表”规范中。

仅在 DELETE 或 UPDATE 语句的 WHERE 子句中，那些返回多行和与括起来的 DML 语句操作的同一表或视图的子查询才是有效的。即使在此上下文中，这样的子查询也返回错误 -360，除非满足所有下列条件：

- 该子查询不引用它的 FROM 列表中的任何列名称，它在 projection 列表中未指定的表中
- 使用带有 Subquery 语法的 Condition 指定该子查询。
- 该子查询内的任何 SPL 例程不可引用正在修改的表。

下列程序片断包括在 UPDATE 和 DELETE 语句中带有子查询的条件的示例：

```
CREATE TABLE t1 ( a INT, a1 INT)
CREATE TABLE t2 ( b INT, b1 INT) ;
. . .
UPDATE t1 SET a = a + 10 WHERE EXISTS
(SELECT a FROM t1 WHERE a > 1);
UPDATE t1 SET a = a + 10 WHERE a IN
(SELECT a FROM t1, t2   WHERE a > b
AND a IN
(SELECT a FROM t1 WHERE a > 50 ) );
DELETE FROM t1 WHERE EXISTS
(SELECT a FROM t1);
```

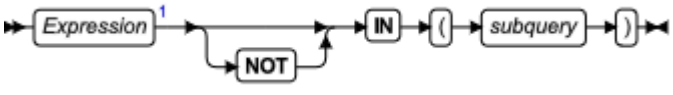
要获取更多关于在 DELETE 语句中的子查询的信息，请参阅 DELETE 的 WHERE 子句中的子查询。

要获取更多关于在 UPDATE 语句中的子查询的信息，请参阅 UPDATE 的 WHERE 子句中的子查询。

## IN 子查询

如果表达式的值与来自子查询的一个或多个值相匹配，则 IN 子查询条件为 TRUE。（该子查询必须仅返回一行，但它可返回多个列。）关键字 IN 等同于 =ANY 规范。关键字 NOT IN 等同于 !=ALL 规范。请参阅 ALL、ANY 和 SOME 子查询。

IN 子查询




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *subquery* | 内嵌的查询 | 不可包含 FIRST 子句也不可包含 ORDER BY 子句 | SELECT 语句 |

下列 IN 子查询的示例查找不包含 baseball gloves （stock_num = 1）的订单的订单号：

```
WHERE order_num NOT IN
    (SELECT order_num FROM items WHERE stock_num = 1)
```

由于 IN 子查询测试行的出现，因此子查询结果中的重复的行不影响主查询的结果。因此，子查询中的 UNIQUE 或 DISTINCT 关键字对查询结果不起作用，尽管不测试重复可提升查询性能。

## EXISTS 子查询条件

如果子查询返回行，则 EXISTS 子查询条件求值为 TRUE。以 EXISTS 子查询，可返回一个或多个列。该子查询总是包含对主查询中表的列的引用。如果您在不包含 HAVING 子句的 EXISTS 子查询中使用聚集函数，则总会返回至少一行。

EXISTS 子查询


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *subquery* | 内嵌的查询 | 不可包含 FIRST 子句也不可包含 ORDER BY 子句 | SELECT 语句 |

下列带有 EXISTS 子查询的 SELECT 语句的示例返回从未被订购（且因此未罗列在 items 表中）的每个项的存货编号和制造商代码。您可在此 SELECT 语句中适当地使用 EXISTS 子查询，因为您使用子查询来测试 items 中的 stock_num 以及 manu_code。

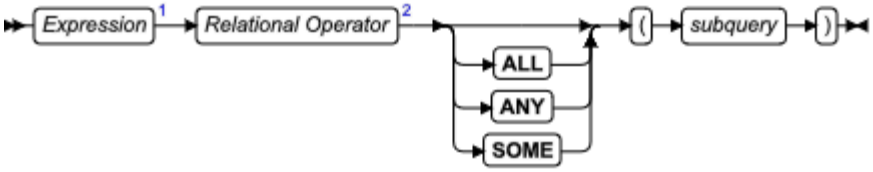
```
SELECT stock_num, manu_code FROM stock
    WHERE NOT EXISTS (SELECT stock_num, manu_code FROM items
    WHERE stock.stock_num = items.stock_num AND
          stock.manu_code = items.manu_code);
```

如果您在列名称的位置中的子查询中使用 SELECT *，则前面的示例同样奏效，因为测试整行的存在；不测试特定的列值。

## ALL、ANY 和 SOME 子查询

使用 ALL、ANY 和 SOME 关键字来指定何种情况下产生条件 TRUE 或 FALSE。当使用 ALL 关键字时，当使用 ANY 关键字时为 TRUE 的搜索条件可能不为 TRUE，反之亦然。

ALL、ANY、SOME 子查询




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *subquery* | 嵌入的查询 | 不可包含 FIRST 或 ORDER BY 子句 | SELECT 语句 |

## *使用 ALL 关键字*

如果子查询返回的每个值的比较都为 TRUE，则 ALL 关键字指定该搜索条件为 TRUE。如果子查询未返回值，则该条件为 TRUE。

在下列示例中，第一个条件测试是否每一 total_price 都大于订单号 1023 中每个项的总价。第二个条件使用 MAX 聚集函数来产生同样的结果。

```
total_price > ALL (SELECT total_price FROM items
  WHERE order_num = 1023)

  total_price > (SELECT MAX(total_price) FROM items
      WHERE order_num = 1023)
```

使用带有 ALL 子查询的 NOT 关键字测试对于子查询返回的至少一个元素是否有一个表达式不为 TRUE。例如，当表达式 total_price 不大于所有被选择的值时，下列条件为 TRUE。也就是说，当 total_price 不大于订单号 1023 中最高的总价时，它为 TRUE。

```
NOT total_price > ALL (SELECT total_price FROM items
      WHERE order_num = 1023)
```

## *使用 ANY 或 SOME 关键字*

ANY 关键字表示，如果对于至少一个返回值的比较是 TRUE，则搜索条件为 TRUE。如果子查询未返回值，则搜索条件为 FALSE。SOME 关键字是 ANY 的同义词。

当总价大于订单号 1023 中至少一个项的总价时，下列条件为 TRUE。第一个条件使用 ANY 关键字；第二个使用 MIN 聚集函数：

```
total_price > ANY (SELECT total_price FROM items
      WHERE order_num = 1023)

total_price > (SELECT MIN(total_price) FROM items
      WHERE order_num = 1023)
```

使用带有 ANY 子查询的 NOT 关键字测试对于子查询返回的所有元素表达式是否不是 TRUE。例如，当表达式 total_price 不大于任何被选择的值时，下列条件为 TRUE。也就是说，当 total_price 不大于订单号 1023 中的总价时，它是 TRUE。

```
NOT total_price > ANY (SELECT total_price FROM items
     WHERE order_num = 1023)
```

## *省略 ANY、ALL 或 SOME 关键字*

如果您知道子查询将正好返回一个值，则您可省略子查询中的关键字 ANY、ALL 或 SOME。如果您省略 ANY、ALL 或 SOME 关键字，且该子查询返回多个值，则您收到错误。在下列示例中的子查询仅返回一行，因为它使用聚集函数：

```
SELECT order_num FROM items
   WHERE stock_num = 9 AND quantity =
    (SELECT MAX(quantity) FROM items WHERE stock_num = 9);
```

## NOT 运算符

如果您以关键字 NOT 作为条件的开始，则仅当 NOT 限定的条件为 FALSE 时，该测试才是 TRUE。如果 NOT 限定的条件有一 NULL 或一 UNKNOWN 值，则 NOT 运算符无效。

下列真值展示带有 3– 有值的布尔运算对象的 NOT 的作用。在此，T 代表 TRUE 条件，F 代表 FALSE 条件，且问号（?）表示 UNKNOWN 条件。（当运算对象为 NULL 时，可发生 UNKNOWN 值）。

| NOT | |
|---|---|
| T | F |
| ? | ? |
| F | T |

左边的列展示 NOT 运算符的运算对象的值，右边的列展示将 NOT 应用于操作对象之后返回的值。

## 带有 AND 或 OR 的条件

您可将简单的条件与逻辑运算符 AND 或 OR 组合来形成复合的条件。

下列 SELECT 语句在它们的 WHERE 子句中包含复合的条件的示例：

```
SELECT customer_num, order_date FROM orders
    WHERE paid_date > '1/1/97' OR paid_date IS NULL;
SELECT order_num, total_price FROM items
    WHERE total_price > 200.00 AND manu_code LIKE 'H
 SELECT lname, customer_num FROM customer
    WHERE zipcode BETWEEN '93500' AND '95700'
          OR state NOT IN ('CA', 'WA', 'OR');
```

下列真值表展示 AND 和 OR 运算符的作用。字母 T 代表 TRUE 条件，F 代表 FALSE 条件，问号（?）代表 UNKNOWN 值。当使用逻辑运算符的表达式的一部分为 NULL 时，可发生 UNKNOWN 值。

| OR | T | ? | F |
|---|---|---|---|
| T | T | T | T |
| ? | T | ? | ? |
| F | T | ? | F |

| AND | T | ? | F |
|---|---|---|---|
| T | T | ? | F |
| ? | ? | ? | F |
| F | F | F | F |

在左边的临界值代表第一个操作对象，最上面一行中的值代表第二个操作对象。每一 3x3 矩阵内的值展示将该操作符应用于那些值的操作对象之后所返回的值。

如果布尔表达式球值为 UNKNOWN。则不满足该条件。

请考虑下列 WHERE 子句内的示例：

```
WHERE ship_charge/ship_weight < 5 AND order_num = 1023
```

**order_num** = 1023 的行是 **ship_weight** 为 NULL 的行。因为 **ship_weight** 为 NULL，所以 **ship_charge/ship_weight** 也是 NULL；因此， **ship_charge/ship_weight** < 5 的真值是 UNKNOWN。

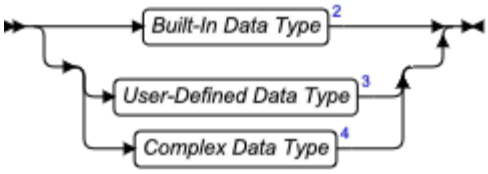
因为 **order_num** = 1023 为 TRUE，因此，AND 表声明整个条件的真值为 UNKNOWN。因此，不选择那行。如果在 AND 的位置使用 OR 作为条件，则该条件会是 TRUE。

## 4.6 数据类型

“数据类型”段指定列的、集合的组件的、ROW 类型内的字段的、例程参数的、表达式的返回值的或强制转型函数的返回值的数据类型。无论您何时看到对语法图中数据类型的引用时，请使用此段。

**语法**

数据类型




**用法**

下面的部分总结这些数据类型。要获取更多信息，请参阅 *《GBase 8s SQL 指南：参考》* 中关于数据类型的章节。

### 内建的数据类型

内建的数据类型是由数据库服务器定义的数据类型。

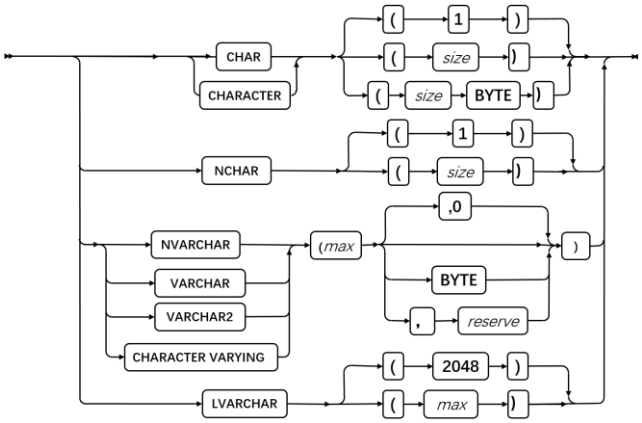
内建的数据类型


在解释和转换所需的信息和支持函数的意义上讲，这些是“构建在数据库服务器之内”，这些数据类型是数据库服务器软件的一部分，支持 ***character***、***numeric***、***large-object*** 和 ***time*** 内建数据类型的类别。在后面的部分中描述这些。

### 字符数据类型

字符数据类型使得数据库服务器能够存储文本字符串。

字符数据类型




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *max* | 以字节计的最大大小。对于 VARCHAR 和 NVARCHAR，这是必需的。LVARCHAR 缺省为 2048 | VARCHAR，VARCHAR2 和 NVARCHAR：<br>整数；1 ≤ *max* ≤ 32,765<br>LVARCHAR：1 ≤ *max* ≤ 32,739 | 精确数值 |
| *reserve* | 保留的字节。缺省为 0。 | 整数；0 ≤ *reserve* ≤ *max* | 精确数值 |
| *size* | 以字节计的大小。缺省为 1。 | 整数；1 ≤ *size* ≤ 32,767 | 精确数值 |

如果数据类型声明包括空的圆括号，比如 LVARCHAR( )，则数据库服务器发出错误。要声明缺省的长度的 CHAR 或 LVARCHAR 数据类型，简单地省略任何 **(*size*)** 或 **(*max*)** 规范。 GBase 8s 的 CREATE TABLE 语句接受没有 **(*max*)** 也没有 **(*max*, *reserve*)** 规范的 VARCHAR 和 NVARCHAR 列声明，使用 (1,0) 作为该列的 **(*max*, *reserve*)** 缺省值。

下表总结内建的字符数据类型。

| 数据类型 | 描述 |
|---|---|
| CHAR | 存储固定长度(最多 32,767 字节)的单字节或多字节文本字符串；支持文本数据的次序中的代码集顺序。缺省的大小为 1 字节。 |
| CHARACTER | CHAR 的同义词 |
| CHARACTER VARYING | VARCHAR 的符合 ANSI 的同义词 |
| LVARCHAR | 存储可变长度(最多 32,739 字节)的单字节或多字节文本字符串。在同一表中其他列的大小可进一步降低此上限。缺省的大小为 2,048 字节。 |
| NCHAR | 存储固定长度(最多 32,767 字节)的单字节或多字节文本字符串；支持文本数据的本地化次序。 |
| NVARCHAR | 存储可变长度(最多 32,765 字节)的单字节或多字节文本字符串；支持文本数据的本地化次序。 |

| 数据类型 | 描述 |
| --- | --- |
| VARCHAR/VARCAHR2 | 存储可变长度（最多 32765 字节）的单字节或多字节文本字符串；支持文本数据的代码集顺序次序。 |

## *单字节和多字节字符和语言环境*

所有内建的字符数据类型可支持 **DB_LOCALE** 设置指定的字符集中的单字节和多字节字符。大多数欧洲和中东语言的语言环境仅支持单字节代码集，但 Unicode 语言环境的 **UTF-8** 代码集以及一些东亚语言环境的代码（比如中文 **GB18030-2000** 语言环境）支持多字节逻辑字符。

当启用 SQL_LOGICAL_CHAR 配置参数时，您可指导数据库服务器将内建的字符数据类型的声明中显式的或缺省的大小参数解释为指定可存储的逻辑字符的数目，而不是字节数。这些逻辑的字符语义也适用于其基础类型为内建的字符类型的 DISTINCT 类型，并适用于命名的或未命名的 ROW 数据类型的声明中的内建的字符类型的字段。然而，此特性不支持存储字符字符串的用户定义的数据类型（UDT）。要获取更多关于此特性的信息，请参阅 *GBase 8s 管理员参考手册* 对 SQL_LOGICAL_CHAR 配置参数的描述。

TEXT 和 CLOB 数据类型也支持单字节或多字节字符数据，但大多数操作字符串的内建的函数不支持 TEXT 也不支持 CLOB 数据。要获取更多信息，请参阅 大对象数据类型。

## *固定长度或可变长度字符数据类型*

数据库服务器支持固定长度和可变长度字符数据的存储。***固定长度***列需要定义的字节数而不管实际的数据大小。CHAR 数据类型属于固定长度。例如，CHAR(25) 列对于所有值需要 25 字节的存储，因此字符串 `"This is a text string"` 使用 25 字节的存储。

***可变长度***列大小可为它的数据占据的字节数。NVARCHAR、VARCHAR、VARCHAR2 数据类型是可变长度字符数据类型。例如，VARCHAR(25) 列为列值最多保留 25 字节的存储，但字符串 `"This is a text string"` 仅使用保留的 25 字节中的 21 字节。VARCHAR 数据类型最多可存储 32765 字节数据。要获取关于 **IFX_PAD_VARCHAR** 环境变量的信息，其设置控制数据库服务器发送和接收 VARCHAR 和 NVARCHAR 数据值的方式，请参阅 *《GBase 8s SQL 指南：参考》*。

**访问有可变长度列的大型表**

对于带有多于一百万行的表，如果查询执行轻扫描，而不是缓冲池扫描，则使用全表扫描或跳跃扫描访问方法的查询更加高效。然而，在包括 NVARCHAR、VARCHAR 或 LVARCHAR 数据类型列，或其基础类型为***可变长度***列的 DISTINCT 数据类型的列的表上，不支持轻扫描，除非将 BATCHEDREAD_TABLE 配置参数（或 BATCHEDREAD_TABLE 会话环境选项）设置为 `1`。

**限制：**

在正启用 BATCHEDREAD_TABLE 上轻扫描的依赖还适用于其模式或存储属性包括下列任何情况的表：

- 表压缩
- 任何可变长度数据类型的列

- 占据超过单个存储页的行。

要获取更多关于查询优化器何时选择执行轻扫描的执行路径来访问大型表的信息，请参阅您的 *GBase 8s 性能指南*。

## *LVARCHAR 数据类型*

GBase 8s 的 LVARCHAR 类型可存储最多 32,739 字节的文本，但如果您在 LVARCHAR 数据类型声明中未指定 ***size***，则缺省的长度为 2,048 字节。LVARCHAR 是内建的 opaque 数据类型。不像大多数内建的 opaque 类型那样，可在分布式查询或其他 DML 操作中的非本地的 GBase 8s 实例的数据库中访问 LVARCHAR 列值，且 LVARCHAR 可为访问本地数据库之外的数据的 UDR 的参数或返回值的数据类型。

GBase 8s 在对 opaque 数据类型的跨服务器 I/O 操作中使用 LVARCHAR 数据类型。在此上下文中，LVARCHAR 数据值的最大大小仅受操作系统的限制。

在包括 LVARCHAR 列的表上不支持查询执行期间的轻扫描，除非将 BATCHEDREAD_TABLE 配置参数（或 BATCHEDREAD_TABLE 会话环境选项）设置为 1。

## *NCHAR 和 NVARCHAR 数据类型*

字符数据类型 NCHAR 和 NVARCHAR 可支持在某些数据库语言环境中次序的本地化顺序。在以 NLSCASE INSENSITIVE 属性创建的数据库中，NCHAR 和 NVARCHAR 列（以及强制转型到这些数据类型的字符串值）可支持区分大小写的查询。

字符数据类型 CHAR、LVARCHAR 和 VARCHAR 支持数据的***代码集顺序***次序。这是在 **DB_LOCALE** 环境变量指定的数据库语言环境的代码集内，在其中定义字符的顺序。缺省的（U.S. English）语言环境是为了排序 CHAR、LVARCHAR 和 VARCHAR 字符串值而使用次序的代码集顺序的语言环境的一个示例。

要获取关于 **DB_LOCALE**、**CLIENT_LOCALE** 和 **SERVER_LOCALE** 环境变量的设置（或缺省值）确定为次序使用哪一语言环境的方式的信息，请参阅 *GBase 8s GLS 用户指南*。

然而，有些语言环境指定不同于代码集顺序的一个次序的顺序。要支持任何次序的特定于语言环境的顺序，您可使用 NCHAR 和 NVARCHAR 数据类型。NCHAR 数据类型是支持本地化次序的固定长度字符数据类型。NVARCHAR 数据类型是可存储最多 32,765 字节的文本数据的可变长度字符数据类型，且支持本地化次序。在代码集未定义次序的本地化顺序的语言环境中，比如缺省的语言环境，在 CHAR 与 NCHAR 数据类型之间没有差异，在 VARCHAR 与 NVARCHAR 数据类型之间也没有差异，除了在区分大小写的数据库中之外。

在以 NLSCASE INSENSITIVE 属性创建的数据库中，存储这些数据类型的值恰如将它们加载到数据库内，但在数据处理操作中，包括 NVARCHAR 和 NCHAR 字符串的比较和对照，数据库服务器不理会字母大小写，不考虑大小写对数据值排序。例如，在有次序的列表中，NCHAR 或 NVARCHAR 字符串 "PH" 可能在 "pH" 或 "ph" 之前也可能在之后，在其中将这三个字符串视作重复，依赖于检索这些值的顺序。要获取更多关于在区分大小写的数据库中 NCHAR 或 NVARCHAR 数据处

理的信息，请参阅 指定 NLSCASE 区分大小写、在 NLSCASE INSENSITIVE 数据库中重复的行 和在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

对于 NCHAR 或 NVARCHAR 值，SQL 的 SET COLLATION 语句可通过指定另一语言环境来覆盖当前会话的本地化的次序顺序。根据当创建索引时生效了的本地化的次序顺序，对 NCHAR 或 NVARCHAR 列上的索引值排序，如果其不同于当前的次序顺序的话。要获取更多关于 SET COLLATION 语句可影响索引、约束、游标、准备好的对象和 SPL 例程的排序行为的方式的信息，请参阅 由数据库对象执行的对照。

如果您在声明 VARCHAR 或 NVARCHAR 列的 CREATE TABLE 或 ALTER TABLE 语句中未指定参数，则新列的缺省的 ***max*** 大小为 1 字节，且 ***reserve*** 大小为零。

**在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式**

在以 NLSCASE INSENSITIVE 属性创建的数据库中，数据库服务器在 NCHAR 和 NVARCHAR 表达式中相同字母的大写变量与小写变量之间不做区分，不论是否为语言环境定义了本地化的次序顺序。

如果在关系运算符的运算对象之中，或在字符串函数的参数之中，字母大小写变量是唯一的区别，则与在区分大小写的数据库中同一表达式上的同一操作相对比，对字母大小的这种忽略可更改 NCHAR 或 NVARCHAR 表达式上区分大小写的操作返回的值。

例如，假设对于缺省的语言环境中的数据库的表中的记录，NCHAR 列 **lname** 存储值 `McDavid`。

在区分大小写的数据库中，布尔表达式 `lname > "MCDAVID"` 求值为真，因为数据库服务器使用缺省的语言环境的代码集顺序来比较这两个运算对象。虽然两个字符串都是以大写字母 `M` 开头，但该列值中的下一字符是小写字母 `c`（ASCII 99 代码点），但括起来的字符串中的下一字符是大写字母 `C`（ ASCII 67 代码点）。由于 99 大于 67，因此，在区分大小写的数据库中，列值大于括起来的字符串。

然而，在不区分大小写的数据库中，同一表达式 `lname > "MCDAVID"` 求值为假，因为在数据库服务器比较这两个运算对象时，它不管字母大小写变量。两个字符串都有同样序列的同样字符，因此按照这些标准，列值与括起来的字符串是一样的。

在包括 NCHAR 或 NVARCHAR 运算对象的比较过程中，由于有 NLSCASE INSENSITIVE 属性的数据库不管字母的大小写， 因此，在不区分大小写的数据库中，在 NCHAR 或 NVARCHAR 字符串上的操作产生的结果可不同于在区分大小写的数据库中的结果。 区分大小写的数据库在其中的上下文与不区分大小写的数据库在其中的上下文可能使用相同的 SQL 操作来从相同的数据集返回不同的结果，包括这些：

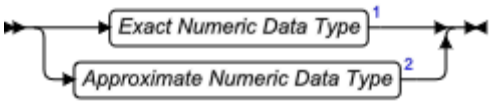
- 排序和次序
- 外键和主键依赖
- 强制唯一的约束
- 集群化的索引
- 访问方式优化器伪指令
- 带有 WHERE 谓词的查询

- 在 projection 子句中带有 UNIQUE 或 DISTINCT 的查询
- 带有 ORDER BY 子句的查询
- 带有 GROUP BY 子句的查询
- 级联的 DELETE 操作
- 表或索引存储分发 BY EXPRESSION
- 表或索引存储分发 BY LIST
- 来自 UPDATE STATISTICS 操作的数据分发。

## 数值数据类型

数值数据类型使得数据库服务器能够在列中存储诸如整数和实数这样的数。

数值数据类型

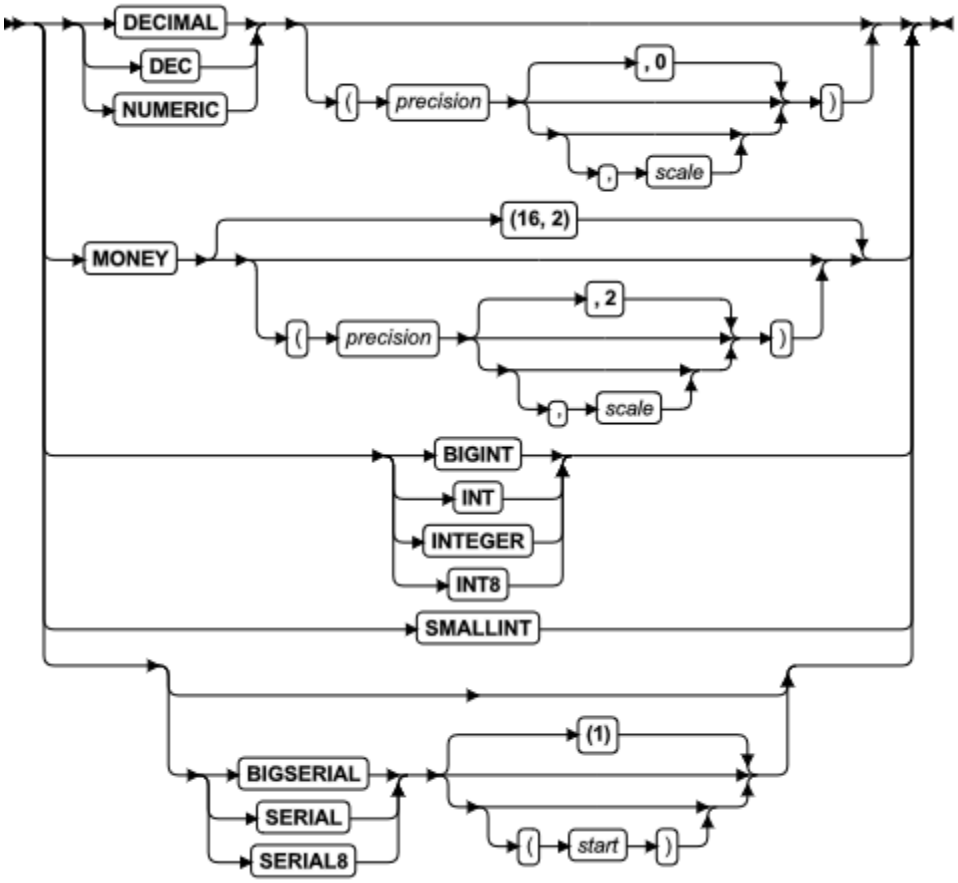



数目的值被存储为 ***exact numeric*** 数据类型或为 ***approximate numeric*** 数据类型。

### *精确的数值数据类型*

***精确的数值***数据类型存储指定的精度和范围的数值。

精确的数值数据类型




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *precision* | 有效数字 | 必须为整数；1 ⩽ *precision* ⩽ 32 | 精确数值 |
| *scale* | 小数部分中的数字 | 必须为整数；1 ⩽ *scale* ⩽ *precision* | 精确数值 |
| *start* | 整数起始值 | 对于 SERIAL：1 ⩽ *start* ⩽ 2,147,483,64；对于 BIGSERIAL 和 SERIAL8：1 ⩽ *start* ⩽ 9,223,372,036,854,775,807 | 精确数值 |

数据类型的***精度***是该数据类型存储的数字的数目。***范围***是小数点分隔符右边的数字的数目。

下表总结可用的精确的数值数据类型。

| 数据类型 | 描述 |
|---|---|
| DEC(*p, s*) | DECIMAL(*p, s*) 的同义词 |
| DECIMAL(*p, s*) | 存储实数的定点小数值，在小数部分中最多 20 位有效数字，或在小数点的左边最多 32 位有效数字。 |
| INT | INTEGER 的同义词 |

| | |
|---|---|
| INTEGER | 存储 4 字节整数值。这些值的取值范围可从 -(231-1) 至 231-1（从 -2,147,483,647 至 2,147,483,647）。 |
| BIGINT 和 INT8 | 存储 8 字节整数值。这些值的取值范围可从 -(263-1) 至 263-1（从 -9,223,372,036,854,775,807 至 9,223,372,036,854,775,807）。 BIGINT 有比 INT8 更大的存储和处理优势。 |
| MONEY(*p, s*) | 存储定点货币值。这些值与定点 DECIMAL(p,s) 值有相同的内部数据格式。 |
| NUMERIC(*p, s*) | DECIMAL(*p, s*) 的符合 ANSI 的同义词 |
| SERIAL | 存储数据库服务器生成的 4 字节正整数。取值范围从 1 至 231-1（即，从 1 至 2,147,483,647）。 |
| BIGSERIAL 和 SERIAL8 | 存储数据库服务器生成的 8 字节正整数。取值范围从 1 至 263-1（即，从 1 至 9,223,372,036,854,775,807）。BIGSERIAL 比 SERIAL8 有更大的存储和处理优势。 |
| SMALLINT | 存储 2 字节的整数值。这些值的取值范围从 -(215-1) 至 215-1（即，从 -32,767 至 32,767）。 |

**DECIMAL(p,s) 数据类型**

参数 ***p*** 指定***精度***（数字的总数目），第二个参数（***s***）指定***范围***（小数部分中的数字的数目）。如果您仅提供一个参数，则符合 ANSI 的数据库将它解释为定长数值的精度，且缺省的范围为 0。如果您未指定参数，且该数据库符合 ANSI，则在缺省情况下精度为 16 且范围为 0。

如果数据库不符合 ANSI，且您指定的参数少于 2 个，则您声明浮点 DECIMAL，这不是精确的数值数据类型。（另请参阅 近似的数值数据类型 部分。）

DECIMAL(***p, s***) 值的内部存储方式是，第一个字节代表符号位，以及一个 excess-65 格式的 7 位指数。其他的字节表示尾数作为 base-100 数字。这表示如果 ***s*** 是奇数，则 DECIMAL(32,***s***) 数据类型仅在小数点的右边存储 ***s***-1 个小数数字。

**Serial 数据类型**

您可声明 SERIAL、BIGSERIAL 或 SERIAL8 数据类型的列。如果用户定义的例程需要变量、参数或返回的数据类型的全数值值，则请指定 INT、BIGINT 或 INT8 作为数据类型，而不是 SERIAL、BIGSERIAL 或 SERIAL8。这些数据类型是整数数据类型，其主要的不同之处在于它们的名称、它们的范围和它们的存储需求。序列数据类型的列不可存储小于 1 的值。表可只有一个 SERIAL 列且只有一个 BIGSERIAL 或 SERIAL8 列。因为通过数据库服务器指定该序列值，因此您不可使用 UPDATE 语句来更改数据库中现有的序列值。

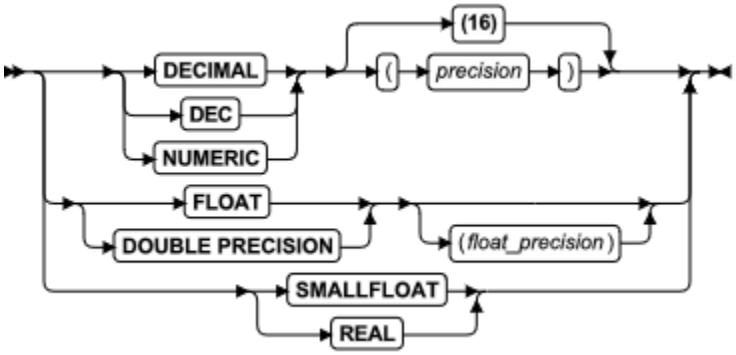
要将显式的值插入到 SERIAL、BIGSERIAL 或 SERIAL8 列内，请指定任何大于零的整数。要获取生成整数值的替代方法的详细信息，请参阅 CREATE SEQUENCE 语句。

仅当您在列上设置唯一索引，SERIAL、BIGSERIAL 或 SERIAL8 列才是唯一的。（该索引还可为主键或唯一约束的形式。）带有唯一索引，序列数据类型列中的值保证是唯一的，但随后的值没有必要是连续的。

## *近似的数值数据类型*

***近似的数值***数据类型近似地表示数值值。

近似的数值数据类型




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *float_precision* | 忽略 *float_precision*，但这符合 ANSI/ISO。 | 必须为正整数。指定的值没有作用。 | 精确数值 |
| *precision* | 有效数字。缺省值为 16。 | 整数；1 ≤ *precision* ≤ 32 | 精确数值 |

对于在算术操作期间容许一定程度舍入的很大和很小的数值，请使用近似的数值数据类型。

下表总结内建的近似的数值数据类型。

| 数据类型 | 描述 |
|---|---|
| DEC(***p***) | DECIMAL(p) 的同义词 |
| DECIMAL(***p***) | 存储从 1.0E-130 至 9.99E+126 的近似的范围内的浮点小数值<br>参数 ***p*** 指定精度。如果未指定精度，则缺省值为 16。仅在不符合 ANSI 的数据库中才可用浮点数据类型作为近似的数值类型。 在符合 ANSI 的数据库中，实现 DECIMAL(***p***) 作为定点 DECIMAL；请参阅 精确的数值数据类型。 |
| DOUBLE PRECISION | FLOAT 的 符合 ANSI 的同义词。当您在数据类型声明中使用此同义词时，*float_precision* 术语无效。 |

| | |
|---|---|
| FLOAT | 存储最多带有 16 位有效数字的双精度浮点数值。为了符合 SQL 的 ANSI/ISO 标准，在数据类型声明中接受 *float-precision* 参数，但此参数对数据库服务器存储的值的实际精度不起作用。 |
| NUMERIC(*p*) | DECIMAL(*p*) 的符合 ANSI 的同义词。在符合 ANSI 的数据库中，这是作为精确的数值类型来实现的，带有指定的精度和范围零，而不是近似的数值（浮点）数据类型。 |
| REAL | SMALLFLOAT 的符合 ANSI 的同义词 |
| SMALLFLOAT | 存储近似地带有 8 位有效数字的单精度浮点数值 |

GBase 8s 数据库服务器的内建的数值数据类型支持实数。它们不直接地存储虚数或复数。

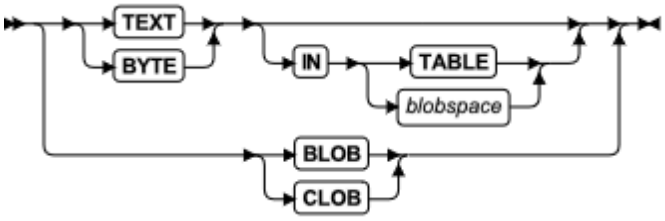
在 GBase 8s 中，您必须为支持可有虚数部分的值的应用创建用户定义的数据类型。

外部 UDR 的不超过九个参数可为 UDR 声明作为 Java™ 语言的 BigDecimal 数据类型的 SQL 的 DECIMAL 数据类型。

## 大对象数据类型

大对象数据类型可独立于列存储极大的列值，诸如图片和文档。

大对象数据类型




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *blobspace* | 现有的 blobspace 的名称 | 必须存在 | 标识符 |

大对象数据类型可划分为两类：

- 简单大对象：TEXT 和 BYTE
- 智能大对象：CLOB 和 BLOB

### *简单大对象数据类型*

简单大对象数据类型在 blobspace 中存储文本或二进制数据。

这些是简单大对象数据类型：

TEXT　存储最大 $2^{31}$ 字节的文本数据

BYTE　存储最大 $2^{31}$ 字节的任何数字化数据

在需要 TEXT 的地方，请不要提供 BYTE 值。没有内建的强制转换支持由 BYTE 到 TEXT 的数据类型转换。

要获取更多关于简单大对象数据类型的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

要获取关于如何创建 blobspace 的信息，请参阅 *GBase 8s 管理员指南*。

## *存储 BYTE 和 TEXT 数据*

简单大对象数据类型可在 blobspace 或在表中存储文本或二进制数据。数据库服务器可完整地访问 BYTE 或 TEXT 值。当您指定 BYTE 或 TEXT 数据类型时，您可指定存储它的位置。您可以表存储数据，或在单独的 blobspace 中存储数据。

如果您正在创建有 BYTE 或 TEXT 字段的命名的 ROW 数据类型，则您不可使用 IN 子句来指定单独的存储空间。

下列示例展示如何指定 blobspace 和 dbspace。用户创建 resume 表。数据值存储在 employ dbspace 中。以表存储 vita 列中的数据，但与 photo 列相关联的数据存储在名为 photo_space 的 blobspace 中。

```
CREATE TABLE resume
    (
    fname           CHAR(15),
    lname           CHAR(15),
    phone           CHAR(18),
    recd_date       DATETIME YEAR TO HOUR,
    contact_date    DATETIME YEAR TO HOUR,
    comments        VARCHAR(250, 100),
    vita            TEXT IN TABLE,
    photo           BYTE IN photo_space
    )
    IN employ;
```

## *智能大对象数据类型*

智能大对象数据类型在 sbspace 中存储文本或二进制数据。

数据库服务器可提供对智能大对象值的随机访问。也就是说，它可访问智能大对象值的任何部分。这些数据类型是可恢复的。下列列表总结 GBase 8s 支持的智能大对象数据类型。

BLOB　　存储最多 4 TB（$4*2^{40}$ 字节）的二进制数据

CLOB　　存储最多 4 TB（$4*2^{40}$ 字节）的文本数据

在单个 sbspace 中存储智能大对象。SBSPACENAME 配置参数指定在其中创建智能大对象的系统缺省的 sbspace，除非您指定另一存储区域。要获取关于 CREATE TABLE 语句如何可为 BLOB 或 CLOB 列指定非缺省的存储位置和非缺省的存储特征的信息，请参阅 PUT 子句 的描述。

这两个都是内建的 opaque 数据类型。像大多数 opaque 类型那样，不可通过分布式查询或通过其他 DML 操作在非本地数据库服务器的数据库中访问它们，也不可通过 UDR 从另一数据库服务器的数据库返回它们。然而，要获取关于在本地服务器的其他数据库中访问 BLOB 或 CLOB 值的信息，请参阅 BOOLEAN 和其他内建的 Opaque 数据类型。

智能大对象数据类型是不能并行的。Dynamic Serve 的 PDQ 特性对加载或卸载 BLOB 或 CLOB 值的操作不起作用，对在查询或在其他 DML 操作中处理它们的操作也不起作用。

要获取更多关于智能大对象数据类型的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

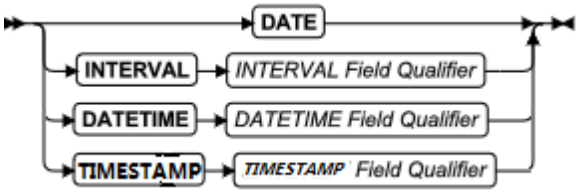
要获取关于如何创建 sbspace 的信息，请参阅您的 *GBase 8s 管理员指南*。

要获取关于您可用于导入、导出或复制智能大对象的内建的函数的信息，请参阅 智能大对象函数 和 *GBase 8s SQL 教程指南*。

## 时间数据类型

时间数据类型存储日历日期、时间点以及时间间隔。

时间数据类型




下表总结内建的时间数据类型。

| 数据类型 | 描述 |
|---|---|
| DATE | 存储日期值作为 Julian 日期，取值范围从公元 1 年 1 月 1 日直至 公元 ***9999*** 年 12 月 31 日。 |
| DATETIME | 存储时间点日期（*年*、*月*、*日*）和每日时间（*小时*、*分*、*秒*和*几分之一*秒），取值范围从 1 年至 9999 年。也支持这些时间单位的相邻子集。 |
| INTERVAL | 存储时间范围，以*年数*和/或*月数*的形式，或以更小的时间单位的形式（*天数*、*小时数*、*分钟数*、*秒数* 和/或*几分之一*秒），最大的时间单位达到 9 位数值精度，如果这不是 FRACTION 的话。还支持这些时间单位的相邻子集。 |
| TIMESTAMP | 存储时间点日期（*年*、*月*、*日*）和每日时间（*小时*、*分*、*秒*和*几分之一*秒）。 |

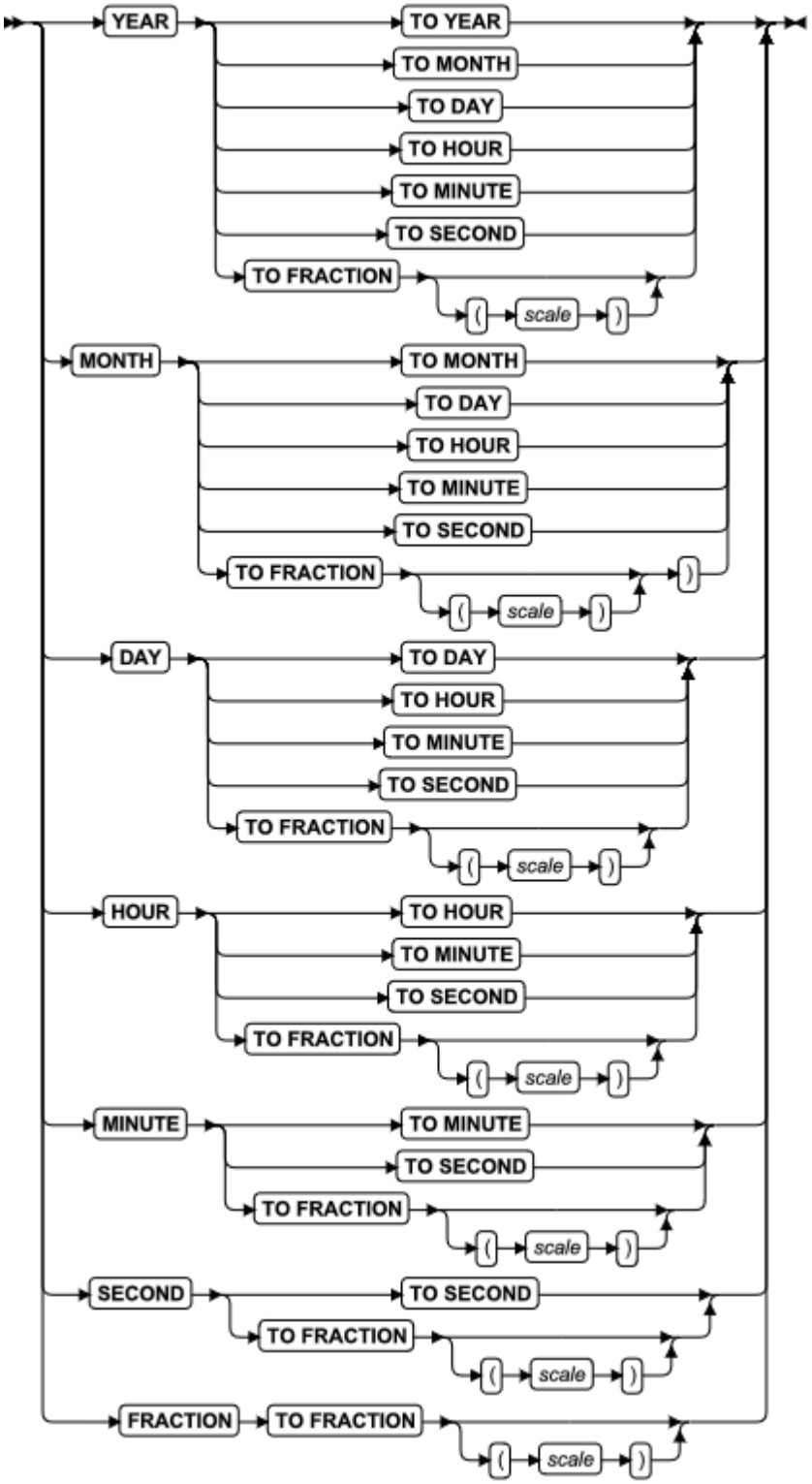
要了解可指定内建的时间数据类型的显示和数据条目格式的 GBase 8s 环境变量之中的优先顺序，请参阅主题 DATE 和 DATETIME 格式规范的优先顺序。

### *DATETIME 字段限定符*

使用 DATETIME Field Qualifier 来指定 DATETIME 列或值中的最大和最小时间单位。无论何时在语法图中查看对 DATETIME Field Qualifier 的引用时，请使用此段。

## 语法

DATETIME Field Qualifier




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *scale* | 几分之一秒。缺省值为 3。 | 整数（1 ≤ *scale* ≤ 5） | 精确数值 |

**用法**

此段指定 DATETIME 数据类型的精度和小数位。

作为第一个关键字，指定 DATETIME 列将存储的最大时间单位。在关键字 TO 之后，指定最小的单位作为最后一个关键字。它们可以是同一个关键字。如果不同，那么限定符暗示在第一个和最后一个之间的中间时间单位通过 DATETIME 数据类型记录。

这些关键字可为 DATETIME 列指定下列时间单位。

| 时间的单位 | 描述 |
| --- | --- |
| YEAR | 指定年份，取值范围从公元 1 年至 9999 年 |
| MONTH | 指定月份，范围为从 1（一月）值 12（十二月） |
| DAY | 指定日期，范围为从 1 至 28、29、30 或 31（根据特定月份） |
| HOUR | 指定小时，范围为从 0（午夜）至 23 |
| MINUTE | 指定分钟，范围为从 0 至 59 |
| SECOND | 指定秒，范围为从 0 至 59 |
| FRACTION | 指定几分之一秒，最多五位小数 |

缺省的范围为三位数字（千分之一秒）。

与 INTERVAL 限定符不同，DATETIME 限定符不可指定非缺省的精度（当 FRACTION 是该限定符中最小的单位时，FRACTION 除外）。DATETIME 限定符的一些示例如下：

YEAR TO MINUTE　　MONTH TO MONTH

DAY TO FRACTION(4)　　MONTH TO DAY

在某些平台上，系统时钟不可支持大于 FRACTION(3) 的精度。

如果第一个关键字表示的时间单位比最后的一个关键字所表示的小，或使用了关键字的复数形式（如 MINUTES），就会产生错误。

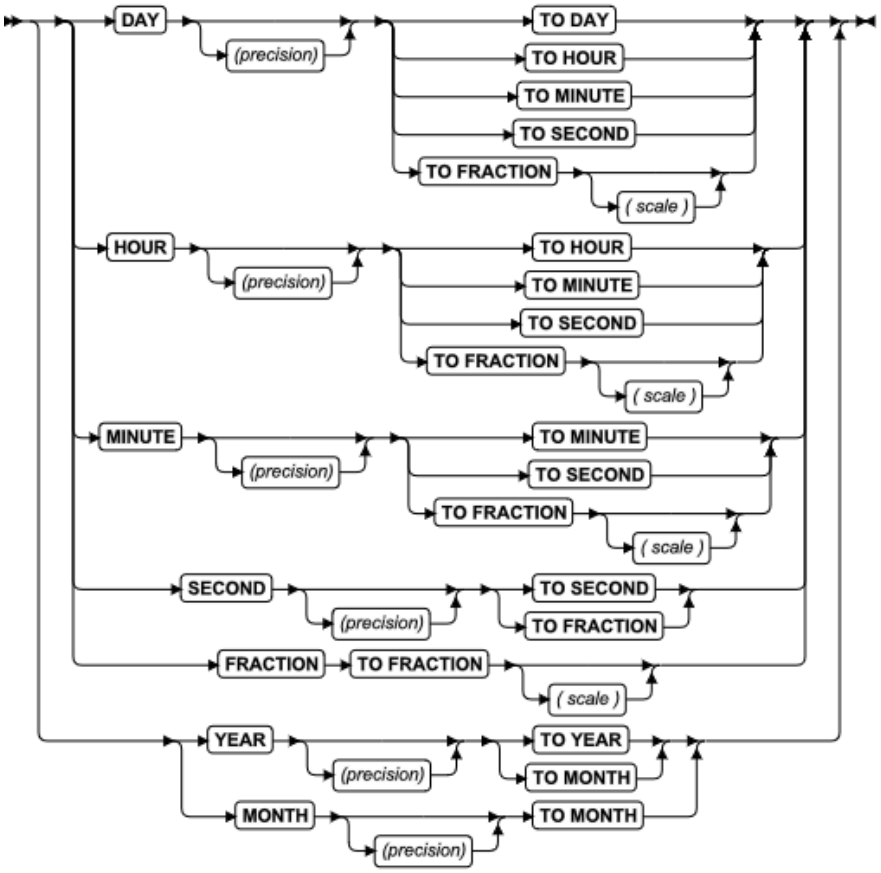
对在限定符中不包含 YEAR 的 DATETIME 的值的运算，使用系统时钟日历值来提供所有额外的精度。如果限定符的第一个术语是 DAY，且当前月份少于 31 天，则可能产生意外的结果。

## *INTERVAL 字段限定符*

INTERVAL 字段限定符以时间单位指定 INTERVAL 值的精度。每当您在语法图中看到对 INTERVAL 字段的引用时，请使用 INTERVAL Field Qualifier 段。

**语法**

INTERVAL 字段限定符




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *scale* | FRACTION 字段中数字的个数。缺省值为 3。 | 取值范围必须为从 1 至 5 | 精确数值 |
| *precision* | INTERVAL 包含的最大时间单位中数字的个数。对于 YEAR，缺省值为 4。对于除 FRACTION 外的其他所有时间单位，缺省值为 2。 | 取值范围必须为从 1 至 9 | 精确数值 |

## 用法

此段指定 INTERVAL 数据类型的精度和小数位。

指定 *largest* 时间单位的关键字必须是第一个关键字，而指定 *smallest* 时间单位的关键字必须跟在 TO 关键字之后。它们可以是同一个关键字。此段与 DATETIME 字段限定符 的语法类似，但有这些例外：

- 如果最大时间单位关键字是 YEAR 或 MONTH，那么最小时间单位关键字不能指定小于 MONTH 的时间单位。
- 可在第一个时间单位后面指定最大为 9 位数字的 ***precision***，除非 FRACTION 是第一个时间单位（在此情况下在第一个 FRACTION 关键字之后，没有任何 ***precision*** 是有效的，但可在第二个 FRACTION 关键字之后指定多达 5 位数字的 ***scale***）。

由于 ***year*** 和 ***month*** 不是定长的时间单位，数据库服务器将在限定符中包含 YEAR 或 MONTH 关键字的 INTERVAL 数据类型与限定符小于 MONTH 的时间单位的 INTERVAL 数据类型不兼容。数据库服务器不支持在这两种 INTERVAL 数据类型之间的隐式强制转型。

以下两个示例显示 INTERVAL 数据类型的 YEAR TO MONTH 限定符。第一个示例可以允许最大为 999 年 11 个月的时间间隔，因为它给出了 3 作为 YEAR 字段的精度。第二个示例对 YEAR 字段使用缺省精度，因此可允许最大为 9999 年和 11 个月的时间间隔。

```
YEAR (3) TO MONTH
YEAR TO MONTH
```

当需要一个值只指定一种时间单位时，第一个和最后一个限定符是相同的。例如，整年之间的时间间隔可以限定为 YEAR TO YEAR 或 YEAR (5) TO YEAR。用来指定最大 9999 年的时间间隔。

下列示例显示 INTERVAL 字段限定符的几种格式：

```
YEAR(5) TO MONTH
DAY (5) TO FRACTION(2)
DAY TO DAY
FRACTION TO FRACTION (4)
```

要获取关于如何指定 INTERVAL 字段限定符以及如何在算术和关系运算中使用 INTERVAL 数据的信息，请参阅相关的参考，INTERVAL 数据类型。

### *TIMESTAMP 字段限定符*

此段指定 TIMESTAMP 数据类型的精度和范围。

TIMESTAMP Field Qualifier 与 DATETIME Field Qualifier 的功能类似。唯一不同的是，其中指定几分之一秒的 FRACTION 时间单位值可以是 1 到 6 范围内的数字，缺省值为 0。

## XMLTYPE 数据类型

GBase 8s 支持创建 xmltype 数据类型用于操作 xml 结构数据。建表时支持创建 xmltype 类型，插入数据为 xml 格式。

## BOOLEAN 和其他内建的 Opaque 数据类型

GBase 8s 还支持 BOOLEAN 数据类型，这是可存储 `true`、`false` 或 NULL 值的***内建的 opaque 数据类型***。符号 `t` 表示字面的 BOOLEAN `true` 值，`f` 表示字面的 BOOLEAN `false` 值。

BOOLEAN 是 LVARCHAR 可通过跨服务器分布式查询或通过其他跨服务器分布式 DML 操作返回的唯一内建的 opaque 数据类型。不可通过分布式查询（在远程数据库上通过 INSERT、DELETE 或 UPDATE 操作不修改）检索其他内建的 opaque 数据类型的列值，除非该 DML 操作访问的所有表都在本地的 GBase 8s 实例的数据库中。

类似地，在其他 GBase 8s 实例的数据库上执行分布式操作的 UDR 中，BOOLEAN 和 LVARCHAR 是作为参数或作为返回的该 UDR 的数据类型唯一有效的内建的 opaque 类型，必须在所有参与的数据库中定义该类型。

除了 BOOLEAN 类型之外， GBase 8s 的其他内建的 opaque 数据类型包括 BLOB、CLOB、LVARCHAR、IFX_LO_SPEC、IFX_LO_STAT、INDEXKEYARRAY、POINTER、RTNPARAMTYPES、SELFUNCARGS、STAT、CLIENTBINVAL 和 XID 数据类型。在本地数据库中以及在同一服务器实例的跨数据库分布式操作中，支持这十二个内建的 opaque 类型。在本章的随后部分中讨论这些类型中的前三个。

GBase 8s 还支持内建的 opaque 数据类型 LOLIST、IMPEX、IMPEXBIN 和 SENDRECV。然而，不可通过 DML 操作在远程数据库中访问这些类型，也不通过 UDR 从远程数据库返回，因为这些数据类型没有所需要的支持函数。要获取更多的关于 GBase 8s 在分布式事务中支持的数据类型的信息，请参阅 分布式查询中的数据类型。

## IDSSECURITYLABEL 数据类型

GBase 8s 的 IDSSECURITYLABEL 类型在受安全策略保护的表中存储安全标签。仅持有 DBSECADM 角色的用户可创建、修改或删除此数据类型的列。这是内建的 DISTINCT OF VARCHAR(128) 数据类型，但未将它分类作为字符数据类型，因为将它的使用限定在基于标签的访问控制。有安全策略的表可有多个 IDSSECURITYLABEL 列，未与安全策略相关联的表可没有这样的列。

DBSECADM 可使用 GRANT 语句来将特定的安全标签与用户相关联，且 REVOKE 语句可取消用户持有的安全标签。对于给定的安全策略，用户可有多个既支持读访问也支持写访问的标签，或只有一个写访问标签和只有一个读访问标签。对于受安全策略保护，但已将自主访问权限授予了用户的数据，数据库服务器通过将数据的安全标签与用户的安全标签相对比来确定特定的用户可否访问该数据，同时还考虑该用户持有的安全策略规则是否存在任何豁免。

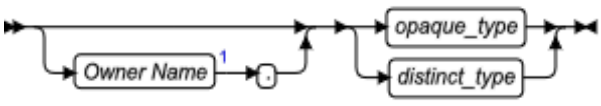
要获取指定 IDSSECURITYLABEL 值的方式的信息，请参阅 安全标签支持函数。

要获取安全策略、安全组件、安全标签以及基于标签的访问控制（LBAC）的其他概念的讨论，请参阅 GBase 8s 安全指南。

## 用户定义的数据类型

用户定义的数据类型是用户为数据库服务器定义的一种数据类型。GBase 8s 支持两类用户定义的数据类型，即 *distinct 数据类型*和 *opaque 数据类型*。这是用户定义的数据类型的声明语法：

用户定义的数据类型




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *distinct_type* | 带有与现有的数据类型有相同结构的 distinct 数 | 在数据库中的数据类型名称之中必须 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 据类型 | 是唯一的 | |
| *opaque_type* | opaque 数据类型的名称 | 在数据库中的数据类型名称之中必须是唯一的 | 标识符 |

在本文档中，***用户定义的数据类型***通常缩写为 UDT。

## distinct 数据类型

DISTINCT 数据类型是基于下列数据类型的用户定义的数据类型：

- 内建的类型（包括内建的 opaque 类型）
- 用户定义的 opaque 类型
- 命名的 ROW 类型
- 现有的 DISTINCT 类型。

DISTINCT 类型的基本类型不可为任何下列数据类型：

- 未命名的 ROW 类型
- LIST、MULTISET、SET 或通用的 COLLECTION 类型。

DISTINCT 类型继承存储中它的基本类型的长度和对齐方式。 GBase 8s 自动地在 DISTINCT 类型与它的基本类型之间创建显式的强制转型。要创建 DISTINCT 类型，您必须使用 CREATE DISTINCT TYPE 语句（要获取更多信息，请参阅 CREATE DISTINCT TYPE 语句。）

### *分布式操作中的 DISTINCT 类型*

不可通过分布式查询从同一 GBase 8s 实例的另一数据库检索 DISTINCT 列值（也不可通过 INSERT、DELETE、MERGE 或 UPDATE 跨数据库分布式操作来修改），除非所有下列条件都为真：

- 在下列基本类型之一上定义该 DISTINCT 类型：
  - 非 opaque 内建的数据类型
  - BOOLEAN 或 LVARCHAR 数据类型
  - 在 BOOLEAN 上、在 LVARCHAR 上，或在非 opaque 内建的数据类型上创建的 DISTINCT 类型。

  （此条件也递归地适用于 DISTINCT 类型的 DISTINCT 类型，在此，最终的基础类型为 BOOLEAN，或 LVARCHAR，或非 opaque 内建的数据类型。）
- 显式地强制转型为 BOOLEAN、LVARCHAR 或非 opaque 内建的数据类型的 DISTINCT 类型
- DISTINCT 类型，在所有参与的数据库中都正好以同一种方式定义它的层级和它的向内建的类型的显式的强制转型。

对于在分布式操作中的 DISTINCT 数据类型，数据类型层级必须有这些形式中的一种，不随所在的参与的数据库的不同而变化：

**重要：**

上图展示任何 DISTINCT 数据类型的基础类型的一般性逻辑层级。然而，如在上图中那样递归地使用 DISTINCT OF 关键字是无效的 SQL 语法。CREATE DISTINCT TYPE 语句必须为新的 DISTINCT 类型正好指定一个基础类型。要创建 DISTINCT 数据类型的层级，您必须为层级中的每个 DISTINCT 类型发出一个单独的 CREATE DISTINCT TYPE 语句。要了解定义新的 DISTINCT 数据类型的 SQL 语法，请参阅主题 CREATE DISTINCT TYPE 语句。

在受保护的表的行中存储安全标签的 IDSSECURITYLABEL 数据类型是满足此要求的内建的 DISTINCT 类型，因为它的基础类型是内建的 VARCHAR(128) 数据类型。

用户定义的例程可从同一 GBase 8s 实例的另一数据库将 DISTINCT 数据类型返回给本地数据库，仅当所有上述条件都为真，且在所有参与的数据库中定义 UDR 的话。

适用于同一 GBase 8s 实例的跨数据库的分布式操作中的 DISTINCT 数据类型的那些规则，也适用于在不同的 GBase 8s 实例的数据库上的跨服务器分布式操作中的 DISTINCT 数据类型。

要获取关于 GBase 8s 在分布式操作中支持的数据类型的附加信息，请参阅 分布式查询中的数据类型。

## Opaque 数据类型

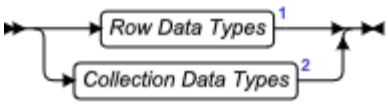
opaque 数据类型是可以与内建的数据类型相同的方式使用的用户定义的数据类型。要创建 opaque 类型，您必须使用 CREATE OPAQUE TYPE 语句。由于 opaque 类型是被封装的，因此您要创建支持函数来访问 opaque 类型的单个组件。该类型的内部存储细节是隐藏的或不透明的。

要获取更多关于如何创建 opaque 数据类型以及它的支持函数的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## 复合的数据类型

***复合的数据类型***是您从内建的类型、opaque 类型、distinct 类型或其他复合的类型创建的 ROW 类型或 COLLECTION 类型。

复合的数据类型




单个复合的数据类型可包括多个组件。当您创建符合的类型时，您定义该复合的类型的组件。然而，不像 opaque 类型那样，不封装复合的类型。您可使用 SQL 来访问复合的数据类型的个别的组件。复合的数据类型的个别的组件称为***元素***。

GBase 8s 支持下列类别的复合的数据类型：

- ROW 数据类型：命名的 ROW 类型和未命名的 ROW 类型
- COLLECTION 数据类型：SET、MULTISET 和 LIST

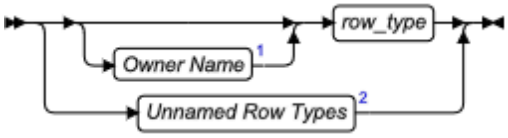
COLLECTION 数据类型的元素必须都是同一数据类型。您可使用 SPL 数据类型声明中的关键字 COLLECTION 来指定 untyped 集合变量。在 COLLECTION 数据类型的元素中不支持 NULL 值。

ROW 数据类型的元素可以是不同的数据类型，但对于给定的 ROW 数据类型，从第一个到最后一个元素，数据类型的模式不可变化。在 ROW 数据类型的元素中支持 NULL 值，除非您在数据类型声明中或在约束中另行指定。

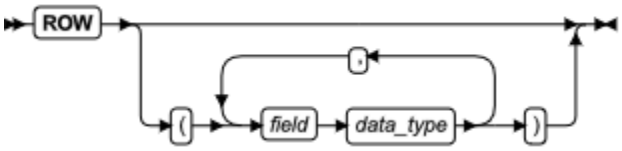
## ROW 数据类型

这是将一列定义为命名的或未命名的 ROW 类型的语法。

Row 数据类型




未命名 Row 类型




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | *field* 的数据类型 | 除了 BYTE 或 TEXT 之外的任何数据类型 | 数据类型 |
| *field* | *row_type* 内字段的名称 | 在同一 ROW 类型的字段之中必须是唯一的 | 标识符 |
| *row_type* | 通过 CREATE ROW TYPE 语句定义的某些 ROW 数据类型 | 在数据库中 ROW 类型必须存在 | 标识符；数据类型 |

您可将命名的 ROW 类型指定给表、给列或给 SPL 变量。您用来创建 typed 表或用来定义列的命名的 ROW 类型必须已存在。要获取关于如何创建命名的 ROW 数据类型的信息，请参阅 CREATE ROW TYPE 语句。

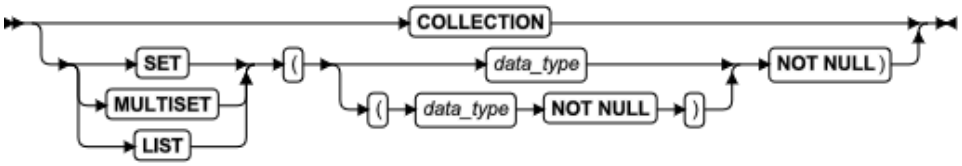
要在符合 ANSI 的数据库中指定命名的 ROW 数据类型，如果您不是 ***row_type*** 的所有者，则您必须以它的 ***owner*** 名称来限定 ***row_type***。

通过未命名的 ROW 数据类型结构来标识它，其指定您以它的 ROW 构造函数创建的字段。您可指定一列或一 SPL 变量作为未命名的 ROW 数据类型。要获取为未命名的 ROW 类型指定值的语法，请参阅 ROW 构造函数。

# 集合数据类型

此图展示定义一列或一 SPL 变量作为集合数据类型的语法。要了解指定集合元素的值的语法，请参阅 集合构造函数。

集合数据类型




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | 每一集合元素的数据类型 | 可为除了 BIGSERIAL、BYTE、SERIAL、SERIAL8 或 TEXT 之外的任何数据类型 | 数据类型 |

SET 是元素的无序的集合，每一元素都有唯一的值。当您想要存储其元素不包含重复的值且没有相关联的顺序的集合时，请将一列定义为 SET 数据类型。

MULTISET 是可有重复的值的元素的无序的集合。当您想要存储其元素可能不是唯一的且没有与它们相关联的特定顺序时，您可将一列定义为 MULTISET 集合类型。

LIST 是一个可包括重复的元素的元素的有序的集合。与 MULTISET 不同的是，LIST 集合中的每一元素都在该集合中有一有序的位置。当您想要存储其元素可能不是唯一的但有一与它们相关联的特定顺序的集合时，您可将一列定义为 LIST 集合类型。

可在 SPL 数据类型声明中使用关键字 COLLECTION，来指定一 untyped 集合变量。

如果您尝试将一个包括一个或多个重复的值的集合插入到 SET 列内，则 GBase 8s 不发出错误，但忽略重复的值，且只插入唯一的值。

## *SET 列上 DML 操作中的重复元素*

SET 数据类型不允许在同一集合中有重复的元素值。如果您尝试将重复的元素插入到 SET 数据类型内，或将 SET 列或变量更新为包括重复的元素的值，则当执行 INSERT 或 UPDATE 语句时，数据库服务器不发出错误或警告，但在 SET 列或变量中仅存储重复的元素之一。

例如，假如您以 SET 数据类型的列 a 创建表 t3，然后您插入四行，其中一些包括有相同的值的元素：

```
> CREATE TABLE t3(a SET(INT NOT NULL));

  Table created.

> INSERT INTO t3 VALUES( SET{10, 20, 30} );
```

```
1 row(s) inserted.

> INSERT INTO t3 VALUES( SET{10, 20, 10});

1 row(s) inserted.

> INSERT INTO t3 VALUES( SET{10, 10, 10});

1 row(s) inserted.

> INSERT INTO t3 VALUES( SET{10,10,10});

1 row(s) inserted.
```

当您查看插入到列 t3.a 内的那些数据值时，插入的四行不包括重复的元素值：

```
> SELECT * FROM t3;

a  SET{10        ,20        ,30        }
a  SET{10        ,20        }
a  SET{10        }
a  SET{10        }

4 row(s) retrieved.
```

在此示例中， GBase 8s 静默地从为每一 SET 值指定的 INSERT 语句的 VALUES 子句的元素中舍弃一个实例之外的所有重复的元素。

如果 UPDATE 语句的 SET 子句在同一 SET 值内包括重复的元素，则会发生类似的行为。如果您想要数据库存储可在同一集合内包括重复的元素的无序集，则请声明 MULTISET 数据类型的集合列，而不是 SET 数据类型的。

### *定义元素类型*

元素类型可为除了 TEXT、BYTE、SERIAL、SERIAL8 或 BIGSERIAL 之外的任何数据类型。您可使用集合类型的元素嵌套集合类型。

每个元素必须是同一类型的。例如，如果集合数据类型的元素类型是 INTEGER，则每个元素的类型必须是 INTEGER。

要获取附加的信息，请参阅 集合构造函数。

如果集合的元素类型是未命名的 ROW 类型，则未命名的 ROW 类型不可包含持有未命名的 ROW 类型的字段。也就是说，集合不可包含嵌套的未命名的 ROW 数据类型。

集合的元素不可为 NULL。当您定义一列为集合数据类型时，您必须使用 NOT NULL 关键字来指定该集合的元素不可为 NULL。

对集合数据类型的权限就是数据库列的那些权限。您不可在集合的个别元素上指定权限。

# 4.7 表达式

SQL 语句中的数据值必须表示为表达式。***表达式***是一种规范，其可包括运算符、运算对象和圆括号，数据库服务器可对其求得一个或多个值，或引用某个数据库对象。

表达式可引用数据库的表中已有的值，或从此数据派生的值，但有些表达式（诸如 TODAY、USER 或字面值）可返回独立于数据库的值。您可使用表达式来指定数据操纵语句中的值，定义分片策略和指定其他上下文中的值。只要当您看到语法图中对表达式的引用时，就可使用 Expression 段。

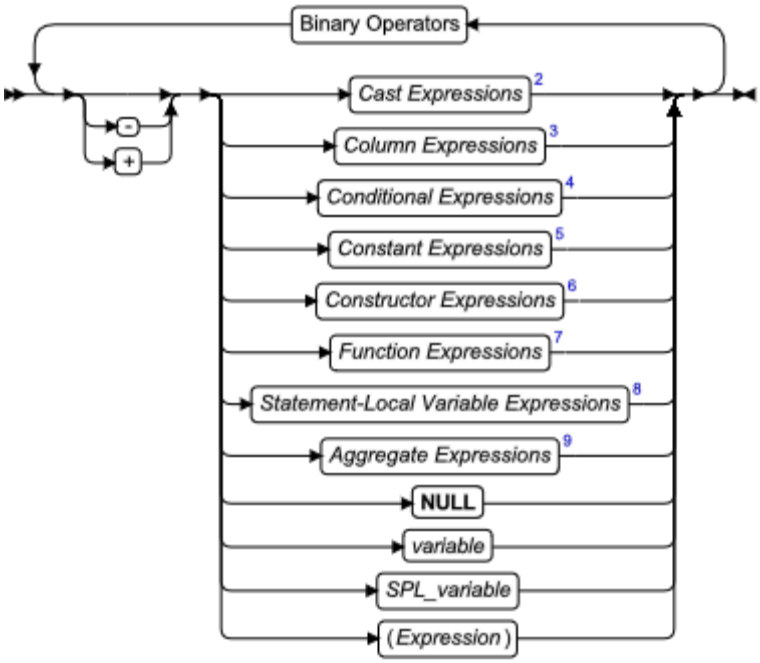
然而，在大多数上下文中，会限定您使用其返回值为某种特定数据类型的表达式，或其数据类型可通过数据库服务器转换为某些需要的数据类型。

要获取在本段中描述的内建的运算符和函数的按字母排序的列表，请参阅 表达式的列表。

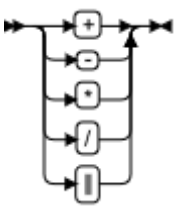
## SQL 表达式的语法

下面的部分描述 SQL 表达式，其为返回一个或多个值或引用数据库对象的规范。 GBase 8s 数据库服务器支持系列类别的表达式：

SQL 表达式




二进制运算符




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |

| | | | |
|---|---|---|---|
| *SPL_variable* | 在 SPL 例程中，包含语法图展示的某个表达式类型的变量 | 必须符合那个类型的表达式的规则 | 标识符 |
| *variable* | 包含语法图展示的某个表达式类型的主变量或程序变量 | 必须符合那个类型的表达式的规则 | 名称的特定于语言的规则 |

## 用法

下表罗列 SQL 表达式的类型，如同在 表达式 的图中标识的那样，且描述每一类型返回的内容。

| 表达式类型 | 描述 |
|---|---|
| 聚集函数 | 从内建的或从用户定义的聚集返回值 |
| 算术运算符 | 支持对一个（一元运算符）或两个（二元运算符）数值运算对象的算术操作 |
| 串联运算符 | 串联两个字符串值 |
| 强制转型运算符 | 从一种数据类型显式强制转型到另一种 |
| 列表达式 | 列值 |
| 条件表达式 | 返回依赖于条件测试的值 |
| 常量表达式 | 在数据操纵（DML）语句中的字面值 |
| 构造函数表达式 | 为复合的数据类型动态地创建值 |
| 函数表达式 | 从内建的或用户定义的函数返回值 |
| 语句-本地的变量表达式 | 在声明了它的 SQL 语句中引用语句-本地的变量（SLV） |

您还可使用主变量或 SPL 变量作为表达式。要获取带有对此章节的页引用的完整列表，请参阅下列"表达式的列表"。

## 表达式的列表

每一类 SQL 表达式都包括许多个别的表达式。

下表以字母顺序罗列所有 SQL 表达式（以及一些运算符）。此表中的列有下列含义：

- **名称**给出每一表达式的名称。
- **描述**给出每一表达式的简短描述。
- **语法**罗列展示该表达式的语法的页。
- **用法**展示描述该表达式的用法的页。

| 名称 | 描述 | 语法 | 用法 |
|---|---|---|---|

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| ABS 函数 | 返回数值参数的绝对值 | 代数函数 | ABS 函数 |
| ACOS 函数 | 返回数值参数的反余弦 | 三角函数 | ACOS 函数 |
| ACOSH 函数 | 返回指定的数值输入的双曲正切 | 三角函数 | ACOSH 函数 |
| ADD_MONTHS 函数 | 添加指定的月数 | 时间函数 | ADD_MONTHS 函数 |
| 加法（+）运算符 | 返回两个数值运算对象的和 | 表达式 | 算术运算符 |
| ASCII 函数 | 返回在它的字符串参数中第一个字符的 ASCII 代码点 | 字符串操纵函数 | ASCII 函数 |
| ASIN 函数 | 返回数值参数的反正弦 | 三角函数 | ASIN 函数 |
| ASINH 函数 | 返回指定的数值输入的双曲正弦 | 三角函数 | ASINH 函数 |
| ATAN 函数 | 返回数值参数的反正切 | 三角函数 | ATAN 函数 |
| ATAN2 函数 | 计算极坐标的角度分量 | 三角函数 | ATAN2 函数 |
| ATANH 函数 | 返回指定的数值输入的双曲反正切 | 三角函数 | ATANH 函数 |
| AVG 函数 | 返回一组数值的平均值 | 聚集表达式 | AVG 函数 |
| BITAND | 返回两个参数的位 AND | 位逻辑函数 | BITAND 函数 |
| BITANDNOT | 返回两个参数的位 ANDNOT | 位逻辑函数 | BITANDNOT 函数 |
| BITNOT | 返回两个参数的位 NOT | 位逻辑函数 | BITNOT 函数 |
| BITOR | 返回两个参数的位 OR | 位逻辑函数 | BITOR 函数 |
| BITXOR | 返回两个参数的位 XOR | 位逻辑函数 | BITXOR 函数 |
| CARDINALITY 函数 | 返回集合数据类型（SET、MULTISET 或 LIST）中元素的数目 | CARDINALITY 函数 | CARDINALITY 函数 |
| CASE 表达式 | 返回一个依赖于哪几个条件的测试求值为真的值 | CASE 表达式 | CASE 表达式 |
| CAST 表达式 | 将表达式转换为指定的数据类型 | 强制转型表达式 | 强制转型表达式 |
| 强制转型（::）运算符 | 请参阅“双冒号（::）强制转型运算符” | 强制转型表达式 | 强制转型表达式 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| CEIL 函数 | 返回大于或等于它的单个参数的最小整数 | 代数函数 | CEIL 函数 |
| CHARACTER_ LENGTH 函数 | 请参阅 CHAR_LENGTH 函数。（在多字节语言环境中，这替代 LENGTH 函数。） | 长度函数 | CHAR_LENGTH 函数 |
| CHAR_LENGTH 函数 | 返回字符串参数中逻辑字符的计数 | 长度函数 | CHAR_LENGTH 函数 |
| CHARINDEX 函数 | 返回子字符串在字符串内的位置 | CHARINDEX 函数 | CHARINDEX 函数 |
| CHR | 从缺省的代码集返回取值范围在 0 至 255 的代码点 | 字符串操纵函数 | CHR 函数 |
| 列表达式 | 来自表的列值 | 列表达式 | 列表达式 |
| CONCAT 运算符函数 | 串联两个表达式的结果 | 字符串操纵函数 | CONCAT 函数 |
| 串联(\|\|）运算符 | 串联两个表达式的结果 | 表达式 | 串联运算符 |
| 常量表达式 | 带有字面的、固定的或可变值的表达式 | 常量表达式 | 常量表达式 |
| COS 函数 | 返回弧度表达式的余弦 | 三角函数 | COS 函数 |
| COSH 函数 | 返回参数的双曲余弦，在此，该参数是以弧度表达的角 | 三角函数 | COSH 函数 |
| COUNT（作为函数集） | 返回频率计数的函数。下面罗列 COUNT 函数的每一形式。 | 聚集表达式 | COUNT 函数概述 |
| COUNT (ALL *column*) 函数 | 请参阅 COUNT (*column*) 函数。 | 聚集表达式 | COUNT 列函数 |
| COUNT (*column*) 函数 | 返回指定的列中非 NULL 值的数目 | 聚集表达式 | COUNT 列函数 |
| COUNT DISTINCT 函数 | 返回指定的列中唯一的非 NULL 值的数目 | 聚集表达式 | COUNT DISTINCT 和 COUNT UNIQUE 函数 |
| COUNT UNIQUE 函数 | 请参阅 COUNT DISTINCT 函数。 | 聚集表达式 | COUNT DISTINCT 和 COUNT UNIQUE 函数 |
| COUNT (*) 函数 | 返回满足查询的一组行的计 | 聚集表达式 | COUNT(*) 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| | 数 | | |
| CUME_DIST 函数 | 返回 OLAP 分区中每一行的百分比排名 | OLAP 分等级函数表达式 | CUME_DIST 函数 |
| CURRENT 运算符 | 返回由天的日期和时间构成的 DATETIME 值的当前时间 | 常量表达式 | CURRENT 运算符 |
| CURRENT_ROLE 运算符 | 返回当前启用的用户的角色 | 常量表达式 | CURRENT_ROLE 运算符 |
| CURRENT_USER 运算符 | 返回用户的授权标识符。USER 运算符的同义词。 | 常量表达式 | USER 或 CURRENT_USER 运算符 |
| *sequence*.CURRVAL | 返回指定的 *sequence* 的当前值 | 常量表达式 | 使用 CURRVAL |
| DATE 函数 | 将非日期参数转换为 DATE 值 | 时间函数 | DATE 函数 |
| DAY 函数 | 将该月的天数作为整数返回 | 时间函数 | DAY 函数 |
| DBINFO (*option*) | 检索数据库和会话信息的函数。下面罗列每一 *option*. | DBINFO 函数 | DBINFO 选项 |
| DBINFO ('bigserial') | 返回最近插入的 BIGSERIAL 值 | DBINFO 函数 | 使用 'serial8' 和 'bigserial' 选项 |
| DBINFO ('cdrsession') | 展示 DML 操作是否为复制的事务的一部分 | DBINFO 函数 | 使用 'cdrsession' 选项 |
| DBINFO ('dbhostname') | 返回客户端引用连接到其上的数据库服务器的主机名称 | DBINFO 函数 | 使用 'dbhostname' 选项 |
| DBINFO ('dbname') | 返回客户端应用连接到其上的数据库的标识符 | DBINFO 函数 | 使用 'dbname' 选项 |
| DBINFO ('dbspace', *tblspace_number*) | 返回对应于 tblspace number 的 dbspace 的名称 | DBINFO 函数 | 使用 ('dbspace', tblspace_num) 选项 |
| DBINFO ('get_tz') | 返回当前会话的时区 | DBINFO 函数 | 使用 'get_tz' 选项 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| DBINFO ('serial8') | 返回最近插入的 SERIAL8 值 | DBINFO 函数 | 使用 'serial8' 和 'bigserial' 选项 |
| DBINFO ('sessionid') | 返回当前会话的会话 ID | DBINFO 函数 | 使用 'sessionid' 选项 |
| DBINFO ('sqlca.sqlerrd1') | 返回插入到表中的最后的 serial 值 | DBINFO 函数 | 使用 'sqlca.sqlerrd1' 选项 |
| DBINFO ('sqlca.sqlerrd2') | 返回通过 DML 语句和通过 EXECUTE PROCEDURE 和 EXECUTE FUNCTION 语句处理的行的数目 | DBINFO 函数 | 使用 'sqlca.sqlerrd2' 选项 |
| DBINFO ('utc_current') | 返回当前的“世界标准时间”（UTC）值。 | DBINFO 函数 | 使用 'utc_current' 选项 |
| DBINFO ('utc_to_datetime', *expression*) | 返回指定 UTC 值的整数或列 *expression* 的 DATETIME 值。 | DBINFO 函数 | 使用 'utc_to_datetime' 选项 |
| DBINFO ('version', *parameter*) | 通过 *parameter* 指定的那样，返回客户端应用连接到的数据库服务器的确切版本的全部或一部分。 | DBINFO 函数 | 使用 'version' 选项 |
| DBSERVERNAME 函数 | 返回数据库服务器的名称 | 常量表达式 | DBSERVERNAME 和 SITENAME 运算符 |
| DECODE 函数 | 对一个或多个表达式对求值，并以指定的值表达式比较每一对中的 *when* 表达式 | DECODE 函数 | DECODE 函数 |
| DECRYPT_ BINARY 函数 | 在处理加密的 BLOB 参数之后，返回明文的 BLOB 数据值 | 加密和解密函数 | DECRYPT_BINARY 函数 |
| DECRYPT_CHAR 函数 | 在处理加密的参数之后，返回明文的字符串或 CLOB | 加密和解密函数 | DECRYPT_CHAR 函数 |
| DEFAULT_ROLE 运算符 | 返回当前用户的缺省的角色 | 常量表达式 | DEFAULT_ROLE 运算符 |
| DEGREES 函数 | 将弧的单位转换为度 | 三角函数 | DEGREES 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| DELETING 布尔运算符 | 如果触发器事件是 DELETE，则返回 't' | 触发器类型的布尔运算符 | 触发器类型的布尔运算符 |
| DENSERANK 函数 | DENSE_RANK 函数的同义词 | OLAP 分等级函数表达式 | DENSE_RANK 函数 |
| DENSE_RANK 函数 | 将 OLAP 分区中的每一行分等级，等级中没有间隔 | OLAP 分等级函数表达式 | DENSE_RANK 函数 |
| 除法（/）运算符 | 返回两个数值运算对象的商 | 表达式 | 算术运算符 |
| 双冒号（::）强制转型运算符 | 将表达式的值转换为指定的数据类型 | 强制转型表达式 | 强制转型表达式 |
| 双管道（\|\|）串联运算符 | 返回将一个字符串运算对象连接到另一字符串运算对象的字符串 | 表达式 | 串联运算符 |
| ENCRYPT_AES 函数 | 在处理明文字符串、BLOB 或 CLOB 之后，返回加密的字符串 | 加密和解密函数 | ENCRYPT_AES 函数 |
| ENCRYPT_TDES 函数 | 在处理明文字符串、BLOB 或 CLOB 之后，返回加密的字符串 | 加密和解密函数 | ENCRYPT_TDES 函数 |
| EXP 函数 | 返回数值表达式的指数 | 指数和对数函数 | EXP 函数 |
| EXTEND 函数 | 重置 DATETIME 或 DATE 值的精度 | 时间函数 | EXTEND 函数 |
| FILETOBLOB 函数 | 从存储在指定的操作系统文件中的数据，创建 BLOB 值 | 智能大对象函数 | FILETOBLOB 和 FILETOCLOB 函数 |
| FILETOCLOB 函数 | 从存储在指定的操作系统文件中的数据，创建 CLOB 值 | 智能大对象函数 | FILETOBLOB 和 FILETOCLOB 函数 |
| FIRST_VALUE 函数 | 对于每一 OLAP window 分区中的第一个行，返回指定表达式的值 | OLAP 聚集函数表达式 | LAST_VALUE 函数 |
| FLOOR 函数 | 返回小于或等于它的单个参数的最大的整数 | 代数函数 | FLOOR 函数 |
| FORMAT_UNITS 函数 | 返回指定内存或存储的数目和缩写的单位的字符串 | FORMAT_UNITS 函数 | FORMAT_UNITS 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| GETHINT 函数 | 在处理加密的数据-字符串参数之后，返回明文的提示字符串 | 加密和解密函数 | GETHINT 函数 |
| GREATEST 函数 | 返回值集中的最大值 | 代数函数 | GREATEST 函数 |
| HEX 函数 | 返回 base-10 整数参数的十六进制编码 | HEX 函数 | HEX 函数 |
| 主变量 | 请参阅变量。 | SQL 表达式的语法 | SQL 表达式的语法 |
| IFX_ALLOW_ NEWLINE 函数 | 设置 newline 会话模式，允许或不允许在括起来的字符串中的换行字符 | IFX_ALLOW_NEWLINE 函数 | IFX_ALLOW_NEWLINE 函数 |
| INITCAP 函数 | 将字符串参数转换为其中仅每一词的首字母为大写的字符串 | 大小写转换函数 | INITCAP 函数 |
| INSERTING 布尔运算符 | 如果触发器事件为 INSERT，则返回 't' | 触发器类型的布尔运算符 | 触发器类型的布尔运算符 |
| INSTR 函数 | 返回子字符串在字符串内第 N 次发生的位置 | INSTR 函数 | INSTR 函数 |
| ISNULL 函数 | 返回非 NULL 参数的值，或如果该参数为 NULL 则返回指定的值 | ISNUL 函数 | ISNULL 函数 |
| LAG 函数 | 在 OLAP 分区内的当前行之前，返回在指定的偏移量的行的表达式值 | OLAP 分等级函数表达式 | ids_sqs_1513.html#ids_sqs_1513 |
| LAST_DAY 函数 | 返回它的参数指定的那个月的最后一天的日期 | 时间函数 | LAST_DAY 函数 |
| LAST_VALUE 函数 | 返回 OLAP window 分区中最后一行的指定的表达式的值 | OLAP 聚集函数表达式 | LAST_VALUE 函数 |
| LEAD 函数 | 在 OLAP 分区中当前行之后，返回指定的偏移量的行的表达式值 | OLAP 分等级函数表达式 | ids_sqs_1513.html#ids_sqs_1513 |
| LEAST 函数 | 返回值集中的最小值 | 代数函数 | LEAST 函数 |
| LEFT 函数 | 返回字符串最左边的 N 个字符 | LEFT 函数 | LEFT 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| LEN 函数 | LENGTH 函数的同义词 | 长度函数 | LENGTH 函数 |
| LENGTH 函数 | 返回字符列中的字节数，不包括拖尾的空格 | 长度函数 | LENGTH 函数 |
| LIST 集合构造函数 | 可包含重复的值的有序的集合的构造函数 | 集合构造函数 | 集合构造函数 |
| 文字 BOOLEAN | BOOLEAN 值的文字表示 | 常量表达式 | 常量表达式 |
| 文字集合 | 代表集合数据类型中的元素 | 常量表达式 | 文字的集合 |
| 文字 DATETIME | 代表 DATETIME 值 | 常量表达式 | 文字的 DATETIME |
| 文字 INTERVAL | 代表 INTERVAL 值 | 常量表达式 | 文字的 INTERVAL |
| 精确数值 | 代表数值 | 常量表达式 | 精确数值 |
| 文字 opaque 类型 | 代表 opaque 数据类型 | 常量表达式 | 常量表达式 |
| 文字 row | 代表 ROW 数据类型中的元素 | 常量表达式 | 文字的 Row |
| LN | 返回数值参数的自然对数 | 指数和对数函数 | LN 函数 |
| LOCOPY 函数 | 创建智能大对象的副本 | 智能大对象函数 | LOCOPY 函数 |
| LOG10 函数 | 返回数值参数的以 10 为底的对数 | 指数和对数函数 | LOG10 函数 |
| LOGN 函数 | 返回数值参数的自然对数 | 指数和对数函数 | LOGN 函数 |
| LOTOFILE 函数 | 将 BLOB 或 CLOB 对象复制到文件 | 智能大对象函数 | LOTOFILE 函数 |
| LOWER 函数 | 将大写字母转换为小写 | 大小写转换函数 | LOWER 函数 |
| LPAD 函数 | 返回由指定数目的填充字符左填充的字符串 | 字符串操纵函数 | LPAD 函数 |
| LTRIM 函数 | 从字符串移除指定的开头填充字符。 | 字符串操纵函数 | LTRIM 函数 |
| MAX 函数 | 返回指定值集中的最大值 | 聚集表达式 | MAX 函数 |
| MDY 函数 | 从整数参数返回 DATE 值 | 时间函数 | MDY 函数 |
| MIN 函数 | 返回指定的值集中的最小值 | 聚集表达式 | MIN 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| MOD 函数 | 从两个数值参数返回模值（整数除的余值） | 代数函数 | MOD 函数 |
| MONTH 函数 | 从 DATE 或 DATETIME 参数返回月份值 | 时间函数 | MONTH 函数 |
| MONTHS_ BETWEEN 函数 | 返回两个时间参数之间的月份差 | 时间函数 | MONTHS_BETWEEN 函数 |
| 乘法（*）运算符 | 返回两个数值运算对象的乘积 | 表达式 | 算术运算符 |
| MULTISET 集合构造函数 | 可包含重复的值的元素的未排序的集合的构造函数 | 集合构造函数 | 集合构造函数 |
| NEXT_DAY 函数 | 返回同时满足两个条件的最早的日历日期 | 时间函数 | NEXT_DAY 函数 |
| *sequence*.NEXTVAL | 增加指定的 *sequence* 的值 | 常量表达式 | 使用 NEXTVAL |
| NTILE 函数 | 将 OLAP 分区中的行划分为近似基数的 N 个分级的类别，称为 *tiles* | OLAP 分等级函数表达式 | NTILE 函数 |
| NULL 关键字 | 未知的、缺失的或逻辑上未定义的值 | NULL 关键字 | NULL 关键字 |
| NULLIF 函数 | 如果两个值相等，则返回 NULL | NULLIF 函数 | NULLIF 函数 |
| NVL 函数 | 返回非 NULL 参数的值，或如果该参数为 NULL 则返回指定的值 | NVL 函数 | NVL 函数 |
| NVL2 函数 | 当第一个参数不是 NULL 时，返回第二个参数 | NVL2 函数 | NVL2 函数 |
| OCTET_LENGTH 函数 | 返回字符列中的字节数，包括任何结尾的空格 | 长度函数 | OCTET_LENGTH 函数 |
| PERCENT_RANK 函数 | 返回 OLAP window 分区中每一行的等级值，规格化到从 0 至 1 的范围 | OLAP 分等级函数表达式 | PERCENT_RANK 函数 |
| POW 函数 | 将一个基值升高到指定阶数的幂 | 代数函数 | POW 函数 |
| POWER® 函数 | POW 函数的同义词 | 代数函数 | POW 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| 过程调用表达式 | 请参阅用户定义的函数。 | 用户定义的函数 | 用户定义的函数 |
| 程序变量 | 请参阅变量。 | SQL 表达式的语法 | SQL 表达式的语法 |
| QUARTER 函数 | 返回 DATE 或 DATETIME 值的日历季度 | 时间函数 | QUARTER 函数 |
| 括起来的字符串 | 文字字符串 | 常量表达式 | 引用的字符串 |
| RADIANS 函数 | 将度数的单位转换为弧度 | 三角函数 | RADIANS 函数 |
| RANGE 函数 | 返回指定的值集的范围 | 聚集表达式 | RANGE 函数 |
| RANK | 返回一个序数数目来划分 OLAP window 中每一行的等级 | OLAP 分等级函数表达式 | RANK 函数 |
| RATIOTOREPORT 函数 | RATIO_TO_REPORT 函数的同义词 | OLAP 聚集函数表达式 | RATIO_TO_REPORT 函数 |
| RATIO_TO_REPORT 函数 | 返回同一 OLAP window 分区中每一行值对于所有行合计值的分数比率 | OLAP 聚集函数表达式 | RATIO_TO_REPORT 函数 |
| REPLACE 函数 | 替换源字符串中指定的字符 | 字符串操纵函数 | REPLACE 函数 |
| REVERSE | 颠倒源字符串中字符的顺序 | 字符串操纵函数 | REVERSE 函数 |
| RIGHT 函数 | 从源字符串返回最右边的 N 个字符 | RIGHT 函数 | RIGHT 函数 |
| ROOT 函数 | 返回实数、正值、数值参数的第 N 个根值 | 代数函数 | ROOT 函数 |
| ROUND 函数 | 返回参数的四舍五入的值 | 代数函数 | ROUND 函数 |
| ROW 构造函数 | 命名的 ROW 数据类型的构造函数 | 构造函数表达式 | ROW 构造函数 |
| ROWNUMBER 函数 | ROW_NUMBER 函数的同义词 | OLAP 编号函数表达式 | OLAP 编号函数表达式 |
| ROW_NUMBER 函数 | 返回 OLAP window 分区中每一行的序列整数 | OLAP 编号函数表达式 | OLAP 编号函数表达式 |
| RPAD 函数 | 返回由指定数目的填充字符 | 字符串操纵函 | RPAD 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| | 右填充的字符串 | 数 | |
| RTRIM 函数 | 从字符串移除结尾的空填充字符 | 字符串操纵函数 | RTRIM 函数 |
| SECLABEL_BY_ COMP 函数 | 返回其组件为该参数的安全标签 | 安全标签支持函数 | SECLABEL_BY_COMP 函数 |
| SECLABEL_BY_ NAME 函数 | 返回其标识符为该参数的安全标签 | 安全标签支持函数 | SECLABEL_BY_NAME 函数 |
| SECLABEL_TO_ CHAR 函数 | 返回其字符串格式为该参数的安全标签 | 安全标签支持函数 | SECLABEL_TO_CHAR 函数 |
| SELECTING 布尔运算符 | 如果触发器事件为 SELECT，则返回 't' | 触发器类型的布尔运算符 | 触发器类型的布尔运算符 |
| SET 集合构造函数 | 唯一的元素的未排序集合的构造函数 | 集合构造函数 | 集合构造函数 |
| SIGN 函数 | 返回数值参数的符号的标志 | SIGN 函数 | SIGN 函数 |
| SIN 函数 | 返回弧度参数的正弦 | 三角函数 | SIN 函数 |
| SINH 函数 | 返回弧度参数的双曲正弦 | 三角函数 | SINH 函数 |
| SITENAME 函数 | 请参阅 DBSERVERNAME 函数。 | 常量表达式 | DBSERVERNAME 和 SITENAME 运算符 |
| SLV 表达式 | 其作用域为声明它的 SQL 语句的语句-本地的变量（SLV） | 语句本地的变量声明 | 语句本地的变量表达式 |
| SPACE 函数 | 返回 N 个空字符的字符串 | 字符串操纵函数 | SPACE 函数 |
| SPL 例程表达式 | 请参阅“用户定义的函数” | 用户定义的函数 | 用户定义的函数 |
| SPL 变量 | 存储表达式的 SPL 变量 | SQL 表达式的语法 | SQL 表达式的语法 |
| SQLCODE 函数 | 将 sqlca.sqlcode 值返回到 SPL UDR | SQLCODE 函数（SPL） | SQLCODE 函数（SPL） |
| SQRT 函数 | 返回数值参数的平方根 | 代数函数 | SQRT 函数 |
| STDDEV 函数 | 返回数据集的标准偏差 | 聚集表达式 | STDDEV 函数 |
| SUBSTR 函数 | 返回源字符串的一子字符串 | SUBSTR 函数 | SUBSTR 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| SUBSTRB 函数 | 返回源字符串的一子字符串 | SUBSTRB 函数 | SUBSTRB 函数 |
| SUBSTRING 函数 | 返回源字符串的一子字符串 | SUBSTRING 函数 | SUBSTRING 函数 |
| SUBSTRING_INDEX 函数 | 返回包括第 N 次出现一定界符的子字符串 | SUBSTRING_INDEX 函数 | SUBSTRING_INDEX 函数 |
| Substring（[ x, *y* ]）运算符 | 从字符串运算对象返回子字符串 | 列表达式 | 使用子字符串运算符 |
| 减法（-）运算符 | 返回两个数值的差 | 表达式 | 算术运算符 |
| SUM 函数 | 返回指定的值集合总和 | 聚集表达式 | SUM 函数 |
| SYSDATE 运算符 | 从系统时钟返回当前的 DATETIME 值。 | 常量表达式 | SYSDATE 运算符 |
| SYSTIMESTAMP 运算符 | 在 Oracle 模式下，从系统时钟返回当前的 DATETIME 值。 | 常量表达式 | SYSTIMESTAMP 运算符 |
| TAN 函数 | 返回弧度表达式的正切 | 三角函数 | TAN 函数 |
| TANH 函数 | 返回弧度参数的双曲正切 | 三角函数 | TANH 函数 |
| TO_CHAR 函数 | 将时间或数值转换成字符串 | 时间函数 | TO_CHAR 函数 |
| TO_DATE 函数 | 将字符串转换成 DATETIME 值 | 时间函数 | TO_DATE 函数 |
| TO_NUMBER 函数 | 将数值或字符串转换成 DECIMAL 值 | TO_NUMBER 函数 | TO_NUMBER 函数 |
| TODAY 运算符 | 返回当前的系统日期 | 常量表达式 | TODAY 运算符 |
| TRIM 函数 | 从字符串参数删除空填充字符 | 字符串操纵函数 | TRIM 函数 |
| TRUNC 函数 | 返回截断的数值或时间值 | 代数函数 | TRUNC 函数 |
| 一元减号（-） | 指定负数（< 0）值 | 表达式 | 算术运算符 |
| 一元加号（+） | 指定整数（> 0）值。 | 表达式 | 算术运算符 |
| UNITS 运算符 | 将整数转化为 INTERVAL 值 | 常量表达式 | UNITS 运算符 |
| UPDATING 布尔运算符 | 如果触发器事件为 UPDATE，则返回 't' | 触发器类型的布尔运算符 | 触发器类型的布尔运算符 |
| UPPER 函数 | 将小写字母转换为大写 | 大小写转换函数 | UPPER 函数 |

| 名称 | 描述 | 语法 | 用法 |
| --- | --- | --- | --- |
| 用户定义的聚集 | 用户定义的聚集（相对于内建的聚集） | 用户定义的聚集 | 用户定义的聚集 |
| 用户定义的函数 | 用户编写的函数（相对于内建的函数） | 用户定义的函数 | 用户定义的函数 |
| USER 运算符 | 返回当前用户的授权表示法 | 常量表达式 | USER 或 CURRENT_USER 运算符 |
| 变量 | 存储值的主变量或程序变量 | SQL 表达式的语法 | SQL 表达式的语法 |
| VARIANCE 函数 | 返回数值值集的差异 | 聚集表达式 | VARIANCE 函数 |
| WEEKDAY 函数 | 返回代表星期几的整数代码 | 时间函数 | WEEKDAY 函数 |
| Window 聚集函数 | 返回来自 OLAP window 分区的聚集结果 | OLAP window 表达式 | OLAP window 聚集函数 |
| YEAR 函数 | 返回表示年份的 4 位整数 | 时间函数 | YEAR 函数 |
| * 符号 | 请参阅“乘法（*）运算符” | SQL 表达式的语法 | 算术运算符 |
| + 符号 | 请参阅“加法”和“一元加号（+）” | SQL 表达式的语法 | 算术运算符 |
| - 符号 | 请参阅“减法”和“一元减号（-）” | SQL 表达式的语法 | 算术运算符 |
| / 符号 | 请参阅“除法运算符” | SQL 表达式的语法 | 算术运算符 |
| :: 符号 | 请参阅“双冒号（::）强制转型运算符” | 强制转型表达式 | 强制转型表达式 |
| \|\| 符号 | 请参阅“双管道（\|\|）串联运算符” | SQL 表达式的语法 | 串联运算符 |
| q’转义 | 请参阅“q’转义运算符” | SQL 表达式语法 | 字符串运算符 |
| [ *first*, *last* ] 符号 | 请参阅“子字符串运算符” | 列表达式 | 使用子字符串运算符 |

下面的部分描述出现在前面表格中的每一表达式的语法和用法。

## 算术运算符

二元算术运算符可组合返回数值的表达式。

| 算术运算 | 算术运算符 | 运算符函数 | 算术运算 | 算术运算符 | 运算符函数 |
|---|---|---|---|---|---|
| 加法 | + | plus( ) | 乘法 | * | times( ) |
| 减法 | - | minus( ) | 除法 | / | divide( ) |

下列示例使用二元算术运算符：

```
quantity * total_price
 price * 2
COUNT(*) + 2
```

如果您将 DATETIME 值与一个或多个 INTERVAL 值组合，则所有 INTERVAL 值的字段都出现在 DATETIME 值中；不执行隐式的 EXTEND 功能。此外，您不可使用带有 DAY 至 SECOND 间隔的 YEAR 至 MONTH 间隔。要获取关于二元算术运算符的附加信息，请参阅 *《GBase 8s SQL 指南：参考》*。

二元算术运算符与运算符函数相关联，如前面的表格所示。以二元运算符连接两个表达式等同于对这些表达式调用相关联的运算符函数。例如， 下列两个语句都选择 total_price 列与 2 的乘积。在第一个语句中，* 运算符隐式地调用 times( ) 函数。

```
SELECT (total_price * 2) FROM items
    WHERE order_num = 1001;
SELECT times(total_price, 2) FROM items
    WHERE order_num = 1001;
```

您不可使用算术运算符来将使用聚集函数的表达式与列表达式组合。

数据库服务器为所有内建的数据类型提供与关系运算符相关联的运算符函数。您可定义这些运算符函数的新版本来处理您自己的用户定义的数据类型。

要获取更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

数据库服务器还支持下列一元算术运算符。

| 数值的符号 | 一元算术运算符 | 运算符函数 |
|---|---|---|
| 正 | + | positive( ) |
| 负 | - | negate( ) |

一元算术运算符有前面的表格展示的相关联的运算符函数。您可定义这些函数的新版本来处理您自己的用户定义的数据类型。要获取更多关于此主题的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

如果参与在算术表达式中的任何值为 NULL，则整个表达式的值为 NULL，如下例所示：

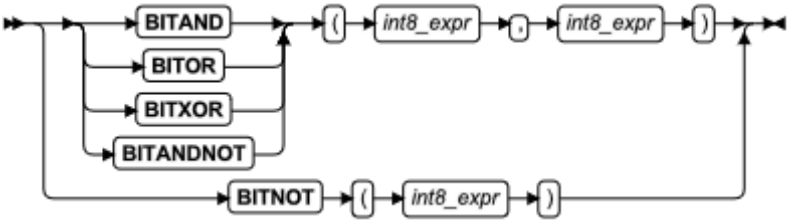
```
SELECT order_num, ship_charge/ship_weight FROM orders
            WHERE order_num = 1023;
```

如果或 **ship_charge** 或 **ship_weight** 为 NULL，则表达式,**ship_charge/ship_weight** 的返回值也为 NULL。 如果在条件中使用 NULL 表达式 **ship_charge/ship_weight**，则它的真值不可为 TRUE，且不满足条件（除非该 NULL 表达式是 **IS NULL** 运算符的一个运算对象）。

## 位逻辑函数

使用位逻辑函数来执行命名的位运算。

位逻辑函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *int8_expr* | 可转化为 INT8 值的数值表达式 | 对于 BITNOT，最大的大小减 1 | 表达式 |

这些函数的参数可为可转换为 INT8 数据类型的任何数值数据类型。

除了带有单个参数的 **BITNOT** 之外，这些位逻辑函数都有两个可转换为 INT8 值的参数。

如果两个参数都有相同的整数类型，则返回值的数据类型与参数的类型相同。如果两个参数的整数类型不同，则返回值为精度较高的整数类型。例如，如果第一个参数类型为 INT，而第二个参数类型为 INT8，则返回值的类型为 INT8。

如果参数为任何其他数值类型，诸如 DECIMAL、SMALLFLOAT、FLOAT 或 MONEY，或那些类型的某种组合，则返回值数据类型为 DECIMAL(32)。

如果使用主变量，且在准备时刻不知道参数的类型，则假设两个参数都是数据类型 INTEGER，且返回值为 INTEGER。在执行时刻准备之后，如果为主变量提供不同的数据类型值，则 GBase 8s 发出 -9750 错误。要防止发生这样的情况，您可通过使用强制转型来指定主变量数据类型，如下列 ESQL/C 程序片断所示：

```
sprintf(query1, ",
    bitand(   ?::int8,   ?::int8) from mytab");
EXEC SQL prepare selectq from :query;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor
    using :hostvar_int8_input1, :hostvar_int8_input2;

EXEC SQL fetch select_cursor into :var_int8_output;
```

## BITAND 函数

**BITAND** 函数有两个参数。这些参数可为可转换为 INT8 值的任何数值类型值。

在位运算之前截断小数值。结果是两个参数的 AND。

如果两个参数有相同的整数类型，则返回值的数据类型与参数的类型相同。如果参数有不同的整数类型（例如，INT 和 INT8），则返回带有更高精度的数据类型。如果参数是任何其他数值类型，诸如 DECIMAL、SMALLFLOAT、FLOAT 或 MONEY，或那些类型的某种组合，则返回的数据类型为 DECIMAL(32)。

下列示例展示调用 BITAND 函数的查询：

```
select task_id, task_status,
        decode(bitand(task_status,1), 1, '    Y', '    N') as task_a,
        decode(bitand(task_status,2), 2, '    Y', '    N') as task_b,
        decode(bitand(task_status,4), 4, '    Y', '    N') as task_c
        from tasks;
```

下表展示此 SELECT 语句的输出。

| task_id | task_status | task_a | task_b | task_c |
|---|---|---|---|---|
| 100 | 1 | Y | N | N |
| 101 | 1 | Y | N | N |
| 102 | 2 | N | Y | N |
| 103 | 4 | N | N | Y |
| 104 | 6 | N | Y | Y |
| 105 | 3 | Y | Y | N |
| 106 | 5 | Y | N | Y |
| 107 | 7 | Y | Y | Y |

## BITOR 函数

**BITOR** 函数有两个参数。这些参数可为可转换为 INT8 值的任何数值类型值。

在位运算之前截断小数值。结果是它的两个参数的位 OR。

如果两个参数都有相同的整数类型，则返回值的数据类型与参数的类型相同。如果参数是不同的整数类型（例如，INT 和 INT8），则返回的类型是有较高精度的类型。如果参数是任何其他的数值类型，诸如 DECIMAL、SMALLFLOAT、FLOAT 或 MONEY，或那些类型的某种组合，则返回的数据类型为 DECIMAL(32)

下列示例说明调用 BITOR 函数的查询：

```
SELECT BITOR(8, 20) AS bitor FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitor |
|---|
| 28 |

# BITXOR 函数

**BITXOR** 函数有两个参数。参数可为可转换为 INT8 值的任何数值类型值。

在位运算之前，截断小数值。结果是它的两个参数的位 XOR。

如果两个参数都有相同的整数类型，则返回值的数据类型与参数的类型相同。如果参数是不同的数据类型（例如，INT 和 INT8），返回的类型是更高精度的类型。如果参数是任何其他的数值类型，诸如 DECIMAL、SMALLFLOAT、FLOAT 或 MONEY，或那些类型的某种组合，则返回的数据类型是 DECIMAL(32)。

下列示例说明调用 BITXOR 函数的查询：

```
SELECT BITXOR(41, 33) AS bitxor FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitxor |
| --- |
| 8 |

此查询调用带有负参数的 BITXOR 函数：

```
SELECT BITXOR(-20, -41) AS bitxor FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitxor |
| --- |
| 59 |

# BITANDNOT 函数

**BITANDNOT** 函数有两个参数。参数可为可转换为 INT8 值的任何数值类型值。

在位运算之前，截断小数值。结果与两个参数的 `BITAND(arg1, BITNOT(arg2))` 相同。

如果两个参数都有相同的整数类型，则返回值的数据类型与参数的类型相同。如果参数是不同的整数类型（例如，INT 和 INT8），则返回的类型是精度更高的类型。如果参数是任何其他的数值类型，诸如 DECIMAL、SMALLFLOAT、FLOAT 或 MONEY，或那些类型的某种组合，则返回的数据类型是 DECIMAL(32)。

下列示例中的查询调用 BITANDNOT 函数：

```
SELECT BITANDNOT(20,-20) AS bitandnot FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitandnot |
| --- |
| 16 |

下列查询为前面的示例中的参数调用等同的 BITAND 和 BITNOT 函数：

```
select bitand(20, bitnot(-20)) as bitandnot from systables
        where tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitandnot |
| --- |
| 16 |

## BITNOT 函数

**BITNOT** 函数可取一个小于最大的 INT8 值的任何数值类型值。

在位运算之前，截断小数值。结果是它的参数的位 NOT。

如果参数是 SMALLINT、INT、BIGINT 或 INT8，则返回的数据类型与参数的类型相同。否则返回的数据类型为 DECIMAL(32)。

下列查询调用 BITNOT 函数：

```
SELECT BITNOT(8) AS bitnot FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitnot |
| --- |
| -9 |

下一查询调用带有负参数的 BITNOT 函数：

```
SELECT BITNOT(-20) AS bitnot FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| bitnot |
| --- |
| 19 |

## 串联运算符

串联运算符是二元运算符，在 SQL 表达式 的通用图中展示其语法。您可使用串联运算符（||）来串联两个求值为字符数据类型或数值数据类型的两个表达式。这些示例展示一些可能的串联的表达式组合。

- 第一个示例将 **zipcode** 列串联到 **lname** 列的前三个字母。
- 第二个示例将后缀 **.dbg** 串联到名为 **file_variable** 的主变量的内容。
- 第三个示例将 **TODAY** 运算符返回的值串联到字符串 Date。

```
lname[1,3] || zipcode

:file_variable || '.dbg'

'Date:' || TODAY
```

您不可在下列嵌入式语言语句中使用串联运算符：

- ALLOCATE COLLECTION
- ALLOCATE DESCRIPTOR
- ALLOCATE ROW
- CREATE FUNCTION FROM
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DEALLOCATE COLLECTION
- DEALLOCATE DESCRIPTOR
- DEALLOCATE ROW DESCRIBE
- DESCRIBE INPUT
- EXECUTE
- FLUSH
- GET DESCRIPTOR
- GET DIAGNOSTICS
- PUT
- SET AUTOFREE
- SET CONNECTION
- SET DESCRIPTOR
- WHENEVER

除非 DECLARE 和 PREPARE 语句另有注明，在下列动态 SQL 语句中，以诸如 GBase 8s ESQL/C 语言这样的外部语言编写的例程不可使用串联运算符：

- CLOSE
- DECLARE
- EXECUTE IMMEDIATE
- FETCH
- FREE
- OPEN
- PREPARE

虽然诸如 ***cursor_id*** 规范这样的 DECLARE 语句的输入参数不可为包括串联运算符的表达式，但 GBase 8s ESQL/C 例程可在 DECLARE 语句内的 SELECT、INSERT、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句中使用此运算符。

GBase 8s ESQL/C 例程可在 SQL 语句的文本中或在您传递到 PREPARE 语句的语句中使用串联运算符。

在 SPL 例程中，您可在指定传递到 EXECUTE IMMEDIATE 语句或 PREPARE 语句的 SQL 语句的文本的表达式中包括串联运算符，即使该 SPL 例程的调用上下文是一 GBase 8s ESQL/C 例程。

您不可随同用户定义的数据类型、随同复合的或大对象数据类型直接地使用串联运算符，也不可随同非内建的字符或数值数据类型的运算对象使用。在您可将结果传递到串联运算符之前，您必须将 UDT 或其他被支持的数据类型显式地强制转型为内建的字符或数值数据类型。

串联运算的结果的数据类型依赖于运算对象的数据类型以及结果字符串的长度，使用 从 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

串联运算符（||）有相关联的名为 **CONCAT** 的运算符函数。不可重载 **CONCAT** 函数。

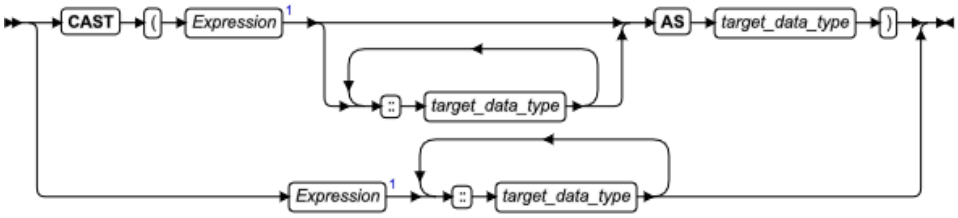
当您定义基于文本的 UDT 时，您可定义 **CONCAT** 函数来串联用户定义的数据类型的对象。要获取更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

## 强制转型表达式

您可使用 CAST 和 AS 关键字或双冒号强制转型运算符（ **::** ）来将表达式强制转型为另一数据类型。运算符和这些关键字都调用从表达式的数据类型到指定的目标数据类型的强制转型。

要调用显式的强制转型，您可使用强制转型运算符或 CAST AS 关键字。如果您使用强制转型运算符或 CAST 和 AS 关键字，但未定义显式的或隐式的强制转型来执行两种数据类型之间的转换，则该语句返回错误。

强制转型表达式




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *target_data_type* | 由强制转型返回的数据类型 | 请参阅“对于目标数据类型的规则” | 数据类型 |

## 对于目标数据类型的规则

下列规则限制强制转型表达式中的***目标数据类型***:

- 目标数据类型必须为内建的、用户定义的数据类型，或数据库中的命名的 row 类型。
- 目标数据类型不可为未命名的 row 类型或集合类型。
- 在下列情况下，目标数据类型可为 BLOB 数据类型：
  - 源表达式（要被强制转型到另一数据类型的表达式）为 BYTE 数据类型。
  - 源表达式是用户定义的类型，且用户已定义了从用户定义的类型到 BLOB 类型的强制转型。
- 在这些条件下，目标数据类型可为 CLOB 类型：
  - 源表达式是 TEXT 数据类型。

- 源表达式是用户定义的类型，且用户已定义了从用户定义的类型到 CLOB 类型的强制转型。

- 您不可将 BLOB 数据类型强制转型为 BYTE 数据类型。
- 您不可将 CLOB 数据类型强制转型为 TEXT 数据类型。
- 必须存在可将源表达式的数据类型转换为目标数据类型的显式的或隐式的强制转型。

## 强制转型表达式的示例

下例示例展示将 *x* 与 *y* 的总和转化为用户定义的数据类型 user_type 的两种不同方法。这两种方式产生相同的结果。二者都需要存在从由 (x + y) 返回的类型到用户定义的类型的显式的或隐式的强制转型：

```
CAST ((x + y) AS user_type)
 (x + y)::user_type
```

下列示例展示查找等同于表达式 expr 的整数的两种不同方法。二者都需要存在从数据类型 expr 到 INTEGER 数据类型的隐式的或显式的强制转型：

```
CAST (expr AS INTEGER)
expr::INTEGER
```

在下列示例中，用户将 BYTE 列强制转型为 BLOB 类型，并将 BLOB 数据复制到操作系统文件：

```
SELECT LOTOFILE(mybytecol::blob, 'fname', 'client')
            FROM mytab
   WHERE pkey = 12345;
```

在下列示例中，用户将 TEXT 列强制转型为 CLOB 值，然后将同一表中的 CLOB 列更新为从 TEXT 列派生的 CLOB 值：

```
UPDATE newtab SET myclobcol = mytextcol::clob;
```

## 强制转型表达式中的关键字 NULL

强制转型表达式可出现在 projection 列表中，包括形如 NULL::*datatype* 的表达式，在此，*datatype* 是数据库已知的任何数据类型：

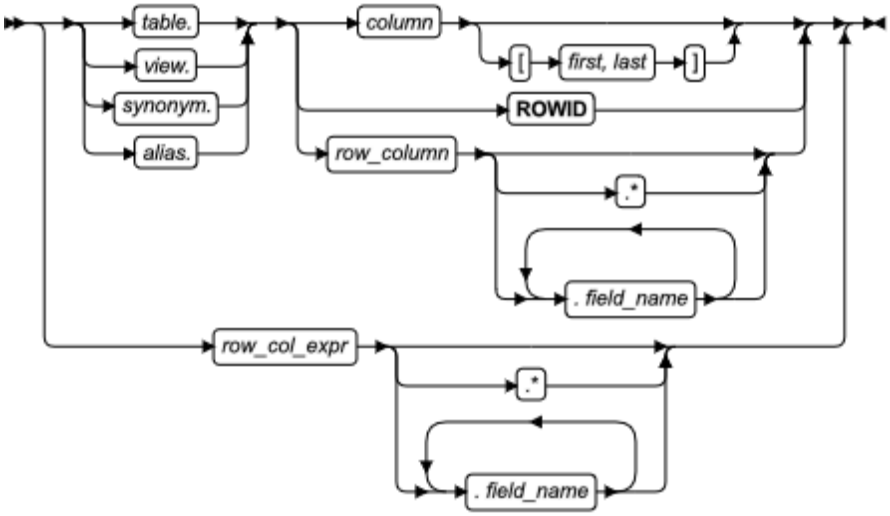
```
SELECT newtable.col0, null::int FROM newtable;
```

关键字 NULL 在表达式内有全局的引用作用域。在 SQL 中，关键字 NULL 是访问 NULL 值的唯一语法机制。对于关键字 NULL 的全局作用域的任何重新定义或限制的尝试（例如，声明名为 null 的 SPL 变量），都会禁用涉及 NULL 值的任何强制转型表达式。请确保关键字 NULL 在所有表对象上下文中收到它的全局作用域。

## 列表达式

*列表达式*指定数据库中列的数据值，或值的子字符串，或 ROW 类型列内的字段。这是列表达式的语法。

列表达式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | 表或视图的临时可替换的名称，在查询的 FROM 子句中声明 | 必须返回字符串。限制依赖于 *alias* 发生在其中的 SELECT 语句的子句 | 标识符 |
| *column* | 列的名称 | 限制依赖于 *column* 发生位置的 SQL 语句 | 标识符 |
| *field_name* | 在 ROW 列或 ROW 列表达式中 ROW 字段的名称 | 必须为 *row-column name* 或 *row_col_expr* 或 *field name*（对于嵌套的行）指定的行的成员 | 标识符 |
| *first*、*last* | 指示 *column* 内第一个字符和最后一个字符位置的整数 | *column* 必须为 CHAR、VARCHAR、NCHAR、NVARCHAR、BYTE 或 TEXT 类型，且 0 < *first* ≤ *last* | 精确数值 |
| *row_col_expr* | 返回 ROW 类型值的表达式 | 必须返回 ROW 数据类型 | 表达式 |
| *row_column* | ROW 类型列的名称 | 必须为命名的 ROW 数据类型或未命名的 ROW 数据类型 | 标识符 |
| *synonym*、*table*、*view* | 包含 *column* 的表、视图或（表或视图的）同义词 | 同义词以及它指向的表或视图必须存在 | 数据库对象名称，数据库对象名 |

下列示例展示列表达式：

company
items.price
cat_advert [1,15]

每当有必要区分那些有相同的名称但在不同的表中的列时，您必须以*表*名称或*别名*限定*列*。下列展示 SELECT 语句的示例使用来自 customer 和 orders 表的 customer_num。第一个示例将表名称置于列名称之前。第二个示例将表别名置于列名称之前。

```
SELECT * FROM customer, orders
  WHERE customer.customer_num = orders.customer_num;
SELECT * FROM customer c, orders o
   WHERE c.customer_num = o.customer_num;
```

## 使用点表示法

点表示法（有时称为*成员运算符*）允许您以为其组件的另一 SQL 标识符限定 SQL 标识符。 您以句号（ . ）分隔这些标识符。例如，您可以任何下列 SQL 标识符限定列名称：

- 表名称：***table_name.column_name***
- 视图名称：***view_name.column_name***
- 同义词名称：***syn_name.column_name***

这些点表示法的格式称为*列 **projections***。

您还可使用点表示法来直接地访问命名的或未命名的 ROW 列的字段，如下例所示：

***row-column name. field name***

点表示法的这种使用称为*字段 projection*。例如，假设您有带有下列定义的名为 rect 的列：

```
CREATE TABLE rectangles
        (
        area float,
        rect ROW(x int, y int, length float, width float)
        );
```

下列 SELECT 语句使用点表示法来访问 rect 列的字段 length：

```
SELECT rect.length FROM rectangles WHERE area = 64;
```

### *以星号表示法选择 ROW 列的所有字段*

如果您想要选择有 ROW 类型的列的所有字段，则您可不使用点表示法指定列名称。例如，您可选择 rect 列的所有字段，如下：

```
SELECT rect FROM rectangles WHERE area = 64;
```

您还可使用星号（*）表示法来投影有 ROW 数据类型的列的所有字段。例如，如果您想要使用星号表示法来选择 rect 列的所有字段，您可输入下列语句：

```
SELECT rect.* FROM rectangles WHERE area = 64;
```

与分别地指定 rect 列的每一字段相比，星号表示法更容易：

```
SELECT rect.x, rect.y, rect.length, rect.width
        FROM rectangles
        WHERE area = 64;
```

在 SELECT 语句的 projection 列表中，ROW 字段星号表示法是有效的。它可指定 ROW 类型列的所有字段，或 ROW 列表达式返回的数据。

星号表示法不必带有 ROW 类型列，因为您可单独指定列名称来投影它的所有字段。然而，带有 ROW 类型表达式的星号表示法非常有用，诸如返回 ROW 类型值的子查询和用户定义的函数。要获取更多信息，请参阅 使用带有 Row 类型表达式的点表示法。

您仅可在 SELECT 语句的 projection 列表中使用带有 ROW 数据类型的列或表达式的星号表示法。您不可在 SELECT 语句的任何其他子句中使用带有 ROW 类型的列和表达式的星号表示法。

### *选择嵌套的字段*

当定义列的 ROW 类型本身包含其他 ROW 类型时，该列包含嵌套的字段。使用点表示法来访问列内的这些嵌套的字段。

例如，假设 employee 表的 address 列包含字段：street、city、state 和 zip。此外，zip 字段包含嵌套的字段：z_code 和 z_suffix。zip 字段上的查询返回 z_code 和 z_suffix 字段的值。然而，您可指定查询仅返回特定嵌套的字段。下列示例展示如何使用点表示法来构造 SELECT 语句，该语句仅返回 address 列的 z_code 字段的行：

```
SELECT address.zip.z_code    FROM employee;
```

### *优先级的规则*

数据库服务器使用下列优先级规则来解释点表示法：

1. schema ***name_a*** . table ***name_b*** . column ***name_c*** . field ***name_d***
2. table ***name_a*** . column ***name_b*** . field ***name_c*** . field ***name_d***
3. column ***name_a*** . field ***name_b*** . field ***name_c*** . field ***name_d***

当标识符的含义不明确时，数据库服务器使用优先级规则来确定标识符指定的是哪个数据库对象。请考虑下列两个表：

```
CREATE TABLE b (c ROW(d INTEGER, e CHAR(2));
        CREATE TABLE c (d INTEGER);
```

在下列 SELECT 语句中，表达式 c.d 引用表 c 的列 d（而不是表 b 中列 c 的字段 d），因为表表达式比列标识符有更高的优先级：

```
SELECT * FROM b,c WHERE c.d = 10;
```

要获取更多关于优先级规则以及如何随同 ROW 列使用点表示法的信息，请参阅 *GBase 8s SQL 教程指南*。

### *使用带有 Row 类型表达式的点表示法*

除了指定 ROW 数据类型的列之外，您还可使用带有任何求值为 ROW 类型的表达式的点表示法。例如，在 INSERT 语句中，您可在返回值的单独行的子查询中使用点表示法。假设您创建名为 row_t 的 ROW 类型：

```
CREATE ROW TYPE row_t (part_id INT, amt INT);
```

还假设您创建了基于 row_t ROW 类型的名为 tab1 的 typed 表：

```
CREATE TABLE tab1 OF TYPE row_t;
```

还假设您将下列值插入了表 tab1：

```
INSERT INTO tab1 VALUES (ROW(1,7));
        INSERT INTO tab1 VALUES (ROW(2,10));
```

最后，假设您创建了另一名为 tab2 的表：

```
CREATE TABLE tab2 (colx INT);
```

现在，您可使用点表示法来将仅从表 tab1 的 part_id 列的值插入 tab2 表内：

```
INSERT INTO tab2
        VALUES ((SELECT t FROM tab1 t
        WHERE part_id = 1).part_id);
```

当您想要选择 ROW 类型列的所有字段时，星号形式的点表示法不是必须的，因为您可单独指定列名称来选择所有它的字段。然而，当您如在前面示例中那样使用子查询，或当您调用返回 ROW 类型值的用户定义的函数时，点表示法的星号形式可非常有用。

假设名为 new_row 的用户定义的函数返回 ROW 类型值，且您想要调用此函数来将该 ROW 类型值插入表内。星号表示法使得指定将 new_row( ) 函数返回的所有 ROW 类型列插入表内变得非常容易：

```
INSERT INTO mytab2 SELECT new_row (mycol).* FROM mytab1;
```

在分片表达式中不允许引用 ROW 类型列的字段或 ROW 类型表达式。分片表达式是在像 CREATE TABLE、CREATE INDEX 和 ALTER FRAGMENT 那样的 SQL 语句中定义表分片或索引分片的表达式。

## 使用子字符串运算符

您可在 CHAR、VARCHAR、NCHAR、NVARCHAR、BYTE 和 TEXT 列上使用子字符串运算符来定义***列子字符串***作为通过该表达式指定的列的一部分。

当一对方括号（[ ]）括起以逗号分隔的无符号整数，其中 ***first*** 整数大于零但不大于 ***last*** 整数时，在字符列的标识符之后， GBase 8s 将方括号解释为子字符串运算符。表达式返回列中数据值的从 ***first*** 直到 ***last*** 字符，在此 ***first*** 和 ***last*** 定义子字符串。例如，在表达式 `cat_advert[6,15]` 中，返回值是列 **cat_advert** 的从第 6 个字符直到第 15 个字符。

在缺省的语言环境中，如果数据值占据至少 15 字节，则此表达式求值为包括该列值的十字节的一子字符串。但在多字节语言环境中，此表达式返回十个连续的逻辑字符的字符串，其存储长度可能超过 10 字节，以第六个逻辑字符开头。要获取关于列子字符串的 GLS 方面的更多信息，请参阅 *GBase 8s GLS 用户指南*。

在下列示例中，如果 customer 表的 lname 列中的值是 Greenburg，则下列表达式求值为 burg：

```
lname[6,9]
```

条件表达式可包括使用子字符串运算符 ( [ *first*, *last* ] ) 的列表达式，如下例所示：

```
SELECT lname FROM customer WHERE phone[5,7] = '356';
```

此处需要引号来防止数据库服务器将数值过滤器应用到标准值中的数字。

另请参阅 字符串操纵函数 部分，其描述两个可指定 SQL 语句内子字符串表达式的内建的 SQL 函数，**SUBSTR( )** 和 **SUBSTRING( )**。

**注：** 数据库服务器可使用通过子字符串运算符定义的子字符串作为查询中的索引过滤器。然而，对于由 SUBSTR( ) 或 SUBSTRING( ) 定义的子字符串，或对于其他内建的字符串操纵函数，情况并非如此。

## 使用 Rowid

在 GBase 8s 中，您可使用与表行相关联的 **rowid** 列作为该行的一个属性。**rowid** 列本质上是未分片的表中以及以 WITH ROWIDS 子句创建了的分片的表中的一个隐藏列。对于每一行，**rowid** 列是唯一的，但它不必是顺序的。然而，推荐您使用主键作为访问方式，而不是利用 **rowid** 列。

下列示例使用 SELECT 语句中的 ROWID 关键字：

```
SELECT *, ROWID FROM customer;
SELECT fname, ROWID FROM customer ORDER BY ROWID;
SELECT HEX(rowid) FROM customer WHERE customer_num = 106;
```

最后的示例展示如何获得您的行的定位的页号（0x 之后的前六位）和槽号（最后两位）。

在包含聚集函数的查询的 Projection 子句的选择列表中，您不可使用 ROWID 关键字。

## 使用 ROWNUM

ROWNUM 是数据库系统对结果集的编序排列，结果集中第一行的 ROWNUM 值为 1，第二行的 ROWNUM 值为 2 ，依次类推。

可以使用 ROWNUM 限制查询返回的总行数，如下所示。

```
SELECT * FROM customer WHERE ROWNUM < 10;
```

如果在同一查询连用 ROWNUM 和 ORDER BY，则会先根据 ROWNUM 条件取结果集，然后再重新排序，因此，以下语句可能返回与上述示例不同的结果：

```
SELECT * FROM customer WHERE ROWNUM < 10 ORDER BY fname;
```

如果想要先排序再应用 ROWNUM 条件，则可以在子查询中嵌入 ORDER BY 子句，而将 ROWNUM 条件放置顶层查询中。例如，以下查询返回按客户编号大小排序的前 10 行结果集。

```
SELECT * FROM
    (SELECT * FROM customer ORDER BY customer_num)
    WHERE ROWNUM < 11;
```

在此示例中，ROWNUM 值是第一层 SELECT 语句的值，因此，它们在子查询已经通过 customer_num 排序后生成。

**注意：**

1. ROWNUM 不能使用“> ”条件。

2. 建表 CREATE TABLE 不支持定义 ROWNUM 作为列名。例如，create table tab1 ( rownum int)，执行报错。

## 使用智能大对象

SELECT、UPDATE 和 INSERT 语句不直接操纵智能大对象的值。相反，它们使用***句柄值***（一类指针）来访问 BLOB 或 CLOB 值，如下：

- SELECT 语句返回指向 projection 列表指定的 BLOB 或 CLOB 值的句柄值。SELECT 不返回 projection 列表指定的 BLOB 或 CLOB 列的实际值。相反，它返回指向该列数据的句柄值。
- INSERT 和 UPDATE 语句不将 BLOB 或 CLOB 列的实际数据发送到数据库服务器。相反，它们接受指向此数据的句柄值作为要插入或更新的值。

要访问智能大对象列的数据，您必须使用下列应用编程接口（API）之一：

- 从一 GBase 8s ESQL/C 程序内，使用访问智能大对象的 GBase 8s ESQL/C 库函数。要获取更多信息，请参阅 *GBase 8s ESQL/C 程序员手册*。
- 从诸如 DataBlade 模块这样的 C 程序内，使用客户端和服务器 API。

您不可在涉及算术运算符的表达式中使用智能大对象列的名称。例如，对智能大对象句柄值的诸如加法或减法这样的运算没有意义。

当您选择智能大对象列时，您可将句柄值赋予任何数目的列：带有相同句柄值的所有列共享该 CLOB 或 BLOB 值。这种存储管理降低 CLOB 或 BLOB 值的磁盘空间量，但当几个列分享同一智能大对象值时，导致下列情况：

- 提高在 CLOB 或 BLOB 列上锁争夺的机会。如果两列共享同一智能大对象值，则该数据库可能被需要访问它的一列锁定。
- 可从多个点更新 CLOB 或 BLOB 值。

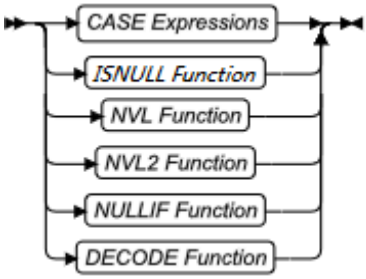
要移除这些限制，您可为需要访问它的每一列都创建单独的 BLOB 或 CLOB 数据的副本。您可使用 **LOCOPY** 函数来创建现有智能大对象的副本。

您还可使用内建的函数 **LOTOFILE**、**FILETOCLOB** 和 **FILETOBLOB** 来访问智能大对象值，如智能大对象函数 中描述的那样。要获取更多关于 BLOB 和 CLOB 数据类型的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 条件表达式

条件表达式返回依赖于条件测试的结果的值。此图展示条件表达式的语法。

条件表达式

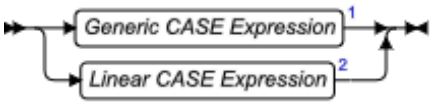



## CASE 表达式

CASE 表达式允许诸如 SELECT 语句这样的 SQL 语句返回几个可能的结果之一，依赖于这几个条件中那哪个求值为真。

CASE 表达式有两种形式：通用的 CASE 表达式和线性的 CASE 表达式。

CASE 表达式




在 CASE 表达式中，您必须包括至少一个 WHEN 子句。随后的 WHEN 子句和 ELSE 子句是可选的。您可在 SQL 语句中您可使用列表达式的任何地方使用通用的或线性的 CASE 表达式（例如，在 SELECT 语句的 Projection 子句中）。

搜索条件中的表达式或结果值表达式可包含子查询，且您可在另一 CASE 表达式中嵌套 CASE 表达式。

当在聚集表达式中出现 CASE 表达式时，您不可使用 CASE 表达式中的聚集函数。

您可指定触发器类型布尔运算符（DELETING、INSERTING、SELECTING 或 UPDATING）作为仅在触发器例程内的 CASE 表达式中的条件。

下列查询片断声明两个聚集列表达式的别名：

```
SELECT . . .
SUM(orders.ship_weight) as o2,
COUNT(DISTINCT
CASE WHEN orders.backlog MATCHES 'n'
THEN orders.order_num END ) AS o3,
. . .
```

在此，SUM 的参数为 DECIMAL(8,2) 列值，且 COUNT DISTINCT 聚集以 CASE 表达式作为它的参数。

请不要弄混 CASE 表达式与 SPL 的 CASE 语句，它们支持不同的语法和功能。

### *CASE 表达式数据类型兼容性*

在 CASE 表达式中，所有的结果都应为同一数据类型或可兼容的数据类型。

如果在所有 WHEN ... THEN 分支子句中的结果不是同一数据类型或兼容的数据类型，则发生错误。

下表展示哪些字符数据类型是兼容的，以及为每一组合返回的数据类型。

表 1. 从兼容的字符数据类型返回的数据类型

| 数据类型 | NCHAR (>255) | NCHAR (<=255) | NVARCHAR | CHAR (<=255) | CHAR (>255) | VARCHAR | LVARCHAR (>255) | LVARCHAR (<=255) |
|---|---|---|---|---|---|---|---|---|
| NCHAR (>255) | NCHAR | NCHAR | NCHAR | NCHAR | NCHAR | NCHAR | NCHAR | NCHAR |
| NCHAR (<=255) | NCHAR | NCHAR | NVARCHAR | NCHAR | NCHAR | NVARCHAR | NCHAR | NCHAR |
| NVARCHAR | NCHAR | NVARCHAR | NVARCHAR | NVARCHAR | NCHAR | NVARCHAR | NCHAR | NVARCHAR |
| CHAR (<=255) | NCHAR | NCHAR | NVARCHAR | CHAR | CHAR | VARCHAR | CHAR | CHAR |
| CHAR (>255) | NCHAR | NCHAR | NCHAR | CHAR | CHAR | CHAR | CHAR | CHAR |
| VARCHAR | NCHAR | NVARCHAR | NVARCHAR | VARCHAR | CHAR | VARCHAR | CHAR | VARCHAR |
| LVARCHAR (>255) | NCHAR | NCHAR | NCHAR | CHAR | CHAR | CHAR | LVARCHAR | LVARCHAR |
| LVARCHAR (<=255) | NCHAR | NCHAR | NVARCHAR | CHAR | CHAR | VARCHAR | LVARCHAR | LVARCHAR |

下表展示哪些数值数据类型是兼容的，以及为每一组合返回的数据类型。

表 2. 从可兼容的数值数据类型返回的数据类型

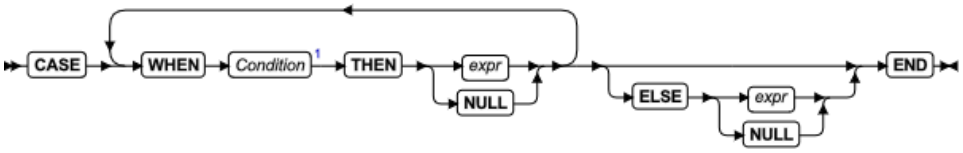
| 数据类型 | INTEGER | SMALLINT | SERIAL | DECIMAL | FLOAT | SMALLFLOAT | MONEY | BIGINT | BIGSERIAL |
|---|---|---|---|---|---|---|---|---|---|
| INTEGER | INTEGER | INTEGER | INTEGER | DECIMAL | DECIMAL | DECIMAL | MONEY | DECIMAL | DECIMAL |
| SMALLINT | INTEGER | SMALLINT | INTEGER | DECIMAL | DECIMAL | DECIMAL | MONEY | DECIMAL | DECIMAL |
| SERIAL | INTEGER | INTEGER | SERIAL | DECIMAL | DECIMAL | DECIMAL | MONEY | DECIMAL | DECIMAL |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | MONEY | DECIMAL | DECIMAL |
| FLOAT | DECIMAL | DECIMAL | DECIMAL | DECIMAL | FLOAT | FLOAT | MONEY | DECIMAL | DECIMAL |
| SMALLFLOAT | DECIMAL | DECIMAL | DECIMAL | DECIMAL | FLOAT | SMALLFLOAT | MONEY | DECIMAL | DECIMAL |
| MONEY | MONEY | MONEY | MONEY | MONEY | MONEY | MONEY | MONEY | MONEY | MONEY |
| BIGINT | DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | MONEY | BIGINT | BIGINT |
| BIGSERIAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | DECIMAL | MONEY | BIGINT | BIGSERIAL |

## *通用的 CASE 表达式*

通用的 CASE 表达式测试 WHEN 子句中为真的条件。如果它发现为真的条件，则它返回在 THEN 子句中指定的结果。

通用的 CASE 表达式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 返回某种数据类型的表达式 | 在 THEN 子句中的 *expr* 的数据类型必须与在其他 THEN 子句中的表达式的数据类型相兼容 | 表达式 |

数据库服务器以 WHEN 子句在该语句出现的顺序处理它们。如果 WHEN 子句的搜索条件求值为 TRUE，则数据库服务器使用对应的 THEN 表达式的值作为结果，并停止处理该 CASE 表达式。

如果没有 WHEN 条件求值为 TRUE，则数据库服务器使用 ELSE 表达式作为总的结果。如果没有 WHEN 条件求值为 TRUE，且未指定了 ELSE 子句，则返回的 CASE 表达式值为 NULL。您可使用 IS NULL 条件来处理 NULL 结果。要获取更多关于如何处理 NULL 值的信息，请参阅 IS NULL 和 IS NOT NULL 条件。

下一示例展示 Projection 子句中通用的 CASE 表达式。

在此示例中，用户检索每一客户的名称和地址以及基于那个顾客存在的问题的数目而计算出的数值：

```
SELECT cust_name,
        CASE
        WHEN number_of_problems = 0
        THEN 100
        WHEN number_of_problems > 0 AND number_of_problems < 4
```

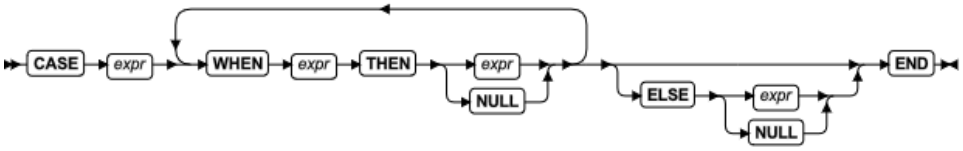
```
THEN number_of_problems * 500
WHEN number_of_problems >= 4 and number_of_problems <= 9
THEN number_of_problems * 400
ELSE
(number_of_problems * 300) + 250
END,
cust_address
FROM custtab
```

在通用的 CASE 表达式中，所有的结果都应为同一数据类型，或它们应求值为以共同兼容的数据类型。如果在所有 WHEN 子句中的结果不是同一数据类型，或如果它们未求值为相互兼容的类型的值，则发生错误。要获取更多关于返回的数据类型的兼容性的信息，请参阅 CASE 表达式数据类型兼容性。

## *线性的 CASE 表达式*

线性的 CASE 表达式将跟在 CASE 关键字之后的表达式的值与 WHEN 子句中的表达式作比较。

线性的 CASE 表达式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 返回某种数据类型的值的表达式 | 跟在 WHEN 关键字之后的 *expr* 的数据类型必须与跟在 CASE 关键字之后的表达式的数据类型相兼容。THEN 子句中的 *expr* 的数据类型必须与其他 THEN 子句中表达式的数据类型相兼容。 | 表达式 |

数据库服务器对跟在 CASE 关键字之后的表达式求值，然后顺序地处理 WHEN 子句。如果 WHEN 关键字之后的表达式返回的值与跟在 CASE 关键字之后的表达式的一样，则数据库服务器使用跟在 THEN 关键字之后的表达式的值作为该 CASE 表达式的总结果。然后，数据库服务器停止处理该 CASE 表达式。

如果没有 WHEN 表达式返回与跟在 CASE 关键字之后的表达式相同的值，则数据库服务器使用 ELSE 子句的表达式作为该 CASE 表达式的总结果（或，如果未指定了 ELSE 子句，则该 CASE 表达式的返回值为 NULL）。

下一示例展示 SELECT 语句的 Projection 子句的 projection 列表中的线性的 CASE 表达式。对于电影标题表中的每部电影，该查询返回电影的标题、成本和类型。该语句使用 CASE 表达式来派生每部电影的类型：

```
SELECT title, CASE movie_type
WHEN 1 THEN 'HORROR'
```

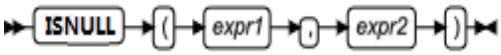
```
WHEN 2 THEN 'COMEDY'
WHEN 3 THEN 'ROMANCE'
WHEN 4 THEN 'WESTERN'
ELSE 'UNCLASSIFIED'
END,
our_cost FROM movie_titles;
```

在线性的 CASE 表达式中，WHEN 子句表达式的数据类型必须与跟在 CASE 关键字之后的表示的数据类型相兼容。

## ISNULL 函数

ISNULL 表达式返回不同的结果，这取决于于它的第一个参数求值是否为 NULL。

ISNULL 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr1*<br>*expr2* | 返回兼容的数据类型的值的表达式 | 无 | 表达式 |

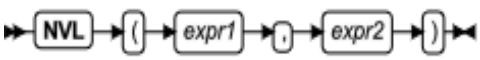
**ISNULL** 对 ***expression1*** 求值。如果 ***expression1*** 不是 NULL，则 **ISNULL** 返回 ***expression1*** 的值。如果 ***expression1*** 为 NULL，则 **ISNULL** 返回 ***expression2*** 的值。

表达式 ***expression1*** 和 ***expression2*** 可为任何数据类型，只要可将它们强制转型为共同的兼容的数据类型。

## NVL 函数

NVL 表达式返回不同的结果，这依赖于它的第一个参数求值是否为 NULL。

NVL 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr1*<br>*expr2* | 返回兼容的数据类型的值的表达式 | 不可为主变量或 BYTE 或 TEXT 对象 | 表达式 |

**NVL** 对 ***expression1*** 求值。如果 ***expression1*** 不是 NULL，则 **NVL** 返回 ***expression1*** 的值。如果 ***expression1*** 为 NULL，则 **NVL** 返回 ***expression2*** 的值。表达式 ***expression1*** 和 ***expression2*** 可为任何数据类型，只要可将它们强制转型为共同的兼容的数据类型。

假设 employees 表的 addr 列在有些行中有 NULL 值，且用户想要能够为这些行打印标签 Address unknown。 当 addr 列有 NULL 值时，用户输入下列 SELECT 语句来显示标签 Address unknown：

```
SELECT fname, NVL (addr, 'Address unknown') AS address  FROM employees;
```

## NULLIF 函数

NULLIF 表达式返回不同的结果，这依赖于它的两个参数是否相等。

NULLIF 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr1*<br>*expr2* | 返回兼容的数据类型的值的表达式 | 不可为 BYTE 或 TEXT 数据类型 | 表达式 |

NULLIF 对它的两个参数 *expr1* 和 *expr2* 求值。

- 如果它们的值相等，则 **NULLIF** 返回 NULL。
- 如果它们的值不等，则 **NULLIF** 返回 ***expr1***。

*expr1* 与 *expr2* 参数可为存在内建的比较函数的任何数据类型，或可强制转型为有内建的比较函数的相兼容的数据类型的任何两种数据类型。

下列示例使用 NULLIF 函数来将布尔 FALSE 值（'f'）转换为 NULL 值：

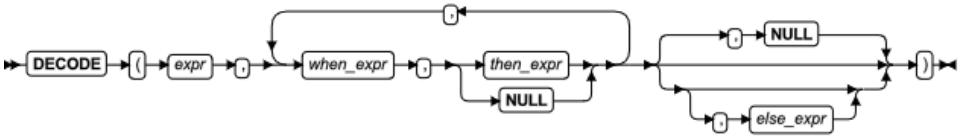
```
SELECT name, answer, NULLIF(answer, 'f') FROM booktab;
```

在此，第一个参数是可有真（'t'）或假（'f'）值的布尔列表达式，且第二个布尔参数始终为 'f'（表示 FALSE）。对于在 **answer** 列中有 'f' 的行，通过 **NULLIF** 函数返回的值会是 NULL（因为当参数相等时，返回 NULL 值）。然而，对于有 't' 作为第一个参数，通过 **NULLIF** 返回的值总是 't'，因为当一个是 't' 而其他的是 'f' 时，两个参数不可相等；当两个值不相等时，返回第一个参数。

## DECODE 函数

**DECODE** 表达式类似于 CASE 表达式，它能够根据指定列中找到的值打印不同的结果。

DECODE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 可对其值和数据类型求值的表达式 | 无 | 表达式 |
| *when_expr* | 对比值 | *when_expr* 和 *expr* 的数据 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| | | 类型必须一致、*when_expr* 的值不可为 NULL。 | |
| *then_expr* | 对应 *when_expr* 的返回值 | 与 *when_expr* 成对出现，多个 *then_expr* 值可以是不同数据类型。但是：<br>• 当第一个 ***then_expr*** 是数值型时，其它 ***then_expr*** 不允许为日期型；<br>• 当第一个 ***then_expr*** 是日期型时，其它 ***then_expr*** 不允许为非整数数值型； | 表达式 |
| *else_expr* | 缺省值 | 可以省略。 | 表达式 |

表达式 ***expr***、***when_expr*** 和 ***then_expr*** 是必需的。**DECODE** 对 ***expr*** 求值，并把它和 ***when_expr*** 比较。如果 ***when_expr*** 的值与 ***expr*** 的值相匹配，则 **DECODE** 返回 ***then_expr***。

表达式 ***when_expr*** 和 ***then_expr*** 是一个表达式对，并且可以在 **DECODE** 函数中指定任意数量的表达式对。在所有情况下，**DECODE** 都把表达式对中的第一个成员和 ***expr*** 比较，如果第一个成员和 ***expr*** 相匹配，那么就返回第二个成员。

如果没有表达式与 ***expr*** 匹配，**DECODE** 就返回 ***else_expr***。如果没有表达式与 ***expr*** 相匹配，并且未指定 ***else_expr***，则 **DECODE** 返回 NULL。

假设用户要将 students 表的 evaluation 列中的描述性值在输出中的转换为数字值。下表给出了 students 表的内容。

| firstname | evaluation |
|---|---|
| Edward | Great |
| Joe | Not done |
| Mary | Good |
| Jim | Poor |

现在，用户输入带有 DECODE 函数的查询，将 evaluation 列中的描述性值转换为等同的数值：

```
SELECT firstname, DECODE(evaluation,
            'Poor', 0,
            'Fair', 25,
            'Good', 50,
            'Very Good', 75,
            'Great', 100,
```

```
        -1) as grade
        FROM students;
```

此 SELECT 语句的输出如下所示：

```
firstname       evaluation
---------  ----------
Edward          100
Joe             -1
Mary            50
Jim              0
```

可以为参数指定任何数据类型。如果多个 ***then_expr*** 值的数据类型不一致，DECODE() 函数返回的结果值的数据类型同第一个 ***then_expr*** 值的数据类型相同。

当返回的结果值数据类型与第一个 ***then_expr*** 值的数据类型不同时，会自动按照以下规则进行隐式转换：

- 支持数值型、日期型转换为字符型；
- 支持纯数值（科学计数法）字符串转换为数值型；
- 支持日期型字符串转换为日期型；
- 支持整数数值型转换为日期型。

  a) 整数值范围：[-2147483648,214783647]

  b) 整数 0 转换为 1899-12-31 00:00:00.00000，数值每增加或减少 1，日期相应增加或减少 1 天。

- 支持数值型字符串转换为日期型

  将数值型字符串转换为数值型数据。对于浮点数，截断小数点后面的数字，只保留整数部分，然后按照上一条规则进行处理。

**示例 1：返回字符型结果值**

SELECT DECODE（1,1,’aa’,2,456,3,’789’,000）from dual;

返回结果为：aa

该语句返回的结果值的数据类型为字符型，与第一个 ***then_expr*** 值（ ‘aa’）的类型一致。

**示例 2：返回数值型结果值**

SELECT DECODE（400,100,1+2,300,’aa’,66）from dual;

返回结果为：66.0000000000000

该语句最终返回的结果值的数据类型为 decimal 。

**示例 3：返回日期型结果值**

SELECT DECODE（3,100,today,2,’aa’,3,sysdate,6）from dual;

返回结果为：2018-08-30 16:00:48.00000

SELECT DECODE（4,100,today,2,'aa',3,sysdate,1）from dual;

返回结果为：1900-01-01 00:00:0.00000

该语句最终返回的结果值的数据类型为日期型，与第一个 ***then_expr*** 值（today）的类型一致。并将符合条件的返回值 1 转换为日期 1900-01-01 00:00:0.00000。

## 常量表达式

返回固定的值的某些表达式称为*常量表达式*。这些包括读取系统时钟的变量函数运算符，但在字面常量也是有效的上下文中才是有效的。

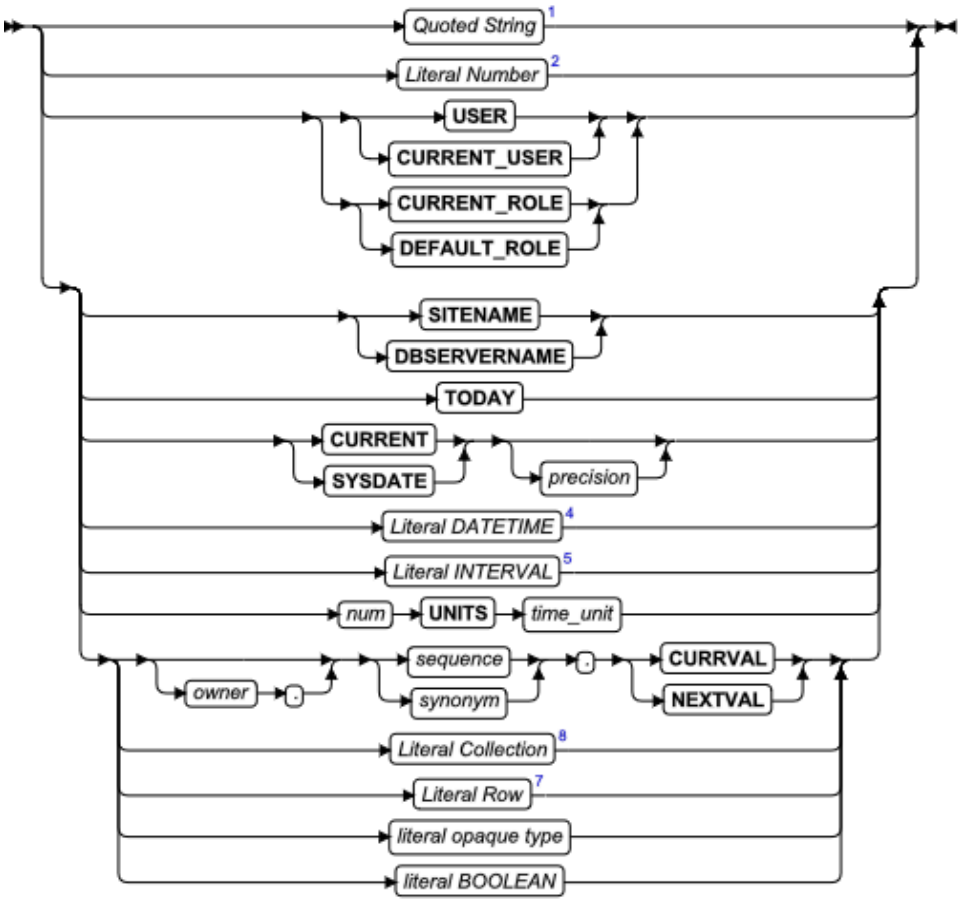
在这些表达式之中有下列运算符（或***系统常量***），在运行时确定其返回的值：

- **CURRENT** 从系统时钟返回当前的时间和日期。
- **CURRENT_ROLE** 返回角色的名称（如果有的话），为当前用户启用其权限。
- **CURRENT_USER** 是 **USER** 的同义词。
- **DEFAULT_ROLE** 返回角色的名称（如果有的话），是当前用户的缺省角色。
- **DBSERVERNAME** 返回当前数据库服务器的名称。
- **SITENAME** 是 **DBSERVERNAME** 的同义词。
- **SYSDATE** 从系统时钟读取 DATETIME 值，像 **CURRENT** 运算符一样，但有不同的缺省精度。
- **TODAY** 从系统时钟返回当前的日历日期。
- **USER** 返回当前用户的登录名称（也称为***授权标识符***）。

除了这些运算符，术语***常量表达式***还指括起来的字符串、文字值或带有运算对象的 **UNITS** 运算符。

常量表达式段有下列语法。

常量表达式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *literal Boolean* | BOOLEAN 值的文字表示 | 必须为 *t* (TRUE) 或 *f* (FALSE) | 引用字符串 |
| *literal opaque type* | opaque 数据类型的值的文字表示 | 必须被 opaque 类型的输入支持函数所识别 | 由 UDT 开发者定义 |
| *num* | 指定时间单位的数量。请参阅 UNITS 运算符。 | 如果 *num* 不是整数，则截断小数部分 | 精确数值 |
| *owner* | 序列的所有者的名称 | 必须拥有序列 | 所有者名称 |
| *precision* | 返回的 DATETIME 表达式的精度 | 在 Windows™ 系统上，秒的最大范围是 FRACTION(3)。 | DATETIME 字段限定符 |
| *sequence* | 序列的名称 | 在当前数据库中必须存在 | 标识符 |
| *synonym* | 序列的名称的同义词 | 在当前数据库中必须存在 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *time_unit* | 指定时间单位的关键字：YEAR、MONTH、DAY、HOUR、MINUTE、SECOND *或* FRACTION | 必须为左边的关键字之一。不区分大小写，但不可括在引号内 | 请参阅 Restrictions 列。 |

## 引用的字符串

下列示例展示作为表达式的引用的字符串：

```
SELECT 'The first name is ', fname FROM customer;
  INSERT INTO manufact VALUES ('SPS', 'SuperSport');
  UPDATE cust_calls SET res_dtime = '2007-1-1 10:45'
                WHERE customer_num = 120 AND call_code = 'B';
```

要获取更多信息，请参阅 引用字符串。

## 精确数值

精确数值指定数值值。

下列示例展示作为表达式的精确数值：

```
INSERT INTO items VALUES (4, 35, 52, 'HRO', 12, 4.00);
INSERT INTO acreage VALUES (4, 5.2e4);
SELECT unit_price + 5 FROM stock;
SELECT -1 * balance FROM accounts;
```

要获取更多信息，请参阅 精确数值。

## USER 或 CURRENT_USER 运算符

**USER** 运算符返回包含正在运行该进程的当前用户的登录名称（也称为授权标识符）的字符串。**CURRENT_USER** 运算符是 **USER** 运算符的同义词。

下列语句展示您可以如何使用 USER 运算符：

```
INSERT INTO cust_calls VALUES
            (221,CURRENT,USER,'B','Decimal point off', NULL, NULL);
SELECT * FROM cust_calls WHERE user_id = USER;
UPDATE cust_calls SET user_id = USER WHERE customer_num = 220;
```

**USER** 不更改用户 ID 的字母大小写。如果您在表达式中使用 **USER** 且当前用户为 **Robertm**，则 **USER** 运算符返回 **Robertm**，而不是 **robertm** 或 **ROBERTM**。

如果您指定 **USER** 作为缺省的列值，则 ***column*** 必须为类型 CHAR、VARCHAR、NCHAR、NVARCHAR 或 LVARCHAR。

如果您指定 **USER** 作为列的缺省值，则 ***column*** 的大小应不小于 32 字节。如果该列长度太小以至于不能存储缺省值，则在诸如 INSERT 或 ALTER TABLE 这样的操作期间，您会面临出错的风险。

在符合 ANSI 的数据库中，如果您不将 ***owner*** 名称括在引号中，则将表所有者的名称存储为大写字母。如果您使用 **USER** 运算符作为条件的一部分，则您必须确保存储 user 名称的方法与 USER 运算符返回的内容在字母大小写方面相匹配。

## CURRENT_ROLE 运算符

**CURRENT_ROLE** 运算符返回包含正在运行会话的用户的当前启用的角色的名称的字符串。或使用 SET ROLE 语句在会话中显式地设置了此 ***role*** ，或当当前用户连接到数据库时，隐式地设置为缺省的角色。如果该用户不持有角色，或如果未授予当前启用的用户角色，则 **CURRENT_ROLE** 返回 NULL 值。如果尚未授予了用户角色，但已将缺省角色授予了 PUBLIC，且已显式地或隐式地启用了此缺省角色，则 **CURRENT_ROLE** 返回此缺省角色的名称。

下一语句展示您可以如何使用 CURRENT_ROLE 运算符：

```
select CURRENT_ROLE FROM systables WHERE tabid = 1;
```

**CURRENT_ROLE** 运算符不更改角色的标识符的字母大小写。如果您在表达式中使用 **CURRENT_ROLE**，且当前角色为 **Czarina**，则 **CURRENT_ROLE** 运算符返回 **Czarina**，而不是 **czarina**。

如果您指定 **CURRENT_ROLE** 作为列的缺省值，则该列必须有 CHAR、VARCHAR、LVARCHAR、NCHAR 或 NVARCHAR 数据类型。因为角色的名称是授权标识符，因此如果列长度小于 32 字节，则可能发生截断。

## DEFAULT_ROLE 运算符

**DEFAULT_ROLE** 运算符求值为包含已授予了正在运行会话的用户的缺省角色的名称的字符串。此缺省角色无需当前已经启用，但自从最近的 GRANT DEFAULT ROLE 语句在 TO 子句引用的该用户或 PUBLIC 以来，它必须未被调用。

如果没有为当前用户显式地定义缺省角色，但 PUBLIC 有缺省角色，则 **DEFAULT_ROLE** 返回 PUBLIC 的缺省角色。

如果用户没有缺省角色，或如果最近将缺省角色显式地授予了该用户，或随后通过 REVOKE DEFAULT ROLE 语句作为 PUBLIC 调用了，则 **DEFAULT_ROLE** 返回 NULL 值。如果用户尚未被单个地授予了缺省角色，但已将缺省角色授予了 PUBLIC，则 **DEFAULT_ROLE** 运算符返回此缺省角色的名称。然而，如果当前未为用户也未为 PUBLIC 定义缺省角色，则 **DEFAULT_ROLE** 返回 NULL。

SET ROLE 语句对 **DEFAULT_ROLE** 运算符不起作用，但如果 SET ROLE 已激活了某其他角色，或如果 SET ROLE 指定了 NULL 或 NONE 作为用户的当前角色，则用户没必要获得缺省角色的任何访问权限。

下一语句展示您可以如何使用 DEFAULT_ROLE 运算符：

```
select DEFAULT_ROLE from systables where tabid = 1;
```

**DEFAULT_ROLE** 不更改角色的标识符的字母大小写。

如果您指定 **DEFAULT_ROLE** 作为列的缺省值，则该列必须有 CHAR、VARCHAR、LVARCHAR、NCHAR 或 NVARCHAR 数据类型。由于角色的名称是授权标识符，因此如果列宽小于 32 字节，则可能发生截断。（要了解授权标识符的语法，请参阅 所有者名称。）

## DBSERVERNAME 和 SITENAME 运算符

**DBSERVERNAME** 运算符返回数据库服务器的 SQL 标识符，如同当前数据库所在的 GBase 8s 实例的 ONCONFIG 文件中 DBSERVERNAME 参数所定义的那样，或如同 **GBASEDBTSERVER** 环境变量中所指定的那样。**SITENAME** 是 **DBSERVERNAME** 运算符的关键字同义词。

您可使用 **DBSERVERNAME** 运算符来指定表的位置，将信息放到表内，或从表抽取信息。您可将 **DBSERVERNAME** 插入到简单字符字段内或使用它作为列的缺省值。

如果指定 **DBSERVERNAME** 作为 CREATE TABLE 或 ALTER TABLE 语句中的缺省列值，则该列必须为 CHAR、VARCHAR、LVARCHAR、NCHAR 或 NVARCHAR 数据类型。

如果您指定 **DBSERVERNAME** 或 **SITENAME** 作为列的缺省值，则该列的大小应至少为 128 字节长。如果列的长度太小以至于不能存储缺省值，则在 INSERT 和 ALTER TABLE 操作期间，您会面临得到错误消息的风险。

下列示例在 DML 语句中使用 DBSERVERNAME 或 SITENAME。

- 第一个 SELECT 语句返回 **customer** 表所在的数据库服务器实例的名称。（因为查询不受 WHERE 子句的限制，所以它为表中每行返回相同的 **DBSERVERNAME** 值。如果您在 projection 子句中包括 DISTINCT 关键字，则数据库仅返回 **DBSERVERNAME** 一次。）
- 第二个语句将包含当前数据库服务器的名称的行添加到表。
- 第三个语句返回在 **host_tab.site_col** 列中有当前数据库服务器的名称的所有行。
- 最后的语句将当前数据库服务器的名称更改为其 **customer_num** 的 SERIAL 值为 120 的行中 **customer.company** 列的值：

```
SELECT DBSERVERNAME FROM customer;
INSERT INTO host_tab VALUES ('1', SITENAME);
SELECT * FROM host_tab WHERE site_col = DBSERVERNAME;
UPDATE customer SET company = SITENAME    WHERE customer_num = 120;
```

## TODAY 运算符

使用 **TODAY** 运算符来返回系统日期作为 DATE 数据类型。如果您指定 **TODAY** 作为缺省的列值，则该列必须为 DATE 列。

下列示例展示您可以在 INSERT、UPDATE 或 SELECT 语句中如何使用 TODAY 运算符：

```
UPDATE orders (order_date) SET order_date = TODAY WHERE order_num = 1005;
INSERT INTO orders VALUES (0, TODAY, 120, NULL, N, '1AUE217', NULL, NULL, NULL,
NULL);
SELECT * FROM orders WHERE ship_date = TODAY;
```

要获取设置非缺省的时区的代码示例，请参阅 CURRENT 运算符。

## CURRENT 运算符

**CURRENT** 运算符返回带有今日的日期和时间的 DATETIME 值，展示当前时刻。

如果您未指定 DATETIME 限定符，则缺省的限定符为 YEAR TO FRACTION(3)。当数据库服务器从操作系统获取当前时间时，USEOSTIME 配置参数指定它是否使用亚秒精度。要获取更多关于 USEOSTIME 配置参数的信息，请参阅您的 *GBase 8s 管理员参考手册*。

您可在字面 DATETIME 为有效的任何上下文中使用 **CURRENT**。（请参阅 文字的 DATETIME）。如果您指定 **CURRENT** 作为列的缺省值，则它必须为 DATETIME 列且 **CURRENT** 的限定符必须与列限定符相匹配，如下列示例所示：

```
CREATE TABLE new_acct (col1 INT, col2 DATETIME YEAR TO DAY
DEFAULT CURRENT YEAR TO DAY);
```

始终在当前数据库所在的数据库服务器中为 **CURRENT** 求值。如果当期数据库在远程数据库服务器中，则从远程主机返回值。

SQL 不是过程的语言，且 **CURRENT** 可能不以在语句中它的位置的词典次序执行。在 SQL 语句的执行中，您不应使用 **CURRENT** 来标记开始、终止、特定的点。

如果您在单个语句中使用 **CURRENT** 运算符超过一次，则可能由 CURRENT 的每一实例返回相同的值。您不可依靠 **CURRENT** 来在它每次执行时返回不同的值。

当指定 CURRENT 的 SQL 语句启动执行时，该返回值基于系统时钟且是固定的。例如，从 EXECUTE FUNCTION（或 EXECUTE PROCEDURE）语句调用的 SPL 函数之内，对 **CURRENT** 的任何调用都返回当 SPL 函数启动时系统时钟的值。

在 UNIX™ 和 Linux™ 系统上，通过 CURRENT 运算符返回的值的精度由它的 DATETIME 限定符决定，精度范围可从单个时间单位（诸如 MONTH TO MONTH）直到 YEAR TO FRACTION (5)。然而，在 Windows™ 上的系统时钟仅返回亚秒精度。即使您在 DATETIME 限定符中指定 "FRACTION(5)"，Windows 上的 CURRENT 运算符也不支持高于 "FRACTION(3)" 的精度。

如果您的平台不提供以亚秒精度的当前时间返回的系统调用，则 **CURRENT** 为 FRACTION 字段返回零。

在下列示例中，第一个语句使用 WHERE 条件中的 **CURRENT**。第二个语句使用 **CURRENT** 作为 **DAY** 函数的参数。最有的查询选择其 **call_dtime** 值在从 2007 年初到当前时刻的范围内的行：

```
DELETE FROM cust_calls WHERE res_dtime < CURRENT YEAR TO MINUTE;
SELECT * FROM orders WHERE DAY(ord_date) < DAY(CURRENT);
SELECT * FROM cust_calls WHERE call_dtime BETWEEN '2007-1-1 00:00:00' AND CURRENT;
```

要获取更多信息，请参阅 DATETIME 字段限定符。

# SYSDATE 运算符

**SYSDATE** 运算符从系统时钟返回当前的 DATETIME 值。**SYSDATE** 与 **CURRENT** 运算符是相同的，除了 **SYSDATE** 的缺省精度是 YEAR TO FRACTION(5)，而 **CURRENT** 的缺省精度是 YEAR TO FRACTION(3)。

在不支持 ***seconds*** 范围大于 FRACTION(3) 的 Windows™ 平台上，**SYSDATE** 实际上是 **CURRENT** 运算符的同义词。

您可在 **CURRENT** 运算符有效的任何上下文中使用 **SYSDATE**。

在下列示例中的 SQL 语句使用 SYSDATE 运算符来为数据库的两个 DATETIME 列指定缺省值，并将新行插入到该表内：

```
CREATE TABLE tab1 (
            id SERIAL,
            value CHAR(20),
            time1 DATETIME YEAR TO FRACTION(5) DEFAULT SYSDATE,
            time2 DATETIME YEAR TO SECOND DEFAULT SYSDATE YEAR TO SECOND
            );
INSERT INTO tab1 VALUES (0, 'description', SYSDATE, SYSDATE);
```

下列查询访问在前面的示例中创建了的表：

```
SELECT SYSDATE AS sysdate, * FROM tab1;
```

当发出 INSERT 和 SELECT 语句时，结果对日期和时间是灵敏的，但在 2007 年 9 月 23 日该查询可能返回这些值：

```
sysdate   2007-09-23 21:30:23.00000
id        1
value     description
time1     2007-09-23 21:29:27.00000
time2     2007-09-23 21:29:27
```

下一查询访问同一表，使用 WHERE 子句中的 SYSDATE 作为 DAY 函数的一个参数：

```
SELECT *, DAY(time1) AS day FROM tab1
            WHERE DAY(time1) = DAY(SYSDATE);
```

在 2007 年 9 月 23 日，该查询可能返回这些值：

```
id        1
value     description
time1     2007-09-23 21:29:27.00000
time2     2007-09-23 21:29:27
day       23
```

仅 GBase 8s 支持 **SYSDATE**。除了它的名称和它的缺省精度之外，在本文档中 **CURRENT** 运算符的描述也适用于 **SYSDATE** 运算符。

在 Oracle 模式下使用 SYSDATE 运算符，**SYSDATE** 的返回精度是 YEAR TO SECOND。

下列示例中的 SQL 语句，在 Oracle 模式下使用 SYSDATE 运算符返回当前系统时间：

```
SET ENVIRONMENT SQLMODE 'oracle';
SELECT SYSDATE AS sysdate FROM DUAL;
```

该查询返回如下值：

```
SYSDATE    2007-09-23 21:29:27
```

当前版本 Oracle 兼容模式与 GBase 模式在切换过程中，未开启 USEOSTIME，SYSDATE 运算符由于精度取值的差异，Oracle 模式切换至 GBase 模式的转换处理如下：

- 切换模式后小于 1 秒内，再次调用 SYSDATE ，返回数值精度为 YEAR TO FRACTION(5)，并且读取系统时钟小数秒数值，例如可能返回值 2007-09-23 21:29:27.14871 ；
- 切换模式后大于 1 秒，再次调用 SYSDATE ，返回数值精度为 YEAR TO FRACTION(5)，并且仅读取系统时钟至整数秒，例如可能返回值 2007-09-23 21:29:27.00000 。

由于以上转换处理，建议用户在使用 SYSDATE 运算符时，尽量保持在统一模式下。

## SYSTIMESTAMP 运算符

在 Oracle 模式下，SYSTIMESTAMP 关键字返回数据库所在系统的系统日期，包括年、月、日、时、分、秒、小数秒，返回类型是 DATETIME 数据类型。默认输出格式为：DD-MM-YY HH:mm:ss.fffff；

日期时间输出格式可由 DBCENTURY 环境变量、DBDATE 环境变量、DBTIME 环境变量、GL_DATE 环境变量、GL_DATETIME 环境变量决定；

SYSTIMESTAMP 关键字可以指定参数 n，即 SYSTIMESTAMP(n)。参数 n 指定了小数秒的精度，取值范围为 1-5，默认为 5。当参数 n 超出精度范围时，报错：201 语法错误。

示例：

```
CREATE TABLE tab1 (
        id SERIAL,
        value CHAR(20),
        time1 DATETIME YEAR TO FRACTION(5) DEFAULT SYSTIMESTAMP,
        time2 DATETIME YEAR TO SECOND DEFAULT SYSTIMESTAMP YEAR TO
SECOND
        );
INSERT INTO tab1 VALUES (0, 'description', SYSTIMESTAMP, SYSTIMESTAMP);
```

下列查询访问在前面的示例中创建了的表：

```
SELECT SYSTIMESTAMP AS SYSTIMESTAMP, * FROM tab1;
```

当发出 INSERT 和 SELECT 语句时，结果对日期和时间是灵敏的，但在 2023 年 2 月 24 日该查询返回这些值：

```
SYSTIMESTAMP   2023-02-24 15:17:44.125
ID             1
VALUE          description
TIME1          2023-02-24 15:15:40.659
```

```
TIME2                2023-02-24 15:15:40
```

下一查询访问同一表，使用 WHERE 子句中的 SYSTIMESTAMP 作为 DAY 函数的一个参数：

```
SELECT *, DAY(time1) AS day FROM tab1
        WHERE DAY(time1) = DAY(SYSTIMESTAMP);
```

在 2023 年 2 月 24 日，该查询可能返回这些值：

```
ID          1
VALUE    description
TIME1     2023-02-24 15:15:40.659
TIME2     2023-02-24 15:15:40
DAY        24
```

## 文字的 DATETIME

文字的 DATETIME 指定 DATETIME 数据类型的值，包括它的限定的时间单位。

下列示例展示作为表达式的文字的 DATETIME：

```
SELECT DATETIME (2007-12-6) YEAR TO DAY FROM customer;
UPDATE cust_calls SET res_dtime = DATETIME (2008-07-07 10:40)
    YEAR TO MINUTE
    WHERE customer_num = 110
    AND call_dtime = DATETIME (2008-07-07 10:24) YEAR TO MINUTE;
SELECT * FROM cust_calls
     WHERE call_dtime
            = DATETIME (2008-12-25 00:00:00) YEAR TO SECOND;
```

要获取更多信息，请参阅 文字的 DATETIME。

## 文字的 INTERVAL

文字的 INTERVAL 指定 INTERVAL 数据类型的值，包括它的限定的时间单位。

下列每一示例使用文字的 INTERVAL 作为表达式：

```
INSERT INTO manufact VALUES ('CAT', 'Catwalk Sports',
INTERVAL (16) DAY TO DAY);
SELECT lead_time + INTERVAL (5) DAY TO DAY FROM manufact;
```

第二个示例将五天添加到从 **manufact** 表选择的每一 **lead_time** 的值。

要获取更多信息，请参阅 文字的 INTERVAL。

## UNITS 运算符

UNITS 运算符指定其精度仅包括一个时间单位的 INTERVAL 值。您可在 INTERVAL 或 DATETIME 值中增加或减少一个时间单位的算术表达式中使用 UNITS。

如果 ***num*** 运算对象不是整数，则当数据库服务器为该表达式求值时，将它截断到与指定的值相同的（或更接近于零的）完整数。

在下列示例中，第一个 SELECT 语句使用 UNITS 运算符来选择所有增加了五天的 **manufacturer.lead_time** 值。第二个 SELECT 语句找到放置了超过 30 天的所有呼叫。

如果 WHERE 子句中的表达式返回一大于 99（最大的天数）的值，则查询失败。最后的语句为 ANZE 制造商增加两天的交付时间：

```
SELECT lead_time + 5 UNITS DAY FROM manufact;
SELECT * FROM cust_calls WHERE (TODAY - call_dtime) > 30 UNITS DAY;
UPDATE manufact SET lead_time = 2 UNITS DAY + lead_time
        WHERE manu_code = 'ANZ';
```

## NEXTVAL 和 CURRVAL 运算符

您可在 SQL 语句中使用 **NEXTVAL** 或 **CURRVAL** 运算符来访问序列的值。您必须以在同一数据库中存在的序列对象的名称（或同义词）来限定 **NEXTVAL** 或 **CURRVAL**，使用格式 ***sequence*.NEXTVAL** 或 ***sequence*.CURRVAL**。表达式还可通过 ***owner*** 名称来限定 ***sequence***，就像在 **zelaine.myseq.CURRVAL** 中那样。您可指定 ***sequence*** 的标识符或有效的同义词，如果存在的话。

在符合 ANSI 的数据库中，如果您不是所有者，则您必须以其所有者的名称（***owner.sequence***）来限定 ***sequence*** 的名称。

要随同序列使用 **NEXTVAL** 或 **CURRVAL**，您必须在该序列上有 Select 权限或在数据库上有 DBA 权限。要获取更多关于序列级权限的信息，请参阅 GRANT 语句 语句。

**示例**

在下列示例中，假设当前没有其他用户正在访问该序列，且用户依顺序执行这些语句。

这些示例基于下列序列对象和表：

```
CREATE SEQUENCE seq_2
        INCREMENT BY 1 START WITH 1
        MAXVALUE 30 MINVALUE 0
        NOCYCLE CACHE 10 ORDER;
CREATE TABLE tab1 (col1 int, col2 int);
        INSERT INTO tab1 VALUES (0, 0);
```

您可在 INSERT 语句的 Values 子句中使用 NEXTVAL（或 CURRVAL），如下列示例所示：

```
INSERT INTO tab1 (col1, col2)
        VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL);
```

在前面的示例中，数据库服务器将增加的值（或该序列的第一个值，其为 1）插入到表的 **col1** 和 **col2** 列内。

您可在 UPDATE 语句的 SET 子句中使用 NEXTVAL（或 CURRVAL），如下列示例所示：

```
UPDATE tab1
```

```
SET col2 = seq_2.NEXTVAL
WHERE col1 = 1;
```

在前面的示例中，**seq_2** 序列的增加的值（其为 2）替代 **col2** 中 **col1** 等于 1 的值。

下列示例展示您可如何使用 SELECT 语句的 Projection 子句中的 NEXTVAL 和 CURRVAL：

```
SELECT seq_2.CURRVAL, seq_2.NEXTVAL FROM tab1;
```

在前面的示例中，数据库服务器从 **CURRVAL** 和 **NEXTVAL** 表达式返回增加的值的两行，3 和 4。对于 **tab1** 的第一行，数据库服务器为 **CURRVAL** 和 **NEXTVAL** 返回增加的值 3；对于 **tab1** 的第二行，它返回增加的值 4。

## *使用 NEXTVAL*

要首次访问序列，在您可引用 ***sequence*.CURRVAL** 之前，你必须引用 ***sequence*.NEXTVAL**。第一个对 **NEXTVAL** 的引用返回该序列的初始值。对 **NEXTVAL** 的每一后续的引用，都按照定义的 ***step*** 来增加该序列的值，并返回该序列的新的增加的值。

在单个 SQL 语句之内，您可对给定的序列仅增加一次。即使您在单个语句之内指定 ***sequence*.NEXTVAL** 多次，该序列也仅增加一次，因此，在同一 SQL 语句中 ***sequence*.NEXTVAL** 的每次发生都返回相同的值。

除了在同一语句之内多次发生的情况之外，每个 ***sequence*.NEXTVAL** 表达式增加该 ***sequence***，不管您随后是提交还是回滚当前事务。

如果您在最终回滚了的事务中指定 ***sequence*.NEXTVAL**，则可能跳过某些序列编号。

## *使用 CURRVAL*

任何对 **CURRVAL** 的引用都返回指定序列的当前值，这是您最后对 **NEXTVAL** 引用返回的值。在您以 **NEXTVAL** 创建新值之后，您可继续使用 **CURRVAL** 来访问那值，不管另一用户是否增加该序列。

如果在 SQL 语句中同时发生 ***sequence*.CURRVAL** 和 ***sequence*.NEXTVAL**，则该序列仅增加一次。在此情况下，每一 ***sequence*.CURRVAL** 和 ***sequence*.NEXTVAL** 表达式返回相同的值，不管 ***sequence*.CURRVAL** 和 ***sequence*.NEXTVAL** 在语句内的顺序。

## *对序列的并发访问*

序列总是生成数据库内的唯一值，即使当多个用户并发地引用同一序列，也觉察不到等待或锁定。当多个用户使用 **NEXTVAL** 来增加序列时，每一用户生成其他用户不可见的唯一值。

当多个用户并发地增加同一序列时，每一用户看到的值发生差异。例如，一个用户可能从序列生成一系列值，诸如 1、4、6 和 8，而另一用户从同一序列对象并发地生成值 2、3、5 和 7。

## *对序列运算符的限制*

NEXTVAL 和 CURRVAL 仅在 SQL 语句中有效，在 SPL 语句中不是直接有效的。（但可在 SPL 例程中使用使用 NEXTVAL 和 CURRVAL 的 SQL 语句。）下列限制适用于 SQL 语句中的这些运算符：

- 您必须有对*序列*的 Select 权限。
- 在 CREATE TABLE 或 ALTER TABLE 语句中，您不可在下列上下文中指定 **NEXTVAL** 或 **CURRVAL**：
  - 在列定义的 Default 子句中
  - 在检查约束的定义中。
- 在 SELECT 语句中，您不可在下列上下文中指定 **NEXTVAL** 或 **CURRVAL**：
  - 当使用 DISTINCT 关键字时，在 projection 列表中
  - 在 WHERE、GROUP BY 或 ORDER BY 子句中
  - 在子查询中
  - 当 UNION 运算符组合 SELECT 语句时。
- 在这些上下文中，你也不可指定 **NEXTVAL** 或 **CURRVAL**：
  - 在分片表达式中
  - 在对另一数据库中的远程序列对象的引用中。

## 文字的 Row

在 Literal Row 部分中描述命名的或未命名的 ROW 数据类型的值的文字表示的语法。下列示例展示作为表达式的文字的 row：

```
INSERT INTO employee VALUES
        (ROW('103 Baker St', 'San Francisco',
        'CA', 94500));

        UPDATE rectangles
        SET rect = ROW(8, 3, 7, 20)
        WHERE area = 140;

        EXEC SQL update table(:a_row)
        set x=0, y=0, length=10, width=20;

        SELECT row_col FROM tab_b
        WHERE ROW(17, 'abc') IN (row_col);
```

要了解求值为 ROW 数据类型的字段值的表达式的语法，请参阅 ROW 构造函数。

## 文字的集合

GBase 8s 支持内建的或用户定义的数据类型的值的文字表示的表达式。下列示例展示作为表达式的文字的集合：

```
INSERT INTO tab_a (set_col) VALUES ("SET{6, 9, 3, 12, 4}");
INSERT INTO TABLE(a_set) VALUES (9765);
```

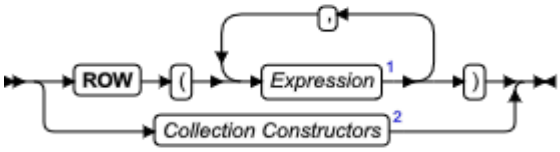
```
UPDATE table1 SET set_col = "LIST{3}";
SELECT set_col FROM table1   WHERE SET{17} IN (set_col);
```

要获取更多信息，请参阅 文字的集合。要了解元素值的语法，请参阅 集合构造函数。

## 构造函数表达式

***构造函数***是数据库服务器用来创建特定的数据类型的实例的函数。数据库服务器支持 ROW 构造函数和集合构造函数。

构造函数表达式




## ROW 构造函数

您使用 ROW 构造函数来生成 ROW 类型列的值。

假设您创建下列命名的 ROW 类型以及包含命名的 ROW 类型 row_t 和未命名的 ROW 类型的表：

```
CREATE ROW TYPE row_t ( x INT, y INT);
        CREATE TABLE new_tab
        (
        col1 row_t,
        col2 ROW( a CHAR(2), b INT)
        );
```

当您定义列作为命名的 ROW 类型或未命名的 ROW 类型时，您必须使用 ROW 构造函数来生成 ROW 类型列的值。要为命名的 ROW 类型或未命名的 ROW 类型创建值，您必须完成下列步骤：

- 以 ROW 关键字开始该表达式。
- 为每一 ROW 类型的字段指定值。
- 将以逗号分隔的字段值的列表括在圆括号内。

每一字段的值的格式必须与将它指定到的那个 ROW 字段的数据类型相兼容。

您可使用任何类型的表达式作为带有 ROW 构造函数的值，包括文字、函数和变量。下列示例展示使用不同类型的带有 ROW 构造函数的表达式来指定值：

```
ROW(5, 6.77, 'HMO')
ROW(col1.lname, 45000)
ROW('john davis', TODAY)
ROW(USER, SITENAME)
```

下列语句使用带有 ROW 构造函数的字面的数值和引用的字符串来将值插入到 new_tab 表的 col1 和 col2 内：

```
INSERT INTO new_tab
        VALUES
```

```
(
ROW(32, 65)::row_t,
ROW('CA', 34)
);
```

当您使用 ROW 构造函数来生成命名的 ROW 类型的值时，您必须显式地将 ROW 强制转型为适当的命名的 ROW 类型。 强制转型有必要生成命名的 ROW 类型的值。要将 ROW 值强制转型为命名的 ROW 类型，您可使用强制转型运算符（:: ）或 CAST AS 关键字，如下例所示：

```
ROW(4,5)::row_t
CAST (ROW(3,4) AS row_t)
```

您可使用 ROW 构造函数来在 INSERT、UPDATE 和 SELECT 语句中生成 ROW 类型值。在下一示例中，SELECT 语句的 WHERE 子句指定强制转型为类型 person_t 的一个 ROW 类型值：

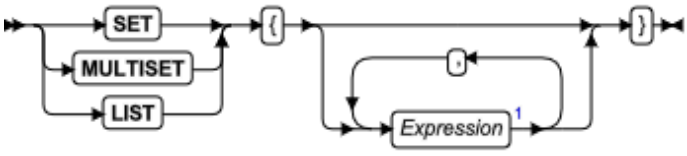
```
SELECT * FROM person_tab
WHERE col1 = ROW('charlie','hunter')::person_t;
```

要获取更多关于在 INSERT 和 UPDATE 语句中使用 ROW 构造函数的信息，请参阅本文档中的 INSERT 和 UPDATE 语句。要获取关于命名的 ROW 类型的信息，请参阅 CREATE ROW TYPE 语句。要获取关于未命名的 ROW 类型，请参阅 *《GBase 8s SQL 指南：参考》* 中的 ROW 数据类型的讨论。

## 集合构造函数

使用集合构造函数来为集合列指定值。

集合构造函数




您可在 SELECT 语句的 WHERE 子句中以及 INSERT 语句的 VALUES 子句中使用集合构造函数。您还可将集合构造函数传递给 UDR。

此表区分您可构造的集合的类型。

| 关键字 | 描述 |
| --- | --- |
| SET | 表明带有下列性质的集合元素：<br>• 该集合必须包含唯一的值。<br>• 元素有与他们相关联的特定的顺序。 |
| MULTISET | 表明带有下列性质的元素的集合：<br>• 该集合可包含重复的值。<br>• 元素没有与他们相关联的特定的顺序。 |
| LIST | 表明带有下列性质的元素的集合： |

| 关键字 | 描述 |
| --- | --- |
| | • 该集合可包含重复的值。<br>• 元素有顺序位置。 |

集合的元素类型可为任何内建的或扩展的数据类型。您可随同集合构造函数使用任何类型的表达式，包括文字、函数和变量。

当您使用带有表达式的列表的集合构造函数时，数据库服务器将每一表达式求值为它等同的文字，并使用文字的值来构造该集合。

您可以一系列大括号（ **{ }** ）指定空集合。

集合的元素不可为 NULL。如果集合元素求值为 NULL 值，则数据库服务器返回错误。

每一表达式的元素类型必须全都是完全相同的数据类型。要实现这一点，将整个集合构造函数表达式强制转型为集合类型，或将个别的元素表达式强制转型为同一类型。如果数据库服务器不可确定集合类型与元素类型是同类的，则集合构造函数返回错误。在主变量的情况下，在客户端声明主变量的元素类型的绑定时刻作出此决定。

当集合的有些元素是 VARCHAR 数据类型但其他的长度不长于 32,765 字节时，可发生对此限制的例外。在此，集合构造函数可将 CHAR(***n***) 类型指定给所有元素，***n*** 是以字节计的最长的元素的长度。（但是，请参阅 集合数据类型 了解基于此例外的示例，用户通过显式的强制转型为 LVARCHAR 数据类型来避免固定长度的 CHAR 元素。）

## *集合构造函数的示例*

下列示例展示您可以不同的表达式构造集合，如果结果值是同一数据类型的话：

```
CREATE FUNCTION f (a int) RETURNS int;
        RETURN a+1;
        END FUNCTION;
        CREATE TABLE tab1 (x SET(INT NOT NULL));
        INSERT INTO tab1 VALUES
        (
        SET{10,
        1+2+3,
        f(10)-f(2),
        SQRT(100) +POW(2,3),
        (SELECT tabid FROM systables WHERE tabname = 'sysusers'),
        'T'::BOOLEAN::INT}
        );
        SELECT * FROM tab1 WHERE
        x=SET{10,
        1+2+3,
        f(10)-f(2),
        SQRT(100) +POW(2,3),
```

```
(SELECT tabid FROM systables WHERE tabname = 'sysusers'),
'T'::BOOLEAN::INT}
};
```

这假设存在从 BOOLEAN 到 INT 的强制转型。（要了解对指定集合值的更限制性的语法，请参阅文字的集合。）

## NULL 关键字

在您可指定值的大部分上下文中，NULL 关键字是有效的。然而，它指定的内容没有任何值（或未知的或遗失的值）。

NULL 关键字


在 SQL 内，关键字 NULL 是访问 NULL 值的唯一语法机制。NULL 不等同于零，也不等同于任何特定的值。在升序的 ORDER BY 操作中，NULL 值排在任何非 NULL 值之前；在降序排序中，NULL 值跟在任何非 NULL 值之后。在 GROUP BY 操作中，所有 NULL 值都组在一起。（如果它们包括遗失的值或未知的值，则这样的分组可能在逻辑上是各种各样的。）

在表达式的语法上下文中，关键字 NULL 是全局的符号，意味着它的引用作用域是全局的。

每种数据类型，不论内建或用户定义的，都可以代表 NULL 值。本版本 GBase 8s 支持以下两种形式的 NULL 查询：

- 在投影列表中直接包含 NULL 关键字，无需强制转型。
- 在投影列表中包含 NULL**::*datatype*** 形式的强制转型表达式，其中 ***datatype*** 是数据库服务器已知的任何数据类型。

GBase 8s 在一般表达式中支持已归类的 NULL 关键字。在某些情境下单独的 NULL 将导致 -201 语法错误。因此，如果 NULL 定义为列名称或过程名称，那么它必须以通过表别名被引用。否则，将返回 -201 语法错误。以下示例和结果可总结该行为：

```
create table tab1 (a int, null int);
create table tab2 (a int, b int);
```

表 1. NULL 行为

| 语句 | 结果 |
|---|---|
| select null from tab1 where a = 1 | -201 语法错误 |
| select * from tab1 where null = a | -201 语法错误 |
| select * from tab1 where tab1.null = a | 有效的语法 |
| select * from tab1 where a = null | -201 语法错误 |
| select * from tab2 where a = null | -201 语法错误 |
| select * from tab2 where null = a | -201 语法错误 |

| | |
|---|---|
| select * from tab2 where null = a | -201 语法错误 |
| select NULL::int from tab1 | 有效的语法 |
| select NULL from tab1 | 有效的语法 |
| select NULL::int from tab1 | 有效的语法 |
| select 1 + NULL::int from tab1 | 有效的语法 |
| select 1 + NULL::int from tab2 | 有效的语法 |
| select NULL::int + 1 from tab1 | 有效的语法 |

GBase 8s 禁止重新定义 NULL，因为允许这样定义会限制 NULL 关键字的全局作用域。为此，任何限制全局作用域或重新定义关键字 NULL 的作用域的机制都会在语法上禁用涉及 NULL 值的任何强制转型表达式。您必须确保关键字 NULL 的发生在所有表达式上下文中收到它的全局作用域。

例如，请考虑下列 SQL 代码：

```
CREATE TABLE newtable
        (
        null int
        );

SELECT null, null::int FROM newtable;
```

CREATE TABLE 语句是有效的，因为列标识符具有限定到表定义的引用的作用域；仅可在表的作用域内访问它们。

然而，在该示例中的 SELECT 语句引起一些语法的多义性。出现在 projection 列表中的标识符 null 引用全局的关键字 NULL 吗？它引用在 CREATE TABLE 语句中声明了的列标识符 null 吗？

- 如果将标识符 **null** 解释作为列名称，则带有 NULL 关键字的强制转型表达式的全局作用域将会受限。
- 如果将标识符 **null** 解释作为 NULL 关键字，则 SELECT 语句必须为 **null** 的发生生成语法错误，因为 NULL 关键字仅可作为强制转型表达式出现在 projection 列表中。

下列形式的 SELECT 语句是有效的，因为以表名称限定 newtable 的 NULL 列：

```
SELECT newtable.null, null::int FROM newtable;
```

在有名为 null 的变量的 SPL 例程的上下文中，会出现更多语法的多义性。示例如下：

```
CREATE FUNCTION nulltest() RETURNING INT;
            DEFINE a INT;
            DEFINE null INT;
            DEFINE b INT;
            LET a = 5;
            LET null = 7;
            LET b = null;
            RETURN b;
```

```
        END FUNCTION;

EXECUTE FUNCTION nulltest();
```

当在 DB-Access 中执行前面的函数时，在 LET 语句的表达式中，创建标识符 **null** 作为关键字 NULL。该函数返回 NULL 值，而不是 7。

使用 null 作为 SPL 例程的变量会限制在 SPL 例程体中对 NULL 值的使用。因此，前面的 SPL 代码是无效的，并导致 GBase 8s 返回下列错误：

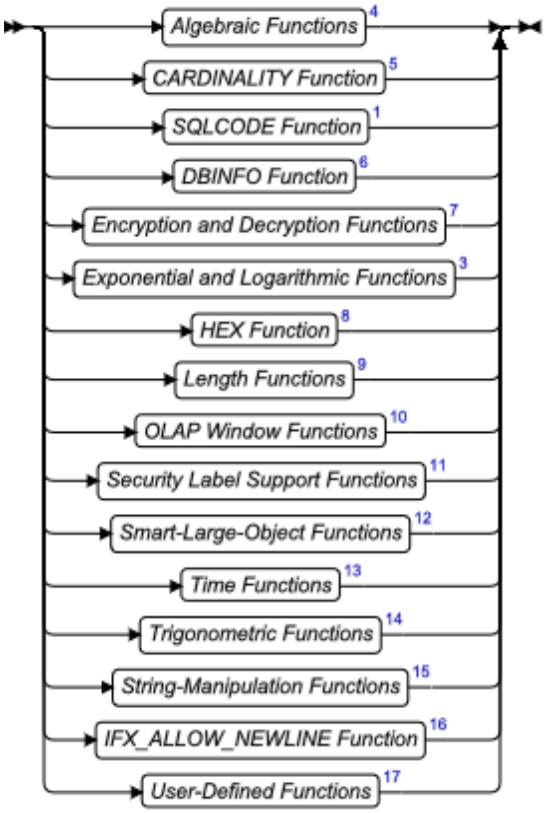
-947 Declaration of an SPL variable named 'null' conflicts with SQL NULL value.

在 ESQL/C 中，如果有 SELECT 语句会返回 NULL 值的可能性，则您应使用指示符变量。

## 函数表达式

函数表达式可从内建的 SQL 函数或从用户定义的函数返回一个或多个值，如下图所示。

函数表达式




下列示例展示函数表达式：

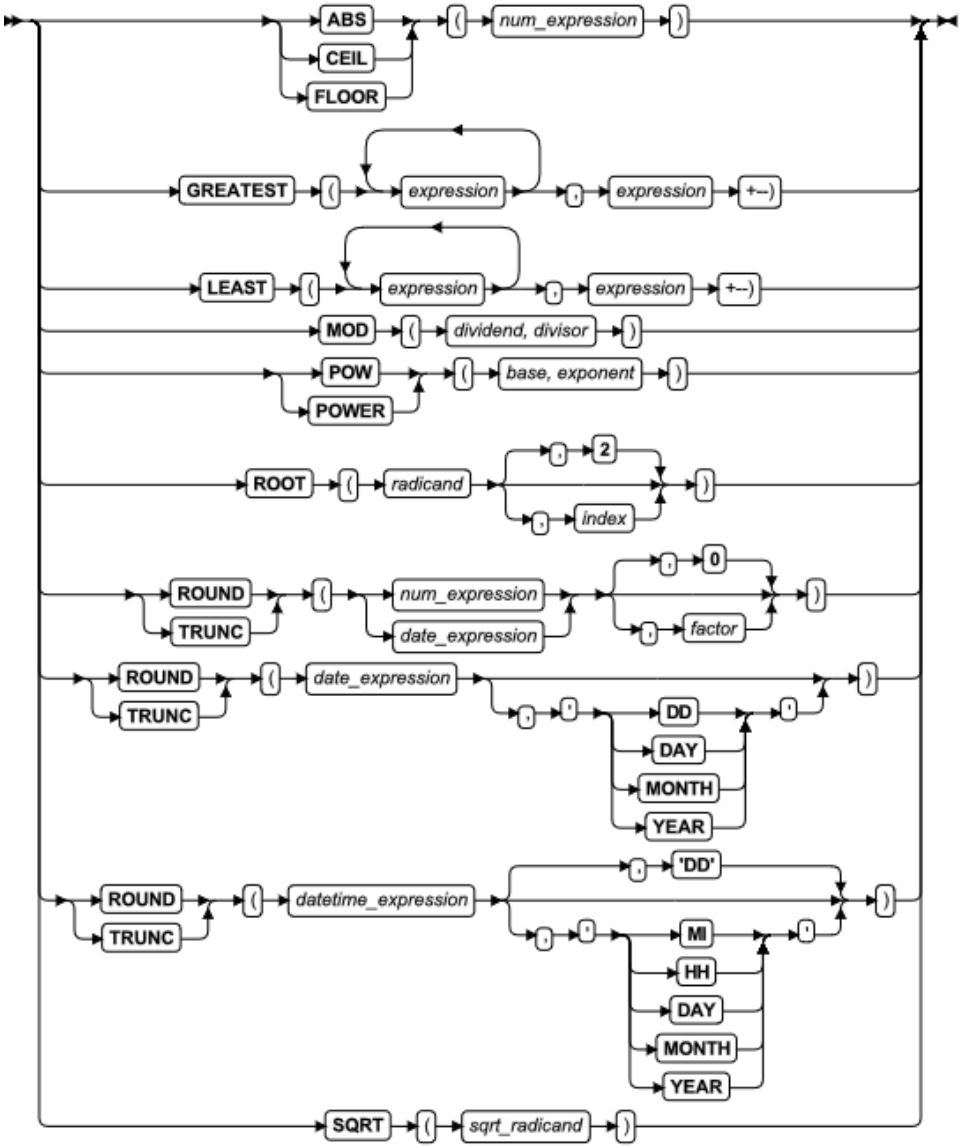
```
EXTEND (call_dtime, YEAR TO SECOND)

HEX (LENGTH(123))

MDY (12, 7, 1900 + cur_yr)

TAN (radians)
```

```
DATE (365/2)

ABS (-32)

LENGTH ('abc') + LENGTH (pvar)

EXP (3)

HEX (customer_num)

MOD (10,3)
```

## 代数函数

代数函数采用一个或多个数值数据类型的参数。除了支持的数值参数之外，CEIL 和 FLOOR 函数还可采用可转换为 DECIMAL 值的字符串参数，且 ROUND 和 TRUNC 函数还可采用 DATE 或 DATETIME 参数。

代数函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *base* | 要升至 *exponent* 中指定的幂的值 | 必须返回实数值 | 表达式 |
| *date_expression* | 求值为（或强制转型为）DATE 值的表达式 | 必须返回 DATE 值 | 表达式 |
| *datetime_expression* | 求值为（或强制转型为）DATETIME 值的表达式 | 必须返回 DATETIME 值 | 表达式 |
| *dividend* | 要被 *divisor* 除的值 | 实数值 | 表达式 |
| *divisor* | 用来除 *dividend* 的值 | 非零的实数值 | 表达式 |
| *exponent* | *base* 要升到的幂 | 实数值 | 表达式 |
| *factor* | 在返回的值中以零替换 | 取值范围为 +32 | 精确数 |

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| | 的有效数字位数。默认值为四舍五入的或截断的第一个参数的整数部分。 | 至 -32 的整数。正值或无符号值适用于小数点的右边，负值适用于左边。 | 值 |
| *index* | 要抽取的根。缺省值为 2。 | 非零的实数值 | 表达式 |
| *num_expression* | 求值为(或强制转型为）数值值的表达式 | 实数值 | 表达式 |
| *radicand* | 要返回其根的值 | 实数值 | 表达式 |
| *sqrt_radicand* | 带有实平方根的数值 | 非负的实数值 | 表达式 |

## *ABS 函数*

ABS 函数返回它的数值参数的绝对值，返回的数据类型与它的参数相同。下列示例中的查询返回所有以现金（+）或作为商店信用卡（-）支付了的 ship_charge 大于 $20 的所有订单。

```
SELECT order_num, customer_num, ship_charge
        FROM orders WHERE ABS(ship_charge) > 20;
```

## *CEIL 函数*

**CEIL** 函数将数值表达式，或可转换为 DECIMAL 数据类型的字符串作为它的参数，并返回大于或等于它的单个参数的最小整数的 DECIMAL(32) 表示。

下列查询返回 33 作为大于或等于 CEIL 参数 32.3 的最小整数：

```
SELECT CEIL(32.3) FROM systables WHERE tabid = 1;
```

下一示例返回 -32 作为大于或等于 CEIL 参数 -32.3 的最小整数：

```
SELECT CEIL(-32.3) FROM systables WHERE tabid = 1;
```

## *FLOOR 函数*

**FLOOR** 函数将数值表达式，或可转换为 DECIMAL 数据类型的字符串作为它的参数，并返回小于或等于它的单个参数的最大整数的 DECIMAL(32) 表示。

下列查询返回 32 作为小于或等于 FLOOR 参数 32.3 的最大整数：

```
SELECT FLOOR(32.3) FROM systables WHERE tabid = 1;
```

下一示例返回 -33 作为小于或等于 FLOOR 参数 -32.3 的最大整数：

```
SELECT FLOOR(-32.3) FROM systables WHERE tabid = 1;
```

这些示例说明当 FLOOR 和 CEIL 函数有非零小数部分的相同参数时，它们如何提供差值为 1 的上界和下界。对于整数参数，FLOOR 和 CEIL 返回与它们参数相同的 DECIMAL(32) 表示。

## *GREATEST 函数*

**GREATEST** 函数返回表达式的列表中的最大值。

此函数的参数必须是求值为相兼容的数据类型的以逗号分隔的表达式。

这是 **GREATEST** 函数的语法：

GREATEST 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 可比较其值的表达式 | 数据类型不可为集合或大对象。 | 表达式 |

这些参数必须是相兼容的数据类型。不支持复合的数据类型，或 BYTE、TEXT、BLOB、CLOB 对象，或基于任何这些数据类型的 DISTINCT 类型的参数。您指定作为 **GREATEST** 函数的参数的任何用户定义的数据类型必须执行 **greaterthan( )** 函数。

如果必要，数据库服务器将指定的 ***expression*** 参数转换为返回的值的数据类型。由 ***expression*** 的所有运算对象确定此返回数据类型，可兼容性规则与 CASE 表达式一致。

**GREATEST** 函数的返回值是它的最大参数值。如果一个或多个参数求值为 NULL，则结果为 NULL。如果 **GREATEST** 是用于比较 DATE 或 DATETIME 值，则返回值是最近的日期。

假设表 **T1** 包含三列 **C1**、**C2** 和 **C3**，其值为 1、7 和 4。下列查询返回值 7：

```
SELECT GREATEST (C1, C2, C3) FROM T1;
```

然而，如果列 **C3** 有 NULL 值，而不是 4，则同一查询返回 NULL 值。

## *LEAST 函数*

LEAST 函数返回一组值中的最小值。

LEAST 函数


参数必须是相兼容的，且每一参数都必须是表达式，表达式的返回值的数据类型不可为复合的类型、BYTES、TEXT、BLOB、CLOB，或基于任何这些类型的用户定义的类型。用户定义的类型必须实现对函数 **lessthan()** 的支持，以便使用 LEAST 函数。如有必要，将选择的参数转换为结果的数据类型。由所有的运算对象确定结果数据类型，且可兼容性规则与 CASE 表达式保持一致。

该函数的结果是最小的参数值。如果至少一个参数可为空，则结果为空。如果 LEAST 用于比较日期，则返回值是最早的日期。

假设表 T1 包含三列 C1、C2 和 C3，取值为 1、7 和 4。查询返回值 1。

```
SELECT LEAST (C1, C2, C3) FROM T1
```

如果列 **C3** 有值 NULL 而不是 4，则同一查询返回 NULL 值。

## *MOD 函数*

**MOD** 函数以两个实数值运算对象作为参数，并返回第一个参数（***被除数***）的整数部分除以第二个参数（***除数***）的整数部分的整数商的余数。返回的值为 INT 数据类型（或对于 INT 范围之外的余数，为 INT8）。丢弃商和余数的任何小数部分。***除数*** 不可为 0。因此，MOD (x,y) 返回 y (modulo x)。请确保收到该结果的任何变量都是可存储返回的值的数据类型。

此示例测试的是当前日期是否在 30 天记账周期之内：

```
SELECT MOD(TODAY - MDY(1,1,YEAR(TODAY)),30) FROM orders;
```

## *POW 函数*

**POW** 函数求得它的第一个数值参数 ***base*** 的第二个数据值参数 ***exponent*** 的幂。返回的值是 FLOAT 数据类型。

下列示例从 circles 表返回所有行，其中 radius 列值表示小于 1,000 平方单位的面积，使用范围为 4 的 pi 的近似值：

```
SELECT * FROM circles WHERE (3.1416 * POW(radius,2)) < 1000;
```

函数标识符 **POWER®** 是 **POW** 的同义词。

要使用自然对数的基数 *e*，请参阅 EXP 函数。

## *ROOT 函数*

**ROOT** 函数从它的第一个数值表达式参数 ***radicand*** 抽取正实数根值，返回为 FLOAT 数据类型。

如果您指定第二个数值参数作为 ***index***（不可为零），则返回值的 ***index*** 幂等于（在四舍五入误差范围内）***radicand*** 参数。如果仅提供 ***radicand*** 参数，则 2 是缺省的 ***index*** 值。您不可指定零作为 ***index*** 的值。

在下列示例中的第一个 SELECT 语句，使用缺省的 *index* 值 2，返回文字数值 9 的正平方根。第二个示例返回文字数值 64 的立方根。

```
SELECT ROOT(9) FROM angles;          -- 9 的平方根
SELECT ROOT(64,3) FROM angles;       -- 64 的立方根
```

调用仅带有单个参数的 ROOT 等同于调用 SQRT 函数。

## *SQRT 函数*

**SQRT** 函数返回它的参数的正值平方根，该参数必须为非负的数值表达式。

下列示例为 angles 表的每一行返回 9 的平方根：

```
SELECT SQRT(9) FROM angles;
```

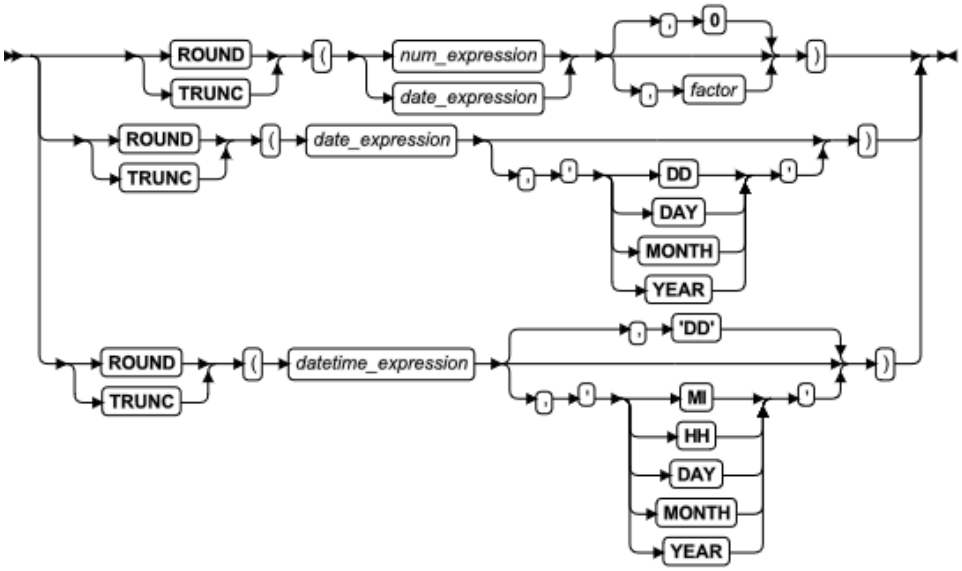
**SQRT** 函数等同于 ROOT(x)，**ROOT** 函数的第二个参数的缺省值 2 指定该指数。

## *ROUND 函数*

**ROUND** 函数可降低它的第一个数值、MONEY、DATE 或 DATETIME 参数的精度，并返回四舍五入了的值。如果第一个参数不是数值、MONEY 值或时间点，则必须将它强制转型为数值的、MONEY、DATE 或 DATETIME 数据类型。

下图展示 ROUND 和 TRUNC 代数函数的语法，它们支持相同的语法。然而，由于它们的语义不同，它们可从相同的参数列表返回不同的值。 仅 ROUND 可返回大于它的第一个参数的绝对值。

ROUND 和 TRUNC 代数函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *date_expression* | 求值为(或强制转型为) DATE 值的表达式 | 必须返回 DATE 值 | 表达式 |
| *datetime_expression* | 求值为(或强制转型为) DATETIME 值的表达式 | 必须返回 DATETIME 值 | 表达式 |
| *factor* | 在返回的值中以零替代的有效数字的数目。缺省的是返回四舍五入的或截断的第一个参数的整数部分。 | 取值范围为 +32 至 -32 的整数。正值或无符号值适用于小数点的右边，而负值适用于左边。 | 精确数值 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *num_expression* | 求值为(或强制转型为)数值值的表达式 | 实数 | 表达式 |

**用法**

ROUND 函数与 TRUNC 函数相似，其语法如上所示。然而，ROUND 的不同之处在于它如何处理精度之内小于最小有效数字或时间单位的它的第一个参数的任何部分，它的显式的或缺省的第二个参数指定该精度。

- 如果此部分的绝对值等于或大于该精度内最小单位的一半，则那个数字或时间单位的值为由 ROUND 返回的值中增加 1。然而，如果这部分小于一个单位的一半，则丢弃它，仅返回指定的或缺省的精度之内第一个参数的数值或时间单位。

  也就是说，如果第一个参数大于零，则

  - **ROUND** 函数舍去在第二个参数的精度之内小于最小有效数字或时间的一半单位的它的第一个参数的任何部分，
  - 但向上舍入等于或大于半个单位的第一个参数的任何部分。

  例如，ROUND(3.5,0) = 4 且 ROUND(3.4,0) = 3。

  但如果第一个参数小于零，则

  - ROUND 函数向上舍入在第二个参数的精度之内的小于最小有效数字或时间单位的半个单位的它的第一个参数的任何部分，
  - 但舍去等于或大于半个单位的第一个参数的任何部分。

  例如，ROUND(-3.5,0) = -4 而 ROUND(-3.4,0) = -3。

- 相反地，TRUNC 函数以零替代数值表达式的小于指定精度的任何数值。对于 DATE 或 DATETIME 表达式，TRUNC 替换小于指定格式字符串的任何时间单位，以 1 替换 month 或 day 时间单位，或以零替换小于 day 的时间单位。

**ROUND** 函数可接受可选的第二个参数，其指定返回的值的精度。第二个参数的语法和语义依赖于第一个字符是否为数值表达式、DATETIME 表达式或 DATE 表达式。

**对数值和 MONEY 值的四舍五入**

- 当第一个参数是数值表达式时，返回的值是 DECIMAL，且第二个参数可为取值范围从 -32 至 +32 （包含 -32 和 +32）的整数，指定返回的值的最后有效数字（相对于小数点）的位置。当第一个参数是数值时，如果您省略 factor 规范，则 ROUND 返回舍入到个位或单位位置的第一个参数的整数值。

  正的数字值指定舍入到小数点的右边；负的数值值指定舍入到小数点的左边，如 图 1 所示：

  *图: 负的、零和正的舍入因子的示例*


下列示例使用带有列表达式作为它的第一个参数且没有第二个参数的 ROUND 函数，因此，数值表达式被舍入到范围零。此查询返回订单号和其总价（四舍五入到缺省的个位）等于 $124.00 的项的四舍五入的总价。

```
SELECT order_num , ROUND(total_price) FROM items
WHERE ROUND(total_price) = 124.00;
```

如果您使用 MONEY 数据类型作为 ROUND 函数的参数，且四舍五入到显式的或缺省的个位，则返回的值以 .00 表示小数部分。下列示例中的 SELECT 语句四舍五入 125.46 和 MONEY 列值。该查询返回 125 和 items 表中每一行的 xxx.00 形式的四舍五入的价格。

```
SELECT ROUND(125.46), ROUND(total_price) FROM items;
```

**DATE 和 DATETIME 值的四舍五入**

- 当 ROUND 的第一个参数为 DATETIME 表达式时，返回的值是 DATETIME YEAR TO MINUTE 数据类型，且第二个参数必须为在返回的值中指定最小有效时间单位的引用的字符串。如果您省略第二个参数，则缺省的格式字符串为 'DD'，以四舍五入到 00:00 的小时和分钟指定最近的一天。下列格式字符串作为第二个参数是有效的：

表 1. ROUND 函数的 DATETIME 参数的格式字符串

| 格式字符串 | 对返回的 DATETIME 值的影响 |
|---|---|
| 'YEAR' | 舍入到最近一年之初，以六月 30 日之后的日期舍入到下一年。*month*、*day*、*hour* 和 *minute* 值舍入为 -01-01 00:00。 |
| 'MONTH' | 舍入到最近一月之初。将 15 日以后的日期舍入到下一月。*day*、*hour* 和 *minute* 值舍入为 01 00:00。 |
| 'DD' | 舍入到最近一天之初（00:00 = midnight）。将中午 12:00 之后的 DATETIME 舍入到下一天。 |
| 'DAY' | 舍入到最近的星期天之初。周三、周四、周五或周六的日期舍入到下一星期天。 |
| 'HH' | 舍入到最近一小时之初。以 *minute*:*second* 晚于 29:59 的时间值舍入到下一小时。分钟舍入为零。 |
| 'MI' | 舍入到最近一分钟之初。以 *second* 晚于 30 的时间值舍入到下一分钟。 |

如果您在初始的 DATETIME 表达式参数之后省略格式字符串规范，则返回的值是将第一个参数舍入到最近一天的值，就如同您已指定了 'DD' 作为格式字符串。

下列示例在 SELECT 语句中使用带有返回 DATETIME YEAR TO FRACTION(5) 值的列表达式的 **ROUND** 函数。在这些查询中，表 **mytab** 仅有单个行，且在那行中，**mytab.col_dt** 的值是 2012-12-07 14:30:12.12300。

下列查询指定 'YEAR' 作为 DATETIME 格式字符串：

```
SELECT ROUND(col_dt, 'YEAR') FROM mytab;
```

返回的值为 2013-01-01 00:00。

下一查询与前面的查询相似，但将返回的值强制转型为 DATE 数据类型：

```
SELECT ROUND(col_dt, 'YEAR')::DATE FROM mytab;
```

返回的值为 01/01/2013。

此示例指定 'MONTH' 作为 DATETIME 格式字符串：

```
SELECT ROUND(col_dt, 'MONTH') FROM mytab;
```

返回的值为 2012-12-01 00:00。

此示例将 DATETIME 表达式舍入到 YEAR TO HOUR 精度：

```
SELECT ROUND(col_dt, 'HH') FROM mytab;
```

返回的值为 2012-12-07 15:00。

- 如果第二个参数是在返回的值中指定最小时间单位的引用字符串，则当第一个参数为 DATE 表达式时，返回的值也是 DATE 数据类型。这些是与舍入 DATETIME 值相同的格式字符串，除了 'HH' 和 'MI' 不是有效的 DATE 值之外。对于舍入 DATE 参数，没有缺省的格式字符串。

要返回格式化的 DATE 值，您必须指定下列引用字符串之一作为 **ROUND** 函数的第二个参数：

表 2. ROUND 函数的 DATE 参数的格式字符串

| 格式字符串 | 对返回的 DATE 值的影响 |
|---|---|
| 'YEAR' | 四舍五入到最近一年之初。一月 30 日之后的日期舍入到下一年。*month* 和 *day* 值均舍入为 01。 |
| 'MONTH' | 四舍五入到最近一月之初。15 日之后的日期舍入到下一月。返回的 *day* 值为 01。 |
| 'DD' | 返回第一个 *date_expression* 参数的 DATE 值。 |
| 'DAY' | 将该值舍入到最近的星期天。如果第一个参数为星期天，则返回那个日期。周三、周四、周五和周六的日期舍入到下一星期天。 |

当第一个参数为 DATE 数据类型时，如果您未指定格式字符串作为第二个参数，则缺省情况下没有格式字符串生效。不发出错误，但将第一个参数作为求值为整数的数值表达式来处理，而不作为 DATE 值。GBase 8s 在内部将 DATE 值存储为从 1899 年 12 月 31 日以来的整数计数。对于 21 世纪中的日期，等同于 DATE 值的整数是 5 位整数，取值范围大约在 37,000 与 74,000 之间。

例如，查询 SELECT ROUND(TODAY) FROM systables 不为 DATE 表达式提供格式字符串，且如果在 2012 年 4 月 1 日发出该查询，则返回整数 40999。

如果您应用数值格式规范作为第二个参数，则非负的数值对 DATE 值不起作用，但下列示例将返回值的最后两个数字舍入为零：

```
SELECT ROUND(TODAY, -2) FROM systables;
```

在 2012 年 4 月 1 日，上述查询会返回整数值 40900。

在下一天，2012 年 4 月 2 日，同一查询会返回整数值 41000。

对于像 41000 这样的整数格式日期是有用的应用，您可使用 'YEAR'、'MONTH'、'DAY' 或 'DD' 格式字符串作为 ROUND 函数的第二个参数来防止将 DATE 参数处理成 一个数值表达式。在 2012 年 4 月 1 日，下列查询返回 DATE 值 04/01/2012，如果 MDY4/ 是 DBDATE 环境变量设置的话：

```
SELECT ROUND(TODAY, 'DD') FROM systables WHERE tabid = 1;
```

在下列示例中，在 2012 年 4 月 3 日（星期二）发出一查询：

```
SELECT ROUND(TODAY, 'DAY') FROM mytab;
```

返回的值为 03/31/2012，当前的日期四舍五入到最近的星期天。

如果您正在使用主变量来在动态的 SQL 中存储四舍五入了的时间点值，且在准备时刻不知道第一个参数的数据类型，则 GBase 8s 假设 ROUND 函数的第一个参数是 DATETIME 数据类型，并返回 DATETIME YEAR TO MINUTE 四舍五入的值。在执行时刻，则准备该语句之后，如果为该主变量提供 DATE 值，则发出错误 -9750。要防止发生此错误，您可通过使用强制转型为主变量指定数据类型，如此程序片断中所示。

```
sprintf(query1, ",
"select round( ?::date, 'DAY') from mytab");
EXEC SQL prepare selectq from :query;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;
EXEC SQL fetch select_cursor into :var_date_output;
```

要了解可为内建的按时间顺序排列的数据类型指定显示和数据条目格式的 GBase 8s 环境变量之中的优先顺序，请参阅主题 DATE 和 DATETIME 格式规范的优先顺序。

## *TRUNC 函数*

通过返回截断的值，**TRUNC** 函数可降低它的第一个数值的、DATE 或 DATETIME 参数的精度。如果第一个参数既不是数值也不是时间点，则必须将它强制转型为数值、DATE 或 DATETIME 数据类型。

通过返回截断的值，**TRUNC** 函数可降低它的第一个数值的、DATE 或 DATETIME 参数的精度。如果第一个参数既不是数值也不是时间点，则必须将它强制转型为数值、DATE 或 DATETIME 数据类型。

TRUNC 函数与 ROUND 函数相似，但它截断（而不是舍入到最近的整数）第一个参数中小于它的第二个参数指定的精度之内的最小有效数字或时间单位的任何部分。

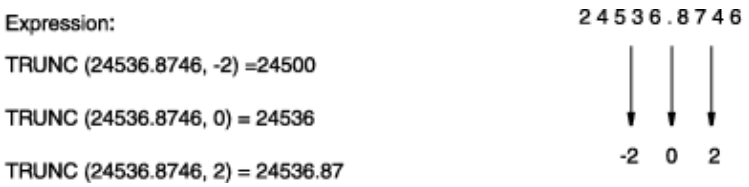
- 对于数值表达式，**TRUNC** 以零替换小于指定精度的任何数字。
- 对于 DATE 或 DATETIME 表达式，**TRUNC** 替换小于格式规范的任何时间单位，以 1 替换 ***month*** 或 ***day*** 时间单位，或以 0 替换小于 ***day*** 的时间单位。

TRUNC 函数可接受指定返回的值的精度的可选的第二个参数。

- 当第一个参数为数值的表达式时，第二个参数必须为取值范围从 -32 至 +32（包含-32 和 +32）的整数，指定返回的值的最后有效数字（相对于小数点）的位置。当第一个参数为数值时，如果您省略 factor 规范，则 TRUNC 返回截断到个位或单位位置的第一个参数的值。

  正的数字值指定截断到小数点的右边；负的数字值指定截断到左边，如下图所示。

  *图: 负的、零和正的截断因子的示例*




  下列示例在 SELECT 语句中以返回数值值的列表达式调用 TRUNC 函数。此语句显示订单号以及其总价（截断到缺省的个位小数位置）等于 $124.00 的项的被截断的总价。

```
SELECT order_num , TRUNC(total_price) FROM items
WHERE TRUNC(total_price) = 124.00;
```

  如果在一个指定个位的 TRUNC 函数调用中，如果 MONEY 数据类型是参数，则在返回的值中小数部分成为 .00。例如，下列 SELECT 语句截断 125.46 和 MONEY 列值。它为 items 表中的每一行返回 125 以及形如 xxx.00 的截断的价格。

```
SELECT TRUNC(125.46), TRUNC(total_price) FROM items;
```

- 当 TRUNC 的第一个参数为 DATETIME 表达式时，第二个参数必须为指定返回值中最小有效时间单位的引用的字符串。仅下列格式的字符串是有效的第二个参数：

  表 1. TRUNC 函数的 DATETIME 参数的格式字符串

| 格式字符串 | 对返回的值的影响 |
|---|---|
| 'YEAR' | 截断到年初。*month*、*day*、*hour* 和 *minute* 值截断到 01-01 00:00。 |
| 'MONTH' | 截断到该月的第一天之初。*hour* 和 *minute* 值截断到 00:00。 |
| 'DD' | 阶段到同一天之初（00:00 = 午夜）。 |

| 格式字符串 | 对返回的值的影响 |
| --- | --- |
| 'DAY' | 如果第一个参数是星期天，则返回那天的午夜（00:00）。对于该星期的任何其他天，返回前一个星期天的午夜。 |
| 'HH' | 截断到该小时之初。*minute* 值截断为零。 |
| 'MI' | 截断到最近的分钟之初。对于所有这些格式字符串，丢弃小于 *minute* 的时间单位。 |

如果您在初始的 DATETIME 表达式参数之后省略格式字符串规范，则返回的值是将第一个参数截断到天的值，就如同您指定了 'DD' 作为格式字符串一样。

下列示例在 SELECT 语句中以返回 DATETIME YEAR TO FRACTION(5) 值的列表达式调用 **TRUNC** 函数。在这些示例中，表 **mytab** 仅有单个行，且在那行中 **mytab.col_dt** 的值是 2006-12-07 14:30:12.12300。

此查询指定 'YEAR' 作为 DATETIME 格式字符串：

```
SELECT TRUNC(col_dt, 'YEAR') FROM mytab;
```

返回的值为 2006-01-01 00:00。

下一查询与前一查询相似，但将截断的值强制转型为 DATE 数据类型：

```
SELECT TRUNC(col_dt, 'YEAR')::DATE FROM mytab;
```

返回的值为 01/01/2006。

此示例指定 'MONTH' 作为 DATETIME 格式字符串：

```
SELECT TRUNC(col_dt, 'MONTH') FROM mytab;
```

返回的值为 2006-12-01 00:00。

下列示例将 DATETIME 表达式截断到 YEAR TO HOUR 精度：

```
SELECT TRUNC(col_dt, 'HH') FROM mytab;
```

返回的值为 2006-12-07 14:00。

- 当第一个参数为 DATE 表达式时，第二个参数通常应为指定返回的值中最小时间单位的引用的字符串。 除了 'HH' 和 'MI' 不是有效的日期之外，这些是与截断 DATETIME 值相同的格式字符串，且对于截断 DATAE 表达式参数，没有缺省的格式字符串。

  要返回格式化的 DATE 值，您必须使用下列引用的字符串之一作为 **TRUNC** 函数的第二个参数：

表 2. TRUNC 函数的 DATE 参数的格式字符串

| 格式字符串 | 对返回的值的影响 |
| --- | --- |
| 'YEAR' | 截断到该年之初。*month* 和 *day* 值都为 01。 |
| 'MONTH' | 截断到该月之初。*day* 值为 01。 |
| 'DD' | 返回第一个 *date_expression* 参数的 DATE 值。 |

| 格式字符串 | 对返回的值的影响 |
| --- | --- |
| 'DAY' | 如果第一个参数是星期天，则返回那天。对于该星期的任何其他天，返回前一星期天的日期。 |

当第一个元素为 DATE 数据类型时，如果您未指定格式字符串作为第二个参数，则没有格式字符串作为缺省值生效。不发出错误，但将第一个参数作为求值为整数的数值表达式处理，而不是作为 DATE 值。 GBase 8s 将 DATE 值作为自从 1899 年 12 月 31 日以来的整数天数在内部存储。

例如，查询 SELECT ROUND(TODAY) FROM systables 未为 DATE 表达式提供格式字符串，如果在 2008 年 4 月 1 日提交该查询，则返回整数 39538。

如果您应用数值的格式规范作为第二个参数，则非负的数值对 DATE 值不起作用，但下列示例将返回的值的最后两位数值舍入为零：

```
SELECT TRUNC(TODAY, -2) FROM systables;
```

对于类似于 39500 这样的整数日期不适用的应用，请使用 'YEAR'、'MONTH'、'DAY' 或 'DD' 格式字符串作为 TRUNC 函数的第二个参数，来防止将 DATE 表达式作为数值表达式来处理。在 2008 年 4 月 1 日，如果 MDY4/ 是 DBDATE 环境变量的设置，下列查询返回 DATE 值 04/01/2008：

```
SELECT TRUNC(TODAY, 'DD') FROM systables;
```

如果您正在使用主变量来在动态的 SQL 中存储截断的时间点值，且在准备时刻不知道第一个参数的数据类型，则 GBase 8s 假设 DATETIME 数据类型是 TRUNC 函数的第一个参数，并返回 DATETIME YEAR TO MINUTE 截断的值。在执行时刻，在准备该语句之后，如果为主变量提供 DATE 值，则发出错误 -9750。要防止发生此错误，您可通过使用强制转型为主变量指定数据类型，如此程序片断中所示：

```
sprintf(query2, "%s",
"select trunc( ?::date, 'DAY') from mytab");
EXEC SQL prepare selectq from :query2;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;
EXEC SQL fetch select_cursor into :var_date_output;
```

要了解 GBase 8s 环境变量之中的优先顺序，这些环境变量可为内建的按时间排序的数据类型指定显示和数据条目格式，请参阅主题 DATE 和 DATETIME 格式规范的优先顺序.

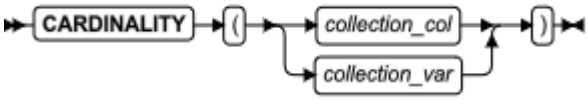
请注意，**TRUNC** 函数名称是基于英文单词 "truncate"，这与在 SQL 的 TRUNCATE 语句中它的含义不同。**TRUNC** 以另一个更小精度或相同精度的值替代它的第一个参数的值。TRUNCATE 语句从数据库表删除所有行，而不删除表模式。

## CARDINALITY 函数

**CARDINALITY** 函数返回集合列（SET、MULTISET、LIST）中元素的数目。

CARDINALITY 函数有下列语法。

CARDINALITY 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *collection_col* | 现有的集合列 | 必须声明为集合数据类型 | 标识符 |
| *collection_var* | 主或程序集合变量 | 必须声明为集合数据类型 | 特定于语言 |

假设 set_col SET 列包含下列值：

{3, 7, 9, 16, 0}

下列 SELECT 语句返回 5 作为 set_col 列中元素的数目：

```
SELECT CARDINALITY(set_col) FROM table1;
```

如果集合包含重复的元素，**CARDINALITY** 为每一个别的元素计数。

## SQLCODE 函数（SPL）

**SQLCODE** 函数不用参数，但将当前 SPL 例程已执行了的最近执行的（不论静态的还是动态的）SQL 语句的 **sqlca.sqlcode** 值返回到它的调用上下文。仅在游标的上下文中使用 **SQLCODE**。

SQLCODE


您可在 SPL 例程内的表达式中使用 **SQLCODE** 来标识动态游标的状态。在错误处理中以及在诸如确定查询或函数调用是否尚未返回行的上下文中，或当游标已达到了活动集的最后行时，或当 SPL 程序控制应从循环中退出时要标识其他条件，此内建的函数是有用的。

下列 SPL 程序片断说明使用 **SQLCODE** 来检测 WHILE 循环内游标的活动集的末尾。

```
CREATE PROCEDURE ...
        ...
        DEFINE myc1 ...
        ...
        PREPARE p FOR "SELECT c1 FROM t1";
        DECLARE cur FROM s;
        OPEN cur;

        FETCH cur INTO myc1;
        WHILE (SQLCODE != 100)
        FETCH cur INTO myc1;
        -- process myc1
        ...
        END WHILE;
```

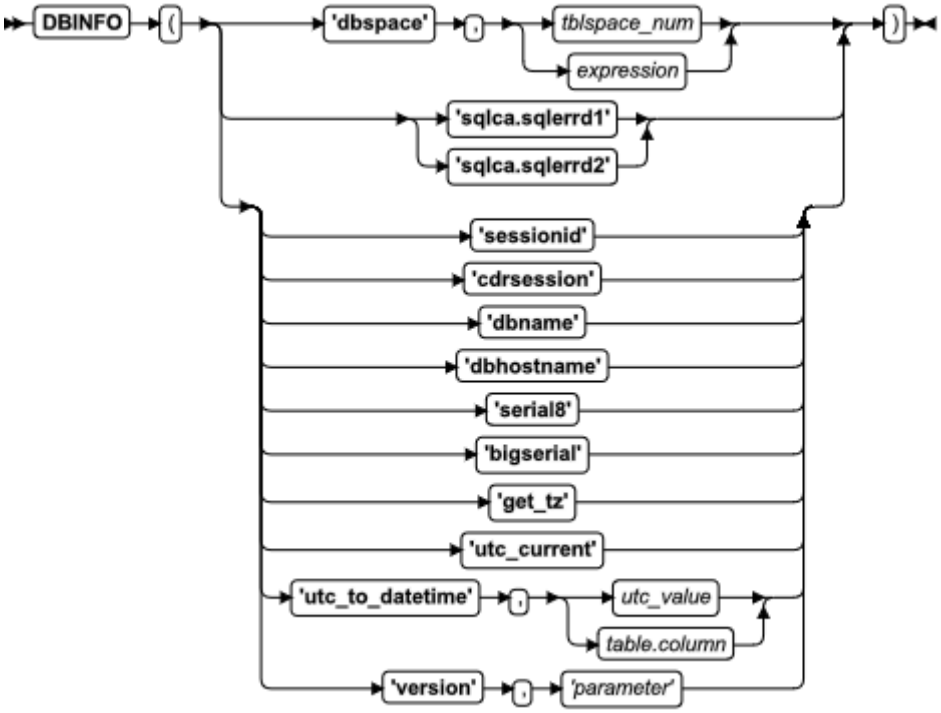
```
END PROCEDURE;
```

在以 ESQL/C 编写的 UDR 中不需要 **SQLCODE**，其通过“动态的 SQL”的 GET DIAGNOSTICS 语句以及有其他机制直接访问“SQL 通信区域”（**SQLCA**）。如果内建的 **SQLCODE** 函数的调用上下文不在 SPL 例程中，则数据库服务器发出错误。

## DBINFO 函数

下图展示 DBINFO 函数的语法。

DBINFO 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | *table* 中的列名称 | 在 *table* 中必须存在 | 标识符 |
| *expression* | 求值为 *tblspace_num* 的表达式 | 可包含列名称、SPL 变量、主变量或子查询，但必须返回数值值 | 表达式 |
| *parameter* | 指定返回 *version* 字符串的哪一部分的引用的字符串 | 要了解有效的 *parameter* 值，请参阅 使用 'version' 选项 | 请参阅 **限制**栏。 |
| *table* | 要显示 dbspace 名称或包含 UTC 值的整数 *column* 列的表。 | 必须与查询的 FROM 子句中表的名称相匹配 | 标识符 |
| *tblspace_num* | 表的 tblspace 号（分 | 在数据库的 systables | 精确数 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 区号） | 表的 partnum 列中必须存在 | 值 |
| *utc_value* | 要转换为 DATETIME 等价的 UTC 值 | 必须为数值表达式，求值为从 1970-01-01 00:00:00+00:00 以来的秒数 | 表达式，精确数值 |

## *DBINFO 选项*

**DBINFO** 函数实际上是返回关于数据库的不同类型信息的函数集。要调用每一函数，请在 **DBINFO** 关键字之后指定特定的选项。您可在 SQL 语句之内和 UDR 之内的任何地方使用任何 **DBINFO** 选项。

下表展示数据库的类别以及 GBase 8s 可通过有效的 **DBINFO** 选项检索的数据库服务器信息。

- **参数**栏展示以圆括号限定的每一有效的 **DBINFO** 选项的参数列表。
- **返回的信息**栏展示**参数**选项检索的数据库信息的类型。
- **页**栏展示您可找到关于**参数**选项的更多信息的位置。

| 参数 | 返回的信息 | 页 |
| --- | --- | --- |
| ('dbhostname') | 客户端应用连接到的数据库服务器的主机名称 | 使用 'dbhostname' 选项 |
| ('dbname') | 客户端应用连接到的数据库的标识符 | 使用 'dbname' 选项 |
| ('dbspace' *tblspace_num*) | 与 tblspace 编号相对应的 dbspace 的名称 | 使用 ('dbspace', tblspace_num) 选项 |
| ('get_tz') | 会话的时区，**$TZ**，如同通过客户端作为字符串指定的那样。 | 使用 'get_tz' 选项 |
| ('serial8') | 插入在表中的最后的 SERIAL8 值 | 使用 'serial8' 和 'bigserial' 选项 |
| ('bigserial') | 插入在表中的最后的 BIGSERIAL 值 | 使用 'serial8' 和 'bigserial' 选项 |
| ('sessionid') | 当前会话的会话 ID 编号 | 使用 'sessionid' 选项 |

| | | |
|---|---|---|
| ('cdrsession') | 线程是否正在执行 Enterprise Replication 操作 | 使用 'cdrsession' 选项 |
| ('sqlca.sqlerrd1') | 插入在表中的最后的 SERIAL 值 | 使用 'sqlca.sqlerrd1' 选项 |
| ('sqlca.sqlerrd2') | 通过 SELECT、INSERT、DELETE、UPDATE、EXECUTE PROCEDURE 和 EXECUTE FUNCTION 语句处理的行的数目 | 使用 'sqlca.sqlerrd2' 选项 |
| ('utc_current') | 当开始执行 SQL 语句时，当前的 UTC 时间值（作为一个从 1970-01-01 00:00:00+00:00 以来的秒的整数值）。 | 使用 'utc_current' 选项 |
| ('utc_to_datetime', *table.column*) | 对应于包含 UTC 时间值（作为一个从 1970-01-01 00:00:00+00:00 以来的秒的整数值）的指定的整数列的 DATETIME 值。 | 使用 'utc_to_datetime' 选项 |
| ('utc_to_datetime', *utc_value*) | 对应于指定的 UTC 时间值（作为一个从 1970-01-01 00:00:00+00:00 以来的秒的整数值）的 DATETIME 值。 | 使用 'utc_to_datetime' 选项 |
| ('version', '*parameter*') | 客户端应用连接到的数据库服务器的类型及其发行版本。(如果 *parameter* 没有为版本信息指定格式，则调用 DBINFO 失败并报错。) | 使用 'version' 选项 |

## *使用 ('dbspace', tblspace_num) 选项*

'dbspace' 选项返回包含对应于 tblspace 编号的 dbspace 的名称的字符串。您必须提供附加的参数，或 *tblspace_num*，或求值为 *tblspace_num* 的表达式。下列示例使用 'dbspace' 选项。首先，它查询 systables 系统目录表来确定表客户的 *tblspace_num*，然后它执行该函数来确定 *dbspace* 名称。

```
SELECT tabname, partnum FROM systables
```

```
        where tabname = 'customer';
```

如果查询返回一个分区编号 1048892，则您将那个值插入到第二个参数内来找到包含 customer 表的那个 dbspace，如下例所示：

```
SELECT DBINFO ('dbspace', 1048892) FROM systables
        where tabname = 'customer';
```

如果您想要知道其 dbspace 名称的表是分片的，则你必须查询 **sysfragments** 系统目录表来找到每一表分片的 tblspace 编号。然后您必须在单独的 **DBINFO** 查询中提供每一 tblspace 编号来找到跨分片的表的所有 dbspace。

### *使用 'sqlca.sqlerrd1' 选项*

'sqlca.sqlerrd1' 返回提供插入到表内的最后的 serial 值的单个整数。要确保有效的结果，请紧跟在将带有 serial 值的单个行插入到表内的单 SELECT 语句之后使用此选项。

**提示：** 要获取插入到表内的最后的 SERIAL8 值的值，请使用 DBINFO 的 'serial8' 选项。要获取更多信息，请参阅 使用 'serial8' 和 'bigserial' 选项。

下列示例使用 'sqlca.sqlerrd1' 选项：

```
EXEC SQL create table fst_tab (ordernum serial, partnum int);
EXEC SQL create table sec_tab (ordernum serial);
EXEC SQL insert into fst_tab VALUES (0,1);
EXEC SQL insert into fst_tab VALUES (0,4);
EXEC SQL insert into fst_tab VALUES (0,6);
EXEC SQL insert into sec_tab values (dbinfo('sqlca.sqlerrd1'));
```

此示例将包含主键 serial 值的一行插入到 **fst_tab** 表内，然后使用 DBINFO 函数来将同一 serial 值插入到 **sec_tab** 表内。DBINFO 函数返回的值是被插入到 **fst_tab** 内的最后一行的 serial 值。

由于 SQLCA 结构不记录通过触发器插入的 serial 值，因此您不可以 'sqlca.sqlerrd1'、'bigserial' 或 'serial8' 选项调用 DBINFO 函数来返回触发器的活动插入的 serial 值。

要获取更多关于“SQL 通信区域”（SQLCA）数据结构的信息（sqlca.sqlerrd1 在其内是一个字段），请参阅 *GBase 8s SQL 教程指南*。

### *使用 'sqlca.sqlerrd2' 选项*

**'sqlca.sqlerrd2'** 选项返回提供 SELECT、INSERT、DELETE、UPDATE、EXECUTE PROCEDURE 和 EXECUTE FUNCTION 语句处理了的行的数目的单个整数。要确保有效的结果，请在 SELECT、EXECUTE PROCEDURE 和 EXECUTE FUNCTION 语句已执行完成之后使用此选项。此外，当您在游标内使用此选项时，要确保有效的结果，请确保在关闭游标之前取回所有行。

下列示例展示 SPL 例程，该例程使用 ’sqlca.sqlerrd2’ 选项来确定从表删除的行的数目：

```
CREATE FUNCTION del_rows (pnumb INT)
        RETURNING INT;

        DEFINE nrows INT;

        DELETE FROM fst_tab WHERE part_number = pnumb;
        LET nrows = DBINFO('sqlca.sqlerrd2');
        RETURN nrows;
END FUNCTION;
```

要获取更多关于“SQL 通信区域”（SQLCA）数据结构的信息（**sqlca.sqlerrd2** 在其内是一个字段），请参阅 *GBase 8s SQL 教程指南*。

## *使用 'sessionid' 选项*

**DBINFO** 函数的 **'sessionid'** 选项返回您的当前会话的会话 ID。当客户端应用连接到数据库服务器时，数据库服务器启动与客户端的会话，并为该客户端指定一会话 ID。该会话 ID 用作客户端与数据库服务器之间给定连接的唯一的标识符。

数据库服务器在称为***会话控制块***的共享内存中的数据结构中存储该会话 ID 的值。给定会话的会话控制块还包括用户 ID、客户端的进程 ID、主机计算机的名称和各种状态标志。

当您指定 **'sessionid'** 选项时，数据库服务器从会话控制块检索您的当前会话的会话 ID，并将此值作为整数返回给您。**sysmaster** 数据库中的某些“系统监视接口”（SMI）表包括会话 ID 的列，因此您可使用 **DBINFO** 函数获取了的会话 ID 来从这些 SMI 表抽取关于您自己的会话的信息。要获取关于会话控制块的更多信息，请参阅 *GBase 8s 管理员指南*。要获取关于 **sysmaster** 数据库和 SMI 表的更多信息，请参阅 *GBase 8s 管理员参考手册*。

在下列示例中，用户在 SELECT 语句中指定 DBINFO 函数来获取当前会话 ID 的值。用户产生针对 systables 系统目录表的查询，并使用 WHERE 子句来将查询结果限定为单个行。

```
SELECT DBINFO('sessionid') AS my_sessionid
        FROM systables
        WHERE tabname = 'systables';
```

在前面的示例中，SELECT 语句针对 systables 系统目录表查询。然而，您可通过针对任何系统目录表或数据库中的用户表进行查询，获取当前会话的会话 ID。 例如，您可输入下列查询来获取您的当前会话的会话 ID：

```
SELECT DBINFO('sessionid') AS user_sessionid
        FROM customer
        WHERE customer_num = 101;
```

您不仅可在 SQL 语句中使用 DBINFO ’sessionid’ 选项，还可在 SPL 例程中使用。 下列示例展示返回当前会话 ID 的值的 SPL 函数来调用程序或例程：

```
CREATE FUNCTION get_sess()
        RETURNING INT;
```

```
        RETURN DBINFO('sessionid');
END FUNCTION;
```

### *使用 'cdrsession' 选项*

DBINFO() 函数的 'cdrsession' 选项检测是否执行 INSERT、UPDATE 或 DELETE 语句作为复制的事务的一部分。

您可能想要升级触发器、存储过程或用户定义的例程来采取不同的活动，这依赖于是否执行事务作为 Enterprise Replication 的一部分。如果线程执行的数据库操作是 Enterprise Replication apply 或 sync 线程，则 DBINFO() 函数的 'cdrsession' 选项返回 1；否则，该函数返回 0。

下列示例展示使用 'cdrsession' 选项的 SPL 函数，来确定线程是否正在执行 Enterprise Replication 操作：

```
CREATE FUNCTION iscdr ()
        RETURNING int;

        DEFINE iscdrthread int;
        SELECT DBINFO('cdrsession') into iscdrthread
        from systables where tabid = 1;
        RETURN iscdrthread;
END FUNCTION
```

### *使用 'dbname' 选项*

您可使用 **'dbname'** 选项来检索当前数据库的名称。此选项返回客户端会话当前连接到的数据库的标识符。

在下列示例中，用户在 SELECT 语句中输入 DBINFO 的 'dbname' 选项来检索 DB-Access 连接到的数据库的名称：

```
SELECT DBINFO('dbname')
        FROM systables
        WHERE tabid = 1;
```

下表展示此查询的结果。

| (constant) |
| --- |
| stores_demo |

### *使用 'dbhostname' 选项*

您可使用 **'dbhostname'** 选项来检索数据库客户端连接到的数据库服务器的主机名称。

此选项检索数据库服务器正运行在其上的计算机的物理计算机名称。

在下列示例中，用户在 SELECT 语句中的 DBINFO 的 'dbhostname' 选项来检索 DB-Access 连接到的数据库服务器的主机名称：

```
SELECT DBINFO('dbhostname')
        FROM systables
        WHERE tabid = 1;
```

下列表格展示此查询的结果。

| (constant) |
| --- |
| rd_lab1 |

### *使用 'version' 选项*

您可使用 **DBINFO** 函数的 **'version'** 选项来从消息日志检索关于针对客户端应用正在运行的数据库服务器的类型和发布版本的信息。

您必须在 **'version'** 选项之后包括 ***'parameter'*** 规范来表明您想要检索的版本字符串的哪一部分。

如果在 'version' 之后，您指定 'full' 作为 *parameter* 值，则 DBINFO 返回完整的版本字符串，其与 oninit 实用程序的 -V 选项显示的值相同。 下列表格罗列 DBINFO 的所有有效的 *parameter* 参数，其可检索关于数据库服务器的版本信息：

- **参数**栏展示每一有效的 **DBINFO ( 'version', '*parameter*')** 组合的以圆括号限定的参数列表。
- **返回的版本字符串的部分**栏展示每一**参数**列表返回的版本字符串的哪一部分。
- **返回的值的示例**栏展示 **Arguments** 选项的每一 ***parameter*** 值返回的示例。

每一示例返回完整的版本字符串 GBase 8s Version 11.50.UC6 的一部分。

| 参数 | 返回的版本字符串的部分 | 返回的值的示例 |
| --- | --- | --- |
| ('version', 'server-type') | 数据库服务器的类型 | GBase 8s |
| ('version', 'major') | 当前数据库服务器版本的主要版本号 | 11 |
| ('version', 'minor') | 当前数据库服务器版本的次要版本号 | 50 |
| ('version', 'os') | 在版本字符串内的操作系统标识符：<br>T = 32 位 Windows™ 平台<br>U = 运行在 32 位操作系统上的 UNIX™ 32 位 | U |

| 参数 | 返回的版本字符串的部分 | 返回的值的示例 |
|---|---|---|
| | H = 运行在 64 位操作系统上的 UNIX 32 位<br>F = 所有 64 位平台 | |
| ('version', 'level') | 当前数据库服务器版本的临时发布级别 | C6 |
| ('version', 'full') | 会由 oninit -V 返回的完整的版本字符串 | GBase 8s, Version 11.50.UC6 |

**Important：** 不是所有 UNIX 环境都适用前面的表格中操作系统（os）的字长描述。例如，某些 U 版本可运行在 64 位操作系统上。类似地，有些 F 版本可运行在支持 64 位应用的带有 32 位内核的操作系统上。

下列示例展示如何使用 SELECT 语句中的 **DBINFO** 的 **'version'** 选项来检索 DB-Access 客户端连接到的数据库服务器的主要版本号：

```
SELECT DBINFO('version', 'major')
        FROM systables
        WHERE tabid = 1;
```

下列表格展示此查询的结果：

| (constant) |
|---|
| 7 |

## *使用 'version_gbase' 选项*

DBINFO ('version', parameter)用于返回客户端应用连接到的数据库服务器的确切版本的全部或一部分，本次在保留'version'选项的基础上，增加了'version_gbase'选项。DBINFO ('version_gbase ', parameter)中参数parameter支持：'server-type'、'major'、'minor'、'full'。

| 参数 | 返回的版本字符串的部分 | 返回的值的示例 |
|---|---|---|
| ***('version_gbase ', server-type)*** | 产品名称 | 使用 select dbinfo('version_gbase', 'server-type') from sysdual 语句进行查询，返回结果格式为：产品名称，示例：GBase8s。 |
| ***('version_gbase ', major)*** | 主要版本号 | 使用 select dbinfo('version_gbase', 'major') from sysdual 语句进行查询，返回结果格式为：主版本号，示例：V8.8。 |

| 参数 | 返回的版本字符串的部分 | 返回的值的示例 |
| --- | --- | --- |
| ***('version_gbase ', minor)*** | 次要版本号 | 使用 select dbinfo('version_gbase', 'minor') from sysdual 语句进行查询，系统显示数据库版本号，返回结果格式为：版本级别+_+版本号+专用版本代号及版次+_+迭代序号（送测序号）+补充版本标识+_+6 位哈希 ID，示例：AEE_3.0.0G5_1P20200918_1fc8fc。 |
| ***('version_gbase ', full)*** | 完整的版本字符串 | 使用 select dbinfo('version_gbase', 'full') from sysdual 语句进行查询，系统显示数据库版本号，返回结果格式为：产品名称+主版本号+_+版本级别+_+版本号+专用版本代号及版次+_+迭代序号（送测序号）+补充版本标识+_+6 位哈希 ID，示例：GBase8sV8.8_AEE_3.0.0G5_1P20200918_1fc8fc。 |

## *使用 'serial8' 和 'bigserial' 选项*

'bigserial' 和 'serial8' 选项分别返回指定插入到了表内的最后的 SERIAL8 或 BIGSERIAL 值的单个整数。要确保有效的结果，请紧跟在插入 SERIAL8 或 BIGSERIAL 值的 INSERT 语句之后使用此选项。

**提示：** 要获取被插入到表内的最后 SERIAL 值的值，请使用 DBINFO( ) 的 'sqlca.sqlerrd1' 选项。要获取更多信息，请参阅 使用 'sqlca.sqlerrd1' 选项。

下列示例使用 'serial8' 选项：

```
EXEC SQL CREATE TABLE fst_tab
        (ordernum SERIAL8, partnum INT);
        EXEC SQL CREATE TABLE sec_tab (ordernum SERIAL8);

        EXEC SQL INSERT INTO fst_tab VALUES (0,1);
        EXEC SQL INSERT INTO fst_tab VALUES (0,4);
        EXEC SQL INSERT INTO fst_tab VALUES (0,6);

        EXEC SQL INSERT INTO sec_tab
        SELECT dbinfo('serial8')
        FROM fst_tab WHERE partnum = 6;
```

此示例将包含主键 SERIAL8 值的行插入到 **fst_tab** 表内，并使用 DBINFO 函数来将相同的 SERIAL8 值插入到 **sec_tab** 表内。DBINFO 函数返回的值是被插入到 **fst_tab** 内的最后一行的 SERIAL8 值。在随后行中的子查询包含 WHERE 子句，因此返回单个值。

SQLCA 结构不记录由触发器插入的 serial 值。您不可以 'bigserial' 选项调用 DBINFO 函数来返回由表上的触发器的触发器活动直接地插入了的最近的 BIGSERIAL 值（视图上的 INSTEAD OF 触发器的也不可返回）。出于同样的原因，DBINFO ('serial8') 函数不可返回由表上的触发器插入了的 SERIAL8 值，由视图上的 INSTEAD OF 触发器也不可返回。

### *使用 'get_tz' 选项*

**'get_tz'** 选项返回展示当前会话的时区的 **$TZ** 字符串。

下列示例在 stores_demo 数据库的 cust_calls 表的查询中使用 'get_tz' 选项：

```
EXEC SQL select first call_dtime, dbinfo('get_tz')
        from cust_calls where customer_num = 106;
```

此示例返回会话时区的字符串值以及 **customer_num** 值为 106 的 **cust_calls** 表中的第一个 **call_dtime** 值。

### *使用 'utc_current' 选项*

**'utc_current'** 选项返回“全球标准时间”（UTC）的当前值，作为展示在 1970-01-01 00:00:00+00:00 与当前 SQL 语句开始执行时刻之间已消耗了的秒数的整数值。

“全球的时间”（UT）从地球的旋转计算持续的秒数。UTC 不是这样，UTC 基于高精度原子时钟使用固定长度的秒数。

由于地球逐渐地减小的旋转速度的变化，在 UTC 中一次又一次地引入闰年***跳跃秒数***来减少与 UT 时间的差异。在缺省情况下，GBase 8s 忽略在 DATETIME 和 INTERVAL 算术中的跳跃秒数。然而，当采用跳跃秒数的操作系统支持 GBase 8s 时，在操作系统为跳跃秒数调整系统时钟之后，在后续的 DATETIME 和 INTERVAL 操作中反映跳跃秒数。

### *使用 'utc_to_datetime' 选项*

考虑到数据库服务器的时区，DBINFO 函数的 'utc_to_datetime' 选项将 UTC 秒数返回到服务器会生成的 DATETIME 值，如果 UNIX™ time( ) 系统调用返回了秒参数的值的话。

'utc_to_datetime' 选项将它的最后的参数强制转型为 DATETIME 值，该参数必须是表示“全球标准时间”（UTC）的数值表达式。如果这求值为带有小数部分的数值，则忽略任何小数的秒。

在下列第一个示例中，最后一个参数是表示为字面整数的 UTC 值。在第二个示例中，最后一个参数是指定存储 UTC 值的整数列的列表达式。在两个示例中，DBINFO 都将 UTC 值强制转型为数据库服务器的时区中的 DATETIME 值：

```
DBINFO ('utc_to_datetime', 1299912999 )
DBINFO ('utc_to_datetime', timesheet.utc_checkin )
```

如果最后一个参数的值是负的，则该函数从较早的 UNIX epoch 返回 DATETIME 值，如下例中所示：

```
SELECT DBINFO("utc_to_datetime", -2134567890.91234)
    FROM 'sysmaster:"gbasedbt".sysdual';
```

此查询返回 DATETIME 值 1902-05-12 08:28:30。

这些示例时间都假设服务器在特定的时区中。下列查询返回四个 DATETIME 值：

```
SELECT
    DBINFO('utc_to_datetime', -32767) AS min_smallint,
    DBINFO('utc_to_datetime', +32767) AS max_smallint,
    DBINFO('utc_to_datetime', 1299912999),
    DBINFO("utc_to_datetime", -2134567890.91234)
    FROM 'sysmaster:"gbasedbt".sysdual';
```

这些是从 United States Pacific 时区中的服务器返回的 DATETIME 值：

```
1969-12-31 06:53:53     1970-01-01 01:06:07
2011-03-11 22:56:39     1902-05-12 01:28:30
# Server running in TZ=US/Pacific
```

这些是从 UTC0 时区中的服务器从同一查询返回的 DATETIME 值：

```
1969-12-31 14:53:53     1970-01-01 09:06:07
2011-03-12 06:56:39     1902-05-12 08:28:30
# Server running in TZ=UTC0
```

请注意，第三个 DBINFO 结果中的 DAY 组件对于 United States Pacific 时区与对于 UTC0 时区是不同的，因为这那两个时区之间有 8 小时的偏移量。

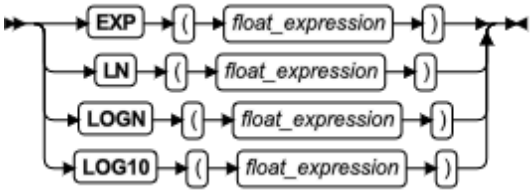
对于时间中的点，数据库服务器时区可类似地影响从其他表达式的返回值，诸如 CURRENT、SYSDATE 和 TODAY，其 DATETIME YEAR TO SECOND 或 DATE 表示依赖于服务器的时区。

## 指数和对数函数

指数和对数函数至少有一个参数，且返回 FLOAT 数据类型。

指数和对数函数有下列语法。

指数和对数函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *float_expression* | EXP、LN、LOGN 或 LOG10 函数的一个参数。要了解这些函数中 *float_expression* 的含义，其参阅后面几页上每一函数的单独的标题。 | 该域是实数集，且范围是正实数集 | 表达式 |

## *EXP 函数*

**EXP** 函数返回数值表达式的指数。

下列示例为 angles 表的每一行返回 3 的指数：

```
SELECT EXP(3) FROM angles;
```

对于此函数，基数始终为 *e*，自然对数的基数，如下例所示：

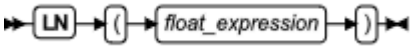
```
e=exp(1)=2.718281828459
```

当您想要使用自然对数的基数作为基数值时，请使用 **EXP** 函数。如果您想要指定特定的值来升至特定的幂，其参阅 POW 函数。

## *LN 函数*

**LN** 函数是 **LOGN** 函数的别名，返回数值参数的自然对数。此值与指数值相反。

LN 函数




下列查询为 history 表中的每一行返回 population 的自然对数：

```
SELECT LN(population) FROM history WHERE country='US' ORDER BY date;
```

## *LOG10 函数*

LOG10 函数返回以 10 为基数的对数。下列示例为 travel 表的每一行返回距离的以 10 为基数的对数：

```
SELECT LOG10(distance) + 1 digits FROM travel;
```

## *LOGN 函数*

**LOGN** 函数返回数值参数的自然对数。

此返回值与 **EXP** 函数从同一参数返回的指数值相逆。

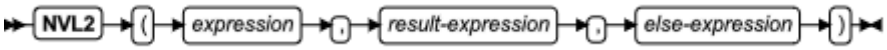
下列查询为 history 表的每一行返回 population 的自然对数：

```
SELECT LOGN(population) FROM history WHERE country='US'          ORDER BY date;
```

## NVL2 函数

当第一个参数不为 NULL 时，返回第二个参数。如果第一个参数为 NULL，则返回第三个参数。

NVL2 函数



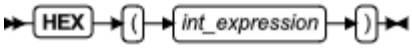

NVL2 函数是下列代码的同义词：

```
CASE WHEN expression IS NOT NULL
THEN result-expression
ELSE else-expression
```

## HEX 函数

HEX 函数返回整数表达式的十六进制编码。

HEX 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *int_expression* | 您想要等同的十六进制的表达式 | 必须是文字整数或返回整数的某个其他表达式 | 表达式 |

下一示例显示十六进制格式的 orders 表的列的数据类型和列长度。 对于 MONEY 和 DECIMAL 列，您可从最低的和次低的字节确定精度和范围。对于 VARCHAR 和 NVARCHAR 列，您可从最低的和次低的字节来确定最小空间和最大空间。 要获取更多关于编码的信息的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

```
SELECT colname, coltype, HEX(collength)
FROM syscolumns C, systables T
WHERE C.tabid = T.tabid AND T.tabname = 'orders';
```

下列示例罗列当前数据库中所有表的名称以及十六进制格式的对应的 tblspace 编号。

```
SELECT tabname, HEX(partnum) FROM systables;
```

十六进制编号中两个最高有效字节构成 dbspace 编号。它们在 GBase 8s 中的 **gcheck** 输出中标识该表。

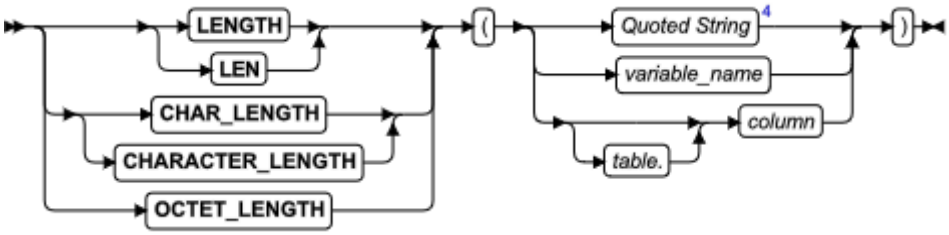
HEX 可在表达式上操作，如下一示例所示：

```
SELECT HEX(order_num + 1) FROM orders;
```

# 长度函数

使用长度函数来确定字符列、字符串或变量的长度，或字符表达式返回的值的长度，或逻辑字符的数目（对于多字节语言环境中的 **CHAR_LENGTH**）。

长度函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | *table* 中列的名称 | 必须有字符数据类型 | 标识符 |
| *table* | 指定的列在其中的表的名称 | 必须存在 | 标识符 |
| *variable* | 包含字符串的主变量或 SPL 变量 | 必须有字符数据类型 | 请参阅名称的特定于语言的规则。 |

这些函数的每一个都有明确的用途：

- **LENGTH** （又称为 **LEN**）
- **OCTET_LENGTH**
- **CHAR_LENGTH**（又称为 **CHARACTER_LENGTH**）

## *LENGTH 函数*

**LENGTH** 函数（又称为 **LEN**）返回字符列中的字节数，但不包括任何末尾的空格。

对于 BYTE 或 TEXT 列，**LENGTH** 返回全部字节数，包括任何末尾的空格。

在 GBase 8s ESQL/C 中，**LENGTH** 还可返回字符变量的长度。

下列示例说明 LENGTH 函数的使用：

```
SELECT customer_num, LENGTH(fname) + LENGTH(lname),
        LENGTH('How many bytes is this?')
        FROM customer WHERE LENGTH(company) > 10;
```

下一示例通过该函数的其他名称 LEN 调用它：

```
EXECUTE FUNCTION LEN("www.gbase.com");
```

上面的 SQL 语句返回整数值 11。

另请参阅 *GBase 8s GLS 用户指南* 中 LENGTH 的讨论。

### *OCTET_LENGTH 函数*

**OCTET_LENGTH** 返回字符列中的字节数，包括任何末尾的空格。另请参阅 *GBase 8s GLS 用户指南*。

### *CHAR_LENGTH 函数*

CHAR_LENGTH 函数返回在它的参数中的逻辑字符的数目，该参数可为字符列、字符变量或引用的字符串。还可调用此内建的函数作为 **CHARACTER_LENGTH**。

在缺省的 U.S. English 语言环境和其他单字节语言环境中，**CHAR_LENGTH** 的行为正像 **LENGTH** 函数一样，并返回在它的参数中的字节数。

然而，对于支持各种 Unicode、东亚和其他非缺省的语言环境的多字节代码集，返回值可小于该参数中的字节数。要获取此函数的讨论，请参阅 *GBase 8s GLS 用户指南*。

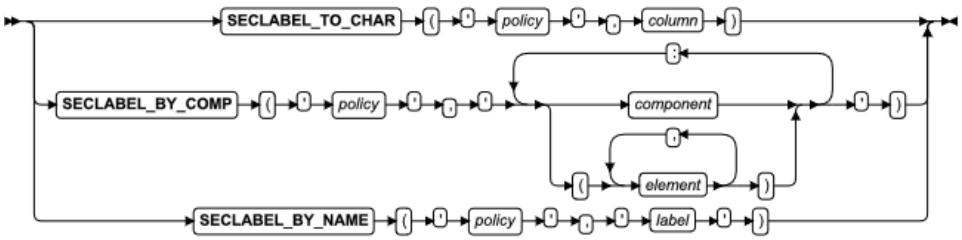
## 安全标签支持函数

安全标签支持函数使得用户能够操纵安全标签。可以三种不同的方式引用安全标签：

- 名称，如在 CREATE SECURITY LABEL 或 RENAME SECURITY LABEL 语句中声明的那样。
- 安全标签的安全策略的每一组件的值的列表。
- IDSSECURITYLABEL 数据类型存储的内部的编码的值。

这些函数可在不同形式的安全标签之间转换。它们通常用于指定 DML 操作的标签，这些操作处理那些由基于标签的访问控制（LBAC）保证安全的数据行。然而，与通过调用该函数的用户的安全凭证已提供的访问相比，在这些操作中，安全标签支持函数不提供针对受保护的数据任何更多的访问。

安全标签支持函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 类型 IDSSECURITYLABEL 的列 | 必须存在并必须存储 *policy* 的一个标签 | 标识符 |
| *component* | *policy* 的组件的值 | 必须存在且必须为 *policy* 的一个组件 | 引用字符串 |

| | | | |
|---|---|---|---|
| *element* | *component* 的值的列表内的值 | 必须存在且必须为 *policy* 的单个组件的元素 | 引用字符串 |
| *label* | 该函数返回其值的安全标签的标识符 | 必须存在且必须为 *policy* 的一个标签 | 引用字符串 |
| *policy* | 由该函数返回其值的安全标签支持的安全策略 | 必须存在且必须为保证该表安全的安全策略 | 引用字符串 |

这些函数返回指定的安全***策略***的安全标签。 可在引用受保护的数据库表的 DML 语句内使用它们，但它们还可在其他调用上下文中求值为安全标签。每一这些函数需要不同的参数列表：

- **SECLABEL_TO_CHAR** 需要安全策略名称以及返回 IDSSECURITYLABEL 对象的表达式，诸如那种数据类型的列的名称。
- **SECLABEL_BY_COMP** 需要安全策略名称以及该安全标签的个别组件的值。
- **SECLABEL_BY_NAME** 需要安全策略和安全标签的名称。

## *SECLABEL_BY_NAME 函数*

**SECLABEL_BY_NAME** 函数使得用户能够通过指定安全标签的名称来直接提供它。

下列 INSERT 语句将一行插入到表 T1 内，该表受到名为 'MegaCorp' 的安全策略的保护。INSERT 语句的 VALUES 子句为通过使用 SECLABEL_BY_NAME 语句要被插入的行提供安全标签 'mylabel'。

```
INSERT INTO T1 VALUES (SECLABEL_BY_NAME ('MegaCorp', 'mylabel'), 1, 'xyz');
```

此 **SECLABEL_BY_NAME** 函数调用的成功并不保证在此示例中 INSERT 操作的成功，因为 **MegaCorp** 安全策略的 **IDSLBACWRITE** 规则影响到该用户是否有充足的安全凭证来将标签 **mylabel** 插入到该行内。

## *SECLABEL_BY_COMP 函数*

**SECLABEL_BY_COMP** 函数返回 IDSSECURITYLABEL 对象，其为它的内部编码的字符串格式的安全标签。此函数使得用户能够通过指定它的组件值直接地提供安全标签。

如果安全标签组件需要多个值，则可通过将那些值放在圆括号之间来指定这样的多个值，比如 `(value_1, value_2, ...)`。当特定的安全标签中的组件需要为空时，可通过在左圆括号和右圆括号之间什么都不放来指定它，比如 `()`。由于在安全组件的元素之中，空格（ASCII 32）是有效的字符，因此出现在安全标签字符串中的任何空格都作为那个组件的元素值的一部分来处理。

安全标签字符串被限定为最大 32 KB。如果该字符串长度超出此限制，则返回错误。

下列 INSERT 语句将一行插入到表 T1 内，该表受到名为 'MegaCorp' 的安全策略保护，该策略有三个组件：'level'、'compartments' 和 'groups'。在此，用户为由指定 SECLABEL_BY_COMP 函数被插入的行提供安全标签。在此示例中，对于 level 组件，安全标签有值 'VP'，对于 compartments 组件，安全标签有值 'Marketing'，对于 groups 组件，安全标签有值 'West'。在 SECLABEL_BY_COMP 的参数中，冒号符号分隔这些安全组件元素值，且引号定界安全标签的组件值的列表。

```
INSERT INTO T1 VALUES (SECLABEL_BY_COMP ('MegaCorp', 'VP:Marketing:West'), 1, 'xyz';
```

在下一示例中，INSERT 语句在表 T1 中插入一行，该表受到同一 MegaCorp 安全策略的保护，该策略有与前面的示例相同的三个组件：level、compartments 和 groups。用户为通过指定该策略名称要被插入的行提供安全标签，以及安全组件元素的列表作为 SECLABEL_BY_COMP 函数的参数。在此，对于 level 组件，安全标签有值 'Director'，对于 compartments 组件，安全标签有值 'HR' 和 'Finance'，对于 groups 组件，安全标签有值 'East' 。

```
INSERT INTO T1 VALUES (SECLABEL_BY_COMP ('MegaCorp', 'Director:(HR,Finance):East'), 1,
'xyz');
```

下列示例将一行插入到表 T1 内，该表受到 MegaCorp 安全组件的保护，其三个组件为 level、compartments 和 groups。SECLABEL_BY_COMP 函数为要被插入的行指定安全标签。在此示例中，对于 level 组件，安全标签有值 'CEO'，对于 compartments 组件，安全标签为空集，对于 groups 组件，安全标签有值 'EntireRegion'。

```
INSERT INTO T1 VALUES (SECLABEL_BY_COMP ('MegaCorp', 'CEO:():EntireRegion'), 3,
'abc');
```

如在所有这些示例中那样，**SECLABEL_BY_COMP** 函数调用的成功不保证 INSERT 语句的成功，因为在数据库服务器允许或拒绝对于插入新行的访问之前，首先要使用 **MegaCorp** 安全策略的 IDSLBACRWRITE 规则，将用户的安全凭证与保护表 **T1** 的安全标签进行对比。

## *SECLABEL_TO_CHAR 函数*

**SECLABEL_TO_CHAR** 函数返回一个安全标签字符串格式的安全标签。

执行此函数的用户的安全凭证可影响该函数的输出。如果用户没有对那个元素的读访问权限，则在输出中不包括安全标签组件的元素。如果用户提供对该数据的读访问的安全凭证，由仅包含那个元素而没有其他元素的安全标签保护数据，则用户有对该元素的访问权限。

对于规则集 **IDSLBACRULES**，仅类型 TREE 的组件可包含用户没有对元素的子集的读访问权限的元素。对于其他类型组件，如果任何元素阻止读访问，则用户根本不可读该行。因此，仅类型 TREE 的安全组件可有以此方式排除的安全组件元素的子集。

例如，如果用户的安全标签的 TREE 类型组件为 {**A**}，且行安全标签的 TREE 类型组件为 {**A**, **B**}，则仅返回组件 **A**，且用户感觉不到在行安全标签中存在 **B**。然而，如果用户持有 **IDSLBACREADTREE** 规则上的豁免，则返回的安全组件为 **A** 和 **B**。

在下一示例中，MegaCorp 安全策略有名为 mylabel 的安全标签，该标签由其值为 'Director' 的 level 组件，以及带有值 'HR' 和 'Finance' 的 compartments 组件组成。被授予了 'mylabel' 的用户已将带有那个安全标签的一行插入到表 T1 内。在此上下文中，在下列 T1 上的 SELECT 语句中，由 SECLABEL_TO_CHAR 函数返回的安全标签字符串如下。

```
SELECT SECLABEL_TO_CHAR ('MegaCorp', C1) FROM T1;
Row returned:
'Director:(HR,Finance)'
```

由于数据库服务器允许对安全策略名称和安全标签组件的值的读访问，根据 **MegaCorp** 安全策略的 **IDSLBACREAD** 规则，此查询的成功说明 **SECLABEL_TO_CHAR** 函数成功，且该用户的安全凭证是充分的。

限定安全标签字符串的最大大小为 32 KB。如果要被返回的安全标签字符串的长度超出此上限，则发出警告，且返回截断了的 32 KB 字符串。

## SIGN 函数

SIGN 函数返回参数的符号的指示符。

SIGN 函数


如果参数小于零，则返回 -1。如果参数等于零，则返回 0。如果参数大于零，则返回 1。返回的结果始终是这些值之一的整数。

## 智能大对象函数

智能大对象函数支持 BLOB 和 CLOB 数据类型的对象：

智能大对象函数有下列语法：

智能大对象

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| BLOB_*column*、CLOB_*column* | 类型 BLOB 的列；类型 CLOB 的列 | *column* 数据类型必须为 BLOB 或 CLOB | 标识符 |
| *column* | *table* 内 BLOB 或 CLOB 值的副本的列 | 必须有 CLOB 或 BLOB 作为它的数据类型 | 引用字符串 |
| *file_destination* | 在其上放置或获取智能大对象的系统 | 有效的值仅为字符串 'server' 或 'client' | 引用字符串 |
| *pathname* | 定位智能大对象的目录路径和 filename | 不超过 256 字节。在 *file_destination* 系统上必须存在。另请参阅 带逗号的 pathname。 | 引用字符串 |
| *table* | 包含 BLOB 或 CLOB 值的副本的 *column* 的表 | 以逗号（不是句号）分隔 '*table*' 和 '*column*' 参数 | 引用字符串 |

### *FILETOBLOB 和 FILETOCLOB 函数*

**FILETOBLOB** 函数为存储在指定的操作系统文件中的数据创建 BLOB 值。类似地，**FILETOCLOB** 函数为存储在操作系统文件中的数据值创建 CLOB 值。

这些函数从下列参数来确定要使用的操作系统文件：

- ***pathname*** 参数标识源文件的目录路径和名称。
- ***file destination*** 参数标识此文件所在的计算机，'client' 或 'server'：
  - 将 ***file destination*** 设置为 'client' 来标识客户端计算机作为源文件的位置。***pathname*** 既可是完整路径也可是当前目录的相对路径。
  - 将 ***file destination*** 设置为 'server' 来标识服务器计算机作为源文件的位置。***pathname*** 必须为完整路径。

*table* 和 *column* 参数是可选的：

- 如果您省略 ***table*** 和 ***column***，则 **FILETOBLOB** 函数以系统指定的存储缺省值来创建 BLOB 值，且 **FILETOCLOB** 函数以系统指定的存储缺省值来创建 CLOB 值。

  这些函数从 ONCONFIG 文件或从 sbspace 获取系统特定的存储特征。要获取更多关于系统指定的存储缺省值的信息，请参阅 *GBase 8s 管理员指南*。
- 如果您指定 ***table*** 和 ***column***，则 **FILETOBLOB** 和 **FILETOCLOB** 函数为它们创建的 BLOB 或 CLOB 值使用来自指定的列的存储特征。

**FILETOBLOB** 返回一个指向新的 BLOB 值的句柄值（指针）。类似地，**FILETOCLOB** 返回一个指向新的 CLOB 值的句柄值。两个参数都不实际地将智能大对象值复制到数据库列内。您必须将 BLOB 或 CLOB 值指定到适当的列。

当 **FILETOCLOB** 函数将文件从客户端或服务器计算机复制到数据库时，它执行任何可能需要的代码集转换。

下列 INSERT 语句使用 FILETOCLOB 函数来从 smith.rsm 文件中的值创建 CLOB：

```
INSERT INTO candidate (cand_num, cand_lname, resume)
VALUES (2, 'Smith', FILETOCLOB('smith.rsm', 'client'));
```

在前面的示例中，**FILETOCLOB** 函数读取客户端计算机上当前目录中的 **smith.rsm** 文件，并返回一指向包含此文件中的数据的 CLOB 值的句柄值。由于 **FILETOCLOB** 函数未指定表和列名称，此新的 CLOB 值有系统指定的存储特征。然后，该 INSERT 语句将此 CLOB 值指定给 **candidate** 表中的 **resume** 列。

下列 INSERT 语句使用 FILETOBLOB 函数从本地数据库服务器上的 photos.xxx 文件中的值创建 BLOB 值，并将那个值插入到 rdb 数据库的 election2008 表内，其为本地数据库服务器的另一数据库：

```
INSERT INTO rdb@:election2008 (cand_pic)
VALUES (FILETOBLOB('C:\tmp\photos.xxx', 'server',
            'candidate', 'cand_photo'));
```

在前面的示例中，**FILETOBLOB** 函数读取本地数据库服务器上指定的目录中的 **photos.xxx** 文件，并返回指向此包含文件中的数据的 BLOB 值的句柄值。然后，该 INSERT 语句将此 BLOB 值指定到本地数据库服务器的 **rdb** 数据库中的 **election2008** 表中的 **cand_pic** 列。此新的 BLOB 值有本地数据库中的 **candidate** 表中的 **cand_photo** 列的存储特征。

在下列示例中，新的 BLOB 值有 rdb2 数据库中的 election96 表中的 cand_pix 列的存储特征，在此，rdb1 和 rdb2 是本地 GBase 8s 实例的数据库：

```
INSERT INTO rdb1:election2008 (cand_pic)
VALUES (FILETOBLOB('C:\tmp\photos.xxx', 'server',          'rdb2:election96', 'cand_pix'));
```

当您以远程数据库和远程数据库服务器的名称限定 **FILETOBLOB** 或 **FILETOCLOB** 函数时，***pathname*** 和 ***file destination*** 成为对远程数据库服务器的相对值。

当您指定 server 作为文件目的地时，如下例所示，FILETOBLOB 函数在远程数据库服务器上查找源文件（在此情况下，为 photos.xxx）：

```
INSERT INTO rdb@rserv:election (cand_pic)
VALUES (rdb@rserv:FILETOBLOB('C:\tmp\photos.xxx', 'server'));
```

然而，当您指定 client 为文件目的地时，如下例所示，FILETOBLOB 函数在本地客户端计算机上查找源文件（在此情况下，为 photos.xxx）：

```
INSERT INTO rdb@rserv:election (cand_pic)
VALUES (rdb@rserv:FILETOBLOB('photos.xxx', 'client'));
```

## *带逗号的 pathname*

如果逗号（,）符号在函数的 *pathname* 内，则数据库服务器期望该 pathname 有下列格式：

***"offset, length, pathname"***

对于包含逗号的 pathname，您还必须指定偏移量和长度，如下例所示：

```
FILETOBLOB("0,-1,/tmp/blob,x","server");
```

引用的 ***pathname*** 字符串中的第一个术语是 ***offset*** 0，其指导数据库服务器在该文件的开头开始读。

第二个术语是 ***length*** -1，其指导数据库服务器继续读，直到整个文件的结尾为止。

第三个术语是 ***/tmp/blob,x pathname***，指定读哪个文件。（请注意在 ***x*** 前面的逗号。）

由于 ***pathname*** 包括逗号，因此当调用 **FILETOBLOB** 时，在此示例中以逗号分隔的 ***offset*** 和 ***length*** 规范是避免错误所必要的。您不需要为不包括逗号但包括 0、-1 的 pathname 指定 ***offset*** 和 ***length***，作为 pathname 字符串的初始字符以避免任何有效的 pathname 的此类错误。

## *LOTOFILE 函数*

**LOTOFILE** 函数将智能大对象复制到操作系统文件。

第一个参数指定要复制的 BLOB 或 CLOB 列。该函数从下列参数确定要创建什么文件：

- ***pathname*** 标识目录路径和源文件名称。
- ***file destination*** 标识此文件在其上的计算机，'client' 或 'server'：
  - 设置 ***file destination*** 为 'client' 来标识客户端计算机作为源文件的位置。***pathname*** 可为完整的 pathname 或相对于当前目录的路径。
  - 设置 ***file destination*** 为 'server' 来标识服务器计算机作为源文件的位置。要求完整的 pathname。

在缺省情况下，LOTOFILE 函数生成这种形式的 filename：

### *file. hex_id*

在此格式中，***file*** 是您在 ***pathname*** 中指定的 filename，而 ***hex_id*** 是唯一的十六进制智能大对象标识符。智能大对象标识符的最大位数为 17。然而，大多数智能大对象可能有位数较少的标识符。

例如，假设您指定 UNIX™ *pathname* 值如下：

'/tmp/resume'

如果 CLOB 列有标识符 203b2，则 LOTOFILE 创建文件：

/tmp/resume.203b2

对于另一示例，假设您指定 Windows™ *pathname* 值如下：

'C:\tmp\resume'

如果 CLOB 列有标识符 203b2，则 LOTOFILE 函数会创建文件：

C:\tmp\resume.203b2

要更改缺省的 filename，您可在 *pathname* 的 filename 中指定下列通配符：

- ***在 filename 中的一个或多个连续的问号（?）可生成唯一的 filename。***

  **LOTOFILE** 函数以来自 BLOB 或 CLOB 列的标识符的十六进制数字替换每一问号。

  例如，假设您指定 UNIX ***pathname*** 值如下：

  '/tmp/resume??.txt'

  **LOTOFILE** 函数将十六进制标识符的 2 位数字放到该名称内。如果 CLOB 列有标识符 **203b2**，则 **LOTOFILE** 函数会创建文件：

  /tmp/resume20.txt

  如果您指定多于 17 个问号，则 **LOTOFILE** 忽略它们。

- ***filename 末尾的叹号（!）表示 filename 不需要是唯一的。***

  例如，假设您指定 Windows pathname 值如下：

  'C:\tmp\resume.txt!'

  **LOTOFILE** 函数不使用 filename 中的智能大对象标识符，因此它生成下列文件：

  C:\tmp\resume.txt

如果您指定的 filename 已存在，则 **LOTOFILE** 返回错误。

当 **LOTOFILE** 函数将 CLOB 值从数据库复制到客户端或服务器计算机上的文件时，它执行任何可能需要的代码集转换。

当您以远程数据库和远程数据库服务器的名称限定 **LOTOFILE** 时，BLOB 或 CLOB 列、***pathname*** 和 ***file destination*** 成为对远程数据库服务器的相对值。

当您指定 server 作为文件目的地时，如下一示例中那样，LOTOFILE 函数将智能大对象从远程数据库服务器复制到在远程数据库服务器上指定目录中的源文件：

```
rdb@rserv:LOTOFILE(blob_col, 'C:\tmp\photo.gif!', 'server')
```

如果您指定 client 作为文件目的地，如在下例中那样，则 LOTOFILE 函数将智能大对象从远程数据库服务器复制到本地客户端计算机上指定的目录中的源文件：

```
rdb@rserv:LOTOFILE(clob_col, 'C:\tmp\essay.txt!', 'client')
```

## *LOCOPY 函数*

**LOCOPY** 函数创建智能大对象的一个副本。

第一个参数指定要复制的 BLOB 或 CLOB 列。*table* 和 *column* 参数是可选的。

- 如果您省略 table 和 column 参数，则 LOCOPY 函数以系统指定的存储缺省值创建智能大对象，并将 BLOB 或 CLOB 列中的数据复制到它之内。

  **LOCOPY** 函数从 ONCONFIG 文件或 sbspace 获取系统特定的存储缺省值。要获取更多关于系统指定的存储缺省值的信息，请参阅 *GBase 8s 管理员指南*。
- 当您指定 ***table*** 和 ***column*** 时，**LOCOPY** 函数为它创建的 BLOB 或 CLOB 值从指定的 ***column*** 使用存储特征。

**LOCOPY** 函数返回指向新的 BLOB 或 CLOB 值的句柄值（一个指针）。此函数***不是*** 实际地将新的智能大对象值存储到数据库中的列内。您必须将该 BLOB 或 CLOB 值指定到适当的列。

下列 GBase 8s ESQL/C 代码片段将 candidate 表的 resume 列中的 CLOB 值复制到 interview 表的 resume 列：

```
/* Insert a new row in the 在 interviews 表中插入新行并
        * （从 sqlca.sqlerrd[1]）获得结果 SERIAL 值
        */
        EXEC SQL insert into interviews (intrv_num, intrv_time)
        values (0, '09:30');
        intrv_num = sqlca.sqlerrd[1];

        /* 以 candidate 编号更新此 interviews 行
        * 并从 candidate 表恢复。使用 LOCOPY 来
        * 在 candidate 表的 resume 列中
        * 创建 CLOB 值的副本。
        */
        EXEC SQL update interviews
        SET (cand_num, resume) =
        (SELECT cand_num,
        LOCOPY(resume, 'candidate', 'resume')
        FROM candidate
        WHERE cand_lname = 'Haven')
```

```
    WHERE intrv_num = :intrv_num;
```

在前面的示例中，**LOCOPY** 函数为 **candidate** 表中的 CLOB **resume** 列的副本返回句柄值。由于 **LOCOPY** 函数指定表和列名称，因此这个新的 CLOB 值有此 **resume** 列的存储特征。如果您省略表（**candidate**）和列（**resume**）名称，则 **LOCOPY** 函数为新的 CLOB 值使用系统定义的存储缺省值。然后，UPDATE 语句将这个新的 CLOB 值指定到 **interviews** 表中的 **resume** 列。

在下列示例中，在本地数据库上执行 LOCOPY 函数并在本地服务器上为 rdb 中的 election2008 表中的 BLOB cand_pic 列的副本返回句柄值，rdb 是本地数据库服务器的另一数据库。然后，INSERT 语句将这个新的 BLOB 值指定到本地的 candidate 表中的 cand_photo 列。

```
INSERT INTO candidate (cand_photo)
SELECT LOCOPY(cand_pic) FROM rdb:election2008;
```

当在与分布式查询中原始的 BLOB 或 CLOB 列一样的数据库服务器上执行 **LOCOPY** 函数时，函数产生 BLOB 或 CLOB 值的两个副本，一个在远程数据库中，另一个在本地数据库中，如下列两个示例所示。

在第一个示例中，在远程 rdb 数据库上执行 LOCOPY 函数，并为远程 election2008 表中的 BLOB cand_pic 列的副本在远程服务器中返回句柄值。然后，INSERT 语句将这个新的 BLOB 值指定到本地 candidate 表中的 cand_photo 列：

```
INSERT INTO candidate (cand_photo)
SELECT rdb:LOCOPY(cand_pic) FROM rdb:election2008;
```

在第二个示例中，在本地数据库上执行 LOCOPY 函数，并为本地 candidate 表中的 BLOB cand_photo 列的副本在本地数据库上返回句柄值。然后，该 INSERT 语句将这个新的 BLOB 值指定到远程 rdb 数据库中 election2008 表中的 cand_pic 列：

```
INSERT INTO rdb:election2008 (cand_pic)
SELECT LOCOPY(cand_photo) FROM candidate;
```

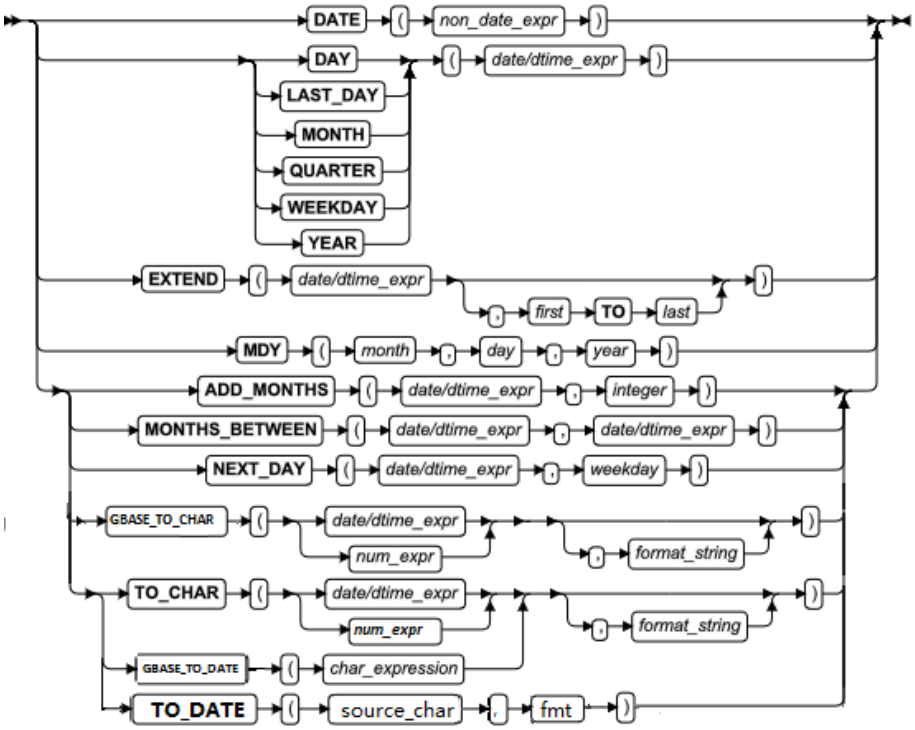
内建的 **LOCOPY** 函数的 BLOB 和 CLOB 参数是内建的 opaque 数据类型。这些可为由跨数据库 DML 操作返回的值，或为跨数据库函数调用返回的值，但内建的 opaque 类型不支持跨数据库服务器实例的分布式操作。如果本地数据库与 **rdb** 数据库是不同的 GBase 8s 实例的数据库，则在前面的两个示例中的 INSERT 语句失败并报错 -999。

## 时间函数

GBase 8s 的时间函数接受 DATE 或 DATETIME 参数，或 DATE 或 DATETIME 值的字符表示。它们通常返回 DATE 或 DATETIME 值，或将它们从 DATE 或 DATETIME 值抽取的信息转换为字符串或整数。

另请参阅在 代数函数 部分中的 **ROUND** 和 **TRUNC** 函数的描述，其可更改 DATE 或 DATETIME 值的精度。

时间函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *char_expression* | 要被转换为 DATE 或 DATETIME 值的表达式 | 必须为文字、主变量、表达式或字符数据类型的列 | 表达式 |
| *date/dtime_expr* | 返回 DATE 或 DATETIME 值的表达式 | 可为主变量、表达式、列或常量。 | 表达式 |
| *day* | 返回该月的天数的表达式 | 必须返回 > 0 但不大于指定的月中天数的整数 | 表达式 |
| *first* | 结果中的最大时间单位。如果您省略 *first* 和 *last*，则缺省的 *first* 是 YEAR。 | 必须是指定不小于 *last* 的时间单位的 DATETIME 限定符关键字 | DATETIME 字段限定符 |
| *format_string* | 包含第一个参数的格式掩码的字符串 | 必须为指定有效的格式的字符数据类型。可为列、主变量、表达式或常量 | 引用字符串 |
| *integer* | 指定月的整数的表达式 | 必须求值为正的或负的整数 | 表达式 |
| *last* | 结果中的最小时 | 必须是指定不小于 | DATETIME |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 间单位 | *first* 的时间单位的 DATETIME 限定符关键字 | 字段限定符 |
| *month* | 表示月的数值的表达式 | 必须求值为取值范围从 1 至 12（包括 1 和 12）的整数 | 表达式 |
| *non_date_expr* | 表示要被转换为 DATE 数据类型的值的表达式 | 通常是一个表达式，该表达式返回可被转换为 DATE 数据类型的 CHAR、DATETIME 或 INTEGER 值的表达式 | 表达式 |
| *num_expr* | 求值为实数值的表达式 | 必须返回数值数据类型 | 表达式 |
| *weekday* | 星期几的缩写名称 | 包含星期几的有效缩写的字符数据类型 | 引用字符串 |
| *year* | 表示年份的数值表达式 | 必须求值为 4 位整数。您不可使用 2 位缩写。 | 表达式 |
| *source_char* | 要被转换为 DATETIME 值的源字符串 | 设置为空字符串或 NULL 时，返回结果为空。 | 引用字符串 |
| *fmt* | DATETIME 类型的格式化字符串 | 设置为空字符串或 NULL 时，返回结果为空。 | 引用字符串 |

## *ADD_MONTHS 函数*

**ADD_MONTHS** 函数采用 DATETIME 或 DATE 表达式作为它的第一个参数，并需要第二个参数指定要添加到第一个参数值上的月数。第二个参数可为正的或负的。

返回的值是基于第二个参数指定的月数，作为 INTERVAL UNITS MONTH 值的第一个参数的 DATE 或 DATETIME 值的总和。

返回的数据类型依赖于第一个参数的数据类型：

- 如果第一个参数求值为 DATE 值，则 **ADD_MONTHS** 返回 DATE 值。
- 如果第一个参数求值为 DATETIME 值，则 **ADD_MONTHS** 返回 DATETIME YEAR TO FRACTION(5) 值，对于时间单位小于 ***day*** 的，其值与第一个参数中的相同。

如果在第一个参数中的 ***day*** 和 ***month*** 时间单位指定该月的最后一天，或如果结果月的天数少于第一个参数中的 ***day***，则返回的值为结果月的最后一天。否则，返回的值与第一个参数为该月的同一天。

如果结果月晚于第一个参数中那年的十二月（或对于负的第二个参数，早于一月），则返回的值可在不同的年份中。

下列查询使用列表达式作为参数，在 Projection 子句中调用 **ADD_MONTHS** 函数两次。在此，列名称表明列数据类型，且 **DBDATE** 设置为 MDY4/：

```
SELECT a_serial, b_date, ADD_MONTHS(b_date, a_serial),
        c_datetime, ADD_MONTHS(c_datetime, a_serial)
        FROM mytab WHERE a_serial = 7;
```

在此示例中，ADD_MONTHS 返回 DATE 和 DATETIME 值：

```
a_serial        7
b_date          07/06/2007
(expression)    02/06/2008
c_datetime      2007-10-06 16:47:49.00000
(expression)    2008-05-06 16:47:49.00000
```

如果您使用主变量来存储 ADD_MONTHS 的参数，但在准备时刻不知道该参数的数据类型，则 GBase 8s 假设数据类型为 DATETIME YEAR TO FRACTION(5)。如果在运行时刻，在已准备了该语句之后，用户为主变量提供 DATE 值，则数据库服务器发出错误 -9750。要防止此错误，请通过使用强制转型来指定主变量的数据类型，如在此程序片断中所示：

```
sprintf(query, "
"select add_months(?::date, 6) from mytab");
EXEC SQL prepare selectq from :query;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;
EXEC SQL fetch select_cursor into :var_date_output;
```

## *DATE 函数*

**DATE** 函数将它的参数转换为 DATE 值。

它的非 DATE 参数可为可转换为 DATE 值的任何表达式，通常是 CHAR、DATETIME 或 INTEGER 值。下列 WHERE 子句指定引用的字符串作为它的 CHAR 参数：

```
WHERE order_date < DATE('12/31/07')
```

当 **DATE** 函数解释一 CHAR 非 DATE 表达式时，它期望此表达式符合任何 **DBDATE** 环境变量指定的 DATE 格式。例如，假设当您执行下列查询时，**DBDATE** 设置为 Y2MD/：

```
SELECT DISTINCT DATE('02/01/2008') FROM ship_info;
```

此 SELECT 语句生成错误，因为 **DATE** 函数不可转换此字符串表达式。**DATE** 函数将该日期字符串的第一部分（02）解释为年份，第二部分（01）为月份。

对于第三部分（2008），当 **DATE** 函数期望两位的日期（有效的日期值必须介于 01 与 31 之间）时，它遇到了四位数字。因此，它不可转换该值。对于要以 **DBDATE** 的 Y2MD/ 值成功地执行的 SELECT 语句，该参数需要为 '08/02/01'。要获取关于 **DBDATE** 的格式的信息，请参阅 *《GBase 8s SQL 指南：参考》*。

要获取那些可指定 DATE 值的显示和数据条目格式的 GBase 8s 环境变量之中优先级顺序的信息，请参阅主题 DATE 和 DATETIME 格式规范的优先顺序。

当您为非 DATE 表达式指定正的 INTEGER 值时，**DATE** 函数将此解释为 1899 年 12 月 31 日之后的天数。

如果整数值为负的，则 DATE 函数将该值解释为 1899 年 12 月 31 日之前的天数。下列 WHERE 子句为非 DATE 表达式指定 INTEGER 值：

```
WHERE order_date  <  DATE(365)
```

数据库服务器搜索 **order_date** 值小于 1900 年 12 月 31 日(其为 12/31/1899 加上 365 天)的行。

## *DAY 函数*

**DAY** 函数采用 DATE 或 DATETIME 参数，并返回该月的日期作为取值范围从 1 至当前月中天数的一个整数。

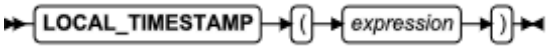
下列语句片段调用带有 CURRENT 函数的作为参数的 DAY 函数，比较 order_date 列值与该月的当前日期：

```
WHERE DAY(order_date) > DAY(CURRENT)
```

## *LOCAL_TIMESTAMP 函数*

在数据类型 TIMESTAMP 的值中返回会话时区中的当前日期和时间。

LOCAL_TIMESTAMP 函数




此函数与 CURRENT_TIMESTAMP 之间的不同在于，LOCALTIMESTAMP 返回 TIMESTAMP 值，而 CURRENT_TIMESTAMP 返回 TIMESTAMP WITH TIME ZONE 值。

## *MONTH 函数*

**MONTH** 函数返回对应于它的 DATE 或 DATETIME 参数的 ***month*** 部分的整数。

像 **DAY**、**YEAR**、**WEEKDAY** 和 **QUARTER** 内建的时间函数一样，**MONTH** 函数从作为它的参数的单个 DATE 或 DATETIME 表达式抽取信息。返回值是在日历年中月份序列内 ***month*** 的顺序位置。例如，在 9 月 23 日，该函数表达式 MONTH(TODAY) 返回 9。

下列示例返回取值范围从 1 至 12 的一个数值来表示下订单的月份：

```
SELECT order_num, MONTH(order_date) FROM orders;
```

## *QUARTER 函数*

**QUARTER** 函数返回一个取值范围从 1 至 4 的整数，对应于包括它的 DATE 或 DATETIME 参数的那个日历年的***季度***。

例如，一月、二月或三月中的任何日期都返回整数 1。

该参数必须为求值为 DATE 或 DATETIME 数据类型的表达式。

**QUARTER 函数表达式的示例**

下列函数表达式返回 3，因为八月在一年中的第三个季度中。

```
QUARTER('2014-08-25')
```

下列示例返回取值范围可从 1 至 4 的一个数值，来表明下订单时的季度：

```
SELECT order_num, QUARTER(order_date) FROM orders;
```

下列查询包括 QUARTER 函数表达式，其参数为 order_date 列和 CURRENT 运算符。WHERE 子句将结果集限定为 order_date 值所在季度早于当前年当前季度的那些行：

```
SELECT * FROM orders
        WHERE (QUARTER(order_date) < QUARTER(CURRENT))
        AND YEAR(order_date) = YEAR(CURRENT);
```

然而，在第一季度期间，此查询不返回行，因为从小于当前季度的季度里，不可有数据。也就是说，没有值为零的季度。

## *WEEKDAY 函数*

**WEEKDAY** 函数接受 DATE 或 DATETIME 参数，并返回取值范围从 0 至 6 代表星期几的整数。

作为返回值，零（0）代表星期天，一（1）代表星期一，以此类推。

下列查询返回与当前日期相同的星期几支付了的所有订单：

```
SELECT * FROM orders
   WHERE WEEKDAY(paid_date) = WEEKDAY(CURRENT);
```

## *YEAR 函数*

**YEAR** 函数采用 DATE 或 DATETIME 参数，并返回表示该年的四位整数。

下列示例罗列了其 ship_date 早于当前年初的订单：

```
SELECT order_num, customer_num FROM orders
        WHERE year(ship_date) < YEAR(TODAY);
```

类似地，由于 DATE 值是一个简单的日历日期，您不可以一个其 ***last*** 限定符小于那个 DAY 的 INTERVAL 值来加上或减去一个 DATE 值。在此情况下，请将 DATE 值转换为 DATETIME 值。

## *MONTHS_BETWEEN 函数*

**MONTHS_BETWEEN** 函数接受两个 DATE 或 DATETIME 表达式参数，并返回一个带符号的 DECIMAL 值，该值量化这些参数之间的月数间隔，就好像 *month* 是时间的单位一样。

此函数需要两个参数，每一参数可为 DATE 表达式或 DATETIME 表达式。

返回的值为 DECIMAL 数据类型，表示两个参数之间的差异，表达为基于 31 天为单位的 DECIMAL 值。如果第一个参数是晚于第二个参数的时间点，则返回的值的符号为正。如果第一个参数早于第二个参数，则返回的值的符号为负。如果两个参数相等，则返回值为零。

如果两个参数的日期都是一个月的同一天，或都是一个月的最后一天，则结果为整数。否则，基于一个月 31 天，来计算结果的小数部分。此小数部分还可包括 ***hour***、***minute*** 和 ***second*** 时间单位中的差异，除非两个参数都是 DATE 表达式。

下列查询使用通过 **TO_DATE** 表达式作为参数返回的两个 DATE 值，调用 Projection 子句中的 **MONTHS_BETWEEN** 函数。

```
SELECT MONTHS_BETWEEN(TO_DATE('2-2-2005', '%m-%d-%Y'),
        TO_DATE('1-1-2005',  '%m-%d-%Y'))
        AS lunations FROM systables WHERE tabid = 1;
```

该查询返回的值表示两个 DATE 参数之间有 32 天的差异，作为 31 天月份的正值：

```
months
1.03225806451613
```

下一示例将 DATETIME 列表达式参数返回到 MONTHS_BETWEEN 表达式，以及对于表的两行，它们的以月计算的差异：

```
SELECT d_datetime, e_datetime,
MONTHS_BETWEEN(d_datetime, e_datetime) AS months_between
FROM mytab1;

d_datetime          2007-11-01 09:00:00.00000
e_datetime          2007-12-07 14:30:12.12345
months_between    -1.2009453405018
d_datetime          2007-12-13 09:40:30.00000
e_datetime          2007-11-13 08:40:30.00000
months_between    1.00000000000000
```

在此，第一个 MONTHS_BETWEEN 结果包括以小于天的时间单位计的差异。第二个结果没有小数部分，因为两个参数的 *day* 时间单位有相同的值。

下一示例中的 MONTHS_BETWEEN 表达式比较 DATE 与 DATETIME 值：

```
SELECT col_datetime, col_date,                MONTHS_BETWEEN(col_datetime, col_date) AS
months_between
FROM mytab2;
col_datetime       2008-12-13 08:40:30.00000
col_date             11/13/2007
months_between    13.0000000000000
```

由于两个参数指定该月的同一天，因此结果没有小数部分。

## *LAST_DAY 函数*

**LAST_DAY** 函数需要一个 DATE 或 DATETIME 表达式作为它的唯一参数。它返回它的参数指定的那个月的最后一天的日期。

此返回的值的数据类型与参数的数据类型相同。返回的值与参数之间的差为那个月剩余的天数。

下列查询返回当前日期的 DATE 表示、当前月中最后一天的日期，以及当前月中最后一天之前的天数（由第二个 DATE 值减去第一个计算）：

```
SELECT TODAY AS today, LAST_DAY(TODAY) AS last,
        LAST_DAY(TODAY) - TODAY AS days_left
        FROM systables WHERE tabid = 1;
```

如果在 2008 年 3 月 12 日发出了该查询，以 MDY4/ 作为缺省的语言环境的 DBDATE 设置，则它会返回下列信息：

```
today          last          days_left
03/12/2008   03/31/2008          19
```

在此示例的 SELECT 语句中，在 Projection 子句中 TODAY 运算符与标识符 today 之间没有名称冲突，因为 AS 关键字向 GBase 8s 表明 today 为一显示标签。

如果您使用主变量来存储 LAST_DAY 的参数，但在准备时刻不知道该参数的数据类型，则 GBase 8s 假设该数据类型为 DATETIME YEAR TO FRACTION(5)。如果在运行时刻，在已准备了该语句之后，用户为该主变量提供 DATE 值，则发出错误 -9750。要防止发生此错误，请通过使用强制转型来指定该主变量的数据类型，如在此程序片段中所示：

```
sprintf(query, '',
"select last_day(?::date) from mytab");
EXEC SQL prepare selectq from :query;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;
EXEC SQL fetch select_cursor into :var_date_output;
```

## *NEXT_DAY 函数*

**NEXT_DAY** 函数返回晚于它的第一个 DATE 或 DATETIME 参数的最早的日期，并落在它的第二个参数指定的星期几上。第二个参数是三个 ASCII 字符的加引号的字符串，是星期几的英文名称缩写。

NEXT_DAY 函数需要两个参数：

- 求值为早于该返回值的日期的 DATE 或 DATETIME 表达式。
- 至少三个 ASCII 字符的字符串，对应于取值范围从 ASCII 65 至 ASCII 90 的大写字母。这三个字母编码为星期几的英文名称缩写。

成功地执行此函数，返回满足两个条件的最早的日历日期：

- 该日期晚于第一个参数指定的日期。
- 该日期落在第二个参数指定的星期几上。

NEXT_DAY 接受下列星期几的缩写字符串：

| 星期 | 缩写 | 星期 | 缩写 |
|---|---|---|---|
| 星期日 | 'SUN' | 星期三 | 'WED' |
| 星期一 | 'MON' | 星期四 | 'THU' |
| 星期二 | 'TUE' | 星期五 | 'FRI' |
| | | 星期六 | 'SAT' |

忽略跟在这些缩写字符串的第三个字符之后的任何字符。例如，对于第二个参数，'MONDAY' 和 'MONTAG' 都是有效的规范，每一都指定在第一个参数的日期之后的星期一。然而，如果第二个字符串的前三个字符不与上表中*星期*缩写之一相匹配，比如 'MODNAY'， 则 GBase 8s 发出错误。

例如，下列查询包括有效的 NEXT_DAY 表达式：

```
SELECT ship_date, NEXT_DAY(ship_date, 'SAT') AS next_saturday,
NEXT_DAY(ship_date, 'SAT') - ship_date AS num_days FROM orders;
```

此查询的结果集可能包括来自 orders 表的下列数据：

```
ship_date      next_saturday    num_days
06/01/2006     06/03/2006         2
02/12/2007     02/17/2007         5
05/31/2007     06/02/2007         2
05/23/2007     05/26/2007         3
```

由 **NEXT_DAY** 返回的值与第一个参数有相同的数据类型。如果此参数为 DATE 类型，则 **NEXT_DAY** 返回 DATE 值。如果第一个参数为 DATETIME 类型，则 **NEXT_DAY** 返回 DATETIME YEAR TO FRACTION(5) 值。

由于在前面示例中的 **ship_date** 是 DATE 列，所以返回的日期格式化为 DATE 值，而不是 DATETIME 格式。

如果您使用主变量来存储 NEXT_DAY 的参数，但在准备时刻不知道该参数的数据类型，则 GBase 8s 假设该数据类型为 DATETIME YEAR TO FRACTION(5)。如果在运行时刻，在已准备了该语句之后，用户为主变量提供 DATE 值，则发出错误 -9750。要避免这种错误，请使用强制转型来指定该主变量的类型，如此程序片段中所示：

```
sprintf(query, '', "select next_day(?::date, 'SUN') from mytab");
EXEC SQL prepare selectq from :query;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;
EXEC SQL fetch select_cursor into :var_date_output;
```

## *EXTEND 函数*

**EXTEND** 函数调整 DATETIME 或 DATE 值的精度。

作为它的第一个参数的 DATETIME 或 DATE 表达式不可为 DATE 值的加引号的字符串表示。

如果您未指定 **first** 和 **last** 限定符，则缺省的限定符为 YEAR TO FRACTION(3)。

如果表达式包含未通过时间单位限定符指定的字段，则丢弃那些字段。

如果 **first** 限定符指定比表达式中存在的更大的(即，更多有效位的)时间单位，则以 **CURRENT** 函数返回的值填充新的字段。如果 **last** 限定符比表达式中存在的更小的（即，更少有效位的）时间单位，则以常量值填充新的字段。以 1 填充丢失的 MONTH 或 DAY 字段，且以 0 填充丢失的 HOUR 或 FRACTION 字段。

在下列表达式中，EXTEND 调用返回带有 YEAR TO SECOND 表达式的 call_dtime 列值：

```
EXTEND (call_dtime, YEAR TO SECOND)
```

您可使用 *EXTEND* 函数来执行 DATETIME 值与没有相同时间单位限定符的 INTERVAL 值的加法或减法。下一表达式将文字的 DATETIME YEAR TO DAY 值扩展为 YEAR TO MINUTE 精度，因而可从它减去间隔的 YEAR TO MINUTE 值：

```
EXTEND (DATETIME (2009-8-1) YEAR TO DAY, YEAR TO MINUTE)
            - INTERVAL (720) MINUTE (3) TO MINUTE
```

您可使用 *EXTEND* 函数来选择性地更新 DATETIME 值中时间单位的子集。在下一示例中的 UPDATE 语句仅更新 DATETIME YEAR TO MINUTE 列中的 *hour* 和 *minute* 时间单位值。

```
UPDATE cust_calls SET call_dtime = call_dtime -
  (EXTEND(call_dtime, HOUR TO MINUTE) - DATETIME (11:00)
HOUR TO MINUTE) WHERE customer_num = 106;
```

从由 EXTEND 返回的 DATETIME HOUR TO MINUTE 值中减去 11:00 生成一个正的或负的 INTERVAL HOUR TO MINUTE 值。从 call_dtime 列中原始的值减去此差异，在 cust_calls.call_dtime 列中将更新的 *hour* 和 *minute* 时间单位值强制为 11:00。

## *MDY 函数*

**MDY** 函数将表示 ***month***、***day*** 和 ***year*** 的三个整数表达式作为它的参数，并返回类型 DATE 值。

- 第一个参数表示月份的数值（1 至 12）。
- 第二个参数表示该月的日期的数值（1 至 28、29、30 或 31，与月份相对应）。
- 第三个参数表示 4 位的年份。您不可使用 2 位缩写。

**UPDATE 语句中 MDY 函数的示例**

下列示例更新 orders 表中的一行，将其采购订单号为 8052 的 paid_date 列值更改为当前月的第一天：

```
UPDATE orders SET paid_date = MDY(MONTH(TODAY), 1, YEAR(TODAY))
            WHERE po_num = '8052';
```

在此，MDY 函数的第一个和最后一个参数是返回对应于当前的 *month* 和 *year* 的整数的时间表达式。第二个参数指定*该月的日期*为以文字整数。颠倒此示例中前两个参数的顺序，会将 paid_date 更改为当前年的 1 月 13 日之前的某一天，除非同一应用还包括了错误检查代码，标识那个时期过早，不是有效的。

## *GBASE_TO_CHAR 函数*

**GBASE_TO_CHAR** 函数将一求值为 DATE、DATETIME 或数值值的表达式转换为字符串。

返回的字符串表示第一个参数指定的数据值，使用第二个参数在 ***format_string*** 中定义的格式掩码，可包括特殊的格式符号和文字字符。

- 此函数的第一个参数必须为 DATE、DATETIME 或内建的数值数据类型，或可转换为这些数据类型之一的字符串。如果初始的 DATE、DATETIME 或数值参数的值为 NULL，则该函数返回 NULL 值。
- 此函数的第二个参数是指定格式掩码的字符串。哪些特殊的字符适合于格式掩码，主要依赖于 **GBASE_TO_CHAR** 函数的第一个参数是表示时间点还是数值。

**格式化 DATE 和 DATETIME 表达式**

***format_string*** 参数不需要隐含与 **GBASE_TO_CHAR** 函数的第一个参数中的值相同的时间单位。当 ***format_string*** 中隐含的精度与第一个参数中的 DATETIME 限定符不同时，**GBASE_TO_CHAR** 函数扩展 DATETIME 值，就如同它已调用了 **EXTEND** 函数。

在下列示例中，用户想要将 tab1 表的 begin_date 列转换为字符串。begin_date 列定义为 DATETIME YEAR TO SECOND 数据类型。用户使用带有 **GBASE_TO_CHAR** 函数的 SELECT 语句来执行此转换：

```
SELECT GBASE_TO_CHAR(begin_date, '%A %B %d, %Y %R') FROM tab1;
```

此示例的 ***format_string*** 中的符号有下列含义。

| 符号 | 含义 |
|---|---|
| %A | 完整的星期名称，如在语言环境中定义的那样 |
| %B | 完整的月份名称，如在语言环境中定义的那样 |
| %d | 以整数（01 至 31）表示的该月的日期。在单一位的值之前添零（0）。 |
| %Y | 4 位十进制数值的年份 |
| %R | 24 小时表示法的时间（等同于 %H:%M 格式，如下面定义的那样）。 |

请注意，在上例中紧跟在 %d 格式规范之后的逗号（,）是文字字符，而不是 **GBASE_TO_CHAR** 函数的参数的分隔符。第二个参数是引号括起的字符串 '%A %B %d, %Y %R'，定义 **TO_CHAR** 返回的值中表示第一个参数的格式掩码。

将此 *format_string* 应用于 begin_date 列值，返回此结果：

```
Wednesday July 25, 2013 18:45
```

下例中的查询调用 TO_CHAR 来将同一格式字符串应用于 ADD_MONTHS 表达式，并展示该查询的结果：

```
SELECT ship_date, GBASE_TO_CHAR(ADD_MONTHS(ship_date, 1), '%A %B %d, %Y')
        AS survey_date FROM orders;
        ship_date       03/12/2013
        survey_date     Thursday April 12, 2013
```

在以上的查询输出中，

- 根据 **DB_DATE** 环境变量设置格式化 **ship_date** 值，

- 并根据 **GBASE_TO_CHAR** 函数的 '%A %B %d, %Y %R' 格式字符串参数格式化 **survey_date** 值。

对于 DATE 或 DATETIME 值，在 **GBASE_TO_CHAR** 函数的 ***format_string*** 参数中附加的有效字符包括下列。

**符号　含义**

**%a**　缩写的星期名称，如在语言环境中定义的那样

**%b**　缩写的月份名称，如在语言环境中定义的那样

**%C**　表示为整数（00 至 99）的世纪数值（年份除以 100 并截断为整数）

**%D**　与 %m/%d/%y 格式相同

**%e**　数值表示的该月的日期（1 至 31）。单一位值之前添加一空格。

**%Fn**　秒的小数部分的值，由无符号的整数 n 指定精度。n 的缺省值为 2；n 的范围为 0≤n≤5。此值覆盖在 % 与 F 字符之间指定的任何宽度或精度。

**%h**　与 %b 格式相同：缩写的月份名称，如在语言环境中定义的那样

**%H**　2 位整数表示的小时（00 至 23）（24 小时时钟）

**%I**　2 为整数表示的小时（00 至 11）（12 小时时钟）

**%m**　整数表示的月份（01 至 12）。任何单一位值之前添加零（0）。

**%M**　2 位整数表示的分钟（00 至 59）

**%S**　2 位整数表示的秒（00 至 61）。秒值可达到 61（而不是 59），以允许偶然的跳跃秒或双跳跃秒。

**%T**　%H:%M:%S 格式的时间

**%w**　数值表示的星期（0 至 6）；0 表示等同于星期日的语言环境。

**%y**　2 位十进制数值表示的年份。

例如，假设在 2013 年 8 月 23 日，DB-Access 实用程序发出了下列查询：

```
SELECT GBASE_TO_CHAR(CURRENT YEAR TO FRACTION(5),
"%Y-%m-%d %H:%M:%S.%F")
FROM sysmaster:sysdual;
```

在此示例中，格式字符串参数以下列文字字符作为 DATETIME 字段值之间的分隔符指定用户格式：

- ASCII 45 ( - ) 连字符分隔年、月和日值
- ASCII 32 ( ) 空格分隔开日期与小时
- ASCII 58 ( : ) 冒号分隔时、分和秒
- ASCII 46 ( . ) 句号分隔秒与秒的小数部分。

这是以指定的 DATETIME 用户格式返回的值：

```
(expression)     2013-08-23 13:15:53.00
```

当 DATETIME 或 DATE 表达式是第一个参数时，如果您省略 *format_string* 参数，则 TO_CHAR 函数使用 DBTIME 或 DBDATE 环境变量的设置作为缺省值，来格式化在第一个参数中表示的值。在非缺省的语言环境中，通过诸如 GL_DATETIME 和 GL_DATE 这样的环境变量指定 DATETIME 和 DATE 值的缺省格式。

**重要：**对于其精度包括 SECOND 和 FRACTION 数据值的 DATETIME 用户格式，连接那些字段，除非在 %S 与 %F 格式伪指令之间显式地定义分隔符字符。在版本 11.70.xC7 以及更早的 GBase 8s 产品中，在缺省情况下， %F 伪指令在 SECOND 与 FRACTION 字段值之间插入了 ASCII 46 字符（.）。然而，在此产品中，%F 伪指令为隐含缺省的分隔符。

要了解那些可为内建的按时间顺序排列的数据类型指定显示和数据条目格式的 GBase 8s 环境变量之中的优先顺序，请参阅主题 DATE 和 DATETIME 格式规范的优先顺序。

**格式化数值的和 MONEY 表达式**

**GBASE_TO_CHAR** 函数的 ***format_string*** 参数支持与 ESQL 函数使用的相同的数值格式掩码，比如 **rfmtdec( )**、**rfmtdouble( )** 和 **rfmtlong( )**。在 *GBase 8s ESQL/C 程序员手册* 中，是对数值值（当将数值表达式格式化为字符串时）的 GBase 8s 数值格式掩码的详尽描述。以下是数值格式掩码的简短总结描述。

当将数值表达式格式化为字符串时，数值格式掩码指定应用于某数值值的格式。此掩码是下列格式化字符的组合：

**符号　含义**

*****　此字符以星号填充本来为空的显示字段中的任何位置

**&**　此字符以零填充本来为空的显示字段中的任何位置

**#**　此字符将开始的零更改为空格。使用此字符来指定字段向左边的最大扩展。

**<**　此字符对显示字段总的数值进行左调整。它将开始的零更改为 NULL 字符串。

**,**　此字符表明在值的整数部分按三位一组（从单位位置向左数）分隔的符号。在缺省情况下，此符号为逗号。您可以 DBMONEY 环境变量设置该符号。在格式化了的数值中，仅当该值的整数部分有四位或更多位时才出现此符号。

**.**　此字符表明将货币值的整数部分与小数部分分隔的符号。在缺省情况下，此符号为句号。您可以 DBMONEY 环境变量设置该符号。在格式字符串中，您仅可有一个句号。

**-**　此字符为文字字符。当 ***expr1*** 小于零时，它作为负号出现。当您将一行中的几个负号（-）分组时，单个负号浮动到它可占据的最右边的位置；它不影响数值及其币种符号。

**+**　此字符为文字字符。当 ***expr1*** 大于或等于零时，它作为正号出现。当您在将一行中的几个正号分组时，单个加号或减号浮动到它可占据的最右边的位置；它不影响数值及其币种符号。

**(** 此字符为文字字符。它作为负数值的左边的左圆括号（()出现。它是替换负数的减号的一对会计圆括号之一。当您对一行中的几个分组时，单个左圆括号浮动到它可占据的最右边的位置；它不影响数值及其币种符号。

**)** 这是替换负值的减号的一对会计圆括号之一。

**$** 此字符显示在数值值前面的币种符号。在缺省的语言环境中，币种符号是美元号（$）。您可以 DBMONEY 环境变量设置非缺省的币种符号。当您将一行中的几个美元号分组时，单个币种符号浮动到它可占据的最右边的位置；它不影响该数值。

在 **GBASE_TO_CHAR** 函数返回的格式化的值中，按字面重新产生格式掩码中的任何其他字符。

在下面的三个示例中，**GBASE_TO_CHAR** 的 **d_int** 列表达式参数的值为 -12344455。

此查询在 **GBASE_TO_CHAR** 的调用中未指定格式掩码：

```
SELECT GBASE_TO_CHAR(d_int) FROM tab_numbers;
```

下列表格展示此 SELECT 语句的输出。

| (expression) |
| --- |
| -12344455 |

下列查询指定货币格式掩码：

```
SELECT GBASE_TO_CHAR(d_int, "$*********.**") FROM tab_numbers;
```

下列表格展示此 SELECT 语句的输出。

| (expression) |
| --- |
| $12344455.00 |

```
SELECT GBASE_TO_CHAR(d_int, "-$*********.**") FROM tab_numbers;
```

该查询返回 - $12344455.00。

```
SELECT GBASE_TO_CHAR(12344455,"-$*********.**") FROM tab_numbers;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
| --- |
| $12344455.00 |

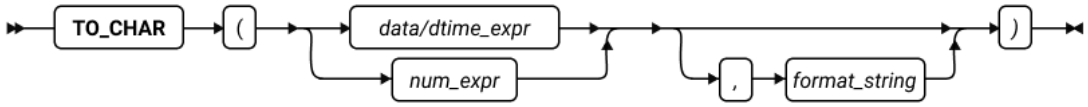
应用来自格式掩码参数的币种（$）符号，但减号（-）不起作用，因为第一个参数的值大于零。

请注意，当 **GBASE_TO_CHAR** 函数的第一个参数是 DATE 或 DATETIME 表达式时，它是时间表达式，或是可被格式化为 DATE 或 DATETIME 表达式的字符串。然而，当它的第一个参数是数值或货币值时，**GBASE_TO_CHAR** 返回那个参数的值的字符串形式，但它不返回时间表达式。

## *TO_CHAR 函数*

**TO_CHAR** 函数将一求值为 DATE、DATETIME 或数值值的表达式转换为字符串。

**语法**




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *date/dtime_expr* | 返回 DATE 或 DATETIME、TIMESTAMP 值的表达式 | 可为主变量、表达式、列或常量。 | 表达式 |
| *num_expr* | 要转换为字符串的数值型数据 | 可为主变量、表达式、列或常量。 | 表达式 |
| *format_string* | 指定的格式化字符串 | 为指定有效的格式的字符数据类型。可为列、主变量、表达式或常量。支持省略。 | 引用字符串 |

在下面的示例中，TO_CHAR() 的 numcol 列表达式参数的值为 13。

```
SELECT TO_CHAR(numcol) FROM tab1;
```

返回结果为：13

**TO_CHAR(datetime)：转换日期型表达式**

TO_CAHR（datetime）将日期类型 DATE、DATETIME、TIMESTAMP 表达式转换为 ***format_string*** 参数中指定格式的 VARCHAR 类型值。

对于日期数据类型，**TO_CHAR** 函数的 ***format_string*** 参数有效的日期元素如下：

| 元素 | 含义（范围） |
|---|---|
| / - , . : | 标点符号在结果中重新复制 |
| YYYY | 4 位的年份 |
| YY | 年份的最后 2 位数字 |
| MM | 月份（01-12） |
| DD | 月中的某一天（01—31） |
| HH、HH12 | 12 小时制（00—12） |
| HH24 | 24 小时制（00—24） |
| MI | 分（00——59） |

| 元素 | 含义（范围） |
| --- | --- |
| SS | 秒（00—59） |
| FF[n] | 亚秒，n 的取值范围为 1—6 |

缺省的转换形式为：YYY-MM-DD HH:MI:SS。

示例

SELECT TO_CHAR（SYSDATE, "YYYYMMDD") FROM DUAL;

返回结果为：20180520

SELECT TO_CHAR（SYSDATE, "YYYY-MM-DD HH24:MI:SS") FROM DUAL;

返回结果为：2018-05-20 14:30:28

SELECT TO_CHAR（SYSDATE, "YYYY/MM/DD HH:MI:SS") FROM DUAL;

返回结果为：2018/05/20 02:30:30

**TO_CHAR(number)：转换数值型表达式**

TO_CAHR（number）将数值型数据 ***num_expr*** 转换为 ***format_string*** 参数指定的格式的字符串。

其用法与 GBASE_TO_CHAR() 函数的转换数值表达式为字符串的用法相同。具体信息，请参见 GBASE_TO_CHAR() 函数。

## *GBASE_TO_DATE 函数*

**GBASE_TO_DATE** 函数将字符串转换为 DATETIME 值。该函数根据 ***format_string*** 第二个参数指定的日期格式，将 ***char_expression*** 第一个参数求值为日期，并返回等同的日期。

如果 ***char_expression*** 为 NULL，则返回 NULL 值。

**GBASE_TO_DATE** 函数的任何参数都必须为内建的数据类型。

如果您省略 ***format_string*** 参数，则 **GBASE_TO_DATE** 函数将缺省的 DATETIME 格式应用于该 DATETIME 值。通过 **GL_DATETIME** 环境变量指定缺省的 DATETIME 格式。

在下列示例中，用户想要将字符串转换为 DATETIME 值，以便以转换了的值来更新 tab1 表的 begin_date 列。begin_date 列定义为 DATETIME YEAR TO SECOND 数据类型。用户使用包含 **GBASE_TO_DATE** 函数的 UPDATE 语句来实现此结果：

```
UPDATE tab1
SET begin_date = GBASE_TO_DATE('Wednesday July 25, 2007 18:45','%A %B %d, %Y %R');
```

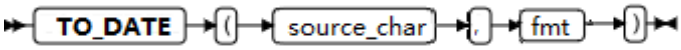
在此示例中的 ***format_string*** 参数告诉 **GBASE_TO_DATE** 函数如何格式化 **begin_date** 列中转化了的字符串。要获取展示在此格式字符串中每一格式符号的作用的表格，请参阅 TO_CHAR 函数。

## *TO_DATE 函数*

**TO_DATE** 函数将字符串转换为 DATETIME 数据类型。支持转换的字符型数据包括：CHAR、VARCHAR2、NCHAR 或 NVARCHAR2。

**TO_DATE** 函数有此语法：

TO_DATE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *source_char* | 要被转换为 DATETIME 类型的源字符串。 | 设置为空字符串或 NULL 时，返回结果为空 | 引用字符串 |
| ***fmt*** | DATETIME 类型的格式化字符串 | 设置为空字符串或 NULL 时，返回结果为空。 | 引用字符串 |

该函数根据 ***fmt*** 第二个参数指定的日期格式，将 ***source_char*** 第一个参数求值为日期，并返回等同的日期。

此函数的第一个参数 *source_char*（源字符串）支持公元、年、月、日、时、分、秒、亚秒等时间格式。通常『年-月-日 时:分:秒』具有的格式如下：

1) 年度部分（YYYY）：支持 1~4 位有效数字；
2) 月份部分（MM）：取值范围为[0,12]，0~9 支持设置为 00~09；
3) 日期部分（DD）：取值范围为[0,31]，0~9 支持设置为 00~09；
4) 小时部分（HH/HH12/HH24）：hh/hh12 取值范围为[0,12]，hh24 取值范围为[0,24]，0~9 均支持设置为 00~09；
5) 分钟部分（MI）：取值范围为[0,59]，0~9 支持设置为 00~09；
6) 秒钟部分（SS）：取值范围为[0,59]，0~9 支持设置为 00~09。

此函数的第二个参数 ***fmt***（格式化串）设置为：『YYYY-MM-DD HH:MI:SS』，连接字符与源字符串相对应，不区分大小写。使用时，用单（双）引号包围。

在下列示例中，用户输入 SELECT 查询语句，对 TO_DATE 函数进行查询，将字符串转换为 DATETIME 类型。

```
select to_date('2017-07-21 13:33:24', 'YYYY-MM-DD HH:MI:SS');
```

返回结果为：

```
2017-07-21 13:33:24
```

如果设置省略参数中对应的元素，则执行结果通过 to_char 函数展示时包括年、月、日、时、分、秒、亚秒全部信息，显示形式为：年-月-日 时:分:秒.亚秒（2017-07-01 00:00:00.000000）。缺省信息系统自动补齐。缺省‘年’则补齐当前年；缺省‘月’则补齐当前月；缺省‘日’则补齐第一天；缺省‘时’、‘分’、‘秒’则补齐‘00’；缺省‘亚秒’则补齐‘000000’。

例如，在以下示例中，对 TO_DATE 函数进行查询。

select to_date('07-21', 'MM-DD');

返回结果如下：

2017-07-21 00:00:00.000000

又如，以下 SQL 语句，对于缺省的‘年、月、日’补齐当前年的当前月的第一天：

select to_date('13:33', 'HH:MI');

返回结果如下：

2017-10-01 13:33:00.000000

如果省略 ***fmt*** 参数（格式化串），只有当 ***source_char*** 参数（源字符串）格式为‘年月日’、‘年月日时分’、‘年月日时分秒’时，TO_DATE 函数执行成功。否则函数执行失败，系统提示错误。

例如，在以下示例中对 TO_DATE 函数进行查询，设置省略 ***fmt*** 参数（格式化串）。

select to_date('2017-07-21 13:33:24');

返回结果如下：

2017-07-21 13:33:24.000000

以下语句执行失败。

select to_date('07-21 13:33:24');

***source_char*** 参数（源字符串）与 ***fmt*** 参数（格式化串）还可以设置为嵌套函数、表达式、字符数据类型列。例如，在以下示例中，将 TO_DATE 函数的第二个参数设置为 CAST 表达式:

select (to_date('10-12 23:56:33', cast('mm-ddhh12:mi:ss'as char(30))) ;

返回结果如下：

2017-10-12 23:56:33.000000

如果 ***source_char*** 参数（源字符串）与 ***fmt*** 参数（格式化串）之间存在连接字符省略的情况，则当源字符串和格式化串省略的位置相同时，TO_DATE 函数执行成功，系统返回正确的结果。例如，

select (to_date('20110911 23:11:24','YYYYMMDD HH:MI:SS ');

返回结果如下：

2011-09-11 23:11:24.000000

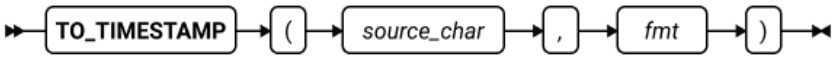
而当源字符串的连接字符省略，对应的格式化串中的连字符未省略时，函数执行失败，系统返回错误。例如，以下查询语句执行失败：

select to_date('201707-21', 'YYYY-MM-DD HH:MI:SS');

此外，如果将这两个参数的连接字符设置为数字、字母，TO_DATE 函数会执行失败，系统返回错误。

## *TO_TIMESTAMP 函数*

**TO_TIMESTAMP** 函数将字符串转换为 TIMESTAMP 数据类型。支持转换的字符型数据包括：CHAR、VARCHAR、NCHAR 或 NVARCHAR。




该函数根据第二个参数 fmt 指定的日期格式，将第一个参数 source_char 求值为 TIMESTAMP 类型值。

**TO_TIMESTAMP** 函数的使用方法与 TO_DATE 函数的用法一致。

例如，将字符串转换为 TIMESTAMP 类型值：

```
SELECT TO_TIMESTAMP('2021-10-16 13:33:24', 'YYYY-MM-DD HH:MI:SS') FROM DUAL;
```

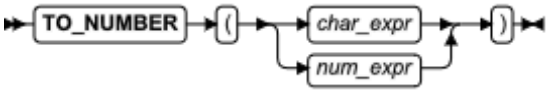
返回结果为：2021-10-16 13:33:24.00000

## TO_NUMBER 函数

**TO_NUMBER** 函数可将表示数值值的数值或字符表达式转换为 DECIMAL 数据类型。

**TO_NUMBER** 函数有此语法：

TO_NUMBER 函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *char_expression* | 要被转换为 DECIMAL 值的表达式 | 必须为文字、主变量、表达式或字符数据类型的列 | 表达式 |
| *num_expression* | 求值为实数值的表达式 | 必须返回数值的数据类型 | 表达式 |

**TO_NUMBER** 函数将它的参数转换为 DECIMAL 数据类型。该参数可为数值或数值表达式的字符串表示。

下列示例检索 TO_NUMBER 函数从 MONEY 值的文字表示返回的 DECIMAL 值：

```
SELECT TO_NUMBER('$100.00') from mytab;
```

下表展示此 SELECT 语句的输出。

| (expression) |
| --- |

| |
|---|
| 100.000000000000 |

在此示例中，从 '$100.00' 字符串舍弃币种符号。

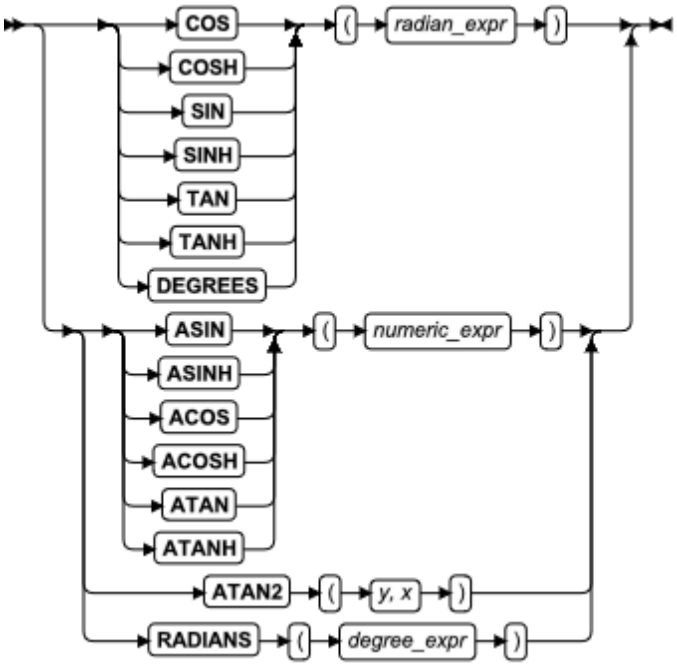
在大多数上下文中不需要 **TO_NUMBER** 函数，因为在缺省情况下， GBase 8s 将包括小数点的数值（以及以有小数点的文字数值为格式的引用字符串）转换为 DECIMAL 数据类型。然而，当您正在迁移那些原本为其他数据库服务器编写的 SQL 应用时，如果该应用调用返回 DECIMAL 值的此名称的函数，则此函数有用。

## 三角函数

内建的三角函数计算直角三角形的边的长度的比率。**DEGREES** 和 **RADIANS** 这两个支持函数可分别将角度值的单位从弧度转换为角度，以及从角度转换为弧度。

内建的三角函数有下列语法。

三角函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *degree_expr* | 表示角度的数值的表达式 | 必须返回可被转换为 DECIMAL 类型的值 | 表达式 |
| *numeric_expr* | 作为 ASIN、ACOS、ATAN、ASINH、ACOSH 或 ATANH 函数的参数的表达式 | 必须返回在 -1 与 1 之间（包括 -1 和 1）的值 | 表达式 |
| *radian_expr* | 表示弧度的数值的表达式 | 必须返回数值值 | 表达 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | | 式 |
| *x* | 在直角坐标对（*x*, *y*）中表示 x 坐标的表达式 | 必须返回数值值 | 表达式 |
| *y* | 在直角坐标对（*x*, *y*）中表示 y 坐标的表达式 | 必须返回数值值 | 表达式 |

后面的部分描述每一这些内建的三角函数。

## *COS 函数*

**COS** 函数返回弧度表达式的余切。

下列示例返回 anglestbl 表中角度列的值的余弦。在此示例中传递到 COS 函数的表达式将角度转换为弧度。

```
SELECT COS(degrees*180/3.1416) FROM anglestbl;
```

## *COSH 函数*

**COSH** 函数返回所需要的参数的双曲余弦，在此，该参数是以弧度表示的角度。

COSH 函数


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *radian_expr* | 求值为以弧度的单位计的角度的表达式 | 必须为数值数据类型 | 表达式 |

下列示例返回 anglestbl 表的角度列中的值的双曲余弦。传递到 COSH 函数的表达式将角度转换为弧度。

```
SELECT COSH(degrees*180/3.1416) FROM anglestbl;
```

## *SIN 函数*

**SIN** 函数返回您指定作为它的弧度表达式参数的角的正弦。

下列查询返回 anglestbl 表的 radians 列的每一行中值的正弦：

```
SELECT SIN(radians) FROM anglestbl;
```

## *SINH 函数*

SINH 函数返回参数的双曲正弦，在此，该参数是以弧度表示的角。

SINH 函数


下列示例返回 anglestbl 表的角度列中值的双曲正弦。传递到 SINH 函数的表达式将角度转换为弧度。

```
SELECT SINH(degrees*180/3.1416) FROM anglestbl;
```

## *TAN 函数*

**TAN** 函数返回它的弧度表达式参数的正切的值。

此示例返回 anglestbl 表的 radians 列中的值的正切：

```
SELECT TAN(radians) FROM anglestbl;
```

## *TANH 函数*

TANH 函数返回参数的双曲正切，该参数是以弧度表示的角。

TANH 函数


下列示例返回 anglestbl 表的角度列中值的双曲正切。传递给 TANH 函数的表达式将角度转换弧度。

```
SELECT TANH(degrees*180/3.1416) FROM anglestbl;
```

## *ACOS 函数*

**ACOS** 函数返回数值表达式的反余弦。

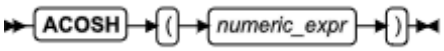
下列示例返回弧度值（-0.73）的反余弦：

```
SELECT ACOS(-0.73) FROM anglestbl;
```

## *ACOSH 函数*

ACOSH 函数返回指定的数值输入的双曲反余弦。

ACOSH 函数




## *ASIN 函数*

**ASIN** 函数返回数值表达式参数的反正弦。

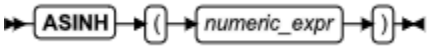
下列示例返回弧度值（-0.73）的反正弦：

SELECT ASIN(-0.73) FROM anglestbl;

## *ASINH 函数*

ASINH 函数返回指定的数值输入的反双曲正弦。

ASINH 函数




## *ATAN 函数*

**ATAN** 函数返回数值表达式的反正切。

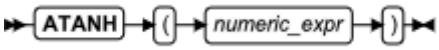
下列示例返回弧度值（-0.73）的反正切：

SELECT ATAN(-0.73) FROM anglestbl;

## *ATANH 函数*

ATANH 函数返回指定的数值输入的反双曲正切。

ATANH 函数




## *ATAN2 函数*

**ATAN2** 函数计算与 (***x, y***) 相关的极坐标 (***r***, q) 的角度分量。

下列示例将 *angles* 与直角坐标 (4, 5) 的 q 相比较：

```
WHERE angles > ATAN2(4,5)        --确定 (4,5) 的 q 并
                                 --与 angles 对比
```

您可使用下列示例所示的表达式来确定径向坐标 ***r*** 的长度：

```
SQRT(POW(x,2) + POW(y,2))        --确定 (x,y) 的 r
```

您可使用下列示例所示的表达式来确定直角坐标 (4,5) 的径向坐标 ***r*** 的长度：

```
SQRT(POW(4,2) + POW(5,2))        --确定 (4,5) 的 r
```

## *DEGREES 函数*

使用 **DEGREES** 函数来将表示弧度数值的表达式或主变量的值转换为等同的角度值。

作为此函数的唯一参数的 ***radian_expression*** 或主变量必须为数值数据类型（或可转换为数值的非数值数据类型），数据库服务器以弧度的单位求值该参数，并转换为角度的单位。

返回值为类型 DECIMAL (32, 255) 的数值。

在下列两个示例中，**DEGREES** 的参数求值为 6 弧度，且返回值为 343.774677078494 角度：

```
EXECUTE FUNCTION DEGREES (6);
```

```
EXECUTE FUNCTION DEGREES ("6");
```

DEGREES 函数根据下列公式将弧度转换为角度：

(number of radians) * (180/pi) = (number of degrees)

在此，pi 代表圆的周长对它的直径的比率。使用超越数值 pi 作为有理数的除数进行算术计算的结果往往包括舍入错误。

### *RADIANS 函数*

使用 **RADIANS** 函数来将表示角度值的表达式或主变量转换为等同的弧度值。

返回值为类型 DECIMAL (32, 255) 的数值。

作为此函数的唯一参数的 *degree_expression* 必须有数值数据类型（或可转换为数值的非数值数据类型），数据库服务器以角度的单位求值该参数，并转换为弧度的单位：

```
EXECUTE FUNCTION RADIANS (100);
EXECUTE FUNCTION RADIANS ("100");
```

在上述两个示例中，RADIANS 参数求值为 100 度，且返回值为 1.745328251994 弧度。您可使用 RADIANS 函数表达式作为 COS、SIN 或 TAN 函数的参数，来返回那个角各自的三角值：

```
COS(RADIANS (100))
SIN(RADIANS ("100"))
TAN(RADIANS (100))
```

RADIANS 函数根据下列公式将角度转换为弧度：

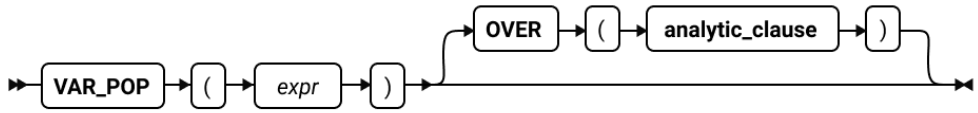
(number of degrees) * (pi/180) = (number of radians)

在此，pi 代表圆的周长对它的直径的比率。使用超越数值 pi 作为有理数的除数进行算术计算的结果往往包括舍入错误。

## 数字函数

### *VAR_POP*

返回输入值的方差，返回值类型为decimal。计算公式为(SUM(expr2) - SUM(expr)2 / COUNT(expr)) / COUNT(expr)。

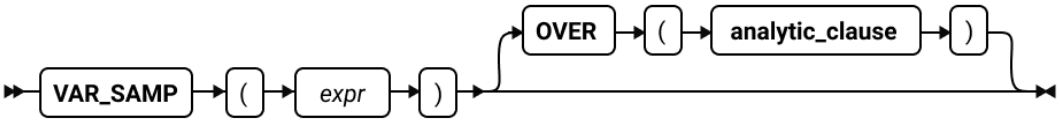



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。参数为包含列的表达式且值包含 null 时，null | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 所在的行不纳入计算。参数为 null 函数返回值为空。 | |

### *VAR_SAMP*

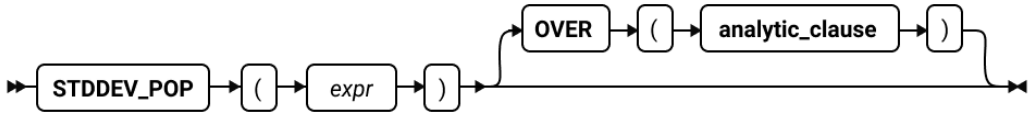
返回一组数据的样本方差，返回值类型为decimal。计算公式为(SUM(expr - (SUM(expr) / COUNT(expr)))2) / (COUNT(expr) - 1) 。




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。参数为包含列的表达式且值包含 null 时，null 所在的行不纳入计算。参数为 null 函数返回值为空。 | 表达式 |

### *STDDEV_POP*

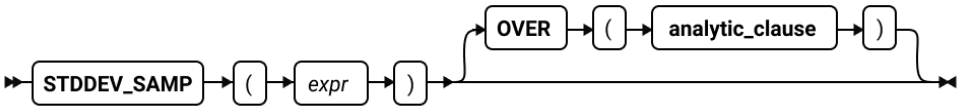
返回输入值的总体标准差，返回值类型为decimal。计算公式为VAR_POP函数值的算术平方根。




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。参数为包含列的表达式且值包含 null 时，null 所在的行不纳入计算。参数为 null，函数返回值为空。 | 表达式 |

### *STDDEV_SAMP*

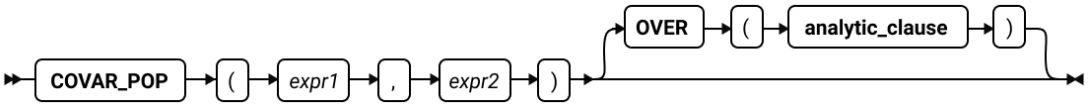
累积样本标准差并返回样本方差的平方根，返回值类型为decimal。计算公式为VAR_SAMP函数的算术平方根。




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。参数为包含列的表达式且值包含 null 时，null 所在的行不纳入计算。参数为 null，函数返回值为空。 | 表达式 |

## *COVAR_POP*

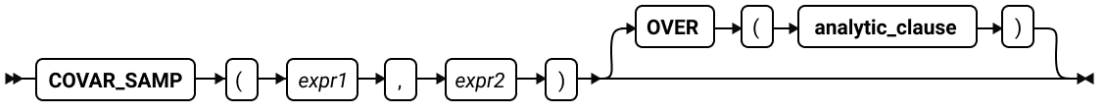
返回一组数对的总体协方差，返回值类型为decimal。计算公式为(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n。




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。任何一个参数，值包含 null 时，null 所在的行不纳入计算。两个参数同时为 null，函数返回值为空。 | 表达式 |
| ***n*** | 表达式 | 指 expr1,expr2 两个表达式值中同时不为 null 的记录数。 | 表达式 |

## *COVAR_SAMP*

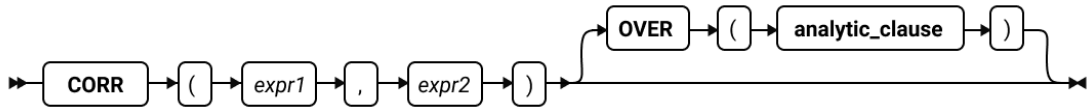
返回一组数对的样本协方差，函数返回值类型为decimal。计算公式为(SUM(expr1 * expr2) - SUM(expr1) * SUM(expr2) / n) / (n-1)。




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 参数为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。任何一个参数，值包含 null 时，null 所在的行不纳入计算。两个参数同时为 null，函数返回值为空。 | 表达式 |
| ***n*** | 表达式 | 指 expr1,expr2 两个表达式值中同时不为 null 的记录数。 | 表达式 |

### *CORR*

返回一组数对的相关系数，函数返回值类型为decimal。计算公式为COVAR_POP(expr1, expr2) / (STDDEV_POP(expr1) * STDDEV_POP(expr2))。



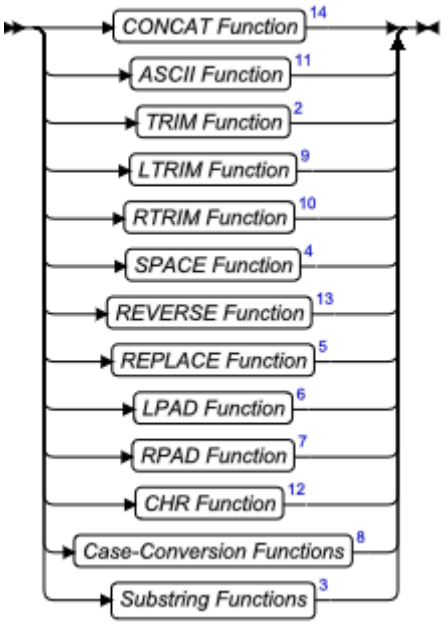

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***expr*** | 表达式 | 为数值型或能通过隐式类型转换为数值型的表达式，其他数据类型报错。任何一个参数，值包含 null 时，null 所在的行不纳入计算。两个参数同时为 null，函数返回值为空。 | 表达式 |

## 字符串操纵函数

字符串操纵函数对字符串执行各种操作。

在下图中标识字符串操纵函数：

字符串操纵函数

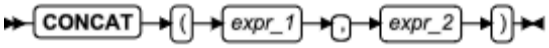



后面的部分描述每一内建的字符串操纵函数。

## *CONCAT 函数*

CONCAT 函数接受两个表达式作为参数，并返回将由它的第二个参数返回的值的字符串表示追加到由它的第一个参数返回的值的字符串表示之后的单个字符串。

CONCAT 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr_1*、*expr_2* | 要将它们的值的字符串表示串联的表达式 | 不可返回复合的、用户定义的或大对象类型。如果是主变量，则它必须有足够的长度来存储组合字符串的结果。 | 表达式 |

**CONCAT** 函数的每一参数可求值为字符、数值或时间数据类型。如果某个或两个被串联的参数为空，则该函数返回 NULL 值。

与 GBase 8s 的其他内建的字符串操纵函数不一样，不可重载 **CONCAT** 函数。

**CONCAT** 是串联（||）运算符的运算符函数。对于给定的表达式参数对，**CONCAT** 返回的字符串与从同一表达式为运算对象的运算符返回的字符串相同。要获取关于串联操作，以及对您在其中调用 **CONCAT** 函数的 SQL 和动态 SQL 语句的限制的附加信息，请参阅 串联运算符。

## *来自 CONCAT 和字符串函数的返回类型*

从成功的 **CONCAT** 函数调用（或从串联（||）操作符，或对于确定他们的返回类型所遵循的规则与 **CONCAT** 相同的其他内建的字符串操纵函数）的返回值的数据类型依赖于参数的数据类型以及结果字符串的长度。在确定返回类型时，两个参数的顺序的意义不大。

对于来自串联多个数据类型指定的值的串联操作的返回类型， GBase 8s 应用下列规则：

- 如果其中一个类型为“国家语言支持”（也就是 NCHAR 和 NVARCHAR）：
  - 返回类型为 NVARCHAR。
- 如果其中一个参数为 VARCHAR 或数值类型，则
  - 返回类型为 VARCHAR。
- 然而，在某些在本地执行远程例程的跨服务器操作中，可发生针对这些规则的例外，且在将串联表达式的返回值发送到远程数据库服务器之前，在本地对它求值。对于不支持在分布式事务中的 LVARCHAR 数据类型的远程服务器，如果发送 LVARCHAR 类型返回错误，则作为 CHAR 数据类型发送串联的结果。

在下列表格中，各行罗列 **CONCAT** 函数的第一个参数的有效的数据类型，各列罗列第二个参数的类型。每一行与列交叉处的单元展示可能的一个或多个返回类型。标明**其他**的行和列表示求值为非字符类型的参数，诸如数值或像 DECIMAL 或 DATE 一样的时间数据类型。

表 1. 来自两个参数上的操作的返回类型

| | NCHAR | NVARCHAR | CHAR | VARCHAR | LVARCHAR | 其他 |
|---|---|---|---|---|---|---|
| NCHAR | nchar | nvarchar *或* nchar | nchar | nvarchar *或* nchar | nvarchar *或* nchar | nvarchar *或* nchar |
| NVARCHAR | nvarchar *或* nchar | nvarchar *或* nchar | nvarchar *或* nchar | nvarchar *或* nchar | nvarchar *或* nchar | nvarchar *或* nchar |
| CHAR | nchar | nvarchar *或* nchar | char | varchar *或* lvarchar | lvarchar | varchar *或* lvarchar |
| VARCHAR | nvarchar *或* nchar | nvarchar *或* nchar | varchar *或* lvarchar | varchar *或* lvarchar | lvarchar | varchar *或* lvarchar |
| LVARCHAR | nvarchar *或* nchar | nvarchar *或* nchar | lvarchar | lvarchar | lvarchar | lvarchar |
| 其他 | nvarchar *或* nchar | nvarchar *或* nchar | varchar *或* lvarchar | varchar *或* lvarchar | lvarchar | varchar *或* lvarchar |

对于 **CONCAT** 之外的其他字符串操纵函数，DATE、DATETIME 或 MONEY 数据类型的参数往往返回 NVARCHAR 或 NCHAR 值，这依赖于结果字符串的长度。

此表格是对称的，因为参数的顺序对返回数据类型没有影响。对于内建的字符串操纵函数或运算符，用户定义的数据类型、大对象类型、复合的类型和其他扩展的数据类型不是有效的参数。

此表格还描述使用串联（||）运算符的表达式的返回数据类型。

对于返回类型提升，下列字符串操纵函数支持与 CONCAT 相同的规则：

- **LPAD**
- **RPAD**
- **REPLACE**
- **SUBSTR**
- **SUBSTRING**
- **TRIM**
- **LTRIM**
- **RTRIM**

下列表格总结 GBase 8s 如何基于参数类型，来确定来自这些字符串操纵函数的返回类型：

表 2. 支持返回类型提升的字符串操纵函数

| 函数 | 如何确定函数的返回类型 |
|---|---|
| CONCAT、\|\| | 返回类型基于两个参数。请参考 表 1。 |
| SUBSTR、SUBSTRING | 返回类型与 *source string* 类型相同。如果 *source string* 为主变量，则依赖于结果的长度，返回类型为 NVARCHAR 或 NCHAR。 |
| TRIM、LTRIM、RTRIM | 返回类型依赖于源类型和返回的长度：<br>• NVARCHAR *返回* NVARCHAR<br>• VARCHAR *返回* VARCHAR<br>• CHAR *返回* VARCHAR（如果 length <= 255 字节的话）<br>• CHAR *返回* LVARCHAR（如果 length > 255 字节的话）<br>• NCHAR *返回* NVARCHAR（如果 length <= 255 字节的话）<br>• NCHAR *返回* LVARCHAR（如果 length > 255 字节的话）<br>• LVARCHAR *返回* LVARCHAR |
| LPAD、RPAD | 返回类型基于 *source_string* 和 *pad_string* 参数。如果未指定 *pad_string*，则返回类型基于 *source_string* 的数据类型。 |
| REPLACE | 返回类型基于 *source_string* 和 *old_string* 参数（以及基于 *new_string* 参数，如果指定了的话）。如果任何参数为主变量，则返回类型为 NCHAR。 |
| ENCRYPT_AES、ENCRYPT_TDES、DECRYPT_BINARY、DECRYPT_CHAR | 对于不是 BLOB 或 CLOB 变量的参数，返回类型基于 *data* 和 *encrypted_data* 参数的数据类型。请参考 表 1。 |

**在 NLSCASE INSENSITIVE 数据库中的数据类型提升**

在有 NLSCASE INSENSITIVE 属性的数据库中，数据库服务器丢弃 NCHAR 和 NVARCHAR 值的大写字母。通过执行隐式的强制转型来在其中避免函数或运算符溢出错误的表达式，可产生与区

分大小写的数据库会返回的结果不同的结果，如果该表达式求值为 NCHAR 或 NVARCHAR 数据类型的话。

当数据库服务器在其上支持数据类型提升的字符串函数或字符串运算符返回一值，对于该表达式的缺省的 VARCHAR 或 NVARCHAR 数据类型会产生溢出错误时，数据库服务器在返回值上执行隐式的强制转型，如同主题 来自 CONCAT 和字符串函数的返回类型 的第一张表格表明的那样：

- 如果没有参数或运算对象是 NCHAR 或 NVARCHAR 数据类型，则表达式求值为 CHAR、LVARCHAR 或 VARCHAR 数据类型。
- 如果任何参数或运算对象是 NCHAR 或 NVARCHAR 数据类型，则表达式求值为 NCHAR 或 NVARCHAR 数据类型。

在有 NLSCASE INSENSITIVE 属性的数据库中，对 CHAR、LVARCHAR 或 VARCHAR 数据类型的操作是区分大小写的，但对 NCHAR 或 NVARCHAR 数据类型的操作不区分大小写。数据类型提升还从包括 CHAR、LVARCHAR 或 VARCHAR 分量的表达式的求值产生不区分大小写的结果（而不是区分大小写），如果同一表达式还包括 NCHAR 或 NVARCHAR 字符串的话。

下列示例说明在 NLSCASE INSENSITIVE 数据库中的此行为，其中的表 t1 有五个字符列，对应五种内建的字符数据类型。该表存储三行，其中的每一列存储 3 个字母字符串的同一字母大小写变量：

```
CREATE DATABASE db NLSCASE INSENSITIVE;
        CREATE TABLE t1 (
        c1 NCHAR(20),
        c2 NVARCHAR(20),
        c3 CHAR((20),
        c4 VARCHAR(20),
        c5 LVARCHAR(20)) ;
INSERT INTO t1 values ('gbase', 'gbase', 'gbase', 'gbase', 'gbase');
INSERT INTO t1 values ('Gbase', 'Gbase', 'Gbase', 'Gbase', 'Gbase');
INSERT INTO t1 values ('GBASE', 'GBASE', 'GBASE', 'GBASE', 'GBASE');
```

下列查询使用其字母均为小写的文字字符串的相等谓词，从 NCHAR 列检索值：

```
SELECT c1 FROM t1 WHERE c1 = 'gbase';
```

由于在此数据库中 NCHAR 值不区分大小写，因此该查询从每一行返回列 c1 值：

```
c1
gbase
Gbase
GBASE
```

下列对同一表的查询从 WHERE 子句将其强制转型为 NCHAR 值的 CHAR 列 **c3**，返回相同的不区分大小写的结果：

```
SELECT c1 FROM t1 WHERE c3 = 'gbase'::NCHAR(10);
```

在强制转型之后，c3 值不区分大小写，因此，c3 中的每行都与字符串 ’gbase’ 相匹配，且对于 c1 中的每行，WHERE 条件都为真：

```
c1
gbase
```

Gbase
GBASE

如同在前面的示例中那样，在相同的序列中出现相同的字母的字符串之中，由于不区分大小写的操作丢弃字母大小写的差异，因此在有 NLSCASE INSENSITIVE 属性的数据库中，请留意避免将不区分大小写的规则应用到您期望区分大小写的操作的数据类型提升上下文。

另请参阅 在 NLSCASE INSENSITIVE 数据库中重复的行 部分。

## *在分布式事务中的返回字符串类型*

在访问同一 GBase 8s 实例的不同数据库中的表的跨数据库分布式查询中，通过来自 CONCAT 函数的返回类型部分描述的 **CONCAT**（以及通过遵循相同的返回类型提升的其他内建的字符串操纵函数）返回相同的类型。

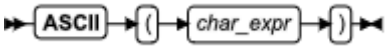
在跨服务器的分布式查询中也返回相同的类型。

对于的 GBase 8s 的所有版本，如果返回的字符串的长度超出 23 KB，则发出错误 -881。

## *ASCII 函数*

ASCII 函数基于在 ASCII 字符集中它的代码点，返回字符串中第一个字符的十进制表示。

ASCII 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *char_expr* | 求值为字符数据类型的表达式 | 必须为类型 CHAR、LVARCHAR、NCHAR、NVARCHAR 或 VARCHAR | 标识符 |

**ASCII** 函数采用任何字符数据类型的单个参数。它基于参数的第一个字符返回一个整数值，对应于在 ASCII 字符集内那个字符的代码点的十进制表示。

如果参数为 NULL，或如果参数为空串，则 **ASCII** 函数返回 NULL 值。

下列查询返回大写 H 的 ASCII 值：

```
SELECT ASCII("HELLO") FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
|---|
| 72 |

下列查询返回小写 h 的 ASCII 值：

```
SELECT ASCII("hello") FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
| --- |
| 104 |

下列查询从空字符串参数返回 ASCII 输出：

```
SELECT ASCII("") FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的 NULL 输出。

| (constant) |
| --- |
| |

下列查询返回从 NULL 参数的 ASCII 输出：

```
SELECT ASCII(NULL) FROM systables WHERE tabid = 1;
```

下列表格展示此 SELECT 语句的 NULL 输出。

| (constant) |
| --- |
| |

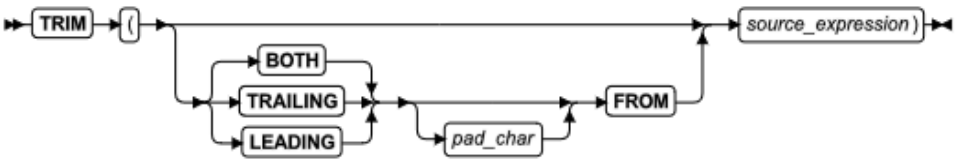
ASCII 函数将此参数解释为 NULL 表达式，而不是为以大写 N 开头的值。

要获取 ASCII 字符集中代码点的数值值的表格，请参阅 U.S. English 数据的排序顺序。

## *TRIM 函数*

TRIM 函数从字符串移除指定的开头或末尾的填充字符。（另请参阅 LTRIM 和 RTRIM 函数的描述，这两个函数提供类似的功能，但支持不同的语法。）

TRIM 函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *pad_char* | 求值为单个字符或 NULL 的表达式。缺省值为空格（= ASCII 32）。 | 必须为字符表达式 | 表达式 |
| *source_expression* | 字符表达式，包括字符列名，或对另一 TRIM 函数的调用 | 不可为 DISTINCT 数据类型 | 表达式 |

**TRIM** 函数返回一个与它的 ***source_expression*** 参数相同的字符串，除了删除由 LEADING、TRAILING 或 BOTH 关键字指定的开头或末尾填充字符之外。如果未指定修正限定符（LEADING、

TRAILING 或 BOTH），则缺省值为 BOTH。如果未指定 ***pad_char***，则缺省值为单个空格（ASCII 32 字符），并从返回的值删除由限定的关键字指定的开头的或末尾的空格。

如果 ***pad_char*** 或 ***source_expression*** 求值为 NULL，则 **TRIM** 函数的结果为 NULL。

返回的值的数据类型依赖于 *source_expression* 参数：

- 如果参数长于 255 字节，则返回的值为 LVARCHAR 类型。
- 如果参数有 255 字节或更少，则返回的值的数据类型依赖于参数的数据类型：
  - 如果参数是 CHAR 或 VARCHAR 类型，则返回 VARCHAR 值。
  - 如果参数是 NCHAR 或 NVARCHAR 类型，则返回 NVARCHAR 值。
  - 如果参数是 LVARCHAR 类型，则返回 LVARCHAR 值。

下列示例展示 TRIM 函数的一些一般使用：

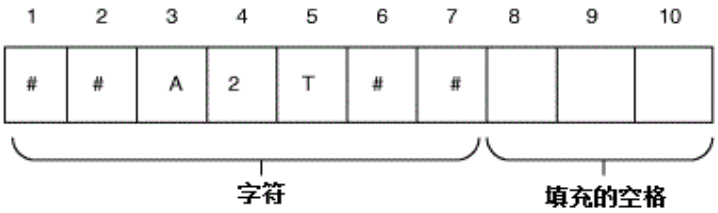
```
SELECT TRIM (c1) FROM tab;
SELECT TRIM (TRAILING '#' FROM c1) FROM tab;
SELECT TRIM (LEADING FROM c1) FROM tab;
UPDATE c1='xyz' FROM tab WHERE LENGTH(TRIM(c1))=5;
SELECT c1, TRIM(LEADING '#' FROM TRIM(TRAILING '%' FROM
        '###abc%%%')) FROM tab;
```

在动态 SQL 中，当您随同在 Projection 子句中调用 **TRIM** 函数的 SELECT 语句来使用 DESCRIBE 语句时，对于在 GBase 8s ESQL/C 源文件的 **sqltypes.h** 头文件中定义的 SQL 数据类型常量，DESCRIBE 返回的修正的列的数据类型依赖于 ***source_expression*** 的数据类型。要获取关于 GBase 8s ESQL/C 中 **TRIM** 函数在 GLS 方面的更多信息，请参阅 *GBase 8s GLS 用户指南*。

**固定的字符列**

可在定长字符列上指定 TRIM 函数。如果字符串的长度未完全地填满，则以空格填充未使用的字符。图 1 展示对于列条目 '##A2T##' 的此概念，在此，定义该列为 CHAR(10)。

*图: 在定长字符列的列条目*

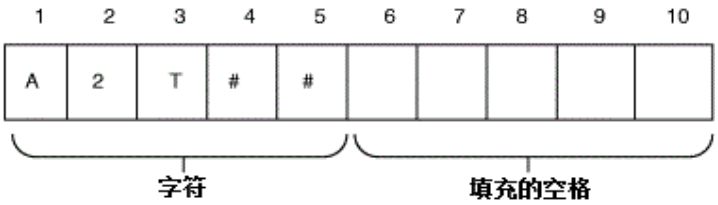



如果您想要从该列修整掉井号（#）***pad_char***，则需要考虑填充的空格以及实际的字符。

例如，如果您指定关键字 BOTH，则修整操作的结果为 A2T##，因为 TRIM 函数与跟在字符串之后的修整的空格不匹配。在此情况下，仅修整在其他字符之前的那些井号（#）。图 2 跟着的 SELECT 语句显示结果。

```
SELECT TRIM(LEADING '#' FROM col1) FROM taba;
```

*图: TRIM 操作的结果*




此 SELECT 语句移除所有出现的井号（#）：

```
SELECT TRIM(BOTH '#' FROM TRIM(TRAILING ' ' FROM col1)) FROM taba;
```

## *LTRIM 函数*

LTRIM 函数从字符串移除指定的开头填充字符。

LTRIM 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *pad_string* | 指定一个或多个要从 *source_string* 删除的字符的表达式 | 必须为字符表达式 | 表达式 |
| *source_string* | 指定要从其删除 *pad_string* 中的字符的字符串的表达式 | 不删除不在 *pad_string* 中任何字符右边的填充字符 | 表达式 |

**LTRIM** 函数的第一个参数必须为要从其删除开头填充字符的字符表达式。可选的第二个参数是求值为填充字符的字符串的字符表达式。如果未提供第二个参数，则仅将空字符作为填充字符。

**LTRIM** 函数的返回数据类型是基于它的 source_string 参数的，使用 来自 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

返回的值包含 ***source_string*** 的子字符串，但已移除了第一个非填充字符左边的任何开头填充字符。如果使用主变量，则返回 LVARCHAR 数据类型。

**LTRIM** 函数从左边扫描 ***source_string*** 的副本，删除出现在 ***pad_string*** 中的任何开头字符。如果未指定 ***pad_string*** 参数，则仅从返回的值删除开头空格。当遇到第一个非填充字符时，该函数返回它的结果字符串并终止。

在下列示例中，*pad_string* 为 'Hello'：

```
SELECT LTRIM('Hellohello world!', 'Hello') FROM mytab;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
| --- |
| hello world! |

在此，删除了 *source_string* 的前五个字符，因为它们与 *pad_string* 中的字符相匹配，但在该函数遇到了小写字母 h 字符之后终止，其保留了它右边的末尾 'ello' 填充字符。

## *RTRIM 函数*

RTRIM 函数从字符串移除指定的末尾填充字符。

RTRIM 函数


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *pad_string* | 指定要从 *source_string* 删除的一个或多个字符的表达式 | 必须为字符表达式 | 表达式 |
| *source_string* | 指定从其删除 *pad_string* 中的字符的字符串的表达式 | 不删除不在 *pad_string* 中的任何字符左边的填充字符 | 表达式 |

**RTRIM** 函数的第一个参数必须为从其删除末尾填充字符的字符表达式。可选的第二个参数是求值为填充字符的字符串的字符串表达式。如果未提供第二个参数，则仅将空字符作为填充字符。

**LTRIM** 函数的返回数据类型是基于它的 ***source_string*** 参数的，使用 来自 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

返回的值包含 ***source_string*** 的子字符串，但已从其移除了第一个非填充字符右边的任何末尾填充字符。如果使用主变量，则返回 LVARCHAR 数据类型。

**RTRIM** 函数从右边扫描 ***source_string*** 的副本，删除出现在 ***pad_string*** 中的任何末尾字符。如果未指定 ***pad_string*** 参数，则仅从返回的值删除末尾空格。当遇到第一个非填充字符时，该函数返回它的结果字符串并终止。

在下列示例中，*pad_string* 为 'theend!*#?'：

```
SELECT RTRIM('good night...     *!#?theend ', ' theend!*#?') AS closing FROM mytab;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
| --- |

| good night... |
|---|

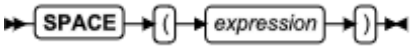
在此，删除了 *source_string* 的最后十五个字符，因为它们与 *pad_string* 中的字符相匹配，但该函数在遇到了句号（.）字符之后终止了，保留了左边的开头的 'thn' 填充字符。

## *SPACE 函数*

SPACE 函数创建指定的空格数量的字符串。返回的字符串值的最大长度可为 32,739 空字符。

该函数有此语法：

SPACE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 求值为非负整数 < 256 的表达式 | 必须为表达式、常量、列或内建的整数类型的主变量，或可转化为整数的表达式 | 表达式 |

SPACE 函数的参数必须为内建的数据类型。

SPACE 函数返回指定数目的空（ASCII 32）字符的 LVARCHAR 字符串。

如果参数求值为 NULL 值，或为小于 1 的数值，则此函数返回 NULL 值，而不是空串。

在下列示例中，SPACE 函数返回单个字符的空字符串：

```
SELECT SPACE(1) FROM tabula_rasa;
```

下列表格展示来自此 SELECT 语句的输出，其为单个空字符：

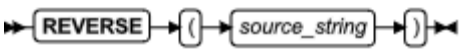
| (constant) |
|---|
| |

## *REVERSE 函数*

**REVERSE** 函数接受字符表达式作为它的参数，并返回同样长度的字符串，但颠倒每个逻辑字符的顺序位置。

这是 REVERSE 函数的语法：

REVERSE 函数



| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *source_string* | 求值为字符串的表达式 | 必须为表达式、常量、列或可转换为字符类型的类型的主变量 | 表达式 |

**REVERSE** 函数的参数不可有用户定义的数据类型。内建的 CHAR、LVARCHAR、NCHAR、NVARCHAR 和 VARCHAR 类型是有效的。

**REVERSE** 函数返回与它的 ***source_string*** 参数相同的数据类型。

如果您指定作为参数的表达式求值为 NULL，则返回值为 NULL。

对于求值为 N 字符的字符串的参数，在 ***source_string*** 中每一字符的顺序位置 p 在返回的字符串中成为 (N + 1 - p)。这颠倒了 ***source_string*** 中字符序列的原始顺序，因此，返回值以 ***source_string*** 中的最后一个字符开头，并以 ***source_string*** 的第一个字符结尾。

例如，函数表达式 REVERSE('Mood') 从加引号的字符串参数返回字符串 dooM。在单字节和多字节代码集中，仅颠倒顺序的位置，而不颠倒字母本身。在上述的函数表达式中，'d' 不成为 'b'，且多字节代码集（例如，**utf8** 或 **GB2312-80**）中的每一逻辑字符作为单一的逻辑单位换位。

如果参数求值为单个字符或空的 ***source_string***，则返回值与 ***source_string*** 相同，如同 **REVERSE** 函数未起作用一样。对于包括多个字符的字符串，仅当 ***source_string*** 是回文时，此等式才为真。对于 MOD(N,2) = 1 的字符串，顺序位置为 (N+1)/2 的字符在 ***source_string*** 中和在返回的字符串中都处于相同的中间位置。

在下列示例中，REVERSE 函数颠倒一个引号括起的字符串参数：

```
SELECT REVERSE('Able was I ere I saw Elba.') FROM Mirror_Table;
```

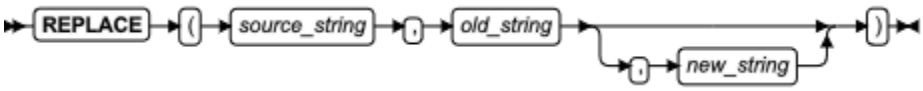
下列表格展示此 SELECT 语句的输出。

| (constant) |
|---|
| .ablE was I ere I saw elbA |

## *REPLACE 函数*

REPLACE 函数以不同的字符替换源字符串内指定的字符。

REPLACE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_string* | 替换字符串中 *old_string* 的单个 | 必须为表达式、常量、列或可转换为字符数据类型的数 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 或多个字符 | 据类型的主变量 | |
| *old_string* | 要被 *new_string* 替换的 *source_string* 中的一个或多个字符 | 必须为表达式、常量、列或可转换为字符数据类型的数据类型的主变量 | 表达式 |
| *source_string* | REPLACE 函数的字符串参数 | 必须为表达式、常量、列或可转换为字符数据类型的数据类型的主变量 | 表达式 |

**REPLACE** 函数的任何参数都必须为内建的数据类型。

**REPLACE** 函数返回 ***source_string*** 的副本，以 ***new_string*** 替换其中的每个 ***old_string***。如果您省略 ***new_string*** 选项，则从返回字符串中略去每个 ***old_string***。

返回数据类型是 ***source_string*** 参数的数据类型。如果返回值的长度超过 2048 字节，则会截断超长的字符，保存截断后的结果。

在下列示例中，REPLACE 函数以 t 替换源字符串中的每个 xz：

```
SELECT REPLACE('Mighxzy xzime', 'xz', 't') FROM mytable;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
| --- |
| Mighty time |

## *LPAD 函数*

LPAD 函数返回 *source_string* 的一个副本，左填充达到由 *length* 指定的总字节数。

LPAD 函数


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *length* | 指定在返回的字符串中总的字节数的整数值 | 必须为表达式、常量、列或可转换为整数数据类型的数据类型的主变量 | 精确数值 |
| *pad_string* | 指定一个或多个填充字符的字符串 | 必须为表达式、常量、列或可转换为字符数据类型的数据类型的主变量 | 表达式 |
| *source_string* | 作为 LPAD 函数 | 必须为表达式、常量、列或可转 | 表达 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 的输入的字符串 | 换为字符数据类型的数据类型的主变量 | 式 |

**LPAD** 函数的任何参数都必须为内建的数据类型。

***pad_string*** 参数指定要被用于填充源字符串的一个或多个字符。填充字符的序列出现的次数与使得返回字符串达到 ***length*** 指定的存储长度的必要次数相同。

如果 ***pad_string*** 中的填充字符的序列太长，以至于不适应 ***length***，则截断它。如果您未指定 ***pad_string***，则缺省值为单个空（ASCII 32）字符。

返回数据类型是基于这三个参数的，使用 来自 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

在下列示例中，用户指定要将源字符串左填充达到总长度 16 字节。用户还指定填充字符是由连字符和下划线（-_）组成的序列。

```
SELECT LPAD('Here we are', 16, '-_') FROM mytable;
```

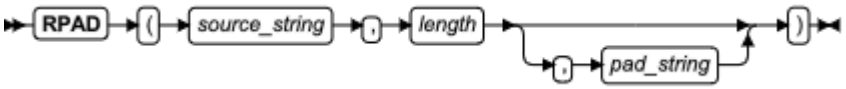
下列表格展示此 SELECT 的输出。

| (constant) |
| --- |
| -_-_-Here we are |

## *RPAD 函数*

RPAD 函数返回 *source_string* 的一个副本，右填充达到 *length* 参数指定的总字节数。

RPAD 函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *length* | 返回值中的总字节数 | 必须为表达式、常量、列或返回整数的主变量 | 精确数值 |
| *pad_string* | 指定一个或多个填充字符的字符串 | 必须为表达式、常量、列或可转换为字符数据类型的数据类型的主变量 | 表达式 |
| *source_string* | 作为 RPAD 函数的输入的字符串 | 同 *pad_string* | 表达式 |

**RPAD** 函数的任何参数都必须为内建的数据类型。

***pad_string*** 参数指定要用来填充源字符串的一个或多个填充字符。

填充字符的序列出现的次数与使得返回字符串达到 ***length*** 指定的长度所必要的次数相同。如果 ***pad_string*** 中的填充字符的序列太长，以至于不适应 ***length***，则截断它。如果您省略 ***pad_string*** 参数，则缺省的值为单个空格（ASCII 32）字符。

返回数据类型是基于 ***source_string*** 和 ***pad_string*** 参数的，如果都指定了的话。如果主变量是源，则返回值为 NVARCHAR 或 NCHAR，根据返回的字符串的长度来确定，使用 来自 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

即使 **RPAD** 函数已将空字符追加到数据值之后，DB-Access 的 UNLOAD 特性也截断 CHAR 或 NCHAR 列中的末尾空格。您必须显式地将 CHAR 或 NCHAR 值强制转型为 VARCHAR、LVARCHAR 或 NVARCHAR 数据类型，如果您需要 UNLOAD 保留 **RPAD** 返回的值中的末尾空字符或不可打印的字符的话。

在下列示例中，用户指定将源字符串右填充到总长度 18 字符。用户还指定要使用的填充字符是由问号和叹号（?!）组成的序列

```
SELECT RPAD('Where are you', 18, '?!')  FROM mytable;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
| --- |
| Where are you?!?!? |

## *CHR 函数*

此函数接受无符号整数参数，并返回单个逻辑字符。

**CHR** 函数有此语法：

CHR 函数


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *expression* | 求值为小于 256 的非负的整数的表达式 | 必须为取值范围从 0 至 255（含 0 和 255）的整数 | 表达式 |

返回值的数据类型为 VARCHAR(1)。

参数可为 SMALLINT、INTEGER、SERIAL、INT8、SERIAL8、BIGINT 或 BIGSERIAL。参数必须求值为整数，取值范围从 0 至 255。

如果参数为取值范围从 0 至 127 的整数，则返回值为对应的单字节 ASCII 代码点。要获取对应于 ASCII 代码点从 0 至 127 的字符的列表，请参阅 U.S. English 数据的排序顺序。

如果参数是取值范围从 128 至 255 的整数，则返回值为缺省的代码集中对应的 2 字节代码点。

- 在 UNIX™ 平台上，缺省的代码集为 ISO8859-1。

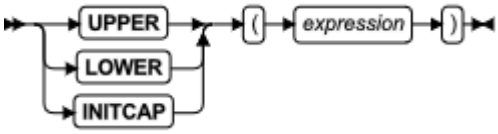
- 在 Windows™ 平台上，缺省的代码集为 Microsoft™ 1252。

## 大小写转换函数

大小写转换函数执行对字母字符的大小写转换。在缺省的语言环境中，这些函数仅可修改范围为 A - Z 以及 a - z 内的 ASCII 字符，这使得您能够在您的查询中执行区分大小写的搜索并能够指定输出的格式。

大小写转换函数是 UPPER、LOWER 和 INITCAP。下图展示这些大小写转换函数的语法。

大小写转换函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 返回字符串的表达式 | 必须为内建的字符类型。如果是主变量，则它的长度必须足够存储转换的字符串。 | 表达式 |

***expression*** 必须返回字符数据类型。当指定列表达式时，由数据库服务器返回的列数据类型为 ***expression*** 的数据类型。例如，如果输入类型为 CHAR，则输出类型也为 CHAR。

这些函数的参数必须为内建的数据类型。

在所有语言环境中，以大小写转换函数从列的描述返回的字节长度是源字符串的输入字节长度。如果您使用带有多字节 ***expression*** 参数的大小写转换函数，则转换可能增加或减少该字符串的长度。如果结果字符串的字节长度超过 ***expression*** 的字节长度，则数据库服务器截断结果字符串来适应 ***expression*** 的字节长度。

仅转换在语言环境文件中指定为 ALPHA 类的字符，且仅当语言环境识别大小写的结构时才会发生。

如果 ***expression*** 求值为 NULL，则大小写转换函数的结果也是 NULL。

在下列实例中，数据库服务器将大小写转换函数处理为 SPL 例程：

- 如果它没有参数
- 如果它有一个参数，且那个参数是命名的参数
- 如果它有多个参数
- 如果它出现在 Projection 列表中，以主变量作为参数

如果未遇到前面的列表中的情况，则数据库服务器将大小写转换函数处理为系统函数。

下列示例在相同的查询中使用所有大小写转换函数来为同一值指定多种输出格式：

Input value:
SAN Jose

Query:
SELECT City, LOWER(City), LOWER("City"),
UPPER (City), INITCAP(City)
FROM Weather;

Query output:
SAN Jose san jose city SAN JOSE San Jose

## *UPPER 函数*

**UPPER** 函数接受一个***表达式***参数，并返回其中的***表达式***中的每个小写字母字符都被对应的大写字母字符替换的字符串。

下列示例使用 UPPER 函数来对带有 Curran 姓氏的所有员工执行 lname 列上的区分大小写搜索：

SELECT title, INITCAP(fname), INITCAP(lname) FROM employees
WHERE UPPER (lname) = "CURRAN"

由于在 projection 列表中指定 **INITCAP** 函数，因此数据库服务器返回混合大小写格式的结果。例如，一个相匹配的行的输出可能为：accountant James Curran.

## *LOWER 函数*

**LOWER** 函数接受一个***表达式***参数，并返回其中***表达式***中的每个大写字母字符都被对应的小写字母字符替换的字符串。

下列示例展示如何使用 LOWER 函数来在 City 列上执行区分大小写的搜索。此语句指导数据库服务器以混合大小写格式 San Jose 替换词语 san jose 的任何实例（即，任何变化形式）。

UPDATE Weather SET City = "San Jose"
WHERE LOWER (City) = "san jose";

## *INITCAP 函数*

**INITCAP** 函数返回***表达式***的一个副本，其中***表达式***中每个词都以大写字母开头。使用这个函数，***词语***在任何字符而不是字母之后开始。因此，除了空格，诸如逗号、句号、冒号等等这样的符号引出新的词语。

要获取 **INITCAP** 函数的示例，请参阅 UPPER 函数。

## *NLSCASE INSENSITIVE 数据库中的大小写转换函数*

指定了 **UPPER** 和 **LOWER** 大小写转换函数来支持在区分大小写的数据库中的区分大小写查询。在有 NLSCASE INSENSITIVE 属性的数据库中不经常需要他们，因为无需调用这些函数，NCHAR 和 NVARCHAR 数据类型就支持区分大小写的查询。您可在 NLSCASE INSENSITIVE 数据库中调用大小写转换函数，它们对 CHAR、LVARCHAR 和 VARCHAR 数据类型的影响与在区分大小写的数据库中相同。

在以 NLSCASE INSENSITIVE 选项创建的数据库中，数据库服务器不理会 NCHAR 和 NVARCHAR 值的字母大小写。调用大小写转换函数的表达式可返回的结果可与区分大小写的数据

库会返回的不同，如果该表达式引用 NCHAR 或 NVARCHAR 对象，或如果数据库服务器以显式的或隐式的强制转型将该表达式求值为 NCHAR 或 NVARCHAR 数据类型的话。

当在带有 NLSCASE INSENSITIVE 属性的数据库中使用 **UPPER、LOWER 或 INITCAP** 函数对字符串表达式求值时，数据库服务器调用该函数，并对它的返回值应用数据类型提升规则，在主题 来自 CONCAT 和字符串函数的返回类型 中总结该规则。

- 如果该表达式求值为 CHAR、LVARCHAR 或 VARCHAR 数据类型，则数据库服务器可使用区分大小写的操作中的结果，如果那些操作不涉及 NCHAR 或 NVARCHAR 对象的话。
- 在执行了 **UPPER、LOWER** 或 **INITCAP** 函数之后，如果该表达式求值为 NCHAR 或 NVARCHAR 值，则在使用来自该表达式的此返回值的后续操作中，不理会在此结果中的字母的大小写。

下列示例说明在 NLSCASE INSENSITIVE 数据库中的此行为，其中表 t1 有这五个内建的字符数据类型的字符列。该表存储三行，其中每一列存储 3 个字母字符串的相同大小写的形式：

```
CREATE DATABASE db NLSCASE INSENSITIVE;
        CREATE TABLE t1 (
        c1 NCHAR(20),
        c2 NVARCHAR(20),
        c4 VARCHAR(20),
        c5 LVARCHAR(20)) ;
INSERT INTO t1 values ('gbase', 'gbase', 'gbase', 'gbase', 'gbase');
INSERT INTO t1 values ('Gbase', 'Gbase', 'Gbase', 'Gbase', 'Gbase');
INSERT INTO t1 values ('GBASE', 'GBASE', 'GBASE', 'GBASE', 'GBASE');
```

在下列示例中，数据库服务器将 **UPPER** 函数应用到 NCHAR 列 **c1**，然后对于 'GBASE' 字符串常量相匹配的列中返回的所有值应用区分大小写的规则。

```
SELECT c1 FROM t1 WHERE UPPER(c1) = ‘GBASE’;
```

由于在此数据库中 NCHAR 值时不区分大小写的，因此该查询从表中的每行返回列 c1 值，由于在每一行中字母的序列与字符串常量相匹配，使用忽略该列值的字母大小写的不区分大小写的规则：

```
c1
gbase
Gbase
GBASE
```

在相同的表上，通过对同一查询的下列修改，也会返回相同的结果集（即’gbase’、’Gbase’ 和 ’ GBASE ’）：

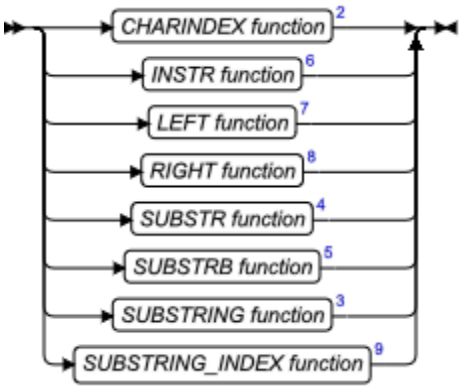
- 如果 projection 子句指定了任何其他列，而不是 **c1**，因为每行存储相同的值，且对于此数据库中字符串 'GBASE' 的所有大小写变化形式，**UPPER** 返回的 NCHAR 值使得 WHERE 子句为真。
- 如果 WHERE 子句中的 'GBASE' 字符串是相同的字符序列的任何其他字母大小写变化形式，因为在此数据库中区分大小写的规则不处理 NCHAR 数据类型。
- 如果大小写转换函数的参数是 NVARCHAR 列 **c2**，而不是 NCHAR 列 **c1**，因为在此数据库中 NCHAR 或 NVARCHAR 都是不区分大小写的数据类型。

- 如果将大小写转换函数 **LOWER** 或 **INITCAP**，而不是 **UPPER**，应用于列 **c1**，因为在此数据库中，那个 NCHAR 列的每个（大小写变化形式）值都与 'GBASE' 相匹配。
- 如果未调用大小写转换函数，但 WHERE 条件反而指定了 c1 = 'GBASE'，因为在此 NLSCASE INSENSITIVE 数据库中，大小写转换函数作为查询过滤器对 NCHAR 或 NVARCHAR 参数不起作用。

## 子字符串函数

内建的 SQL 子字符串函数从字符串参数返回子字符串，或返回子字符串上操作的位置信息。

子字符串函数




下面的部分描述这些子字符串函数的语法和用法。

### *CHARINDEX 函数*

**CHARINDEX** 函数搜索字符串，找到目标子字符串的第一次出现，搜索从源字符串内指定的或缺省的字符位置开始。

CHARINDEX 函数有此语法：

CHARINDEX 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *source_string* | 求值为字符串的表达式 | 必须为一表达式、常量、列或内建的字符类型或可转换为字符类型的类型的主变量 | 表达式 |
| *start_position* | 要在 *source* 中开始搜索的顺序位置，此处的 1 为第一个逻辑字符 | 必须为一表达式、常量、列或内建的字符类型或可转换为字符类型的类型的主变量 | 表达式 |
| *substring* | 求值为字符串的表达式 | 必须为一表达式、常量、列或内建的字符类型或可转换为字 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 符类型的类型的主变量 | |

**CHARINDEX** 的参数不可为用户定义的数据类型。

如果 ***source*** 或 ***substring*** 为 NULL，则此函数返回 NULL。

如果可选的 ***start_position*** 值小于 1，或如果您省略此参数，则在 ***source*** 中的第一个逻辑字符处开始搜索 ***substring***，如同您已指定了 1 作为起始位置一样。

如果找不到与 ***substring*** 相匹配的表达式，则 **CHARINDEX** 返回零（0）。否则，它返回在 ***substring*** 第一次出现的第一个逻辑字符的顺序位置。

如果您指定大于 1 的 *start_position*，则忽略在 *start_position* 之前开始的任何 *substring*，该函数返回下列值之一：

- 或者是在第一个相匹配的子字符串中第一个逻辑字符的位置，其顺序位置等于或大于 ***start_position***，
- 或者是零（0），如果开始于 ***start_position*** 或跟在 ***start_position*** 之后的 ***source*** 中没有 ***substring*** 出现，或如果 ***start_position*** 大于 ***source*** 中逻辑字符的数目。

在支持多字节字符集的语言环境中，返回值为 ***source*** 中逻辑字符之中的顺序值。在单字节语言环境中，比如缺省的语言环境，返回值等同于字节位置，在此，第一个字节位于位置 1。

在以 NLSCASE INSENSITIVE 选项创建的数据库中，如果 ***source*** 或 ***substring*** 是 NCHAR 或 NVARCHAR 数据类型，则数据库服务器在确定 ***source*** 的给定的子字符串是否与目标 ***substring*** 相匹配时，忽略字母大小写的变化形式。

下列函数表达式返回 9：

```
CHARINDEX('com','www.gbase.com')
```

在上面的示例中，**CHARINDEX** 在缺省的起始位置 1 开始它的搜索。

下列函数表达式返回 2：

```
CHARINDEX('w','www.gbase.com',2)
```

在上面的示例中，由于最后的参数在位置 2 开始搜索，因此，**CHARINDEX** 忽略两个其他的相匹配的子字符串：

- 位置 1 中的 'w'，因为搜索开始于 2，
- 以及位置 3 的 'w'，因为该函数仅返回相匹配的子字符串第一次出现的位置。

## *INSTR 函数*

**INSTR** 函数从字符串搜索指定的子字符串，并基于子字符串出现的次数返回在那个字符串中子字符串终止出现处的字符位置。

**INSTR** 函数有此语法：

INSTR 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *count* | 求值为 > 0 整数的表达式 | 必须为表达式、常量、列或内建的整数类型或可转换为整数的主变量。 | 表达式 |
| *source_string* | 求值为字符串的表达式 | 必须为表达式、常量、列或内建的字符数据类型或可转换为字符类型的主变量 | 表达式 |
| *start* | 在 *source_string* 中开始搜索的顺序位置，在此，1 是第一个逻辑字符 | 必须为表达式、常量、列或内建的整数类型或可转换为正的或负的整数的主变量 | 表达式 |
| *substring* | 求值为字符串的表达式 | 必须为表达式、常量、列或内建的字符数据类型或可转换为字符类型的主变量 | 表达式 |

**INSTR** 的参数不可为用户定义的数据类型。

在这些情况下，该函数返回 NULL：

- ***count*** 小于或等于零（0）。
- ***source_string*** 为 NULL 或长度为零。
- ***substring*** 为 NULL 或长度为零。

在下列情况下，返回值为零（0）：

- 如果在 ***source_string*** 发现没有出现 ***substring***，
- 如果 ***start*** 大于 ***source_string*** 的长度。
- 如果在 ***source_string*** 中 ***substring*** 的出现少于 ***count***，

如果您省略可选的 ***count*** 参数，则缺省的 ***count*** 值为 1。

在支持多字节字符集的语言环境中，返回值为 ***source_string*** 中逻辑字符之中的顺序值。在单字节语言环境中，比如缺省的语言环境，返回值等同于字节位置，在此，第一个字节在位置 1 中。

**start 位置**

如果省略 *start* 或指定 *start* 为零，则对 *substring* 的搜索从字符位置 1 开始。如果 *start* 为负的，则在 *source_string* 结束的位置开始搜索 *substring* 的出现，并向着开头方向处理。

- 在从左至右的语言环境中，负的 ***start*** 值指定从右至左的搜索。
- 在从右至左的语言环境中，负的 ***start*** 值指定从左至右的搜索。

然而，在这两种类型的语言环境中，都是在 source_string 之内对应于 start 的绝对值的逻辑字符位置上开始搜索。

在从右至左的语言环境中，负的 ***start*** 值指定从左至右的搜索。

**INSTR 函数表达式的示例**

下列表达式都是基于相同的 *source_string* 和 *substring* 的。此示例返回 3，作为第一个 'er' 子字符串的字符位置：

```
INSTR("wwerw.gbase.cerom", "er")
```

在上面的示例中，***start*** 和 ***count*** 都缺省为 1。

下一个示例在第二个字符位置开始搜索，带有缺省的 *count* 1：

```
INSTR("wwerw.gbase.cerom", "er", 2)
```

上面的表达式返回 3，从左至右的搜索在第一个 'er' 子字符串中遇到的第一个字符的位置。

下一个示例指定 count 2，在 *source_string* 的第一个字符中开始搜索：

```
INSTR("wwerw.gbase.cerom", "er", 1, 2)
```

上面的表达式返回 12，第二个 'er' 开始的位置。

下列示例指定 -5 作为开始的位置，且 *count* 指定在 *source_string* 的第 5 个位置与开头之间 "er" 的第一次出现：

```
INSTR("wwerw.gbase.cerom", "er", -5, 1)
```

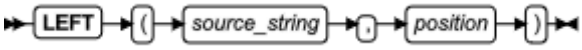
这返回 3，对应于从那个位置开始 "er" 子字符串的第一次出现。负的 *start* 参数指定从右至左的查询，但返回值为 3，因为在缺省的语言环境中，字符串和子字符串的读方向为从左至右的。

## *LEFT 函数*

**LEFT** 函数从字符串参数返回由最左边 ***N*** 个字符组成的子字符串。

该函数有此语法：

LEFT 函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *position* | 在字符串中的（从左边开始的）顺序位置；要返回此字符及左边的所有字符 | 必须为表达式、常量、列或内建的整数类型或可转换为整数的主变量 | 表达式 |
| *source_string* | 求值为字符串的表达式 | 必须为表达式、常量、列或可转换为字符类型的数据类 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 型的主变量 | |

**LEFT** 函数的参数不可为用户定义的数据类型。

在从左至右的语言环境中，比如缺省的 U.S. English 语言环境，此函数从 ***source_string*** 返回开头字符的子字符串。

**LEFT** 函数返回的内容依赖于 ***source_string*** 中逻辑字符的数目以及 ***position*** 的值：

- 如果 ***source_string*** 求值为带有多于 ***position*** 个字符的字符串，则返回值为 ***source_string*** 的子字符串，由指定的 ***position*** 左边的所有字符组成。
- 如果 ***source_string*** 求值为带有不多于 ***position*** 个字符的字符串，则返回值为整个 ***source_string***。
- 如果 ***source_string*** 求值为 NULL，或如果 ***position*** 为零或负的，则返回 NULL。
- 如果未指定 ***position*** 参数，则不返回字符串值，并发出例外。

返回数据类型与它的 ***source_string*** 参数相同。如果主变量是源，则返回值为 NVARCHAR 或 NCHAR，根据返回的字符串的长度来定，使用 来自 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

下列函数表达式请求引号括起的字符串的前五个字符：

```
LEFT('www.gbase.cn',5)
```

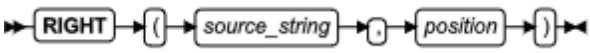
在此示例中，LEFT 函数返回子字符串 www.g

## *RIGHT 函数*

**RIGHT** 函数从字符串参数返回有最右边的 ***N*** 个字符组成的子字符串。

该函数有此语法：

RIGHT 函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *position* | 字符串中的（从右边开始的）顺序位置；返回此字符及右边的所有字符 | 必须为表达式、常量、列或内建的整数类型或可转换为整数的主变量 | 表达式 |
| *source_string* | 求值为字符串的表达式 | 必须为表达式、常量、列或可转换为数值类型的数据类型的主变量 | 表达式 |

**RIGHT** 函数的参数不可为用户定义的数据类型。

在从左至右的语言环境中，比如缺省的 U.S. English 语言环境，此函数从 ***source_string*** 返回末尾的字符的子字符串。

**RIGHT** 函数返回的内容依赖于 ***source_string*** 中逻辑字符的数目以及 ***position*** 的值：

- 如果 ***source_string*** 求值为带有多于 ***position*** 个字符的字符串，则返回值为 ***source_string*** 的子字符串，由指定的 ***position*** 右边的所有字符组成。
- 如果 ***source_string*** 求值为带有不多于 ***position*** 个字符的字符串，则返回值为整个 ***source_string***。
- 如果 ***source_string*** 求值为 NULL，或如果 ***position*** 为零或负的，则返回 NULL。
- 如果未指定 ***position*** 参数，则不返回字符串值，并发出例外。

返回数据类型与它的 ***source_string*** 参数相同。如果主变量是源，则返回值为 NVARCHAR 或 NCHAR，根据返回的字符串的长度来定，使用 来自 CONCAT 的返回类型 部分的返回类型提升规则。

下列函数表达式请求由引号括起的字符串的最后五个字符：

```
RIGHT('www.gbase.cn',5)
```

在此示例中，RIGHT 函数返回子字符串 se.cn

## *SUBSTR 函数*

SUBSTR 函数与 SUBSTRING 函数有相同的目的（返回源字符串的子集），但它使用不同的语法。

SUBSTR 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *length* | 要从 source_string 返回的字符的数目 | 必须为表达式、文字、列或返回整数的主变量 | 表达式 |
| *source_string* | 作为 SUBSTR 函数的输入的字符串 | 必须为表达式、文字、列或可转换为字符数据类型的数据类型的主变量 | 表达式 |
| *start_position* | *source_string* 中的列位置，SUBSTR 函数从此位置开始返回字符 | 必须为整数表达式、文字、列或主变量。可有正号（+）、负号（-）或无符号。 | 精确数值 |

**SUBSTR** 函数的任何参数都必须为内建的数据类型。

SUBSTR 函数返回 *source_string* 的子集。该子集从 *start_position* 指定的列位置开始。下列表格展示数据库服务器如何基于 *start_position.* 的输入值来确定返回的子集的起始位置

| Start_Position 的值 | 数据库服务器如何确定返回的子集的起始位置 |
|---|---|
| 正的 | 从 *source_string* 中的第一个字符开始向前计数 |
| 零（0） | 从 *source_string* 中的第一个字符向前计数（也就是说，将 *start_position* 0 处理为等同于 1） |
| 负的 | 从紧跟在 *source_string* 中最后一个字符的原始字符向后计数。值 -1 返回 *source_string* 中的最后一个字符。 |

***length*** 参数指定子集中逻辑字符的数目（不是字节数）。如果您省略 ***length*** 参数，则 **SUBSTR** 函数返回从 ***start_position*** 处开始的 ***source_string*** 的整个部分。

如果您指定负的 ***start_position***，其绝对值大于 ***source_string*** 中字符的数目，或如果 ***length*** 大于从 ***start_position*** 至 ***source_string*** 的末尾的字符的数目，则 **SUBSTR** 返回 NULL。（在此情况下，**SUBSTR** 的行为不同于 **SUBSTRING** 函数的行为，返回从 ***start_position*** 至 ***source_string*** 的最后一个字符的所有字符，而不是返回 NULL。）

返回数据类型是 ***source_string*** 参数的类型。如果主变量是源，则返回值为 NVARCHAR 或 NCHAR，根据返回的字符串的长度来定，使用 来自 CONCAT 的返回类型 部分描述的返回类型提升规则。

下列示例指定要返回的字符串从 7 个字符的 *source_string* 结束之前的开始位置 3 个字符开始。这意味着开始的位置是 *source_string* 的第五个字符。因为用户未指定 *length* 的值，数据库服务器返回包括从字符位置 5 至 *source_string* 的结尾的所有字符。

```
SELECT SUBSTR('ABCDEFG', -3) FROM mytable;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
|---|
| EFG |

## *SUBSTRB 函数*

返回字符串的子字符串，在字符串中指定的位置开始。

SUBSTRB 函数


| 元素 | 描述 |
|---|---|
| *length* | 指定以字节计的结果的长度的表达式。如果指定，则表达式必须返回内建的数值、CHAR 或 VARCHAR 数据类型的值。如果该值不是 INTEGER 类型，在对该函数求值之前，隐式地将它强制转型为 INTEGER。 |

| 元素 | 描述 |
| --- | --- |
| | 如果 ***length*** 的值大于从字符串的起始位置到结束的字节数，则结果的长度等于第一个参数的长度减去起始位置，加一。<br>如果 ***length*** 的值小于或等于零，则 SUBSTRB 的结果为 NULL 字符串。<br>***length*** 的缺省值是从由 ***starting_position*** 指定的位置到字符串最后的字节的字节数。<br>当指定 *length* 时，将结果字符串的长度截断为 *length* 的值。在下列示例中，*my_string* 是 10 字节的字符串，限定结果字符串为 5 字节：<br>substrB(***my_string***, 3, 5)<br>在前面的示例中，如果 *my_string* 是 4 字节的字符串，且起始位置为第三个字节，则返回 2 字节的字符串。<br>如果未指定 *length*，则结果的长度为从 *starting_position* 开始的 *source_string* 的长度。在下列示例中，*my_string* 是 10 字节的字符串，返回 8 字节的字符串：<br>substrB(***my_string***, 3) |
| *source_string* | 指定从其派生结果的字符串的表达式。该表达式必须返回内建的字符串、数值或 datetime 数据类型的值。如果该值不是字符串数据类型，则在对该函数求值之前，隐式地将它强制转型为 NVARCHAR。对于零长度结果，返回 NULL 值。 |
| *starting_position* | 指定结果子字符串的开头的字符串中起始位置的表达式。该表达式必须返回内建的数值、CHAR 或 VARCHAR 数据类型的值。如果该值不是 INTEGER 类型，则在对该函数求值之前，隐式地将它强制转型为 INTEGER。<br>如果 ***starting_position*** 是正的，则从该字符串的开头计算起始位置。如果 ***starting_position*** 大于字符串的 ***length***，则返回空字符串。如果 ***starting_position*** 是负的，则从该字符串的末尾计算起始位置，且向后对字节数计数。如果 ***starting_position*** 的绝对值大于 ***source_string*** 的 ***length***，则返回空字符串。如果 ***starting_position*** 为 0，则使用起始位置 1。<br>注：所有的 *length* 和 *starting_position* 的单位都以字节表示，即使对于在多字节代码集中编码的字符串。SUBSTR 使用多字节字符串的逻辑字符大小。例如，如果在传统的 SBSTR 中 *starting_position* 为 2，且 |

| 元素 | 描述 |
| --- | --- |
| | 多字节字符串的第一个字符需要 3 字节的存储，则 2 表示字符串中的第四个字节。在 SUBSTRB 中，2 表示字符串中的第二个字节。 |

如果 ***source_string*** 是 CHAR 或 VARCHAR 数据类型，则该函数的结果是 VARCHAR 数据类型。GBase 8s 不支持多代码页；相反， GBase 8s JDBC 或 ODBC 将代码页翻译到数据库。

如果任何参数为空，则结果为空值。

在动态 SQL 中，可由主变量来表示 ***source_string***、***starting_position*** 和 ***length***。如果主变量用作 ***source_string***，则运算对象的数据类型为 VARCHAR，且运算对象可为空。

在上述结果定义中虽未显式地说明，但语义表明，如果 ***source_string*** 是多字节字符串，则结果可能包含多字节字符的片段，这依赖于 starting_position 和 length 的值。例如，结果可能起始于多字节字符的第二个字节，或终止于多字节字符的第一个字节。SUBSTRB 函数检测这些部分的字符并以单个空格字符代替不完整字符的每一字节。SUBSTRB 返回固定的字节数；使用 SUBSTR，返回的数目根据多字节字符串而不同。

## *SUBSTRING 函数*

SUBSTRING 函数返回字符串的子集。

SUBSTRING 函数


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *length* | 要从 *source_string* 返回的字符数 | 必须为返回整数的表达式、常量、列或主变量 | 精确数值 |
| *source_string* | SUBSTRING 函数的字符串参数 | 必须为其值可转换为字符数据类型的表达式、常量、列或主变量 | 表达式 |
| *start_position* | 在 *source_string* 中返回首个字符的位置 | 必须为返回整数的表达式、常量、列或主变量 | 精确数值 |

**SUBSTRING** 函数的任何参数都必须为内建的数据类型。

返回数据类型是 ***source_string*** 参数的数据类型。如果主变量是源，则返回值为 NVARCHAR 或 NCHAR，根据返回的字符串的长度来定，使用 来自 CONCAT 函数的返回类型 部分描述的返回类型提升规则。

该子集开始于 *start_position* 指定的列位置。下列表格展示数据库服务器如何基于 *start_position* 的输入值来确定返回的子集的起始位置。

| Start_Position 的值 | **数据库服务器如何确定返回子集的起始位置** |
|---|---|
| 正的 | 从 ***source_string*** 中的第一个字符向前计数<br>例如，如果 ***start_position*** = 1，则 ***source_string*** 中的第一个字符是返回的子集中的第一个字符。 |
| 零（0） | 从 ***source_string***<br>中的第一个字符之前（即，左边的第一个字符）的一个位置计数<br>例如，如果 ***start_position*** = 0 且 ***length*** = 1，则数据库服务器返回 NULL，反之如果 ***length*** = 2，数据库服务器返回 ***source_string*** 中的第一个字符。 |
| 负的 | 从 ***source_string*** 中的最后一个字符之后（即，右边的第一个字符）的一个位置向后计数<br>例如，如果 ***start_position*** = -1，则返回的子集的起始位置是 ***source_string*** 中的最后一个字符。 |

由 *length* 指定子集的大小。*length* 参数指的是逻辑字符的数目，而不是字节数。如果您省略 *length* 参数，或如果您指定的 *length* 大于从 *start_position* 至 *source_string* 的末尾的字符数，则 SUBSTRING 函数返回开始于 *start_position* 的 *source_string* 的整个部分。下列示例指定起始于列位置 3 的源字符串的子集，且应返回两个字符长。：

```
SELECT SUBSTRING('ABCDEFG' FROM 3 FOR 2) FROM mytable;
```

下列表格展示此 SELECT 语句的输出。

| (constant) |
|---|
| CD |

在下列示例中，用户为返回子集指定负的 *start_position*：

```
SELECT SUBSTRING('ABCDEFG' FROM -3 FOR 7) FROM mytable;
```

数据库服务器在位置 -3 开始（第一个字符之前的四个位置）并向前计数 7 字符。下列表格展示此 SELECT 语句的输出。

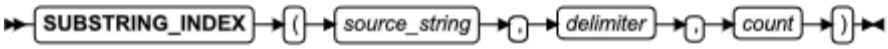
| (constant) |
|---|
| ABC |

## *SUBSTRING_INDEX 函数*

**SUBSTRING_INDEX** 函数搜索指定的定界符字符的字符串，并基于您指定作为该函数的参数的定界符的计数返回开头或收尾字符的子字符串。

SUBSTRING_INDEX 函数有此语法：

SUBSTRING_INDEX 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *source_string* | 求值为字符串的表达式 | 必须为内建的字符数据类型，或可转换为字符类型的表达式、常量、列或主变量 | 表达式 |
| *count* | 求值为正整数或负整数的表达式 | 必须为内建的整数类型，或可转换为整数的表达式、常量、列或主变量。 | 表达式 |
| *delimiter* | 求值为字符串的表达式 | 必须为内建的字符数据类型，或可转换为字符类型的表达式、常量、列或主变量 | 表达式 |

**SUBSTRING_INDEX** 的参数不可为用户定义的数据类型。

在下列的每一情况下，此函数返回 NULL：

- ***source_string*** 为 NULL
- ***delimiter*** 为 NULL
- ***count*** = 零（0）。

如果该搜索在 ***source_string*** 中找到少于 ***count*** 个定界符，则返回值为整个 ***source_string***。

返回值与 ***source_string*** 的数据类型相同。

对于 *source_string*，*count* 的符号决定返回的值是 *source_string* 中开头字符的子字符串还是收尾字符的子字符串：

- 对于 N = count，返回的子字符串的最后的字符紧接在开头的字符的子字符串中那个定界符第 N 次出现的前面。

  例如函数表达式

  SUBSTRING_INDEX("www.gbase.cn", ".", 2)

  返回开头字符 www.gbase，因为 *count* > 0。

- 对于 N = count < 0，返回的子字符串中的第一个字符紧接在收尾字符的子字符串中那个定界符第 N 次出现之前。

  例如，函数表达式

SUBSTRING_INDEX("www.gbase.cn", ".", -2)

返回收尾字符 gbase.cn，因为 *count* < 0。

上述示例适用于诸如缺省的 U.S. English 语言环境这样的从左至右的语言环境，其中 ***count*** 的负值导致此函数从 ***source_string*** 返回收尾字符的子字符串，而 ***count*** 的正值导致此函数从 ***source_string*** 返回开头字符的子字符串。

在支持多字节字符集的语言环境中，返回值是 ***source_string*** 中的逻辑字符之中的顺序值。在诸如缺省的语言环境这样的语言环境中，返回值等同于字节位置，在此，第一个字节在位置 1。

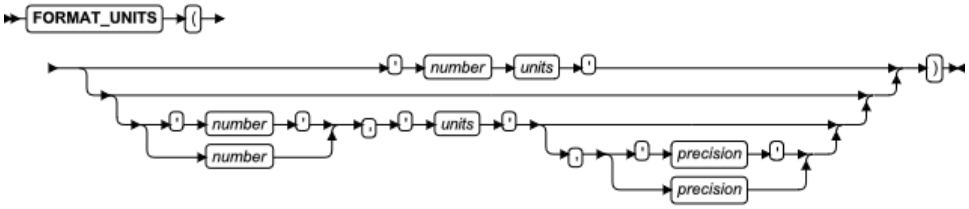
## FORMAT_UNITS 函数

**FORMAT_UNITS** 函数可解释指定内存或大量存储的单位的数目和缩写名称的字符串。

此内建的函数可接受一个、两个或三个由引号括起的字符串参数。您可调用通过内存或大量存储的字节或更大的单位（比如，kilobyte、megabyte、gigabyte 等等）的标准缩写来表示处理大小规范的 SQL 语句中的 **FORMAT_UNITS**。

还可通过 SQL 管理 API **ADMIN** 和 **TASK** 函数来在 **sysadmin** 数据库中内部调用 **FORMAT_UNITS** 函数，在 *GBase 8s 管理员参考* 中有对其的描述。

FORMAT_UNITS 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *number* | 求值为存储或内存 *units* 的数目的表达式 | 必须为文字数值或指定可转换为 FLOAT 的数目的由引号括起的字符串 | 表达式 |
| *precision* | 要从 *number* 返回的有效数值的整数数目 | 必须为文字数值或指定整数的引用字符串 | 表达式 |
| *units* | 存储或内存的单位的缩写；缺省值为 'B'（表示字节） | 必须以 'B'、'K'、'M'、'G'、'T'、'P' 或 'PB'（或这些字母的小写形式）开头。忽略任何收尾字符。 | 引用字符串 |

此内建的函数可接受一个、两个或三个参数。返回的值是展示指定的 ***number*** 和展示存储单位的适当的格式标签的字符串。如果您指定 ***precision*** 作为最后的参数，则以那个精度返回 ***number***。否则，在缺省情况下，将 ***number*** 格式化为精度 3（%3.3lf）。

同样的表示法也适用于所有 SQL 管理 API **ADMIN** 和 **TASK** 命令的参数（效仿 Enterprise Replication **cdr** 实用程序的命令除外），这些参数指定内存、存盘存储或地址偏移量的大小：

| 表示法 | 对应的单位 |
|---|---|
| 'B' 或 'b' | 字节（= 以 2 为底，指数为 0） |
| 'K' 或 'k' | 千字节（= 以 2 为底，指数为 10） |
| 'M' 或 'm' | Megabyte（= 以 2 为底，指数为 20） |
| 'G' 或 'g' | Gigabyte（= 以 2 为底，指数为 30） |
| 'T' 或 't' | Terabyte（= 以 2 为底，指数为 40） |
| 'PB' | Petabyte（= 以 2 为底，指数为 50） |
| 'P' | 页（= 2 千字节或 4 千字节，依赖于系统的基础页大小） |

*unit* 规范中的首字母（'B'、'K'、'M'、'G' 或 'T'）确定计量的单位，并忽略任何收尾的字符。然而，有一个例外，那就是在字符串中首字母 'P'（或 'p'）紧跟着 'B' 或 'b'，因为在此情况下，将该字符解释为 petabyte。将任何其他以 "P"（比如 "PA"、"pc"、"PhD"、"papyrus"，等等）起始的字符串解释为指定 *pages*，而不是指定 *petabytes*。

如果一个参数同时提供 ***number*** 和 ***units*** 规范，则 GBase 8s 忽略将 ***number*** 规范从 **FORMAT_UNITS** 或 SQL 管理 API **ADMIN** 或 **TASK** 函数的同一参数之内的 ***units*** 规范分隔开的任何空格。例如，将规范 '128M' 与 '128 m' 都解释为 128 megabyte。

下列示例以单个参数调用 FORMAT_UNITS 函数：

```
EXECUTE FUNCTION FORMAT_UNITS('1024 M');
```

返回下列字符串值。

| (expression) |
|---|
| 1.00 GB |

```
SELECT FORMAT_UNITS('1024 k') FROM systables WHERE tabid=1;
```

返回下列字符串值。

| (expression) |
|---|
| 1.00 MB |

```
SELECT FORMAT_UNITS(tabid || 'M') FROM systables WHERE tabid=100;
```

返回下列字符串值。

| (expression) |
|---|
| 100 MB |

下列示例展示以两个参数调用 FORMAT_UNITS 函数：

```
EXECUTE FUNCTION FORMAT_UNITS(1024, 'k');
```

返回下列字符串值。

| (expression) |
| --- |
| 1.00 MB |

```
SELECT FORMAT_UNITS( SUM(chksize), 'P') SIZE,
FORMAT_UNITS( SUM(nfree), 'p') FREE FROM syschunks;
size    117 MB
free    8.05 MB
```

此查询返回字符串值 size 117 MB 和 free 8.05 MB。

下列示例以三个参数调用 FORMAT_UNITS 函数：

```
EXECUTE FUNCTION FORMAT_UNITS(1024, 'k', 4);
```

返回下列字符串值。

| (expression) |
| --- |
| 1.000 MB |

```
SELECT FORMAT_UNITS( SUM(chksize), 'P', 4), SIZE,
FORMAT_UNITS( SUM(nfree), 'p', 4) FREE FROM syschunks;
size    117.2 MB
free    8.049 MB
```

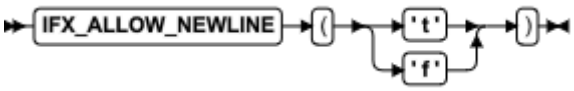
此查询返回字符串值 size 117.2 MB 和 free 8.047 MB。这些结果与前面仅以它们的非缺省的精度进行查询的示例不同，由 FORMAT_UNITS 的最后一个参数指定精度。

## IFX_ALLOW_NEWLINE 函数

**IFX_ALLOW_NEWLINE** 函数设置换行模式，在当前的会话内在用引号括起来的字符串中允许还是不允许换行字符。

IFX_ALLOW_NEWLINE 函数有下列语法。

IFX_ALLOW_NEWLINE 函数




如果您输入 't' 作为此函数的参数，则在该会话中启用用引号括起来的字符串中的换行字符。如果您输入 'f' 作为参数，则在该会话中不允许用引号括起来的字符串中的换行字符。

您可通过将 ONCONFIG 文件中的 ALLOW_NEWLINE 参数设置为值 0（不允许换行字符）或值 1（允许换行字符）来为所有会话设置换行模式。如果您未设置此配置参数，则缺省的值为 0。您每一次启动会话时，新的会话继承 ONCONFIG 文件中的换行模式设置。要更改会话的换行模式，请执行 **IFX_ALLOW_NEWLINE** 函数。一旦您已为会话设置了换行模式，该模式保持有效，直到会话结束为止，或直到您在会话内在此执行 **IFX_ALLOW_NEWLINE** 函数为止。

在下列示例中，假设您未为 ONCONFIG 文件中的 ALLOW_NEWLINE 指定任何值，因此，在缺省情况下，在任何会话中的用引号括起来的字符串中不允许换行字符。在您启动新的会话之后，您可通过执行 IFX_ALLOW_NEWLINE 函数来启用那个会话中的用引号括起来的字符串中的换行字符：

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('t');
```

在 ESQL/C 中，通过 ONCONFIG 文件中 的 ALLOW_NEWLINE 参数设置换行模式，或通过在会话中执行 **IFX_ALLOW_NEWLINE** 函数设置换行模式，换行模式仅适用于 SQL 语句中的用引号括起来的字符串文字。换行模式不适用于 SQL 语句中包含在主变量中的用引号括起来的字符串。主变量可在字符串数据内包含换行字符，不理会当前生效的换行模式。

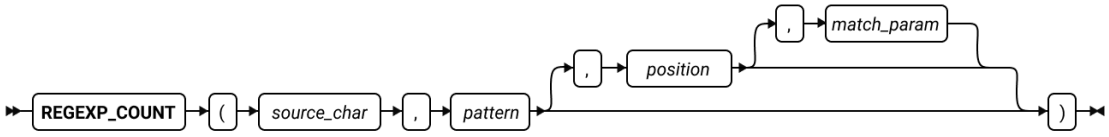
例如，您可使用主变量来将包含换行字符的数据插入到列内，即使 ONCONFIG 文件中的 ALLOW_NEWLINE 参数设置为 0。

要获取关于 **IFX_ALLOW_NEWLINE** 函数如何影响用引号括起来的字符串的更多信息，请参阅引用字符串。要获取关于 ONCONFIG 文件中的 ALLOW_NEWLINE 参数的更多信息，请参阅 *GBase 8s 管理员参考手册*。

## 正则表达式函数

### *REGEXP_COUNT*

匹配字符串在源串中出现的次数。返回 pattern 在 source_char 串中出现的次数。如果未找到匹配，则函数返回 0。




```
REGEXP_COUNT ( source_char, pattern [, position [, match_param]])
```

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***source_char*** | 源字符串 | 可以是字符串或列名。当源字符串 source_char 是列名时，支持的类型为 char 和 varchar 型。 | 表达式 |
| ***pattern*** | 正则表达式 | 支持的数据类型为 char 和 varchar 型。每个正则表达式最多可包含 512 个字节。 | 表达式 |
| ***position*** | 开始匹配的位置 | 如果不指定默认为 1，即从 source_char 的第一个字符开始匹配。 | 表达式 |
| ***match_param*** | 可通过设置该参数改变默认的匹配功能行为 | 默认情况下“.”不匹配换行符，源字符串被看作一行。参数可选项如下：i：大小写不敏感；c：大小写敏感；n：点号(.) | 表达式 |

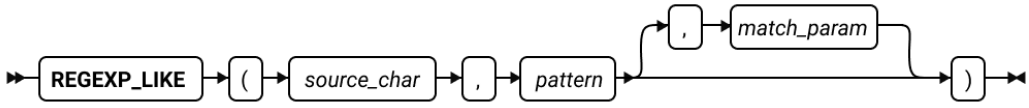
| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | 不匹配换行符号；m：多行模式；x：扩展模式，忽略正则表达式中的空白字符。 | |

例如，返回字符串中’s’出现的次数：

SELECT REGEXP_COUNT('gbase 8s', 's') FROM dual
返回结果为：2。

## *REGEXP_LIKE*

模糊匹配指定的字符串，返回源字符串中与 pattern 指定的正则表达式相匹配的字符串。




REGEXP_LIKE(source_char,pattern[,match_paramater])

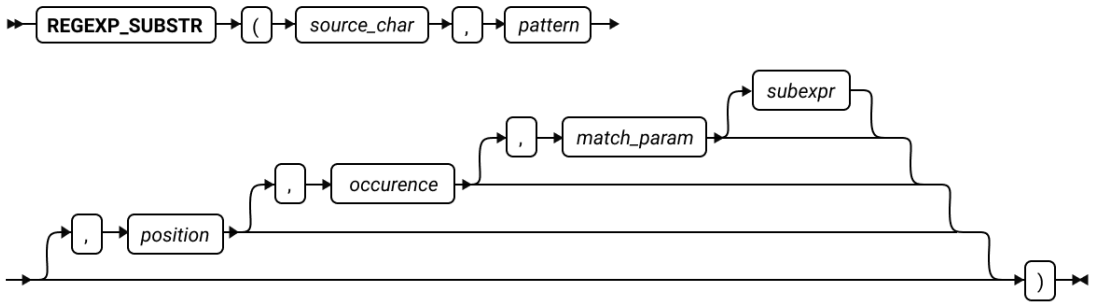
| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***source_char*** | 源字符串 | 可以是字符串或列名。当源字符串 source_char 是列名时，支持的类型为 char 和 varchar 型。 | 表达式 |
| ***pattern*** | 正则表达式 | 支持的数据类型为 char 和 varchar 型。每个正则表达式最多可包含 512 个字节。 | 表达式 |
| ***match_param*** | 可通过设置该参数改变默认的匹配功能行为 | 默认情况下“.”不匹配换行符，源字符串被看作一行。参数可选项如下：i：大小写不敏感；c：大小写敏感；n：点号(.)不匹配换行符号；m：多行模式；x：扩展模式，忽略正则表达式中的空白字符。 | 表达式 |

例如，查询表 t1 中列 FName 中含有’s’或者’S’的记录：

SELECT * FROM t1 WHERE REGEXP_LIKE(FName, 's', ‘i’)

## *REGEXP_SUBSTR*

提取指定字符串的子串，找出源字符串中与 pattern 指定的正则表达式相匹配的字符串。




REGEXP_SUBSTR(source_char,pattern[,position[,occurence[,match_option[,subexpr]]])

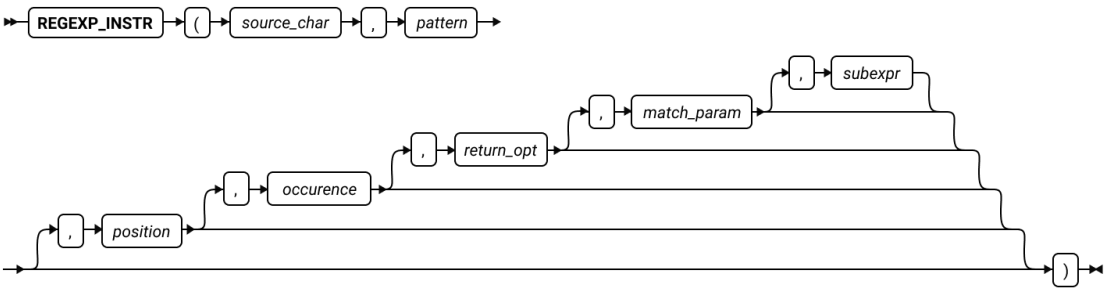
| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***source_char*** | 源字符串 | 可以是字符串或列名。当源字符串 source_char 是列名时，支持的类型为 char 和 varchar 型。 | 表达式 |
| ***pattern*** | 正则表达式 | 支持的数据类型为 char 和 varchar 型。每个正则表达式最多可包含 512 个字节。 | 表达式 |
| ***position*** | 开始匹配的位置 | 如果不指定默认为 1，即从 source_char 的第一个字符开始匹配。 | 表达式 |
| ***occurrence*** | 匹配的次数 | 如果不指定，默认为 1，即从第一次与 pattern 匹配上的字符串开始搜索，如果该值大于 1,表示忽略第一次的匹配，从指定的次数的第一个字符开始搜索。 | 表达式 |
| ***match_param*** | 可通过设置该参数改变默认的匹配功能行为 | 默认情况下“.”不匹配换行符，源字符串被看作一行。参数可选项如下：i：大小写不敏感；c：大小写敏感；n：点号(.)不匹配换行符号；m：多行模式；x：扩展模式，忽略正则表达式中的空白字符。 | 表达式 |
| ***subexpr*** | 对于含有子表达式的 pattern，subexpr 是 0~9 的整数，表示 pattern 中的第几个子串是函数目标 | Subexpr 是 pattern 中圆括号里的字符串片段，子表达式可嵌套。子表达式按照其左括号出现的顺序编号。如果 subexpr 是 0，返回整个与 pattern 匹配的字符串的位置；如果大于 0，返回指定子串的位置。如果没有与 pattern 匹配的字符串，函数返回 0，空 subexpr 返回 null。该值默认为 0。 | 表达式 |

例如：

SELECT REGEXP_SUBSTR('192.168.1.100','[^.]+',1,4) FROM dual
返回结果为：100。

## *REGEXP_INSTR*

获得匹配字符串的起始位置，返回与 pattern 指定的正则表达式相匹配的字符串在源字符串中的位置。




REGEXP_INSTR(source_char,pattern[,position[,occurence[,return_opt[,match_parameter[,subexpr]]]]])

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***source_char*** | 源字符串 | 可以是字符串或列名。当源字符串 source_char 是列名时，支持的类型为 char 和 varchar 型。 | 表达式 |
| ***pattern*** | 正则表达式 | 支持的数据类型为 char 和 varchar 型。每个正则表达式最多可包含 512 个字节。 | 表达式 |
| ***position*** | 开始匹配的位置 | 如果不指定默认为 1，即从 source_char 的第一个字符开始匹配。 | 表达式 |
| ***occurrence*** | 匹配的次数 | 如果不指定，默认为 1，即从第一次与 pattern 匹配上的字符串开始搜索，如果该值大于 1,表示忽略第一次的匹配，从指定的次数的第一个字符开始搜索。 | 表达式 |
| ***return_opt*** | 指定返回值的类型 | 如果该参数为 0，则返回值为匹配位置的第一个字符，如果该值为非 0 则返回匹配值的最后一个位置。 | 表达式 |
| ***match_param*** | 可通过设置该参数改变默认的匹配功能行为 | 默认情况下“.”不匹配换行符，源字符串被看作一行。参数可选项如下：i：大小写不敏感；c：大小写敏感；n：点号(.)不匹配换行符号；m：多行模式；x：扩展模式，忽略正则表达式中的空白字符。 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***subexpr*** | 对于含有子表达式的 pattern，subexpr 是 0~9 的整数，表示 pattern 中的第几个子串是函数目标 | Subexpr 是 pattern 中圆括号里的字符串片段，子表达式可嵌套。子表达式按照其左括号出现的顺序编号。如果 subexpr 是 0，返回整个与 pattern 匹配的字符串的位置；如果大于 0，返回指定子串的位置。如果没有与 pattern 匹配的字符串，函数返回 0，空 subexpr 返回 null。该值默认为 0。 | 表达式 |

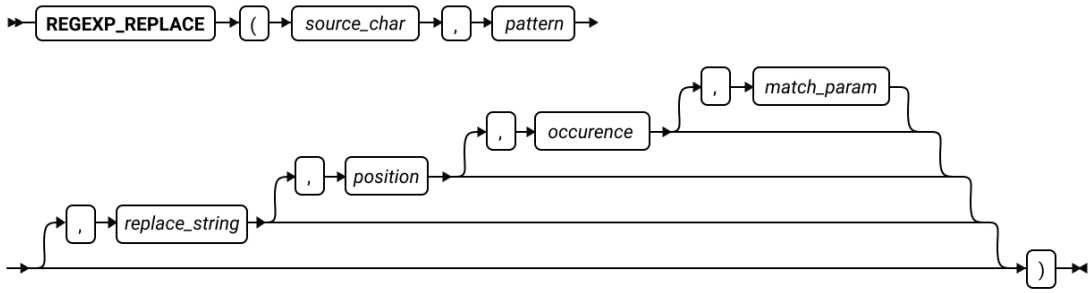
例如，找到字符串中的第一个'e'的位置：

```
SELECT REGEXP_INSTR('hello world', 'e') FROM dual
```

返回结果为：2。

## *REGEXP_REPLACE*

将匹配获得的字符串替换成指定的字符串，用 replace_string 指定的字符串替换字符串中与 pattern 指定的正则表达式相匹配的字符串。




REGEXP_REPLACE(source_char,pattern[,replace_string[,position[,occurence[match_option]]]])

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| ***source_char*** | 源字符串 | 可以是字符串或列名。当源字符串 source_char 是列名时，支持的类型为 char 和 varchar 型。 | 表达式 |
| ***pattern*** | 正则表达式 | 支持的数据类型为 char 和 varchar 型。每个正则表达式最多可包含 512 个字节。 | 表达式 |
| ***replace_string*** | 替换字符串 | 可以是字符串或列名，当替换字符串为列名时，支持的类型为 char 和 varchar 型。替换字符串至多可以包含 500 个反向引用的数字表达式（\n，n 的取值范围是 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | | [1,9]）。如果替换字符串中含有“\”,可通过转义字符“\”转义。 | |
| ***position*** | 开始匹配的位置 | 如果不指定默认为 1，即从 source_char 的第一个字符开始匹配。 | 表达式 |
| ***occurrence*** | 匹配的次数 | 如果不指定，默认为 1，即从第一次与 pattern 匹配上的字符串开始搜索，如果该值大于 1,表示忽略第一次的匹配，从指定的次数的第一个字符开始搜索。 | 表达式 |
| ***match_param*** | 可通过设置该参数改变默认的匹配功能行为 | 默认情况下“.”不匹配换行符，源字符串被看作一行。参数可选项如下：i：大小写不敏感；c：大小写敏感；n：点号(.)不匹配换行符号；m：多行模式；x：扩展模式，忽略正则表达式中的空白字符。 | 表达式 |

例如，将非数字的数据信息替换成空：

```
SELECT REGEXP_REPLACE('hjbe8723r8fb938r', '[^0-9]') FROM dual
```

返回结果为：87238938。

## 加密和解密函数

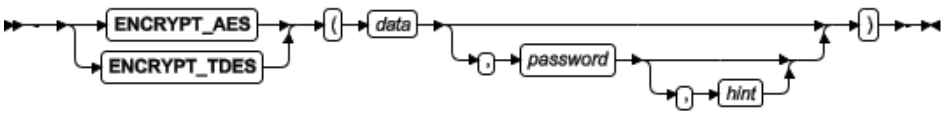
GBase 8s 支持内置加密和解密函数。

加密和解密函数支持列级别加密，即对给定列中的所有值都用相同的 password 进行加密。并且加密和解密函数不支持对大对象数据进行加解密。

### *ENCRPYPT_AES 函数*

**ENCRYPT_AES** 函数使用 AES（高级加密标准）算法进行加密。

加密函数语法：




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *data* | 要进行加密的明文或列名。 | 不可省略。 | 表达式 |
| *password* | 加密函数设置的密码。缺省值是由 SET | 6 字节≤ *password* ≤ | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | ENCRYPTION PASSWORD 语句定义的会话密码值。 | 128 字节。 | |
| *hint* | 密码提示信息。缺省值是的是来自定义 *password* 的 SET ENCRYPTION PASSWORD 的 WITH HINT 子句的值。 | 可以省略，0 字节≤ *hint* ≤32 字节。 | 表达式 |

当使用 **SET ENCRYPTION PASSWORD** 语句设置密码时，加密函数的 ***password*** 参数可以省略，否则不可省略。

***hint*** 的目的是帮助用户记忆 ***password***。

注意：如果列定义长度小于加密函数返回的数据大小，则插入此加密值时它就会被截断。在此情况下，加密数据就无法再通过解密函数解密。

例如，下列语句创建名为 **customer** 的表，其中列 **creditcard** 可存储加密的信用卡号码：

```
CREATE TABLE customer (id CHAR(20), creditcard CHAR(107));
```

指定密码（和可选的提示），使用 **ENCRYPT_AES** 函数进行加密：

```
SET ENCRYPTION PASSWORD 'abc123' WITH HINT 'zimushuzi';
INSERT INTO customer VALUES ('1001', ENCRYPT_AES('1234567890123456'));
```

在此，**SET ENCRYPTION PASSWORD** 定义的 *session password* 和 *hint*，用作 **ENCRYPT_AES** 函数缺省的第二和第三个参数。

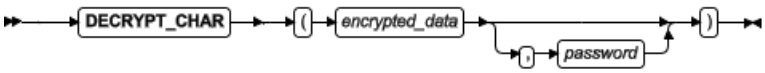
## *ENCRPYPT_TDES 函数*

**ENCRYPT_TDES** 函数使用 TDES（三重数据加密标准）算法进行加密。其用法与 **ENCRYPT_AES** 函数相同。

## *DECRYPT_CHAR 函数*

**DECRYPT_CHAR** 函数是解密函数，函数的返回值为加密前的原始数据。

DECRYPT_CHAR 函数




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *encrypted_data* | 加密函数的返回值，可以是加密后字符串或列。 | 不可省略。 | 表达式 |
| *password* | 加密函数设置的密码。缺省值是由 SET ENCRYPTION PASSWORD 语句定义的会话密码值。 | 6 字节≤ *password* ≤128 字节。 | 表达式 |

解密需要加密密码 ***password***，当已经使用 **SET ENCRYPTION PASSWORD** 语句设置密码时，可以省略 ***password*** 参数，否则不能省略此参数。

如果用于解密的 ***password*** 与用于加密的 ***password*** 不同，那么会产生错误。

例如，基于前面的示例，使用解密功能，查询加密了的数据：

```
SELECT id, DECRYPT_CHAR(creditcard,'abc123') FROM customer;
```

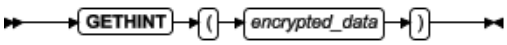
返回结果如下：

```
id          creditcard
----------------------------
1001        1234567890123456
```

## *GETHINT 函数*

**GETHINT** 函数获取密码提示信息，返回值为加密函数设置的 ***hint*** 值或先前执行 **SET ENCRYPTION PASSWORD** 语句设置的 ***hint*** 值。

GETHINT 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *encrypted_data* | 要获取密码提示的数据，可以是加密后的字符串或列。 | 不可省略。 | 表达式 |

例如，基于前面的示例，以下语句获取 customer 表的提示信息：

```
SELECT GETHINT(creditcard) FROM customer;
```

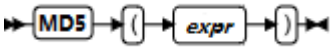
返回结果：

```
zimushuzi
```

## *MD5 函数*

MD5 函数生成并返回符合 MD5 算法的数据特征值。

MD5 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 要获取特征值的数据。 | 不可省略。不能是大对象、布尔类型、集合数据类型。 | 表达式 |

expr 设置为 NULL 时，函数返回 NULL。

例如，假定 tab1 表不为空，执行以下语句：

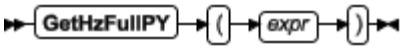
```
SELECT MD5('abc') FROM tab1;
```

# 汉字转拼音函数

## *GetHzFullPY 函数*

GetHzFullPY 函数将汉字转换为全拼。

GetHzFullPY 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 要转换为全拼的汉字。 | 不能为 NULL，可以是字符型、整数型数据。 | 表达式 |

函数返回值为字符型。使用时，除汉字按顺序转换为全拼外，其余字符（包括数字）都不进行转换，保留原值。转换完的拼音字符串缺省为小写。

例如，假定 tab1 表内容不为空，执行以下语句：

SELECT GETHZFULLPY('汉字 123') FROM tabl;

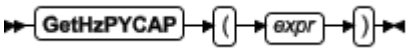
返回结果如下：

hanzi123

## *GetHzPYCAP 函数*

**GetHzPYCAP** 函数将汉字转换为拼音首字母。

GetHzPYCAP 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 要转换成拼音首字母的汉字。 | 不能为 NULL，可以是字符型、整数型数据。 | 表达式 |

函数返回值为字符型。使用时，除汉字按顺序转换为拼音首字母外，其余字符（包括数字）都不进行转换，保留原值。转换完的拼音字符串缺省为小写。

例如，假定 tab1 表内容不为空，执行以下语句：

SELECT GETHZPYCAP('汉字') FROM tabl;

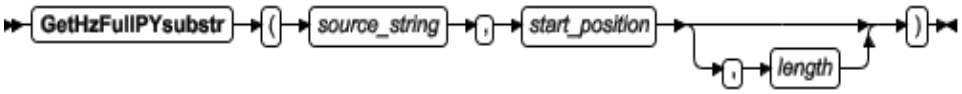
返回结果如下：

hz

## *GetHzFullPYsubstr 函数*

**GetHzFullPYsubstr** 函数将汉字转换为拼音后，提取指定个连续字符。

GetHzFullPYsubstr 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *source_string* | 要转换为拼音的汉字。 | 不能省略，必须是字符型数据。 | 表达式 |
| *start_position* | 源字符串转换为拼音后，提取字符的偏移量。 | 不能省略，不能为 0 或负数。必须是数值型数据。 | 表达式 |
| *length* | 返回的字符个数。 | 可以省略，不能为 0 或负数。必须是数值型数据。 | 表达式 |

**GetHzFullPYsubstr** 函数返回将 *source_string* 转换为拼音的字符串的子集。该子集从 *start_position* 指定的位置开始向前计数。*start_position* 值从 1 开始，值 1 表示字符串第一个字符位置。

***length*** 参数指定提取字符的长度。如果您省略 ***length*** 参数，则 **GetHzFullPYsubstr** 函数返回从 ***start_position*** 处开始的拼音字符串。

例如，假定 tab1 表不为空，执行以下语句：

```
SELECT GETHZFULLPYSUBSTR('汉字',1,2) FROM tabl;
```

返回结果如下：

```
ha
```

如果 ***length*** 为小数，则只取其整数部分。例如，执行以下语句：

```
SELECT GETHZFULLPYSUBSTR('汉字',1,2.6) FROM tabl;
```

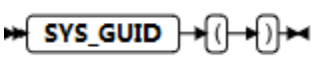
返回结果如下：

```
ha
```

## SYS_GUID 函数

**SYS_GUID** 函数生成并返回一个全球唯一标识符，它由 16 个字节组成。在大多数平台，生成的标识符由主机标识符、执行函数的进程或线程标识符、和进程或线程的一个非重复的值（字节序列）。

SYS_GUID 函数




以下示例，用户使用 SYS_GUID() 函数获得一个全球唯一标识符。

```
SELECT sys_guid() FROM sysmaster:sysdual;
```

返回结果如下：

4A4F072262CC6D8AE050A8C0EB075197

# 分组统计函数

## *GROUPING(expr) 函数*

GROUPING(expr) 函数表示参数列 expr 是否为分组列（分组列指参与分组的列），并返回一个数值。当该参数对应的列为分组列时，返回值为 0；当该参数对应的列非分组列时，返回值为 1。

GROUPING(expr) 函数


| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *expr* | 判断是否参与分组的表达式 | 必须是 GROUP BY 子句中的表达式之一 | 标识符 |

GROUPING 函数的返回值类型为整型数值。

GROUPING 函数的参数仅支持一个。

GROUPING 函数的参数 expr 必须是 GROUP BY 子句中的表达式之一。

GROUPING 函数的参数 expr 不支持为 NULL。

GROUPING_ID 作为分组函数，不能出现在 WHERE 或连接条件中。

当在视图中使用 SELECT GROUPING ()时，必须使用别名，否则报错。

使用 HAVING GROUPING ()时，不可用别名，须直接使用函数表达式。

示例：

例如，表 tab 1 表结构如下：

```
CREATE TABLE tab1 (
c1 int,
c2 int,
c3 varchar(20)
);
```

插入如下数据：

```
INSERT INTO tab1 values(1,1,'test1');
INSERT INTO tab1 values(1,2,'test2');
INSERT INTO tab1 values(2,1,'test3');
INSERT INTO tab1 values(2,2,'test4');
```

对表 tab1 的 c1、c2、c3 列进行 ROLLUP 分组，统计分组组合包括 (c1,c2,c3)，(c1,c2)，(c1)，全空 四种分组组合。

使用 GROUPING()函数返回 c1,c2,c3 列是否参与分组。

```
SELECT GROUPING (C1),GROUPING (C2),GROUPING (C3),c1,c2,c3 FROM tab1 GROUP BY ROLLUP (c1,c2,c3);
```

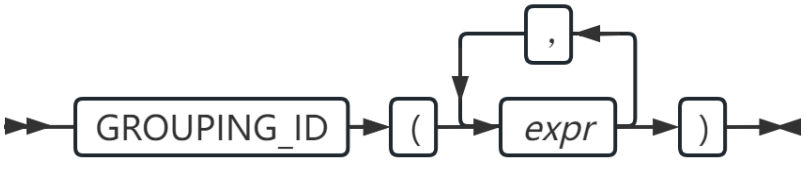
返回结果如下：

```
(grouping) |(grouping) |(grouping) |c1 |c2 |c3    |
-----------|-----------|-----------|---|---|------|
0          |0          |0          |2  |2  |test4 |
0          |0          |0          |1  |2  |test2 |
0          |0          |0          |2  |1  |test3 |
0          |0          |0          |1  |1  |test1 |
0          |0          |1          |1  |2  |      |
0          |0          |1          |2  |1  |      |
0          |0          |1          |2  |2  |      |
0          |0          |1          |1  |1  |      |
0          |1          |1          |2  |   |      |
0          |1          |1          |1  |   |      |
1          |1          |1          |   |   |      |
```

## *GROUPING_ID 函数*

GROUPING_ID 函数表示参数是否参与分组的向量值（分组统计语法中，分组组合指其中的单列或多列表达式的某种分组情况），返回数值的二进制位对应表示参数列是否为分组列。当参与分组时，参数位为 0 ；否则为 1 。

GROUPING_ID 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 判断是否参与分组的表 | 必须是 GROUP BY 子句 | 标识符 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | 达式 | 中的表达式之一 | |

返回值计算规则如下：

GROUPING_ID（expr1，expr2，……）中的每个参数 expr 对应一个列，当该参数对应的列为分组列时，该参数位的值为 0，否则为 1。最后将所有参数位的值串联在一起，组成一个 0 和 1 的二进制数，再将该二进制数转换为一个十进制数，即为 GROUPING_ID 的返回值。

例如，GROUPING_ID(c1,c2) 返回值为 2 ，对应二进制数值为 10 ，表示 c1 列未参与， c2 列参与分组。

GROUPING_ID 函数的返回值类型为整型数值。

GROUPING_ID 函数的参数支持多个。参数限制和 GROUP BY 参数限制保持一致。

GROUPING_ID 函数的参数 expr 必须是 GROUP BY 子句中的表达式的子集。

GROUPING_ID 函数的参数 expr 不支持为 NULL。

GROUPING_ID 作为分组函数，不能出现在 WHERE 或连接条件中。

当在视图中使用 SELECT GROUPING_ID ()时，必须使用别名，否则报错。

使用 HAVING GROUPING_ID ()时，不可用别名，需直接使用函数表达式。

示例：

例如，表 tab 1 表结构如下：

```
CREATE TABLE tab1 (
c1 int,
c2 int,
c3 varchar(20)
);
```

插入如下数据：

```
INSERT INTO tab1 values(1,1,'test1');
INSERT INTO tab1 values(1,2,'test2');
INSERT INTO tab1 values(2,1,'test3');
INSERT INTO tab1 values(2,2,'test4');
```

对表 tab1 的 c1、c2 列进行 CUBE 分组，统计分组组合包括 (c1,c2)，(c1)，(c2)，全空 四种分组组合。

使用 GROUPING_ID(c1,c2) 函数返回 c1,c2 列是否参与分组的向量值。

```
SELECT GROUPING (C1),GROUPING (C2),GROUPING_ID (c1,c2),c1,c2 FROM tab1 GROUP BY CUBE (c1,c2);
```

返回结果如下：

```
(grouping) |(grouping) |(grouping_id) |c1 |c2 |
-----------|-----------|--------------|---|---|
1          |1          |3             |   |   |
0          |1          |1             |2  |   |
0          |1          |1             |1  |   |
1          |0          |2             |   |1  |
1          |0          |2             |   |2  |
0          |0          |0             |1  |1  |
0          |0          |0             |1  |2  |
0          |0          |0             |2  |2  |
0          |0          |0             |2  |1  |
```

## *GROUP_ID 函数*

GROUP_ID 函数返回重复分组的唯一标识值，用于表示结果集来自于哪一个分组，区别相同分组的结果集。

GROUP_ID 函数


返回值计算规则如下：

如果有 N 个相同分组，则 GROUP_ID 取值从 0..N-1。每组的初始值为 0。

GROUP_ID 函数的返回值类型为整型数值。

GROUP_ID 作为分组函数，不能出现在 WHERE 或连接条件中。

当在视图中使用 SELECT GROUP_ID ()时，必须使用别名，否则报错。

ORDER BY 选择项使用 GROUP_ID 时，GROUP_ID 必须加别名。例如，select group_id() as gid ,c1,c2 from tab1 group by rollup (c1,c2) order by gid; 。

示例：

例如，表 tab 1 表结构如下：

```
CREATE TABLE tab1 (
c1 int,
c2 int,
c3 varchar(20)
);
```

插入如下数据：

```
INSERT INTO tab1 values(1,1,'test1');
INSERT INTO tab1 values(1,2,'test2');
INSERT INTO tab1 values(2,1,'test3');
INSERT INTO tab1 values(2,2,'test4');
```

对表 tab1 的 c1、c1、c1 列进行 GROUPING SETS 分组，统计分组组合包括 (c1)，(c1)，(c1) 三种分组组合。

使用 GROUP_ID () 函数返回重复分组 (c1) 的唯一标识值。

```
SELECT GROUP_ID(),c1 FROM tab1 GROUP BY GROUPING SETS (c1,c1,c1);
```

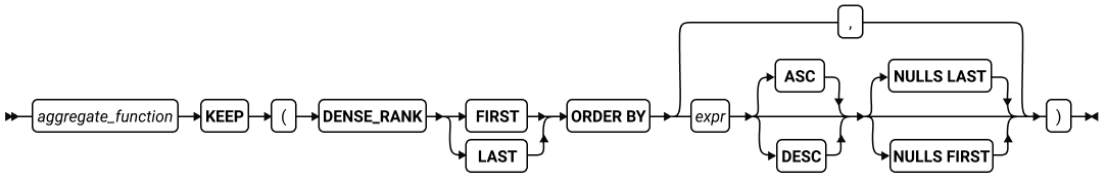
返回结果如下：

```
group_id() |c1 |
-----------|---|
2          |2  |
2          |1  |
1          |2  |
1          |1  |
0          |2  |
0          |1  |
```

# 分析函数

## *KEEP(DENSE_RANK FIRST/LAST)函数*

**KEEP(DENSE_RANK FIRST/LAST)**函数首先根据 SQL 语句中的 group by 分组（如果没有指定分组则所有结果集为一组），然后在组内进行排序。根据 FIRST/LAST 计算第一名（最小值）或者最后一名（最大值）的指定字段的值。




| 元素 | 描述 | 限制 |
| --- | --- | --- |
| ***aggregate_function*** | 聚集函数名称。 | 不能省略。 |
| ***exrp*** | 按照此表达式排序的列名或常量表达式。 | 不能为聚集表达式。 |

FIRST 和 LAST 函数对一组行中的一组值进行聚集操作，这些行根据给定的排序规则排序。如果 FIRST 或 LAST 的结果只有一行，则聚集只对一行进行操作。

aggregate_function 是 MIN、MAX、SUM、AVG、COUNT、RANGE、VARIANCE 或 STDDEV 函数中的任何一个。它对来自 FIRST 或 LAST 排序的行的值进行聚集操作。如果 FIRST 或 LAST 只有一行结果，则聚集只对一行进行操作。

KEEP 关键字用于语义清晰度。它限定了 aggregate_function，表示只返回 aggregate_function 的 FIRST 或 LAST 值。

DENSE_RANK FIRST 或 DENSE_RANK LAST 表示仅聚集具有最小（FIRST）或最大（LAST）值的行。

示例：

获取部门内年龄最小的人中，工资最高的记录。首先构造一下临时数据：

```
WITH workers AS(
   SELECT 'DOM1' dept, 'zsan' names       , 23 age, 4000 salaries FROM dual UNION ALL
   SELECT 'DOM1' dept, 'lisi' names       , 35 age, 9000 salaries FROM dual UNION ALL
   SELECT 'DOM2' dept, 'wangwu' names     , 26 age, 6500 salaries FROM dual UNION ALL
   SELECT 'DOM2' dept, 'maliu' names      , 28 age, 6000 salaries FROM dual UNION ALL
   SELECT 'DOM2' dept, 'zhaoqi' names     , 26 age, 5000 salaries FROM dual UNION ALL
   SELECT 'DOM1' dept, 'liba' names       , 23 age, 3000 salaries FROM dual
)
```

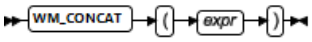
在这六条数据中，我们期望的数据是：(DOM1,4000)和(DOM2,6500)，SQL 如下：

```
SELECT w.dept, MAX(w.salaries)
   KEEP(DENSE_RANK FIRST ORDER BY w.age) max_salary
FROM workers w
WHERE 1=1
GROUP BY dept;
```

# 列转行函数

## *WM_CONCAT 函数*

WM_CONCAT 函数可以将结果集中指定列的数据合并成一行。WM_CONCAT 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 要合并为一行的数据 | 可以是数值型、字符型和日期型数据。 | 表达式 |

函数返回值为 LVARCHAR 数据类型。返回值的长度不能超过 16380 字节。转换完指定列的多条记录合并到一行，多个记录使用逗号分隔。

如果返回值的长度超过 16380 字节，则会截断超长的字符，保存截断后的结果。所以对于列转行函数返回值长度超过 16380 字节的情况，请使用 WM_CONCAT_TEXT() 函数以保存完整值。

例如，假定 tab1 表内容不为空：

```
SELECT col1 FROM tabl;
```

返回结果如下：

```
col1
-----------
1001
1002
1003
1004
```

执行以下语句：

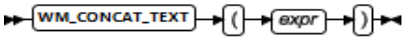
```
SELECT wm_concat (col1) col1 FROM tabl;
```

返回结果如下：

```
col1
-----------------------
1001,1002,1003,1004
```

## *WM_CONCAT_TEXT 函数*

WM_CONCAT_TEXT 函数可以将结果集中指定列的数据合并成一行。WM_CONCAT_TEXT 函数



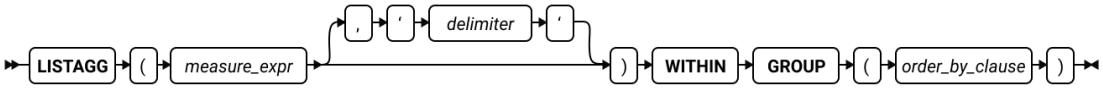

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expr* | 要转为一行的数据 | 可以是数值型、字符型和日期型数据。 | 表达式 |

函数返回值为 TEXT 数据类型。转换完指定列的多条记录合并到一行，多个记录使用逗号分隔。

当函数的返回值超过 16380 字节时，推荐使用该函数。

## *LISTAGG 函数*

LISTAGG 函数可以将多行记录合并成单行。LISTAGG 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| ***measure_expr*** | 需要合并多行记录的表达式 | 支持列名、常量、列表达式，不支持大对象、集合等复杂数据类型，支持普通函数嵌套，不支持聚合函数嵌套。如果为''或者 Null，则函数返回 Null。 | 表达式 |
| ***delimiter*** | 分隔多行记录的分隔符 | 使用时，用''包围，支持字母、数字、下划线及,\|*"等字符，不支持'。支持一个或多个字符，多字符不少于 3 个。支持省略，省略时按空字符处理。 | 引用字符串 |
| ***order_by_clause*** | 多行记录合并时，按照该参数指定的列排序 | 支持多个列名，多列名时，按照从前到后的顺序依次排序。不支持省略。 | 表达式 |

函数的返回值类型为字符型，WITHIN GROUP 子句不能省略，与 group by 子句连用时为聚集函数，返回的结果集按照 group by 子句分组，同一组内 measure_expr 的多个记录按照 order_by_clause 指定列名排序合并成一行记录，结果集的条数是组的个数，且投影列必须包含 group by 子句的列。不带 group by 子句使用时，函数返回值为单行，此时不能与返回多行的表达式同时出现在投影列中。注意 LISTAGG 函数暂不支持与 OVER 子句连用。

例如，用户表 TUser，字段 FDepartmentID 记录了用户所属部门 id，可以通过 LISTAGG 函数得到各个部门下所有用户：

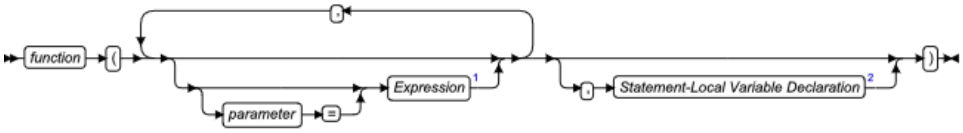
```
SELECT FDepartmentID, LISTAGG(FName, ',')
WITHIN GROUP(order by FDepartmentID) FName
FROM TUser
GROUP BY FDepartmentID
```

# 用户定义的函数

用户定义的函数（UDF）是一个例程，您以 SPL 或诸如 C 或 Java™ 这样的数据库的外部语言编写该例程，且该例程将值返回到它的调用上下文。

作为表达式，UDF 有下列语法：

User-Defined Functions




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *function* | 函数的名称 | 函数必须存在 | 数据库对象名 |
| *parameter* | 在 CREATE FUNCTION 语句中声明了的参数的名称 | 对于被调用的函数中的任何参数，如果您使用 *parameter* = 选项，则您必须对所有参数使用它 | 标识符 |

您可在 SQL 语句内调用用户定义的函数。与内建的函数不一样，仅可由该函数的创建者和 DBA 以及已被授予了对该函数的 Execute 权限的用户可调用用户定义的函数。要获取更多信息，请参阅 例程级权限。

下列示例展示某些用户定义的函数表达式。在第一个示例罗列函数参数时，省略它的 *parameter* 选项：

```
read_address('Miller')
```

第二个示例使用 *parameter* 选项来指定参数值：

```
read_address(lastname = 'Miller')
```

当您使用 *parameter* 选项时，*parameter* 名称必须与该函数注册中对应的参数的名称相匹配。例如，前面的示例假设注册了 read_address( ) 函数如下：

```
CREATE FUNCTION read_address(lastname CHAR(20))
RETURNING address_t ... ;
```

*语句本地的变量*（SLV）使得应用能够将值从用户定义的函数调用传递到同一 SQL 语句的另一部分。

**随同用户定义的函数调用来使用 SLV**

1. 为用户定义的函数写一个或多个 OUT 参数（对于以 Java 或以 SPL 语言编写的 UDR，为 INOUT 参数）。

   要获取关于如何以 OUT 或 INOUT 参数编写 UDR 的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

2. 当您注册用户定义的函数时，请在每一 OUT 参数之前指定 OUT 关键字，并在每一 INOUT 参数之前指定 INOUT 关键字。

   要获取更多信息，请参阅 为用户定义的例程指定 INOUT 参数 和 为用户定义例程指定 OUT 参数。

3. 请在以每一 OUT 和 INOUT 参数调用用户定义的函数的函数表达式中声明 SLV。

   必须在 WHERE 内进行用户定义的函数的调用。要获取关于声明 SLV 的语法的信息，请参阅 语句本地的变量声明。

4. 请在 SQL 语句内使用该用户定义的函数已初始化了的 SLV。

   在用户定义的函数的调用已初始化了 SLV 之后，您可在其中声明了 SLV 的同一 SQL 语句的其他部分中使用此值，包括其 WHERE 子句包括该 SLV 声明的查询的子查询。要获取关于在 SELECT 语句内使用 SLV 的信息，请参阅 语句本地的变量表达式。

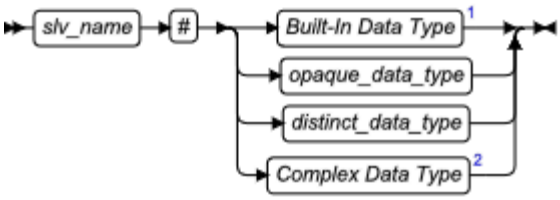
除了使用 SLV 来从 OUT 或 INOUT 参数检索值之外，您还可使用本地变量或 SPL 例程的参数来从有 OUT 或 INOUT 参数的 SPL 或 C 例程检索值。

## *语句本地的变量声明*

“语句本地的变量声明”在对定义一个或多个 OUT 或 INOUT 参数的用户定义的函数的调用中声明语句本地的变量（SLV）。

“语句本地的变量声明”有此语法：

语句本地的变量声明




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *distinct_data_type* | distinct 数据类型的名称 | distinct 数据类型必须在数据库中已经存在 | 标识符 |
| *opaque_data_type* | opaque 数据类型的名称 | opaque 数据类型必须在数据库中已经存在 | 标识符 |
| *slv_name* | 您正在定义的语句本地的变量的名称 | *slv_name* 仅在该语句的生命期内有效，且 在该语句内必须是唯一的 | 标识符 |

如果下列两个条件都为真，则您可在一个用户定义的函数的调用中声明 SLV：

- UDF 有一个或多个 OUT 或 INOUT 参数
- 当在查询的 WHERE 子句中调用 UDF 时，声明 SLV。

WHERE 子句中的 SLV 声明将 OUT 或 INOUT 参数的值指定到 SLV，使用 SLV 的标识符与它的声明的数据类型之间的井号（#）。可以 SPL、C 或 Java™ 语言编写 UDF。例如，如果您以下列 CREATE FUNCTION 语句注册函数，则可将它的 y 参数（其为 OUT 参数）的值指定到 WHERE 子句中的 SLV：

```
CREATE FUNCTION find_location(a FLOAT, b FLOAT, OUT y INTEGER)
RETURNING VARCHAR(20)
EXTERNAL NAME "/usr/lib/local/find.so"
LANGUAGE C;
```

在此示例中，**find_location( )** 接受两个表示纬度和经度的 FLOAT 值，并返回带有额外的表示最近城市的人口等级的 INTEGER 类型的值的城市名称。

现在，您可在 WHERE 子句中调用 find_location( )：

```
SELECT zip_code_t FROM address
WHERE address.city = find_location(32.1, 35.7, rank # INT)
AND rank < 101;
```

该函数表达式将两个 FLOAT 值传递到 **find_location( )** 并声明名为 **rank** 的 INT 类型的 SLV。在此情况下，**find_location( )** 将返回距离纬度 32.1 和经度 35.7 最近的城市（可能是人口稠密的地区）的名称，其人口等级在 1 与 100 之间。然后，该语句返回对应于那个城市的邮政编码。

仅在 SELECT 语句的 WHERE 子句中的 UDR 的调用中可声明 SLV。对该 SLV 的引用的作用域包括同一 SELECT 语句的其他部分。然而，下列 SELECT 语句是***无效的***，因为 SLV 声明是在 Projection 子句的 projection 列表之中，而不是在 WHERE 子句中：

```
-- 无效的 SELECT 语句
SELECT title, contains(body, 'dog and cat', rank # INT), rank
FROM documents;
```

当您声明 SLV 时，您指定的数据类型必须与 CREATE FUNCTION 语句中对应的 OUT 或 INOUT 参数的数据类型相同。如果您使用不同的但相兼容的数据类型，比如 INTEGER 和 FLOAT，则数据库服务器自动地在数据类型之间执行强制转型。

SLV 与 UDR 变量以及涉及 SQL 语句的表的列名称共享该命名空间。 因此，数据库使用下列优先顺序的降序来解决下列对象之间的名称冲突：

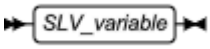
- UDR 变量
- 列名称
- SLV

在 UDF 的调用将 OUT 或 INOUT 参数的值指定到 SLV 之后，您可在同一查询的其他部分引用该 SLV。要获取更多信息，请参阅 语句本地的变量表达式。

## 语句本地的变量表达式

“语句本地的变量表达式”指定您可在同一 SELECT 语句中的其他地方使用的语句本地的变量（SLV）。

语句本地的变量表达式




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *SLV_variable* | 在同一查询中的用户定义的函数的调用中指定的语句本地的变量（SLV） | *SLV_variable* 仅在查询执行期间存在。在该查询中它的名称必须是唯一的 | 标识符 |

请您在 SELECT 语句的 WHERE 子句中对用户定义的函数的调用中定义 SLV。必须以一个或多个 OUT 或 INOUT 参数来定义用户定义的函数。对用户定义的函数的调用将 OUT 或 INOUT 参数的值指定到 SLV。要获取更多信息，请参阅 语句本地的变量声明。

一旦用户定义的函数将它的 OUT 或 INOUT 参数指定到 SLV，您可在同一 SELECT 语句的其他部分中使用这些值，服从下列引用的作用域规则：

- 在定义 SLV 的查询（或子查询）中该 SLV 是*只读的*。
- SLV 的作用域从定义该 SLV 的查询向下扩展至所有嵌套的子查询内。
- 在嵌套的查询中，SLV 的作用域不向上扩展。

  换句话说，如果查询包含一个或多个子查询，则在该查询中定义的 SLV 对于那个查询的所有子查询也是可见的。 但如果在子查询中定义 SLV，则它对于父查询是不可见的。
- 在包括 UNION 运算符的查询中，该 SLV 仅在定义它的查询中是可见的。

  该 SLV 对于在 UNION 中指定的任何其他查询都不是可见的。
- 对于 INSERT、DELETE 和 UPDATE 语句，在该语句的 SELECT 部分之外，SLV 不是可见的。

  在 DML 语句的此 SELECT 部分中，上述作用域规则都适用。

**重要：** 仅对于在单个 SQL 语句期间，语句本地的变量在作用域中。

下列 SELECT 语句调用 WHERE 子句中的 find_location( ) 函数，并定义 rank SLV。在此，find_location( ) 接受表示纬度和经度的两个值，并返回最近的城市的名称，带有表示该城市人口等级的 INTEGER 类型的额外的值。

```
SELECT zip_code_t FROM address
    WHERE address.city = find_location(32.1, 35.7, rank # INT)
    AND rank < 101;
```

当成功地完成 find_location() 函数的执行时，该函数已初始化了 rank SLV。然后，SELECT 在第二个 WHERE 子句条件中使用此 rank 值。在此示例中，“语句本地的变量表达式”是第二个 WHERE 子句条件中的变量 rank：

```
rank < 101
```

一个 UDF 可有的 OUT 和 INOUT 城市以及 SLV 的数目不受限制。（早于 Version 9.4 的 GBase 8s 产品将用户定义的函数限制为单个 OUT 参数且没有 INOUT 参数，因此限定 SLV 的数目不超过一个。）

如果在一个语句的迭代中*未*执行初始化该 SLV 的用户定义的函数，则每一 SLV 有一 NULL 值。跨语句的迭代中，不保持 SLV 的值。在每一迭代的开始时刻，数据库服务器将该 SLV 值设置为 NULL。

下列部分语句调用两个带有 OUT 参数的用户定义的函数，以 SLV 名称 out1 和 out2 引用其值：

```
SELECT...
WHERE func_2(x, out1 # INTEGER) < 100
AND (out1 = 12 OR out1 = 13)
AND func_3(a, out2 # FLOAT) = "SAN FRANCISCO"
AND out2 = 3.1416;
```

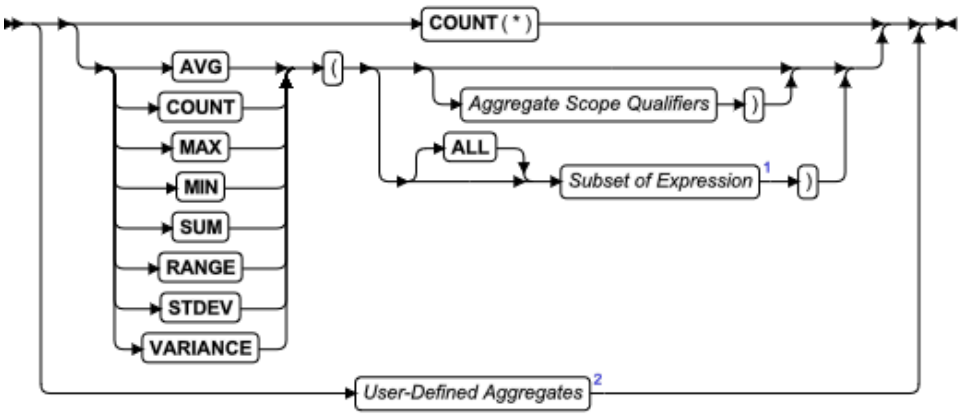
如果函数将来自本地数据库服务器的另一数据库的一个或多个 OUT 或 INOUT 参数值指定到 SLV，则这些值必须为内建的数据类型，或其基础类型为内建的数据类型（以及您显式地强制转型为内建的数据类型）的 DISTINCT 数据类型，或必须为您显式地强制转型为内建的数据类型的的 opaque UDT。在所有的参与数据库中，所有 opaque UDT、DISTINCT 类型、类型层级和强制转型的定义都必须完全相同。

要获取关于如何编写带有 OUT 或 INOUT 参数的用户定义的函数的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

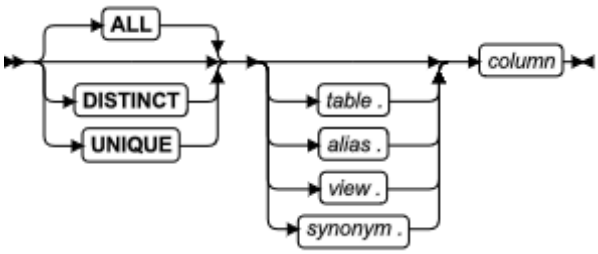
## 聚集表达式

聚集表达式使用聚集函数来汇总选择的数据库数据。内建的聚集函数有下列语法：

聚集表达式




聚集作用域限定符



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *column* | 要应用聚集函数的列 | 请参阅后面页上的单个关键字的标题 | 标识符 |
| *alias*、*synonym*、*table*、*view* | 包含 *column* 的同义词、表、视图或别名 | *同义词* 以及它指向的*表*或*视图*必须存在 | 标识符 |

您不可在作为 WHERE 子句的一部分的条件中使用聚集表达式，除非您在子查询内使用该聚集表达式。您不可将聚集函数应用于 BYTE 或 TEXT 列。要了解其他一般的限制，请参阅 聚集表达式中有效的表达式的子集。

聚集函数为一组查询到的行返回一个值。下列示例展示 SELECT 语句中的聚集函数：

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013;
SELECT COUNT(*) FROM orders WHERE order_num = 1001;
SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer;
```

如果您使用聚集函数以及 Projection 子句的 projection 列表中的一个或多个列，则您必须包括所有列名称，不将这些用作 GROUP BY 子句中的聚集或时间表达式的一部分。

## 聚集表达式的类型

SQL 语句可包括*内建的*聚集和*用户定义的*聚集。内建的聚集包括在 聚集表达式 中的语法图中展示的除了“用户定义的聚集”类别之外的所有聚集。用户定义的聚集是用户以 CREATE AGGREGATE 语句创建的任何新的聚集。

### *内建的聚集*

内建的聚集是由数据库服务器定义的聚集函数，比如 AVG、SUM 和 COUNT。这些聚集仅与诸如 INTEGER 和 FLOAT 这样的内建的数据类型一起工作。您可将这些内建的聚集扩展到与扩展的数据类型工作。要扩展内建的聚集，您必须创建重置若干二目运算符的 UDR。

在仅重置内建的聚集的二目运算符之后，您可在 SQL 语句中随同扩展的数据类型使用那个聚集。例如，如果您已重载了 SUM 聚集的 **plus** 运算符来与指定的行类型工作，并将此行类型指定到 **complex_tab** 表的 **complex** 列，则您可将 SUM 聚集应用到 **complex** 列：

```
SELECT SUM(complex) FROM complex_tab;
```

要获取更多关于如何扩展内建的聚集的信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 。要获取关于如何调用内建的聚集的信息，请参阅下面页中个别内建的聚集的描述。

### *用户定义的聚集*

用户定义的聚集是您定义的来执行数据库服务器不提供的聚集计算的聚集。例如，您可创建名为 SUMSQ 的用户定义的聚集，返回指定列的平方值的合计。用户定义的聚集可与内建的数据类型或扩展的数据类型或两者一起工作，这依赖于您如何为用户定义的聚集定义支持函数。

要创建用户定义的聚集，请使用 CREATE AGGREGATE 语句。在此语句中，您命名新的聚集并为该聚集指定支持函数。一旦您创建了新的聚集及其支持函数，则可在 SQL 语句中使用该聚集。例如，如果您创建了 SUMSQ 聚集并指定了它与 FLOAT 数据类型一起工作，则您可将 SUMSQ 聚集应用于 test 表中名为 digits 的 FLOAT 列：

```
SELECT SUMSQ(digits) FROM test;
```

要获取更多关于如何创建用户定义的聚集的信息，请参阅 CREATE AGGREGATE 语句 以及 *GBase 8s 用户定义的例程和数据类型开发者指南* 中对用户定义的聚集的讨论。要获取如何调用用户定义的聚集的信息，请参阅 用户定义的聚集。

## 聚集表达式中有效的表达式的子集

如在 聚集表达式 和 用户定义的聚集 的图中指明的那样，当您使用聚集表达式时，不是所有表达式都可用。例如，聚集函数的参数自身不可包含聚集函数。在下列上下文中，您不可使用聚集函数：

- 在 WHERE 子句中，但有这两个例外：
  - 除非在 WHERE 子句内的子查询的 Projection 子句中指定该聚集，
  - 或除非该查询在来自父查询的相关列上，且 WHERE 子句在 HAVING 子句内的子查询中。
- 作为聚集函数的一个参数。

  下列嵌套的聚集表达式不是有效的：

  ```
  MAX (AVG (order_num))
  ```
- 在任何下列数据类型的列上：
  - 大对象（BLOB、BYTE、CLOB、TEXT）
  - 集合数据类型（LIST、MULTISET、SET）
  - ROW 数据类型（命名的或未命名的）
  - OPAQUE 数据类型（除了支持 opaque 类型的用户定义的聚集函数）。

您不可使用集合数据类型的列作为下列聚集函数的参数：

- **AVG**
- **SUM**
- **MIN**
- **MAX**

内建的聚集的表达式或 ***column*** 参数（除了 **COUNT**、**MAX**、**MIN** 和 **RANGE** 之外）必须返回数值或 INTERVAL 数据类型，但 **RANGE** 也接受 DATE 和 DATETIME 参数。

对于 SUM 和 AVG，您不可直接地使用两个 DATE 值之间的差异作为聚集的参数，但您可使用 DATE 差异作为算术表达式参数内的运算对象。例如：

```
SELECT . . .  AVG(ship_date - order_date);
```

返回错误 -1201，但下列等同的表达式是有效的：

```
SELECT . . .  AVG((ship_date - order_date)*1);
```

下列查询片段使用有效的语法来为两个列表达式声明别名：

```
SELECT . . .
SUM(orders.ship_charge) as o2,
COUNT(DISTINCT
CASE WHEN orders.backlog MATCHES 'n'
THEN orders.order_num END ) AS o3,
. . .
```

在此，SUM 的参数是 MONEY(6) 列值，且 COUNT DISTINCT 聚集采用 CASE 表达式作为它的参数。

## 包括或排除结果集中的重复值

您可使用 ALL、DISTINCT 或 UNIQUE 关键字来限定聚集函数的作用域。

如果您包括聚集作用域限定符，则它必须为参数列表中的第一项。

ALL 关键字指定在计算中使用从列或表达式选择的所有值，包括任何重复的值。由于 ALL 是聚集函数的缺省的作用域，因此下列两个聚集表达式是等同的，且它们的返回值基于 ship_weight 列中所有符合条件的行的值：

```
AVG(ship_weight)
AVG(ALL ship_weight)
```

包括 DISTINCT 关键字作为聚集函数的第一个参数将它的后续的参数限定到来自指定的列的唯一值。在此上下文中，UNIQUE 与 DISTINCT 关键字是同义词。下列两个聚集表达式是等同的，且他们的返回值基于 ship_weight 列的符合条件的行中唯一值的集合：

```
AVG(DISTINCT ship_weight)
AVG(UNIQUE ship_weight)
```

如果几个符合条件的行有相同的 ship_weight 值，则在计算该聚集的值中仅包括那个值的一个实例。

如果查询包括 Projection 子句中的 DISTINCT 或 UNIQUE 关键字（而不是 ALL 关键字或没有关键字），其 Select 列表还包括其参数列表以 DISTINCT 或 UNIQUE 关键字开头的聚集函数，则数据库服务器发出错误，如在下列示例中所示：

```
SELECT DISTINCT AVG(DISTINCT ship_weight)
FROM orders;
```

也就是说，在 Projection 子句和聚集函数的同一查询中，要将结果集限制到唯一的值是无效的。

然而，如果 Projection 子句为指定 SELECT 语句的 DISTINCT 或 UNIQUE 关键字，则该查询可包括一个或多个聚集函数，每一函数包括 DISTINCT 或 UNIQUE 关键字作为参数列表中的第一个规范，如在下列示例中所示：

```
SELECT AVG(UNIQUE ship_weight), COUNT (DISTINCT customer_num) FROM orders;
```

## AVG 函数

**AVG** 函数返回指定的列或表达式中所有值的平均值。

您仅可对数值列应用 AVG 函数。下列示例中的查询找到头盔的平均价格：

```
SELECT AVG(unit_price) FROM stock WHERE stock_num = 110;
```

通过将 **unit_price** 值的总和除以符合条件的行的基数来计算返回值。

如果您使用 DISTINCT 或 UNIQUE 关键字作为第一个参数，则仅从指定的列或表达式中 distinct 值计算平均值（表示*平均*）。在下列示例中，当计算总和和基数时，仅包括任何重复的值的一个实例：

```
SELECT AVG(DISTINCT unit_price) FROM stock WHERE stock_num = 110;
```

如果该数据集不包括重复的值，则上述两个示例都返回同样的 AVG 值。

忽略 NULL 值，除非该列或表达式中的每个值都是 NULL。如果每个值都是 NULL，则 **AVG** 函数为那个列或表达式返回 NULL。

## COUNT 函数概述

**COUNT** 函数实际是使得您能够根据 COUNT 关键字之后的参数，以不同的方式对列值进行计数的一组函数。

在下列小节中，说明 **COUNT** 函数的每一形式。要了解 **COUNT** 函数的不同形式的对比，请参阅 COUNT 函数的参数。

## COUNT(*) 函数

**COUNT (*)** 函数返回满足 SELECT 语句的 WHERE 子句的行数。

下列示例找到在 stock 表中有多少行在 manu_code 列中有值 HRO：

```
SELECT COUNT(*) FROM stock WHERE manu_code = 'HRO';
```

下列示例查询“系统监视接口”(SMI）表之一来找到 customer 表中 extent 的数目：

```
SELECT COUNT(*) FROM sysextents WHERE dbs_name = 'stores' AND tabname = customer";
```

您可使用 COUNT(*) 作为此一般格式的查询中的 Projection 子句来从 SMI 表获取信息。要了解关于 sysextents 和其他 SMI 表的信息，请参阅描述 sysmaster 数据库的 *GBase 8s 管理员参考手册* 章节。

如果 SELECT 语句没有 WHERE 子句，则 COUNT (*) 函数返回该表中行的总数。下列示例找到在 stock 表中有多少行：

```
SELECT COUNT(*) FROM stock;
```

如果 SELECT 语句包含 GROUP BY 子句，则 COUNT (*) 函数反映在每一组中值的数目。下列示例按第一个名称分组；如果数据库服务器发现同一名称多次出现，则选择这些行：

```
SELECT fname, COUNT(*) FROM customer GROUP BY fname
HAVING COUNT(*) > 1;
```

如果一行或多行的值为 NULL，则 **COUNT (*)** 函数在计数中包括 NULL 列，除非 WHERE 子句显式地省略它们。

# COUNT DISTINCT 和 COUNT UNIQUE 函数

**COUNT DISTINCT** 和 **COUNT UNIQUE** 函数返回唯一的值。

COUNT DISTINCT 函数返回列或表达式中唯一值的数目，如下例所示。

SELECT COUNT (DISTINCT item_num) FROM items;

如果 **COUNT DISTINCT** 函数遇到 NULL 值，则它忽略它们，除非指定的列中的每个值都是 NULL。如果每个列值都是 NULL，则 **COUNT DISTINCT** 函数返回零（0）。

UNIQUE 关键字与 COUNT 函数中的 DISTINCT 关键字有相同的含义。UNIQUE 关键字指导数据库服务器返回列或表达式中唯一的非 NULL 值的数目。下列示例调用 COUNT UNIQUE 函数，但它等同于调用 COUNT DISTINCT 函数的前一示例：

SELECT COUNT (UNIQUE item_num) FROM items;

如果 Projection 子句未指定 SELECT 语句的 DISTINCT 或 UNIQUE 关键字，则该查询可包括多个 COUNT 函数，每一函数包括 DISTINCT 或 UNIQUE 关键字作为参数列表中的第一个规范，如下例所示：

SELECT COUNT (UNIQUE item_num), COUNT (DISTINCT order_num) FROM items;

# COUNT 列函数

COUNT *列*函数返回列或表达式中非 NULL 值的总数目，如下例所示：

SELECT COUNT (item_num) FROM items;

为清楚起见，可将 ALL 关键字置于指定的列名称前面，但不论您包括 ALL 关键字还是省略它，查询结果都一样。

下列示例展示如何在 COUNT *列*函数中包括 ALL 关键字：

SELECT COUNT (ALL item_num) FROM items;

COUNT 函数的参数

**COUNT** 函数接受其他内建的聚集函数的参数列表中允许的相同的表达式作为它的参数，以及仅 **COUNT** 支持的星号（*）表示法。支持下列内建的表达式的类别作为 **COUNT** 的参数，如下列示例所示：

- 算术表达式

  COUNT(times(gbasedbt.sysfragments.evalpos,2))

  SELECT COUNT(a+1), COUNT(2*a), COUNT(5/a), COUNT(times(a, 2)) FROM myTable;
- 位逻辑函数

  COUNT(BITAND(gbasedbt.systables.flags,1))

  SELECT COUNT(BITAND(a,1)), COUNT(BITOR(8, 20)), COUNT(BITXOR(41, 33)),

  COUNT(BITANDNOT(20,-20)), COUNT(BITNOT(8)) FROM myTable;
- 强制转型表达式

  COUNT(NULL::int)

- 条件表达式

  COUNT(CASE WHEN stock.description = "baseball gloves" THEN 1 ELSE NULL END)

  SELECT COUNT(CASE WHEN s=14 THEN 1 ELSE NULL END) AS cnt14 FROM all_types;

  SELECT COUNT(NVL (ch, 'Addr unk')) FROM all_types;

  SELECT COUNT(NULLIF(ch, NULL)) FROM all_types;

- 常量表达式

  COUNT(CURRENT_ROLE)

  COUNT(DATETIME (2007-12-6) YEAR TO DAY)

  SELECT COUNT("XX"), COUNT(99),COUNT("t") FROM sysmaster:sysdual;

  SELECT COUNT(SET{6, 9, 9, 4}) FROM sysmaster:sysdual;

  SELECT COUNT("ROW(7, 3, 6.0, 2.0)") FROM sysmaster:sysdual;

  SELECT COUNT(USER), COUNT(CURRENT), COUNT(SYSDATE) from sysmaster:sysdual;

  SELECT COUNT(CURRENT_ROLE), COUNT(DEFAULT_ROLE) from sysmaster:sysdual;

  SELECT COUNT(DBSERVERNAME), COUNT(TODAY), COUNT(CURRENT) from sysmaster:sysdual;

  SELECT COUNT(DATETIME (2007-12-6) YEAR TO DAY) from sysmaster:sysdual;

  SELECT COUNT(INTERVAL (16) DAY TO DAY) FROM sysmaster:sysdual;

  SELECT COUNT(5 UNITS DAY) FROM sysmaster:sysdual;

- 函数表达式

  COUNT(LENGTH ('abc') + LENGTH (stock.description}

  COUNT(DBINFO('sessionid'))

  COUNT(user_proc())  --> proc() 是用户定义例程列表达式

  COUNT(gbasedbt.sysfragauth.fragment)

您还可使用星号（*）字符，或列名称，或带有 ALL、DISTINCT 或 UNIQUE 聚集作用域限定符的列名称作为 **COUNT** 函数的参数，来检索关于表的不同类型的信息。下面的表格总结带有星号或列名称参数的 **COUNT** 函数的每一下列形式的含义。

| COUNT 函数 | 描述 |
|---|---|
| COUNT (*) | 返回满足查询的行的数目。如果您未指定 WHERE 子句，此函数返回表中行的总数目。 |
| COUNT (DISTINCT) 或 COUNT (UNIQUE) | 返回指定的类中唯一的非 NULL 值的数目 |
| COUNT (*column*) 或 COUNT (ALL *column*) | 返回指定的列中非 NULL 值的总数目 |

有些示例可帮助展示引用一列的不同形式的 **COUNT** 函数之间的差异。大部分下列示例查询对应的是 **stores_demo** 演示数据库中 **orders** 表的 **ship_instruct** 列。要获取关于 **orders** 表的模式以及 **ship_instruct** 列中的数据值的信息，请参阅 *《GBase 8s SQL 指南：参考》* 中对演示数据库的描述。

### *COUNT(*) 函数的示例*

在下列示例中，用户想要知道 orders 表中行的总数目。于是，用户在不带有 WHERE 子句的 SELECT 语句中调用 COUNT(*) 函数：

```
SELECT COUNT(*) AS total_rows FROM orders;
```

下列表格展示此查询的结果。

| total_rows |
|---|
| 23 |

在下列示例中，用户想要知道在 orders 表中有多少行在 ship_instruct 列中有 NULL 值。用户在带有 WHERE 子句的 SELECT 语句中调用 COUNT(*) 函数，并在 WHERE 子句中指定 IS NULL 条件：

```
SELECT COUNT (*) AS no_ship_instruct FROM orders
        WHERE ship_instruct IS NULL;
```

下列表格展示此查询的结果。

| no_ship_instruct |
|---|
| 2 |

在下列示例中，用户想要知道在 orders 表中有多少行在 ship_instruct 列中有值 express。 于是，用户在 projection 列表中调用 COUNT(*) 函数，并在 WHERE 子句中指定等于（=）关系运算符。

```
SELECT COUNT (*) AS ship_express FROM ORDERS
        WHERE ship_instruct = 'express';
```

下列表格展示此查询的结果。

| ship_express |
|---|
| 6 |

### *COUNT DISTINCT 函数的示例*

在下一示例中，用户想要知道在 orders 表的 ship_instruct 列中有多少个唯一的非 NULL 值。用户在 SELECT 语句的 projection 列表中调用 COUNT DISTINCT 函数：

```
SELECT COUNT(DISTINCT ship_instruct) AS unique_notnulls
FROM orders;
```

下列表格展示此查询的结果。

| unique_notnulls |
|---|

| 16 |
|---|

### *COUNT column 函数的示例*

在下列示例中，用户想要知道在 orders 表的 ship_instruct 列中有多少非 NULL 值。该用户在 SELECT 语句的 Projection 列表中调用 COUNT(*column*) 函数：

```
SELECT COUNT(ship_instruct) AS total_notnulls FROM orders;
```

下列表格展示此查询的结果。

| total_notnulls |
|---|
| 21 |

对于 ship_instruct 列中非 NULL 值的一个类似的查询可在跟在 COUNT 关键字之后的圆括号中包括 ALL 关键字：

```
SELECT COUNT(ALL ship_instruct) AS all_notnulls FROM orders;
```

下列表格展示该查询结果，不论您包括还是省略 ALL 关键字（因为缺省值为 ALL），查询结果都一样。

| all_notnulls |
|---|
| 21 |

## MAX 函数

**MAX** 函数返回指定的列或表达式中的最大值。

使用 DISTINCT 关键字不会更改结果。在下列示例中的查询找到在库的但已被订购的最贵的项：

```
SELECT MAX(unit_price) FROM stock
        WHERE NOT EXISTS (SELECT * FROM items
        WHERE stock.stock_num = items.stock_num AND
        stock.manu_code = items.manu_code);
```

忽略 NULL，除非该列中的每个值都是 NULL。如果每个列值都是 NULL，则 **MAX** 函数为那个列返回 NULL。

## MIN 函数

MIN 函数返回列或表达式中的最低值。使用 DISTINCT 关键字不会更改结果。下列示例找到 stock 表中最廉价的项：

```
SELECT MIN(unit_price) FROM stock;
```

忽略 NULL 值，除非该列中的每个值都是 NULL。如果每个列值都是 NULL。则 **MIN** 函数为那列返回 NULL。

## SUM 函数

SUM 返回指定的列或表达式中所有值的总和，如下例所示：

SELECT SUM(total_price) FROM items WHERE order_num = 1013;

如果您包括 DISTINCT 或 UNIQUE 关键字，则返回的值仅对于该列或表达式中的 distinct 值：

SELECT SUM(DISTINCT total_price) FROM items WHERE order_num = 1013;

忽略 NULL 值，除非该列中的每个值都是 NULL。如果每个列值都是 NULL，则 **SUM** 为那列返回 NULL。您不可使用带有非数值列的 **SUM** 函数。

## RANGE 函数

**RANGE** 函数返回数值列表达式参数的值的范围。

它计算最大值与最小值之间的差异，如下所示：

range(expr) = max(expr) - min(expr);

您仅可对数值列应用 RANGE 函数。下列查询找到人口的年龄范围：

SELECT RANGE(age) FROM u_pop;

与其他聚集一样，当查询包括 GROUP BY 子句时，RANGE 函数应用于组中的行，如下列示例所示：

SELECT RANGE(age) FROM u_pop GROUP BY birth;

由于将 DATE 值在内部存储为整数，因此，您可对 DATE 列使用 **RANGE** 函数。对于 DATE 列，返回值是该列中最早日期与最晚日期之间的天数。

忽略 NULL 值，除非列中的每个值都是 NULL。如果每个列值都是 NULL，则 RANGE 函数为那列返回 NULL。

**重要：** 以 32 位数字精度执行 RANGE 函数的所有计算，对于许多输入数据集，这都应足够。然而，当所有输入数据值都有 16 位数字或更高的精度时，该计算会丢失精度或返回不正确的结果。

## VARIANCE 函数

**VARIANCE** 函数返回总体方差的估计值，即标准差的平方。

VARIANCE 计算下列值：

(SUM(Xi2) - (SUM(Xi)2)/N)/(N - 1)

在此公式中，

- Xi 是该列中的每一值，
- N 是该列中非 NULL 值的总数目（除非所有值都是 NULL，在此情况下，逻辑上未定义方差，且 **VARIANCE** 函数返回 NULL）。

您仅可对数值列应用 **VARIANCE** 函数。

下列查询估算人口的 age 值的方差：

SELECT VARIANCE(age) FROM u_pop WHERE u_pop.age > 0;

同其他聚集一样，当查询包括 GROUP BY 子句时，对组的行应用 VARIANCE 函数，如此例所示：

SELECT VARIANCE(age) FROM u_pop GROUP BY birth

```
WHERE VARIANCE(age) > 0;
```

如前面指出的那样，**VARIANCE** 忽略 NULL 值，除非对于指定的列每个限定的行都是 NULL。如果每个值都是 NULL，则 **VARIANCE** 为那列返回 NULL 结果。（这通常表示丢失数据，且不可避免地不是潜在的总体方差的一个好的估计。）

如果限定的非 NULL 列值的总数目 N 等于 1，则 VARIANCE 函数返回零（真实总体方差的另一不可信的估算）。要忽略此特殊情况，您可修改查询。例如，您可以包括 HAVING COUNT(*) > 1 子句。

**重要：** 以 32 位数字精度执行 VARIANCE 函数的所有计算，对于许多输入数据集，这应足够了。然而，当所有输入数据值都有 16 位数字或更高的精度时，该计算会丢失精度或返回不正确的结果。

虽然在内部将 DATE 数据存储为整数，但您不可在 DATE 数据类型的列上使用 **VARIANCE** 函数。

## ESQL/C 中的错误检查

聚集函数往往就返回一行。如果未选择行，则函数返回 NULL。您可使用 **COUNT (*)** 函数来确定是否选择了任何行，且您可使用指示符变量来确定是否某些选择了的行为空。以与聚集函数相关联的游标取回行，往往返回一行；因此，对于首次 FETCH 尝试，表示数据结束的 100 从不会返回到 **sqlcode** 变量内。

您还可使用 GET DIAGNOSTICS 语句进行错误检查。

## 聚集函数行为的总结

一个示例可帮助总结聚集函数的行为。假设 testtable 有单个名为 num 的 INTEGER 列。此表的内容如下。

| num |
|---|
| 2 |
| 2 |
| 2 |
| 3 |
| 3 |
| 4 |
| (NULL) |

您可使用聚集函数来获取关于 num 列和 testtable 表的信息。下列查询使用 AVG 函数来获取 num 列中所有非 NULL 值的平均值：

```
SELECT AVG(num) AS average_number FROM testtable;
```

下列表格展示此查询的结果。

| average_number |
| --- |
| 2.66666666666667 |

您可使用类似于前面示例的 SELECT 语句中的其他聚集函数。 如果您输入一系列在 projection 列表中有不同的聚集函数的 SELECT 语句，且不包括 WHERE 子句，则您收到下列表格展示的结果。

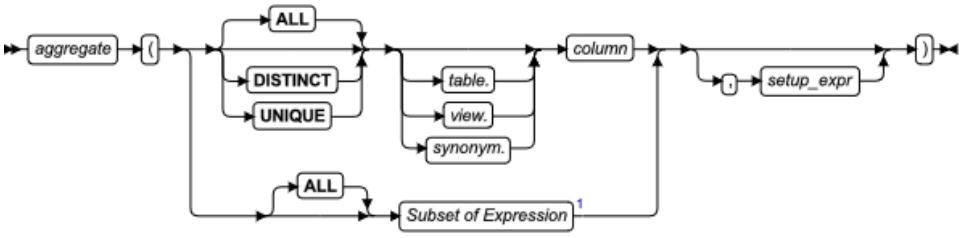
| 函数 | 结果 | 函数 | 结果 |
| --- | --- | --- | --- |
| COUNT (*) | 7 | MAX | 4 |
| COUNT (DISTINCT) | 3 | MAX(DISTINCT) | 4 |
| COUNT (ALL *num*) | 6 | MIN | 2 |
| COUNT ( *num* ) | 6 | MIN(DISTINCT) | 2 |
| AVG | 2.66666666666667 | RANGE | 2 |
| AVG (DISTINCT) | 3.00000000000000 | SUM | 16 |
| STDDEV | 0.74535599249993 | SUM(DISTINCT) | 9 |
| VARIANCE | 0.55555555555556 | | |

## 用户定义的聚集

您可以 CREATE AGGREGATE 语句创建您自己的聚集表达式，然后在您可调用内建的聚集的任何地方调用这些聚集。

下图展示调用用户定义的聚集的语法。

用户定义的聚集




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *aggregate* | 要调用的用户定义的聚集的名称 | ***聚集***以及为***聚集***定义的支持函数必须存在 | 标识符 |
| *column* | *table* 之内列的名称 | 必须存在且有数值数据类型 | 引用字符串 |
| *setup_expr* | 为特定的调用定制 | 不可为孤立的主变量。在 | 表达式 |

| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| | *聚集*的设置表达式 | *setup_expr* 中引用的任何列都必须在该查询的 GROUP BY 子句中 | |
| *synonym*、*table*、*view* | *column* 在其中发生的同义词、表或视图 | *synonym* 和它指向的表或视图必须存在 | 标识符 |

使用 DISTINCT 或 UNIQUE 关键字来指定仅将用户定义的聚集应用于命名的列或表达式中唯一的值。使用 ALL 关键字来指定将该聚集应用于命名的列或表达式中所有的值。

如果您省略 DISTINCT、UNIQUE 和 ALL 关键字，则缺省值为 ALL。要获取关于 DISTINCT、UNIQUE 和 ALL 关键字的更多信息，请参阅 包括或排除结果集中的重复值。

当您指定设置表达式时，将此值传递到 **INIT** 支持函数，在 CREATE AGGREGATE 语句中为用户定义的聚集定义了该支持函数。

在下列示例中，您将名为 **my_avg** 的用户定义的聚集应用到 **items** 表中 **quantity** 列的所有值：

```
SELECT my_avg(quantity) FROM items
```

在下列示例中，您将名为 **my_sum** 的用户定义的聚集应用于 **items** 表中 **quantity** 列的唯一的值。您还支持值 5 作为设置表达式。此值可以指定 **my_avg** 将计算的总和的初始值为 5。

```
SELECT my_sum(DISTINCT quantity, 5) FROM items
```

在下列示例中，您将名为 **my_max** 的用户定义的聚集应用于远程 **items** 表中 **quantity** 列的所有值：

```
SELECT my_max(remote.quantity) FROM rdb@rserv:items remote
```

如果将 **my_max** 聚集定义为 EXECUTEANYWHERE，则可将该分布式查询推送到远程数据库服务器 **rserv** 来执行。如果未将 **my_max** 聚集定义为 EXECUTEANYWHERE，则该分布式查询扫描远程 **items** 表，并在本地数据库服务器上计算 **my_max** 聚集。

您不可以远程数据库服务器的名称来限定用户定义的聚集，如下例所示。在此情况下，数据库服务器返回错误：

```
SELECT rdb@rserv:my_max(remote.quantity)
        FROM rdb@rserv:items remote
```

要获取关于用户定义的聚集的更多信息，请参阅 CREATE AGGREGATE 语句 以及 *GBase 8s 用户定义的例程和数据类型开发者指南* 中对用户定义的聚集的讨论。

## 网格查询中的聚集表达式

网格查询中的聚集表达式要求发出网格查询的数据库服务器从多网格服务器组合聚集结果。

网格查询是在数据库服务器的限定的行上返回逻辑 UNION 或 UNION ALL 的分布式 SELECT 语句，这些表在 GBase 8s 网格的数据库中有相同的模式。在 GRID 子句 主题和在 *GBase 8s Enterprise Replication 指南* 中描述网格查询。

网格查询中的聚集表达式要求发出该网格查询的数据库服务器组合来自多网格服务器的聚集结果。由于您指定的网格查询转换为跨多网格服务器的 UNION 或 UNION ALL 查询，因此，在指定的网格或区域内在每一参与的服务器上独立地计算每一聚集。将这些单独的聚集返回到发出该网格查询的数据库服务器，其计算它们的组合的 UNION 或 UNION ALL 值。

为了从这些单独的聚集计算全局的聚集，该网格查询必须是子查询。例如，假设我们想要查看东南地区（下列示例中的 **SW_USA**）的总销售额和平均销售额。因为如果在该网格内跨数据集的限定的行值的数目不同，则采用平均组的平均值是不正确的，因此正确的网格查询需要类似这样：

```
SELECT SUM(amt) AS total_sales ,
SUM(amt) / SUM(cnt) AS avg_sale FROM
(
   SELECT COUNT(*) cnt, SUM(amt) amt
   FROM sales GRID ALL 'SW_USA'
);

total_sales            avg_sale
$8300.00               $103.75
```

在此示例中，网格查询的 projection 列表指定来自东南地区（编码为 `'SW_USA'`）内每一网格数据库的 **sales** 表的 COUNT(*) 表达式（别名为 `cnt`）和聚集数量 `SUM(amt)`（别名为 `amt`）。使用来自每一远程网格服务器的这些值，本地的网格服务器可计算该地区的平均值。通过把求值表达式 `SUM(amt)` 的总销售额加起来，并将那个值除以总计数 `SUM(cnt)` 来实现，此处，**cnt** 和 **amt** 是网格子查询的结果集中的列。

请注意，GRID ALL 关键字指定来自每一参与的网格服务器的限定的行的 UNION ALL，来避免消除重复的行。

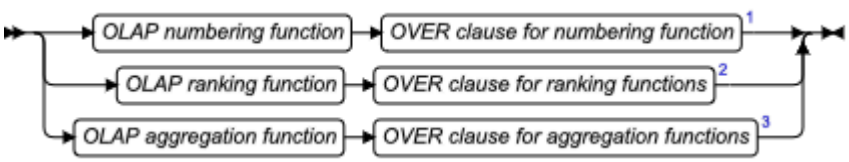
## 4.8 OLAP window 表达式

您可在 SELECT 语句中包括“联机分析处理”（OLAP）表达式来在查询或子查询的结果集中的行的子集上操作。对于数据中的模式、趋势和例外，您可使用 OLAP window 表达式来检测满足条件的行的子集。

OLAP window 表达式允许应用开发人员更简单地和高效地构成分析业务查询。例如，可在各种不同的间隔之上计算移动平均值和移动总和。可随着选择了的列值更改，重置聚集和等级。可以简单的术语表示复杂的比率。

**语法**

OLAP window 表达式




在 SELECT 语句的这些子句中，OLAP window 表达式是有效的：

- Projection 子句的 Select 列表
- SELECT 语句的 ORDER BY 子句
- Projection 子句中的子查询规范

OLAP window 表达式需要 OLAP window 函数和 OVER 子句。

OLAP window 函数

OLAP window 函数为数据库服务器表示一个要在查询结果集中的行上执行的操作。OLAP window 函数分为下列函数类别：

- OLAP 编号函数将唯一的行编号指定到每一行：
  - ROW_NUMBER 和 ROWNUMBER，其为同义词
- OLAP 分等级函数为每一行指定等级：
  - LAG
  - LEAD
  - RANK
  - DENSE_RANK
  - PERCENT_RANK
  - CUME_DIST
  - NTILE
- OLAP 聚集函数聚集行数据：
  - FIRST_VALUE
  - LAST_VALUE
  - RATIO_TO_REPORT
  - AVG
  - COUNT
  - MAX
  - MIN
  - RANGE
  - STDDEV
  - SUM
  - VARIANCE

OVER 子句

OVER 子句定义在其上执行指定的操作的结果集。OVER 子句由下列功能：

- 以 PARTITION BY 子句定义 window 分区。OLAP window 分区是查询结果集中行的子集，基于罗列在 PARTITION BY 子句中的一个或多个列表达式的值。
- 以 ORDER BY 子句将行排序。如果您包括 PARTITION BY 子句，则在每一 window 分区内对结果排序。否则，对整个结果排序。
- 以 OLAP window 聚集函数的 ROWS 或 RANGE 规范定义 window 框架。window 框架定义 window 分区内的一组行。聚集函数在移动的 window 框架的内容上操作，而不是在整个分区上操作。

例如，下列查询包含 OLAP 聚集函数 SUM：

```
SELECT c,d,
        SUM(d) OVER(
        PARTITION BY a,b
        ORDER BY c,d
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        FROM table1;
```

PARTITION BY 子句为每一组列 **a** 和 **b** 的值创建 window 分区。

ORDER BY 子句按照列 **c** 和 **d** 的值对每一 window 分区内的数据排序。

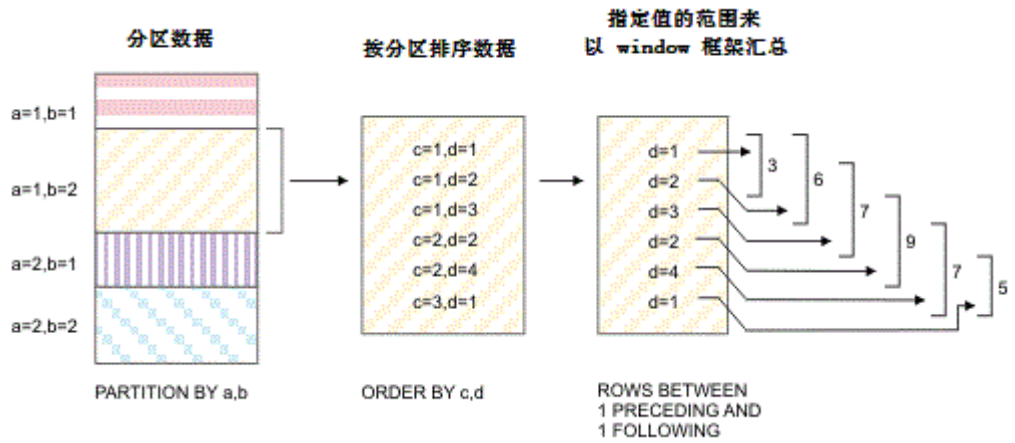
以 ROWS 关键字开启的 window 框架子句创建由三行组成的 window 框架：当前行、当前行的前一行，以及当前行的后一行。当前行是 window 框架的引用点。下列表格展示当每一行轮流成为当前行时，window 框架如何在结果集中移动。

表 1. 结果集中的 window 框架.

| 行编号 | 第一个框架 | 第二个框架 | 第三个框架 | 第四个框架 | 第五个框架 | 第六个框架 |
|---|---|---|---|---|---|---|
| 1 | 当前的行 | 当前的行 - 1 | | | | |
| 2 | 当前的行 + 1 | 当前的行 | 当前的行 - 1 | | | |
| 3 | | 当前的行 + 1 | 当前的行 | 当前的行 - 1 | | |
| 4 | | | 当前的行 + 1 | 当前的行 | 当前的行 - 1 | |
| 5 | | | | 当前的行 + 1 | 当前的行 | 当前的行 - 1 |
| 6 | | | | | 当前的行 + 1 | 当前的行 |

SUM 函数为在该查询的作用域中的每一 window 分区加上三行的 **d** 列的值。

下列图示展示该查询如何对数据分区、排序和聚集。




该查询返回单个的 window 分区的 **c** 和 **d** 列的值以及列 **d** 的三个值的总和：

| c | d | (sum) |
|---|---|---|
| 1 | 1 | 3 |
| 1 | 2 | 6 |
| 1 | 3 | 7 |
| 2 | 2 | 9 |
| 2 | 4 | 7 |
| 3 | 1 | 5 |

**sum** 列中的第一个值是 **d** 列中第一个和第二个值的总和 (1 + 2 = 3)。对于第一个值，不存在当前行的前一个值。

**sum** 列中的第二个值是 **d** 列中第一个、第二个和第三个值的总和 (1 + 2 + 3 = 6)。

**sum** 列中的第三个值是 **d** 列中第二个、第三个和第四个值的总和 (2 + 3 + 2 = 7)。

**sum** 列中的最后一个值是 **d** 列中第五个和第六个值的总和 (1 + 4 = 5)。对于最后的值，不存在当前行的下一个值。

## OLAP 编号函数表达式

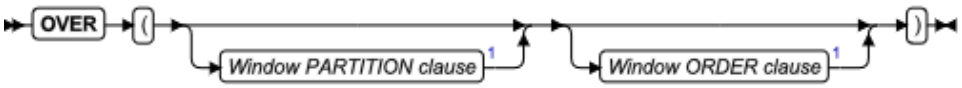
OLAP 编号函数表达式为单个查询的结果集中的每一行返回一序列的编号。

OLAP 编号函数表达式是您可在 SELECT 语句的 Projection 列表中包括，或在 SELECT 语句的 ORDER BY 子句中包括的 OLAP window 表达式。

### 语法

OLAP OLAP window


编号函数的 OVER 子句




用法

对于同一函数，关键字 ROW_NUMBER 与 ROWNUMBER 是同义词。此编号函数就像是一个不要求 window ORDER 子句且不检查重复值的简化的 RANK 函数。ROW_NUMBER 函数往往为每一 OLAP window 分区中的每一行返回唯一的值。

ROW_NUMBER 函数不带参数，但您必须在 ROW_NUMBER（或 ROWNUMBER）关键字之后包括空的圆括号。

ROW_NUMBER 函数为每一 OLAP 分区中的每行返回一无符号整数。每一分区中的行编号的序列起始于 1，且每一后续的行增 1，不论 window 分区中连续的行是否有相同的或不同的列值。

如果未指定 window PARTITION 子句，则完整的结果集从 1 至 *n* 编号，此处，*n* 是查询或子查询返回的满足条件的行的数目。

OVER 子句为编号函数定义的 OLAP window 有此语法：

- window PARTITION 子句是可选的。如果未指定，则编号函数的作用域为查询或子查询的整个结果集，而不是分区了的子集。
- window ORDER 子句是可选的。如果未指定，则返回的行编号是基于在查询处理时这些行的缺省顺序。如果指定 window ORDER 子句，则 ORDER BY 规范决定行编号的分配。
- 对于 OLAP 编号函数，不支持 window Frame 子句。

如果为 ROW_NUMBER 函数定义 OLAP window 的 OVER 子句省略 window PARTITION 子句和 window ORDER 子句，则您必须在 OVER 关键字之后包括空的圆括号。

**示例：ROW_NUMBER 函数**

下列查询通过不同的包类型（pkg_type）对产品表中的行进行分区，并为每一分区指定从 1 重新开始的行编号。

```
SELECT ROW_NUMBER()
        OVER(PARTITION BY pkg_type ORDER BY prod_name)
        AS rownum, prod_name, pkg_type
        FROM product;

ROWNUM   PROD_NAME              PKG_TYPE
     1   Aroma Sounds CD         Aroma designer box
     2   Aroma Sounds Cassette   Aroma designer box
     1   Christmas Sampler       Gift box
     2   Coffee Sampler          Gift box
     3   Easter Sampler Basket   Gift box
     4   Spice Sampler           Gift box
     5   Tea Sampler             Gift box
     1   Aroma Roma              No pkg
```

```
2    Aroma baseball cap          No pkg
3    Aroma t-shirt               No pkg
4    Assam Gold Blend            No pkg
5    Assam Grade A               No pkg
6    Breakfast Blend             No pkg
7    Cafe Au Lait                No pkg
8    Coffee Mug                  No pkg
9    Colombiano                  No pkg
10   Darjeeling Number 1         No pkg
11   Darjeeling Special          No pkg
12   Demitasse Ms                No pkg
13   Earl Grey                   No pkg
...
```

由于 window ORDER 子句指定 prod_name 列作为排序键，因此，在缺省的（ASC）顺序中，返回的行编号是基于产品名称的字母顺序。例如，在 "No pkg" 分区内，将三个 "Aroma" 产品编号为 1、2 和 3。（在缺省的 ASCII 顺序规则中大写字母排序在小写之上。）

## OLAP 分等级函数表达式

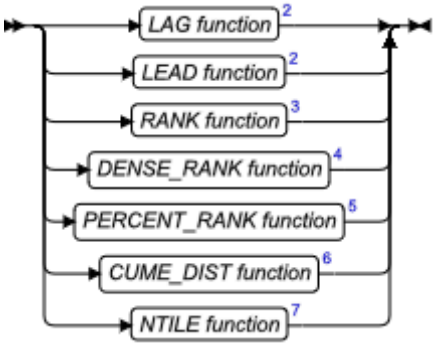
您可包括 OLAP 分等级函数表达式来计算可应用于查询或子查询的分区了的结果集中每一行的顺序等级。

OLAP 分等级函数表达式是您可包括在 SELECT 语句的 Projection 列表中，或 SELECT 语句的 ORDER BY 子句中的 OLAP window 表达式。

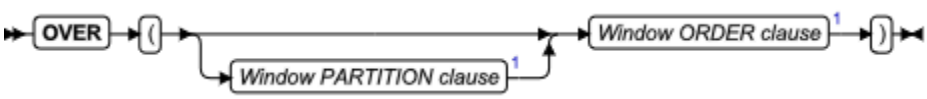
### 语法

OLAP 分等级函数表达式


OLAP 分等级函数




分等级函数的 OVER 子句



### 用法

这些函数返回的分等级的值依赖于 OVER 子句内的 window ORDER 子句。ORDER 子句定义数据库服务器用来计算分等级的值的列或表达式。

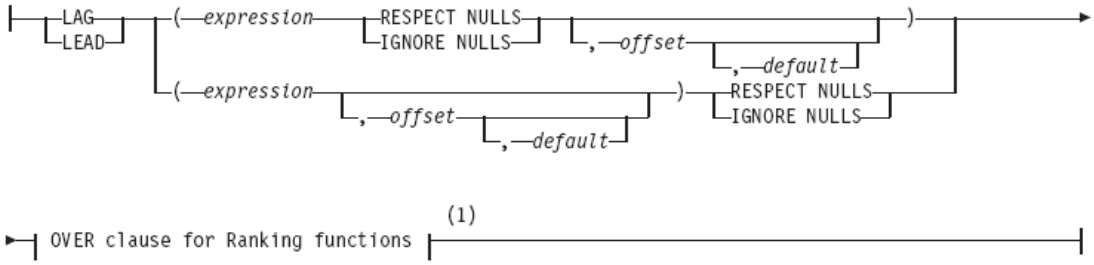
如果您省略 window PARTITION 子句，则分等级函数的作用域是查询或子查询的整个结果集，而不是结果的分区了的子集。

## LAG 和 LEAD 函数

LAG 和 LEAD 函数是 OLAP 分等级函数，在从当前 window 分区内的当前行的指定偏移量上，为该行返回它们的 *expression* 参数的值。

### 语法

LAG 和 LEAD 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *default* | 如果 *offset* 超出当前的 window 分区，则返回的值 | 如果未指定 *default*，则对于任何作用域之外的行返回 NULL 值。 | 列表达式 |
| *expression* | 为从当前行的 *offset* 行的行返回的列名称、别名或常量表达式 | 如果 *expression* 引用一列，则该列必须也在 Projection 子句的选择列表中。 | 列表达式 |
| *offset* | 定义从当前行的该位置的偏移量的非负整数常量 | 需要 window PARTITION 子句。如果为零，则指定当前的行。如果未指定 *offset*，则使用值 1。 | 文字整数 |

### 用法

需要 *expression* 参数。从 LAG 或 LEAD 函数返回值的数据类型为该表达式的数据类型。

基于 window ORDER 子句对每一 window 分区执行的排序顺序，LEAD 和 LAG 函数返回在从当前行的 *offset* 行的每行的表达式的值：

- 对于 LAG 函数，*offset* 表明在当前行前面 *offset* 行的那一行。
- 对于 LEAD 函数，*offset* 表明当前行后面 *offset* 行的那一行。

如果 OVER 子句未包括 window PARTITION 子句，则这些函数为查询的整个结果集返回 *expression* 值。

如果指定 window PARTITION 子句，则 LAG 函数的第二个参数（*offset*）意味着当前行之前的 *offset* 行，且在当前分区内。对于 LEAD 函数，第二个参数意味着当前行之后的 *offset* 行，且在当前分区内。

对于两个函数，如果未指定 *offset*，则使用值 1。如果指定可选的第三个参数（*default*），其可为表达式，则如果 *offset* 超出当前分区的作用域，则返回它的值。否则，返回 NULL 值。当指定第三个参数时，也必须指定第二个参数。

**处理 NULL 值**

在 LAG 或 LEAD 函数表达式中，可以在两个位置之一中指定可选的 RESPECT NULLS 或 IGNORE NULLS：

- 在参数列表中
- 紧跟在定界参数列表的收圆括号之后

然而，如果您在同一 LAG 或 LEAD 表达式内在这两个位置都包括 RESPECT NULLS 或 IGNORE NULLS，则数据库服务器发出例外。

RESPECT NULLS 和 IGNORE NULLS 关键字有这些作用：

- 如果您指定 RESPECT NULLS 关键字，则当计数到 *offset* 行时，包括其 *expression* 求值为 NULL 的行。
- 如果您指定 IGNORE NULLS 关键字，则当计数到 *offset* 行时，不包括其 *expression* 求值为 NULL 的任何行。

如果您指定 IGNORE NULLS，且该 window 分区中行的所有 expression 值都是 NULL，则 LAG 或 LEAD 函数为每一行都返回 default 值。如果未指定 default 参数，则该函数返回 NULL 值。

**示例：LEAD 和 LAG 函数**

在下列查询中，LAG 函数和 LEAD 函数分别定义了按照部门将员工分区的 OLAP window，并罗列它们的薪资。LAG 函数展示与同一部门中享有下一更低薪资值的员工相比，每一员工收到多少更多的补偿。LEAD 函数指示与同一部门中享有下一更高薪资值的员工相比，每一员工收到的少多少。

```
SELECT name, salary,  LAG(salary)
        OVER (PARTITION BY dept ORDER BY salary),
        LEAD(salary, 1, 0)
        OVER (PARTITION BY dept ORDER BY salary)
        FROM employee;
```

| name | salary | (lag) | (lead) |
|---|---|---|---|
| John | 35,000 | | 38,400 |
| Jack | 38,400 | 35,000 | 41,200 |

| | | | |
|---|---|---|---|
| Julie | 41,200 | 38,400 | 45,600 |
| Manny | 45,600 | 41,200 | 47,300 |
| Nancy | 47,300 | 45,600 | 49,500 |
| Pat | 49,500 | 47,300 | 51,300 |
| Ray | 51,300 | 49,500 | 0 |

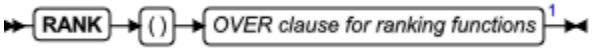
对于 LAG 函数，名称为 John 的第一行有一 NULL 值，因为未指定缺省值。对于 LEAD 函数，名为 Ray 的最后一行有一 0 值，因为在该 LEAD 函数中指定 0 作为第三个参数的缺省值。

## RANK 函数

RANK 函数是一个 OLAP 分等级函数，为 OLAP window 中的每一行计算分等级的值。返回值是一个顺序编号，其基于 OVER 子句中所需要的 ORDER BY 表达式。

### 语法

RANK 函数




### 用法

行的等级定义为 1 加上排在该行的等级前面的行数。如果两行或多行有相同的值，则这些行也得到相同的等级。结果在连续的分了等级的值的序列中可有间隔。例如，如果将两行分为等级 1，则下一等级为 3。对于包括非唯一的值的分等级的行，DENSE_RANK 函数使用不同的规则。

RANK 函数没有参数，但必须指定空的圆括号。如果 OVER 子句指定可选的 window PARTITION 子句，则在每一分区定义的行的子集内计算等级。

### 示例：RANK 函数

下列查询按照销售人员的销售量对他们分等级。这些等级不是连续的，因为销售量相同的人员都指定相同的等级值，且跳过下一等级值。

```
SELECT emp_num, sales,
        RANK() OVER (ORDER BY sales) AS rank
        FROM sales;
```

| emp_num | sales | rank |
|---|---|---|
| 101 | 2,000 | 1 |
| 102 | 2,400 | 2 |
| 103 | 2,400 | 2 |
| 104 | 2,500 | 4 |
| 105 | 2,500 | 4 |
| 106 | 2,650 | 6 |

# DENSE_RANK 函数

DENSE_RANK 函数是一个 OLAP 分等级函数，为 OLAP window 中的每一行计算等级值。返回值是一个顺序编号，其基于在 OVER 子句中所需要的 ORDER BY 表达式。

**语法**

DENSE_RANK 函数


**用法**

将一行的等级定义为 1 加上该行的等级前面的等级数目。如果两行或多行有相同的值，则这些行得到相同的等级。然而，与 RANK 函数相反，如果两行或多行同级，则在分等级的值的序列中没有间隔。例如，如果两行都分等级为 1，则下一等级仍为 2。

此函数没有参数，但必须指定空的圆括号。如果 OVER 子句指定可选的 window PARTITION 子句，则在每一 window 分区定义的行的子集之内计算 DENSE_RANK 等级。

**示例：DENSE_RANK 函数**

下列查询按照销售人员的销售量对他们分等级。即使多个销售量有相同的等级，等级也是连续的。

```
SELECT emp_num, sales,
   DENSE_RANK() OVER (ORDER BY sales) AS dense_rank,
   FROM sales;
```

| emp_num | sales | dense_rank |
|---|---|---|
| 101 | 2,000 | 1 |
| 102 | 2,400 | 2 |
| 103 | 2,400 | 2 |
| 104 | 2,500 | 3 |
| 105 | 2,500 | 3 |
| 106 | 2,650 | 4 |

# PERCENT_RANK 函数

PERCENT_RANK 函数是一个 OLAP 分等级函数，为 OLAP window 中的每一行计算等级值，规格化为从 0 至 1 的范围。

计算每一 PERCENT_RANK 值为该行的 RANK 减去 1，除以该分区内行的数目减去 1。通常，越接近于 1 的值表示等级越高，而越接近 0 的值通常表示越低的等级。

**语法**

PERCENT_RANK 函数


**用法**

此函数不用参数，但必须指定空的圆括号。如果还指定可选的 window PARTITION 子句，则在每一分区定义的行的子集之内计算等级。如果分区中有单个行，则它的 PERCENT_RANK 值为 0。

**示例：PERCENT_RANK 函数**

下列查询按照销售人员的销售量将他们分等级。

```
SELECT emp_num, sales,
        PERCENT_RANK() OVER (ORDER BY sales) AS per_rank
        FROM sales;

        emp_num         sales    per_rank

        101         2,000            0
        102         2,400          0.2
        103         2,400          0.2
        104         2,500          0.6
        105         2,500          0.6
        106         2,650          1.0
```

## CUME_DIST 函数

CUME_DIST 函数是一个 OLAP 分等级函数，计算累计分布作为每一行的百分比等级。该等级表示为取值范围从 0 至 1 的实际值小数。

**语法**

CUME_DIST 函数


用法

CUME_DIST 函数计算等级低于或等于当前的行，包括当前行的行的数目，除以该分区中行的总数。越接近 1 的值表示越高的等级，而越接近 0 的值表示越低的等级。

此函数不用参数，但必须指定空的圆括号。如果还指定可选的 window PARTITION 子句，则在每一分区定义的行的子集之内计算等级。如果在该分区中有单个行，则它的 CUME_DIST 值为 1。

**示例：CUME_DIST 函数**

下列查询展示每个销售人员的销售量的累计分布。

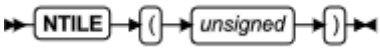
```
SELECT emp_num, sales,
        CUME_DIST() OVER (ORDER BY sales) AS cume_dist
        FROM sales;

        emp_num         sales       cume_dist

        101         2,000       0.166666667
        102         2,400       0.500000000
        103         2,400       0.500000000
        104         2,500       0.833333333
        105         2,500       0.833333333
        106         2,650       1.000000000
```

## NTILE 函数

NTILE 函数是一个 OLAP 分等级函数，将每一分区中的行划分成 N 等级的类别，称为片，每一类别包括大约相等的行数。

**语法**

NTILE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *unsigned* | 指定要分等级为多少个类别，或片的无符号整数 | 不可为零 | 文字整数 |

**用法**

由该函数的无符号整数参数设置分等级的类别，或片的数量，且在 OVER 子句的 ORDER BY 表达式之上。

例如，对于查询结果集中的所有行的 dollars 列中的值，下列分等级函数表达式返回从 1 至 100 的百分比的等级：

NTILE(100) OVER(ORDER BY dollars)

当该参数为 4 时，返回的值将 OVER 子句定义的每一分区中的行排序成四个等分。当一组值不能由指定的整数参数分开时，NTILE 函数将剩余的行放置在低等级的片中。

**示例：NTILE 函数**

下列查询按照员工薪酬将部门中的员工分等级，并对于每一部门计算 1 至 5 的片数。

```
SELECT name, salary,
 NTILE(5) OVER (PARTITION BY dept ORDER BY salary)
```

FROM employee;

| name | salary | (ntile) |
|---|---|---|
| John | 35,000 | 1 |
| Jack | 38,400 | 1 |
| Julie | 41,200 | 2 |
| Manny | 45,600 | 2 |
| Nancy | 47,300 | 3 |
| Pat | 49,500 | 4 |
| Ray | 51,300 | 5 |

从最低到最高对薪酬排序，因为 ORDER BY 子句的缺省的排序方向为升序。如果您在 ORDER BY 子句中包括 DESC 关键字，则从最高到最低对薪酬排序。

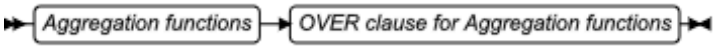
## OLAP 聚集函数表达式

OLAP 聚集函数表达式可返回关于查询的分区了的结果中行的聚集信息。

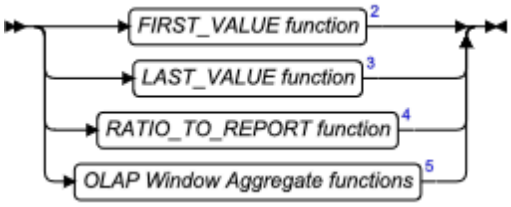
OLAP 聚集函数表达式是 OLAP window 表达式，您可在 SELECT 语句的 Projection 列表中，或在 SELECT 语句的 ORDER BY 子句中包括它。

**语法**

OLAP 聚集函数表达式




OLAP 聚集函数




聚集函数的 OVER 子句


**用法**

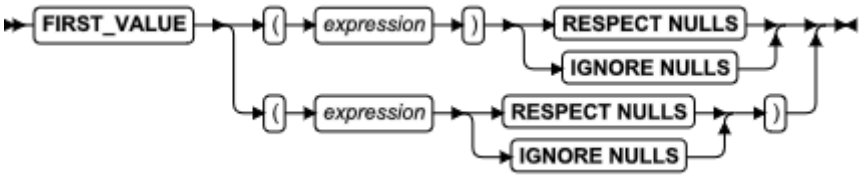
您可通过指定 window 框架将 OLAP 聚集函数应用于 OLAP window 分区中行的子集。

## FIRST_VALUE 函数

FIRST_VALUE window 聚集函数为每一 OLAP window 分区中的第一行返回指定的表达式的值。

**语法**

FIRST_VALUE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 列名称、别名或常量表达式 | 如果 *expression* 引用一列，则该列必须也在该 Projection 子句的选择列表中 | 列表达式 |

**用法**

FIRST_VALUE 函数返回的数据类型是指定的表达式的数据类型。结果可为 NULL。如果指定 IGNORE NULLS，则在计算中不考虑行的表达式值求值为 NULL 的所有行。如果指定 IGNORE NULLS，且在该 OLAP window 中的所有值都是 NULL，则 FIRST_VALUE 函数返回 NULL 值。

或可在紧跟在表达式之后的圆括号内，或可在圆括号的外部指定 RESPECT NULLS 或 IGNORE NULLS 选项，但仅允许一个这样的规范。

**示例：FIRST_VALUE 函数**

下列语句返回 window 分区中按天排的存货价格，以及与第一个值 18.25 的存货价格的差异。

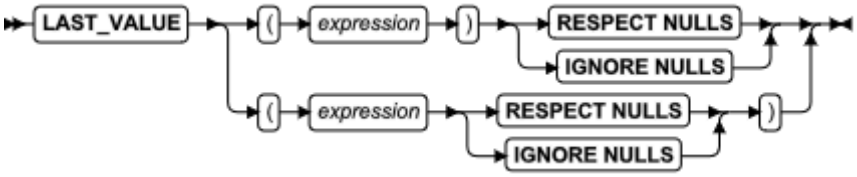
```
SELECT price, price - FIRST_VALUE(price)
        OVER (PARTITION BY year ORDER BY tradingday)
        AS diff_price
        FROM stock_price
        WHERE tradingday between '2012-11-01' and '2012'-11-07';

        price           diff_price
        18.25                0
        18.37              0.12
        19.03              0.78
        18.59              0.34
        18.21             -0.04
```

# LAST_VALUE 函数

LAST_VALUE window 聚集函数为每一 OLAP window 分区中的最后一行返回指定的表达式的值。

**语法**

LAST_VALUE 函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 列名称、别名或常量表达式 | 如果 *expression* 应用一列，则该列必须也在该 Projection 子句的选择列表中 | 列表达式 |

**用法**

LAST_VALUE 函数的返回数据类型是指定的表达式的数据类型。该结果可为 NULL。如果指定 IGNORE NULLS，则在计算中不考虑行的表达式值求值为 NULL 的所有行。如果指定 IGNORE NULLS 且 OLAP window 中的所有值都是 NULL，则 LAST_VALUE 函数返回 NULL 值。

或可在紧跟在该表达式之后的圆括号之内，或可在圆括号外部设置 RESPECT NULLS 或 IGNORE NULLS 选项，但仅允许一个这样的规范。

## RATIO_TO_REPORT 函数

RATIO_TO_REPORT 基于该函数的数值参数，计算每一行对于该 window 分区中剩余行的分数比率。

**语法**

RATIO_TO_REPORT 函数


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *expression* | 列名称、别名或常量表达式 | 该 *expression* 必须求值为数值的数据类型。DATE、DATETIME 和 INTERVAL 列不是有效的。如果 *expression* 引用一列，则该列也必须在该 Projection 子句的选择列表中。 | 列表达式 |

**用法**

此函数计算一行中指定的数值列的值对 OLAP window 框架中每一分区中所有行的值的总和的比率。名称 RATIOTOREPORT 是 RATIO_TO_REPORT 的关键字同义词。

同所有 OLAP window 聚集函数一样，Window PARTITION、Window ORDER 和 Window Frame 子句是可选的。可将该比率应用于分区了的行，或应用于整个查询结果集。如果还指定 window Frame 子句，则该比率应用于来自当前 window 框架的所有行。

所需要的 *expression* 参数必须指定数值数据类型的列，或为求值为数值数据类型的常量表达式。如果 *expression* 不是数值数据类型，则该函数失败并报错消息 -25862。

如果当前 window 分区中所有行的 *expression* 值的总和为零，则此函数为那个分区中的每一行返回 NULL 值。

如果诸如 SUM 或 MAX 这样的另一 OLAP 聚集函数有一空的 OVER 子句，或如果 OVER 子句仅包含单个 window PARTITION 规范，则对于该分区中的每行，该 OLAP 函数的结果都一样。然而，对于 RATIO_TO_REPORT，不是这种情况，因为对同一 window 分区中的不同行指定不同的比率，每一分区合计（大约）为 1。要将比率转换为百分率，将该函数表达式乘以 100，如下例所示。

**示例：RATIO_TO_REPORT 函数**

下列示例基于该查询返回的所有行，计算每一城市的销售额的十进制小数，作为每一城市展示销售额合计的单个报告。

```
SELECT city, SUM(dollars) AS SALES,
        RATIO_TO_REPORT(SUM(dollars)) OVER() *100 AS RATIO_DOLLARS
        FROM sales, store, period
        WHERE sales.store_id = store.store_id
        AND sales.period_id = period.period_id
        GROUP BY CITY
        ORDER BY SALES DESC;
```

| CITY | SALES | RATIO_DOLLARS |
|---|---|---|
| San Jose | 896931.15 | 12.58 |
| Atlanta | 514830.00 | 7.22 |
| Miami | 507022.35 | 7.11 |
| Los Angeles | 503493.10 | 7.06 |
| Phoenix | 437863.00 | 6.14 |
| New Orleans | 429637.75 | 6.03 |
| Cupertino | 424215.00 | 5.95 |
| Boston | 421205.75 | 5.91 |
| Houston | 417261.00 | 5.85 |
| New York | 397102.50 | 5.57 |
| Los Gatos | 394086.50 | 5.53 |
| Philadelphia | 392377.75 | 5.50 |
| Milwaukee | 389378.25 | 5.46 |
| Detroit | 305859.75 | 4.29 |
| Chicago | 294982.75 | 4.14 |
| Hartford | 236772.75 | 3.32 |
| Minneapolis | 165330.75 | 2.32 |

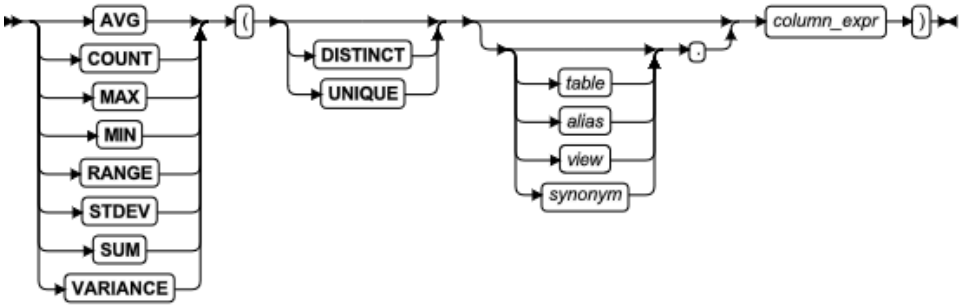
在上述示例中，RATIO_TO_REPORT 的 *expression* 参数是数值的聚集函数表达式 `SUM(dollars)`。输出的最后一行表明 **city** Minneapolis 的 **sales** 值大约是该查询报告的合计销售额的 2.32%。

# OLAP window 聚集函数

从查询的结果返回聚集结果的几个函数，诸如总和和平均值，还可用作来自 OLAP window 的上下文的 OLAP 函数。

**语法**

Window 聚集函数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column_expr* | 聚集函数的列表达式参数 | 请参阅下面的单独函数的标题 | 标识符 |
| *alias, synonym, table, view* | 包含 *column* 的同义词、表、视图或别名 | *Synonym* 以及它指向的 *table* 或 *view* 必须存在 | 标识符 |

**用法**

这些聚集函数不要求 OLAP window 从查询结果集计算聚集值。然而，在调用的上下文中它们的行为像 OLAP window 聚集函数一样，在此，函数表达式中的 OVER 子句定义一个或多个 window 分区，或包括 window ORDER 子句和 window Frame 子句。

**重要：** 当 DISTINCT 或 UNIQUE 关键字是 window 聚集函数规范的一部分时，window 聚集表达式的 OVER 子句不可包括 Window ORDER 子句或 Window Frame 子句。

下列聚集函数可返回关于 OLAP window 分区中的行的信息。

**AVG 函数**

对于在 OVER 子句中定义的每一分区中的行，AVG 函数返回在查询结果的 window 分区中指定的列或表达式中所有值的平均值。如果 OVER 子句包括 Window Frame 子句，则 AVG 为 window frame 中的每一组行返回一个值。

您仅可将 AVG 函数应用于数值数据类型的列。如果您使用 DISTINCT（或 UNIQUE）关键字，则仅从指定的列或表达式中 distinct 值计算平均值（含义为平均），且 OVER 子句不可包括 Window ORDER 或 Window Frame 子句。

忽略 NULL 值，除非该列或表达式中的每个值都是 NULL。如果每个值都是 NULL。则 AVG 函数为那个列或表达式返回 NULL。

您不可以非数值的列或表达式使用 AVG 函数。

**COUNT 函数**

对于在 OVER 子句中定义的每一分区，COUNT 函数返回查询结果的 window 分区中指定的列或表达式中非 NULL 值的基数。如果该 OVER 子句包括 Window Frame 子句，则 COUNT 为 window frame 中的每一组行返回一个值。

如果您使用 DISTINCT（或 UNIQUE）关键字，则仅从指定的列或表达式中 distinct 值计算该分区中行的基数，且 OVER 子句不可包括 Window ORDER 或 Window Frame 子句。

忽略 NULL 值，除非该列或表达式中的每个值都是 NULL。如果每个值都是 NULL，则 COUNT 函数为那个列或表达式返回 NULL。

**MAX 函数**

对于 OLAP window OVER 子句中定义的每一分区中的行，MAX 函数返回查询结果的 window 分区中列或表达式中的最大值。

指定 DISTINCT 或 UNIQUE 关键字，对结果没有影响，但（同其他内建的聚集函数一样）不允许 Window ORDER 或 Window Frame 子句。

如果 OVER 子句包括 Window Frame 子句，则 MAX 为 window frame 中每一组行返回一个值。

当指定列表达式作为 COUNT 的参数时，忽略 NULL 值，除非指定的列表达式中每个值都是 NULL。如果每个值都是 NULL，则 MAX 为那个列或表达式返回 NULL 值。当指定 COUNT(*) 时，如同其他值一样，对 NULL 值计数。

**MIN 函数**

对于 OLAP window OVER 子句定义的每一分区，MIN 函数返回查询结果的 window 分区中列或表达式中的最小值。如果 OVER 子句包括 Window Frame 子句，则 MIN 函数为每一组行返回一个值。

指定 DISTINCT 或 UNIQUE 关键字，对结果没有影响。但（同其他内建的聚集函数一样）不允许 Window ORDER 或 window Frame 子句。

如果 OVER 子句包括 Window Frame 子句，则 MIN 为 window frame 中的每一组行返回一个值。

忽略 NULL 值，除非指定列表达式中的每个值都是 NULL。如果每个值都是 NULL，则 MIN 为那个列表达式返回 NULL 值。

**RANGE 函数**

对于 OLAP window OVER 子句定义的每一分区，RANGE 函数返回查询结果的 window 分区中列或表达式中值的范围。如果 OVER 子句包括 Window Frame 子句，则 RANGE 为 window frame 中每一组行返回一个值。

RANGE 函数计算最大值与最小值之间的差，如下：

range(expr) = max(expr) - min(expr)

您仅可将 RANGE 函数应用于数值的列表达式。下列查询找到人口的年龄的范围：

```
SELECT RANGE(age) OVER () FROM u_pop;
```

由于 DATE 值在内部作为整数保存，因此您可对 DATE 列使用 RANGE 函数。使用 DATE 列，返回值是该列表达式中最早日期与最晚日期之间的天数。

忽略 NULL 值，除非列表达式中的每个值都是 NULL。如果每个列表达式值都是 NULL，则 RANGE 函数为那个列表达式返回 NULL。

**STDDEV 函数**

STDDEV 函数返回列或表达式的标准差，使用下列公式：

SQRT((SUM(Xi2) - (SUM(Xi)2)/N)/(N - 1))

在此公式中，Xi 是 OVER 子句指定的 window 分区或 frame 中每一列表达式值，N 是列表达式中非 NULL 值的总数。

如果 OVER 子句包括 Window Frame 子句，则 STDDEV 函数为 window frame 中每一组行返回一个值。

忽略 NULL 值，除非指定的列表达式中每个值都是 NULL。如果每个列表达式都是 NULL，则 STDDEV 函数为那个列表达式返回 NULL。

您仅可对数值的列表达式应用 STDDEV 函数。您不可对 DATE 类型的列表达式使用此函数。

**SUM 函数**

对于 OLAP window OVER 子句中定义的每一分区中的行，SUM 函数计算并返回查询结果的 window 分区中列表达式的所有值的总和。

如果您使用 DISTINCT（或 UNIQUE）关键字作为参数列表中的第一项，则该总和仅针对于列或表达式中的 distinct 值，且不允许 Window ORDER 或 Window Frame 子句。

忽略 NULL 值，除非每个值都是 NULL。如果每个值都是 NULL，则 SUM 函数为那个列或表达式返回 NULL 值。

您不可以非数值的列或表达式使用 SUM 函数。

**VARIANCE 函数**

对于跟在 OLAP window VARIANCE 表达式之后的 OVER 子句中定义的查询结果的分区，VARIANCE 函数计算并返回均方差作为对指定的数值列或表达式中值的总体方差的估算。

如果 OVER 子句包括 Window Frame 子句，VARIANCE 函数为每一组行返回一个值，使用下列公式：

(SUM(Xi2) - (SUM(Xi)2)/N)/(N - 1)

在此公式中，

- Xi 是 OVER 子句指定的 window 分区或 frame 中每一列值，
- N 是该列中非 NULL 值的合计数（除非所有值都是 NULL，在此情况下未逻辑地定义该方差，且 VARIANCE 函数返回 NULL）。

您仅可对数值的列应用 VARIANCE 函数。

**示例：带有分区的 AVG 函数**

在下列示例中，在 OLAP window 聚集表达式中使用 AVG 函数来返回两个 window 分区在 2012 年期间的移动平均数 closeprice 列值，基于 symbol 列中的 ABC 和 XYZ 值作为分区键。

```
SELECT symbol, tradingdate,
  AVG(closeprice) OVER (PARTITION BY symbol
    ORDER BY tradingdate
  ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
FROM dailystockdata
WHERE symbol IN ('ABC', 'XYZ')
  AND tradingdate BETWEEN '2012-01-01' AND '2012-12-31';
```

window ORDER 子句指定 tradingdate 列值作为排序键，且 window Frame 子句定义基于 30 个连续的 tradingdate 值的移动 window，以当前行的 tradingdate 结束。

**示例：不带有分区的 AVG**

下列查询返回按天排序的存货价格以及当前天、前一天和后一天的平均价格。由于该查询未包括 PARTITION BY 子句，因此结果集不分区。

```
SELECT price,   AVG(price) OVER (ORDER BY tradingday
  ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
  FROM stock_price
WHERE tradingday BETWEEN '2012-11-01' AND '2012-11-07';
```

| price | (avg) |
|---|---|
| 18.25 | 18.31 |
| 18.37 | 18.31 |
| | 18.37 |
| | 19.03 |
| 19.03 | 18.81 |
| 18.59 | 18.61 |
| 18.21 | 18.40 |

**avg** 列中的第一个值是 **price** 列中前两个值的平均值，因为对于 **price** 列的第一个值没有前面的值。

**avg** 列中的第二个值是 **price** 列中前两个值的平均值，因为 **price** 列的第三行没有值。

**avg** 列中的第三个值等于 **price** 列中的第二个值，因为 **price** 列中的第三行和第四行没有值。

**示例：COUNT 函数**

下列查询返回装运日期、装运费用和按客户排列的每一订单的订单数目。按客户编号对查询结果分区，并限定客户编号小于或等于 110。

```
SELECT customer_num, ship_date, ship_charge,
        COUNT(*) OVER (PARTITION BY customer_num)
FROM orders
WHERE customer_num <= 110;
```

| customer_num | ship_date | ship_charge | (count(*)) |
|---|---|---|---|
| 101 | 05/26/2008 | $15.30 | 1 |
| 104 | 05/23/2008 | $10.80 | 4 |
| 104 | 07/03/2008 | $5.00 | 4 |
| 104 | 06/01/2008 | $10.00 | 4 |
| 104 | 07/10/2008 | $12.20 | 4 |
| 106 | 05/30/2008 | $19.20 | 2 |
| 106 | 07/03/2008 | $12.30 | 2 |
| 110 | 07/06/2008 | $13.80 | 2 |
| 110 | 07/16/2008 | $6.30 | 2 |

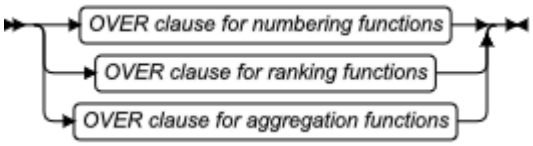
客户 104 在列表中出现四次。客户 104 在 **count** 列中的值始终是 4。

## OLAP window 表达式的 OVER 子句

OVER 子句定义在其上执行 OLAP window 表达式的结果集。

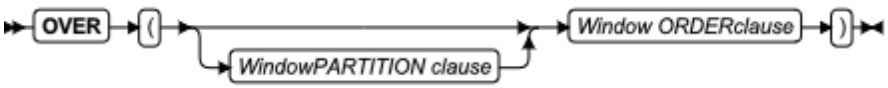
### 语法

OVER 子句



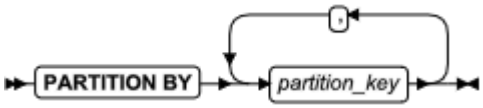

编号函数的 OVER 子句


分等级函数的 OVER 子句

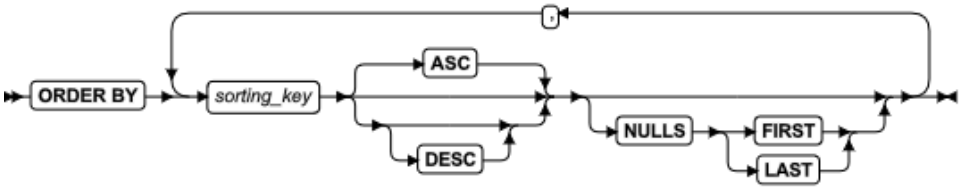



聚集函数的 OVER 子句

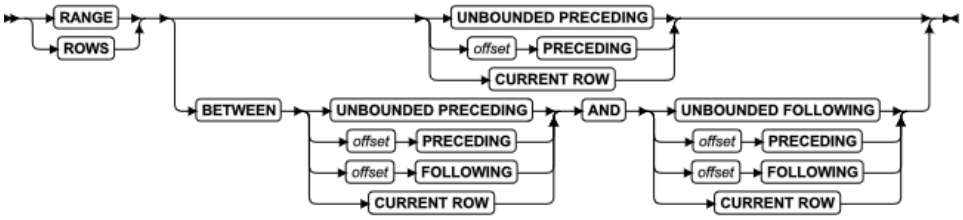
Window PARTITION 子句




Window ORDER 子句




Window Frame 子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *offset* | 表示从当前行位置的偏移量的无符号整数 | 不可为负的。如果为零，则指定当前的行 | 整数 |
| *partition_key* | 由其对行分区的列名称、别名或常量表达式 | 必须在 Projection 子句的选择列表中 | 列表达式 |
| *sorting_key* | 由其对行排序的列名称、别名或常量表达式 | 与对于 *partition_key* 的限制相同。对于 RANGE window frame，仅允许单个排序键，且数据类型必须为数值的、DATE 或 DATETIME。 | 列表达式 |

如果 OVER 子句为空，则您必须还包括空的圆括号。

**Window PARTITION 子句**

OLAP window 分区是由查询返回的行的子集。通过定义该 window 的 OVER 子句的 PARTITION BY 规范中的一个或多个列表达式定义每一分区。数据库服务器将指定的 OLAP window 函数应用于每一 window 分区中的所有行。如果在 OVER 子句中未定义分区，则将 window 函数应用于该查询的结果集中的每行。

**Window ORDER 子句**

数据库服务器根据 window ORDER 子句中的排序键（或多个排序键）对每一 window 分区中的行排序。如果您未指定升序（ASC）或降序（DESC）顺序，则缺省值为 ASC。如果未指定 ORDER 子句，则按照检索到的行的顺序排列符合条件的行。

**Window Frame 子句**

window Frame 子句返回每一 window 分区中的行的子集，称为*聚集组*。由特定数目的行或值的范围来定义 window frame。

基于行的 window frame

ROWS 关键字创建基于行的 window frame，这由在当前行之前或之后或之前与之后的特定数量的行组成。该偏移量表示要返回的行的数目。下列示例返回包括当前行以及当前行之前六行的七行：

```
AVG(price) OVER (ORDER BY year, day
   ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
```

在基于行的 window frame 子句中，偏移量表示为无符号整数，因为关键字 FOLLOWING 指定从当前行的正偏移量，而关键字 PRECEDING 指定从当前行的负偏移量。关键字 UNBOUNDED 指的是从当前行至该 window 分区的限度的所有行。作为在 window Frame 规范中 ROWS 关键字之后的第一个术语，UNBOUNDED PRECEDING 意味着起始边界为该分区中的第一行， 而 UNBOUNDED FOLLOWING 意味着终止边界为该分区中的最后一行。

基于值的 window frame

RANGE 关键字创建基于值的 frame 子句，由当前行与满足标准的行组成，通过 ORDER 子句中的排序键设置该标准并符合指定的偏移量。偏移量表示排序键的数据类型的单位数目。排序键必须为数值的、DATE 或 DATETIME 数据类型。例如，如果排序键为 DATE 数据类型，则偏移量表示特定的天数。下列示例返回发运日期在当前行的两天之内的行的数目加上当前行的总数：

```
COUNT(*) OVER (ORDER BY ship_date
   RANGE BETWEEN 2 PRECEDING AND 2 FOLLOWING)
```

基于值的 window frame 定义在包含指定范围的数值值的 window 分区内的行。OVER 函数的 window ORDER 子句定义应用 RANGE 规范的数值的、DATE 或 DATETIME 列，现对于那一列的当前行值。在基于值的 window frame 的 ORDER 子句中仅允许排序键。

在基于行和基于值的这两种情况下，在此 window frame 的内容上计算 OLAP 函数，而不是在整个分区的固定的内容上计算。window frame 不需要包含当前行。例如，下列规范定义仅包含当前行之前的行的 window frame：

```
ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING
```

如果您未为 window 聚集函数指定 window ORDER 子句，则在缺省情况下，不限制结果集，其等同于下列 window frame 规范：

```
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

如果您为 window 聚集函数指定 ORDER 子句但无 window frame 子句，则在缺省情况下，返回当前行之前的所有行以及当前行，其等同于下列 window frame 规范：

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

**示例：不带有 window frame 的 SUM 函数**

下列查询返回按一年的季度的销售额，以及按季度的销售额的累积总和。

```
SELECT sales, SUM(sales) OVER (ORDER BY quarter)
FROM sales WHERE year = 2012
```

```
sales       (sum)
120          120
135          255
127          382
153          535
```

第四季度的销售额的总和等于所有四个季度中的销售额。

由于该查询未包括 window frame 子句，因此，SUM 函数如通过 FROM 子句指定的那样，在整个结果集上操作。

**示例：基于行的 window frame**

下列查询按照团队分区并按照分数排序来返回运动员。在每一分区内，对该运动员以及与前面的运动员的分数求平均值：

```
select team, player, points,
    AVG(points) OVER(PARTITION BY team ORDER BY points
        ROWS 1 PRECEDING AND CURRENT ROW) AS olap_avg
FROM points;
```

```
TEAM       PLAYER       POINTS OLAP_AVG
A           Singh            7    7.00000000000
A           Smith           14 10.50000000000
B           Osaka            8    8.00000000000
B           Ricci           12 10.00000000000
B           Baxter          18 15.00000000000
C           Chun            13 13.00000000000
D           Kwan             9    9.00000000000
D           Tran            16 12.50000000000
```

**示例：基于范围的 window frame**

下列查询按照团队分区并按照年龄排序，返回运动员。在每一分区内，对每一运动员以及最多大 9 岁的运动员的分数求平均值：

```
SELECT player, age, team, points,
   AVG(points) OVER(PARTITION BY team ORDER BY age
   RANGE BETWEEN CURRENT ROW AND 9 FOLLOWING) AS olap_avg
FROM points_age;
```

```
PLAYER    AGE    TEAM          POINTS OLAP_AVG
Singh      25      A              7 10.50000000000
Smith      26      A             14 14.00000000000
Baxter     27      B             18 13.00000000000
Osaka      35      B              8 10.00000000000
Ricci      40      B             12 12.00000000000
Chun       21      C             13 13.00000000000
Kwan       22      D              9 12.50000000000
Tran       31      D             16 16.00000000000
```

在分区 A 中，Singh 的平均值包括 Smith 的分数，因为 Smith 比 Singh 大一岁。Smith 的平均值不包括来自 Singh 的分数，因为 Singh 比 Smith 年轻。

在分区 B 中，Baxter 的平均值包括 Osaka 的分数，其比 Baxter 大 8 岁，但不包括 Ricci，其比 Baxter 大 13 岁。

在分区 D 中，Kwan 的平均值包括 Tran 的分数，因为 Tran 比 Kwan 大 9 岁。

**示例：不带有当前行的 Window frame**

下列查询计算分区中前面两行的分数的平均值：

```
SELECT player, age, team, points,
    AVG(points) OVER(PARTITION BY team ORDER BY age
    ROWS BETWEEN 2 PRECEDING AND 1 PRECEDING) AS olap_avg
    FROM points_age;
```

```
PLAYER      AGE    TEAM        POINTS     OLAP_AVG
Singh        25      A            7                NULL
Smith        26      A           14      7.00000000000
Baxter       27      B           18                NULL
Osaka        35      B            8     18.00000000000
Ricci        40      B           12     13.00000000000
Chun         21      C           13                NULL
Kwan         22      D            9                NULL
Tran         31      D           16      9.00000000000
```

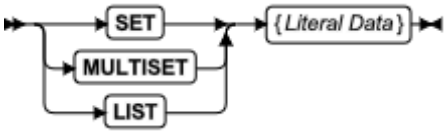
在分区 B 中，Ricci 的平均值是基于 Baxter 和 Osaka 的分数合计：(18 + 8 = 26)/2 = 13。当当前行没有前面的行用于计算时，结果为 NULL。

## 4.9 文字的集合

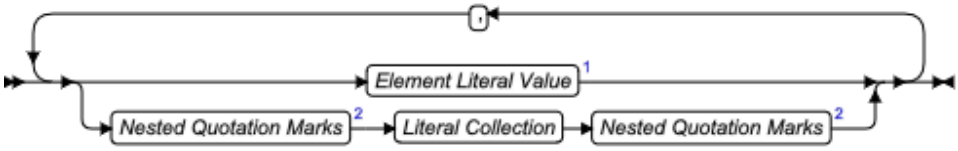
使用 Literal Collection 段来指定集合数据类型的值。要了解返回集合内的单个元素的值的表达式的语法，请参阅 集合构造函数。

**语法**

文字的集合




文字的数据




**用法**

您可为 SET、MULTISET 或 LIST 数据类型指定文字的集合值。

要指定单个文字的集合值，请指定集合类型和文字的值。下列的 SQL 语句将四个整数值插入到声明为 SET(INT NOT NULL) 的名为 set_col 的列内：

```
INSERT INTO table1 (set_col) VALUES (SET{6, 9, 9, 4});
```

以一对空的大括号（{ }）指定空的集合。此示例将空的列表插入到声明为 LIST(INT NOT NULL)： 的列 list_col 内：

```
INSERT INTO table2 (list_col) VALUES ('LIST{}');
```

一对单引号（ '）或双引号（"）可定界集合。然而除了定界 SQL 标识符之外，在启用了定界的表达式的数据库中，双引号不是有效的。

如果您将一个文字的集合作为参数传递到 SPL 例程，则请确保在围绕着参数的圆括号与表明文字集合的开头与结尾的引号之间有空格。

## 元素文字的值

文字的集合 的图引用此部分。对于下列数据类型，集合的元素可为文字的值。

| 对于类型的集合 | 文字的值语法 |
|---|---|
| BOOLEAN | t 或 f，作为引号括起的字符串表示 TRUE 或 FALSE |
| CHAR、VARCHAR、NCHAR、NVARCHAR、CHARACTER VARYING、LVARCHAR、DATE | 引用字符串 |
| DATETIME | 文字的 DATETIME |
| DECIMAL、MONEY、FLOAT、INTEGER、INT8、SMALLFLOAT、SMALLINT | 精确数值 |
| INTERVAL | 文字的 INTERVAL |

| 对于类型的集合 | 文字的值语法 |
| --- | --- |
| Opaque 数据类型 | 引用字符串。对于相关联的 opaque 类型，输入支持函数必须识别该字符串文字。 |
| Row 类型 | Literal Row。当集合元素类型为命名的 ROW 类型时，您不需要将插入了的值强制转型为命名的 ROW 类型。 |

**重要：** 您不可指定简单大对象数据类型（BYTE 和 TEXT）作为集合的元素类型。

必须以不同类型的引号指定引用字符串，而不是括起集合的引号，以便于数据库服务器可解析引用字符串。因此，如果您使用双引号（"）来指定集合，则请使用单引号（'）来指定单个的引用字符串元素。（然而，在启用定界的标识符的数据库中，除了用来定界 SQL 标识符之外，双引号不是有效的。）

## 嵌套的引号

文字的集合 的图引用此部分。

***嵌套的集合***是为另一集合的元素类型的集合。

无论您何时嵌套集合文字，请使用嵌套的引号。在这些情况下，您必须遵循嵌套的引号的规则。否则，数据库不可正确地解析该字符串。

通用的规则是，对于每一新的嵌套级别，您必须加倍引号的数目。例如，如果您为第一级使用双引号（"），则你必须为第二级使用两个双引号，为第三级使用四个双引号，为第四级使用八个，为第五级使用十六个，依此类推。

同样地，如果您对第一级使用单引号（'），则您必须为第二级使用两个单引号，为第三级使用四个单引号。对您可嵌套的级数没有限制，只要您遵循此规则即可。

下列示例说明两级嵌套的集合文字的情况，使用双引号（"）。在此，表 **tab5** 是单列表，其唯一的列 **set_col** 为嵌套的集合类型。

下列语句创建 tab5 表：

```
CREATE TABLE tab5 (set_col SET(SET(INT NOT NULL) NOT NULL));
```

下列语句将值插入到表 tab5 内：

```
INSERT INTO tab5 VALUES ( "SET{""SET{34, 56, 23, 33}""}" );
```

对于每一文字值，开引号与收引号必须相匹配。因此，如果您以两个双引号开启文字，则您必须以两个双引号关闭那个文字。（""a literal value""）。

要在 GBase 8s ESQL/C 程序中的 SQL 语句内指定嵌套的引号， 对于由单引号定界的字符串之中的每个双引号，请使用 C 转义字符。否则 GBase 8s ESQL/C 预处理器 不可正确地解释文字的集合值。例如，前面的对于 tab5 表的 INSERT 语句会如下出现在 GBase 8s ESQL/C 程序中：

```
EXEC SQL insert into tab5 values ('set{\"set{34, 56, 23, 33}\"}');
```

要获取更多信息，请参阅 *GBase 8s ESQL/C 程序员手册* 中关于复合的数据类型的章节。

如果该集合为嵌套的集合，则您必须为每一级集合类型包括集合构造函数语法。假设您定义下列列：

```
nest_col SET(MULTISET (INT NOT NULL) NOT NULL);
```

下列语句将三个元素插入到 nest_col 列内：

```
INSERT INTO tabx (nest_col)
VALUES ("SET{'MULTISET{1, 2, 3}'}");
```

## 4.10 文字的 DATETIME

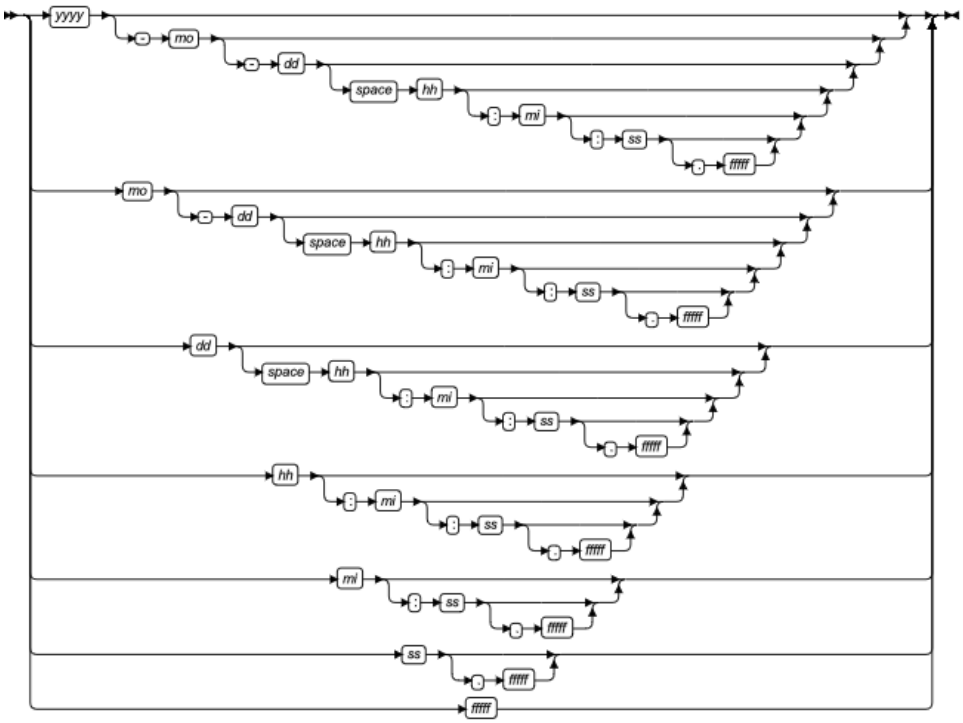
文字的 DATETIME 段指定 DATETIME 值

当您在语法图中看到对文字的 DATETIME 的引用时，请使用此段。

**语法**

文字的 DATETIME


数值的日期和时间




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dd* | 月的日，以数字表示 | 1 ≤ *dd* ≤ 28、29、30 或 31 | 精确数值 |
| *fffff* | 秒的小数部分，以数字表示 | 0 ≤ *fffff* ≤ 99999 | 精确数值 |
| *hh* | 日的小时，以数字表示 | 0 ≤ *hh* ≤ 23 | 精确数值 |

| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *mi* | 小时的分钟，以数字表示 | 0 ≤ *mi* ≤ 59 | 精确数值 |
| *mo* | 年的月，以数字表示 | 1 ≤ *mo* ≤ 12 | 精确数值 |
| *space* | 空格（ASCII 32） | 正好 1 空字符 | 文字的空格 |
| ss | 分钟的秒，以数字表示 | 0 ≤ *ss* ≤ 59 | 精确数值 |
| *yyyy* | 年，以数字表示 | 不超过 4 位数字 | 精确数值 |

**用法**

您必须为文字的 DATETIME 段中的此日期既指定数字的日期也指定 DATETIME 字段。DATETIME 字段限定符必须对应于您指定的数值的日期。 例如，如果您指定包括年作为最大单位且分钟作为最小单位的数值的日期，则您必须还指定 YEAR TO MINUTE 作为 DATETIME 字段限定符。

如果您为概念指定两位数字，则数据库服务器使用 **DBCENTURY** 环境变量的设置来将缩写的年份值扩展为四位数字。 如果未设置 **DBCENTURY**，则使用当前年份的前两位数值来扩展缩写的年份值。

下列示例展示文字的 DATETIME 值：

```
DATETIME (07-3-6) YEAR TO DAY
DATETIME (09:55:30.825) HOUR TO FRACTION
DATETIME (07-5) YEAR TO MONTH
```

下列示例展示随同 EXTEND 函数使用的文字的 DATETIME 值：

```
EXTEND (DATETIME (2007-8-1) YEAR TO DAY, YEAR TO MINUTE)
                - INTERVAL (720) MINUTE (3) TO MINUTE
```

## DATE 和 DATETIME 格式规范的优先顺序

如果不同的设置存在冲突，或如果未指定格式，则其设置可为 DATE 数据类型的值指定显示和数据项格式的 GBase 8s 环境变量有下列优先顺序：

1. DBDATE
2. GL_DATE
3. 在客户端语言环境中定义的信息（如果设置 CLIENT_LOCALE 的话）
4. 缺省的日期格式为 `%m/%d/%iy`（如果未定义 DBDATE 和 GL_DATE，且未指定语言环境的话）

GBase 8s 环境变量可为 DATETIME 数据类型的值指定显示和数据项格式。如果不同的设置存在冲突，或如果未指定格式，则它们的显式的或缺省的设置有下列降序的优先顺序（从最高至最低）：

1. DBDATE 和 DBTIME
2. GL_DATETIME
3. 在客户端语言环境中定义的信息（如果设置 CLIENT_LOCALE 的话）

4. 缺省的 DATETIME 格式为 %iY-%m-%d %H:%M:%S（如果未设置 CLIENT_LOCALE、DBTIME 和 GL_DATETIME 的话）。

如果将 GL_DATETIME 设置为非缺省的值，则在您可在下列上下文中正确地处理本地化的 DATETIME 值之前，您还必须将 USE_DTENV 环境变量设置为 1：

- dbexport 实用程序
- dbimport 实用程序
- DB-Access 的 LOAD 语句
- DB-Access 的 UNLOAD 语句
- 由 CREATE EXTERNAL TABLE 语句定义的对象上的 DML 操作。

对于按发生时间顺序排列的数据值，要了解您可如何设置这些环境变量来定义格式的详细信息，请参阅 *GBase 8s GLS 用户指南* 和 *《GBase 8s SQL 指南：参考》*。

## 将数值的日期和时间字符串强制转型为 DATE 数据类型

数据库服务器提供内建的强制转型来将 DATETIME 值转换为 DATE 值，如在下列 SPL 程序片段中所示：

```
DEFINE my_date DATE;
DEFINE my_dt DATETIME YEAR TO SECOND;
  . . .
LET my_date = CURRENT;
```

在此，CURRENT 返回的 DATETIME 被隐式地强制转型为 DATE。您还可显式地将 DATETIME 强制转型为 DATE：

```
LET my_date = CURRENT::DATE;
```

这两个 LET 语句将来自 DATETIME 值的 ***year***、***month*** 和 ***day*** 指定到 DATE 类型的本地 SPL 变量 **my_date**。

类似地，如同在 文字的 DATETIME 语法图中定义的那样，您可显示地将有“数值的日期和时间”段的格式的字符串强制转型为 DATETIME 数据类型，如下例所示：

```
LET my_dt = ('2008-02-22 05:58:44.000')::DATETIME YEAR TO SECOND;
```

然而，对于直接地将有“数值的日期和时间”格式的字符串转换为 DATE 值，既没有隐式的也没有显式的内建的强制转型。 例如，下列两个语句都失败并报错 -1218：

```
LET my_date = ('2008-02-22 05:58:44.000');
LET my_date = ('2008-02-22 05:58:44.000')::DATE;
```

要将指定有效的数值日期和时间值的字符串转换为 DATE 数据类型，您必须首先将该字符串强制转型为 DATETIME，然后再将结果 DATETIME 值强制转型为 DATE，如此例所示：

```
LET my_date =
        ('2008-02-22 05:58:44.000')::DATETIME YEAR TO SECOND::DATE;
```

仅当字符串指定有效的 DATE 值时，直接的从字符串到 DATE 的强制转型才可成功。

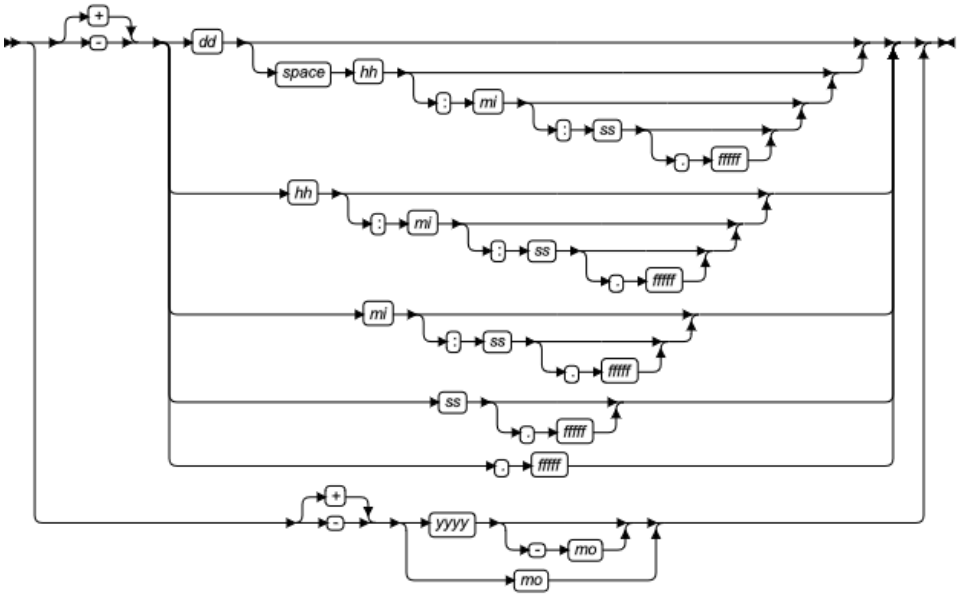
# 4.11 文字的 INTERVAL

文字的 INTERVAL 段指定文字的 INTERVAL 值。每当您在语法图中看到对文字的 INTERVAL 引用时，请使用此段。

**语法**

文字的 INTERVAL


数值的时间跨度




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dd* | 天数 | -10**10 < *dd* < 10**10 | 精确数值 |
| *fffff* | 秒的小数部分 | 0 ≤ *fffff* ≤ 9999 | 精确数值 |
| *hh* | 小时数 | 如果不是第一个，则 0 ≤ *hh* ≤ 23 | 精确数值 |
| *mi* | 分钟数 | 如果不是第一个，则 0 ≤ *mi* ≤ 59 | 精确数值 |
| *mo* | 月数 | 如果不是第一个，则 0 ≤ *mo* ≤ 11 | 精确数值 |
| *space* | 空格（ASCII 32） | 要求正好 1 个空字符 | 文字的空格 |
| ss | 秒数 | 如果不是第一个，则 0 ≤ *ss* ≤ 59 | 精确数值 |
| *yyyy* | 年数 | -10**10 < *yyyy* < 10**10 | 精确数值 |

**用法**

不像 DATETIME 文字那样，INTERVAL 文字可包括一元加号（+）或一元减号（ -）。如果您未指定加减号，则缺省值为加号。

可由 INTERVAL 限定符指定第一个时间单位的精度。FRACTION 可有不超过 5 位数字的精度，除了 FRACTION 之外，第一个时间单位可有高达 9 位数字的精度，如果您在 INTERVAL 列或变量的声明中指定了非缺省的精度的话。

下列示例展示文字的 INTERVAL 值：

```
INTERVAL (3-6) YEAR TO MONTH
INTERVAL (09:55:30.825) HOUR TO FRACTION
INTERVAL (40 5) DAY TO HOUR
INTERVAL (299995.2567) SECOND(6) TO FRACTION(4)
```

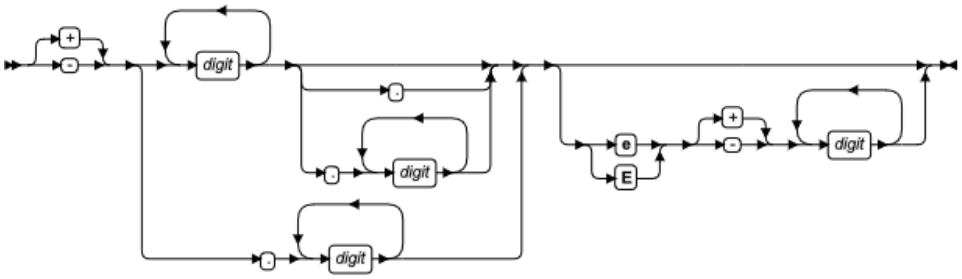
仅最后一个示例有非缺省的精度。要了解声明 INTERVAL 数据类型的精度的语法以及每一时间单位的缺省值，请参阅 INTERVAL 字段限定符。

## 4.12 精确数值

***精确数值***是作为整数的、作为定点小数的或以指数计数法的以 10 为基数的实数表示。每当您在语法图中看到对文字数值的引用时，请使用 Literal Number 段。

**语法**

精确数值




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *digit* | 取值范围从 0 至 9 的整数 | 必须为 ASCII 数字 | 从键盘输入的文字。 |

**用法**

您可包括逗号（,）或空字符（ASCII 32）。一元加号（+）或减号（-）可出现在精确数值、假数或指数之前。

您不可在精确数值中包括非 ASCII 数字，诸如某些非缺省的语言环境支持的 Hindi 数值。

### 整数

整部没有小数部分且不可包括小数点。可精确地表示为文字的整数的内建的 SQL 数据类型包括 BIGINT、BIGSERIAL、DECIMAL(*p*, 0)、INT、INT8、SERIAL、SERIAL8 和 SMALLINT。

如果您在文字整数有效的任何上下文中，使用 10 以外的基数（诸如二进制、八进制或十六进制）来表示数目，则数据库服务器会试图将该值解释成以 10 为基本数字的文字整数。对于大多数的数据值，结果将是不正确。

下列示例展示一些有效的整数：

10 -27 +25567

在整数中，千位分隔符（比如逗号）不是有效的，在任何其他精确数值用也无效。

### 定点小数

定点小数正好可表示 DECIMAL(*p, s*) 和 MONEY 值。这些可包括小数点：

-123.456 00123456 +123456.0

在这些示例中小数点右边的数字是该数值的小数部分。

### 浮点小数

浮点小数正好表示 FLOAT、SMALLFLOAT 和 DECIMAL(*p*) 值，使用小数点或指数表示法，或两者都用。 您可以指数表示法粗略地表示实数。下一示例展示浮点数值：

-123.45E6 1.23456E2 123456.0E-3

在前面的示例中的 E 是指数表示法的符号。跟在 E 之后的数字是指数的值。例如，数值 3E5（或 3E+5）意味着 3 乘以 10 的五次方，而数值 3E-5 意味着 3 乘以 10 的五次方的倒数。

### 精确数值和 MONEY 数据类型

当您使用精确数值作为 MONEY 值时，请不要包括币种符号或包括逗号。**DBMONEY** 函数或语言环境文件可确定在输出中如何显示 MONEY 值的格式。

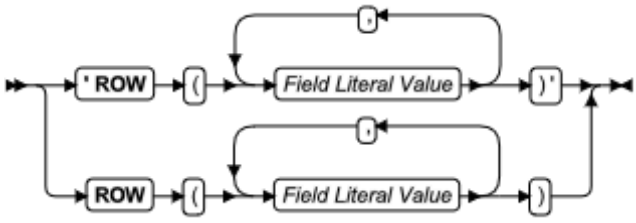
## 4.13 Literal Row

Literal Row 段指定命名的和未命名的 ROW 数据类型的文字的值的语法。

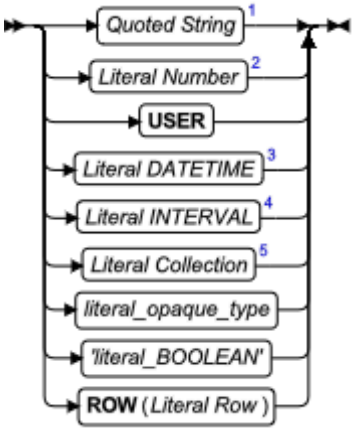
要了解在 ROW 数据类型内求值为字段值的表达式的信息，请参阅 ROW 构造函数。

**语法**

Literal Row




字段文字的值




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *literal_opaque_type* | opaque 数据类型的文字的表示 | 对于相关联的 opaque 数据类型，必须为输入支持函数识别的文字 | 由 opaque 数据类型的开发者定义 |
| *literal_BOOLEAN* | BOOLEAN 值的文字的表示 | 必须指定 't'(= TRUE) 或 'f'(= FALSE) 作为引用字符串 | 引用字符串 |

**用法**

您可为命名的 ROW 和未命名的 ROW 数据类型指定文字的值。ROW 构造函数引入文字的 ROW 值，在引号之间可可选地包括它。

ROW 类型的每一字段的值的格式必须与相应的字段的数据类型相兼容。

**重要：** 您不可指定简单大对象（BYTE 或 TEXT）作为一行的字段类型。

一行的字段可为下列表格中数据类型的文字的值。

| 对于类型的字段 | 文字的值语法 |
|---|---|
| BOOLEAN | t 或 f，表示 TRUE 或 FALSE |
| CHAR、VARCHAR、LVARCHAR、NCHAR、NVARCHAR、CHARACTER VARYING、DATE | 引用字符串 |
| DATETIME | 文字的 DATETIME |
| DECIMAL、MONEY、FLOAT、INTEGER、INT8、SMALLFLOAT、SMALLINT | 精确数值 |
| INTERVAL | 文字的 INTERVAL |
| Opaque 数据类型 | 引用字符串<br>对于相关联的 opaque 类型，该字符串必须是由输入支持函数识别的文字。 |
| 集合类型（SET、MULTISET、LIST） | 文字的集合<br>要获取关于作为变量或列值的文字的 |

| | |
|---|---|
| | 集合值的信息，请参阅 嵌套的引号。要获取关于 ROW 类型的文字的集合值的信息，请参阅 嵌套的 Row 的文字。 |
| 另一 ROW 类型（命名的或未命名的） | 要获取关于 ROW 类型值的信息，请参阅 嵌套的 Row 的文字。 |

## 未命名的 Row 类型的文字

要为未命名的 ROW 类型指定文字的值，请以 ROW 构造函数引入文字的行；您必须将这些值括在圆括号之间。例如，假设您定义 rectangles 表如下：

```
CREATE TABLE rectangles
        (
        area FLOAT,
        rect ROW(x INTEGER, y INTEGER, length FLOAT, width FLOAT),
        )
```

下列 INSERT 语句将这些值插入到 rectangles 表的 rect 列内：

```
INSERT INTO rectangles (rect)
    VALUES ("ROW(7, 3, 6.0, 2.0)")
```

## 命名的 Row 类型的文字

要为命名的 ROW 类型指定文字的值，请以 ROW 类型构造函数引入文字的行，并将每一字段的文字的值括在圆括号中。此外，您可将行文字强制转型为适合的命名的 ROW 类型，来确保生成行值作为命名的 ROW 类型。下列语句创建命名的 ROW 类型 address_t 以及 employee 表：

```
CREATE ROW TYPE address_t
        (
        street CHAR(20),
        city CHAR(15),
        state CHAR(2),
        zipcode CHAR(9)
        );

        CREATE TABLE employee
        (
        name CHAR(30),
        address address_t
        );
```

下列的 INSERT 语句将值插入到 employee 表的 address 列内：

```
INSERT INTO employee (address)
        VALUES (
        "ROW('103 Baker St', 'Tracy','CA', 94060)"::address_t)
```

### 嵌套的 Row 的文字

如果是嵌套的 row 的文字的值，请为每一行级别指定 ROW 类型构造函数。如果您包括引号作为定界符，则应将它们括在最外面的 row。例如，假设您创建 emp_tab 表：

```
CREATE TABLE emp_tab
        (
        emp_name CHAR(10),
        emp_info ROW( stats ROW(x INT, y INT, z FLOAT))
        );
```

下列 INSERT 语句将一行添加到 emp_tab 表：

```
INSERT INTO emp_tab VALUES ('joe boyd', "ROW(ROW(8,1,12.0))" );
```

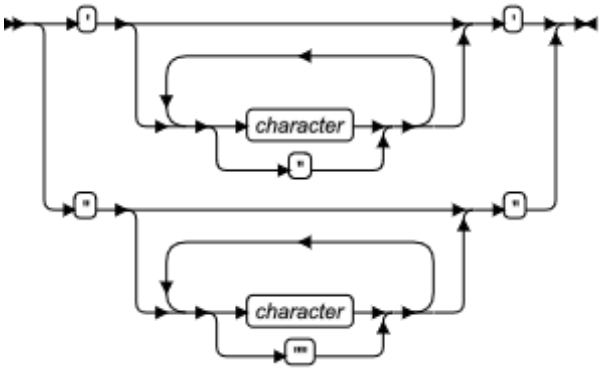
类似地，如果 row 字符串文字包含嵌套的集合，则仅可将最外面的文字的 row 括在引号之间。请不要将引号置于嵌套的集合类型的内部。

## 4.14 引用字符串

引用字符串是引号之间的字符串文字。每当您在语法图中看到对引用字符串的引用时，请使用此段。

**语法**

引用字符串




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *character* | 引用字符串之内的代码集元素 | 如果设置 DELIMIDENT 环境变量，则不可括在双引号之间 | 来自键盘的文字值 |

**用法**

使用引用字符串来指定在数据操纵语句和其他 SQL 中的字符串文字。例如，您可在 INSERT 语句中使用引用字符串来将值插入到字符数据类型的列内。

### 对指定用引号括起来的字符串中字符的限制

在用引号括起来的字符串中的 *character* 上，您必须遵守下列限制：

- 如果您正在使用 ASCII 代码集，则您可指定任何可打印的 ASCII 字符，包括单引号（'）或双引号（"）。要了解应用于用引号括起来的字符串中使用引号的限制，请参阅 使用字符串中的引号。
- 在某些语言环境中，您可指定非 ASCII 字符，包括该语言环境支持的多字节字符。请参阅 *GBase 8s GLS 用户指南* 中关于用引号括起来的字符串的讨论。
- 如果您为用引号括起来的字符串启用换行字符，则您可在用引号括起来的字符串中嵌入换行字符。要获取更多信息，请参阅 用引号括起来的字符串中的换行字符。
- 您可输入 DATETIME 和 INTERVAL 数据值作为用引号括起来的字符串。要了解应用于在用引号括起来的字符串中输入的 DATETIME 和 INTERVAL 数据的限制，请参阅 作为字符串的 DATETIME 和 INTERVAL 值。
- 在搜索条件中随同 LIKE 或 MATCHES 关键字使用用引号括起来的字符串，可包括在该搜索条件中有特定含义的通配字符。要了解更多信息，请参阅 条件中的 LIKE 和 MATCHES。
- 当您插入一个用引号括起来的字符串值时，您必须遵守若干限制。要获取更多信息，请参阅 插入值作为用引号括起来的字符串。

## DELIMIDENT 环境变量

如果在数据库服务器上设置 **DELIMIDENT** 环境变量，则您不可使用双引号（"）来定界文字的字符串。如果设置 **DELIMIDENT**，则数据库服务器将括在双引号中的字符串解释为 SQL 标识符，而不是作为文字的字符串。如果未设置 **DELIMIDENT**，则将双引号之间的字符解释为文字的字符串，而不是标识符。要获取更多信息，请参阅 使用字符串中的引号，以及 *《GBase 8s SQL 指南：参考》* 中关于 **DELIMIDENT** 的描述。

在客户端系统上也支持 **DELIMIDENT**，在此，可将它设置为 y、为 n，或不设置。

- y 指定客户端应用必须使用单引号（'）来定界文字的字符串，且必须仅在定界的 SQL 标识符周围使用双引号（"）。与未定界的标识符相比，定界的标识符可支持更大的字符集。定界的字符串或定界的标识符之内的字母是区分大小写的。
- n 指定客户端应用可使用双引号（"）或单引号（'）来定界字符串，但不定界 SQL 标识符。如果数据库服务器在需要 SQL 标识符的上下文中遇到由双引号或单引号定界的字符串，则它发出错误。然而，可由单引号（'）定界限定 SQL 标识符的所有者名称。您必须使用一对相同的引号符号来定界字符串。
- 在客户端系统上不带有值地指定 **DELIMIDENT**，要求客户端应用使用 **DELIMIDENT** 设置，其为它们的应用编程接口（API）的缺省值。

GBase 8s 的客户端 API 使用下列缺省的 DELIMIDENT 设置：

- 对于 OLE DB 和 .NET，缺省的 **DELIMIDENT** 设置为 y
- 对于 ESQL/C、JDBC 和 ODBC，缺省的 **DELIMIDENT** 设置为 n
- 以 ESQL/C 作为底层的 API、DataBlade API (LIBDMI) 以及 C++ API 的行为如同 ESQL/C，并使用 'n' 作为缺省值，如果在客户端系统上未指定 **DELIMIDENT** 的值的话。

即使设置 DELIMIDENT，您也可使用单引号（’）来定界*授权标识符*作为数据库对象名称的所有者名称组件，如下例所示：

```
RENAME COLUMN 'Owner'.table2.collum3 TO column3;
```

然而，通用的规则是，当设置 DELIMIDENT 时，SQL 解析器将由单引号定界的字符串解释为字符串文字，将由双引号（″）定界的字符串解释为 SQL 标识符。

## 用引号括起来的字符串中的换行字符

在缺省情况下，必须在单行上编写字符串常量。也就是说，您不可在用引号括起来的字符串中使用嵌入的换行字符。然而，您可以两种方法之一来重写此缺省的行为：

- 要在所有会话中启用用引号括起来的字符串中的换行字符，请将 ONCONFIG 文件中的 ALLOW_NEWLINE 参数设置为 1。
- 要为当前的会话的启用用引号括起来的字符串中的换行字符，请执行内建的函数 **IFX_ALLOW_NEWLINE**。

这为当前的会话启用用引号括起来的字符串中的换行字符：

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('T');
```

如果未为会话启用用引号括起来的字符串中的换行字符，则下列语句是无效的并返回错误：

```
SELECT 'The quick brown fox
jumped over the old gray fence'
FROM customer
WHERE customer_num = 101;
```

然而，如果您为会话启用用引号括起来的字符串中的换行字符，在前面的使用中的语句是有效的并执行成功。

要获取更多关于 **IFX_ALLOW_NEWLINE** 函数的信息，请参阅 IFX_ALLOW_NEWLINE 函数。要获取关于 ONCONFIG 文件中 ALLOW_NEWLINE 参数的更多信息，请参阅您的 *GBase 8s 管理员参考手册*。

## 使用字符串中的引号

在由双引号定界的字符串文字中，单引号（'）没有特别的意义。反过来，在由单引号定界的字符串中，双引号（"）没有特别的意义。例如，这些字符串是有效的：

```
"Nancy's puppy jumped the fence"
'Billy told his kitten, "No!" '
```

由双引号定界的字符串可通过以另一个双引号在前来包括一双引号字符，如下列字符串所示：

```
"Enter ""y"" to select this row"
```

当设置 **DELIMIDENT** 环境变量时，双引号仅可定界 SQL 标识符，不可定界字符串。要获取关于定界的标识符的更多信息，请参阅 定界标识符。

您可以 文字的 DATETIME 和 文字的 INTERVAL 中描述的文字的形式输入 DATETIME 和 INTERVAL 数据，或者，您可输入它们作为用引号括起来的字符串。

自动地将作为字符串输入的有效的文字转换为 DATETIME 或 INTERVAL 值。

这些语句输入 INTERVAL 和 DATETIME 值作为用引号括起来的字符串：

```
INSERT INTO cust_calls(call_dtime) VALUES ('2007-5-4 10:12:11');
INSERT INTO manufact(lead_time) VALUES ('14');
```

在用引号括起来的字符串中的值的格式，必须与由该列的 INTERVAL 或 DATETIME 限定符指定的格式完全相匹配。对于前面的示例中的第一个 INSERT，必须以对于 INSERT 语句为有效的限定符 YEAR TO SECOND 来定义 **call_dtime** 列。

## 条件中的 LIKE 和 MATCHES

在条件中带有 LIKE 或 MATCHES 关键字的用引号括起来的字符串可包括通配符字符。要获取如何使用通配符字符的完整描述，请参阅 条件。

## 插入值作为用引号括起来的字符串

在缺省的语言环境中，如果您正在插入一个为用引号括起来的字符串的值，则您必须遵守下列限制：

- 将 CHAR、VARCHAR、NCHAR、NVARCHAR、DATE、DATETIME、INTERVAL 和 LVARCHAR 值括在引号中。
- 以 ***mm/dd/yyyy*** 格式指定 DATE 值（或以 **DBDATE** 或 **GL_DATE** 环境变量指定的格式，如果设置了的话）。
- 您不可插入长度大于 2GB 的字符串。
- 带有小数值的数值必须包括小数分隔符。在缺省的语言环境中，逗号（,）不是有效的小数分隔符。
- MONEY 值不可包括美元号（$）或逗号。
- 如果一列接受空值，则您可在列中输入 NULL。

## 在字符列上的数值操作

请避免将数值文字与字符列比较。它要求将所有被比较的字符串转换为数值，这会花费比比较两个字符串长得多的时间。

例如，假设您想要找到 356 电话交换代码内所有的客户：

```
SELECT lname FROM customer WHERE phone [5,7] = '356';
```

请注意，将其值为 356 的运算对象括在引号中。该引号表明数据库服务器必须将该过滤器处理为字符串。相反，当该运算对象不在引号中时，服务器将每一检索的值作为数值处理，并必须隐式地将从该表检索到的每一值强制转型为数值的数据类型。

下列示例导致 phone 子字符串的隐式的数据类型转换：

```
SELECT lname FROM customer WHERE phone [5,7] = 356;
```

如果已经在此列上运行了 UPDATE STATISTICS MEDIUM 或 UPDATE STATISTICS HIGH 语句，则查询优化器试图通过将查询中的常量与保存在分布 bin 中的子字符串相匹配，来确定谓词的选择性。对字符列中每行进行数据类型转换，以便它可与数值过滤器相比较，与第一个示例中查询的成本相比，这种需要毫无必要地增加了忽略 356 周围的引号定界符的查询的成本。

如果数据库服务器不可转换该字符串，则比较字符串与数值的查询可失败并报错 EM -1213。如果您不可避免地将数值的过滤器应用于字符数值，则请仅在数值列上尝试这种操作，其字符限制为取值范围从 ASCII 0x30 至 0x39 的数字，以及小数点（ASCII 0x2e）。此取值范围又称为***半数值的***。

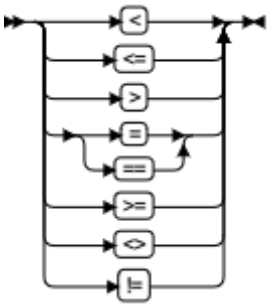
当 DML 语句将带有非字符值的字符列与在长度上不等于该字符列的非字符值相比较时，数据库服务器不使用索引。

## 4.15 关系运算符

关系运算符定量地比较两个表达式。每当您在语法图中看到对关系运算符的引用时，请使用 Relational Operator 段。

**语法**

关系运算符



**用法**

SQL 的关系运算符有下列含义。

| 关系运算符 | 含义 |
|---|---|
| < | 小于 |
| <= | 小于或等于 |
| > | 大于 |
| = ***或*** == | 等于 |
| >= | 大于或等于 |
| <> ***或*** != | 不等于 |

**用法**

对于数值表达式，***大于***意味着在实数轴的右边。

对于 DATE 和 DATETIME 表达式，***大于***意味着时间上较晚。

对于 INTERVAL 表达式，***大于***意味着更大的时间跨度。

对于 CHAR、VARCHAR 和 LVARCHAR 表达式，***大于***意味着在代码集顺序***之后***。

对于 NCHAR 和 NVARCHAR 表达式，***大于***意味着在本地化的排序顺序***之后***，如果存在的话；否则，***大于***意味着在代码集顺序***之后***。

如果为语言环境定义基于语言环境的排序顺序，则将其用于 NCHAR 和 NVARCHAR 表达式。因此，对于 NCHAR 和 NVARCHAR 表达式，*大于*意味着在基于语言环境的排序顺序*之后*。要获取更多关于基于语言环境的排序顺序以及 NCHAR 和 NVARCHAR 数据类型的信息，请参阅 *GBase 8s GLS 用户指南*。

要获取关于在有 NLCASE INSENSITIVE 属性的数据库中带有 NCHAR 和 NVARCHAR 运算对象的关系运算符表达式与区分大小写的数据库中它们的行为有何差异的信息，请参阅主题 在区分大小写的数据库中的 NCHAR 和 NVARCHAR 表达式。

## 使用运算符函数代替关系运算符

每一关系运算符都绑定特定的运算符函数，如表格所示。运算符函数接受两个值，并返回真、假或未知的布尔值。

| 关系运算符 | 相关联的运算符函数 |
|---|---|
| < | **lessthan( )** |
| <= | **lessthanorequal( )** |
| > | **greaterthan( )** |
| >= | **greaterthanorequal( )** |
| = *或* == | **equal( )** |
| <> *或* != | **notequal( )** |

以关系运算符连接两个表达式等同于对表达式调用运算符函数。例如，下两个语句都选择运费多于或等于 $18.00 的订单。

第一个语句中的 >= 运算符隐式地调用 greaterthanorequal( ) 运算符函数：

```
SELECT order_num FROM orders
WHERE ship_charge >= 18.00;
SELECT order_num FROM orders
WHERE greaterthanorequal(ship_charge, 18.00);
```

对于所有内建的数据类型，数据库服务器提供与关系运算符相关联的运算符函数。当您开发用户定义的数据类型时，您必须为那种数据类型定义运算符函数，以便用户能对该类型使用关系运算符。

如果您为用户定义的类型定义 **lessthan( )**、**greaterthan( )** 以及其他运算符函数，则您还应定义 **compare( )**。类似地，如果您定义 **compare( )**，则您还应定义 **lessthan( )**、**greaterthan( )** 和其他运算符函数。必须以一致的方式定义所有这些函数，以避免当在 SELECT 的 WHERE 子句中比较 UDT 值时产生不正确的查询结果的可能性。

## U.S. English 数据的排序顺序

如果您正在使用缺省的语言环境（U.S. English），则当数据库服务器比较关系运算符之前与之后的字符表达式时，它使用缺省的代码集的代码集顺序。

在 UNIX™ 上，缺省的代码集是 ISO8859-1 代码集，由下列字符集组成：

- 取值范围在 0 至 127 的代码点的 ASCII 字符。

此范围包含控制字符、标点符号、英语字符和数字。

- 取值范围在 128 至 255 的代码点的 8-bit 字符。

  此范围包括许多非英语字符（诸如 é、∞、Ö 和 ñ）和符号（诸如 £、© 和 ¿）。

在 Windows™ 中，缺省的代码集是 Microsoft™ 1252。此代码集既包括 ASCII 代码集，也包括一组 8-bit 字符。

此表格罗列 ASCII 代码集。**编号**列展示 ASCII 代码点编号，**字符** 列显示对应的 ASCII 字符。在缺省的语言环境中，根据他们的代码集顺序对 ASCII 字符排序。因此， 小写字母跟在大写字母之后，且都跟在数字之后。在此表格中，ASCII 32 是空字符，插入号（^）代表 CTRL 键。例如，^X 意味着 CONTROL-X。

| 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ^@ | 20 | ^T | 40 | ( | 60 | < | 80 | P | 100 | d | 120 | x |
| 1 | ^A | 21 | ^U | 41 | ) | 61 | = | 81 | Q | 101 | e | 121 | y |
| 2 | ^B | 22 | ^V | 42 | * | 62 | > | 82 | R | 102 | f | 122 | z |
| 3 | ^C | 23 | ^W | 43 | + | 63 | ? | 83 | S | 103 | g | 123 | { |
| 4 | ^D | 24 | ^X | 44 | , | 64 | @ | 84 | T | 104 | h | 124 | \| |
| 5 | ^E | 25 | ^Y | 45 | - | 65 | A | 85 | U | 105 | i | 125 | } |
| 6 | ^F | 26 | ^Z | 46 | . | 66 | B | 86 | V | 106 | j | 126 | ~ |
| 7 | ^G | 27 | esc | 47 | / | 67 | C | 87 | W | 107 | k | 127 | del |
| 8 | ^H | 28 | ^\ | 48 | 0 | 68 | D | 88 | X | 108 | l | | |
| 9 | ^I | 29 | ^] | 49 | 1 | 69 | E | 89 | Y | 109 | m | | |
| 10 | ^J | 30 | ^^ | 50 | 2 | 70 | F | 90 | Z | 110 | n | | |
| 11 | ^K | 31 | ^_ | 51 | 3 | 71 | G | 91 | [ | 111 | o | | |
| 12 | ^L | 32 | | 52 | 4 | 72 | H | 92 | \ | 112 | p | | |
| 13 | ^M | 33 | ! | 53 | 5 | 73 | I | 93 | ] | 113 | q | | |
| 14 | ^N | 34 | ″ | 54 | 6 | 74 | J | 94 | ^ | 114 | r | | |
| 15 | ^O | 35 | # | 55 | 7 | 75 | K | 95 | _ | 115 | s | | |
| 16 | ^P | 36 | $ | 56 | 8 | 76 | L | 96 | ` | 116 | t | | |
| 17 | ^Q | 37 | % | 57 | 9 | 77 | M | 97 | a | 117 | u | | |
| 18 | ^R | 38 | & | 58 | : | 78 | N | 98 | b | 118 | v | | |
| 19 | ^S | 39 | ’ | 59 | ; | 79 | O | 99 | c | 119 | w | | |

## 对非缺省的代码集（GLS）中 ASCII 字符的支持

大多数非缺省的语言环境的代码集（称为***非缺省的代码集***）支持 ASCII 字符。在非缺省的语言环境中，对于 CHAR 和 VARCHAR 表达式中的 ASCII 数据，如果该代码集支持这些 ASCII 字符，则数据库服务器使用 ASCII 代码集顺序。然而，如果当前的排序顺序（如由 **DB_LOCALE** 或由 SET COLLATION 指定的那样）支持本地化的排序顺序，则当数据库服务器对 NCHAR 或 NVARCHAR 值排序时，使用本地化的顺序。

## 作为运算对象的精确数值

如果您指定作为一个运算对象的精确数值，其正好可表示由关系运算符进行比较的另一值的数据类型，则您会获得意外的结果。例如，由于舍入错误，如果一个运算对象返回 FLOAT 值，而其他返回 INTEGER，则像 = 或 **equals( )** 运算符函数一样的关系运算符不可返回 TRUE。要获取关于哪些内建的数据类型存储完全表示为精确数值值的信息，请参阅 精确数值 部分。

# 5. 其它语法段

本主题描述 ***syntax segments***，它们是语言元素，例如数据库对象名称或优化程序伪指令，它们在某些 SQL 或 SPL 语句的语法图中作为子图引用显示。

大多数只能在一个语句中出现的段在 SQL 语句或语句描述中的 SPL 语句中描述。然而，为了清楚、易于使用及综合处理的原因，可出现在不同 SQL 或 SPL 语句中的以及非数据类型、非表达式的大多数段在本部分中进行了单独讨论。

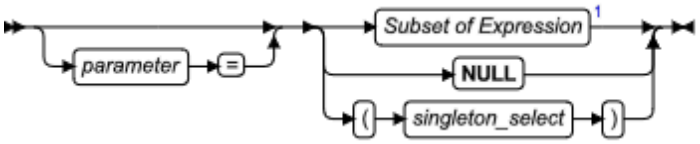
前一章描述指定数据类型和表达式的语法段。本章描述不是数据类型、表达式和完整 SQL 语句或 SPL 语句的其它语法段。在出现在本手册的第二章以及其它章节中的各种语法图中引用到这些段。

## 5.1 参数

使用参数段向例程传递一个特定值作为输入。无论何处，只要您在语法图中看到对 ***argument*** 的引用，就请使用本段。

**语法**

参数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *parameter* | 由您指定它的值的参数 | 必须和 CREATE FUNCTION 或 CREATE PROCEDURE 语句声明的名称相匹配 | 标识符 |
| *singleton_select* | 返回单值的内嵌查询 | 必须返回一个与 *parameter* 兼容的数据类型和长度的值 | SELECT 语句 |

**用法**

CREATE PROCEDURE 或 CREATE FUNCTION 语句可以为 UDR 定义一个参数列表。如果参数列表非空，则调用 UDR 时必须输入参量。参量是一个特定的值，它的数据类型兼容相应的 UDR 参数。

当执行一个 UDR 时，可以任选下面两种方法之一输入参量：

- 和参数名一起（以 ***parameter name = expression)*** 的形式）输入，即使参量和参数的顺序不同

- 按照位置输入，不加 ***parameter*** 名称，每个 ***expression*** 的顺序和参量对应的参数相同。（这有时称为 ***ordinal*** 格式。）

在例程的一次单个调用中不能混合使用这两种方法来指定参量。例如：如果为一个参量指定了一个参数名，那么必须对所有参量都使用参数名。

在以下示例中，用户定义的过程要求三个字符参量 t 、d 和 n，两个语句都是有效的：

```
EXECUTE PROCEDURE add_col (t ='customer', d ='integer', n ='newint');
EXECUTE PROCEDURE add_col ('customer','newint','integer') ;
```

## 比较参量和参数列表

当用 CREATE PROCEDURE 或 CREATE FUNCTION 创建或注册一个 UDR 时，用 UDR 要求的参数名和数据类型声明一个***参数列表***。（对于用 C 或 Java™ 语言编写的外部例程，参数名是可选的。）有关声明参数的详细信息，请参阅例程参数列表。

如果不同的例程具有相同的标识符但是声明的参数个数不同，用户定义的例程可以是***过载的***。有关重载的更多信息，请参阅 例程重载以及例程签名。

如果试图以超出 UDR 要求的参量来执行 UDR ，就会接收到一条错误消息。

如果已少于 UDR 要求的参量来调用 UDR，那么就可以将省略的参量说成***缺少***。数据库服务器将缺少的参量初始化为它们相应的缺省值。这个初始化过程发生在 UDR 主体中第一条可执行语句之前。

如果缺少的参量没有缺省值。则 GBase 8s 发出错误。

已命名的参数不能用于调用在它们的例程特征符中重载数据类型的 UDR。只有例程的特征符具有不同数量的参数时，命名的参数用于解析非唯一的例程名称才是有效的：

```
func( x::integer, y );      -- VALID if only these 2 routines
func( x::integer, y, z );   -- 具有相同的 'func' 标识

func( x::integer, y );      -- NOT VALID if both routines have
func( x::float, y) ;         -- 相同的标识和两个参数
```

对于顺序和命名的参数，如果两个或更多的 UDR 特征符具有多个数目的缺省值，则会执行参数最少的例程：

```
func( x, y default 1 )
func( x, y default 1, z default 2 )
```

如果两个同时称作 func 的已注册的 UDR 具有以上所示的特征符，那么语句 EXECUTE func(100) 调用 func(100, 1)。

不能使用命名的参数提供缺省值的子集，除非这些参数遵循例程特征符的位置顺序。即，不能跳过一些参量而依赖于数据库服务器提供它们的缺省值。

例如，给定以下特征符：

```
func( x, y default 1, z default 2 )
```

可执行：

```
func( x=1, y=3 )
```

但不能执行：

```
func( x=1, z=3 )
```

## 表达式的子集作为参量有效

参数的语法图涉及这一节的内容。

除了聚集函数以外，可以使用任何表达式作为参量。如果使用子查询或函数调用作为参量，那么子查询或函数必须返回适当数据类型和大小的单个值。有关 SQL 表达式的用途和语法，请参阅表达式。

## 远程数据库中的 UDR 参量

远程数据库中的 UDR 参数在大多数上下文中，UDR 在跨数据库和跨服务器分布式操作中有效，但每个参与的数据库必须具有相同的日志记录模式。

除了 BIGSERIAL 、BYTE 、SERIAL 、SERIAL8 和 TEXT，可作为跨服务器的 UDR 的参数的有效的数据类型 包括分布式查询中的数据类型中列出的不透明的内置 SQL 数据类型和这些附加内置透明和 DISTINCT 数据类型：

- BOOLEAN
- LVARCHAR
- 不透明的内置类型的 DISTINCT
- BOOLEAN DISTINCT
- LVARCHAR DISTINCT
- 以上列出的 DISTINCT 类型的 DISTINCT

如果 UDR 是在所有参与的数据库中定义 ，则这些数据类型可以是 SPL 、C 或 Java™ 语言的 UDR 的参数。在这些数据类型上定义的任何隐式或显式转换必须在所有参与的 GBase 8s 实例之间复制。DISTINCT 数据类型必须在参与分布式查询的所有数据库中定义完全相同的数据类型层次结构。

相同的数据类型在调用相同的 GBase 8s 实例的其它数据库中的 UDR 时也作为参数有效，以及以下其它类型的参数：

- BLOB
- CLOB
- 您显式转换为内置类型的 UDT

所有的 UDR 、UDT 、DISTINCT 数据类型 、DISTINCT 类型层次结构、转型和转型函数必须在所有参与的数据库中注册。有关分布式操作中的 DISTINCT 类型的更多信息，请参阅 分布式操作中的 DISTINCT 类型。

## 新增 SQL 支持?作为占位符

在 SQL 语句中，可以使用问号?作为占位符，例如：

```
INSERT INTO t1 VALUES (?,?,?)
```

客户程序中可以使用 GCI 绑定变量传参或者在 JDBC 中使用 PreparedStatement 对象调用相关 set 方法传参。本次扩展支持以下几种方式：

**SELECT 语句投影列支持?作为占位符**

SELECT ?, ? FROM t1

下面是一个包括了 5 个使用问号?作为占位符的 MERGE 语句的例子：

```
MERGE INTO T1
USING (SELECT ? AS a2, ? AS b2 FROM t2) T2 ON (T1.a1=T2.a2)
WHEN MATCHED THEN UPDATE SET T1.v1=?
WHEN NOT MATCHED THEN INSERT (a1, b1) VALUES(?, ?)
```

**投影列的表达式支持?作为占位符**

函数参数中可以支持，例如：

SELECT tan(?) FROM t1

运算表达式中可以支持，例如：

SELECT ?+? FROM t1

**SELECT 语句的分页语法参数支持?作为占位符**

分页语法包括 SKIP、FIRST、LIMIT、OFFSET、TOP 等选项，例如：

```
SELECT a1, a2 FROM t1 LIMIT ? OFFSET ?
SELECT a1, a2 FROM t1 TOP ?,?
```

**子查询支持?作为占位符**

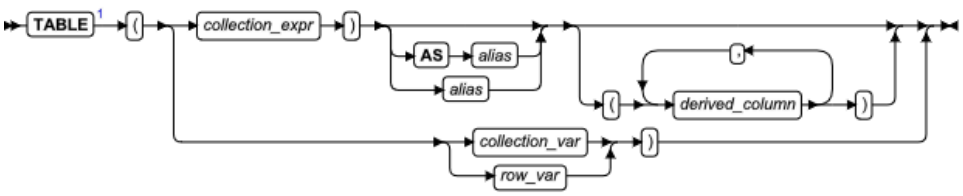
SELECT tabname FROM (SELECT tabname FROM systables WHERE tabname LIKE ?)

## 5.2 集合派生表

***集合派生表***是一个虚拟表，其中表行中的值等价于集合的各个元素。当您在语法图中看到对集合派生表的引用时，使用本段。该语法是 SQL ANSI/ISO 标准的扩展。

**语法**

集合派生表




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|

| | | | |
|---|---|---|---|
| *alias* | 作用域与 SELECT 语句的集合派生表的临时名称。缺省值取决于实现。 | 如果存在潜在的多义性，必须把 *alias* 放在关键字 AS 前面。请参阅 AS 关键字。 | 标识符 |
| *collection_expr* | 任何对单个集合的元素求值的表达式 | 请参阅集合表达式格式的限制. | 表达式 |
| *collection_var, row_var* | 已归类或未归类的集合变量名，或者包含集合派生表的 GBase 8s ESQL/C **row** 变量的名称 | 必须在 GBase 8s ESQL/C 程序中或（对 *collection_var*）在 OSPL 策略中已声明 | 请参阅 *GBase 8s ESQL/C 程序员手册* 或 DEFINE. |
| *derived _column* | 表中派生的列的临时名称 | 如果基础集合不是 ROW 数据类型，那么只能指定一个派生列名 | 标识符 |

**用法**

集合派生表可以出现在 UPDATEA 语句、SELECT 或 DELETE 语句的 FROM 子句或 INSERT 的 INTO 子句中 ***table*** 名称有效的地方。

使用集合派生表段完成这些任务：

- 如同访问表行一样，访问集合元素。
- 指定要访问的集合变量，而不是表名。
- 指定要访问的 ESQL/C **row** 变量，而不是表名。

TABLE 关键字把一个集合转换成虚拟表。可以使用集合表达式格式来查询集合的列，或则使用 **collection** 变量或 **row** 变量格式来操纵集合列中的数据。

## 通过虚拟表访问集合

当使用集合派生表段的集合表达式格式来访问集合元素时，可以通过虚拟表之间选择集合元素。只能在 SELECT 语句的 FROM 子句中使用这种格式。FROM 子句可在查询或子查询中。

用这种格式可以使用连接、聚集、WHERE 子句、表达式、ORDER BY 子句以及在使用集合变量格式时不可用的其它操作。这种格式减少了对多个游标和临时表的需求。

可能的集合表达式示例包括列的引用、标量子查询、点表达式、函数、运算符（通过重载）、集合子查询、文字集合、集合构造函数和强制转型函数等。

以下示例在 FROM 子句中使用 SELECT 语句，其结果集定义由第五十一到第七十限定行组成的虚拟表，并按照 employee_id 列值排序。

```
SELECT * FROM TABLE(MULTISET(SELECT SKIP 50 FIRST 20 * FROM employees ORDER BY employee_id)) vt(x,y), tab2 WHERE tab2.id = vt.x;
```

以下示例使用连接查询创建不超过 20 行（从第 41 行开始）的虚拟表，并按照集合派生表的 salary 列的值排序：

```
SELECT emp_id, emp_name, emp_salary
        FROM   TABLE(MULTISET(SELECT SKIP 40 LIMIT 20 id, name, salary   FROM e1, e2
        WHERE e1.id = e2.id ORDER BY salary ))
        AS etab(emp_id, emp_name, emp_salary);
```

## FROM 子句中的 Table 表达式

GBase 8s 支持 SELECT 查询和子查询的 FROM 子句中的表表达式的 ANSI/ISO 标准语法，作为 GBase 8s 扩展集合派生表语法的替代。在版本 10.00 和更早版本的需要关键字 TABLE 和 MULTISET 。支持用于 SQL 的 ANSI/ISO 标准的这些扩展，但不再需要在 SELECT 语句的 FROM 子句中的集合派生表规范。

以下两个查询返回相同的结果集，但只要第二个查询符合 ANSI/ISO 标准：

```
SELECT * FROM TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 100))    AS
vtab(c1), (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1) ORDER BY c1;

SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
        (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1)
        ORDER BY c1;
```

同一个 SELECT 语句可以合并派生表的 GBase 8s 扩展和 ANSI/ISO 语法的示例：

```
SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
        TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 10)) AS vtab1(vc1)
                ORDER BY c1;
```

子查询必须以两种格式的圆括号分隔，但紧跟在 TABLE 关键字后面并包含 MULTISET 集合子查询规范的外部圆括号（( ) ）是 ANSI/ISO 语法的扩展。此 ANSI/ISO 语法仅在 SELECT 语句的 FROM 子句中有效。在任何其它上下文中，您不能从集合子查询规范中省略这些关键字和括号。

## 集合表达式格式的限制

当使用集合表达式格式时，有一定的限制：

- 集合派生表是只读的。
  - 它不能成为 INSERT 、UPDATE 或 DELETE 语句的目标。

    要执行 INSERT 、UPDATE 或 DELETE 操作，必须使用集合变量格式。
  - 它不能成为可更新游标或视图的基础表。
- 在 SELECT 语句的 FROM 子句中，SPL 的 CALL 关键字不能优先于表表达式的 TABLE 关键字。
- 如果集合是 LIST 数据类型，那么由此得到的集合派生表不会保存 LIST 中元素的顺序。
- 基础集合表达式求值不能等于 NULL。
- 集合表达式不能包含对远程数据库服务器上集合的引用。

- 集合表达式不能包含对出现在同一 FROM 子句的表的列引用。也就是说，集合派生表必须独立于 FROM 子句中的其它表。

  例如，下面的语句返回了一个错误值。因为集合派生表 TABLE (parents.children)，引用了表 **parents**，这个表也在 FROM 子句中引用：

```
SELECT COUNT(*) FROM parents,
                TABLE(parents.children) c_table
                WHERE parents.id = 1001;
```

  要克服此限制，您可能需要写一个在 Projection 子句中包含子查询的查询：

```
SELECT (SELECT COUNT(*)
            FROM TABLE(parents.children) c_table)
            FROM parents WHERE parents.id = 1001;
```

## 适用于 ESQL/C 的附加限制

除了前面描述的限制以外，在对 GBase 8s ESQL/C 使用集合表达式格式时还有下列限制：

- 不能将未归类的 COLLECTION 指定为主变量数据类型。
- 不是使用 TABLE(?) 这种格式。

  基本集合变量的数据类型必须是静态确定的。要克制这个限制，可以把变量显式地强制转型成数据库服务器认可的已归类 Collection 数据类型（SET 、MULTISET 或 LIST）。例如：

```
TABLE(CAST(? AS type))
```

- 不是使用 TABLE(:hostvar) 这种格式。

  要克服这个限制，必须把变量显式地强制转型成数据库服务器的已归类 Collection 数据类型（SET 、MULTISET 或 LIST），例如：

```
TABLE(CAST(:hostvar AS type))
```

### 产生集合派生表的 Row 类型

如果没有指定派生列名，那么数据库服务器的行为取决于基本集合元素的数据类型。

虽然集合派生表看起来包含单独的数据类型的列，但这些列实际上是 ROW 数据类型的字段。ROW 类型的数据类型和列名取决于几个元素。

如果基础集合表达式元素的数据类型是 ***type***，那么数据库服务器通过以下规则来确定集合派生表的 ROW 类型：

- 如果 ***type*** 是 ROW 数据类型，而且没有指定派生列的列表，那么集合派生表的 ROW 类型就是 ***type***。
- 如果 ***type*** 是 ROW 数据类型，而且指定了派生列的列表，那么集合派生表的 ROW 类型就是未命名的 ROW 类型，其列数据类型和 ***type*** 相同，列名从派生列的列表中获取。
- 如果 ***type*** 不是 ROW 数据类型，那么集合派生表的 ROW 类型就是未命名的 ROW 类型，它包含一个 ***type*** 列，并且名称指定在派生列的列表中。如果不能指定名称，数据库服务器会为列分配一个取决于实现的名称。

下表给出的引申示例举例说明了这些规则。该表使用以下模式为例：

```
CREATE ROW TYPE person (name CHAR(255), id INT);
        CREATE TABLE parents
        (
        name CHAR(255),
        id INT,
        children LIST (person NOT NULL)
        );
        CREATE TABLE parents2
        (
        name CHAR(255),
        id INT,
        children_ids LIST (INT NOT NULL)
        );
```

| ROW 类型 | 显式派生列列表 | 集合派生表产生的 ROW 类型 | 代码示例 |
|---|---|---|---|
| 是 | 否 | *Type* | SELECT (SELECT c_table.name FROM TABLE(parents.children) c_table WHERE c_table.id = 1002) FROM parents WHERE parents.id = 1001;<br>在此示例中，c_table 的 ROW 类型是 parents。 |
| 是 | 是 | 未命名的 ROW 类型，它的列类型是 *Type* ，而列名是派生列的列表中的名称 | SELECT (SELECT c_table.c_name FROM TABLE(parents.children) c_table(c._name, c_id) WHERE c_table.c_id = 1002) FROM parents WHERE parents.id = 1001;<br>在此示例中，c_table 的 ROW 类型是 ROW(c_name CHAR(255), c_id INT)。 |
| No | No | 未命名的 ROW ，它包含一个已指定依实现而定的名称的 *Type* 列 | 在以下示例中，如果不指定 c_id，数据库服务器会为派生列指定一个名称。在这种情况下，表 c_table 的 ROW 类型是 ROW(*server_defined_name* INT)。 |
| 否 | 是 | 未命名的 ROW 类型，它包含一个 *Type* 列。列名在派生列的列表中 | SELECT(SELECT c_table.c_id FROM TABLE(parents2.child_ids) c_table (c_id) WHERE c_table.c_id = 1002) FROM parents WHERE parents.id = 1001;<br>这里，c_table 的 ROW 类型是 ROW(c_id INT)。 |

下面的程序分段用返回单个值的 SPL 函数创建了一个集合派生表：

```
CREATE TABLE wanted(person_id int);
        CREATE FUNCTION
```

```
wanted_person_count (person_set SET(person NOT NULL))
RETURNS INT;
RETURN( SELECT COUNT (*)
FROM TABLE (person_set) c_table, wanted
WHERE c_tabel.id = wanted.person_id);
END FUNCTION;
```

下面的程序段给出了用返回多个值的 SPL 函数创建一个集合派生表的更通用的示例：

```
-- Table of categories and child categories,
-- allowing any number of levels of subcategories
    CREATE TABLE CategoryChild (
    categoryId          INTEGER,
    childCategoryId SMALLINT
        );

        INSERT INTO CategoryChild VALUES (1, 2);
        INSERT INTO CategoryChild VALUES (1, 3);
        INSERT INTO CategoryChild VALUES (1, 4);
        INSERT INTO CategoryChild VALUES (2, 5);
        INSERT INTO CategoryChild VALUES (2, 6);
        INSERT INTO CategoryChild VALUES (5, 7);
        INSERT INTO CategoryChild VALUES (7, 8);
        INSERT INTO CategoryChild VALUES (7, 9);
        INSERT INTO CategoryChild VALUES (4, 10);

        -- "R" == ROW type
        CREATE ROW TYPE categoryLevelR (
        categoryId          INTEGER,
        level     SMALLINT );

        -- DROP FUNCTION categoryDescendants (
        --                      INTEGER, SMALLINT );
        CREATE FUNCTION categoryDescendants (
        pCategoryId INTEGER,
        pLevel          SMALLINT DEFAULT 0 )
        RETURNS MULTISET (categoryLevelR NOT NULL)

        -- "p" == Prefix for Parameter names
        -- "l" == Prefix for Local variable names
        DEFINE lCategoryId LIKE CategoryChild.categoryId;
        DEFINE lRetSet MULTISET (categoryLevelR NOT NULL);
        DEFINE lCatRow categoryLevelR;
```

```
        -- TRACE ON;
        -- Must initialize collection before inserting rows
        LET lRetSet = 'MULTISET{}' :: MULTISET (categoryLevelR NOT NULL);
FOREACH
            SELECT childCategoryId INTO lCategoryId
            FROM CategoryChild WHERE categoryId = pCategoryId;
            INSERT INTO TABLE (lRetSet)
            VALUES (ROW (lCategoryId, pLevel+1)::categoryLevelR);

            -- INSERT INTO TABLE (lRetSet);
            -- EXECUTE FUNCTION categoryDescendantsR ( lCategoryId,
            -- pLevel+1 );
            -- Need to iterate over results and insert into SET.
            -- See the SQL Tutorial, pg. 10-52:
            -- "Tip: You can only insert one value at a time
            -- into a simple collection."
            FOREACH
            EXECUTE FUNCTION categoryDescendantsR ( lCategoryId, pLevel+1 )
            INTO lCatRow;
            INSERT INTO TABLE (lRetSet)
            VALUES (lCatRow);
            END FOREACH;
            END FOREACH;

            RETURN lRetSet;
            END FUNCTION
            ;
            -- "R" == recursive
            -- DROP FUNCTION categoryDescendantsR (INTEGER, SMALLINT);
            CREATE FUNCTION categoryDescendantsR (
            pCategoryId INTEGER,
            pLevel      SMALLINT DEFAULT 0
            )
            RETURNS categoryLevelR;
            DEFINE lCategoryId      LIKE CategoryChild.categoryId;
            DEFINE lCatRow            categoryLevelR;

            FOREACH
            SELECT   childCategoryId
            INTO     lCategoryId
            FROM     CategoryChild
            WHERE    categoryId = pCategoryId
            RETURN ROW (lCategoryId, pLevel+1)::categoryLevelR WITH RESUME;
```

```
FOREACH
EXECUTE FUNCTION categoryDescendantsR ( lCategoryId, pLevel+1 )
INTO      lCatRow
RETURN    lCatRow WITH RESUME;
END FOREACH;
END FOREACH;
END FUNCTION;

-- Test the functions:
SELECT lev, col
FROM      TABLE ((
categoryDescendants (1, 0)
)) AS CD (col, lev);
```

## 通过集合变量访问集合

当使用集合派生表段的集合变量格式时，将使用主变量或程序变量来访问和操纵集合元素。这种格式允许修改变量的内容（如同您对数据库中的表一样）。然后使用 **collection** 变量的内容更新实际表。

可以使用集合变量格式（关键字 TABLE 在 collection 变量的前面）代替下列 SQL 语句中（或在 SPL 的 FOREACH 语句中）的表名、同义词名或视图名：

- SELECT 语句的 FROM 子句（用于访问 **collection** 变量的元素）
- INSERT 语句的 INTO 子句（用于向 **collection** 变量添加新元素）
- DELETE 语句（用于从 **collection** 变量除去元素）
- UPDATE 语句（用于修改 **collection** 变量中的现有元素）
- DECLARE 语句（用于声明 Select 或 Insert 游标来访问 GBase 8s ESQL/C **collection** 主变量的多个元素）
- FETCH 语句（用于检索与 Select 游标相关联的 **collection** 主变量中的单个元素）
- PUT 语句（用于检索与 Insert 游标相关联的 **collection** 主变量中的单个元素）
- FOREACH 语句（用于声明一个游标来访问 SPL 集合变量的多个元素以及检索此 **collection** 主变量中的单个元素）

## 使用集合变量操纵集合元素

当使用 GBase 8s 的数据操纵语句（SELECT 、INSERT 、UPDATE 或 DELETE）和 **collection** 变量一起使用时，您可以修改集合中的一个和多个元素。

### 修改集合中的约束

1. 在 SPL 例程或 GBase 8s ESQL/C 程序中创建一个 **collection** 变量。

有关如何在 GBase 8s ESQL/C 中声明 **collection** 集合变量的信息，请参阅 *GBase 8s ESQL/C 程序员手册*。有关如何在 SPL 中定义 **COLLECTION** 变量，请参阅 DEFINE。

2. 在 GBase 8s ESQL/C 中，为集合分配内存；请参阅 ALLOCATE COLLECTION 语句。
3. 可选地，使用 SELECT 语句将一个 COLLECTION 列选择到 **collection** 变量中。

   如果变量是一个为归类的 COLLECTION 变量，那么在集合派生表段中使用该变量之前，必须从 COLLECTION 列执行 SELECT。SELECT 语句允许数据库服务器获取集合数据类型。
4. 使用适当的带集合派生表段的数据操纵语句在集合变量中添加、删除或修改元素。

   要在集合中插入多个元素或删除一个***指定***的元素，必须对集合变量使用游标。

   - 有关如何在 ESQL/C 中使用更新游标的更多信息，请参阅 DECLARE 语句。
   - 关如何在 SPL 中使用更新游标的更多信息，请参阅 FOREACH。
5. 集合变量具有正确的元素以后，对包含实际集合列的表或视图使用 INSERT 或 UPDATE 语句，来保存集合变量的更改。

   - 使用 UPDATE，在 SET 子句中指定集合变量。
   - 使用 INSERT，在 VALUES 子句中指定集合变量。

集合变量存储集合的元素。不过，它与数据库列之间没有内在连接。一旦集合变量包含了正确的元素，那么必须用 INSERT 或 UPDATE 语句把变量保存到表的实际集合列中。

## 从 ESQL/C 中的集合执行删除操作的示例

假设把表 table1 某一行的 set_col 列定义为 SET ，并且有一行的值为 {1,8,4,5,2}。下面的 GBase 8s ESQL/C 代码段使用更新游标以及带有 WHERE CURRENT OF 子句的 DELETE 语句来删除值为 4 的元素：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(smallint not null) a_set;
        int an_int;
        EXEC SQL END DECLARE SECTION;
        ...
        EXEC SQL allocate collection :a_set;
        EXEC SQL select set_col into :a_set from table1 where int_col = 6;
        EXEC SQL declare set_curs cursor for
        select * from table(:a_set) for update;

        EXEC SQL open set_curs;
        while (i<coll_size)
        {
        EXEC SQL fetch set_curs into :an_int;
        if (an_int = 4)
        {
```

```
            EXEC SQL delete from table(:a_set) where current of set_curs;
            break;
            }
            i++;
            }
EXEC SQL update table1 set set_col = :a_set
              where int_col = 6;
              EXEC SQL deallocate collection :a_set;
              EXEC SQL close set_curs;
              EXEC SQL free set_curs;
```

执行 DELETE 语句以后，集合变量包含元素 {1,8,5,2}。位于代码段末尾的 UPDATE 语句把修改后的集合保存到 **set_col** 列中。如果没有 UPDATE 语句，集合列中的元素 4 就没有删除。

## 从集合中执行删除操作的示例

假设把表 table1 某一行的 set_col 列定义为 SET ，并且有一行的值为 {1,8,4,5,2}。下面的 SPL 代码段使用 FOREACH 循环和带有 WHERE CURRENT OF 子句的 DELETE 语句来删除值为 4 的元素：

```
CREATE_PROCEDURE test6()

            DEFINE a SMALLINT;
            DEFINE b SET(SMALLINT NOT NULL);
            SELECT set_col INTO b FROM table1
            WHERE id = 6;
            -- Select the set in one row from the table
            -- into a collection variable
            FOREACH cursor1 FOR
            SELECT * INTO a FROM TABLE(b);
            -- Select each element one at a time from
            -- the collection derived table b into a
            IF a = 4 THEN
            DELETE FROM TABLE(b)
            WHERE CURRENT OF cursor1;
            -- Delete the element if it has the value 4
            EXIT FOREACH;
            END IF;
            END FOREACH;

            UPDATE table1 SET set_col = b
            WHERE id = 6;
            -- Update the base table with the new collection

            END PROCEDURE;
```

此 SPL 例程声明了两个 SET 变量，**a** 和 **b**，每一个都具有一组 SMALLINT 值。第一个 SELECT 语句把 **table1** 一行中的 SET 列复制到变量 **b**。然后例程声明了一个名为 **cursor1** 的游标，它一次把一个元素从 **b** 复制到 SET 变量 **a**。当游标位于值为 4 的元素时，DELETE 语句就从 SET 变量 **b** 中删除这个元素。最后，UPDATE 语句把表 **table1** 的这一行用存储在变量 **b** 中的新集合替代。

有关如何在 SPL 例程中使用集合变量的信息，请参阅 *GBase 8s SQL 教程指南*。

## 更新集合的示例

假设把表 table1 某一行的 set_col 列定义为 SET ，并且有一行的值为 {1,8,4,5,2}。下面的 GBase 8s ESQL/C 程序把值为 4 的元素改为 10：

```
main
    {
    EXEC SQL BEGIN DECLARE SECTION;
    int a;
    collection b;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL allocate collection :b;
    EXEC SQL select set_col into :b from table1
    where int_col = 6;

    EXEC SQL declare set_curs cursor for
    select * from table(:b) for update;
    EXEC SQL open set_curs;
    while (SQLCODE != SQLNOTFOUND)
    {
    EXEC SQL fetch set_curs into :a;
    if (a = 4)
    {
    EXEC SQL update table(:b)(x)
    set x = 10 where current of set_curs;
    break;
    }
    }
    EXEC SQL update table1 set set_col = :b
    where int_col = 6;
    EXEC SQL deallocate collection :b;
    EXEC SQL close set_curs;
    EXEC SQL free set_curs;
    }
```

执行这个 GBase 8s ESQL/C 程序以后，表 **table1** 的 **set_col** 列具有值 {1,8,10,5,2}。

这个 GBase 8s ESQL/C 程序定义了两个 **collection** 变量，**a** 和 **b**，并且从 **table1** 中选择了一个 SET 到 **b**.。WHERE 子句确保只返回一行。然后程序定义一个集合游标，它从 **b** 中一次选择一个元素到 **a**。当程序找到值为 4 的元素时，第一个 UPDATE 语句把该元素的值改为 10 并退出循环。

在第一个 UPDATE 语句中，**x** 是一个派生列名，用来在集合派生表中更新当前元素。第二个 UPDATE 语句用新的集合更新基表 **table1**。

有关如何在 GBase 8s ESQL/C 中使用 **collection** 主变量的信息，请参阅 *GBase 8s ESQL/C 程序员手册* 中关于复杂数据类型的讨论。

## 在多重集合中插入值的示例

假定 GBase 8s ESQL/C 主变量 a_multiset 有下列声明：

```
EXEC SQL BEGIN DECLARE SECTION;
client collection multiset(integer not null) a_multiset;
EXEC SQL END DECLARE SECTION;
```

下面的 INSERT 语句把一个新的 MULTISET 元素 142,323 添加到 a_multiset：

```
EXEC SQL allocate collection :a_multiset;
        EXEC SQL select multiset_col into :a_multiset from table1
        where id = 107;
        EXEC SQL insert into table(:a_multiset) values (142323);
        EXEC SQL update table1 set multiset_col = :a_multiset
        where id = 107;
EXEC SQL deallocate collection :a_multiset;
```

当要把元素插入到 **client-collection** 变量中时，不能在 INSERT 语句的 VALUES 子句中指定 SELECT 语句或 EXECUTE FUNCTION 语句。然而，当要把元素插入到 **server-collection** 变量中时，SELECT 和 EXECUTE FUNCTION 语句在 VALUES 子句中是有效的。有关 **client-** 和 **server-collection** 变量的更多信息，请参阅 *GBase 8s ESQL/C 程序员手册*。

## 访问嵌套集合

如果集合的元素本身就是复杂类型（collection 或 row 类型），那么这个集合就是一个***嵌套集合***。例如，假设 GBase 8s ESQL/C collection 变量 a_set 是一个嵌套集合，定义如下：

```
EXEC SQL BEGIN DECLARE SECTION;
client collection set(list(integer not null)) a_set;
client collection list(integer not null) a_list;
int an_int;
EXEC SQL END DECLARE SECTION;
```

要访问一个嵌套集合的元素（或字段），可以使用匹配元素类型（前面代码段中的 **a_list** 和 **an_int**）的 **collection** 或 **row** 变量和 Select 游标。

## 访问 Row 变量

TABLE 关键字可以把 GBase 8s ESQL/C **row** 变量变成集合派生表。也就是说，一行在 SQL 语句中作为表出现。对于**行**变量，把集合派生表看作只有一行的表，**行**类型的每个字段就是行的一列。使用关键字 TABLE 在这些 SQL 语句中代替表、同义词或视图的名称：

- SELECT 语句的 FROM 子句（用来访问 **row** 变量的一个字段）
- UPDATE 语句（用来修改 **row** 变量中已有的字段）

DELETE 和 INSERT 语句不支持集合派生表段的 **row** 变量。

例如，假设 ESQL/C 主变量 a_row 有以下声明：

```
EXEC SQL BEGIN DECLARE SECTION;
row(x int, y int, length float, width float) a_row;
EXEC SQL END DECLARE SECTION;
```

下面的 ESQL/C 代码段把 a_row 变量中的字段添加到表 tab_row 的 row_col 列：

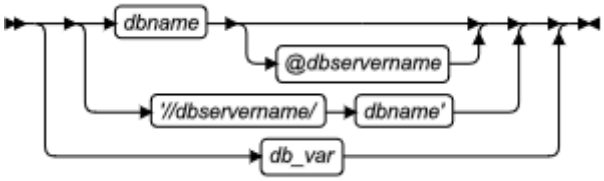
```
EXEC SQL update table(:a_row)
set x=0, y=0, length=10, width=20;
EXEC SQL update rectangles set rect = :a_row;
```

# 5.3 数据库名

使用数据库名段来指定数据库的名称。当看到在语法图中引用数据库名时，使用本段。

### 语法

数据库名




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *dbname* | 数据库名称（不包括路径名和数据库服务器名） | 必须在数据库服务器上的数据库名中是唯一的 | 标识符 |
| *dbservername* | 数据库 *dbname* 驻留的数据库服务器 | 必须存在。不能有空格把 @ 和 *dbservername* 分隔开 | 标识符 |
| *db_var* | 主变量，它的值指定数据库环境 | 变量必须是固定长度的字符数据类型 | 特定于语言 |

### 用法

数据库名称不区分大小写。数据库名不能使用定界标识符。

***dbname*** 和 ***dbservername*** 标识符各自最多可以有 128 字节。

如果数据库服务器的名称是定界标识符或如果它包含大写字母，则数据库服务器不参阅跨服务器分布式 DML 操作。要避免此限制，请在声明数据库服务器的名称或别名时只使用未定界的不包含大写字符的名称。

在非缺省的语言环境中，***dbname*** 可以包含该语言环境代码集中的字母字符。在支持多字节代码集的语言环境中，必须记住数据库名的最大长度是指字节数而不是字符数。有关命名数据库的 GLS 方面的更多信息，请参阅 *GBase 8s GLS 用户指南*。

### 使用关键字作为表名

您可以通过指定数据库服务器名称，选择另一个数据库服务器上的数据库作为您的当前数据库。***dbservername*** 指定的数据库服务器必须与您的 **sqlhosts** 信息中列出的数据库服务器的名称相匹配。

### 使用 @ 符号

@ 符号是一种文字字符。如果指定一个数据库服务器名，@ 符号和数据库服务器名之间的空格是无效的。可以在 ***dbname*** 和 @ 符号之间加一个空格，或者不加空格。

以下示例显示了由数据库服务器名限定的有效数据库规范：

```
empinfo@personnel
empinfo @personnel
```

在这些示例中，**empinfo** 是数据库名，**personnel** 是数据库服务器名。

### 使用路径类型命名法

如果指定一个路径名，在引号、斜线和名称之间不要加空格。以下示例指定一个有效的 UNIX™ 路径名：

```
'//personnel/empinfo'
```

此处 **empinfo** 是 ***dbname***，**personnel** 是数据库服务器名。

### 使用主变量

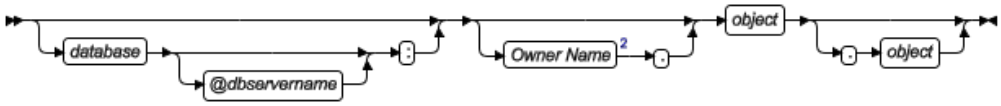
在 GBase 8s ESQL/C 应用程序中可以使用主变量来存储一个代表数据库环境的值。

## 5.4 数据库对象名

使用数据库对象名段来指定数据库对象的名称，例如列、表、视图或用户定义的例程。当看到引用数据库对象名时，使用本段。

**语法**

```
Database Object Name
```




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *database* | *object* 驻留的数据库 | 必须存在 | 数据库名 |
| *dbservername* | *database* 的数据库服务器 | 必须存在。在 @ 后面没有空格。 | 标识符 |
| *object* | 数据库对象的名称 | 请参阅用法 | 标识符 |

**用法**

数据库对象名可以包含限定符和分隔符以指定数据库、服务器、协同服务器（仅对于 XPS）、所有者和（对于某些对象）另一个数据库对象（当前数据库对象是其组成部分）。

GBase 8s 支持使用以冒号（:）—“数据库名:表名”的形式访问指定数据库中的指定表。

例如，以下表达式指定在数据库服务器 butler 的 stores_demo 数据库中由用户 gbasedbt 拥有的 stock 表的 unit-price 列。

```
stores_demo@butler:gbasedbt.stock.unit_price
```

如果创建或重命名一个数据库对象，那么声明的新名称在数据库中相同类型的对象中必须是唯一的。因而，新视图的名称必须在相同数据库内存在的表、视图和序列对象的名称和同义词中是唯一的。（但视图可以和相同服务器的不同数据库中的视图具有相同的名称，或例如和触发器具有相同名称，因为它们是不同类型的对象。）

在兼容 ANSI 的数据库中，***owner.object*** 组合对于对象类型在数据库中必须是唯一的。数据库对象规范必须包含不属于您的数据库对象所有者名称。例如，如果指定了一个不属于您的表，那么也必须指定表的所有者。所有系统目录表的所有者都是 **gbasedbt**。

在 GBase 8s 中，唯一性要求不适用于用户定义的例程（UDR）的名称。有关更多信息，请参阅例程重载以及例程签名。

数据库语言环境代码集中的字符用在数据库对象名中是有效的。有关更多信息，请参阅 *GBase 8s GLS 用户指南*。

## 在外部数据库中指定数据库对象

既可以在本地数据库服务器的外部数据库中，也可以在远程数据库服务器的外部数据库中指定数据库对象。

## 在跨数据库查询中指定数据库对象

要在本地数据库服务器的另一个数据库中指定一个对象，必须使用数据库名（如果外部数据库是符合 ANSI 的，还要加上使用者名称）来限定对象标识符，如此示例所示：

corp_db:hrdirector.executives

在此示例中，外部数据库的名称是 **corp_db**。表所有者的名称是 **hrdirector**。表的名称为 **executives**。这里的冒号分隔符（ **:** ）要跟在***数据库***限定符后面。

在 GBase 8s 中，对本地数据库服务器的其它数据库的查询及其其它数据操纵语言（DML）操纵可以访问在跨数据库事务中的数据类型 中列出的内置不透明数据类型。DML 操作也可以访问能够强制转型为内置类型的用户定义的数据类型（UDT）及基于内置类型的 DISTINCT 类型（如果每个 DISTINCT 类型和 UDT 显式地强制转型为内置类型，并且所有的 DISTINCT 类型、UDT 和强制类型转换在所有参与的数据库中都已定义）。相同的数据类型限制也适用于参量。同时适用于访问本地 GBase 8s 示例的其它数据库的"用户定义的例程"（UDR）的返回值（如果该 UDR 定义在所有参与的数据库中）。

## 在跨服务器查询中指定数据库对象

要在远程数据库服务器的数据库中指定一个对象，必须除数据库对象名以外，还使用指定数据库、数据库服务器和所有者（如果外部数据库是符合 ANSI 的）的 ***fully-qualified identifier*** 。例如，**hr_db@remoteoffice:hrmanager.employees** 是一个标准的表名称。

这里，数据库是 **hr_db**，数据库服务器是 **remoteoffice**，表所有者是 **hrmanager**，表名是 **employees**。在 ***database*** 和 ***database server*** 限定符之间需要不带空格的 at （ **@** ）分隔符。跨服务器查询只能访问不是不透明数据类型的内置数据类型。在跨服务器操作中不能访问 UDT，也不能访问不透明、复杂或其它扩展数据类型。（有关 GBase 8s 在跨服务器操作中支持的 DISTINCT 和内置的 OPAQUE 数据类型的列表，请参阅跨服务器事务中的数据类型。）

在 GBase 8s 中，如果 UDR 在远程数据库服务器上存在，则必须为 UDR 指定一个标准标识符。与跨服务器 DML 操作相似，远程 UDR 的参量、参数和返回值被限制只能针对内置的非不透明数据类型。（有关 GBase 8s 在跨服务器操作中支持的数据类型的列表，请参阅跨数据库事务中的数据类型。）

只能在下列语句中引用远程数据库。有关这些跨本地服务器的数据库或跨数据库服务器的语句中的支持的信息，请参阅 *GBase 8s SQL 教程指南*。

- CREATE DATABASE
- CREATE SYNONYM
- CREATE VIEW
- DATABASE
- DELETE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- INFO
- INSERT
- LOAD
- LOCK TABLE
- SELECT

- UNLOAD
- UNLOCK TABLE
- UPDATE

如果数据库服务器的名称是分隔标识符，或者如果它包含大写字母，则该数据库服务器不能参与分布式 DML 操作。要避免此限制，在声明数据库服务器的名称或别名时，请仅使用不包含大写字母的未限制名称。

### 例程重载以及例程签名

因为例程重载，用户定义的例程的名称对于数据库不必唯一。只要每一个 UDR 的 ***routine signature*** 是不同的，就可以用同一名称定义多个 UDR 。

UDR 是由它们的特征符唯一标识的。 UDR 的签名包括下列几项信息：

- 例程类型（函数或过程）
- 例程的标识符
- 参数的基数、数据类型和顺序
- 在兼容 ANSI 的数据库中的所有者名称

对于任何给定的 UDR，例程签名中至少必须有一项在所有在数据库中注册的 UDR 中是唯一的。

在不兼容 ANSI 的数据库中，除非在 **sysdbopen( )** 和 **sysdbclose( )** 例程的特殊情况下，否则具有不同所有者的两个例程不能拥有相同的签名。有关这些会话配置例程的所有者在定义这些例程的数据库连接或断开连接时影响的信息，请参阅 IFX_REPLACE_MODULE 函数。

### 指定一个已有的 UDR

当引用现有的 UDR 时，如果所使用的名称不能唯一地标识 UDR ，那么您还必须在 UDR 名称后面，以 UDR 创建时声明的相同顺序，指定参数数据类型。然后 GBase 8s 使用例程解析规则来识别 UDR 实例以进行修改、删除或执行。如果在 UDR 创建是已经为它声明了一个名称，则也可以选择指定它的 ***specific name***。具体名称描述在专用名这个部分中。有关例程解析的更多信息，请参阅比较参量和参数列表和 *GBase 8s 用户定义的例程和数据类型开发者指南* 。

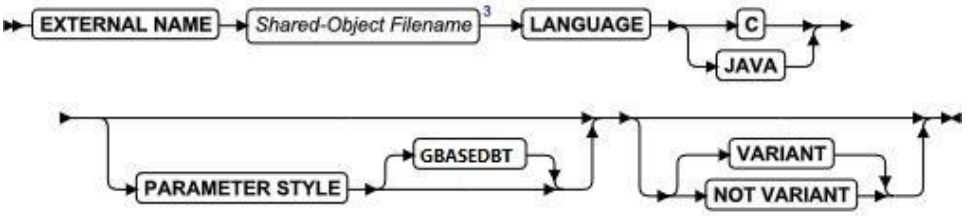
### 由 UDR 创建的对象的所有者

当在所有者特权的 UDR 的 DDL 已经创建了新数据库对象时，例程所有者（而不是执行它的用户，如果该用户不是例程的所有者）成为新数据库对象的所有者。但是，对于 DBA 特权的 UDR ，执行例程的用户（以及必须具有 DBA 特权的用户）将成为 UDR 创建的任何对象的所有者。

## 5.5 外部例程引用

当编写外部例程时使用外部例程引用。该项对 SPL 例程无效。

外部例程引用




**用法**

如果 IFX_EXTEND_ROLE 配置参数设置为 ON 或 1，则只能授予数据库服务器管理员（DBSA）及 DBSA 已经授予 EXTEND 角色的用户使用这个段的权限。缺省情况下，DBSA 是用户 **gbasedbt**。此外，您不能创建外部例程除非您持有数据库的 Resource 或 DBA 特权，并且还拥有编写此例程的外部程序语言的 Usage 特权。有关 GRANT USAGE ON LANGUAGE C 和 GRANT USAGE ON LANGUAGE JAVA 语句的语法，请参阅 语言级权限。

这个段指定以下关于外部例程的信息：

- 存储在共享对象文件中的可执行目标代码的路径名

  对于 C 例程，这个文件可以是 DLL 或共享库，这取决于您的操作系统。

- 对于 Java 例程，这个文件是 jar 文件。在能够创建用 Java 语句编写的 UDR 之前，必须用 sqlj.install_jar 过程分配一个 jar 标识符给外部 jar 文件。有关更多信息，请参阅 sqlj.install_jar。

- 用来编写 UDR 的编程语言的名称

- UDR 的参数样式

  缺省情况下，参数样式是 GBASEDBT。（这意味着如果指定 OUT 或 INOUT 参数，则 OUT 或 INOUT 值通过引用来传递。）

- VARIANT 或 NOT VARIANT 选项。如果指定其中一个，缺省为 VARIANT。如果例程包含任何 SQL 语句，它是一个 VARIANT 例程。如果包含外部例程引用子句的 DDL 语句还包含例程修改符子句，那么请不要在其中一个子句中将相同的 UDR 分类为 VARIANT ，而在其它子句中将 VARIANT 分类为 NOT VARIANT 。

**示例**

下面的示例包含 Java 语言编写 UDR 引用的外部例程。您必须首先使用过程 install_jar（*<absolute path><jar file name>,<internal registered name>*）注册 demo_jar。

```
CREATE FUNCTION delete_order(int) RETURNING int
    EXTERNAL NAME 'gbasedbt.demo_jar:delete_order.delete_order()'
    LANGUAGE JAVA;
```

## VARIANT 或 NOT VARIANT 选项

如果函数以相同参量调用时返回不同结果，或者如果它修改数据库或变量的状态，则函数是 ***variant***。例如，返回当前日期和时间的函数就是一个可变函数。

缺省情况下，用户定义函数是可变的。如果在创建或修改函数时指定 NOT VARIANT，那么函数就不能包含任何 SQL 语句。

如果函数是不变的，数据库服务器可以存储返回可变函数。更多关于函数型索引的信息，请参阅 CREATE INDEX 语句。

要注册一个不变函数，在这个子句或例程修饰符讨论的例程修饰符子句中添加 NOT VARIANT 选项。然而，如果在两处都指定修饰符，必须在两个子句中都使用同一修饰符（VARIANT 或 NOT VARIANT）。

### 用户定义的函数的示例

下面的例子注册了一个名为 equal( ) 的外部函数，接受两个 point 数据类型值作为参量。在这个例子中，point 是不透明数据类型，指定一个二维点的 x 和 y 坐标。

```
CREATE FUNCTION equal( a point, b point ) RETURNING BOOLEAN;
    EXTERNAL NAME "/usr/lib/point/lib/libbtype1.so(point1_equal)"
LANGUAGE C
END FUNCTION;
```

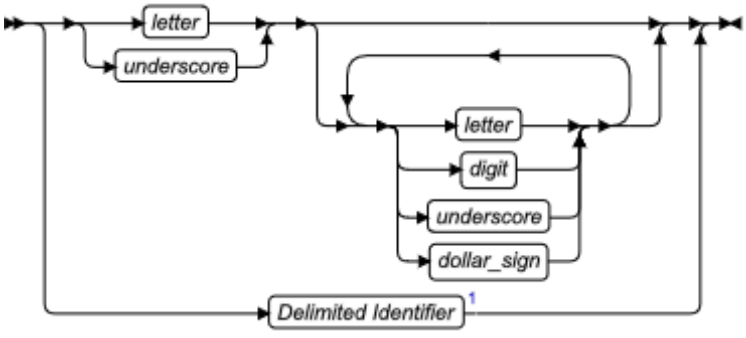
函数返回一个 BOOLEAN 类型的值。外部名称指定存储函数目标代码的 C 共享对象文件的路径。外部名称指出库包含另一个函数 **point1_equal( )**，它在 **equal( )** 执行时调用。

## 5.6 标识符

***identifier*** 指定数据库对象的未限定名称，例如存取方法、聚集、别名、blobspace 、强制转型、列、约束、相关性、数据类型、索引、运算符类、优化程序伪指令、分区、过程、表、触发器、序列、同义词或视图。当看到在语法图表中引用标识符时，使用标识符段。

**语法**

标识符




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *digit* | 范围在 0 到 9 之间的整数 | 不能作为第一个字符 | 精确数值 |

| | | | |
|---|---|---|---|
| *dollar_sign* | 美元（$）符号 | 不能作为第一个字符 | 从键盘输入的文字符号 |
| *letter* | 字母表中的大小或小写字母 | 在缺省的语言环境下，必须是从 A 到 Z 或 a 到 z 范围内的 ASCII 字符。 | 从键盘输入的文字符号 |
| *underscore* | 下划线（ _ ）字符 | 不能代替空格、连字号或其它非字母数字字符 | 从键盘输入的文字符号 |

**用法**

这是数据库对象名的一个逻辑子集，这一段详细说明了外部对象的***所有者***、***数据库***和***数据库服务器***。

要在标识符中包含其它非字母数字符号，例如空格（ASCII 32），必须使用定界标识符。建议在标识符中不要使用美元符号（$），因为这是一个特殊字符，在标识符中使用它可能会导致和其它语法元素的冲突。有关更多信息，请参阅定界标识符。

标识符长度至少要为 1 字节，但不超过 128 字节。例如，**employee_information** 作为表名称是有效的。如果使用的是多字节代码集，必须记住标识符的最大长度是指字节数而不是逻辑字符个数。

对于非缺省语言环境下的字母字符的情况，请参阅标识符中对非 ASCII 字符的支持。有关标识符 GLS 方面的更多信息，请参阅 *GBase 8s GLS 用户指南*的第三章。

当在 GBase 8s 中使用 ESQL/C 时，数据库服务器通过检查客户机应用程序的内部版本号和环境变量 **IFX_LONGID** 的设置，来确定客户端应用程序是否支持长标识符（最大长度为 128 字节）。有关更多信息，请参阅 *《GBase 8s SQL 指南：参考》*。

当数据库服务器使用长标识符时，您可能会遇到在 SQL 标识符或消息文本的错误消息、警告消息或其它消息。但是，如果标识符具有 18 个或更少的字节，通常可以避免截断。如果不同 SQL 对象的标识符在前 18 个字符中相同，那么您的代码可能很难读取或维护。

## 大写字符的使用

可以使用大写字符来指定数据库对象的名称，但是数据库服务器会把名称改为小写字符，除非设置环境变量 **DELIMIDENT** 并且数据库对象的标识用双引号（"）括起。在这种情况下，数据库服务器把数据库对象名作为定界标识符，保持其中的大写字符的形式，如定界标识符中所述。

如果数据库服务器包含大写字符，则数据库服务器不能参与分布式 DML 操作。要避免此限制，请在您声明数据库服务器的名称或别名时，仅使用不包含大写字符的未定界的名称。

## 使用关键字作为标识符

虽然几乎可以使用任何词作为标识符，但是在 SQL 语句中使用关键字作为标识符可能导致语法二义性。语句可能出错或不能产生期望结果。对于使用关键字作为标识符可能导致语法二义性的讨论以及这些问题的变通方法的说，请参阅潜在的多义性和语法错误。

定界标识符提供了最简单和最安全的方法来使用关键字作为标识符而不产生语法多义性。关键字作为定界标识符不需要变通方法。关于定界标识符的语法和使用，请参阅定界标识符。然而，定界标识符要求代码总是使用单引号（'）而非双引号（"）来划定字符串文字的界限。

关于在 GBase 8s SQL 实现的关键字，请参阅 GBase 8s 的 SQL 关键字。

**提示：** 如果错误消息看起来和导致错误的语句无关，那么请检查一下语句中是否使用了关键字作为未定界的标识。

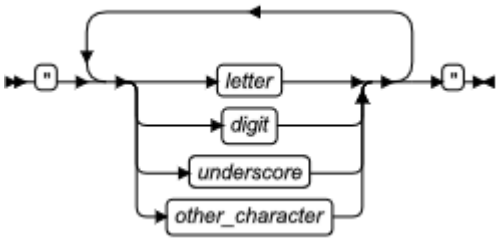
## 标识符中对非 ASCII 字符的支持

在非缺省的语言环境下，可以在 SQL 标识符中使用该 语言环境识别***字母***的任何字符。这一特性允许在某些数据库对象名中使用非 ASCII 字符。有关支持非 ASCII 字符的对象，请参阅 *GBase 8s GLS 用户指南*。

## 定界标识符

缺省情况下，有效的 SQL 标识符的字符集限制为字母、数字、下划线和美元符号。但是如果设置了 **DELIMIDENT** 环境变量，SQL 标识符也可以包含 **DB_LOCALE** 环境变量设置所指示的代码集中的附加字符。

定界标识符




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *digit* | 范围在 0 到 9 之间的整数 | 不能是第一个字符 | 精确数值 |
| *letter* | 组成定界标识符的字母 | 定界标识符中的字母是区分大小写的 | 从键盘输出的文字值 |
| *other_character* | 非字母数字字符，如 # 、$ 或空格 | 必须是数据库语言环境代码集中的元素 | 从键盘输出的文字值 |
| *underscore* | 在定界标识符中的下划线（ _ ） | 不能包含超过 128 个 | 从键盘输出的文字值 |

如果数据库服务器支持定界标识符，在代码中必须使用双引号（"）括起每个 SQL 标识符，并使用单引号（'）而非双引号（ "）为所有字符串文字定界。

定界标识符使您可以声明以其它方式相同与 SQL 关键字的名称，例如 TABLE 、WHERE 、DECLARE 等等。唯一不能指定定界标识的对象类型是数据库名称。

定界标识符中的字母是区分大小的。如果使用缺省的语言环境，*字母*必须是在 A 到 Z 或 a 到 z 范围内的大写和小写字符（在 ASCII 代码集中）。如果使用非缺省语言环境，*字母*必须是语言环境支持的字母字符。有关更多信息，请参阅定界标识中对非 ASCII 字符的支持（GLS）。

定界标识符服从 ANSI/ISO 标准中关于 SQL 的部分。

当创建一个数据库对象时，避免在定界标识符的第一个定界引号和第一个非空格或其它空白字符。（否则，可能无法在某些上下文中引用该对象）

如果数据库服务器的名称是定界标识符或如果它包含大写字符，则数据库服务器不能参与分布式 DML 操作。要避免此限制，请在您声明数据库服务器的名称或别名时，仅使用不包含大写字符的未定界的名称。

## 对非字母数字字符的支持

缺省情况下，SQL 标识符和所有语言环境的存储对象标识符中都支持 ASCII 、数字和下划线（ASCII 95）字符。要在数据库对象的名称中包含 **DB_LOCALE** 设置所隐含的代码集的其它字符，必须使用定界标识符。

然而，在声明或引用存储对象的名称（例如：dbspace 、partition 、blobspace 或 sbspace）时，不能使用定界标识符指定那些不是字母、数字或下划线（_ ）字符的字符。

## 定界标识中对非 ASCII 字符的支持（GLS）

当使用的非缺省语言环境其代码集支持非 ASCII 字符时，可以在定界标识符中指定非 ASCII 字符。规则是如果可以在标识符的非定界形式中指定非 ASCII 字符，那么也可以在同一标识符的定界形式中指定非 ASCII 字符。有关支持非 ASCII 字符的标识符列表以及在定界标识符中的非 ASCII 字符的信息，请参阅 *GBase 8s GLS 用户指南*。

## 启用定界标识符

要使用定界标识符，必须设置 **DELIMIDENT** 环境变量。当设置了 **DELIMIDENT**，将双引号（ " ）中的字符串视为数据库对象的标识符，单引号中的字符串视为文字字符串。但是，如果未设置 **DELIMIDENT** 环境变量，则双引号中的字符串也将视为文字字符串。

如果设置了 DELIMIDENT ，则为了将以下示例中的 SELECT 语句视引用字符串则必须将其放在单引号中：

```
PREPARE ... FROM 'SELECT * FROM customer';
```

如果在定义视图的 SELECT 语句中使用定界标识符，那么必须设置 **DELIMIDENT** 环境变量才能访问视图，即使视图名称本身不包含特殊字符也是如此。

在 UNIX™ 和 Linux™ 系统中，您可以通过设置环境变量的过程来设置 **DELIMIDENT** ，这些过程在 *《GBase 8s SQL 指南：参考》*中描述。

## 定界标识符的示例

以下示例显示了如果创建一个名称区分大小写的表：

```
CREATE TABLE "Proper_Ranger" (...);
```

下面的示例创建了一个表，它的名称包含一个空白字符。如果表的名称没有用双引号（"）括起来。并且没有设置 DELIMIDENT ，就不能在标识符中使用空格。

```
CREATE TABLE "My Customers" (...);
```

下一个例子创建了一个以关键字作名称的表：

```
CREATE TABLE "TABLE" (...);
```

下面的 GBase 8s 示例说明了当 DELETE 语句中省略了关键字 FROM 时，如何从名为 FROM 的表中删除所有行：

```
DELETE "FROM";
```

## 在定界标识中使用双引号

为了在定界标识符中包含双引号（"），您必须在双引号（"）前面加另一个双引号（" ）。如以下示例语句段指定 My "Good" Data 作为表名：

```
CREATE TABLE "My ""Good"" Data" (...);
```

### 潜在的多义性和语法错误

GBase 不建议使用 SQL 的任何关键字作为标识符，因为这样做容易使代码更难阅读及维护。但如果您忽略这个对阅读者潜在的问题，您几乎可以使用任何关键字作为 SQL 标识符，但可能会出现各种形式的语法多义性，一个多义性的语句可能不能产生期望的结果。下面几节说明了当将关键字声明为标识符或当不同的数据库对象就相同的标识符时一些潜在的多义性和解决方法。

### 使用内置函数的名称作为列名

下面的两个例子列出了用 SELECT 语句把内置函数作为列名的一个变通方法。这种变通方法应用于聚集函数（AVG 、COUNT 、MAX 、MIN 、SUM）和函数表达式（代数、指数和对数、time 、HEX 、length 、DBINFO 、trigonometric 和 TRIM 函数）。

使用 avg 作为列名导致下面这个例子的失败因为数据库服务器把 avg 解释为聚集函数而不是列名：

```
SELECT avg FROM mytab; -- fails
```

如果设置了 DELIMIDENT 环境变量，就可以使用 avg 作为列名，如下面的例子所示：

```
SELECT "avg" from mytab; -- successful
```

下面的例子中的变通方法通过在列名中包含表名，去除多义性：

```
SELECT mytab.avg FROM mytab;
```

如果您使用 TODAY 、CURRENT 、SYSDATE 或 USER 关键字作为列名，就可能产生多义性，如下面的例子所示：

```
CREATE TABLE mytab (user char(10),
```

```
CURRENT DATETIME HOUR TO SECOND,TODAY DATE);
INSERT INTO mytab VALUES('josh','11:30:30','1/22/2008');
SELECT user,current,today FROM mytab;
```

数据库服务把 SELECT 语句中的 **user** 、**current** 和 **today** 解释成内置函数 USER 、CURRENT 和 TODAY。于是 SELECT 语句返回了当前用户名、当前时间和当前日期而不是 josh, 11:30:30,1/22/2008。SYSDATE 关键字在 GBase 8s 数据库中具有类似的效果。

如果想选择表中实际的列，必须把 SELECT 语句写成下面几句之一：

```
SELECT mytab.user,  mytab.current,  mytab.today FROM mytab;
 EXEC SQL select * from mytab;
```

## 使用关键字作为列名

有特定的变通方法在 SELECT 语句或其它 SQL 语句中使用关键字作为列名。在某些情况下，有不止一个变通方法可用。

## 使用 ALL 、DISTINCT 或 UNIQUE 作为列名

如果想要在 SELECT 语句中使用 ALL 、DISTINCT 或 UNIQUE 这些关键字作为列名，可以采用一种变通方法。

首先，考虑一下当不采用变通方法时试图使用这些关键字之一会发生什么情况。在下面的例子中，使用 all 作为列名导致 SELECT 语句失败，因为数据库服务器将 all 解释成关键字而不是列名：

```
SELECT all FROM mytab -- fails;
```

必须使用一个变通方法来使这个 SELECT 语句执行成功。如果设置了环境变量 DELIMIDENT ，就可以通过吧 all 用双引号括起来使用 all 作为列名。在下面的例子里，SELECT 语句执行成功，因为数据库服务器把 all 解释为列名：

```
SELECT "all" from mytab; -- successful
```

下面的例子中的变通方法把关键字 ALL 和列名 all 一起使用：

```
SELECT ALL all FROM mytab;
```

下面的示例给出了几种在 CREATE TABLE 语句中使用关键字 UNIQUE 或 DISTINCT 作为列名的变通方法。

下一示例没能声明一个名为 unique 的列，因为数据库服务器将 unique 解释成关键字而不是列名：

```
CREATE TABLE mytab (unique INTEGER); -- fails
```

下面的变通方法使用了两个 SQL 语句。第一个语句创建列 mycol；第二个语句将列 mycol 重命名为 unique：

```
CREATE TABLE mytab (mycol INTEGER);
 RENAME COLUMN mytab.mycol TO unique;
```

下面的变通方法也使用了两个 SQL 语句。第一个语句创建列 mycol；第二个语句修改表，在其中 unique，并删除列 mycol：

```
CREATE TABLE mytab (mycol INTEGER);
        ALTER TABLE mytab
        ADD (unique INTEGER),
        DROP (mycol);
```

## 使用 INTERVAL 或 DATETIME 作为列名

这一节的例子给出了在 SELECT 语句中使用关键字 INTERVAL（或 DATETIME）作为列名的变通方法。

使用 interval 作为列名导致下面的例子失败，因为数据库服务器把 interval 解释为关键字并认为后面应该跟一个 INTERVAL 限定符：

```
SELECT interval FROM mytab; -- fails
```

如果设置了 DELIMIDENT 环境变量，则您可以使用 interval 作为列名，如下例所示：

```
SELECT "interval" from mytab; -- successful
```

下面例子的变通方法通过在列名中指定表名，去除了多义性：

```
SELECT mytab.interval FROM mytab;
```

下面例子的变通方法在表名中包含了所有者的名称：

```
SELECT josh.mytab.interval FROM josh.mytab;
```

## 使用 rowid 作为列名

每个非分片表都有一个名为 rowid 的虚拟列。为了避免多义性，不可用使用 rowid 作为列名。执行以下操作会导致出错：

- 创建一个名为 **rowid** 的表或视图
- 通过提交列名为 **rowid** 来修改表
- 把一列重命名为 **rowid**

然而，您可以使用术语 rowid 作为表名：

```
CREATE TABLE rowid (column INTEGER, date DATE, char CHAR(20));
```

**重要：** 建议将主键用作存取方法而不是利用 rowid 列。

## 使用关键字作为表名

数据库服务器在表对象的未限定的标识符也是有效的 SQL 关键字的上下文中发出错误。您可以通过使用表的所有者的授权标识符对其进行限定来消除表名称的歧义。

以下示例来说明当关键字 STATISTICS 、OUTER 或 FROM 已声明为表名或同义词时，所有者名称限定符作为解决方法。（如果任何关键字是视图的标识符，则也适用于这些示例。）

将 statistics 作为表标识符会导致以下 UPDATE 语句示例失败。发生异常是因为数据库服务器将统计信息解释为语法不正确的 UPDATE STATISTICS 语句中的关键字，而不是 UPDATE 语句中的目标表名：

```
UPDATE statistics SET mycol = 10;   -- fails
```

以下示例中的解决方法使用所有者名称限定表名称，以避免多义性：

```
UPDATE josh.statistics SET mycol = 10;
```

使用 outer 作为别名会导致下列示例失败，因为数据库服务器将 outer 解释为执行执行语法不正确的外部连接：

```
SELECT mycol FROM outer; -- fails
```

下面这个成功的示例使用所有者命名来避免多义性：

```
SELECT mycol FROM josh.outer;
```

下面的 DELETE 语句，其目标表在创建时使用 **from** 作为它的标识符，返回一个语法错误：

```
DELETE from;      -- fails
```

因为 FROM 也是一个可选的关键字，紧跟在要记录其记录的表的名称之前，所以数据库服务器在 FROM 之后需要一个表名。找不到，它会发生异常。

相反，下面的示例从正确识别为 **from** 表的行中删除行，因为该表名由其所有者的名称限定：

```
DELETE zelaine.from;
```

如果在数据库服务器上设置了 **DELIMIDENT** 环境变量，则代替的变通方法为使用双引号（"）作为分隔符。

```
DELETE "from";
```

这通过指示 **from** 是 SQL 标识符，而不是字符串文字或 SQL 关键字。

尽管有这些解决方法，如果您避免声明 SQL 关键字作为表、视图或其它数据库对象的标识符，那么您的代码将更容易阅读和维护。

## 使用关键字 AS 的变通方法

在某些情况下，虽然语句并没有多义性而且语法也正确，但是数据库服务器却返回语法错误的信息。前页给出了几种情况下现有的语法变通方法。可以使用关键字 AS 来提供对于异常的一种变通方法。

可以在列标号或表别名前面使用 AS 关键字。

下面的例子把关键字 AS 和列标号一起使用：

```
SELECT column_name AS display_label FROM table_name;
```

下面的例程使用 AS 关键字作为表的别名：

```
SELECT select_list FROM table_name AS table_alias;
```

## 将 AS 和列标签一起使用

这一节的例子给出了关键字 AS 和列标签一起使用的变通方法。前面两个例子给出了如何使用关键字 UNITS （或 YEAR 、MONTH 、DAY 、HOUR 、MINUTE 、SECOND 或 FRACTION）作为列标签。

使用 units 作为列标签导致下一示例失败，因为数据库服务器将它解释成 INTERVAL 表达式的一部分，在这个表达式中 mycol 列是 UNITS 运算符的操作数：

```
SELECT mycol units FROM mytab;
```

下面例子中的变通方法包含了关键字 AS ：

```
SELECT mycol AS units FROM mytab;
```

以下示例使用了关键字 AS 或 FROM 作为列标签。

使用 as 作为列标签导致下一示例失败，因为数据库服务器将 as 解释成把 from 作为列标签，于是发现不需要的 FROM 子句：

```
SELECT mycol as from mytab; -- fails
```

下面这个成功的例子重复使用了一次关键字 AS ：

```
SELECT mycol AS as from mytab;
```

使用 from 作为列标签导致下一示例失败，因为数据库服务器认为第一个 from 后面应该跟一个表名：

```
SELECT mycol from FROM mytab; -- fails
```

这个例子使用关键字 AS 把第一个 from 看作列标签：

```
SELECT mycol AS from FROM mytab;
```

## 将 AS 和表别名一起使用

本节中的示例显示了将 AS 关键字和表别名一起使用的变通方法。开始的两个例子说明了如果使用 ORDER 、FOR 、GROUP 、HAVING 、INTO 、UNION 、WITH 、CREATE 、GRANT、KEY、ROLE 或 WHERE 关键字作为表别名。

使用 order 作为表别名导致下面的例子失败，因为数据库服务器把 order 解释为 ORDER BY 子句的一部分：

```
SELECT * FROM mytab order; -- 失败
```

下面例子的变通方法使用关键字 AS 把 order 看作表别名：

```
SELECT * FROM mytab AS order;
```

接下来两个例子说明了如何使用关键字 WITH 作为表别名。

使用 with 作为表别名导致下面的例子失败，因为数据库服务器把 with 解释为 WITH CHECK OPTION 语法的一部分：

```
EXEC SQL select * from mytab with; -- 失败
```

下面例子的变通方法使用关键字 AS 把 with 作为表别名：

```
EXEC SQL select * from mytab as with; -- 成功
```

下面的两个示例使用关键字 CREATE 作为表别名。使用 create 作为表别名导致下面的例子失败，因为数据库服务器将此关键字解释成超级新数据库对象的语法的一部分，如创建表、同义词或视图：

```
EXEC SQL select * from mytab create; -- 失败
EXEC SQL select * from mytab as create; -- 成功
```

变通方法使用关键字 AS 将 **create** 识别为表别名。（使用 grant 作为别名将同样失败，但在 AS 关键字之后是有效的。）

## 调取以关键字作为名称的游标

在某些情况下，对于关键字用作 SQL 程序标识符时产生的多义性没有现成的变通方法。

在下面的例子中，FETCH 语句指定了一个游标名为 next ，FETCH 语句产生了语法错误，因为预处理器把 next 解释为关键字，标记为活动集中的下一行，并认为 next 后面要跟一个游标名。只有当关键字 NEXT 、PREVIOUS 、PRIOR 、FIRST 、LAST 、CURRENT 、RELATIVE 或 ABSOLUTE 用作游标名时都会发生这种情况：

```
/* This code fragment fails */
EXEC SQL declare next cursor for
select customer_num, lname from customer;
EXEC SQL open next;
EXEC SQL fetch next into :cnum, :lname;
```

## 调取以关键字作为名称的游标

如果使用下列关键字中的任何一个作为用户定义的例程（UDR）中变量的标识符，就会产生语法歧义：

- CURRENT
- DATETIME
- GLOBAL
- INTERVAL
- NULL
- OFF
- OUT
- PROCEDURE
- SELECT
- SYSDATE

## 在 INSERT 语句中使用 CURRENT 、DATETIME 、INTERVAL 和 NULL

UDR 例程中不能插入用关键字 CURRENT 、DATETIME 、INTERVAL 或 NULL 作为名称声明的变量。例如，如果声明了一个名为 null 的变量，当试图把值 null 插入一列时，就会收到语法错误，如下面的例子所示：

```
 CREATE PROCEDURE problem()
          . . .
   DEFINE null INT;
   LET null = 3;
 INSERT INTO tab VALUES (null); -- error, inserts NULL, not 3
```

## 在条件中使用 NULL 和 SELECT

如果声明了一个用 null 或 select 命名的变量，则将它包含在使用关键字 IN 的条件中，就会产生多义性。下面的例子给出了产生问题的三种条件：在 IF 语句中，在 SELECT 语句的 WHERE 子句中，以及在 WHILE 条件中：

```
CREATE PROCEDURE problem()
  . . .
  DEFINE x,y,select, null, INT;
  DEFINE pfname CHAR[15];
  LET x = 3; LET select = 300;
  LET null = 1;
  IF x IN (select, 10, 12) THEN LET y = 1; -- problem if

  IF x IN (1, 2, 4) THEN
  SELECT customer_num, fname INTO y, pfname FROM customer
  WHERE customer IN (select , 301 , 302, 303); -- problem in

  WHILE x IN (null, 2)        -- problem while
  . . .
  END WHILE;
```

如果可以确保不是列表中的第一个元素，就可以在 IN 列表中使用变量 select。下面例子中的变通方法改正了前面例子中的 IF 语句：

```
IF x IN (10, select, 12) THEN LET y = 1; -- problem if
```

对于把 **null** 用作变量名并试图把该变量用在 IN 条件的情况，没有可用的变通方法。

## 将关键字或例程名称声明为 SPL 变量

如果声明一个变量与关键字或例程的名称相同，可能会出现歧义。 GBase 8s 使用以下规则来解析 SPL 变量、UDR 名称和内置 SQL 函数名称之间的名称冲突。

- 在 DEFINE 语句中声明的变量名采用最高优先级。
- 在 CREATE PROCEDURE 或 CREATE FUNCTION 语句中定义的用户定义例程优先于内置 SQL 函数。
- 在 DEFINE 语句中使用 PROCEDURE 关键字声明的过程优先于内置 SQL 函数。
- 内置 SQL 函数优先于数据库中存在的 SQL 过程，但在 DEFINE 语句中未显式标识为过程。

如果您需要调用 SQL 函数，请不要使用内置 SQL 函数的名称作为 SPL 变量。 例如，如果您还需要调用这些聚集函数，则不要声明名称为 count 或 max 的变量。

## 与列名冲突的变量

如果对 SPL 变量和列名称使用相同的标识符，则在变量引用范围内，数据库服务器将未限定标识符的任何实例解释为变量。要使用标识符指定列名称，请使用 ***table.column*** 表示法将列名称与表名称进行限定。在以下示例中，过程变量 **lname** 与列名称相同。在以下 SELECT 语句中，**customer.lname** 是数据库中的一个列，**lname** 是一个 SPL 变量：

```
CREATE PROCEDURE table_test()
DEFINE lname CHAR(15);
LET lname = "Miller";
SELECT customer.lname FROM customer INTO lname
    WHERE customer_num = 502;
```

此示例有效，但依赖 GBase 8s 的优先级规则来解决 SPL 变量和列名称之间的名称冲突可能会使您的代码难以解读和维护。重复使用与变量和列名称相同的标识符的替代方法是，DEFINE 语句声明标识符的一些前缀，例如本示例中的 **v_lname**，以指示此变量存储列 **lname** 的值。

## 在 TRACE 语句中使用 ON 、OFF 或 PROCEDURE

如果定义了一个 SPL 变量命名为 **on** 、**off** 或 **procedure**，并且试图把它用在 TRACE 语句中，就不能跟踪得到变量的值。而是执行了 TRACE ON 、TRACE OFF 或 TRACE PROCEDURE 语句。可以通过把变量标记在一个更复杂的表达式中来得到变量的值。

下面的例子给出了使用算术或字符串表达式计算变量产生歧义的语法和变通方法：

```
DEFINE on, off, procedure INT;

TRACE on;     --产生多义性
TRACE 0+ on;   --正确
TRACE off;     --产生多义性
TRACE ''||off;     --  正确

TRACE procedure;   --产生多义性
TRACE 0+procedure;     --  正确
```

## 使用 GLOBAL 作为变量名

如果试图以 global 作为名称来定义变量，define 操作会失败。下面例子中给出的语法和定义全局变量的语法冲突：

```
DEFINE global INT; --  失败;
```

如果设置了环境变量 DELIMIDENT ，就可以使用 global 作为变量名，如下面的例子所示：

```
DEFINE "global" INT; -- successful
```

**重要：** 虽然前几节给出的变通方法可以在关键字用作标识符时避免编译或运行时语法冲突，但是必须记住这种标识符容易把代码变得更加难以理解和维护。

## 使用 EXECUTE 、SELECT 或 WITH 作为游标名

请勿使用 EXECUTE 、SELECT 或 WITH 关键字作为游标名。如果试图在 FOREACH 语句中使用这些关键字之一作为游标名，那么游标名会被解释成 FOREACH 语句中的关键字。没有可用的变通方法。

下面的例子不能生效：

```
DEFINE execute INT;
```

FOREACH execute FOR SELECT col1 -- 错误，解析器将视为 INTO var1 FROM tab1; -- 'FOREACH EXECUTE PROCEDURE'

## WHILE 和 FOR 语句中的 SELECT 语句

如果在 WHILE 或 FOR 循环中使用 SELECT 语句，并且需要用括号把它括起来，那么应该把整个 SELECT 语句括在 BEGIN…END 语句块中。在下面的例子中，第一个 WHILE 语句中的 SELECT 语句被解释为调用过程 var1；对第二个 WHILE 语句的解释是正确的：

```
DEFINE var1, var2 INT;
WHILE var2 = var1
    SELECT col1 INTO var3 FROM TAB -- error, interpreted as call var1()
    UNION
    SELECT co2 FROM tab2;
END WHILE;

WHILE var2 = var1
    BEGIN
        SELECT col1 INTO var3 FROM TAB -- ok syntax
        UNION
        SELECT co2 FROM tab2;
    END
END WHILE;
```

## SPL 的 ON EXCEPTION 语句中的 SET 关键字

如果在 ON EXCEPTION 语句中使用以关键字 SET 开头的语句，必须把它包括在 BEGIN ... END 语句块中。

下面的列表给出了一些以关键字 SET 开头的 SQL 语句：

- SET AUTOFREE
- SET CONNECTION
- SET CONSTRAINTS
- SET DATASKIP
- SET DEBUG FILE
- SET DEFERRED_PREPARE
- SET DESCRIPTOR
- SET ENCRYPTION
- SET ENVIRONMENT
- SET EXPLAIN
- SET INDEXES
- SET ISOLATION
- SET LOCK MODE
- SET LOG
- SET OPTIMIZATION
- SET PDQPRIORITY

- SET ROLE
- SET STATEMENT CACHE
- SET TABLE
- SET TRANSACTION
- SET TRIGGERS

下面的例子给出了 SET LOCK MODE 语句在 ON EXCEPTION 语句中的错误和正确用法。

下面的 ON EXCEPTION 语句返回了错误，因为 SET LOCK MODE 语句没有包括在 BEGIN ... END 语句块中：

```
ON EXCEPTION IN (-107)
SET LOCK MODE TO WAIT; -- error, value expected, not 'lock'
END EXCEPTION;
```

下面的 ON EXCEPTION 语句执行成功，因为 SET LOCK MODE 语句被包括在 BEGIN ... END 语句块中：

```
ON EXCEPTION IN (-107)
BEGIN
SET LOCK MODE TO WAIT; -- ok
END
END EXCEPTION;
```

## 5.7 Jar 名称

使用 Jar 名称段来指定 jar ID 的名称。当看到在语法图表中引用 Jar 名称时，使用本段。

**语法**

Jar 名称


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *database* | 要在其中安装或访问 *jar_id* 的数据库 | 标准的 *database.package.jar_id* 标识符不能超过 255 字节 | 数据库名 |
| *jar_id* | 包含要访问的 Java™ 类的 .jar 文件 | 文件必须在 *database.package* 中存在 | 标识符 |
| *package* | 数据包的名称 | 数据包必须在 *database* 中存在 | 标识符 |

如果 jar 名称被指定为 **sqlj.install_jar** 、**sqlj.replace_jar** 或 **sqlj.remove_jar** 过程的字符串参量，那么 jar 名称中所有定界标识符将包含两边的双引号字符。

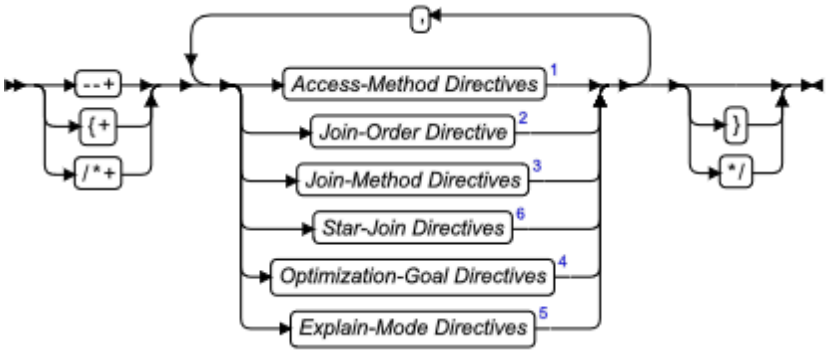
在可以用任何方法访问 ***jar_id*** 之前（包括在 CREATE FUNCTION 或 CREATE PROCEDURE 语句中使用），必须在当前数据库中用 **install_jar( )** 过程对它进行定义。有关更多信息，请参阅 EXECUTE PROCEDURE 语句。

## 5.8 优化程序伪指令

优化程序伪指令段指定可用以部分或完全指定优化程序查询计划的关键字。当看到语法图表中引用优化程序伪指令时，使用本段。

**语法**

优化程序伪指令




**用法**

使用一个或多个优化程序伪指令来部分或完全规定优化程序查询计划。伪指令的作用域只有当前查询。

伪指令缺省是启用的。要获得关于如何处理指定的伪指令的信息，可以查看 SET EXPLAIN 语句的输出。要禁用伪指令，可以把环境变量 **IFX_DIRECTIVES** 设置成 0，或者把 ONCONFIG 文件中的 DIRECTIVES 参数设置成 0。

上述的语法图被简化，并且不显示结束注释指示符必须遵循与开始注释指示符相同额注释样式。有关更多信息，请参阅优化程序伪指令作为注释。

### 优化程序伪指令作为注释

优化程序伪指令需要有效的注释指示符作为定界符。

您使用的结束的定界符取决于开始的定界符：

- 如果 { 表示开始的定界符，那么必须使用 } 作为结束的定界符。
- 如果 /* 是开始的定界符，那么必须使用 */ 作为结束的定界符。
- 如果 -- 是开始的定界符，就不需要结束的定界符。

一条优化程序伪指令或一列优化程序伪指令以注释形式紧跟在关键字 DELETE 、SELECT 或 UPDATE 后面，在注释符号后面，优化程序伪指令的第一个字符总是加号（+）。在注释指示符和加号之间不允许有空格或其它空字符。

可以使用下列任何一种注释指示符：

- 双连字号（--）定界符

  双连字号不需要结束符，因为它表示只有当前行的剩余部分是注释。当使用这种符号时，只包括当前行的优化程序伪指令。

- 括号（**{ . . . }**）定界符

  从左括号（ **{** ）到右括号（}）之间的部分都是注释的；可以在同一行也可以在后面的几行。

- C 语言风格斜线和星号（/* . . . */）定界符

  从开始的斜线-星号（/*）对到同一行或下面几行的下一个星号-斜线（*/）字符之间都是注释。

  在 GBase 8s ESQL/C 中，**esql** 编译器的 **-keepccomment** 命令可选项在使用 C 语言样式的注释时必须指定。

有关其它信息，请参阅如何输入 SQL 注释。

如果在同一查询中指定多个伪指令，那么必须使用空格、逗号或所选择的任何字符把它们分隔开。建议用逗号分隔连续的指令。

如果查询为表声明了别名，那么在优化程序伪指令规范中使用这个别名（而不是实际的表名）。因为系统生成的索引名是以空字符开头的，所以要用引号为这种名称定界。

优化程序伪指令中的语法错误不会导致合法查询的失败。使用 SET EXPLAIN 语句可以获得这种错误的相关信息。

在分布式查询中，优化程序伪指令可以通过使用 ***database*:*table*** or ***database*:*owner*.*table*** 表示法来引用同一服务器实例中的其它数据库的对象，以限定本地数据库服务器的另一个数据库中的表的名称。

GBase 8s 的 ORACLE 模式（SQLMODE=ORACLE）支持使用 DATABASE.OBJECT 方式引用数据库对象。数据库对象包括：表、视图、索引、序列、存储过程、存储函数、内置函数、包、内置包。

## 优化程序伪指令上的限制

除非包含下列语法元素的任何一项，否则可以对 DELETE 、SELECT 或 UPDATE 语句的任何查询指定优化程序伪指令：

- 访问当前数据库以外的表的查询
- 在 GBase 8s ESQL/C 中，带有 WHERE CURRENT OF ***cursor*** 子句的语句

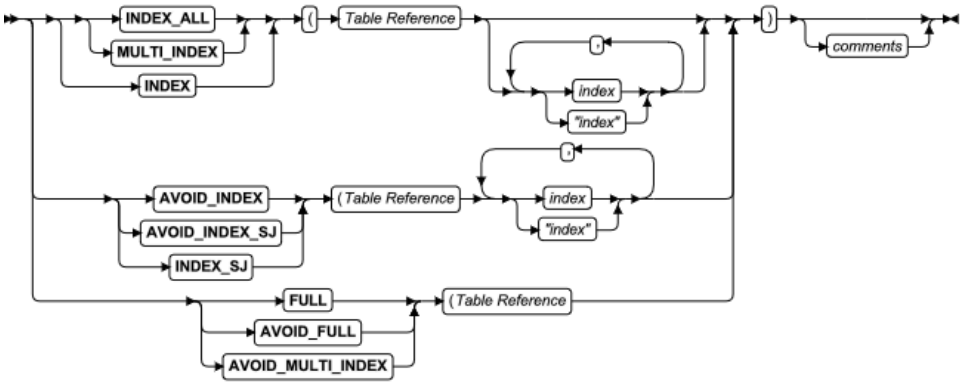
对于使用符合 ANSI/ISO 语法指定连接的查询，该查询优化程序不遵守某些伪指令：

- 将忽略 join-method 伪指令（USE_NL 、AVOID_NL 、USE_HASH 、AVOID_HASH 、/BUILD 和 /PROBE），除非优化程序重写查询以使其不再使 ANSI/ISO 语法。
- 在指定 RIGHT OUTER JOIN 或 FULL OUTER JOIN 关键字的符合 ANSI 的连接查询中忽略连接顺序指定（ORDERED）。

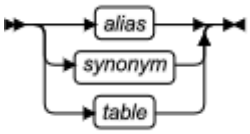
## 存取方法伪指令

使用存取方法伪指令来指定优化程序搜索表的方式。

存取方法伪指令




表参考




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *alias* | FROM 子句中声明的临时备用表名 | 如果声明了一个 *alias*，它必须用在优化程序伪指令中（而不是 *table* or *synonym*） | 标识符 |
| *comments* | 为优化程序伪指令提供文档的文本 | 必须在括号外面但在注释符号里面 | 字符串 |
| *index* | 要指定查询计划伪指令的索引 | 必须存在。对 AVOID_INDEX 、AVOID_INDEX_SJ 和 INDEX_SJ ， 至少需要一个 *index* | 标识符 |
| *synonym*, *table* | 要指定伪指令的查询中的同义词或表 | 同义词以及它所指向的表必须存在 | 标识符 |

用逗号或空格来分隔括号中的元素。

下面的表说明了每个存取方法伪指令并指出了它如何影响优化的查询计划。

| 关键字 | 作用 | 优化程序的操作 |
|---|---|---|

| | | |
|---|---|---|
| AVOID_FULL | 列出的表没有全表扫描 | 优化程序考虑它能扫描的不同索引，如果没有索引，优化程序执行全表扫描。 |
| AVOID_INDEX | 不使用任何列出的索引 | 优化程序考虑剩余的索引和全表扫描。如果一个表的所有索引都指定了，优化程序就使用全表扫描来访问表。 |
| AVOID_INDEX_SJ | 对指定的索引不使用索引自连接路径 | 优化程序不考虑用于在索引自连接路径中扫描表的指定索引。 |
| AVOID_MULTI_INDEX | 对指定的表不使用多索引扫描 | 优化程序不考虑指定表的多索引扫描路径。 |
| FULL | 实行全表扫描 | 即使一列存在一个索引，优化程序也使用全表扫描来访问表。 |
| INDEX | 使用指定的索引来访问表 | 如果指定了多个索引，优化程序会选择产生最小成本的索引。如果没有指定索引，那么所有可用的索引都会被考虑。 |
| INDEX_ALL ***或*** MULTI_INDEX | 使用指定索引来访问表（多索引扫描） | 这些关键字是同义词。有关其用法信息，请参阅下面的“多索引扫描”。 |
| INDEX_SJ | 使用指定使用扫描索引自连接路径中的表 | 优化程序被强制使用具有指定索引的索引自连接路径（或者索引自连接路径在索引列表中选择成本最低的索引）扫描表。 |

AVOID_FULL 和 INDEX 关键字都表示优化程序应该避免全表扫描。然而，建议使用关键字 AVOID_FULL 来表示避免全表扫描的意图。

AVOID_MULTI_INDEX 伪指令不接受索引的列表作为它的参数。这是因为 AVOID_INDEX 伪指令还会阻止在多索引扫描执行路径中使用指定的索引。

### 多索引扫描

可在表上定义多达十六个（16）索引。基于在同一表上使用多个索引的访问方法的搜索路径被称为 ***多索引扫描***。 MULTI_INDEX 或 INDEX_ALL 伪指令强制查询优化程序考虑多索引扫描以搜索指定的表以进行限定。MULTI_INDEX 或 INDEX_ALL 指令的参数列表具有以下语义：

- 如果指定表作为指令的唯一参数，则优化程序会考虑该表上的所有可用的索引，并在搜索表中的限定行时使用所有这些索引（或子集）。

- 如果指定表且仅指定单个索引，则优化程序会考虑仅使用该索引来扫描表。
- 如果指定表和多个索引，则优化程序会考虑使用所有指定索引的搜索路径。

**使用 skip-scan 存取方法的多索引扫描**

多索引扫描路径通过跳过扫描访问方法使用 ROWID 的排序列表访问。排序列表通常使用 INDEX_ALL 或 MULTI_INDEX 指令指定的所有索引从多索引扫描存取方法生成。

例如，如果查询谓词指定 `col1 <= 10` 和 `col2 BETWEEN 15 AND 25`，则执行计划可以使用两个索引：**col1** 上的第一个索引，**col2** 上的第二个索引。每个索引扫描返回满足相应索引的搜索条件的所有 ROWID。ROWID 的两个列表的逻辑交叉仅包括满足两个搜索条件的行。然后，数据库服务器对组合的 ROWID 列表进行排序，并使用此排序列表来扫描表查询的结果集。

如果查询包含多于两个索引列的谓词，则每个索引扫描返回的 ROWID 列表必须合并，以生成所有限定行的排序 ROWID 列表。

因为每个 ROWID 表示一行的物理位置（在哪个页面上和哪个 slot 中），执行路径简单地访问该物理位置以检索该行。由于术语 "skip-scan" 建议，在排序列表中通常存在从一个 ROWID 到下一个 ROWID 的间隙，是的数据库服务器从结果集合的一个合格行“跳过”到下一个合格行。

排序的 ROWID 的列表可以从多个索引扫描生成，如上所述，或者从单个索引扫描生成。在单个索引的情况下，跳过扫描执行路径执行以下操作：

1. 单索引扫描创建所有限定行的 ROWID 的未排序列表。
2. 此未排序的列表按 ROWID 值排序。
3. 数据库服务器按照 ROWID 的顺序检索合格行。

Skip-scan 存取方法类似于顺序扫描，但是有时可能更有效。顺序扫描检索表中的每一行，但跳过扫描只检索限定的 ROWID 的行。

**查询执行的多索引扫描路径的限制**

事务隔离级别影响 MULTI_INDEX 或 INDEX_ALL 伪指令是否可强制多索引扫描执行路径，当隔离级别为 Cursor Stability 时，或者使用 LAST COMMITTED 选项的 Committed Read 时无效。（但是，在 Dirty Read 和 Repeatable Read 隔离级别，和不带有 LAST COMMITTED 选项的 Committed Read 隔离级别中支持这些伪指令）。

以下附加的限制适用于多索引扫描访问路径：

- 该索引必须是 B-tree 索引。它们可以是连接的或拆离的索引。
- 这些伪指令忽略 R-tree 索引、函数使用和基于虚拟索引接口（VII）的索引。
- 该表不能是远程表、伪表、系统目录表、外部表或层次结构表。
- 多索引扫描不支持连接谓词为基于索引扫描的索引过滤器。
- 多索引扫描忽略除主列之外的复合索引的所有列。
- 执行级联删除或声明语句局部变量（SLV）的 DML 语句不能使用多索引扫描 s。
- 更新激活 FOR EACH ROW 触发操作的查询不能使用多索引扫描。

- 在兼容 ANSI 的数据库中，如果 FROM 子句只指定一个表，那么对于没有 GROUP BY 子句和没有 FOR READ ONLY 子句的 SELECT Y 语句，不遵循 MULTI_INDEX 或 INDEX_ALL 伪指令。（在这种特殊情况下，查询具有与多索引扫描访问路径冲突的隐式游标行为。）

**存取方法伪指令组合**

通常，您只能为一个表指定一个存取方法。只有下列存取方法伪指令的组合在同一查询的同一表中有效：

- INDEX, AVOID_INDEX_SJ
- AVOID_FULL, AVOID_INDEX
- AVOID_FULL, AVOID_INDEX_SJ
- AVOID_INDEX, AVOID_INDEX_SJ
- AVOID_FULL, AVOID_INDEX, AVOID_INDEX_SJ
- AVOID_FULL, AVOID_MULTI_INDEX
- AVOID_INDEX, AVOID_MULTI_INDEX
- AVOID_INDEX_SJ, AVOID_MULTI_INDEX
- AVOID_FULL, AVOID_INDEX_SJ, AVOID_MULTI_INDEX
- AVOID_INDEX, AVOID_INDEX_SJ, AVOID_MULTI_INDEX

当指定 AVOID_FULL 和 AVOID_INDEX 存取方法伪指令时，优化程序避免表的全表扫描并且它避免使用指定的索引。此负指令的组合允许优化程序使用指定的存取方法伪指令之后创建的索引。

因为如果指定 INDEX 或 AVOID_FULL 伪指令，优化程序会自动考虑索引自连接路径，所以 INDEX_SJ 伪指令只是强制使用指定索引的索引自连接路径（或者在逗号分隔的列表中选择成本最低的索引的索引）。当多列索引包括仅提供低选择性的列作为索引键过滤时，INDEX_SJ 伪指令可以提高性能。

指定 INDEX_SJ 伪指令规避了索引引导键上数据分布统计信息的常见优化程序要求。此指令使优化程序考虑索引自连接路径，即使数据分布统计信息不可用于索引键列。在这种情况下，优化程序仅包括索引键列的最小数量作为满足指令的引导键。

例如，如果在列 **c1** 、**c2** 、**c3** 、**c4** 上定义了索引，并且查询为这四个列指定了过滤器，但没有数据分布在任何列上可用，则在此索引上指定 INDEX_SJ 会导致列 **c1** 被用作索引自连接路径中的引导键。如果希望优化程序使用索引但不考虑索引自连接路径，则必须指定 INDEX 或 AVOID_FULL 伪指令选择索引，并且还必须指定 AVOID_INDEX_SJ 伪指令以防止优化程序考虑任何其它索引自连接路径。

如果 AVOID_INDEX_SJ 与 INDEX 伪指令一起使用，作为显式 INDEX 或等效的 AVOID_FULL 和 AVOID_INDEX 组合，则 AVOID_INDEX_SJ 伪指令中指定的索引必须是 INDEX 伪指令中指定的索引的子集。有关 INDEX_SJ 和 AVOID_INDEX_SJ 伪指令的作用的更多信息，请参阅描述优化程序伪指令的 *GBase 8s 性能指南* 一章。

指定 MULTI_INDEX 或 INDEX_ALL 伪指令规避了指定表上统计信息的常见优化程序要求。在考虑表上的多索引扫描路径之前，优化程序通常至少需要对表进行低级统计。

**存取方法伪指令的示例**

假设您有一个名为 emp 的表，它包含列 emp_no 、dept_no 和 job_no ，并且在 dept_no 列上定义了 ids_dept_no 索引，在 job_no 列上定义了 idx_job_no 索引。当您执行在 FROM 子句中包含 emp 表的 SELECT 查询时，可能命令优化程序以以下几种方法存取该表：

- 例如使用正指令：

  SELECT {+INDEX(emp idx_dept_no)} ...

  在上述示例中，此存取方法伪指令强制优化程序考虑扫描 **dept_no** 列上的 **idx_dept_no** 索引的执行路径。

  在下面的示例中，存取方法伪指令强制优化程序考虑使用多索引扫描，它是基于扫描 **dept_no** 列上的 **idx_dept_no** 索引和 **job_no** 列上的 **idx_job_no** 索引的组合结果。

  SELECT {+MULTI_INDEX(emp idx_dept_no ids_job_no)} ...

- 例如使用负指令：

  SELECT {+AVOID_INDEX(emp idx_loc_no, idx_job_no), AVOID_FULL(emp)} ...

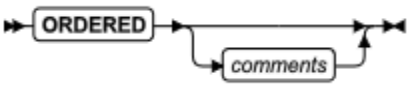
  该示例包含多个存取方法伪指令。这些伪指令通过指示优化程序不扫描 **idx_loc_no** 和 **idx_job_no** 索引，页不执行 **emp** 表的全表扫描来强制扫描 **dept_no** 列的 **idx_dept_no** 索引。但是，如果为表 **emp** 创建了一个新的 **idx_emp_no** ，则这些伪指令不会阻止优化程序考虑它。

还请注意，术语***负指令***引用存取方法伪指令中的字符串 "AVOID_" ，并且与开始每个优化程序指令的注释指示符之后的 + 号无关。

## 连接顺序伪指令

使用 ORDERED 连接顺序伪指令强制优化程序以它们在查询的 FROM 子句中出现的顺序连接表或视图。

连接顺序伪指令




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *comments* | 用于记录伪指令的文本 | 必须出现在注释符号之间 | 字符串 |

例如，下面的查询强制数据库服务器连接表 dept 和 job ，然后表结果和表 emp 连接起来：

```
SELECT --+ ORDERED
    name, title, salary, dname
  FROM dept, job, emp WHERE title = 'clerk' AND loc = 'Palo Alto'
    AND emp.dno = dept.dno
    AND emp.job= job.job;
```

因为在表 **dept** 和表 **job** 之间没有出现谓词，所以这个查询强制数据库服务器构造一个笛卡尔积。

当查询涉及视图时，ORDERED 连接顺序伪指令的位置决定所指定的是部分还是完全连接顺序。

- 创建视图时指定部分连接顺序

  如果在创建视图时使用 ORDERED 伪指令，基表就以视图定义的顺序相邻连接。

  对于后面所有关于视图的查询，数据库服务器以视图定义中指定顺序相邻地连接基表。当使用在视图中时，ORDERED 伪指令不影响查询中的 FROM 子句命名的其它表的连接顺序。

- 创建视图时指定完全连接顺序

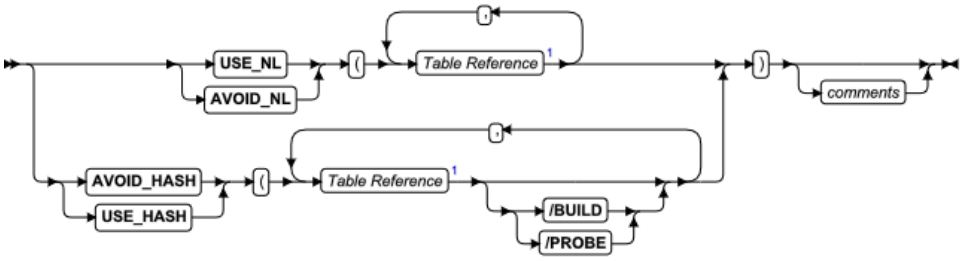
  当在使用视图的查询中指定 ORDERED 连接顺序伪指令时，所有表都以指定顺序连接，即使 是组成视图的那些表。如果视图包含在查询中，基表就以视图定义的顺序相邻地连接。ORDERED 用于示例的例子，可以参考 *GBase 8s 性能指南* 。

因为 OUTER 连接的顺序要求，在符合 ANSI 的连接查询中，指定 RIGHT OUTER JOIN 或 FULL OUTER JOIN 关键字，忽略 ORDERED 连接顺序伪指令，但它在解释输出文件中的 ***Directives Not Followed*** 下列出。

## 连接方法伪指令

使用连接方法伪指令影响表在 GBase 8s 扩展的连接查询中如何连接。

连接方法伪指令




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *comments* | 用于记录伪指令的文本 | 必须出现在注释符号之间 | 字符串 |

使用逗号或空格分隔括号内置的元素。

下表描述了每个连接方法伪指令。

| 关键字 | 作用 |
|---|---|
| USE_NL | 使用指定的表作为嵌套循环连接的内部表<br>如果在 FROM 子句中指定了 ***n*** 个表，那么 USE_NL 连接方法伪指令中最多可以指定（***n***-1）个表。 |
| USE_HASH | 使用哈希连接访问指定的表 |

也可以选择表是用来创建哈希列表或探测哈希表。

AVOID_NL　请勿在嵌套循环连接中使用指定的表作为内部表

这个伪指令列出的表仍然可以作为外部表参与嵌套循环连接。

AVOID_HASH　不要使用哈希连接访问指定的表

可以选择使用哈希连接，但是要对哈希连接中表的作用强行加以限制。

连接方法伪指令优先于 OPTCOMPIND 配置参数强制的连接方法。

当指定 USE_HASH 或 AVOID_HASH 伪指令（分别表示使用或避免使用哈希连接）时，也可以指定每个表的的作用：

- /BUILD

  和 USE_HASH 伪指令一起使用，这个关键字表示指定的表用来构造一个哈希表，和 AVOID_HASH 伪指令一起使用，这个关键字表示指定的表不用来构造哈希表。

- /PROBE

  和 USE_HASH 伪指令一起使用，这个关键字表示指定的表用来探测一个哈希表。和 AVOID_HASH 伪指令一起使用，这个关键字表示指定的表不用来探测一个哈希表。只要至少有一个表没有指定为 PROBE，就可以指定多个探测表。

为使优化程序找到有效的连接查询计划，必须至少为在连接中涉及的每个表运行 UPDATE STATISTICS LOW ，以提供适当的开销估计。否则，优化程序可能会选择将整个表广播给所有实例，即使表非常大。

如果没有指定 /BUILD 关键字也没有指定 /PROBE 关键字，那么优化程序使用成本估计来确定表的作用。

在这个例子中，USE_HASH 伪指令强制优化程序在 dept 表上构造一个哈希表，并且只考虑用哈希表来连接 dept 和其它表。因为没有指定其它伪指令，所以优化程序为查询中的其它连接选择花费最小的连接方法。

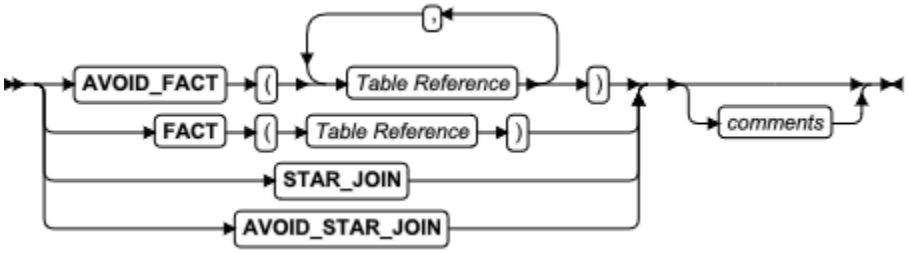
```
SELECT /*+ USE_HASH (dept /BUILD)
   The optimizer must use dept to construct a hash table */
   name, title, salary, dname
   FROM emp, dept, job WHERE loc = 'Phoenix'
   AND emp.dno = dept.dno    AND emp.job = job.job;
```

您为符合 ANSI 连接查询的指定的连接方法优化程序伪指令会被忽略，但是它在解释输出文件中的 ***Directives Not Followed*** 下列出。

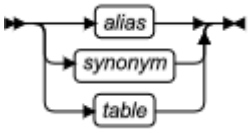
## 星型连接伪指令

使用星型连接伪指令指定优化程序应连接具有星型模式的表的方式。

星型连接伪指令




表引用




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *alias* | FROM 子句中声明的临时的可代替的表名 | 如果声明了 *alias*，那么它必须被使用（而不是使用 *table* 或 *synonym*） | 标识符 |
| *comments* | 用于记录伪指令的可选文本 | 必须在括号外但是注释符号内 | 字符串 |
| *synonym*, *table* | 要应用伪指令的表的名称或同义词 | 它指向的表和同义词必须存在 | 标识符 |

在指定多个表的 AVOID_FACT 伪指令中，使用逗号或空格分隔括号内的元素。

下表描述了每种星连接伪指令并说明了它是如何影响优化程序的查询计划。

| 关键字 | 作用 | 优化程序操作 |
| --- | --- | --- |
| AVOID_FACT | 必须指定至少一个表。不使用将表（或表列表中的任何表）作为星连接优化程序中的事实表 | 优化程序不考虑将指定的表（或表列表中的任何表）视为事实表的星型连接执行计划。 |
| AVOID_STAR_JOIN | 优化程序不会考虑星连接执行计划。 | 优化程序选择一个不是星型连接计划的查询执行计划。 |
| FACT | 必须指定正确的一个表。只能将指定的表视为星型连接执行计划中的事实表 | 这些优化程序考虑查询计划，其中指定的表是星型连接执行计划中的事实表。 |

| | | |
|---|---|---|
| STAR_JOIN | 如果可能，优先选择星型连接计划。 | 如果可能，优化程序优先选择星型连接计划。 |

星型连接伪指令要求启用并行数据库查询功能（PDQ）。当 PDQ 关闭时，禁用星型连接查询优化程序。

星型连接伪指令要求查询中的所有表具有至少低级别的统计。如果查询中的任何表的表统计信息不可用，则禁用星型连接优化程序

SQL 的 SET OPTIMIZATION ENVIRONMENT STAR_JOIN DISABLED 语句会禁用当前会话中的星型连接伪指令。（有关优化程序环境设置的其它信息，请参阅 ENVIRONMENT 选项。）

单独指定 FACT 伪指令不会自动支持星型连接执行计划。您可以通过指定 STAR_JOIN 伪指令和 FACT 伪指令的组合来指示优化程序更喜欢使用特定事实表的星型连接执行计划。

您可以在 SET EXPLAIN 语句的输出文件中查看一个查询的星型连接优化程序路径，也可以使用 GBase Data Studio 获得 Visual Explain 输出。

在集群环境中，星型查询优化程序伪指令在这些类型的辅助服务器上可用：

- 共享磁盘辅助服务器（SDS）
- 远程独立辅助服务器（RSS）
- 高可用数据复制辅助服务器（HDR）。

**星型伪指令的限制**

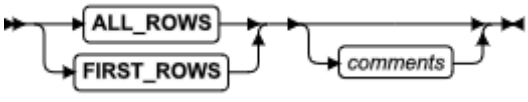
下列限制适用于尝试连接具有星型模式依赖关系的表的查询：

- 必须启用并行数据库查询（PQD）功能才能使星型连接指令有效。
- 查询中的所有表必须至少具有低级别统计信息。
- 星型连接伪指令不支持连接多个事实表。
- 当事务隔离级别为 Committed Read Last Committed 或 Cursor Stability 时，星型连接伪指令无效。（支持所有其它事务隔离级别。）

## 优化目标伪指令

使用优化目标伪指令来指定用于确定查询结果性能的方法。

优化目标伪指令




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *comments* | 用于记录伪指令的文本 | 必须出现在注释符号之间 | 字符串 |

两种优化目标伪指令是：

- FIRST_ROWS

  告诉优化程序选择一个对只查找满足查询的第一屏内容的进程进行优化的方案。使用这个选项来减少那些使用交互模式或只需要返回几行的查询的初始响应时间。

- ALL_ROWS

  这个伪指令告诉优化程序选择一个对查找满足查询的所有含的进行进行优化的方案。

  这种形式的优化是缺省的。

优化目标伪指令优先于 **OPT_GOAL** 环境变量设置和 OPT_GOAL 配置参数。

有关如何在整个会话中设置优化目标的信息，请参阅 SET OPTIMIZATION 语句。

下列上下文中不能使用优化目标伪指令：

- 在视图定义中
- 在子查询中

下列查询返回得到奖金最多的是前 50 位雇员名称。优化目标伪指令引导优化程序尽可能地返回第一屏内容。

```
SELECT {+FIRST_ROWS
    Return the first screenful of rows as fast as possible}
    LIMIT 50 fname, lname    FROM employees ORDER BY bonus DESC;
```

## 说明方式伪指令

使用说明方式伪指令来测试和调试查询计划并把关于查询计划的信息打印到说明输出文件。

说明方式伪指令


| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *comments* | 用于记录伪指令的文本 | 必须出现在注释符号之间 | 字符串 |

下面的表列出了每一个说明方式伪指令的作用。

| 关键字 | 作用 |
|---|---|
| EXPLAIN | 对指定的查询启动 SET EXPLAIN ON |
| AVOID_EXECUTE | 放置执行数据操作语句；代替为把查询计划打印到说明输出文件 |

EXPLAIN 伪指令主要用于测试和调试查询计划。在 SET EXPLAIN ON 已经有效的情况下，它是多余的。在视图定义或子查询中使用是不合法的。

下面的查询执行查询计划并把它打印到说明输出文件：

```
SELECT {+EXPLAIN} c.customer_num, c.lname, o.order_date
    FROM customer c, orders o WHERE c.customer_num = o.customer_num;
```

如果远程表是查询的一部分，AVOID_EXECUTE 伪指令防止在本地或远程位置执行查询。这个伪指令不会防止对查询中的不变函数求值。

下面的查询不返回数据，但把查询计划写到说明输出文件：

```
SELECT {+EXPLAIN, AVOID_EXECUTE}
 c.customer_num, c.lname, o.order_date
 FROM customer c, orders o WHERE c.customer_num = o.customer_num;
```

不执行查询，必须同时使用 EXPLAIN 和 AVOID_EXECUTE 伪指令来查看优化程序的查询计划（在说明输出文件中）。用来分隔这两个伪指令的逗号（,）是可选的。

如果在指定 AVOID_EXECUTE 伪指令时省略 EXPLAIN 伪指令，不会报错，但是查询计划不会写到说明输出文件，并且不执行 DML 语句。

下列上下文中不能使用说明方式伪指令：

- 在视图定义中
- 在触发器中
- 在子查询中

然而，在 INSERT 语句的 SELECT 语句中，它们是合法的。

### 外部伪指令

可以使用 SAVE EXTERNAL DIRECTIVES 语句将优化程序伪指令存储在系统目录的 **sysdirectives** 表中。 GBase 8s 自动将这些外部伪指令应用到随后的查询和符合 SELECT 语句的子查询中。

可以设置 EXT_DIRECTIVES 配置参数和 **IFX_EXTDIRECTIVES** 环境变量，以控制是为数据库服务器实例还是为会话启用或禁用外部伪指令。将这些设置为零将禁用外部伪指令；将两者设置为 1 启用外部伪指令。

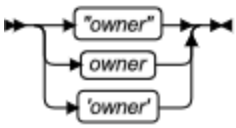
还可以使用 SET ENVIRONMENT 语句的 EXTDIRECTIVES 选项启用或禁用会话期间的外部例程。有关更多信息，请参阅为会话启用或禁用外部伪指令。

## 5.9 所有者名称

所有者名称指定数据库对象的所有者。当您在语法图表中引用所有者名称时使用本段。

**语法**

所有者名称



| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *owner* | 数据库中对象所有者的用户名 | 最大长度为 32 个字节 | 必须遵守操作系统的规则。 |

**用法**

在兼容 ANSI 的数据库中，必须指定您不拥有的数据库对象的***所有者***。在对数据库对象的引用中，所有者名称的 ANSI/ISO 同义词时***授权标识符***。（然而，在对模式对象的引用中， GBase 8s 文档调用的***所有者名称***的 ANSI/ISO 术语是***模式名称***。）

在不兼容 ANSI 的数据库中，***所有者***名称是可选的。当创建数据库对象或用户数据范围语句时，您不需要指定***所有者***。如果在创建数据库对象时不指定***所有者***，那么在大多数情况下，数据库服务器将您的登录名指定为对象的所有者。 有关此规则的例外，请参阅 CREATE FUNCTION 语句描述中的已创建数据库对象的所有权和 CREATE PROCEDURE 语句描述中的创建数据库对象的所有权。当在所有者特权 UDR 中的 DDL 语句创建一个新的数据库对象时，例程的所有者（而非执行它的用户，如果此用户不是例程的所有者）成为新数据库对象的所有者。

如果您在数据访问语句中指定***所有者***，那么数据服务器会检查它的正确性，不加引号时，所有者是不区分大小写的。下面的四个查询都可以访问表 kaths.tab1 的数据：

```
SELECT * FROM tab1;
SELECT * FROM kaths.tab1;
SELECT * FROM KATHS.tab1;
SELECT * FROM Kaths.tab1;
```

在兼容 ANSI 的数据库中，只有表的所有者，用户 **kaths**，可以发出第一个示例中的查询，其指定了一个未限定的表名，但是持有 **tab1** 上的 Select 特权的任何用户可以在不兼容 ANSI 的数据库中发出该查询。有关在兼容 ANSI 数据库中所有者名称的更多信息，请参阅符合 ANSI 的数据库的限制和区分大小写。

CREATE ROLE 语句声明的***角色***是授权标识符，因而会收到所有者名称的语法限制，但是角色不能是数据库对象的所有者。同样，关键字 PUBLIC，它指定所有用户的群组，不能是数据库对象的所有者，除了在特殊的 **sysdbopen( )** 和 **sysdbclose( )** 过程的情况中。有关这些内置会话配置 UDR 的更多信息，请参阅会话配置过程。

## 使用引号

当使用引号时，***所有者***是区分大小写的。换句话说，引号指示数据库服务器在您创建或访问数据库对象时，确切地按照输入读取或存储名称。例如，假设有一个表，它的所有者是 Sam。可以使用下面两个语句中的任何一个来访问表中的数据：

```
SELECT * FROM table1;
SELECT * FROM 'Sam'.table1;
```

第一个查询成功，因为不需要所有者名称。第二个查询成功，因为指定的所有者名称和存储在数据库中的所有者名称匹配。

## 引用 gbasedbt 用户拥有的表

如果使用所有者名称作为从一个系统目录表访问数据库对象信息的选择条件，则所有者名称是区分大小写的。要保留字母大小写，必须将所有者用单引号或双引号括起来，并且必须完全按照其存储在系统目录表中的方式键入所有者名称。在以下两个示例中，只有第二个成功访问表 Kaths.table1 上的信息。

```
SELECT * FROM systables WHERE tabname = 'tab1' AND owner = 'kaths';
SELECT * FROM systables WHERE tabname = 'tab1' AND owner = 'Kaths';
```

用户 gbasedbt 是系统目录表的所有者。并且当 SQL 语句引用系统目录时，在兼容 ANSI 的数据库中必须指定 gbasedbt 作为限定符，除非您是用户 gbasedbt：

```
SELECT * FROM "gbasedbt".systables WHERE tabname = 'tab1' AND owner = 'Kaths';
```

GBase 8s 接受以下任何符号，以指定符合 ANSI 的数据库的系统目录表：

- "gbasedbt"*.system_table*
- gbasedbt*.system_table*
- 'gbasedbt'*.system_table*

然而，在这三种格式中，只有第一种，其中所有者被指定为定界标识符，可以与大多数其它数据库服务器直接交互。对于不带分隔符的格式，SQL 的 ANSI/ISO 标准将小写字母升级为 GBASEDBT，同一标准不支持单引号（'）作为所有者名称或模式名称的有效分隔符。

相反， GBase 8s 将 **gbasedbt** 的名称视为一种特殊情况，并且在指定 **gbasedbt** 时保留小写字母，带或不带分隔符，无论数据库是否符合 ANSI。但是，要编写可移植到非 GBase 8s 数据库服务器的 SQL 代码，应始终使用双引号（" ）将数据库对象的所有者名称分隔开。

以下 SQL 示例使用未定界的所有者名称：

```
CREATE TABLE gbasedbt.t1(i SERIAL NOT NULL);
CREATE TABLE someone.t1(i SERIAL NOT NULL);
```

如果这些语句成功执行，第一个表将在 **systables** 中注册的 gbasedbt 作为所有者，第二个表将 SOMEONE 注册为所有者。当所有者的指定字母大小写，但所有者名称未限定时，字母大小写无关紧要，因为 GBase 8s 将未分隔的所有者名称变成大写，但将未定界的 **gbasedbt** （或 **GBASEDBT**）所有者名称变为小写的 gbasedbt。

例如，假设之前两个 CREATE TABLE 语句成功执行后，用户 gbasedbt 发出下列语句：

```
CREATE TABLE GBASEDBT.t1(i SERIAL NOT NULL);
```

该语句失败，因为*所有者*名称和*表*名称的组合不是唯一的，如果之前注册的 **gbasedbt** 用户拥有的表已经在数据库中存在。

**提示：** USER 操作符返回当前用户在系统上存储的登录名。如果所有者名称与登录名不同（例如，混合大小写所有者名称和全小写登录名），则 owner = USER 语法失败。

## 符合 ANSI 的数据库的限制和区分大小写

下表描述了当您创建、重命名或访问数据库对象时数据库服务器如何读取和存储*所有者*。

| 所有者名称规范 | 兼容 ANSI 的数据库的做法 |
| --- | --- |
| 忽略 | 严格按照登录名存储在系统中的方式读取或存储*所有者*，但如果用户不是*所有者*，则会返回错误。 |
| 不带引号指定 | 以大写字母读取或存储*所有者* |
| 包括在引号中 | 完全按照输入读取或存储*所有者* 。另见使用引号和引用 gbasedbt 用户拥有的表。 |

如果在兼容 ANSI 的数据库中创建或重命名数据库对象时指定所有者名称，必须在数据访问语句中包含所有者名称。当访问不属于您的数据库对象时您必须包含所有者名称。

因为如果所有者不在引号之间，则数据库服务器自动将*所有者*转换为大写字母，区分大小写错误会导致查询失败。例如，如果您是用户 nancy 并且使用以下语句，则产生的视图具有名称 nancy.njcust：

```
CREATE VIEW 'nancy'.njcust AS
SELECT fname, lname FROM customer WHERE state = 'NJ';
```

以下 SELECT 语句失败，因为它试图将 NANCY.njcust 和实际所有者和表名 nancy.njcust 相匹配：

```
SELECT * FROM nancy.njcust;
```

在 GBase 8s 分布式查询中，如果所有者名称不在引号中，则远程数据库遵循本地数据库的大小写约定。如果本地数据库是兼容 ANSI 的，则远程数据库将所有者名称处理为 ***uppercase***。如果本地数据库是不兼容 ANSI 的，则远程数据库将所有者名称处理为 ***lowercase***。

**提示：** 使用所有者名称作为查询中的选择标准之一（例如，WHERE owner = 'kaths'）时，必须确保带引号字符串和存储在数据库中的所有者名称完全匹配。如果数据库服务器找不到数据库对象或数据库，可能需要修改查询使引用字符串使用大写字母（例如，WHERE owner = 'KATHS'）。

因为所有者名称是授权标识符，而不是 SQL 标识符，因此可以在数据库的 SQL 语句中的单引号(')之间包含所有者，其中 **DELIMIDENT** 环境变量指定支持分隔标识符，从而需要双引号（"）围绕 SQL 标识符。

## 为兼容 ANSI 的数据库设置 ANSIOWNER

兼容 ANSI 的数据库的缺省行为是将在任何 ***owner*** 规范中的所有不在引号中的小写字母替换为大写字母。可以通过在数据库服务器初始化之前设置 **ANSIOWNER** 环境变量为 1 ，来阻止这个行为。这将保持您在不加引号指定 ***owner*** 字符串时所使用的任意大小写形式。

## 缺省所有者名称

如果您在不符合 ANSI 的数据库中创建数据库对象时未显式地指定所有者名称，则您的授权标识符（作为对象的缺省所有者）将存储到数据库的系统目录中，如同您已经加上引号指定您的授权标识（即，保持大小写形式）。

如果您在符合 ANSI 的数据库中创建建数据库对象时未显式地指定所有者名称，则您的授权标识符（作为对象的缺省所有者）将以大写字符存储到数据库的系统目录中，除非 **ANSIOWNER** 环境变量在数据库服务器初始化之前已设置为 1 。但如果 **ANSIOWNER** 已设置为 1 ，则数据库服务器将存储对象的缺省所有者作为您的授权标识符，保持其大小写形式。

### 所有者名称的大小写形式规则总结

要创建数据库对象，例如名为 mytab 的表，登录名为 Otho 的用户可以以下列几种方式声明新数据库对象的名称：

1. `CREATE TABLE mytab . . .`
2. `CREATE TABLE Otho.mytab . . .`
3. `CREATE TABLE "Otho".mytab . . .`

未分隔的所有者名称（第二个示例中）存储在 systables 系统目录表的 owner 列的形式取决于本地数据库是否符合 ANSI 。

- 在情况 1 中，未指定所有者名称。表的隐式所有者是 **Otho**，创建该表的用户，并且所有者名称以与所有者的用户标识相同的格式（**Otho**）存储在 **systables** 表中，与数据库的 ANSI 兼容状态无关。
- 在情况 2 中，指定未定界的所有者名称。**systables** 表对于不符合 ANSI 的数据库的数据库，将所有所有者名称字母存储为小写（此处为 **otho**）。对于兼容 ANSI 的数据库（其中不将 **ANSIOWNER** 设置为 1）， **systables** 表将所有所有者名称字母存储为大写（此处为 **OTHO**）。但是，如果 **ANSIOWNER** 设置为 1，则名称以与 DDL 语句中指定的相同大小写形式存储（此处为 **Otho**）。
- 在情况 3 中，定界的所有者名称按照其指定的相同的形式（此处为 **Otho**）存储在 **systables** 表中，与数据库的 ANSI 兼容状态无关。

请注意用户标识符是区分大小写的，但是数据库名称不区分大小写。因此，同一个用户不能用于表 tab 和表 TAB 。

除了这些示例中的 CREATE TABLE 语句之外，所有 SQL 语句和 SPL 语句都遵循这些规则。例如，使用 DROP TABLE 时，在处理语句时所有者名称出现的格式取决于以下相同的条件：

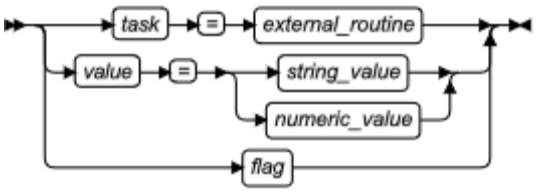
- 是否指定了显式所有者名称。
- 如果指定了显式所有者名称，是否使用引号将所有者名称分隔开。
- 如果没有使用引号将显式所有者名称分隔开，则数据库是否符合 ANSI 标准。
- 如果数据库是符合 ANSI 的，则在数据库初始化之前是否将 **ANSIOWNER** 设置为 1。

## 5.10 用途选项

GBase 8s 的 CREATE ACCESS_METHOD 、CREATE XADATASOURCE TYPE 和 ALTER ACCESS_METHOD 语句可以以下列语法指定用途选项。

**语法**

用途选项




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *external_routine* | 执行 *task* 的用户定义的函数 | 必须在数据库中注册 | 数据库对象名 |
| *flag* | 指出标志所启用的功能的关键字 | 接口指定标志名 | *Flag* 用途分类在用途函数、标志和值表中。 |
| *numeric_value* | 实数值 | 必须在数字数据类型范围内 | 精确数值 |
| *string_value* | 表示成一个或多个字符的值 | 字符必须是数据库代码集里面的 | 引用字符串 |
| *task* | 标识用途函数的关键字 | 可以对它分配函数（函数名不能和关键字相同）的关键字 | *Task* 用途分类在 用途函数、标志和值的表中。 |
| *value* | 标识配置选项的关键字 | 可以赋值的预定义配置关键字 | *Value* 用途分类在 用途函数、标志和值的表中。 |

**用法**

GBase 8s 支持在两类上下文中的用途选项：

- 定义或修改本地或远程表、视图和索引的主和辅存取方法
- 定义对符合 XA 的外部数据源的访问方法。

## 存取方法的用途选项

已注册的存取方法是一组属性，包含名称和称为 *purpose options* 的选项，它们可用于完成以下任务：

- 指定哪一个函数执行是数据访问和操作任务，如打开、读取和关闭一个数据源。
- 设置配置选项，例如存储空间类型。
- 设置标志、如允许 rowid 解释。

用 CREATE ACCESS_METHOD 语句创建存取方法时指定用途选项。要改变一个存取方法的用途选项，使用 ALTER ACCESS_METHOD 语句。

每一个 *task* 、*value* 或 *flag* 关键字对应一个 sysams 系统目录表中的列名。这些关键字允许设置下列属性：

- 用途函数

  ***purpose-function attribute*** 将用户定义的函数或方法的名称映射到 ***task*** 关键字，例如 **am_create** 、**am_beginscan** 或 **am_getnext**。这些关键字的完整列表，请参阅用途函数、标志和值中表中的 "Task" 分类。***external_routine*** 指定提供给存取方法的对于函数（C）。设置举例

  am_create = FS_create

- 用途标志

  ***purpose flag*** 指示存取方法是否支持一个给定的 SQL 语句或关键字。设置举例：

  am_rowids

- 用途值

  这些字符串、字符或数字值给出标志不能提供的配置选项。设置举例：

  am_sptype = 'X'

要允许一个用户定义函数或方法作为用途函数，必须首先使用 CREATE FUNCTION 语句注册执行适当任务的 C 函数或 Java™ 方法，然后把用途关键字设置成等价于已注册的函数或方法名。这将创建一个新的存取方法。 ALTER ACCESS_METHOD 语句页上的示例向现有的存取方法添加用途方法。

要允许使用用途标志，把名称指定为没有对应的值。

要清除 **sysams** 表中的用途选项设置，使用 ALTER ACCESS_METHOD 语句的 DROP 子句。

## 用途函数、标志和值

用途函数、方法和标志定义了存取方法的属性。

下表描述了 sysams 列可能的设置，包括用途函数或方法、标志和值。输入项出现的顺序和对应的 sysams 列相同。

表 1. 用途函数、用途标志和用途值

| 关键字 | 说明 | 类别 | 缺省值 |
|---|---|---|---|
| am_sptype | 一个字符，指定主要和辅助存取方法可以从哪一种类型的存储空间访问数据。am_sptype 字符可以具有下列一种设置：<br>• 'X' 表示只能访问外部空间的方法。<br>• 'S ' 表示只能访问 sbspace 的方法。 | 值 | 虚拟表接口（C）： 'A' |

| 关键字 | 说明 | 类别 | 缺省值 |
| --- | --- | --- | --- |
| | • 'A' 表示能够访问外部空间和 sbspace 的方法。<br>只有对新的存取方法是有效的。不能用 ALTER ACCESS_METHOD 更改或添加一个 **am_sptype** 值。不要把 **am_sptype** 设置为 'D' 或试图在 dbspace 中存储一个虚拟表。 | | |
| am_defopclass | 辅助存取方法的缺省运算符类。在定义运算符类之前，存取方法必须存在，然后在 ALTER ACCESS_METHOD 语句中设置这个值。 | 值 | 无 |
| am_keyscan | 如果设置了标志，它表示 **am_getnext** 返回辅助存取方法的索引键行。如果查询只选择索引键的列，数据库服务器不读取表而是使用辅助方法在共享存储器的索引键行。 | 标志 | 没有设置 |
| am_unique | 如果辅助存取方法支持检查单键，设置此标志 | 标志 | 没有设置 |
| am_cluster | 如果主或辅助存取方法支持表的集群，设置此标志 | 标志 | 没有设置 |
| am_rowids | 如果主或辅助存取方法可以从指定地址检索行，设置此标志 | 标志 | 没有设置 |
| am_readwrite | 如果辅助存取方法支持数据交换，设置此标志。如果没有设置缺省设置，表示虚拟表是只读的。如果应用程序要写数据，为 C 虚拟表接口设置此标志，避免产生下列问题：<br>• INSERT 、DELETE、UPDATE 或 ALTER FRAGMENT 语句导致 SQL 错误。<br>• 不执行函数 **am_insert** 、**am_delete** 或 **am_update** 。 | 标志 | 没有设置 |

| 关键字 | 说明 | 类别 | 缺省值 |
| --- | --- | --- | --- |
| am_parallel | 数据库服务器设置此标志以表示哪一个用途函数或方法可以在主或辅助存取方法中并行执行。如果设置，十六进制 am_parallel 位图包含一个或多个下列位设置：<br>• 第 1 位设置为可并行扫描。<br>• 第 2 位设置为可并行删除。<br>• 第 4 位设置为可并行修改。<br>• 第 8 位设置为可并行插入。<br>在 Java™ Virtual-Table Interface 中不支持插入、删除和修改。 | 标志 | 没有设置 |
| am_expr_pushdown | 启用使用参数描述符的标志 | 标志 | 没有设置 |
| am_costfactor | 数据库服务器把这个值乘以 am_scancost 用途函数或方法返回给主或辅助存取方法的成本。从 0.1 到 0.9 的 am_costfactor 值把成本减少到 am_scancost 计算得到的值的几分之一。 1.1 或更大的 am_costfactor 值增加 am_scancost 值 | 值 | 1.0 |
| am_create | 和用来创建虚拟表或虚拟索引的用户定义函数或方法（UDR）名相关联的关键字 | 任务 | 无 |
| am_drop | 和用来删除虚拟表或虚拟索引的 UDR 名相关联的关键字 | 任务 | 无 |
| am_open | 和用来使分片、extspace 或 sbspace 可用的 UDR 名相关联的关键字 | 任务 | 无 |
| am_close | 和反向 am_open 实行的初始化的 UDR 名相关联的关键字 | 任务 | 无 |
| am_insert | 和用来插入行或索引输入项的 | 任务 | 无 |

| 关键字 | 说明 | 类别 | 缺省值 |
|---|---|---|---|
| | UDR 名相关联的关键字 | | |
| am_delete | 和用来删除行或索引输入项的 UDR 名相关联的关键字 | 任务 | 无 |
| am_update | 和用来修改行或索引输入项的 UDR 名相关联的关键字 | 任务 | 无 |
| am_stats | 和用来建立基于存储空间值分布的统计信息的 UDR 名相关联的关键字 | 任务 | 无 |
| am_scancost | 和用来计算限定及检索数据成本的 UDR 名相关联的关键字 | 任务 | 无 |
| am_check | 和用来测试表的物理结构或执行索引的完整性检查的 UDR 名相关联的关键字 | 任务 | 无 |
| am_beginscan | 和用来建立扫描的 UDR 名相关联的关键字 | 任务 | 无 |
| am_endscan | 和用来反向建立 am_beginscan 初始化的 UDR 名相关联的关键字 | 任务 | 无 |
| am_rescan | 和用来扫描前一次扫描的下一项以完成一次连接或子查询的 UDR 名相关联的关键字 | 任务 | 无 |
| am_getnext | 和扫描满足查询的下一项需要的 UDR 名相关联的关键字 | 任务 | 无 |
| am_getbyid | 和从指定物理地址取回数据的 UDR 名相关联的关键字；am_getbyid 只可用于存取方法 | 任务 | 无 |
| am_truncate | 和删除虚拟表所有行（主存取方法）或删除虚拟索引所有对应键（辅助存取方法）的 UDR 名相对应的关键字 | 任务 | 无 |

下面的规则应用于 CREATE ACCESS_METHOD 和 ALTER ACCESS_METHOD 语句的用途选项规范：：

- 要在一个语句中指定多个用途选项，用逗号分隔。

- CREATE ACCESS_METHOD 语句必须指定和 **am_getnext** 关键字对应的用户定义的函数或方法名。

  ALTER ACCESS_METHOD 语句不能删除与 **am_getnext** 对应的函数或方法，但是可以修改它。

- ALTER ACCESS_METHOD 语句不能添加、删除或修改 **am_sptype** 值。
- 只能用 ALTER ACCESS_METHOD 语句指定 **am_defopclass** 值。

  在分配缺省运算符类之前，必须首先使用 CREATE ACCESS_METHOD 语句注册一个辅助存取方法。

## XA 数据源类型的用途选项

CREATE XADATASOURCE TYPE 语句指定用于访问来自符合 X/Open XA 标准的外部数据源的数据的目的函数。这些函数还使外部数据能够处理根据 GBase 8s 的事务语义进行处理。只有使用事务日志记录的数据库（如符合 ANSI 的数据库和支持显式事务的 GBase 8s 数据库）才支持事务协调。

下面的示例创建一个新的 XA 数据源类型 MQSeries®，其所有者是用户 gbasedbt。

```
CREATE XADATASOURCE TYPE 'gbasedbt'.MQSeries(
        xa_flags      = 1,
        xa_version    = 0,
        xa_open       = gbasedbt.mqseries_open,
        xa_close      = gbasedbt.mqseries_close,
        xa_start      = gbasedbt.mqseries_start,
        xa_end        = gbasedbt.mqseries_end,
        xa_rollback = gbasedbt.mqseries_rollback,
        xa_prepare    = gbasedbt.mqseries_prepare,
        xa_commit     = gbasedbt.mqseries_commit,
        xa_recover    = gbasedbt.mqseries_recover,
        xa_forget     = gbasedbt.mqseries_forget,
        xa_complete = gbasedbt.mqseries_complete);
```

这些值表示 XA Switch Structure 中的字段，如文件 **$GBASEDBTDIR/incl/public/xa.h** 中所列。此示例中的规范的顺序遵循 **sysxasourcetypes** 系统目录表中的列名称的顺序，但是它们可以按任何顺序列出，前提是不重复任何项目。**xa_flags** 和 **xa_version** 值必须是数字；其余的必须是事务管理器可以调用的 UDR 的名称。这些 UDR 必须已存在于数据库中，然后才能发出 CREATE XADATASOURCE TYPE 语句，以在其用途选项规范中引用它们。

DROP FUNCTION 或 DROP ROUTINE 语句不能删除在 CREATE XADATASOURCE TYPE 语句的目的选项中列出的 UDR ，直到删除使用 UDR 定义的所有 XA 数据源类型。

有关如何使用上一示例中的 UDR 来协调与外部 XA 数据源的事务的信息，请参阅 *GBase 8s DataBlade API 程序员指南*。

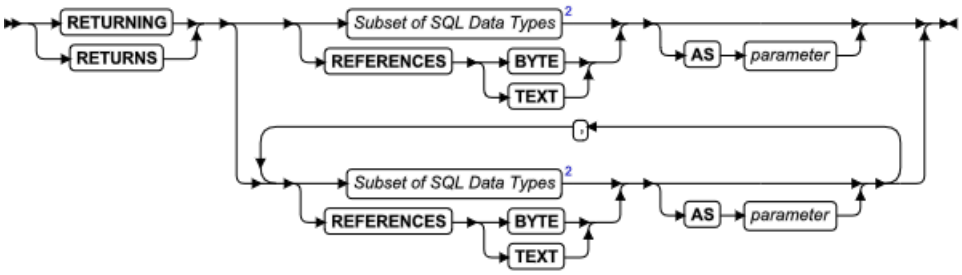
有关 MQDataBlade 模块的信息，请参阅 *GBase 8s 数据库扩展用户指南*。

# 5.11 返回子句

返回子句指定用户定义函数返回的一个或多个值的数据类型。可以在 UDR 定义中使用本段。

**语法**

返回子句




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *parameter* | 在这里为 UDR 返回的参数声明的名称 | 必须在 UDR 返回的*参数*中是唯一的。如果 UDR 任何一个返回值有名称，那么所有返回值都要有名称。 | 标识符 |

**用法**

在 GBase 8s 中，对于向后兼容性，您可以使用 CREATE PROCEDURE 语句创建 SPL 函数。（即，可以在 CREATE PROCEDURE 语句中包含 Return 子句）。但使用 CREATE FUNCTION 创建返回一个或多个值的新的 SPL 例程。

在返回子句表明返回何种数据类型以后，可以在语句块的任何位置使用 SPL 的 RETURN 语句，返回和返回子句中的值对应的 SPL 变量。

## 对返回值的限制

SPL 函数可以在 Return 子句中指定多个数据类型。

外部函数（以 C 或 Java™ 语言编写的函数）在 Return 子句中只能指定一个数据类型。但如果外部函数是迭代函数，那么它可以返回多行数据。有关更多信息，请参阅 ITERATOR。

## SQL 数据类型的子集

用户定义的函数（UDF）可以返回的内置 SQL 数据类型取决于语言。

*有关更多信息，请参阅下面的列表。另见*数据类型。

用给定的语言编写的 UDF 可以返回除下表中标记为 X 的类型的任何数据类型的值。

| 数据类型 | C | Java™ | SPL |
|---|---|---|---|
| BIGSERIAL | X | X | X |

| | | | |
|---|---|---|---|
| BLOB | X | | |
| CLOB | X | | |
| BYTE | X | X | |
| TEXT | X | X | |
| COLLECTION | | X | |
| LIST | | X | |
| MULTISET | | X | |
| ROW | | X | |
| SET | | X | |
| SERIAL | X | X | X |
| SERIAL8 | X | X | X |

在 GBase 8s ，中如果在 Return 子句中使用复杂数据类型，那么发出调用的用户定义例程必须定义相应的复杂类型的变量，以容纳 C 或 SPL 用户定义函数返回的值。

用户定义的函数可以返回数据库中定义的 opaque 或 distinct 数据类型的值。

SPL 函数返回的 DECIMAL 值的缺省精度是 16 位数字。要让函数以不同的有意义的数字位数返回 DECIMAL ，您必须在 Return 子句中显式地指定返回精度。

## 使用 REFERENCES 子句指向一个简单大对象

用户定义函数不能返回一个 BYTE 或 TEXT 值（总称为简单大对象）。然而，用户定义函数可以使用 REFERENCES 关键字，返回一个包含 BYTE 或 TEXT 对象指针的描述符。下面的例子说明了如何在 SPL 例程中选择 TEXT 列然后返回一个值：

```
CREATE FUNCTION sel_text()
RETURNING REFERENCES text;
DEFINE blob_var REFERENCES text;
SELECT blob_col INTO blob_var
FROM blob_table WHERE key_col = 10;
RETURN blob_var;
END FUNCTION;
```

对于作为查询的 Projection 列表中的列值的简单大对象，如在此示例中，返回的描述符中的指针根据 BYTE 或 TEXT 列定义从系统目录引用 **sysblobs.spacename** 值。

然而，对于不对应于永久表的列的简单大对象，指针引用定义了 UDR 的数据库的 dbspace 。这是当数据库服务器不指定 **sysblobs** 表的位置时，UDR 返回的 BYTE 或 TEXT 对象的缺省存储位置。

以下示例中的 DB-Access 会话创建两个例程 udr1 和 udr2，每个返回一个 TEXT 对象：

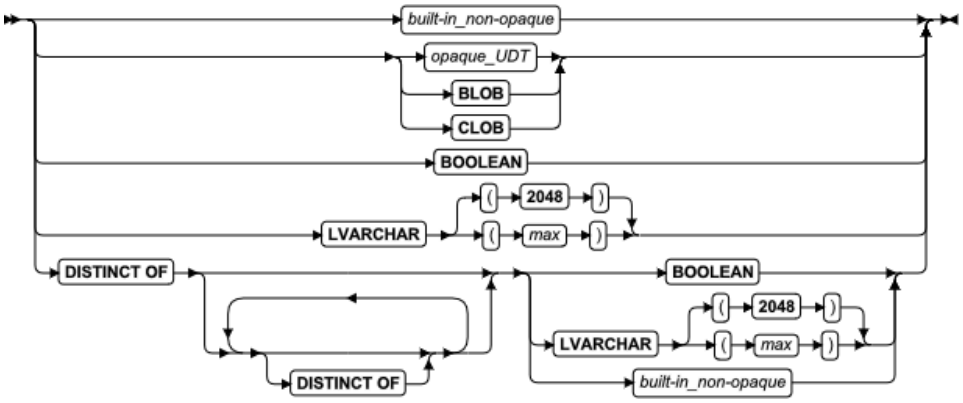
```
CREATE DATABASE db WITH LOG;

    CREATE TABLE t (c2 TEXT);
```

```
CREATE TABLE t1 (c2 TEXT);
LOAD FROM "t.unl" INSERT INTO t;

CREATE FUNCTION udr1 ( param_1
REFERENCES TEXT DEFAULT NULL )
RETURNING REFERENCES TEXT
WITH (NOT VARIANT)
DEFINE var1 REFERENCES TEXT;
ON EXCEPTION
RETURN param_1;
END EXCEPTION;
SELECT t.c2 udr1_col1
INTO var1 FROM t;
RETURN var1;
END FUNCTION;

CREATE PROCEDURE udr2 ( OUT param_1
REFERENCES TEXT DEFAULT NULL )
RETURNING INT;
SELECT t.c2 udr1_col1
INTO param_1 FROM t;
RETURN 1;
END PROCEDURE;

SELECT udr1(t.c2) query_1_col1 FROM t
INTO TEMP mytemp;

SELECT c2, slv1 FROM t1
WHERE udr2(slv1#TEXT) > 0
INTO TEMP mytemp;
```

在调用这些 UDR 的 SELECT 语句中，每个查询返回到 mytemp 临时表的 TEXT 对象存储在 db 数据库的 dbspace 中。

## 从另一个数据库返回值

对于存取本地数据库值为的表和视图的 UDR，只有下列数据类型可作为返回值：




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *built-in _ non-opaque* | 非 Opaque 的内置数据类型的名称 | 不能是 BIGSERIAL、BYTE 、SERIAL 、SERIAL8 或 TEXT | 数据类型 |
| *max* | 最大大小（字节）。缺省值为 2048。 | 必须是整数，1 ≤ *max* ≤ 32,739 | 精确数值 |
| *opaque_UDT* | 用户定义的 Opaque 数据类型的名称 | 必须显式强制转型为内置类型，通过在每个参与的数据中定义强制转型 | 标识符 |

如果 Return 子句从本地 GBase 8s 示例的另一个数据库返回一个值（或多个值，在 SPL 函数的情况中），返回的数据类型支持为下列数据类型：

- 不 Opaque 的内置数据类型
- 大多数***内置 opaque 数据类型***，在跨数据库事务中的数据类型 中列出
- 基于在此列表中标识的内置类型的 DISTINCT 类型
- 基于此列表中任一 DISTINCT 类型的 DISTINCT 类型
- 显式转换为此列表中的任一数据类型的用户定义类型（UDT）

UDF 和所有的 DISTINCT 类型、透明 UDT 、不数据类型层次结构和强制转型在每个数据库中必须具有相同的定义。相同的数据类型限制适用于外部函数从本地 GBase 8s 实例的另一个数据库返回的值。有关跨同一个数据库服务器实例的两个或多个数据库的分布操作中支持的数据类型的详细信息，请参阅跨数据库事务中的数据类型。有关在分布式事务中对 DISTINCT 数据类型有效的数据类型层次结构，请参阅分布式操作中的 DISTINCT 类型。

但是，从其他 GBase 8s 实例的数据库，UDF 只能指定以下作为参数或返回的数据类型：

- 内置不透明的数据类型
- BOOLEAN
- LVARCHAR
- DISTINCT 的透明的内置类型
- DISTINCT BOOLEAN
- DISTINCT LVARCHAR

- DISTINCT 在此列表中的 DISTINCT 类型

UDF 的定义和任何数据类型层次结构、强制转型和 DISTINCT 类型必须在每个参与的数据库中一致。除了在上一列表中标识的 BOOLEAN 、DISTINCT 和 LVARCHAR 数据类型之外，UDF 不能在跨服务器函数调用中返回其他内置不透明数据类型或不透明 UDT。

有关跨两个或多个 GBase 8s 实例的分布式操作中支持的数据类型的详细信息，请参阅跨服务器事务中的数据类型。有关在分布式事务中对 DISTINCT 数据类型有效的数据类型层次结构，请参阅 分布式操作中的 DISTINCT 类型。

### 命名返回参数

可以为 SPL 例程返回的参数或外部函数返回的单个值声明名称。

如果 SPL 例程返回多个值，您必须为所有返回参数声明名称，否则就一个都不声明。名称必须唯一。这里是一个已命名参数的实例：

```
CREATE PROCEDURE p (inval INT DEFAULT 0)
 RETURNING INT AS serial_num,
 CHAR(10) AS name,
 INT AS points;
 RETURN (inval + 1002), "Newton", 100;
 END PROCEDURE;
```

执行此 UDR 将会返回：

```
serial_num      name        points
1002             Newton     100
```

返回参数名和例程实体中的任何变量名之间没有关系。例如，可以定义一个函数返回 INTEGER as **xval**，但是在同一函数中，叫做 **xval** 的变量可以是 INTERVAL YEAR TO MONTH 数据类型。

### 游标函数和非游标函数

***游标***函数允许从返回值生成的结果集中反复依次取得各返回值。这样的函数是***隐式迭代函数***。

只返回一组值（如表的一行中的一列或几列）的函数是***非游标***函数。

返回子句可以出现在游标函数或非游标函数中。在下面的例子中，返回子句如果出现在非游标函数中，可以返回零个（0）或一个值。但如果这个子句和游标函数相关联，则它会返回表的多行，返回的每一行包含零个或一个值：

```
RETURNING INT;
```

在以下示例中，Return 子句如果出现在非游标函数中，可以返回零个（0）或两个值。然而，如果这个子句和游标函数相关联，则它会返回表的多行，返回的每一行包含零个或两个值：

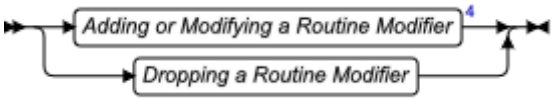
```
RETURNING INT, INT;
```

在前面的两个示例中，接收函数或程序都必须适当编写以接受函数返回的信息。

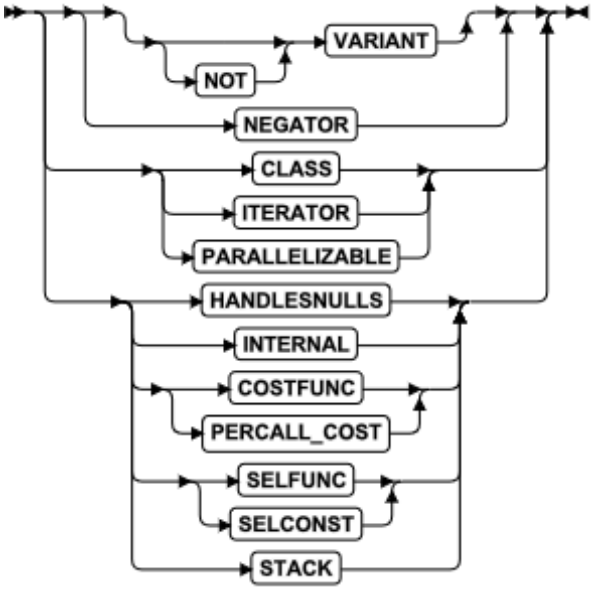
## 5.12 例程修饰符

例程修饰符指定用户定义的例程（UDR）如何工作的特征。

**语法**




删除例程修饰符




| 元素 | 描述 | 限制 | 语法 |
| --- | --- | --- | --- |
| *parameter* | 在这里为 UDR 返回的参数声明的名称 | 必须在 UDR 返回的参数中是唯一的。如果 UDR 任何一个返回值有名称，那么所有返回值都有名称。 | 标识符 |

**用法**

如果在 ALTER FUNCTION 、ALTER PROCEDURE 或 ALTER ROUTINE 语句中删除修饰符，那么如果存在缺省值，数据库服务器就会将修饰符的值设置为缺省值。

有些修饰符只可用于用户定义函数。关于指定的例程修饰符是否只能用于用户定义函数（即，是否不能用于用户定义的过程），请参阅后面一节对修饰符的说明。在这些部分（如同本手册的其它地方）中，外部指的是以 C 或 Java™ 语言编写的 UDR 。只对于一种语言有效的功能在前面的图表中作这样的指定。

除了 VARIANT 和 NOT VARIANT 修饰符，在此段中的其它选项对于 SPL 例程都无效。

**示例**

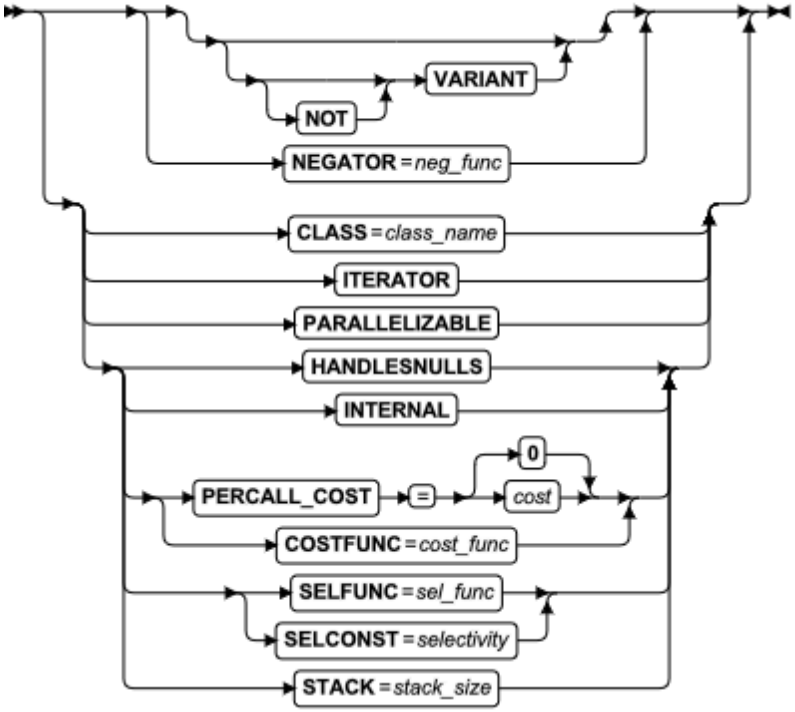
以下语句包含了 Java 语言的外部例程引用。您必须首先使用过程 install_jar（<*absolute path*><*jar file name*>,<*internal registered name*>）注册 *demo_jar*。

```
CREATE FUNCTION delete_order(int) RETURNING int
WITH (NOT VARIANT)
EXTERNAL NAME 'gbasedbt.demo_jar:delete_order.delete_order()'
LANGUAGE JAVA;
```

## 添加或修改例程修饰符

在 ALTER FUNCTION 、ALTER PROCEDURE 或 ALTER ROUTINE 语句中使用此段添加或修改 UDR 的例程修饰符的值。

添加或修改例程修饰符




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *class_name* | 运行外部例程的虚拟处理器（VP） | 任何 C UDR 都必须在 CPU VP 或用户定义的 VP 类中运行 | 引用字符串. |
| *cost* | 每次调用 C 语言的 UDR 的 CPU 使用成本。缺省值为 0 。 | 整数： 1 ≤ *cost* ≤ 231-1 （最高成本）。 | 精确数值 |
| *cost_func* | 要调用的伴随用户定义成本函数名 | 必须具有和 UDR 相同的所有者。需要拥有 Execute 特权。 | 标识符 |
| *neg_func* | 可以代替 UDR 调用的否定函数 | 必须具有和 UDR 相同的所有者。需要拥有 | 标识符 |

| | | | |
|---|---|---|---|
| | | Execute 特权。 | |
| *sel_func* | 要调用的伴随用户定义选择性函数名 | 必须具有和 UDR 相同的所有者。需要拥有 Execute 特权。 | 标识符 |
| *selectivity* | 每次调用 C 语言的 UDR 的 CPU 的使用成本。缺省值为 0。 | 请参阅 选择性的概念. | 精确数值 |
| *stack_size* | 执行 C 语言的 UDR 的线程堆栈大小（以字节计算） | 必须是正整数 | 精确数值 |

可以用任意顺序添加这些修饰符。如果同一修饰符列出多次，那么最后的设置会覆盖前面所有的值。

## 修饰符说明

下面几节说明了可以用来帮助数据库服务器最好地执行 UDR 的修饰符。

## CLASS

使用 CLASS 修饰符指定运行外部例程的虚拟处理器（VP）类的名称。用户定义的 VP 类必须在调用 UDR 之前定义。

可以使用下列几种 VP 类执行 C UDR：

- CPU 虚拟处理器类（CPU VP）
- 用户定义的虚拟处理器类。

如果省略 CLASS 修饰符来为用 C 语言编写的 UDR 指定 VP 类，那么 UDR 就在 CPU VP 中运行。用户定义的 VP 类可以保护数据库服务器不受恶意工作的 C UDR 影响。行为不良的 C UDR 至少具有下列特征中的一个：

- 长时间运行在 CPU VP 中不肯退出。
- 不是安全线程。
- 调用不安全的操作系统例程。

正常工作的 C UDR 不具有这些特征中的任何一项。在 CPU VP 中只执行正常工作的 C UDR。

**警告：** 在 CPU VP 中执行表现不佳的 C UDR 会导致对数据库服务器运行的严重干扰，并且 UDR 可能不会产生正确的结果。有关表现不佳的 UDR 的讨论，请参阅 *GBase 8s DataBlade API 程序员指南*。

缺省情况下，用 Java™ 编写的 UDR 在 Java 虚拟处理器类（JVP）中运行。因此，CLASS 修饰符对 Java 编写的 UDR 是可选的。然而，在注册一个用 Java 编写的 UDR 时使用 CLASS 修饰符，可以提高 SQL 语句的可读性。

## COSTFUNC (C)

使用 COSTFUNC 修饰符指定 UDR 的成本。UDR 的*成本*是对所需执行时间的估计。

有时候，UDR 的成本取决于它的输入。在这种情况下，可以使用用户定义函数来计算取决于输入值的成本。

要执行 *cost_func*，您必须拥有对它以及对 UDR 的 Execute 特权。

## HANDLESNULLS

使用 HANDLESNULLS 修饰符指定 C UDR 可以处理作为参数传递给它的 NULL 值。如果不对 C 语言的 UDR 指定 HANDLESNULLS ，并且传递它的参数具有 NULL 值，那么 UDR 不执行，并返回一个 NULL 值。

缺省情况下，C 语言的 UDR 不处理 NULL 值。

HANDLESNULLS 修饰符不可用于 SPL 例程，因为 SPL 例程缺省情况下处理 NULL 值。

## INTERNAL

对外部例程使用 INTERNAL 修饰符，表示 SQL 或 SPL 语句不能调用外部例程。在例程解析期间不考虑将外部例程指定为 INTERNAL 。对定义存取方法和语言管理等的外部例程使用 INTERNAL 修饰符。

缺省情况下，外部例程不是 internal；也就是说，SQL 或 SPL 语句可以调用例程。

## ITERATOR

对外部例程使用 ITERATOR 修饰符，表示该函数是***迭代函数***。迭代函数是每次函数调用返回单个元素的返回一组数据的函数；也就是说，它的调用包含一个初始调用和零个或多个后续调用，直到完成这一组。

缺省情况下，外部 C 或 Java™ 语言函数不是迭代函数。

SPL 迭代函数需要 RETURN WITH RESUME 语句，而不是 ITERATOR 修饰符。

在 ESQL/C 中，迭代函数需要游标。游标允许客户端应用程序用 FETCH 语句一次检索一个值。

有关如何编写迭代函数的更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 和 *GBase 8s DataBlade API 程序员指南*。

有关在查询的 FROM 子句中通过虚拟表接口使用迭代函数的信息，请参阅 迭代器函数。

## NEGATOR

对返回布尔值的 UDR 使用 NEGATOR 修饰符。

NEGATOR 修饰符为当前函数命名一个伴随用户定义函数，叫做***否定函数***。否定函数以相同的顺序采用相同的参数作为它的伴随函数，但是返回布尔补数。

也就是说，如果一个函数对一组给定的参数返回 TRUE ，那么以相同顺序传递相同的参数时，它的否定函数返回 FALSE 。例如，下面的函数就是否定函数：

equal(a,b)

notequal(a,b)

两个函数都以相同顺序接受相同的参数，但是返回互补的布尔值。在效率更高时，优化程序可以使用否定函数代替指定的函数。

要调用具有否定函数的用户定义函数，必须对两者都有执行权限。此外，函数的所有者必须和它的否定函数相同。

## PARALLELIZABLE

使用 PARALLELIZABLE 修饰符，表示外部例程可以在并行数据查询（PDQ）的上下文中并行执行。

缺省情况下，外部例程是不能并行执行的；即，它要按顺序执行。

如果 UDR 具有复杂或智能大对象数据类型作为参数或返回值，则不能使用 PARALLELIZABLE 修饰符。

如果对一个不能并行的外部例程指定 PARALLELIZABLE 修饰符，数据库服务器会返回一个运行时错误。

只调用 PDQ 线程安全 DataBlade API 函数的 C 语言 UDR 是可并行的。这些类别的 DataBlade API 函数是 PDQ 线程安全的：

- 数据处理

  这个类别的一个例外是集合操作函数（**mi_collection_***）不是 PDQ 线程安全的。
- 会话、线程和事务管理
- 函数执行
- 内存管理
- 异常处理
- 回叫
- 其它

每种类别 DataBlade API 函数的详细信息，请参阅 *GBase 8s DataBlade API 函数参考*。

如果 C 语言 UDR 调用不包含在这些类别之一中的函数，那么它就不是 PDQ 线程安全的，因此不是可并行的。

要并行 Java™ 语言 UDR 调用，数据库服务器必须具有多个 JVP 实例。用 Java 语言编写的打开一个 JDBC 连接的 UDR 不是可并行的。

## PERCALL_COST (C)

使用 PERCALL_COST 修饰符指定每次执行 C 语言 UDR 导致的近似的 CPU 使用成本。

优化程序使用指定的成本来确定评估 UDR 中 SQL 谓词的顺序以获得最佳性能。例如，下面的查询有两个谓词，由逻辑 AND 连接：

```
SELECT * FROM tab1 WHERE func1() = 10 AND func2() = 'abc';
```

在此示例中，如果一个谓词返回 FALSE，则优化程序就不需要评估另一个谓词了。

优化程序使用指定的成本来排列谓词的顺序，以便成本最小的谓词可以最先评估。CPU 使用成本必须是在 1 和 231-1 之间的整数，1 是最低成本，231-1 是最高成本。

要计算每次调用的近似成本，把下面两个数字相加：

- 每次调用 C UDR 执行代码的行数
- 需要一次 I/O 访问的谓词数

一次执行的缺省成本是 0。当删除 PERCALL_COST 修饰符时，一次执行的成本回到 0。

## SELCONST (C)

使用 SELCONST 修饰符指定 UDR 的选择性，UDR 的选择性是对查询选择的行所占分数的估计。

选择性常数的值 **selconst** 是在 0 和 1 之间的浮点数，代表希望 UDR 返回 TRUE 的行所占的分数。

## SELFUNC (C)

在 C UDR 中使用 SELFUNC 修饰符，为当前 UDR 命名一个伴随用户定义函数，称为选择性函数。选择性函数为优化程序提供当前 UDR 的选择性信息。

UDR 的选择性是对查询选择的行所占分数的估计。即，它是对 UDR 执行次数的估计。

要执行 *sel_func*，您必须具有对它以及 UDR 的 Execute 特权。

### *选择性的概念*

***选择性***指的是符合基于等式条件执行搜索的查询的属性。查询的选择性反过来取决于合格行的比例。FROM 子句中表对象所有行中限定行的比例越小。查询的选择性越高。

例如，下面的查询有一个基于 customer 表的 customer_num 列的搜索条件：

```
SELECT * FROM customer WHERE customer_num = 102;
```

因为表中的每行具有不同的客户编号，所以查询的选择性很高。相反，下面的查询具有低选择性：

```
SELECT * FROM customer WHERE state = 'CA';
```

因为 **customer** 表的大多数行都是 California 的顾客，所以会返回超过半数的行。

### *SELFUNC 修饰符的限制*

用 SELFUNC 指定的选择性函数具有特定要求。

指定的选择性函数必须满足以下条件：

- 必须采用与当前 UDR 相同数量的参数。
- 每个参数的数据类型必须是 SELFUNCARGS。

- 必须返回一个范围在 0 到 1 之间的 FLOAT 类型的值。它代表函数选择性百分比。（1 是高选择性；0 是没有选择性。）
- 它可以用数据库服务器支持的任何语言编写。

调用 UDR 的用户必须对这个 UDR 和 SELFUNC 修饰符指定的选择性函数都具有执行特权。

UDR 和此选择性函数必须具有相同的所有者。

有关如何使用 **mi_funcarg*** 函数来提取选择性函数的参数信息，请参阅 *GBase 8s DataBlade API 程序员指南*。

## STACK (C)

同 C UDR 一起使用 STACK 修饰符，覆盖由 STACKSIZE 配置参数指定的缺省堆栈打下。

STACK 修饰符指定线程堆栈的大小（以字节单位），执行 UDR 的用户线程用线程堆栈来保存信息，如例程参量和函数返回值。

UDR 需要足够的堆栈空间来容纳所有本地变量。对一个待定的 UDR，可能需要指定比缺省值更大的堆栈打下以防止堆栈溢出。

当包含 STACK 修饰符的 UDR 执行时，数据库服务器会分配大小为指定字节数的线程堆栈。一旦 UDR 执行结束，后面的 UDR 以 STACKSIZE 配置参数指定的堆栈打下（除非后面的 UDR 中任何一个也指定 STACK 修饰符）在线程中执行。

更多有关线程堆栈的信息，请参阅 *GBase 8s 管理员指南* 和 *GBase 8s DataBlade API 函数参考*。

## VARIANT 和 NOT VARIANT

对 C 用户定义的函数和 SPL 函数使用 VARIANT 和 NOT VARIANT 修饰符。如果以相同参数调用时返回不同结果或者修改数据库或变量状态的函数是可变的。例如，返回当前日期和时间的函数就是一个可变函数。

缺省情况下，用户定义函数是可变的。如果在创建或修改用户定义函数时指定 NOT VARIANT ，则该函数不能包含任何 SQL 语句。

如果用户定义函数是不可变的，那么数据库服务器可以存储成本高的函数的返回值。只能对不变函数创建函数型索引。有关函数型索引的更多信息，请参阅 CREATE INDEX 语句。

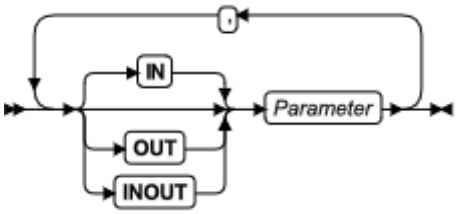
在 ESQL/C 中，可以在这个子句或 EXTERNAL 例程引用中指定 VARIANT 或 NOT VARIANT。有关更多信息，请参阅外部例程引用。如果在两处都指定修饰符，必须在两个子句中都使用同一修饰符。

# 5.13 例程参数列表

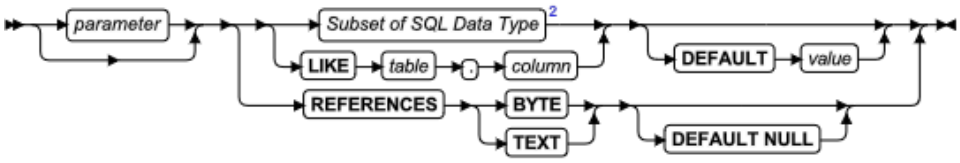
当看到在语法图表中引用例程参数列表时，使用例程参数列表段的适当部分。

**语法**

例程参数列表




参数




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *column* | 数据类型声明为 *parameter* 的列名 | 在指定表中必须存在 | 数据库对象名 |
| *parameter* | UDR 的参数的名称 | SPL 例程需要名称 | 标识符 |
| *table* | 包含 *column* 的表 | 表必须存在于数据库中 | 标识符 |
| *value* | 如果 UDR 调用对 *parameter* 没有值时缺省使用 | 必须是和 *parameter* 相同数据类型的文字，对于不透明类型，必须定义一个输入函数 | 精确数值 |

**用法**

*参数*是 UDR 声明中的形式参数。（接着调用一个带有参数的 UDR 时，必须用实际参量代替参数，除非参数具有缺省值。）

参数名对于 GBase 8s 的外部例程是可选的。

当创建 UDR 时，为每个参数声明 ***name*** 和 ***data type***。您可以直接指定数据类型，或者使用 LIKE 或 REFERENCES 子句指定数据类型。您可以可选地指定缺省值。

可以定义任意数量的 SPL 例程参数，但是传递给 SPL 例程的所有参数的总长度必须小于 2 千兆字节。

在传递给以 Java™ 语言编写的 UDR 的参量中，不能有多于 9 个是 UDR 声明为 Java 语言 BigDecimal 数据类型的 SQL DECIMAL 数据类型。

任何返回透明数据类型的 C 语言 UDR 必须在 C 主变量 `var binary` 声明中指定 **opaque_type**。

## SQL 数据类型的子集

连续和大对象数据类型作为参数是不合法的。UDR 可以声明一个在数据库中定义过的任意数据类型的参数，包括除 BIGSERIAL 、BLOB 、BYTE 、CLOB 、SERIAL 、SERIAL8 或 TEXT 以外的任何内置数据类型。

在 GBase 8s 中，参数也可以是复制数据类型或 UDT ，但复杂数据类型对于用 Java™ 语言编写的外部 UDR 的参数无效。

有关 GBase 8s 数据类型的信息，这些数据类型作为访问本地数据库外部表或视图的例程的参数或返回值，请参阅从另一个数据库返回值。

## 使用 LIKE 子句

使用 LIKE 子句指定参数的数据类型，它与数据库中定义的列相同。如果 ALTER TABLE 语句改变列的数据类型，那么参数的数据类型也会改变。

在 GBase 8s 中，如果使用 LIKE 子句声明参数，就不能重载 UDR。例如，假设您创建以下用户定义过程：

```
CREATE PROCEDURE cost (a LIKE tableX.colY, b INT)
        . . .
        END PROCEDURE;
```

在同一个 GBase 8s 数据库中不能创建另一个名为 ***cost( )*** 的带有两个参数的过程。然而，可以创建一个名为 ***cost( )*** 带有一个参数而不是两个参数的过程。（另一种克服 LIKE 子句限制的方法是通过用户定义的数据类型。）

## 使用 REFERENCES 子句

使用 REFERENCES 子句指定包含 BYTE 或 TEXT 数据的参数。REFERENCES 关键字允许您使用 BYTE 或 TEXT 对象的指针作为参数。如果在 REFERENCES 子句中使用 DEFAULT NULL 选项，并且不带参数地调用 UDR ，那么 NULL 值就用作缺省值。

## 使用 DEFAULT 子句

使用 DEFAULT 关键字后面跟一个表达式自动参数的缺省值。如果为参数提供缺省值，并且以少于 UDR 定义的参量来调用这个 UDR 时，就是要缺省值。如果不为参数提供缺省值，并且以少于 UDR 定义的参数来调用这个 UDR 时，调用应用程序会收到错误。

下面的示例显示了 CREATE FUNCTION 语句，它为参数指定了缺省值。这个函数查找参数 *i* 的平方。如果函数调用时不指定 *i* 参数的参量，数据库服务器使用 0 作为参数 *i* 的缺省值。

```
CREATE FUNCTION square_w_default
(i INT DEFAULT 0) {Specifies default value of i}
RETURNING INT; {Specifies return of INT value}
DEFINE j INT; {Defines routine variable j}
LET j = i * i; {Finds square of i and assigns it to j}
RETURN j; {Returns value of j to calling module}
```

```
END FUNCTION;
```

警告： 当指定一个日期值作为参数的缺省值时，必须确保对年份指定 4 位数而不是 2 位数字。当指定 2 位数字的年份时，环境变量 **DBCENTURY** 的设置会影响数据库服务器如何解释日期值，所以 UDR 可能不使用希望的缺省值。有关更多信息，请参阅 *《GBase 8s SQL 指南：参考》*。

## 为用户定义例程指定 OUT 参数

注册一个 GBase 8s 的用户定义的例程时，可以使用关键字 OUT 来指定列表中的任何一个参数是 OUT 参数。每个 OUT 参数对应例程通过指针直接返回的一个只。例程通过指针返回的值是显式返回的任何值以外的一个额外值。

在注册了带有一个或多个 OUT 参数的用户定义函数以后，可以在 SQL 语句中把函数和语言-局部变量（SLV）一起使用。（关于语句-局部变量的信息，请参阅语句本地的变量表达式。）

如果指定了任何 OUT 参数，并且使用 GBase 8s 样式的参数，参量会通过引用传递给 OUT 参数。OUT 参数在确定例程特征符时没有意义。

例如，下面对 C 用户定义函数的声明允许通过参数 y 返回一个额外值：

```
int my_func( int x, int *y );
```

用 CREATE FUNCTION 语句注册一个 C 函数与此类似：

```
CREATE FUNCTION my_func( x INT, OUT y INT )
        RETURNING INT
        EXTERNAL NAME "/usr/lib/local_site.so"
        LANGUAGE C
        END FUNCTION;
```

下一个示例中，该 Java™ 方法通过传递返回一个额外值：

```
public static String allVarchar(String arg1, String[] arg2)
        throws SQLException
        {
        arg2[0] = arg1;
        return arg1;
        }
```

要将这作为 UDF 注册，请使用与以下示例相似的语句：

```
CREATE FUNCTION all_varchar(VARCHAR(10), OUT VARCHAR(7))
RETURNING VARCHAR(7)
WITH (class = "jvp")
EXTERNAL NAME 'gbasedbt.testclasses.jlm.Param.allVarchar(java.lang.String,
  java.lang.String[ ])'
LANGUAGE JAVA;
```

## 为用户定义的例程指定 INOUT 参数

用 SPL 、C 或 Java™ 语言编写的 UDR 也可以支持 INOUT 参数。当调用 UDR 时，每个 INOUT 参数的值作为参量通过引用传递给 UDR 。

当 UDR 执行完成时，它能够为 INOUT 参数返回一个修改后的值给调用上下文。INOUT 参数可以是 GBase 8s 支持的任意数据类型，包括用户定义的和复杂数据类型和以下例外：

- Serial 类型（BIGSERIAL 、SERIAL 和 SERIAL8）
- 简单大对象类型（BYTE 和 TEXT）。

在以下示例中，CREATE PROCEDURE 语句注册一个带有单个 INOUT 参数的 C 例程：

```
CREATE PROCEDURE CALC ( INOUT param1 float )
EXTERNAL NAME "$GBASEDBTDIR/etc/myudr.so(calc)"
LANGUAGE C;
```

SPL 例程可以调用具有 OUT 和 INOUT 参数的其它 UDR，如果这些 UDR 用 SPL 或 C 语言编写。然而，SPL 例程不能调用参量包含 OUT 或 INOUT 参数的 Java UDR。

支持从 SPL 例程调用具有命名或未命名 ROW 参数的 UDR ，对 ROW 参数类型和调用的 UDR 的编程语言有以下依赖性：

SPL 例程可以调用 ROW 参量是 IN 参数的 C UDR。但是不能调用 ROW 参量是 OUT 或 INOUT 参数的 C UDR 。

SPL 例程可以调用具有任意参数类型（包括 IN 、OUT 和 INOUT）的 ROW 参量的 SPL UDR。

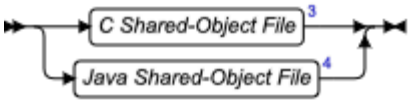
可以将 INOUT 参数分配给语句-局部变量（SLV），语句-局部变量在语句本地的变量表达式部分中描述。

## 5.14 共享对象文件名

在注册或更改外部例程时使用共享对象文件名指定一个可执行对象文件名的路径名。

**语法**

共享对象文件




**用法**

如果 IFX_EXTEND_ROLE 配置参数设置为 1 或 ON，只有 DBSA 已授予内置 EXTEND 角色的用户才有权使用此段。（无论是否启用 IFX_EXTEND_ROLE，必须对数据库保留 Resource 特权或 DBA 特权，然后才能创建、删除或更改外部 UDR。）

DB_LIBRARY_PATH 配置参数设置中包括安全策略授权 DataBlade 模块和 UDR 驻留的每个文件系统。除非缺失 DB_LIBRARY_PATH 或没有设置，否则数据库服务器无法访问此段指定的文件，除非其路径名以与 DB_LIBRARY_PATH 的值完全匹配的字符串开头。

例如，如果 "$GBASEDBTDIR/extend" 是 Linux™ 系统上的 DB_LIBRARY_PATH 值之一，则共享对象文件可以在 $GBASEDBTDIR/extend 文件系统或其子目录中具有路径名。此目录也是内置 DataBlade 模块所在的文件系统。

指定共享对象文件名的语法取决于该外部例程是按 C 语言还是 Java 语言编写。以下几节描述了这些外部语言。

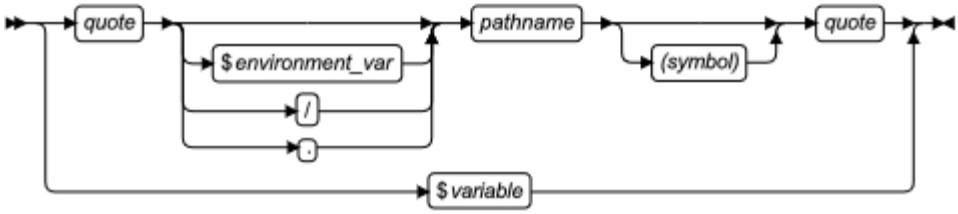
有关出现在 ALTER FUNCTION 、ALTER PROCEDURE 、ALTER ROUTINE、 CREATE FUNCTION 和 CREATE PROCEDUREF 语句的 EXTERNAL NAME 子句中上下文的更多信息，请参阅相关的引用，外部例程引用。

## C 共享对象文件

要指定 C 共享对象文件的位置，在引用路径名中指定动态载人可执行文件的路径或把它作为一个变量。

**语法：**

C 共享对象文件




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *environment_var* | 取决于平台的指示符 | 必须以美元符号（$）开头 | 标识符 |
| *pathname* | 文件的路径名 | 请参阅后面的注释 | 必须符合操作系统约定 |
| *quote* | 单引号（’）或双引号（’’） | 开始和结束引号必须匹配 | 文字符号（’ 或 ’’） |
| *symbol* | 文件的入口点 | 必须用括号括起来 | 必须符合操作系统约定 |
| *variable* | 取决于平台的指示符 | 必须以美元符号（$）开头 | 必须符合 C 语言约定 |

下列规则影响路径名和 C 语言中的文件名规范：

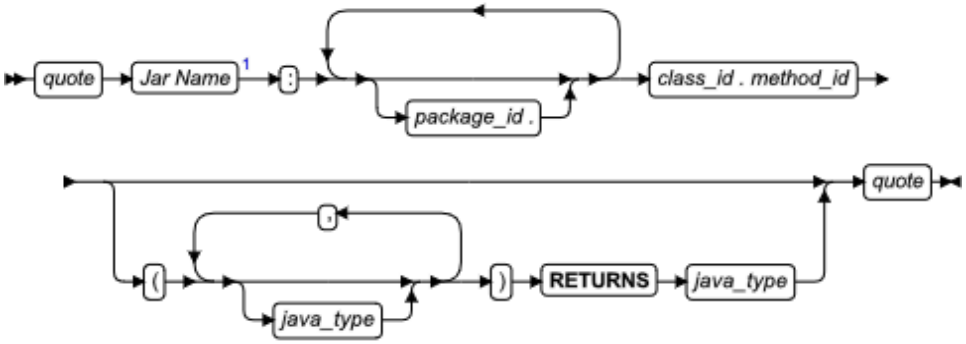
- 文件名（不带*路径名*）可以指定一个内部函数。
- 当 CREATE 或 ALTER 语句运行时，如果*路径名*和当前目录相关可以省略句号（.）。

- 在 UNIX™ 中，绝对路径必须以斜杠（/）符号开头，而且每一个目录名必须以斜杠（/）符号结尾。
- 在 Windows™ 中，绝对路径必须以反斜杠（\）符号开头，而且每一个目录名必须以反斜杠（ \ ）符号结尾。
- *路径名*结尾处的文件名必须有 **.so** 文件扩展名并且必须指向共享对象库中的一个可执行文件。
- 只在动态可载人的可执行对象文件入口点的名称和用 CREATE FUNCTION 或 CREATE PROCEDURE 注册的 UDR 名称不同时，使用 ***symbol***。
- 如果指定一个***变量***，它必须包含可执行文件的完整路径名。
- 在引用路径名中可以包含空白字符，如空格或制表符。

## Java 共享对象文件

要指定 Java™ 更新对象文件，需要指定 UDR 对应的静态 Java 方法和定义这个方法的 Java 二进制文件。

Java 共享对象文件：




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *class_id* | 方法实现 UDR 的 Java™ 类 | 类必须在 Jar 名标识的 **.jar** 文件中存在 | 必须遵守 Java 标识符的规则 |
| *java_type* | 在 Java™ 方法特征符中参数的 Java 数据类型 | 必须在 JDBC 类中定义或通过 SQL-to-Java 映射定义 | 必须遵守 Java 标识符的规则 |
| *method_id* | 实现 UDR 的 Java 方法名 | 必须在 *java_class_name* 指定的 Java 类中存在 | 必须遵守 Java 标识符的规则 |
| *package_id* | 包含 Java 类的数据包名 | 必须存在 | 必须遵守 Java 标识符的规则 |
| *quote* | 单引号（’）或双引号（’’）定界符 | 开始和结束引号必须匹配 | 从键盘输入的文字符号（ ’ 或 ’’） |

在创建用 Java 语言编写的 UDR 之前，必须用 sqlj.install_jar 过程分配一个 jar 标识符给外部 jar 文件。（有关更多信息，请参阅 sqlj.install_jar。）在共享对象文件名中可以包含实现 UDR 的方法的 Java 特征符。

- 如果不指定 Java 特征符，例程管理器从 CREATE FUNCTION 或 CREATE PROCEDURE 语句中的 SQL 特征符确定隐式 Java 特征符。

  它用 JDBC 和 SQL-to-Java 映射把 SQL 数据类型映射到对应的 Java 数据类型。有关吧用户定义的数据类型映射到 Java 数据类型的信息，请参阅 sqlj.setUDTextName。
- 如果指定了 Java 特征符，例程管理使用这个***显式*** Java 特征符作为要使用的 Java 方法名。

例如，如果 Java 方法 explosiveReaction( ) 实现 Java UDR sql_explosive_reaction( )（如 sqlj.install_jar 中所讨论），则它的共享对象文件名可以是：

course_jar:Chemistry.explosiveReaction

前面的共享对象文件名提供了隐式 Java 特征符。下面的共享对象文件名等价于一个显式 Java 特征符：

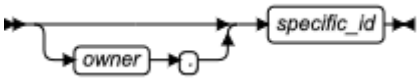
course_jar:Chemistry.explosiveReaction(int)

## 5.15 专用名

使用专用名为 UDR 声明一个在数据库或名称空间中唯一的标识符。当看到在语法图表中引用专用名时，使用专用名段。

**语法**

专用名




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *owner* | UDR 的所有者 | 不超过 32 个字节。必须和这个 UDR 的函数或过程名的***所有者***相同。另见对所有者名称的限制。 | 所有者名称 |
| *specific_id* | UDR 的唯一名称 | 不能超过 128 字节长。另见对专用名的限制。 | 标识符 |

**用法**

***专用名***是由 CREATE PROCEDURE 或 CREATE FUNCTION 语句声明的唯一标识符，作为 UDR 的一个替换名。

因为可以重载例程，所以数据库可以有多个具有相同名称和不同参数列表的 UDR。可以为 UDR 分配一个专用名，唯一标识指定的 UDR 。

如果在创建 UDR 时声明一个专用名，以后在对 UDR 进行更改、删除、授权或取消特权或更新统计信息时可以使用这个名称。否则，如果只有一个名称不能唯一标识 UDR 时，需要对 UDR 名包含参数数据类型。

### 对所有者名称的限制

当声明专用名时，*所有者*必须和限定所创建的 UDR 的函数或过程名的*授权标识符*相同。即，无论是否指定名称限定 UDR 名称或专用名或者同时限定两者。*所有者*名称必须匹配。

当在创建 UDR 的 DDL 语句中未指定所有者名称时， GBase 8s 使用创建 UDR 的用户的登录名。因此，如果您在一个地方指定所有者名称而在另一个地方没有指定，那么所指定的所有者名称必须和您的用户 ID 匹配。

### 对专用名的限制

在不符合 ANSI 的数据库中，***specific_id*** 在数据库的例程名中必须是唯一的。两个 UDR 不能具有相同的 ***specific_id*** ，即使它们有不同的所有者。

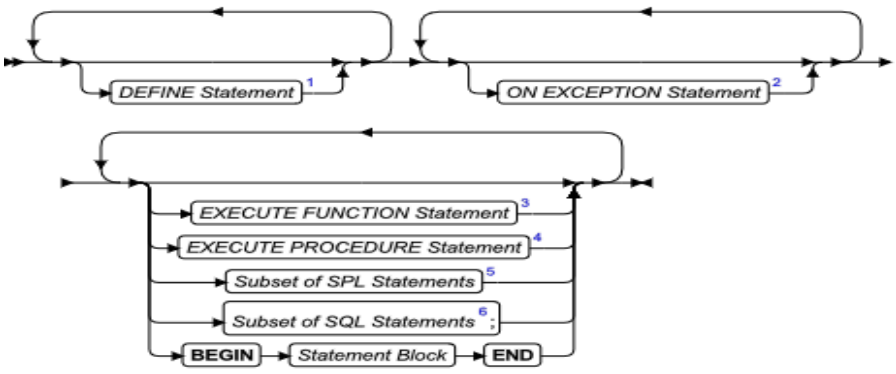
在符合 ANSI 的数据库中，***owner.specific_id*** 组合必须是唯一的。即，专用名在具有相同所有者的 UDR 中必须是唯一的。

## 5.16 语句块

使用语句块指定，当执行一个包含本段的 SPL 语句时，进行 SPL 和 SQL 操作。

**语法**

语句块




**用法**

SPL 和 SQL 语句可以出现在语句块中，语句块是能够定义变量或 SPL 的 ON EXCEPTION 语句的范围的零个或多个语句。如果语句块为空，当 SPL 例程中的执行控制传递给空的 SPL 语句块时不会发生任何操作。

### SPL 语句的子集在语句块中有效

语句块的语法引用了此节。您可以在语句块中使用下列任何 SPL 语句：

- <<***Label*** >>
- CALL
- CASE
- CONTINUE
- EXIT
- FOR
- FOREACH
- GOTO
- IF
- LET
- LOOP
- RAISE EXCEPTION
- RETURN
- SYSTEM
- TRACE
- WHILE

但是，GOTO 和 << *Label* >> 在 ON EXCEPTION 语句块中无效。

## SQL 语句在 SPL 语句块中有效

语句块的图表涉及这一节的内容。大多数 SQL 语句在 SPL 语句块中有效，除了以下列出的语句。下列 SQL 语句在 SPL 语句块中无效：

- CLOSE DATABASE
- CONNECT
- CREATE DATABASE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM DATABASE
- DISCONNECT
- EXECUTE
- FLUSH
- INFO
- LOAD
- OUTPUT
- PUT
- RENAME DATABASE
- SET AUTOFREE
- SET CONNECTION
- UNLOAD

UPDATE STATISTICS 例如，您不能在 SPL 例程中关闭数据库或连接新的数据库。同样，您不能在同一例程中删除当前 SPL 例程。然而，您可以删除另一个 SPL 例程。

只能在两种情况下使用 SELECT 语句：

- 可以使用 INTO TEMP 子句把 SELECT 语句的结果放进一个临时表。
- 可以使用 SELECT 语句的 SELECT ... INTO 形式把结果值放进 SPL 变量。

当在 SELECT ... INTO TEMP 或 SELECT ... INTO ***variable*** 语句中包含 ORDER BY 子句时，这表示此查询返回多行。如果您指定没有 FOREACH 循环的 ORDER BY 子句来在 SPL 例程中单独处理返回的行，那么数据库服务器会发出错误。

如果后面 SPL 例程要作为数据操纵语言（DML）语句的一部分被调用，还有附加的限制。有关更多信息，请参阅在数据操纵语句中 SPL 例程的限制。

### 嵌套语句块

可以使用 BEGIN 和 END 关键字来定界嵌套在另一个语句块中的语句块。

## 引用 SPL 变量和异常处理程序的作用域

BEGIN 和 END 关键字可以限制 SPL 变量和异常处理程序的作用域。在 BEGIN 和 END 语句块中的变量声明和异常处理程序定义是语句块局部的，在语句块外面不可见。下面的代码使用了 BEGIN 和 END 语句块来限定变量引用的作用域：

```
CREATE DATABASE demo;
        CREATE TABLE tracker (
        who_submitted CHAR(80), -- Show what code was running.
        value INT,                      -- Show value of the variable.
        sequential_order SERIAL -- Show order of statement execution.
        );
        CREATE PROCEDURE demo_local_var()
        DEFINE var1, var2 INT;
        LET var1 = 1;
        LET var2 = 2;
        INSERT INTO tracker (who_submitted, value)
        VALUES ('var1 param before sub-block', var1);
        BEGIN
        DEFINE var1 INT;     -- same name as global parameter.
        LET var1 = var2;
        INSERT INTO tracker (who_submitted, value)
        VALUES ('var1 var defined inside the "IF/BEGIN".', var1);
        END
        INSERT INTO tracker (who_submitted, value)
        VALUES ('var1 param after sub-block (unchanged!)', var1);
        END PROCEDURE;
        EXECUTE PROCEDURE demo_local_var();
        SELECT sequential_order, who_submitted, value FROM tracker
        ORDER BY sequential_order;
```

这个示例声明了三个变量，其中两个名为 **var1**。（这里创建的名称冲突是为了说明哪种变量是可见的。通常建议对不同变量不要使用相同的名称。因为冲突的变量名会造成代码可读性差并且难维护。）

因为语句块的关系，每次只有一个 **var1** 变量在作用域中。

在语句块中声明的 **var1** 变量是唯一可以在语句块中引用的 **var1** 变量。

在语句块外面声明的 **var1** 变量在语句块中是不可见的。因为它在作用域之外，所以发生在语句块内部的 **var1** 变量值的更改对它没有影响。所有语句运行以后，外面的 **var1** 的值仍然是 1。

**var2** 变量在语句块中可见，因为它没有因名称冲突而被块专用变量替代。

## 在数据操纵语句中 SPL 例程的限制

如果在非数据操纵语言（DML）语句（即 EXECUTE FUNCTION 或 EXECUTE PROCEDURE）的 SQL 语句中调用 SPL 例程，那么 SPL 例程可以执行未列在 SQL 语句在 SPL 语句块中有效一节中的任何语句。

如果 SPL 例程作为 DML 语句（即 INSERT 、UPDATE 、DELETE 、MERGE 或 SELECT 语句）的一部分调用，那么例程不能执行下面列表中的任何 SQL 语句：

- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER INDEX
- ALTER SEQUENCE
- ALTER TABLE
- BEGIN WORK
- COMMIT WORK
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE DISTINCT TYPE
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE ROLE
- CREATE ROW TYPE
- CREATE SEQUENCE
- CREATE TRIGGER
- DELETE
- DROP ACCESS_METHOD
- DROP AGGREGATE
- DROP INDEX
- DROP OPCLASS
- DROP ROLE
- DROP ROW TYPE
- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER
- DROP TYPE

- DROP VIEW
- INSERT
- MERGE
- RENAME COLUMN
- RENAME DATABASE
- RENAME SEQUENCE
- RENAME TABLE
- ROLLBACK WORK
- SET CONSTRAINTS
- TRUNCATE
- UPDATE

如果 SPL 例程的调用上下文是试图执行以上列出的任何 SQL 语句的 DML 语句，则 GBase 8s 发出错误 -675。

这些限制不适用于触发器调用的 SPL 例程，因为在这些情况下 SPL 例程不是由 DML 语句调用，因而可以包含任意未列在 SQL 语句在 SPL 语句块中有效中的 SQL 语句，例如 UPDATE 、INSERT 和 DELETE。

### SPL 例程中的事务

在不符合 ANSI 的数据库中，可以在 SOL 语句中使用 BEGIN WORK 和 COMMIT WORK 语句启动事务、结束事务、或者在同一 SPL 例程中启动和终止的事务。如果在远程执行的例程中启动一个事务，必须在例程退出之前结束这个事务。

然而，就像前面提到的，ROLLBACK WORK 语句在 SPL 语句块中是无效的。

### 对用户身份和角色的支持

可以在 SPL 例程中使用角色。可以在 SPL 例程中执行角色相关语句（CREATE ROLE 、DROP ROLE 、GRANT 、REVOKE 和 SET ROLE）并且持有 SETSESSIONAUTH 特权的用户可以发出 SESSION AUTHORIZATION 语句。在 SPL 例程中，您可以使用 GRANT 语句

- 授予角色自由访问权限，
- 或向角色授予基于标签的访问凭证（LBAC），
- 或向角色授予其它角色。

SPL 例程还可以使用 REVOKE 语句撤销角色的访问特权、LBAC 凭证或角色。

在授予访问特权、角色和 LBAC 凭证的 SPL 例程完成执行后，用户通过启用角色或通过 SET SESSION AUTHORIZATION 语句在 SPL 例程中获取的访问权限，角色和 LBAC 凭证不会自动退出。所授予的内容持续存在，直到后续的 REVOKE 操作取消 GRANT 操作的效果。

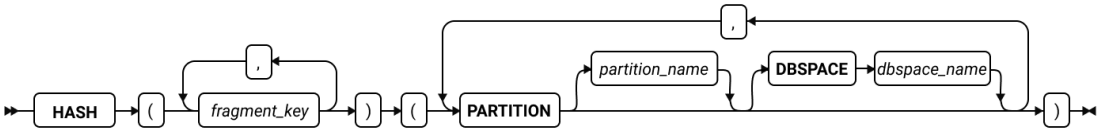
有关角色的更多信息，请参阅第二章中的 CREATE ROLE 语句 、DROP ROLE 语句 、GRANT 语句 、REVOKE 语句 和 SET ROLE 语句。

## 5.17 HASH 分区

创建分区已有 4 种分区方式，包括：轮转法（ROUND ROBIN）、表达式（EXPRESSION）、范围（RANGE）及列表（LIST），在此基础上新增 HASH 分区方式。

HASH 分区表是按分区列的 HASH 计算结果来决定其分区的，而特定的分区列其 HASH 值是固定的，也就是说 HASH 分区表的数据是按分区列值来聚集的，同样的分区列肯定在同一分区。例如在证券行业，我们经常查询某一只股票的 K 线，建成 HASH 分区表，则数据按 HASH 分区列聚集，就更适合 K 线数据的查询，因为同样 id 的记录必定在同一分区，同时，同样 id 值的记录落在同一数据块的几率也增大了，从而一定程度上减少 IO，提高查询效率。

**创建 HASH 分区**




其中：

| 元素 | 描述 | 限制 |
|---|---|---|
| ***fragment_key*** | 分区列名称 | 必选项；<br>必须是表中的列 |
| *partition_name* | 分区名称 | 可选项； |
| *dbspace_name* | dbspace 名称 | 不指定 dbspase 则为当前数据库默认的表空间；指定的 dbspase 需存在，可重复 |

例如，人员表 user 中，按照部门字段 departmentID 作为分区列进行 HASH 分区，如下：

```
CREATE TABLE USER(
name varchar(20),
departmentID int,
age int,
address varchar(50)
)PARTITION BY HASH(departmentID)
(
partition p1 DBSPACE dbs1,
partition p2 DBSPACE dbs2,
partition p3 DBSPACE dbs3
);
```

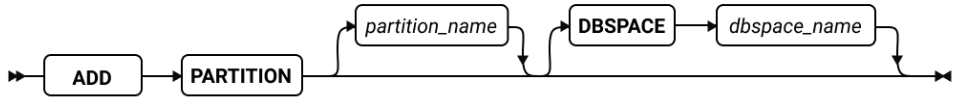
**管理 HASH 分区**

HASH 分区的管理支持增加分区（add 子句）、截断分区（truncate 子句）和结合分区（coalesce 子句）。

增加分区

增加分区，数据会重新在增加分区后的所有分区中根据哈希运算结果重新分布。

在 alter fragment on table…主干语法基础上：




其中 partition_name 是分区名称，dbspace_name 是 dbspace 名称，均可省略。

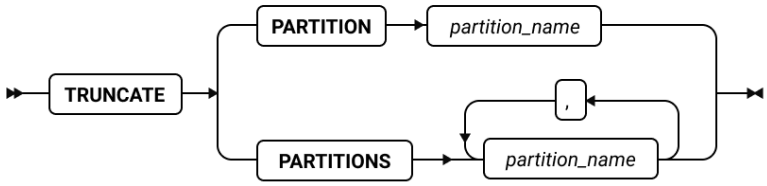
例如，人员表 user 增加一个分区 p4：

```
ALTER FRAGMENT ON TABLE USER ADD PARTITION p4 DBSPACE dbs4;
```

截断分区

主要功能为清空哈希分区表中的数据，但哈希分区表本身并不删除。

在 alter fragment on table…主干语法基础上：




其中 partition_name 是分区名称，不可省略。

例如，人员表 user 清空掉 p1 分区数据

```
ALTER FRAGMENT ON TABLE USER TRUNCATE PARTITION p1;
```

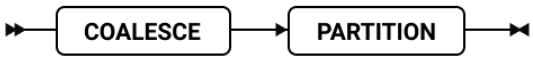
人员表 user 清空掉 p2、p3 分区数据

```
ALTER FRAGMENT ON TABLE USER TRUNCATE PARTITIONS p2,p3;
```

结合分区

结合分区是针对哈希分区或者包含哈希分区的复合分区的，目的是减少分区数。被结合的分区是由数据库自动选择的，结合完成后该分区会被删除，数据会被合并至其它某个数据库指定分区中，分区数大于等于 2 个才支持结合操作。

在 alter fragment on table…主干语法基础上：




例如，人员表 user 执行 1 次结合分区，从而自动减少 1 个分区：

```
ALTER FRAGMENT ON TABLE USER COALESCE PARTITION;
```

**HASH 分区使用**

HASH 分区插入数据

HASH 分区表的数据插入跟其他分区表（如列表或范围分区表）的插入方式相同，数据库根据 HASH 分区规则自动计算分区。

例如，向 user 表中插入数据：

INSERT INTO USER VALUES(‘张三’,2,25,’北京市海淀区’);

HASH 分区更新数据

HASH 分区表的更新数据时跟普通表的更新方式相同，更新分区列数据时数据重新按照 HASH 规则排列。

例如，user 表中更新某类数据：

UPDATE USER SET departmentID=1 WHERE AGE=25;

HASH 分区删除数据

哈希分区表的数据删除跟其他分区表（如列表或范围分区表）的删除方式相同。

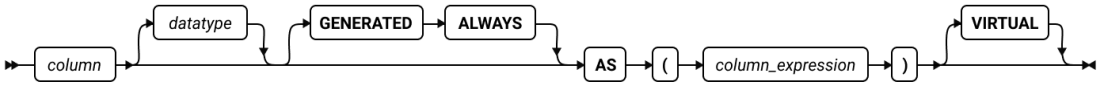
例如，user 表中删除某类数据：

DELETE FROM USER WHERE departmentID=1;

## 5.18 虚拟列

虚拟列是指使用表达式或函数进行定义的数据列。逻辑上，表的虚拟列与普通列具有相同的语法含义，但虚拟列的值并不保存在任何物理存储介质上，而是在 SQL 的执行过程中，根据定义虚拟列的表达式或函数进行计算而获得。

虚拟列语法图：




| 元素 | 描述 | 限制 |
|---|---|---|
| ***column*** | 虚拟列的列名 | 不可与该表其它列重名，不可省略 |
| ***datatype*** | 虚拟列数据类型 | |
| ***GENERTATED ALWAYS*** | 显式声明虚拟列关键字 | 可省略 |
| ***AS*** | 显式声明虚拟列关键字 | 不可省略 |
| ***column_expression*** | 定义虚拟列的列表达式或常量表达式 | 引用当前表中的列 |
| ***VIRTUAL*** | 显式声明虚拟列关键字 | 可省略 |

**创建一个带虚拟列的表**

例如，创建带虚拟列的表 employee，其中 total_sal 为表达式定义的虚拟列：

```
CREATE TABLE employee
(
        empl_ID         INT,
        empl_Nm         VARCHAR(50),
        monthly_Sal      DECIMAL(10,2),
        bonus           DECIMAL(10,2),
        total_Sal        DECIMAL(10,2) GENERATED ALWAYS AS (monthly_Sal*12 + bonus)
);
```

**向带虚拟列的表中插入数据**

对虚拟列执行 INSERT 操作，需明确写出插入表的各字段名称。

例如，向带虚拟列的表中正确插入数据：

```
INSERT INTO employee(empl_ID,empl_Nm,monthly_Sal,bonus)
VALUES(1,'zhangSan',9000,560);
```

返回结果：成功

只写表名不明确写出各字段名称：

```
INSERT INTO employee VALUES(1,'zhangSan',9000,560);
```

返回结果：报错误代码-981

插入字段中包含虚拟列：

```
INSERT INTO employee(empl_ID,empl_Nm,monthly_Sal,bonus,total_Sal)
VALUES(1,'zhangSan',9000,560)
```

返回结果：报错误代码-981

**支持在 UPDATE/DELETE 的 WHERE 语句中使用虚拟列，不允许在虚拟列上执行 UPDATE 语句**

例如，更新虚拟列，会报错误：

```
UPDATE employee SET total_sal = 2000;
```

返回结果：报错误代码-982

```
UPDATE employee SET monthly_Sal = 2000 where total_Sal>100000;
```

返回结果：成功

**修改虚拟列**

支持使用 ALTER TABLE 语句新增虚拟列，不支持 NOT NULL 约束。

例如，向 employee 表中新增一列虚拟列 monthly_Avg：

```
ALTER TABLE employee ADD (monthly_Avg AS ((monthly_Sal * 12 + bonus)/12));
```

支持使用 RENAME COLUMN 子句修改虚拟列名称，要求不可与当前表中已有列名重复。

例如，修改虚拟列 total_Sal 列名为 total：

```
RENAME COLUMN employee.total_Sal TO total;
```

支持使用通过 ALTER MODIFY 语句修改虚拟列的数据类型，包括显示修改和隐式修改，显示修改虚拟列数据类型需要确保 AS 子句的返回值数据类型与虚拟列数据类型一致或支持隐式类型转换，隐式修改虚拟列数据类型可省略数据类型仅通过调整 AS 子句实现，当且仅当只修改数据类型长度时——如 varchar(15)改为 varchar(20)，float 改为 int——可以省略 AS 语句，其他情况不允许省略 AS 子句。

例如，显示修改虚拟列类型：

```
ALTER TABLE employee MODIFY( monthly_Avg FLOAT AS ((monthly_Sal*12 + bonus)/12));
```

隐式修改虚拟列类型：

```
ALTER TABLE employee MODIFY( monthly_Avg AS ((monthly_Sal*12 + bonus)/12));
```

省略 AS 子句修改虚拟列类型：

```
ALTER TABLE employee MODIFY( monthly_Avg INT);
```

支持使用 ALTER MODIFY 语句修改虚拟列定义表达式，此时虚拟列的数据类型与表达式一致。

例如，修改 employee 表中虚拟列 monthly_Avg 的表达式：

```
ALTER TABLE employee MODIFY( monthly_Avg AS ((monthly_Sal*12 + bonus*0.9)/12));
```

**删除虚拟列和普通列**

删除某列时，需要先删除引用该列的所有虚拟列，再删除该列。

例如，emplyee 表中删除虚拟列 mothly_Avg 和普通列 monthly_Sal：

```
ALTER table employee DROP monthly_Sal;
```

返回结果：报错

```
ALTER table employee DROP (monthly_Avg, monthly_Sal);
```

返回结果：报错

```
ALTER table employee DROP monthly_Avg;
ALTER table employee DROP monthly_Sal;
```

返回结果：成功

**虚拟列上建索引**

支持在虚拟列上创建索引。

例如，在 emplyee 表的虚拟列 total_Sal 上创建索引：

```
CREATE INDEX idx_total_sal ON employee(total_Sal);
```

**虚拟列安全策略**

虚拟列不继承定义该虚拟列所涉及普通列上的安全策略。

**注意：**

- CREATE TABLE AS SELECT…FROM...时不支持 SELECT 中含虚拟列；
- WITH AS 子句中不支持使用虚拟列；

- 临时表中不支持使用虚拟列；
- 虚拟列表达式中不支持加密解密函数；
- 虚拟列仅支持 CREATE TABLE 建表语句或 ALTER TABLE ADD 语句添加 CHECK 约束。

# 6. 内置例程

该主题描述创建数据库时被数据库服务器所知的标识符的例程，但是其用法与第 4 章描述的内置 SQL 函数有所不同。

内置例程可以根据它们执行的任务分类：

- 会话配置过程
  - **SYSDbClose( )**
  - **SYSDbOpen( )**
- DataBlade 模块管理函数
  - **SYSBldPrepare( )**
  - **SYSBldRelease( )**
- Visual Explain 输出生成函数
  - **Explain_SQL( )**
- UDR 定义例程
  - **ifx_Replace_Module( )**
  - **ifx_Unload_Module( )**
  - **jvpControl( )**
  - **sqlj.Alter_Java_Path( )**
  - **sqlj.Install_jar( )**
  - **sqlj.Remove_jar( )**
  - **sqlj.Replace_jar( )**
  - **sqlj.SetUDTExtName( )**
  - **sqlj.UnsetUDTExtName( )**
  - **sysgbase.Metadata( )**
  - **sysgbase.sqlcaMessage( )**

如果数据库服务器配置为支持 DRDA 协议，则会自动创建 Metadata 和 sqlcaMessage 例程。SYSDbOpen 和 SYSDbClose 例程可以在任一 GBase 8s 数据库中定义。

随后的章节描述了这些类别和每个类别中的例程。

## 6.1 间隔函数

使用间隔函数从包含表示 INTERVAL 限定符中的时间单位和分隔符的数字和字符串的参数中返回 INTERVAL 值。这些函数在定义或修改索引和表的范围间隔分布式存储策略的 CREATE TABLE 、CREATE INDEX 和 ALTER FRAGMENT 语句中非常有用。

间隔函数将数字或字符串转换为 INTERVAL DAY TO SECOND 或 INTERVAL YEAR TO MONTH 字面值，或第二个参数中指定的时间单位的有效精度。然而，这些函数不支持 FRACTION 或 .FRACTION 作为它们的第二个参数中的最后一个时间单位。

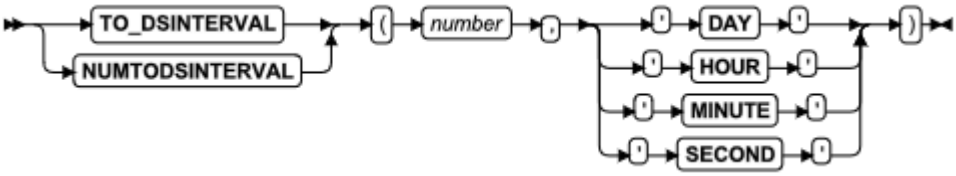
## TO_DSINTERVAL 函数

TO_DSINTERVAL 函数将表示时间单位的字符串转换为 INTERVAL DAY TO SECOND 字符值。此函数还可以接受数字和字符串作为其参数，并以单个时间单位精度 DAY 、HOUR 、MINUTE 或 SECOND 返回 INTERVAL 值。

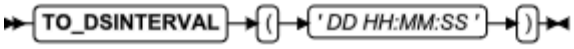
当您定义范围间隔存储分配策略以分片表或索引时，您可以使用单个参数（或两个参数，其同义词 NUMTODSINTERVAL）的 TO_DSINTERVAL 函数来指定间隔范围值。

### 语法

数字转换为 INTERVAL




字符串转换为 INTERVAL




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *DD* | 一位或两位数指定间隔中的天数 | 必须是下列其中之一的数据类型：<br>• CHAR<br>• NCHAR<br>• VARCHAR<br>• NVARCHAR<br>• LVARCHAR | 字符串 |
| *HH*:*MM*:*SS* | 三组两位数字，用冒号（:）符号分隔，指定间隔中的小时数、分钟数和秒数 | 必须是下列其中之一的数据类型：<br>• CHAR<br>• NCHAR<br>• VARCHAR<br>• NVARCHAR<br>• LVARCHAR | 字符串 |
| *number* | 指定间隔中的天数、小时数、分钟数或秒数的数字。<br>可以是表达式，包括列表达式，该表达式解析（或转换）到其中一个有效数字数据类型。 | 必须是下列其中之一的数据类型：<br>• INT<br>• BIGINT<br>• SMALLINT<br>• INT8<br>• DECIMAL<br>• REAL | 数值 |

| | | | |
|---|---|---|---|
| | | • FLOAT<br>• SERIAL<br>• SERIAL8<br>• BIGSERIAL | |

**用法**

当您使用结果分片表或索引时，可以使用 TO_DSINTERVAL 函数指定间隔值。TO_DSINTERVAL 函数在允许内置例程的上下文中有效。NUMTODSINTERVAL 函数与 TO_DSINTERVAL 函数相同都是用来转换数字值。

以下示例显示了 TO_DSINTERVAL 函数是如何解释不同的值：

以下示例指定一天的间隔：

```
TO_DSINTERVAL('1 00:00:00')
TO_DSINTERVAL(1,'DAY')
NUMTODSINTERVAL(1,'DAY')
```

以下示例指定一个小时的间隔：

```
TO_DSINTERVAL('0 01:00:00')
TO_DSINTERVAL(1,'HOUR')
NUMTODSINTERVAL(1,'HOUR')
```

以下示例指定一分 30 秒的间隔：

```
TO_DSINTERVAL('0 00:01:30')
TO_DSINTERVAL(1.5,'MINUTE')
NUMTODSINTERVAL(1.5,'MINUTE')
```

以下示例显示了如何使用表达式作为数字值：

```
TO_DSINTERVAL(10+10+100,'DAY')
```

## TO_YMINTERVAL 函数

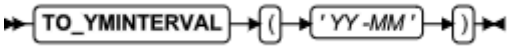
TO_YMINTERVAL 函数将表示时间单位的字符串转换为 INTERVAL YEAR TO MONTH 字符值。此函数还可以接受数字和字符串作为其参数，并返回具有单位时间单位精度 YEAR 或 MONTH 的 INTERVAL 值。

当您定义范围间隔存储分配策略以分片表或索引时，您可以使用单个参数（或两个参数，其同义词 NUMTODSINTERVAL）的 TO_DSINTERVAL 函数来指定间隔范围值。

**语法**

数字转换为 INTERVAL


字符串转换为 INTERVAL




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *number* | 间隔中的年或月的数字。<br>可以是表达式，该表达式解析（或强制转型）到其中一个有效数字数据类型。 | 必须是下列其中之一的数据类型：<br>• INT<br>• BIGINT<br>• SMALLINT<br>• INT8<br>• DECIMAL<br>• REAL<br>• FLOAT<br>• SERIAL<br>• SERIAL8<br>• BIGSERIAL | 数字 |
| *MM* | 指定间隔中月数的两个数字。连字符（-）必须在第一个数字之前。 | 必须是下列其中之一的数据类型：<br>• CHAR<br>• NCHAR<br>• VARCHAR<br>• NVARCHAR<br>• LVARCHAR | 字符串 |
| *YY* | 指定间隔中年数的两个数字 | 必须是下列其中之一的数据类型：<br>• CHAR<br>• NCHAR<br>• VARCHAR<br>• NVARCHAR<br>• LVARCHAR | 字符串 |

**用法**

当按间隔分片表或索引时，可以使用 TO_YMINTERVAL 函数指定间隔值。TO_YMINTERVAL 函数在允许内置例程的上下文中有效。NUMTOYMINTERVAL 函数与 TO_YMINTERVAL 函数相同都是用来转换数字值。

**示例**

以下示例显示了 TO_YMINTERVAL 是如何解释不同的值的。

以下示例指定一年的间隔：

```
TO_YMINTERVAL('01-00')
TO_YMINTERVAL(1,'YEAR')
NUMTOYMINTERVAL(1,'YEAR')
```

以下示例指定一个月的间隔：

```
TO_YMINTERVAL('00-01')
TO_YMINTERVAL(1,'MONTH')
NUMTOYMINTERVAL(1,'MONTH')
```

以下示例指定一年零六个月的间隔：

```
TO_YMINTERVAL('01-06')
TO_YMINTERVAL(1.5,'YEAR')
NUMTOYMINTERVAL(1.5,'YEAR')
```

以下示例显示如何使用表达式作为数字值：

```
TO_YMINTERVAL(10+10+100,'YEAR')
```

以下示例定义了具有范围间隔分片模式的表 **t2** 。这里的 DATETIME 列 **dt1** 是分片键，NUMTOYMINTERVAL 的返回值将间隔大小定义为 25 年。具有年份晚于 2005 年但早于 2031 的 **dt1** 值的行将存储在范围分片 **p1** 中：

```
CREATE TABLE t2 (c1 int, d1 date, dt1 DATETIME YEAR TO FRACTION)
FRAGMENT BY RANGE (dt1) INTERVAL (NUMTOYMINTERVAL (25,'YEAR'))
PARTITION p1 VALUES <
DATETIME(2006-01-01 00:00:00.00000) YEAR TO FRACTION(5) IN dbs1;
```

如果插入一行，其中 **dt1** 中的 YEAR 值小于 2006 或大于 2030，那么数据库服务器将会自动创建一个新的间隔分片，其范围大小是 25 年。有关范围间隔分片的语法和语义的更多信息，请参阅 Interval fragment 子句。

## 6.2 会话配置过程

这些内置 SPL 过程使数据库管理者在用户连接数据库或从数据库断开连接时，自动执行 SQL 和 SPL 语句。

在本手册中这些例程称为内置过程，因为数据库服务器识别它们的名称，并且将它们与处理其它例程的方式区别对待，但数据库服务器不会自动创建这些例程。要使用其特性，DBA 必须发出 CREATE PROCEDURE 语句或 CREATE PROCEDURE FROM 语句来定义这些例程的操作并将其注册到数据库中。只有 DBA 或用户 **gbasedbt** 可以创建、更改或删除这些例程。

如果 DBA 指定用户的登录 ID 作为其中一个过程的***所有者***，那么当指定的用户连接数据库或从数据库断开连接时数据库服务器会指定执行它。如果 DBA 指定 PUBLIC 作为所有者，则当不是任何这些内置会话配置过程的所有者的用户连接到数据库或从数据库断开连接时，将自动执行该例程。同一数据库服务器实例的不同数据库可以为单个用户或 PUBLIC 指定相同或不同的会话配置过程。这些内置过程在设置会话环境或激活代码无法轻易修改的应用程序的用户角色时非常有用。

这些是内置会话配置过程：

- **sysdbclose**
- **sysdbopen**

### 使用 SYSDBOPEN 和 SYSDBCLOSE 过程

要为一个或多个会话设置初始环境，创建并安装 **sysdbopen()** SPL 过程。该过程的典型影响是初始化会话的属性，而不需要在会话中显式定义属性。

如果用户通过不能修改应用程序代码或设置环境选项或环境变量的客户端应用程序访问数据库，则为一个或多个会话设置初始环境非常有用。

只要用户成功发出 DATABASE 或 CONNECT 语句以显式连接到安装了过程的数据库，就会执行 **sysdbopen** 过程。（但是当连接到本地数据库的用户调用远程 UDR 或执行通过使用 ***database*:*object*** or ***database*@*server*:*object*** 符号引用远程数据库对象的分布式 DML 操作是，不会在远程数据库中调用 **sysdbopen** 过程。）

这些过程时一般规则的例外情况，当在与 ANSI 不兼容的数据库中调用例程时， GBase 8s 会忽略 UDR 所有者的名称。对于除 **sysdbopen** 和 **sysdbclose** 之外的 UDR ，具有相同的 SQL 标识符但不同所有者名称的 UDR 的多个版本不能在同一数据库中注册，除非创建数据库的 CREATE DATABASE 语句也包括 WITH LOG MODE ANSI 关键字。

还可以创建 **sysdbclose** SPL 过程，当用户发出 LOSE DATABASE 或 DISCONNECT 语句从数据库断开连接时执行此过程，如果 PUBLIC.**sysdbclose** 过程已在数据库中注册，并且没有为当前用户注册 ***user*.sysdbclose** 过程，则当该用户与数据库断开连接时，将自动执行 PUBLIC.**sysdbclose** 过程。

您可以打开或关闭数据库时包含适当的有效 SQL 或 SPL 语言语句。对 SPL 过程中有效的 SQL 语句的一般限制也适用于这些例程。有关 SPL 例程中的 SQL 和 SPL 语句的限制，请参阅以下各节：

- SPL 语句的子集在语句 r 块中有效。
- SQL 语句在 SPL 语句块中有效。
- 在数据操纵语句中 SPL 例程的限制。

**重要：** sysdbopen 和 sysdbclose 过程是存储过程的作用域规则的例外。在一般 UDR 过程中，变量和语句的范围是局部的。当这些 SPL 过程退出时，SET PDQPRIORITY 和 SET ENVIRONMENT 语句设置不会保留。但是，在 sysdbopen 和 sysdbclose 过程中，设置会话环境的语句将保持有效，直到另一个语句重置选项，或会话结束。

例如，以下过程将事务隔离级别设置为 Repeatable Read，并设置 OPTCOMPIND 环境变量以指示查询优化器优先选择嵌套循环连接。当没有 *user*.sysdbopen 过程的用户连接到数据库时，将执行此例程：

```
CREATE PROCEDURE public.sysdbopen()
SET ISOLATION TO REPEATABLE READ;
SET ENVIRONMENT OPTCOMPIND '1';
END PROCEDURE;
```

过程不接受参数和返回值。**sysdbopen** 和 **sysdbclose** 过程必须在您希望执行的数据库中注册。DBA 可以创建以下四个类别的 **sysdbopen** 和 **sysdbclose** 过程。

| 过程名 | 描述 |
|---|---|
| *user.sysdbopen* | 当指定的 ***user*** 将数据库作为当前数据库打开时执行此过程。 |
| **public.sysdbopen** | 如果没有应用 ***user*.sysdbopen** 过程，那么当将数据库作为当前数据库打开时执行此过程。要避免重复的 SPL 代码，您可以供用户指定的过程调用此过程。 |

***user*.sysdbclose** 当指定的 ***user*** 关闭数据库时执行此过程，从数据库断开连接，或者解释用户会话。但是，如果当会话打开数据库时 ***user*.sysdbclose** 不存在，那么当会话关闭数据库时不会执行此过程。

**public.sysdbclose** 如果没有应用 ***user*.sysdbclose** 过程，则当用户关闭数据库服务器或从数据库服务器断开连接或结束会话时执行此过程。但是，如果当会话打开数据库时，**public.sysdbopen** 不存在，则当会话关闭数据库时不会执行此过程。

如果 CLOSE DATABASE 或 DISCONNECT 语句显式地终止连接，则数据库服务器调用 ***user*.sysdbclose** 过程（如果它在数据库中存在）或 **public.sysdbclose** （如果它存在并且不被用户使用）。如果应用程序在不发出 CLOSE DATABASE 或 DISCONNECT 语句的情况下终止，则数据库服务器强制执行数据库的隐式关闭，并执行 **sysdbclose** 过程（如果有该名称的 UDR 由用户或 PUBLIC 拥有）。

请确保正确设置文件访问权限，以允许预期用户执行 SPL 过程语句。例如，如果 SPL 过程执行将输出写入本地目录的命令。则必须设置权限以允许用户写入此目录。如果希望在许可失败时继续该过程，请为此条件包含 ON EXCEPTION 错误处理程序。

有关可以出现在 SPL 例程中的 SQL 语句以及有关事务和角色的 SPL 支持的更多信息，请参阅语句块一节。

**警告：** 如果 sysdbclose 过程失败，该失败会被忽视。然而，如果 sysdbopen 过程失败，数据库不会被打开。

为了避免无法打开数据库的情况，请在编写和调试 sysdbopen 过程时采取以下预防措施：

在连接到数据库之前设置 **IFX_NODBPROC** 环境变量。当设置 **IFX_NODBPROC** 时，不会执行该过程，并且故障不能阻止数据库打开。

来自这些过程的故障可以有系统生成或由 SPL 的 RAISE EXCEPTION 语句在过程中进行模拟。如果在连接时为用户调用的 sysdbopen 例程包含此语句，则该用户无法连接到数据库。有关更多信息，请参阅 RAISE EXCEPTION 的描述。

出于安全原因，非 DBA 无法阻止这些过程的执行。然而，对于一些应用程序，例如 ***ad hoc*** 查询应用程序，用户可以执行随后该表环境的命令和 SQL 语句。

**sysdbopen** 过程中定义的缺省角色优先于用户建立与数据库的连接时用户持有的任何其它角色，其中 **sysdbopen** 成功地为该用户指定了缺省角色。

由 ***user*.sysdbopen** 或 ***user*.sysdbclose** 过程中的 DDL 语句创建的任何数据库对象都由连接的用户所有，并且在 PUBLIC**.sysdbopen** 或 PUBLIC**.sysdbclose** 内创建的任何对象都由 PUBLIC 用户标识拥有，除非对象名称在 DDL 语句中声明时，被某个其它所有者名称完全限定。

对于兼容 ANSI 的数据库，在 CREATE PROCEDURE 语句的 **sysdbopen** 或 **sysdbclose** 定义的末尾需要显式的 COMMIT WORK 语句，以防止 **sysdbopen** 或 **sysdbclose** 过程执行的 SQL 语句的任何隐式事务在终止过程时回滚。（省略 COMMIT WORK 语句不会导致连接失败，但会在打开和回滚事务时浪费资源。）

有关这些过程中无效的 SQL 语句的列表，请参阅 SQL 语句在 SPL 语句块中有效。有关在这些过程中有效的 SPL 语句，请参阅 SPL 语句的子集在语句块中有效。

有关如何编写和安装 SPL 过程的一般信息，请参阅 *GBase 8s SQL 教程指南*中 SPL 例程一节。

## 在连接和访问时配置会话属性

可以使用 sysdbopen( ) 过程在连接或访问时更改数据库服务器会话的属性。如果无法修改应用程序的源代码以设置环境选项或会话变量，或者包括会话相关的 SQL 语句（例如，因为 SQL 语句包含供应商获取的代码），这将非常有用。

要更改会话的属性，请为各种数据库设计自定义 sysdbopen( ) 和 sysdbclose( ) 过程，以支持特定用户或 PUBLIC 组的应用程序。sysdbopen( ) 和 sysdbclose( ) 过程可以包含数据库服务器为用户或 PUBLIC 组在数据库打开或关闭时执行的 SET 、SET ENVIRONMENT 、SQL 或 SPL 语句的序列。

例如，对于 ***user1***，您可以定义包含在 ***user1*** 使用 DATABASE 或 CONNECT TO 语句打开数据库时执行的 SET PDQPRIORITY 、SET ISOLATION LEVEL 、SET LOCK MODE 、SET ROLE 或 SET EXPLAIN ON 语句的过程。

由于 sysdbopen( ) 过程中的 SET ENVIRONMENT 语句指定的会话环境变量 PDQPRIORITY 和 OPTCOMPIND 的任何设置将在会话持续时间内保持不变。SET PDQPRIORITY 和 SET ENVIRONMENT OPTCOMPIND 语句（对于常规过程不持久）在 sysdbopen( ) 过程中包含它们。

当作为过程所有者的用户从数据库断开连接时（或者 PUBLIC.sysdbclose( ) 运行时，如果它存在且当前用户不拥有 sysdbclose( ) 过程，运行 *user*.sysdbclose( ) 过程。

在自定义 sysdbopen( ) 和 sysdbclose( ) 过程中，当在与 ANSI 不兼容的数据库中调用例程时，GBase 8s 不会忽略 UDR 所有者的名称。

## 配置会话属性

只有 DBA 或用户 **gbasedbt** 可以在 SQL 的 ALTER PROCEDURE 、ALTER ROUTINE 、CREATE PROCEDURE 、CREATE PROCEDURE FROM 、CREATE ROUTINE FROM 、DROP PROCEDURE 或 DROP ROUTINE 语句中创建或更改 sysdbopen( ) 或 sysdbclose( )。

您可以设置 sysdbopen( ) 过程，在连接或访问时更改会话的属性，而不更改会话运行的应用程序。如果无法修改应用程序的源代码以设置环境选项或环境变量或包括会话相关的 SQL 语句，例如，因为 SQL 语句包含供应商获取的代码，这将非常有用。

按照以下步骤设置 sysdbopen() 和 sysdbclose() 过程以配置会话属性：

1. 将 IFX_NODBPROC 环境变量设置为任意值，包括 0，使数据库服务器绕过并阻止 sysdbopen( ) 或 sysdbclose( ) 过程的执行。
2. 编写 CREATE PROCEDURE 或 CREATE PROCEDURE FROM 语句定义特定用户或 PUBLIC 组的过程。
3. 测试此过程，例如，通过在 EXECUTE PROCEDURE 语句中使用 sysdbclose( )。

4. 取消设置 IFX_NODBPROC 环境变量以启用数据库服务器运行 sysdbopen( ) 或 sysdbclose( ) 过程。

**SYSDBOPEN 过程的示例**

以下过程设置指定用户的角色和 PDQ 优先级，并启用 NOVALIDATE 会话环境变量：

```
CREATE PROCEDURE oltp_user.sysdbopen()
    SET ROLE TO oltp;
    SET PDQPRIORITY 5;
    SET ENVIRONMENT NOVALIDATE '1';
    END PROCEDURE;
```

以下示例过程设置 PUBLIC 组的角色和 PDQ 优先级，并将 RETAINUPDATELOCKS 会话环境变量设置为 CURSOR STABILITY

```
CREATE PROCEDURE PUBLIC.sysdbopen()
    SET ROLE TO others;
    SET PDQPRIORITY 1;
    SET ENVIRONMENT
    RETAINUPDATELOCKS 'CURSOR STABILITY';
    END PROCEDURE
```

## 6.3 DataBlade 模块管理函数

从连接到支持显式事务日志记录的 GBase 8s 数据库的会话中，可以通过发出调用内置 **SYSBldPrepare( )** 的 SQL 函数注册或注销 DataBlade 模块。另一个内置函数，**SYSBldRelease( )**，返回本地数据库中 **SYSBldPrepare( )** 函数的版本字符串。

通过 SQL 函数注册和注销 DataBlade 模块的替代方法是使用 **BladeManager** 实用程序。**BladeManager** 实用程序可以执行多种 DataBlade 模块任务，包括注册、注销和显示有关 DataBlade 模块的信息。此实用程序支持命令行界面和图形用户界面。有关使用 BladeManager 实用程序的更多信息，请参阅 GBase 8s DataBlade 模块安装和注册指南。

### SYSBldPrepare 函数

**SYSBldPrepare( )** 是 GBase 8s 在服务器实例的所有数据库中定义的函数签名。可以使用它注册或注销 DataBlade 模块，另一种方法是使用 **BladeManager** 实用程序。

SYSBldPrepare( ) 函数具有以下定义：

```
CREATE FUNCTION gbasedbt.sysbldprepare (CHAR(64), CHAR(18))
    RETURNS INTEGER
    EXTERNAL NAME '$GBASEDBTDIR/extend/ifxmngr/ifxmngr.bld(SYSBldCustomPrepare)'
    LANGUAGE C;
```

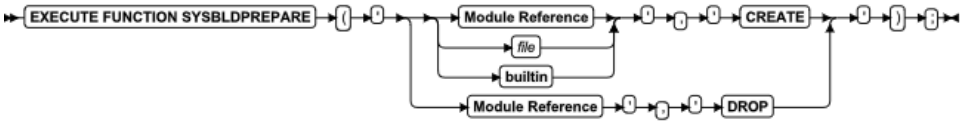
返回的整数表示此函数调用成功（0）或失败（非 0）。

以下限制应用于您调用此内置函数的数据库：

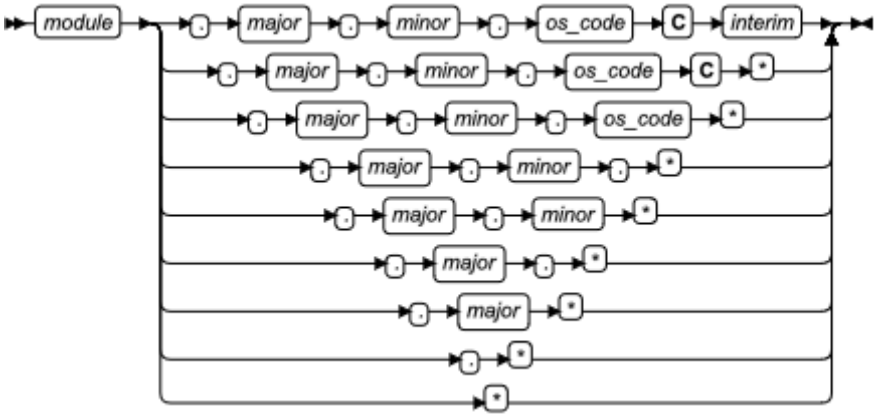
- GBase 8s 实例的配置文件中的最小 STACKSIZE 应该至少为 64 K 。（在某些系统中，缺省的 stack 大小为 32K，但是对使用 **SYSBldPrepare( )** 函数的数据库推荐使用 64 K ）。
- 该函数调用不能引用远程数据库。您只能在当前连接的本地数据库中注册或注销 DataBlade 模块。
- 数据库必须支持显式事务。您不能在兼容 ANSI/ISO 的数据库中调用此函数，或者在不支持事务日志记录的数据库中调用此函数。
- 在 Enterprise Replication 集群环境中，支持数据库的 GBase 8s 实例不能是远程辅助服务器。因为这种服务器不能直接支持 DDL 操作，就像此函数支持的 DDL 操作。如果要在辅助服务器上注册或注销 DataBlade 模块，您必须在辅助服务器复制的主服务器上注册或注销此模块。 .

这是调用 **SYSBldPrepare( )** 的语法:

SYSBldPrepare 函数




Module Reference




| 元素 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *module* | 要注册或注销的 DataBlade 模块的名称 | 要 'CREATE' *module* 必须安装在 **$GBASEDBTDIR/extend** 中。要 'DROP' 它则必须在当前数据库中注册。 | 字符串字符 |
| *file* | 列出一个或多个 DataBlade 模块的文件的名称，每个模块引用格式 | 必须在 **$GBASEDBTDIR/extend/ifxmngr** 目录中存在 | Character string with no suffix |
| *major* | 整数指定主要的 | 必须与已安装或已注册的 | 数字字符 |

| | | | |
|---|---|---|---|
| | GBase 8s 发布版本 | DataBlade 模块或通配符的主要版本匹配 | |
| *minor* | 整数指定 GBase 8s 次要的发布版本 | 必须与已安装或已注册的 DataBlade 模块或通配符的次要版本匹配 | 数字字符 |
| *os_code* | 支持的操作系统的大写字母代码 | 有效选项为 F 、H 、T 或 U。这些代码在 *DataBlade 模块安装与注册指南*的第一章中有所描述。 | 文字字符 |
| *interim* | 整数指定 GBase 8s 临时的发布版本 | 必须与已安装或已注册的 DataBlade 模块或通配符的临时版本匹配 | 数字字符 |

可以使用 SQL 的 EXECUTE FUNCTION 语句或 SPL 的 CALL 语句调用此函数。

**SYSBldPrepare( )** 的第一个参数指定要处理的 DataBlade 模块或文件。第二个参数是注册（'CREATE'）还是 注销（'DROP'）第一个参数必须指定一个 DataBlade 模块，而不是一个文件。

**指定 File 作为 First 参数**

如果 'CREATE' 是第二个参数，则第一个参数必须是单个模块引用或文本文件的名称，它指定一个或多个模块引用的列表，每个模块引用语法的格式如上面的语法图所示。（但是，此文本文件不能列出列出模块引用的另一个文本文件的名称。）。通过将有效文件指定为第一个参数，可以在单次调用 **SYSBldPrepare( )** 函数注册一组 DataBlade 模块。

该文件可以是您创建的文件，也可以是数据库服务器创建的 **builtin** 文件。**builtin** 文件包括 GBase 8s 分类为内置的 DataBlade 模块的列表。这些内置 DataBlade 您可与 GBase 8s 一起分布，并安装在 **$GBASEDBTDIR/extend** 文件系统中，但在它们注册到数据库之前无法访问。不支持用户对数据库服务器维护的此 **builtin** 文件的更新。

**模块引用中的版本字符串和星号（*）符号**

当第一个参数以 DataBlade 模块的名称开头时，您还可以执行句号（ .）分隔符之后的结束版本字符串。 完整的版本字符串与 SQL 的 **DBINFO('version full')** 函数或 **oninit -V** 实用程序的返回值的格式相同， 但是它基于 DataBlade 模块的发布版本。

DataBlade 模块名称或版本字符串可以用星号（*）通配符截断。SYSBldPrepare( ) 如何解释星号符号取决于第二个参数：

- 如果 'CREATE' 是第二个参数，则星号与指定模块的最高安装版本匹配。
- 如果 'DROP' 是第二个参数，则星号与在本地数据库中注册的 DataBlade 模块中的模块的注册版本相匹配。在数据库中可以注册给定 DataBlade 模块的最多一个版本 ，因此替换版本字符串的星号指定已注册的版本。

模块引用中不是最后一个字符的任何星号符号都将解释为文字字符，而不是通配符。

其中 SYSBldPrepare( ) 搜索第一个参数指定的模块取决于第二个参数：

- 如果 'CREATE' 是第二个参数，则该函数在安装 **$GBASEDBTDIR/extend** 目录中的模块之间进行搜索。.
- 如果 'DROP' 是第二个参数，则该函数将在本地数据库中注册的 DataBlade 模块的指定版本。因为在数据库中不能注册多个版本的给定 DataBlade 模块，所以替换版本字符串的星号指定已注册的版本。

**注册和注销 DataBlade 模块**

此函数的第二个参数必须是 'CREATE' 或 'DROP'：

- 使用 'CREATE' 注册第一个参数指定的已安装的 DataBlade 模块（或者一组已安装的 DataBlade 模块）。
- 使用 'DROP' 注销第一个参数指定的已安装的 DataBlade 模块。'DROP' 选项不能在对 **SYSBldPrepare( )** 的单次调用中注销多个 DataBlade 模块。

使用 'CREATE' 作为其第二个参数成功调用 SYSBldPrepare( ) 函数还会注册在第一个参数中指定的模块依赖的任何 DataBlade 模块。例如，下面的 SQL 语句注册 8.21.FC2 版本 的 Example DataBlade 模块 ，并在当前数据库中隐式注册了 R-tree DataBlade 模块的最新安装版本，其中 Example DataBlade 模块具有依赖性，如果 R-tree DataBlade 模块尚未在数据库中注册：

```
EXECUTE FUNCTION sysbldprepare ('example.8.21.FC2', 'create');
```

但是，如果已经在数据库中注册了相同的 DataBlade 模块的不同发行版本，则如果 'CREATE' 是第二个参数，则 m**SYSBldPrepare( )** 会升级。例如，上面的函数调用把 Example DataBlade 模块的版本 8.20.FC1 升级到版本 8.21.FC2，如果版本 8.20.FC1 在调用 **SYSBldPrepare( )** 时已经在同一个数据库中注册，但是 R-tree DataBlade 模块不会被隐式升级。

以下 SQL 语句使用星号表示法注销在数据库中注册的 Node 扩展的最高版本：

```
EXECUTE FUNCTION sysbldprepare ('Node.*', 'drop');
```

与注册操作不同，调用指定 'DROP' 作为第二个参数的 **SYSBldPrepare( )** 不会对第一个参数未指定的任何 DataBlade 模块自动生效。'DROP' 参数不会隐式注销与第一个参数指定的模块具有依赖关系的其它 DataBlade 模块。

**在事务中使用 SYSBldPrepare( )**

**SYSBldPrepare( )** 函数在内部使用显式事务。如果发出 BEGIN WORK 以及开始调用 **SYSBldPrepare( )** 的事务，则在同一事务中，但在调用 **SYSBldPrepare( )** 之前，DML 或 DDL 语句对数据库所做的任何更改的状态是不可预测的。当提交 **SYSBldPrepare( )** 的内部事务时，可能会提交来自 DML 或 DDL 操作的更改，从而使您无法通过 SQL 语句的词法顺序中函数调用之后的错误处理逻辑回滚这些更改。要避免此情况，请不要在显式开始的事务中调用 **SYSBldPrepare( )** 。

**调用 SYSBldPrepare( ) 中的异常**

如果尝试使用 'DROP' 选项注销在当前数据库中注册的另一个 DataBlade 模块所依赖的另一个 DataBlade 模块，则 **SYSBldPrepare( )** 函数会发出错误。例如，当在注册 Example DataBlade 模块时您不能使用此函数注销 R-tree DataBlade 模块 mod。

如果 **SYSBldPrepare( )** 试图注销未注册在数据库中的 DataBlade 模块， GBase 8s 也会发出错误。

以下示例显示尝试注册未安装的 DataBlade 模块以及生成的错误消息：

```
EXECUTE FUNCTION sysbldprepare ('node.2.33', 'create');

(U0001) - registerBlade - Unable to register node.2.33
  - DataBlade module not found
- check online log and sysblderrorlog table for more information
```

如果 IFX_EXTEND_ROLE 配置参数设置为 ON，那么调用此例程的授权仅对数据库服务器管理员（DBSA）和 DBSA 授予的 EXTEND 角色的其它人可用。缺省情况下，DBSA 是用户 **gbasedbt**。

此函数执行时发生的异常可能导致 **SYSBldPrepare( )** 发出不是 GBase 8s 错误消息的诊断错误消息。有关 **SYSBldPrepare( )** 可用发出的错误消息的信息，请参阅 GBase 8s ***DataBlade 模块安装和注册指南*** 。

### SYSBldRelease 函数

**SYSBldRelease( )** 是 GBase 8s 在服务器实例的所有数据库中定义的函数签名。你可以使用 SQL 的 EXECUTE FUNCTION 语句或 SPL 的 CALL 语句调用此函数，以返回 **SYSBldPrepare( )** 函数的版本字符串。

SYSBldRelease( ) 函数的定义如下：

```
CREATE FUNCTION gbasedbt.sysbldrelease()
        RETURNS LVARCHAR
        EXTERNAL NAME
        '$GBASEDBTDIR/extend/%SYSBLDDIR%/ifxmngr.bld(MackRelease)'
        LANGUAGE C NOT VARIANT;
        GRANT EXECUTE ON FUNCTION SYSBldRelease() TO PUBLIC;
```

该函数不采用参数。它返回版本字符串和 SYSBldPrepare( ) 函数的完成日期。返回的版本字符串具有以下格式：

***major*. *minor*. *os_code*C*interim***

此处的 C 是文字字符，***major*** 、***minor*** 、***os_code*** 和 ***interim*** 版本字符串元素具有相同的语义，这些术语包含在 **SYSBldPrepare( )** 函数的 Module Reference 段中，但没有星号(*)通配符的表示法。

当通过 **SYSBldPrepare( )** 问题联系 GBase 支持时，**SYSBldRelease( )** 非常有用。

**SYSBldRelease( )** 返回 **SYSBldPrepare( )** 正确的版本字符串之前，**SYSBldPrepare( )** 函数需要在同一数据库中至少调用一次。对 **SYSBldPrepare( )** 的调用不需要在与调用 **SYSBldRelease( )** 相同的会话中。

## 6.4 EXPLAIN_SQL 例程

GBase Data Studio Administration Edition 可以使用 EXPLAIN_SQL 例程获得 XML 格式的查询计划、解释 XM 、并且可视化地呈现计划。

GBase Data Studio 包含一组工具，用于管理，数据建模以及从数据服务器的数据构建查询。EXPLAIN_SQL 例程准备查询并在 XML 中返回查询计划。

如果您计划使用 GBase Data Studio 获得 Visual Explain 输出，您必须为 ONCONFIG 文件中的 SBSPACENAME 配置参数创建并指定缺省 sbspace 名。EXPLAIN_SQL 例程在此 sbspace 中创建 BLOB 对象。

如果使用 GBase Data Studio 的信息，请参阅 GBase Data Studio 文档。

## 6.5 UDR 定义例程

UDR 定义例程是内置例程，使用户能够执行各种任务，以开发或修改 GBase 8s 的外部用户定义例程，或者启用 GBase Data Server 驱动程序 JDBC 和 SQL 过程来访问 GBase 8s 和 DB2 数据库通过分布式关系数据库架构（DRDA）协议。

这些是内置 UDR 定义例程：

- **ifx_replace_module( )**
- **ifx_unload_module( )**
- **jvpcontrol( )**
- **sqlj.alter_java_path( )**
- **sqlj.install_jar( )**
- **sqlj.remove_jar( )**
- **sqlj.replace_jar( )**
- **sqlj.setUDTextName( )**
- **sqlj.unsetUDTextName( )**
- **sysgbase.Metadata( )**
- **sysgbase.SQLCAMessage( )**

**授权使用 UDR 定义例程**

如果 IFX_EXTEND_ROLE 配置参数设置为 'On' 或 1，则使用操作共享对象的内置例程的授权仅对数据库服务器管理员和 DBSA 授予 EXTEND 角色的用户可用。缺省情况下启用 IFX_EXTEND_ROLE。

对于不需要此安全功能的数据库，请参阅 *GBase 8s 管理员参考手册* 中的 IFX_EXTEND_ROLE 的描述，以获取有关 DBSA 如果通过重置来禁用此配置参数的信息。有关将 EXTEND 角色授予单个用户或 PUBLIC 组的语法，请参阅授予 EXTEND 角色主题。

### IFX_REPLACE_MODULE 函数

IFX_REPLACE_MODULE 函数将使用 C 语言编写的 UDR 的加载共享对象文件替换为具有不同名称或位置的新版本。如果 IFX_EXTEND_ROLE 配置参数设置为 'On' 或 1，则使用此功能的授权仅对数据库服务器管理员（DBSA）和 DBSA 已授予 EXTEND 角色的用户可用。

IFX_REPLACE_MODULE 函数

| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *new_module* | 要替换 *old_module* 指定的共享对象文件的新共享对象文件的完整路径名 | 共享对象文件必须与指定的路径名一起存在，其长度不能超过 255 个字节 | 引用字符串 |
| *old_module* | 要用 *new_module* 指定共享对象文件替换的共享对象文件的完整路径名 | 共享对象文件必须与指定的路径名一起存在，其长度不能超过 255 个字节 | 引用字符串 |

IFX_REPLACE_MODULE 函数是 DBA 特权函数，它返回的整数值表示共享对象文件替换操作的状态：

- 零（0）表示成功
- 负整数表示错误

不要使用 IFX_REPLACE_MODULE 函数重载相同名称的模块。如果发送到 IFX_REPLACE_MODULE 的旧的和新的模块的全名一样，则会发生预期外的结果。

**IFX_REPLACE_MODULE** 完成执行后，数据库服务器会老化 ***old_module*** 共享对象文件；即，**IFX_REPLACE_MODULE** 函数之后的所有语句将使用 ***new_module*** 共享对象文件中的 UDR，并且当使用它的任何语句完成是，旧模块将会被卸载。因此，在短时间内，***old_module*** 和 ***new_module*** 共享对象文件都可以驻留在内存中。如果此老化行为是不受欢迎的，请使用 **IFX_UNLOAD_MODULE** 函数完全卸载共享对象文件。

在 UNIX™ 上，例如，假设您希望替换 circle.so 共享库，它包含以 C 语言编写的 UDR。如果此库的旧版本驻留在 /usr/apps/opaque_types 目录，新版本在 /usr/apps/shared_libs 目录中，则使用下列 EXECUTE FUNCTION 语句执行 IFX_REPLACE_MODULE 函数：

```
EXECUTE FUNCTION ifx_replace_module(
        "/usr/apps/opaque_types/circle.so",
        "/usr/apps/shared_libs/circle.so", "C");
```

在 Windows™ 上，例如，假设您希望替换 circle.dll 动态链接库，其包含 C UDR。如果此库的旧版本驻留在 C:\usr\apps\opaque_types 目录，新版本在 C:\usr\apps\DLLs 目录，则使用下列 EXECUTE FUNCTION 语句执行 IFX_REPLACE_MODULE 函数：

```
EXECUTE FUNCTION ifx_replace_module(
        "C:\usr\apps\opaque_types\circle.dll",
        "C:\usr\apps\DLLs\circle.dll", "C");
```

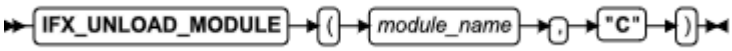
要在 GBase 8s ESQL/C 应用程序中执行 **IFX_REPLACE_MODULE** 函数，您必须将此函数与游标相关联。

有关如何使用 **IFX_REPLACE_MODULE** 替换共享对象文件的更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 中如何设计 UDR 的章节。有关如何使用 **IFX_UNLOAD_MODULE** 函数的信息，请参阅 IFX_UNLOAD_MODULE 函数 一节。

## IFX_UNLOAD_MODULE 函数

IFX_UNLOAD_MODULE 函数从共享内存卸载以 C 语言编写的 UDR 的共享对象文件。

IFX_UNLOAD_MODULE 函数




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *module_name* | 要卸载文件的完整路径名 | 共享对象文件必须存在并且未被使用。路径名最大长度为 255 字节。 | 引用字符串 |

IFX_UNLOAD_MODULE 函数是所有者特权函数，其所有者是用户 gbasedbt。它返回的整数值表示共享对象文件卸载操作的状态：

- 零（0）表示成功
- 负整数表示错误

**IFX_UNLOAD_MODULE** 函数只能卸载未使用的共享对象文件；也就是说，当没有执行的 SQL 语句(在任何数据库中)使用指定的共享对象文件中的任何 UDR 时。如果共享对象文件中的任何 UDR 当前正在使用，则 **IFX_UNLOAD_MODULE** 引发错误。

在 UNIX™ 上，例如，假设您希望卸载 circle.so 共享库，包含 C UDR。如果此库驻留在 /usr/apps/opaque_types 目录，可以使用以下 EXECUTE FUNCTION 语句执行 IFX_UNLOAD_MODULE 函数：

```
EXECUTE FUNCTION ifx_unload_module
        ("/usr/apps/opaque_types/circle.so", "C");
```

在 Windows™ 上，例如，假设希望卸载 circle.dll 动态链接库，包含 C UDR。如果该库在 C:\usr\apps\opaque_types 目录中，您可以使用以下 EXECUTE FUNCTION 语句执行 IFX_UNLOAD_MODULE 函数：

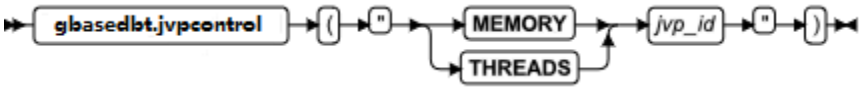
```
EXECUTE FUNCTION ifx_unload_module
        ("C:\usr\apps\opaque_types\circle.dll", "C");
```

有关如何使用 **IFX_REPLACE_MODULE( )** 和 **IFX_UNLOAD_MODULE( )** UDR 定义例程的更多信息，请参阅 *GBase 8s 用户定义的例程和数据类型开发者指南* 和 *GBase 8s DataBlade API 程序员指南*。

## 6.6 jvpcontrol 函数

jvpcontrol( ) 函数是内置迭代函数，可用于获得有关 Java™ 虚拟处理器（JVP）类的信息。

jvpcontrol 函数




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *jvp_id* | 要搜索信息的 Java™ 虚拟处理器（JVP）类的名称 | 指定的 Java 虚拟处理器类必须存在 | 标识符 |

必须将此函数与 Java 语言的中游标的等效项关联。

### 使用 MEMORY 关键字

当您指定 MEMORY 关键字时，jvpcontrol 函数返回您指定的 JVP 类的内存使用情况。以下示例请求有关名为 4 的 JVP 类的内存使用情况的信息：

```
EXECUTE FUNCTION GBASEDBT.JVPCONTROL ("MEMORY 4");
```

### 使用 THREADS 关键字

当您指定 THREADS 关键字时，jvpcontrol 函数返回在您指定的 JVP 类上运行的线程的列表。以下示例请求名为 4 的 JVP 类上的运行的线程的信息：

```
EXECUTE FUNCTION GBASEDBT.JVPCONTROL ("THREADS 4");
```

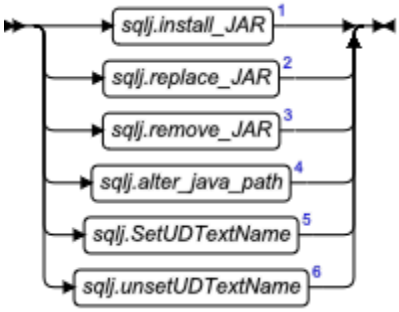
有关如何使用 **jvpcontrol( )** 和内置 **sqlj** 例程的信息，请参阅 *J/Foundation 开发者指南*。

## 6.7 SQLJ Driver 内置过程

使用 SQLJ Driver 内置过程执行以下任务之一：

- 安装、替换或移除一组 Java 类
- 要为包含在 JAR 文件中的 Java 类指定 Java™ 类解析的路径
- 映射或删除用户定义类型与其对应的 Java 类型之间的映射

SQLJ Driver Built-In Procedures




客户端应用程序必须指定 '**sqli**' 所有者名称以从兼容 ANSI 的数据库中调用这些函数。

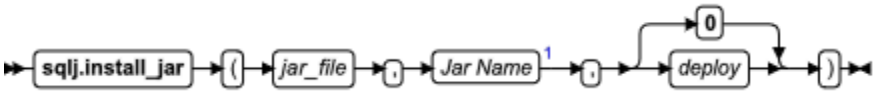
SQLJ 内置过程存储在 sysprocedures 系统目录表中。它们被分组到 sqlj 模式下。

**提示：** 对于任何 Java 静态方法，执行的第一个内置过程必须是 sqlj.install_jar( ) 过程。在创建 UDR 或将用户定义数据类型映射到 Java 类之前必须安装 JAR 文件。同样，不能使用任何其它 SQLJ 内置过程直到使用了 sqlj.install_jar( )。

### sqlj.install_jar

使用 sqlj.install_jar( ) 过程在当前数据库中安装 JAR 文件，并为它指定一个 JAR 标识符。

sqlj.install_jar



| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *deploy* | 导致过程在 JAR 文件中搜寻部署描述符文件的整数 | 无 | 精确数值 |
| *jar_file* | 包含以 Java 语言编写的 UDR 的 JAR 文件的 URL | URL 的最大长度为 255 字节 | 引用字符串 |

例如，假设 Java™ 类 **Chemistry** 包含下列静态方法 **explosiveReaction( )**：

```
public static int explosiveReaction(int ingredient)
```

此处的 **Chemistry** 类驻留在服务器电脑上的这个 JAR 文件中：

```
/students/data/Courses.jar
```

您可以使用以下 **sqlj.install_jar( )** 过程调用在当前数据库的 **Courses.jar** 文件中安装所有类：

```
EXECUTE PROCEDURE
sqlj.install_jar("file://students/data/Courses.jar", "course_jar");
```

**sqlj.install_jar( )** 过程指定 JAR ID，**course_jar**，分配给它在当前数据库中安装的 **Courses.jar** 文件。

在数据库中定义 JAR ID 之后，可以在创建和执行以 Java 语言编写的 UDR 使用该 JAR ID。（您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须具有 Java 语言的 Usage 特权，才能创建或删除 Java UDR。）

当您为第三个参数指定一个非零数字时，数据库服务器将搜索任何包含的部署描述符文件。例如，您可能希望包括包含 SQL 语句的描述符文件，以在 JAR 文件中注册和授予对 UDR 的权限。

如果启用了 IFX_EXTEND_ROLE 配置参数（缺省设置），那么只有 DBSA 或持有 EXTEND 角色的用户可以执行 **sqlj.install_jar( )** 过程。当禁用 IFX_EXTEND_ROLE 时，任何用户都可以执行 **sqlj.install_jar( )**。

## Jar 文件的文件权限

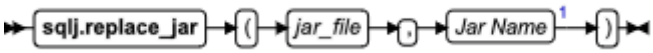
**sqlj.install_jar( )** 在数据库中安装 JAR 文件之后，为此文件声明一个 JAR ID。 GBase 8s 只有在安装了 GBase 8s 实例的用户（通常为用户 **gbasedbt**）具有读取该文件的权限的情况下才能访问该 JAR 文件的 JAR 文件所在的目录。例如，在 UNIX™ 系统上，这意味着尝试读取具有 600 个权限的 JAR 文件失败，并显示 FILENOTFOUND 异常。然而，在 **chmod** 实用程序将权限设置为 660（rw-rw----）后，相同的操作会成功。

您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须具有 Java 语言的 Usage 特权，才能创建或删除 Java UDR。

## sqlj.replace_jar

使用 **sqlj.replace_jar( )** 过程将先前安装的 JAR 文件替换为新版本。使用此语法时，您只需提供新的 JAR 文件，并为其指定要替换的文件的 JAR ID。

```
sqlj.replace_jar
```




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *jar_file* | 包含以 Java 语言编写的 UDR 的 JAR 文件的 URL | URL 的最大长度是 255 字节。 | 引用字符串 |

如果尝试替换由一个或多个 UDR 引用的 JAR 文件，那么数据库服务器会生成错误。在替换 JAR 文件之前，您必须删除引用的 UDR。

例如，以下调用将以前为 course_jar 标识符安装的 Courses.jar 文件替换为 Subjects.jar 文件：

```
EXECUTE PROCEDURE
sqlj.replace_jar("file://students/data/Subjects.jar", "course_jar");
```

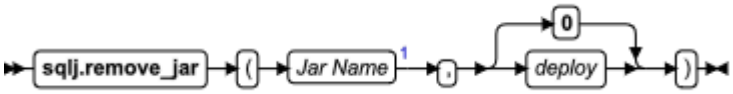
在替换 **Course.jar** 文件之前，必须使用 DROP FUNCTION（或 DROP ROUTINE）语句删除用户定义的函数 **sql_explosive_reaction( )**。（您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须具有 Java 语言的 Usage 特权，才能创建或删除 Java UDR。）

如果启用了 IFX_EXTEND_ROLE 配置参数（缺省设置），那么只有 DBSA 或持有 EXTEND 角色的用户可以执行 **sqlj.replace_jar( )** 过程。当禁用 IFX_EXTEND_ROLE 时，任何用户都可以执行 **sqlj.replace_jar( )**。

## sqlj.remove_jar

使用 sqlj.remove_jar( ) 过程从当前数据库移除先前安装的 JAR 文件。您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须具有 Java 语言的 Usage 特权，才能创建或删除 Java UDR。

sqlj.remove_jar




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *deploy* | 导致过程在 JAR 文件中搜寻部署描述符文件的整数 | 无 | 精确数值 |

当尝试移除被一个或多个 UDR 引用的 JAR 文件时，数据库服务器生成 46003 错误。您必须在替换此 JAR 文件前删除引用的 UDR。 非法 JAR 文件名称生成 46002 错误。

例如，下列 SQL 语句移除与 course_jar JAR ID 相关联的 JAR 文件：

```
DROP FUNCTION sql_explosive_reaction;
        EXECUTE PROCEDURE sqlj.remove_jar("course_jar");
```

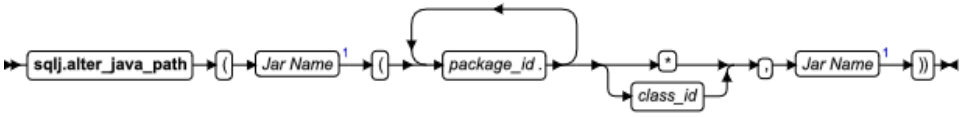
当您为第二个参数指定非零数字时，数据库服务器将搜索任何包含的部署描述符文件。例如，您可能希望包括包含 SQL 语句的描述符文件以撤销关联的 JAR 文件中的 UDR 特权，并从数据库中删除它们。

如果启用 IFX_EXTEND_ROLE 配置参数（缺省设置），那么只有 DBSA 或持有 EXTEND 角色的用户可以执行 **sqlj.remove_jar( )** 过程。当禁用 IFX_EXTEND_ROLE 时，任何用户都可以执行 **sqlj.remove_jar( )**。

## sqlj.alter_java_path

使用 sqlj.alter_java_path( ) 指定当例程管理器解析用 Java 语言编写的 UDR 的 JAR 文件的相关的 Java 类时使用 *jar 文件路径*。

sqlj.alter_java_path




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *class_id* | 包含实现 UDR 的方法的 Java™ 类 | Java 必须在 *jar_id* 指定的 JAR 文件中存在。标识符不能超过 255 个字节。 | 特定于语言 |
| *package_id* | 包含 Java 类的包的名称 | *package_id.class_id* 的完整限定标识符不能超过 255 个字符 | 特定于语言 |

您指定的 JAR ID （即要更改 JAR 文件路径的 JAR 标识和 JAR ID 的解析）必须都已使用 **sqlj.install_jar** 过程安装过。当调用使用 Java 语言编写的 UDR 时，例程管理器尝试加载 UDR 驻留的 Java 类。此时，它必须解析此 Java 类对其它 Java 类所做的引用。

这些类引用的三种类型是：

1. 引用 JVPCLASSPATH 配置参数指定的 Java 类（例如 **java.util.Vector** 的 Java 系统类）
2. 引用与 UDR 位于同一 JAR 文件中的类
3. 引用包含 UDR 的 JAR 文件之外的类。

例程管理隐式解析之前列表中的类型 1 和类型 2。要解析类型 3 的引用，它检查 JAR 文件路径中所有最近调用 **sqlj.alter_java_path( )** 指定的 JAR 文件。

如果例程管理器不能解析类引用，它会发出异常。例程管理器在执行类型 1 和类型 2 解析之后检查 JAR 文件路径以进行类引用。

如果希望从 JAR 文件路径指定的 JAR 文件加载 Java™ 类，请确保 Java 类*不*存在于 JVPCLASSPATH 配置参数中。否则，系统装入程序会首先采用此 Java 类，这可能导致加载不是您期望的类。

假设 sqlj.install_jar( ) 过程和 CREATE FUNCTION 已经如前面部分所述的执行。以下 SQL 语句调用 course_jar JAR 文件中的 sql_explosive_reaction( ) 函数：

```
EXECUTE PROCEDURE alter_java_path("course_jar", "(professor/*, prof_jar)");
EXECUTE FUNCTION sql_explosive_reaction(10000);
```

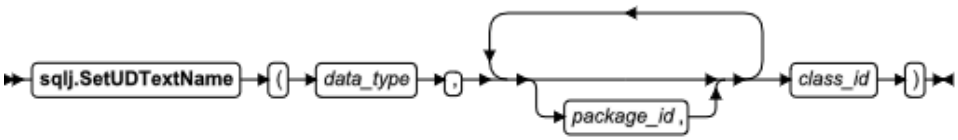
例程管理器尝试加载 **Chemistry** 类。它使用调用 **sqlj.alter_java_path( )** 指定的路径来解析任何类引用。因此，它检查 **prof_jar** 标识的 JAR 文件的 **professor** 包中的类。

如果启用了 IFX_EXTEND_ROLE 配置参数（缺省设置），那么只有 DBSA 或持有 EXTEND 角色的用户可以执行 **sqlj.alter_java_path( )** 过程。当 IFX_EXTEND_ROLE 被禁用时，任何用户都可以执行 **sqlj.alter_java_path( )**。（但是不管 IFX_EXTEND_ROLE 如何设置，您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须拥有 Java 语言的 Usage 特权，才能创建或删除 Java UDR。）

## sqlj.setUDTextName

使用 sqlj.setUDTextName( ) 过程定义用户定义的数据类型和 Java 类之间的映射。

sqlj.SetUDTextName




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *class_id* | 包含 Java 数据类型的 Java 类 | 限定名称 *package_id.class_id* 不能超过 255 个字节 | Java 标识符的特定于语言的规则 |
| *data_type* | 要创建映射的用户定义的类型 | 名称不能超过 255 个字节 | 标识符 |
| *package_id* | 包含 *class_id* Java 类的包的名称 | 与 *class_id* 的限制相同 | Java 标识符的特定于语言的规则 |

必须已经在 CREATE DISTINCT TYPE 、CREATE OPAQUE TYPE 或 CREATE ROW TYPE 语句中注册此用户定义的数据类型。

要查找用户定义数据类型的 Java 类，数据库服务器将在 JAR 文件路径中搜索，该路径由 **sqlj.alter_java_path( )**过程指定，有关 JAR 文件路径的更多信息，请参阅 sqlj.alter_java_path。

SQLJ 驱动程序查找 **CLASSPATH** 在客户端环境中指定的路径，然后向数据库服务器询问 Java 类的名称。

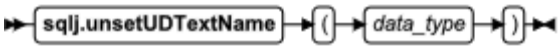
**setUDTextName( )** 例程是使用 Java 编程语言规范的 SQLJ:SQL 例程的扩展。

如果启用了 IFX_EXTEND_ROLE 配置参数（缺省设置），那么只有 DBSA 或持有 EXTEND 角色的用户可以执行 **setUDTextName( )** 过程。当 IFX_EXTEND_ROLE 被禁用时，任何用户都可以执行 **setUDTextName( )**。（但是不管 IFX_EXTEND_ROLE 如何设置，您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须拥有 Java 语言的 Usage 特权，才能创建或删除 Java™ UDR。）

### sqlj.unsetUDTextName

使用 sqlj.unsetUDTextName( ) 例程移除用户定义数据类型到 Java 类的映射。

sqlj.unsetUDTextName




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *data_type* | 要移除映射的用户定义的数据类型 | 必须存在 | 标识符 |

此例程删除 SQL 到 Java 的映射，从而从数据库服务器的共享内存中删除 Java 类的任何缓存副本。

**unsetUDTextName( )** 例程是使用 Java 编程语言规范的 SQLJ:SQL 例程的扩展。

如果启用了 IFX_EXTEND_ROLE 配置参数（缺省设置），那么只有 DBSA 或持有 EXTEND 角色的用户可以执行 **unsetUDTextName( )** 过程。当 IFX_EXTEND_ROLE 被禁用时，任何用户都可以执行 **unsetUDTextName( )**。（但是不管 IFX_EXTEND_ROLE 如何设置，您必须拥有数据库的 Resource 特权或 DBA 特权，并且还必须拥有 Java 语言的 Usage 特权，才能创建或删除 Java™ UDR。）

## 6.8 DRDA 支持函数

当 GBase 8s 实例配置为 DRDA 应用程序服务器时， GBase 8s 提供支持分布式关系数据库架构（DRDA）协议的内置函数。(这通过在配置文件中将 DBSERVERALIASES 设置为 DRDA 来实现。)

- **sysgbase.Metadata** 函数向驱动程序为 JDBC 和 SQL 客户端应用程序提供数据库元数据信息。
- **sysgbase.SCLCAMessage** 函数支持 DRDA 错误处理。

这些函数的 GBase 8s 实现符合 DR Level 5 SQLAM 7 标准。

### Metadata 函数

**sysgbase.Metadata** 函数是一个 SPL 例程，它可由 GBase Data Server 驱动程序调用，以便 JDBC 和 SQL 应用程序动态检索数据库元数据。**Metadata** 例程在配置为 DRDA 应用程序服务器的 GBase 8s 实例的每个数据库中自动创建，客户端应用程序必须指定 '**sysgbase**' 为所有者名称以从符合 ANSI 的数据库调用此函数。

它具有以下常规定义：

```
create procedure sysgbase.METADATA() returning
    integer as allProceduresAreCallable,
    integer as allTablesAreSelectable,
    integer as nullsAreSortedHigh,
    integer as nullsAreSortedLow,
    integer as nullsAreSortedAtStart,
    integer as nullsAreSortedAtEnd,
```

```
integer as usesLocalFiles,
integer as usesLocalFilePerTable,
integer as storesUpperCaseIdentifiers,
integer as storesLowerCaseIdentifiers,
integer as storesMixedCaseIdentifiers,
integer as storesLowerCaseQuotedIdentifiers,
integer as storesMixedCaseQuotedIdentifiers,
lvarchar(4096) as getSQLKeywords,
varchar(100) as getNumericFunctions,
varchar(100) as getStringFunctions,
varchar(100) as getSystemFunctions,
varchar(100) as getTimeDateFunctions,
varchar(25) as getSearchStringEscape,
varchar(25) as getExtraNameCharacters,
integer as supportsAlterTableWithAddColumn,
integer as supportsAlterTableWithDropColumn,
integer as supportsConvert,
varchar(255) as supportsConvertType,
integer as supportsDifferentTableCorrelationNames,
integer as supportsExpressionsInOrderBy,
integer as supportsOrderByUnrelated,
integer as supportsGroupBy,
integer as supportsGroupByUnrelated,
integer as supportsGroupByBeyondSelect,
integer as supportsMultipleResultSets,
integer as supportsMultipleTransactions,
integer as supportsCoreSQLGrammar,
integer as supportsExtendedSQLGrammar,
integer as supportsANSI92IntermediateSQL,
integer as supportsANSI92FullSQL,
integer as supportsIntegrityEnhancementFacility,
integer as supportsOuterJoins,
integer as supportsFullOuterJoins,
integer as supportsLimitedOuterJoins,
varchar(50) as getSchemaTerm,
varchar(50) as getProcedureTerm,
varchar(50) as getCatalogTerm,
integer as isCatalogAtStart,
varchar(50) as getCatalogSeparator,
integer as supportsSchemasInDataManipulation,
integer as supportsSchemasInProcedureCalls,
integer as supportsSchemasInTableDefinitions,
integer as supportsSchemasInIndexDefinitions,
```

integer as supportsSchemasInPrivilegeDefinitions,
integer as supportsCatalogsInDataManipulation,
integer as supportsCatalogsInProcedureCalls,
integer as supportsCatalogsInTableDefinitions,
integer as supportsCatalogsInIndexDefinitions,
integer as supportsCatalogsInPrivilegeDefinitions,
integer as supportsPositionedDelete,
integer as supportsPositionedUpdate,
integer as supportsSelectForUpdate,
integer as supportsStoredProcedures,
integer as supportsSubqueriesInComparisons,
integer as supportsUnion,
integer as supportsUnionAll,
integer as supportsOpenCursorsAcrossCommit,
integer as supportsOpenCursorsAcrossRollback,
integer as supportsOpenStatementsAcrossCommit,
integer as supportsOpenStatementsAcrossRollback,
integer as getMaxBinaryLiteralLength,
integer as getMaxCharLiteralLength,
integer as getMaxColumnNameLength,
integer as getMaxColumnsInGroupBy,
integer as getMaxColumnsInIndex,
integer as getMaxColumnsInOrderBy,
integer as getMaxColumnsInSelect,
integer as getMaxColumnsInTable,
integer as getMaxConnections,
integer as getMaxCursorNameLength,
integer as getMaxIndexLength,
integer as getMaxSchemaNameLength,
integer as getMaxProcedureNameLength,
integer as getMaxCatalogNameLength,
integer as getMaxRowSize,
integer as doesMaxRowSizeIncludeBlobs,
integer as getMaxStatementLength,
integer as getMaxStatements,
integer as getMaxTableNameLength,
integer as getMaxTablesInSelect,
integer as getMaxUserNameLength,
integer as getDefaultTransactionIsolation,
integer as supportsTransactions,
varchar(50) as supportsTransactionIsolationLevel,
integer as supportsDataDefinitionAndDataManipulationTransactions,
integer as supportsDataManipulationTransactionsOnly,

```
    integer as dataDefinitionCausesTransactionCommit,
    integer as dataDefinitionIgnoredInTransactions,
    varchar(100) as supportsResultSetType,
    varchar(100) as supportsResultSetConcurrency,
    varchar(100) as ownUpdatesAreVisible,
    varchar(100) as ownDeletesAreVisible,
    varchar(100) as ownInsertsAreVisible,
    varchar(100) as othersUpdatesAreVisible,
    varchar(100) as othersDeletesAreVisible,
    varchar(100) as othersInsertsAreVisible,
    varchar(100) as updatesAreDetected,
    varchar(100) as deletesAreDetected,
    varchar(100) as insertsAreDetected,
    integer as supportsBatchUpdates,
    integer as supportsSavepoints,
    integer as supportsGetGeneratedKeys
```

## sysgbase.SQLCAMessage 函数

缺省情况下，用于 JDBC 的 GBase Data Server Driver 和用于 GBase 8s 的 SQL 不返回本地化错误消息。但是当在连接 URL 中设置 "retrieveMessagesFromServerOnGetMessage=true" 属性时，需要来自服务器的详细和本地化的错误消息。

**SQLCAMessage** 函数是一个 SPL 例程，支持从远程 DB2 或 GBase 8s 数据库服务器到使用分布式关系数据库架构（DRDA）协议的客户端应用程序检索详细的错误消息文本。在配置为 DRDA 应用程序服务器的 GBase 8s 实例的每个数据库中自动创建 **SQLCAMessage** 例程。用于 JDBC 和 SQL 客户端应用程序 GBase Data Server 驱动程序必须指 '**sysgbase**' 所有者名称，以从符合 ANSI 的数据库调用此函数。

**SQLCAMessage** 函数基于 SQL 通信区域（SQLCA）中的 **SQLSTATE** 代码检索本地化的错误消息。

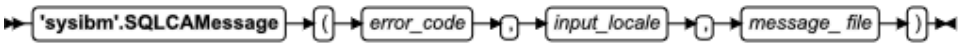
此函数的定义使用 IN 、OUT 和 INOUT 参数：

```
CREATE function sysgbase.SQLCAMessage (
        IN SQLCode          INTEGER,
        IN SQLErrml         SMALLINT,
        IN SQLErrmc         VARCHAR(70),
        IN SQLErrp          CHAR(8),
        IN SQLErrd0         INTEGER,
        IN SQLErrd1         INTEGER,
        IN SQLErrd2         INTEGER,
        IN SQLErrd3         INTEGER,
        IN SQLErrd4         INTEGER,
        IN SQLErrd5         INTEGER,
        IN SQLWarn          CHAR(11),
```

```
    IN SQLState          CHAR(5),
    IN MessageFileName VARCHAR(20),
    INOUT Locale       VARCHAR(33),
    OUT   Message    LVARCHAR(4096),
    OUT   Rcode          INTEGER)
    RETURNING   INTEGER
    EXTERNAL NAME '(SQLCAMessage)'
    LANGUAGE C
```

要调用此函数，可以使用此语法：

sysgbase.SQLCAMessage




| 参数 | 描述 | 限制 | 语法 |
|---|---|---|---|
| *error_number* | 错误的 SQLCODE 值 | 必须存在 | 表达式 |
| *input_locale* | 接收消息的输入语言环境的名称。缺省为 U.S. English 语言环境（en_us） | 必须存在 | 标识符 |
| *message_file* | 消息文件的名称 | 必须存在 | 路径名 |

该函数从指定的 **SQLCODE** 和 ***input_locale*** 的指定 ***message_file*** 检索文本。返回的代码表示调用执行 **SQLCAMessage** 例程成功或失败。

GBase 8s DRDA 应用程序试图使用指定的输出参数检索错误消息文本：

- **SQLCODE**
- ***input_locale*** 和
- ***message_file***

GBase 8s DRDA 应用程序试图使用指定的输出参数检索错误消息文本：如果 **MessageFileName** 参数 ***message_file*** 为 NULL ，则使用缺省的消息文件（**errmsgtxt**）。如果使用指定的 ***input_locale*** 检索失败，则使用缺省语言环境（**en_us**）来检索错误消息。如果适用，令牌数组用于替换检索的消息文本中的标记。

如果检索成功，

- SQLCODE 从错误消息中移除，
- 错误消息复制到 OUT 参数 'Message'
- 用于检索消息的区域设置将复制到 INOUT 参数 'Locale'。

如果检索失败，则错误消息文本 "Message not found" 将出现在 **Message** 参数中。

对于这两种情况，OUT 参数 **Rcod** 设置为执行此 SPL 例程的返回码。

**ISAM** 错误的详细信息从 **SQLERRD[0]** 值提供。**ISAM** 错误消息连接到实际的错误消息字符串并返回到应用程序。

对于 **SQLCAMessage** 函数可以返回相应错误消息文本的 **SQLSTATE** 值的代码，请参阅 SQLSTATE 代码列表。

## 6.9 SQL 包扩展

SQL 包扩展提供可在与 GBase 8s 以外的数据库服务器兼容的应用程序中使用的 SPL 例程。SQL 包扩展是内置扩展。您必须手动注册扩展。

SQL 包扩展中包括以下模块：

- DBMS_ALERT
- DBMS_LOB
- DBMS_OUTPUT
- DBMS_RANDOM
- UTL_FILE

### DBMS_ALERT 包

DBMS_ALERT 包提供一组用于注册警报、发送警报和接收警报的过程。注册包时，警报会存储在数据库中创建的 DBMS_ALERT_EVENTS、DBMS_ALERT_REGISTERED 和 DBMS_ALERT_SIGNALED 表中。当您想要针对特定事件发送警报时，DBMS_ALERT 包中的过程十分有用。例如，您可能想要在因为一个或多个表发生更改而导致触发器激活时发送警报。
DBMS_ALERT 包具有以下系统定义的例程。

### REGISTER 过程

REGISTER 过程注册当前会话以接收指定的警报。
语法
>>-DBMS_ALERT.REGISTER--(--name--)----------><
过程参数
name
类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

### REMOVE 过程

REMOVE 过程从当前会话中除去对指定警报的注册。
语法
>>-DBMS_ALERT.REMOVE--(--name--)----------><
过程参数
name
类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

### REMOVEALL 过程

REMOVEALL 过程从当前会话中除去对所有警报的注册。
语法
>>-DBMS_ALERT.REMOVEALL----------><

### SET_DEFAULTS

SET_DEFAULTS 过程设置 WAITONE 和 WAITANY 过程使用的轮询时间间隔。
语法
>>-DBMS_ALERT.SET_DEFAULTS--(--sensitivity--)----------><

过程参数

sensitivity

类型为 INTEGER 的输入参数，用于指定 WAITONE 和 WAITANY 过程检查信号的时间间隔（以秒为单位）。如果未指定值，那么缺省情况下时间间隔为 1 秒。

## SIGNAL 过程

SIGNAL 过程在指定的警报出现时发出信号。信号包括随警报传递的消息。发出 SIGNAL 调用时，该消息将分发至侦听器（针对警报注册的进程）。

语法

>>-DBMS_ALERT.SIGNAL--(--name--,--message--)------------------><

过程参数

name

类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

message

类型为 VARCHAR(32672) 的输入参数，用于指定随此警报传递的信息。发生警报时，WAITANY 或 WAITONE 过程可返回此消息。

## WAITANY 过程

WAITANY 过程等待任何已注册的警报出现。

语法

>>-DBMS_ALERT.WAITANY--(--name--,--message--,--status--,--timeout--)-><

过程参数

name

类型为 VARCHAR(128) 的输出参数，其中包含警报的名称。

message

类型为 VARCHAR(32672) 的输出参数，其中包含 SIGNAL 过程发送的消息。

status

类型为 INTEGER 的输出参数，其中包含过程返回的状态码。可能为以下值：

0

发生了警报。

1

发生了超时。

timeout

类型为 INTEGER 的输入参数，用于指定等待警报的时间量（以秒为单位）。

## WAITONE 过程

WAITONE 过程等待指定的警报出现。

语法

>>-DBMS_ALERT.WAITONE--(--name--,--message--,--status--,--timeout--)-><

过程参数

name

类型为 VARCHAR(128) 的输入参数，用于指定警报的名称。

message

类型为 VARCHAR(32672) 的输出参数，其中包含 SIGNAL 过程发送的消息。

status

类型为 INTEGER 的输出参数，其中包含过程返回的状态码。可能为以下值：
0
发生了警报。
1
发生了超时。
timeout
类型为 INTEGER 的输入参数，用于指定等待所指定警报的时间量（以秒为单位）。

## DBMS_LOB 包

DBMS_LOB 包能够对大对象执行操作。在描述各个过程和函数的以下各部分中，如果大对象为 BLOB，那么长度和偏移量以字节计。如果大对象为 CLOB，那么长度和偏移量以字符计。DBMS_LOB 包支持最大 10M 字节的 LOB 数据。在分区数据库环境中，如果在 SELECT 语句的 WHERE 子句中执行下列任何例程，那么将收到错误：dbms_lob.compare、dbms_lob.get_storage_limit、dbms_lob.get_length、dbms_lob.instr、dbms_lob.isopen、dbms_lob.substr。包中提供的公用变量：lob_readonly、lob_readwrite。

## APPEND 过程

APPEND 过程能够将一个大对象附加至另一个大对象。注：两个大对象必须为同一类型。
语法
>>-APPEND_BLOB--(--dest_lob--,--src_lob--)--------------------><
>>-APPEND_CLOB--(--dest_lob--,--src_lob--)--------------------><
参数
dest_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定目标对象的大对象定位器。必须与 src_lob 为同一数据类型。
src_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定源对象的大对象定位器。必须与 dest_lob 为同一数据类型。

## COMPARE 函数

COMPARE 函数针对两个大对象的给定偏移量处的给定长度执行确切的逐字节比较。
该函数返回：
零，如果两个大对象的指定偏移量处的指定长度完全相同
非零，如果这些对象不相同
空，如果 amount、offset_1 或 offset_2 小于零。
比较的大对象必须为同一数据类型。
语法
阅读语法图跳过直观语法图
>>-COMPARE--(--lob_1--,--lob_2--------------------------------->
>--+-------------------------------------------+--)---------><
'-,--amount--+----------------------------+-'
'-,--offset_1--+-------------+-'
'-,--offset_2-'
参数

lob_1
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要比较的第一个大对象的定位器。必须与 lob_2 为同一数据类型。
lob_2
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要比较的第二个大对象的定位器。必须与 lob_1 为同一数据类型。
amount
类型为 INTEGER 的可选输入参数。如果大对象的数据类型为 BLOB，那么将针对 amount 字节进行比较。如果大对象的数据类型为 CLOB，那么将针对 amount 字符进行比较。缺省值为大对象的最大大小。
offset_1
类型为 INTEGER 的可选输入参数，用于指定第一个大对象中开始比较的位置。第一个字节（或字符）为偏移 1。缺省值为 1。
offset_2
类型为 INTEGER 的可选输入参数，用于指定第二个大对象中开始比较的位置。第一个字节（或字符）为偏移 1。缺省值为 1。

## COPY 过程

COPY 过程能够将一个大对象复制到另一个大对象。源大对象和目标大对象必须为同一数据类型。
语法
阅读语法图跳过直观语法图

```
>>-COPY_BLOB--(--dest_lob--,--src_lob--,--amount---------------->
>--+-----------------------------------+--)------------------><
   '-,--dest_offset--+---------------+-'
                     '-,--src_offset-'
>>-COPY_CLOB--(--dest_lob--,--src_lob--,--amount---------------->
>--+-----------------------------------+--)------------------><
   '-,--dest_offset--+---------------+-'
                     '-,--src_offset-'
```

参数
dest_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定 src_lob 要复制到的大对象的定位器。必须与 src_lob 为同一数据类型。
src_lob
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要从中复制 dest_lob 的大对象的定位器。必须与 dest_lob 为同一数据类型。
amount
类型为 INTEGER 的输入参数，用于指定要复制的 src_lob 的字节或字符数。
dest_offset
类型为 INTEGER 的可选输入参数，用于指定目标大对象中应开始写入源大对象的位置。第一个位置为偏移 1。缺省值为 1。
src_offset

类型为 INTEGER 的可选输入参数，用于指定源大对象中应开始复制到目标大对象的位置。第一个位置为偏移 1。缺省值为 1。

## ERASE 过程

ERASE 过程能够擦除大对象的一部分。要擦除大对象意味着要将指定部分替换为零字节填充符（对于 BLOB）或空格（对于 CLOB）。大对象的实际大小不会改变。

语法

```
>>-ERASE_BLOB--(--lob_loc--,--amount--+-----------+--)--------><
                                      '-,--offset-'
>>-ERASE_CLOB--(--lob_loc--,--amount--+-----------+--)--------><
                                      '-,--offset-'
```

参数

lob_loc

类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要擦除的大对象的定位器。

amount

类型为 INTEGER 的输入或输出参数，用于指定要擦除的字节或字符数。

偏移量

类型为 INTEGER 的可选输入参数，用于指定大对象中开始擦除的位置。第一个字节或字符为位置 1。缺省值为 1。

## GETLENGTH 函数

GETLENGTH 函数返回大对象的长度。该函数返回 INTEGER 值，该值反映大对象的长度，对于 BLOB，长度以字节计，对于 CLOB，长度以字符计。

语法

```
>>-GETLENGTH--(--lob_loc--)---------------------------------><
```

参数

lob_loc

类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要获取其长度的大对象的定位器。

## INSTR 函数

INSTR 函数返回大对象中指定模式第 n 次出现的位置。该函数返回 INTEGER 值，该值表示大对象中模式第 n 次（由 *nth* 指定）出现的位置。此值从 *offset* 给定的位置开始。

语法

```
>>-INSTR--(--lob_loc--,--pattern--+----------------------+----->
                                  '-,--offset--+--------+-'
                                               '-,--nth-'
>--)-------------------------------------------------------><
```

参数

lob_loc

类型为 BLOB 或 CLOB 的输入参数，用于指定要在其中搜索 pattern 的大对象的定位器。

pattern

类型为 BLOB(32767) 或 VARCHAR(32672) 的输入参数，用于指定要与大对象匹配的字节或字符的模式。

如果 lob_loc 为 BLOB，那么 pattern 必须为 BLOB；如果 lob_loc 为 CLOB，那么 pattern 必须为 VARCHAR。

偏移量
类型为 INTEGER 的可选输入参数，用于指定 lob_loc 中开始搜索 pattern 的位置。第一个字节或字符为位置 1。缺省值为 1。
nth
类型为 INTEGER 的可选参数，用于指定搜索 pattern 的次数（从 offset 给定的位置开始）。缺省值为 1。

## READ 过程

READ 过程能够将大对象的一部分读取到缓冲区。
语法
>>-READ_BLOB--(--lob_loc--,--amount--,--offset--,--buffer--)---><
>>-READ_CLOB--(--lob_loc--,--amount--,--offset--,--buffer--)---><
参数
lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要读取的大对象的定位器。
amount
类型为 INTEGER 的输入或输出参数，用于指定要读取的字节或字符数。
偏移量
类型为 INTEGER 的输入参数，用于指定要开始读取的位置。第一个字节或字符为位置 1。
buffer
类型为 BLOB(32762) 或 VARCHAR(32672) 的输出参数，用于指定要接收大对象的变量。如果 lob_loc 为 BLOB，那么 buffer 必须为 BLOB。如果 lob_loc 为 CLOB，那么 buffer 必须为 VARCHAR。

## SUBSTR 函数

SUBSTR 函数能够返回大对象的一部分。
该函数返回函数所读取大对象的返回部分的 BLOB(32767)（对于 BLOB）或 VARCHAR（对于 CLOB）值。
语法
>>-SUBSTR--(--lob_loc--+-------------------------+--)---------><
'-,--amount--+-----------+-'
'-,--offset-'
参数
lob_loc
类型为 BLOB(10M) 或 CLOB(10M) 的输入参数，用于指定要读取的大对象的定位器。
amount
类型为 INTEGER 的可选输入参数，用于指定要返回的字节或字符数。缺省值为 32,767。
偏移量
类型为 INTEGER 的可选输入参数，用于指定大对象中开始返回数据的位置。第一个字节或字符为位置 1。缺省值为 1。

## TRIM 过程

TRIM 过程能够将大对象截断为指定长度。
语法
>>-TRIM_BLOB--(--lob_loc--,--newlen--)------------------------><

>>-TRIM_CLOB--(--lob_loc--,--newlen--)----------><

参数

lob_loc

类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要修剪的大对象的定位器。

newlen

类型为 INTEGER 的输入参数，用于指定大对象要修剪至的新字节或字符数。

## WRITE 过程

WRITE 过程能够将数据写入大对象。大对象中位于指定偏移量处并且具有给定长度的任何现有数据将由缓冲区中提供的数据覆盖。

语法

>>-WRITE_BLOB--(--lob_loc--,--amount--,--offset--,--buffer--)--><

>>-WRITE_CLOB--(--lob_loc--,--amount--,--offset--,--buffer--)--><

参数

lob_loc

类型为 BLOB(10M) 或 CLOB(10M) 的输入或输出参数，用于指定要写入的大对象的定位器。

amount

类型为 INTEGER 的输入参数，用于指定 buffer 中要写入大对象的字节或字符数。

偏移量

类型为 INTEGER 的输入参数，用于指定从大对象开头到要开始执行写入操作的偏移量（以字节或字符为单位）。大对象的开始值为 1。

buffer

类型为 BLOB(32767) 或 VARCHAR(32672) 的输入参数，其中包含要写入大对象的数据。如果 lob_loc 为 BLOB，那么 buffer 必须为 BLOB。如果 lob_loc 为 CLOB，那么 buffer 必须为 VARCHAR。

## DBMS_OUTPUT 包

DBMS_OUTPUT 包提供一组用于将消息（文本行）放到消息缓冲区中以及从消息缓冲区获取消息的过程。在应用程序调试期间，当您需要将消息写入标准输出时，这些过程十分有用。使用命令行处理器 (CLP) 命令 SET SERVEROUTPUT ON 将输出重定向至标准输出。自主过程内不支持 DISABLE 和 ENABLE 过程。自主过程是这样一个过程：调用时，它在新事务中以独立于原始事务的方式执行。

## DISABLE 过程

DISABLE 过程禁用消息缓冲区。此过程运行后，将废弃消息缓冲区中的所有消息。对 PUT、PUT_LINE 或 NEW_LINE 过程的调用会被忽略，并且不会向发送方返回任何错误。

语法

>>-DBMS_OUTPUT.DISABLE---------------------------------------><

## ENABLE 过程

ENABLE 过程启用消息缓冲区。进行单个会话期间，应用程序可将消息放到消息缓冲区中以及从消息缓冲区获取消息。

语法

>>-DBMS_OUTPUT.ENABLE--(--buffer_size--)----------------------><

过程参数

buffer_size
类型为 INTEGER 的输入参数，用于指定消息缓冲区的最大长度（以字节为单位）。如果对 buffer_size 指定小于 2000 的值，那么缓冲区大小将设置为 2000。如果该值为 NULL，那么缺省缓冲区大小为 20000。

## GET_LINE 过程

GET_LINE 过程从消息缓冲区获取文本行。该文本必须以行结束字符序列终止。
语法
>>-DBMS_OUTPUT.GET_LINE--(--line--,--status--)----------><
过程参数
line
类型为 VARCHAR(32672) 的输出参数，用于从消息缓冲区返回文本行。
status
类型为 INTEGER 的输出参数，用于指示是否从消息缓冲区返回了行：
0 指示已返回行
1 指示未返回行

## GET_LINES 过程

GET_LINES 过程从消息缓冲区获取一行或多行文本并将该文本存储在集合中。该文本的每行必须以行结束字符序列终止。
语法
>>-DBMS_OUTPUT.GET_LINES--(--lines--,--numlines--)----------><
过程参数
lines
类型为 DBMS_OUTPUT.CHARARR 的输出参数，用于从消息缓冲区返回文本行。类型 DBMS_OUTPUT.CHARARR 在内部定义为 VARCHAR(32672) ARRAY[2147483647] 数组。
numlines
类型为 INTEGER 的输入和输出参数。用作输入时，指定要从消息缓冲区检索的行数。用作输出时，指示从消息缓冲区检索到的实际行数。如果 numlines 的输出值小于输入值，那么消息缓冲区中没有其他行。

## NEW_LINE 过程

NEW_LINE 过程将行结束字符序列放到消息缓冲区中。
语法
>>-DBMS_OUTPUT.NEW_LINE----------><

## PUT 过程

PUT 过程将字符串放到消息缓冲区中。字符串结尾不会写入任何行结束字符序列。
语法
>>-DBMS_OUTPUT.PUT--(--item--)----------><
过程参数
item
类型为 VARCHAR(32672) 的输入参数，用于指定要写入消息缓冲区的文本。

## PUT_LINE 过程

PUT_LINE 过程将包括行结束字符序列的单行放到消息缓冲区中。

语法

>>-DBMS_OUTPUT.PUT_LINE--(--item--)--------------------------><

过程参数

item

类型为 VARCHAR(32672) 的输入参数，用于指定要写入消息缓冲区的文本。

## DBMS_RANDOM 包

DBMS_RANDOM 包提供生成随机数的机制。使用 INITIALIZE 过程可设置种子值，随机数生成器使用该值来生成数目。重复足够次数后，生成的一些值可能重复。要降低值重复的可能性，请使用 SEED 过程定期更改种子值。

## INITIALIZE 过程

INITIALIZE 过程使用指定的整数种子值初始化系统包且该过程为可选。

语法

>>-DBMS_RANDOM_INITIALIZE ()---------------------------------><

此示例：

execute procedure dbms_random_initialize (17809465);复制代码

返回此输出：

Routine executed.

## SEED 过程

SEED 过程使用指定的整数值重置种子值。

语法

>>-DBMS_RANDOM_SEED ()----------------------------------------><

此示例：

execute procedure dbms_random_seed (-45902345);复制代码

返回此输出：

Routine executed.

## RANDOM 函数

RANDOM 函数使用种子值返回随机整数。

语法

>>-DBMS_RANDOM_RANDOM ()-------------------------------------><

此示例：

insert into random_test VALUES (0, dbms_random_random());复制代码

返回此输出：

1 row(s) inserted.

## TERMINATE 过程

TERMINATE 过程通过将种子值重置为 0 终止使用系统包且该过程为可选。

语法

>>-DBMS_RANDOM_TERMINATE ()---------------------------------><

此示例：

execute procedure dbms_random_terminate ();复制代码

返回此输出：

Routine executed.

## UTL_FILE 包

UTL_FILE 包提供一组用于读取以及写入数据库服务器文件系统上的文件的例程。

## FCLOSE 过程

FCLOSE 过程关闭指定的文件。

语法

```
>>-UTL_FILE.FCLOSE--(--file--)-------------------------------><
```

过程参数

file

类型为 UTL_FILE.FILE_TYPE 的输入或输出参数，其中包含文件句柄。文件关闭时，此值设置为 0。

## FCLOSE_ALL 过程

FCLOSE_ALL 过程关闭所有打开的文件。即使没有打开的文件要关闭，该过程也会成功运行。

语法

```
>>-UTL_FILE.FCLOSE_ALL---------------------------------------><
```

## FFLUSH 过程

FFLUSH 过程将写入缓冲区中未写入的数据强制写入文件。

语法

```
>>-UTL_FILE.FFLUSH--(--file--)-------------------------------><
```

过程参数

file

类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

## FOPEN 函数

FOPEN 函数打开文件以执行 I/O。

语法

```
>>-UTL_FILE.FOPEN--(--location--,--filename--,--open_mode--+-----------------+--)-><
                                                           '-,--max_linesize-'
```

返回值

此函数返回类型为 UTL_FILE.FILE_TYPE 的值，用于指示所打开文件的文件句柄。

函数参数

location

类型为 VARCHAR(128) 的输入参数，用于指定包含该文件的目录的别名。

filename

类型为 VARCHAR(255) 的输入参数，用于指定该文件的名称。

open_mode

类型为 VARCHAR(10) 的输入参数，用于指定该文件的打开方式：

a

附加到文件

r

读取文件

w

写入文件

max_linesize

类型为 INTEGER 的可选输入参数，用于指定行的最大大小（以字符为单位）。缺省值为 1024 字节。在读取方式下，如果尝试读取大小超过 max_linesize 的行，那么会抛出异常。在写入和附加方式下，如果尝试写入大小超过 max_linesize 的行，那么会抛出异常。行结束字符不计入行大小。

### GET_LINE 过程

GET_LINE 过程从指定的文件获取文本行。该文本行不包括行结束终止符。没有其他要读取的行时，该过程会抛出 NO_DATA_FOUND 异常。

语法

```
>>-UTL_FILE.GET_LINE--(--file--,--buffer--)--------------------><
```

过程参数

file

类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含所打开文件的文件句柄。

buffer

类型为 VARCHAR(32672) 的输出参数，其中包含文件中的文本行。

### NEW_LINE 过程

NEW_LINE 过程将行结束字符序列写入指定的文件。

语法

```
>>-UTL_FILE.NEW_LINE--(--file--+----------+--)-----------------><
                               '-,--lines-'
```

过程参数

file

类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

lines

类型为 INTEGER 的可选输入参数，用于指定要写入文件的行结束字符序列的数目。缺省值为 1。

### PUT 过程

PUT 过程将字符串写入指定的文件。字符串结尾不会写入任何行结束字符序列。

语法

```
>>-UTL_FILE.PUT--(--file--,--buffer--)------------------------><
```

过程参数

file

类型为 UTL_FILE.FILE_TYPE 的输入参数，其中包含文件句柄。

buffer

类型为 VARCHAR(32672) 的输入参数，用于指定要写入文件的文本。

# 附录：GBase 8s 的 SQL 关键字

该附录列出的 GBase 8s SQL 中实现的关键字。

ISO 标准 SQL 语言有很多关键字。一些被指定为*保留字*，而另一些指定为***非保留字***。 在 ISO SQL 中，保留字不能用作数据库对象的标识符，例如：表、列等。要在有效的 SQL 语句中使用保留字作为名称，需要定界标识符（定界标识符）包含在双引号（" "）之间。

相反， GBase 8s 数据库服务器实现的 SQL 方言在遵守标识符（标识符）规则的字符串意义上几乎没有保留字，但是当用作标识符时，总是产生编译错误或运行错误。如果定义与内置 SQL 函数、表达式运算符具有相同的名称的 SPL 例程，则应用程序可能会遇到受限制的功能，或者意外的结果。

请不要将此附录中的任何关键字声明为 SQL 标识符。如果这样做，如果标识符出现在关键字有效的上下文中国，则可能出现错误或语义模糊。此外，您的代码将更难以阅读和维护。 GBase 8s 为内置例程和数据库对象保留前缀 ifx_ 和 **sys** 。不要在数据库结构中使用 C 或 C++ （或者嵌入式模式中使用的任何其它编程语言）。以下按字母顺序排列的列表中的符号 IFX_* 和 SYS*（其中 * 是任意字符串的通配符）。这些表示在数据库对象的用户定义标识符中应避免使用这些前缀。（本附录中未列出以这些前缀作为开头的 SQL 关键字，其目标是帮助 GBase 8s 用户避免数据库服务器内部使用的名称。）

如果接收到到与导致错误的 SQL 语句无关的错误消息，请查看本附录，了解是否将关键字用作标识符。

要避免使用关键字作为标识符，您可以使用所有者名称限定标识符或修改标识符。例如，并非将数据库对象命名为 **CURRENT**，而是将它命名为 **o_current** 或 **juanita.current**。有关使用关键字作为标识符存在的潜在问题的讨论，以及指定关键字的附加解决方法，请参阅使用关键字作为标识符。有关在 SQL 应用程序中使用关键字作为标识符的更多信息，请参阅 *GBase 8s SQL 教程指南*。

**A**

- AAO
- ABS
- ABSOLUTE
- ACCESS
- ACCESS_METHOD
- ACCOUNT
- ACOS
- ACOSH
- ACTIVE
- ADD
- ADDRESS
- ADD_MONTHS
- ADMIN
- AFTER
- AGGREGATE
- ALIGNMENT
- ALL
- ALL_ROWS
- ALLOCATE
- ALTER
- AND
- ANSI
- ANY
- APPEND

- AQT
- ARRAY
- AS
- ASC
- ASCII
- ASIN
- ASINH
- ASYNC
- AT
- ATAN
- ATAN2
- ATANH
- ATTACH
- ATTRIBUTES
- AUDIT
- AUTHENTICATION
- AUTHID
- AUTHORIZATION
- AUTHORIZED
- AUTO
- AUTOFREE
- AUTOLOCATE
- AUTO_READAHEAD
- AUTO_REPREPARE
- AUTO_STAT_MODE
- AVG
- AVOID_EXECUTE
- AVOID_FACT
- AVOID_FULL
- AVOID_HASH
- AVOID_INDEX
- AVOID_INDEX_SJ
- AVOID_MULTI_INDEX
- AVOID_NL
- AVOID_STAR_JOIN

**B**

- BARGROUP
- BASED
- BEFORE
- BEGIN
- BETWEEN
- BIGINT
- BIGSERIAL
- BINARY
- BITAND

- BITANDNOT
- BITNOT
- BITOR
- BITXOR
- BLOB
- BLOBDIR
- BOOLEAN
- BOTH
- BOUND_IMPL_PDQ
- BUCKETS
- BUFFERED
- BUILTIN
- BY
- BYTE

**C**

- CACHE
- CALL
- CANNOTHASH
- CARDINALITY
- CASCADE
- CASE
- CAST
- CEIL
- CHAR
- CHAR_LENGTH
- CHARACTER
- CHARACTER_LENGTH
- CHARINDEX
- CHECK
- CHR
- CLASS
- CLASS_ORIGIN
- CLIENT
- CLOB
- CLOBDIR
- CLOSE
- CLUSTER
- CLUSTER_TXN_SCOPE
- COBOL
- CODESET
- COLLATION
- COLLECTION
- COLUMN
- COLUMNS
- COMMIT

- COMMITTED
- COMMUTATOR
- COMPONENT
- COMPONENTS
- COMPRESSED
- CONCAT
- CONCURRENT
- CONNECT
- CONNECTION
- CONNECTION_NAME
- CONNECT_BY_ISCYCLE
- CONNECT_BY_ISLEAF
- CONNECT_BY_ROOT
- CONST
- CONSTRAINT
- CONSTRAINTS
- CONSTRUCTOR
- CONTEXT
- CONTINUE
- COPY
- COS
- COSH
- COSTFUNC
- COUNT
- CRCOLS
- CREATE
- CROSS
- CUME_DIST
- CURRENT
- CURRENT_ROLE
- CURRENT_USER
- CURRVAL
- CURSOR
- CYCLE

**D**

- DATA
- DATABASE
- DATAFILES
- DATASKIP
- DATE
- DATETIME
- DAY
- DBA
- DBDATE
- DBINFO

- DBPASSWORD
- DBSA
- DBSERVERNAME
- DBSECADM
- DBSSO
- DEALLOCATE
- DEBUG
- DEBUGMODE
- DEBUG_ENV
- DEC
- DECIMAL
- DECLARE
- DECODE
- DECRYPT_BINARY
- DECRYPT_CHAR
- DEC_T
- DEFAULT
- DEFAULTESCCHAR
- DEFAULT_ROLE
- DEFAULT_USER
- DEFERRED
- DEFERRED_PREPARE
- DEFINE
- DEGREES
- DELAY
- DELETE
- DELETING
- DELIMITED
- DELIMITER
- DELUXE
- DENSERANK
- DENSE_RANK
- DESC
- DESCRIBE
- DESCRIPTOR
- DETACH
- DIAGNOSTICS
- DIRECTIVES
- DIRTY
- DISABLE
- DISABLED
- DISCARD
- DISCONNECT
- DISK
- DISTINCT
- DISTRIBUTEBINARY

- DISTRIBUTESREFERENCES
- DISTRIBUTIONS
- DOCUMENT
- DOMAIN
- DONOTDISTRIBUTE
- DORMANT
- DOUBLE
- DROP
- DTIME_T

**E**

- EACH
- ELIF
- ELSE
- ENABLE
- ENABLED
- ENCRYPT_AES
- ENCRYPT_TDES
- ENCRYPTION
- END
- ENUM
- ENVIRONMENT
- ERKEY
- ERROR
- ESCAPE
- EXCEPT
- EXCEPTION
- EXCLUSIVE
- EXEC
- EXECUTE
- EXECUTEANYWHERE
- EXEMPTION
- EXISTS
- EXIT
- EXP
- EXPLAIN
- EXPLICIT
- EXPRESS
- EXPRESSION
- EXTDIRECTIVES
- EXTEND
- EXTENT
- EXTERNAL
- EXTYPEID
- EXTYPELENGTH
- EXTYPENAME

- EXTYPEOWNERLENGTH
- EXTYPEOWNERNAME

**F**

- FACT
- FALSE
- FAR
- FETCH
- FILE
- FILETOBLOB
- FILETOCLOB
- FILLFACTOR
- FILTERING
- FINAL
- FIRST
- FIRST_ROWS
- FIRST_VALUE
- FIXCHAR
- FIXED
- FLOAT
- FLOOR
- FLUSH
- FOLLOWING
- FOR
- FORCE
- FORCED
- FORCE_DDL_EXEC
- FOREACH
- FOREIGN
- FORMAT
- FORMAT_UNITS
- FORTRAN
- FOUND
- FRACTION
- FRAGMENT
- FRAGMENTS
- FREE
- FROM
- FULL
- FUNCTION

**G**

- G
- GB
- GENERAL
- GET

- GETHINT
- GIB
- GLOBAL
- GO
- GOTO
- GRANT
- GREATERTHAN
- GREATERTHANOREQUAL
- GRID
- GRID_NODE_SKIP
- GROUP

**H**

- HANDLESNULLS
- HASH
- HAVING
- HDR
- HDR_TXN_SCOPE
- HEX
- HIGH
- HINT
- HOLD
- HOME
- HOUR

**I**

- IDATA
- IDSLBACREADARRAY
- IDSLBACREADSET
- IDSLBACREADTREE
- IDSLBACRULES
- IDSLBACWRITEARRAY
- IDSLBACWRITESET
- IDSLBACWRITETREE
- IDSSECURITYLABEL
- IF
- IFX_*
- ILENGTH
- IMMEDIATE
- IMPLICIT
- IMPLICIT_PDQ
- IN
- INACTIVE
- INCREMENT
- INDEX
- INDEXES

- INDEX_ALL
- INDEX_SJ
- INDICATOR
- GBASEDBT
- GBASEDBTCONRETRY
- GBASEDBTCONTIME
- INIT
- INITCAP
- INLINE
- INNER
- INOUT
- INSENSITIVE
- INSERT
- INSERTING
- INSTEAD
- INSTR
- INT
- INT8
- INTEG
- INTEGER
- INTERNAL
- INTERNALLENGTH
- INTERSECT
- INTERVAL
- INTO
- INTRVL_T
- IS
- ISCANONICAL
- ISOLATION
- ITEM
- ITERATOR
- ITYPE

**J - K**

- JAVA
- JOIN
- K
- KB
- KEEP
- KEY
- KIB

**L**

- LABEL
- LABELEQ
- LABELGE

- LABELGLB
- LABELGT
- LABELLE
- LABELLT
- LABELLUB
- LABELTOSTRING
- LAG
- LANGUAGE
- LAST
- LAST_DAY
- LAST_VALUE
- LATERAL
- LEAD
- LEADING
- LEFT
- LEN
- LENGTH
- LESSTHAN
- LESSTHANOREQUAL
- LET
- LEVEL
- LIKE
- LIMIT
- LIST
- LISTING
- LOAD
- LOCAL
- LOCATOR
- LOCK
- LOCKS
- LOCOPY
- LOC_T
- LOG
- LOG10
- LOGN
- LONG
- LOOP
- LOTOFILE
- LOW
- LOWER
- LPAD
- LTRIM
- LVARCHAR

**M**

- M

- MATCHED
- MATCHES
- MAX
- MAXERRORS
- MAXLEN
- MAXVALUE
- MB
- MDY
- MEDIAN
- MEDIUM
- MEMORY
- MEMORY_RESIDENT
- MERGE
- MESSAGE_LENGTH
- MESSAGE_TEXT
- MIB
- MIDDLE
- MIN
- MINUS
- MINUTE
- MINVALUE
- MOD
- MODE
- MODERATE
- MODIFY
- MODULE
- MONEY
- MONTH
- MONTHS_BETWEEN
- MORE
- MULTISET
- MULTI_INDEX

**N**

- NAME
- NCHAR
- NEAR_SYNC
- NEGATOR
- NEW
- NEXT
- NEXT_DAY
- NEXTVAL
- NLSCASE
- NO
- NOCACHE
- NOCYCLE

- NOMAXVALUE
- NOMIGRATE
- NOMINVALUE
- NONE
- NON_RESIDENT
- NON_DIM
- NOORDER
- NORMAL
- NOT
- NOTEMPLATEARG
- NOTEQUAL|
- NOVALIDATE
- NTILE
- NULL
- NULLABLE
- NULLIF
- NULLS
- NUMBER
- NUMERIC
- NUMROWS
- NUMTODSINTERVAL
- NUMTOYMINTERVAL
- NVARCHAR
- NVL

**O**

- OCTET_LENGTH
- OF
- OFF
- OLD
- ON
- ONLINE
- ONLY
- OPAQUE
- OPCLASS
- OPEN
- OPTCOMPIND
- OPTIMIZATION
- OPTION
- OR
- ORDER
- ORDERED
- OUT
- OUTER
- OUTPUT
- OVER

- OVERRIDE

**P**

- PAGE
- PARALLELIZABLE
- PARAMETER
- PARTITION
- PASCAL
- PASSEDBYVALUE
- PASSWORD
- PDQPRIORITY
- PERCALL_COST
- PERCENT_RANK
- PIPE
- PLI
- PLOAD
- POLICY
- POW
- POWER®
- PRECEDING
- PRECISION
- PREPARE
- PREVIOUS
- PRIMARY
- PRIOR
- PRIVATE
- PRIVILEGES
- PROCEDURE
- PROPERTIES
- PUBLIC
- PUT

**Q**

- QUARTER

**R**

- RADIANS
- RAISE
- RANGE
- RANK
- RATIOTOREPORT
- RATIO_TO_REPORT
- RAW
- READ
- REAL

- RECORDEND
- REFERENCES
- REFERENCING
- REGISTER
- REJECTFILE
- RELATIVE
- RELEASE
- REMAINDER
- RENAME
- REOPTIMIZATION
- REPEATABLE
- REPLACE
- REPLICATION
- RESOLUTION
- RESOURCE
- RESTART
- RESTRICT
- RESUME
- RETAIN®
- RETAINUPDATELOCKS
- RETURN
- RETURNED_SQLSTATE
- RETURNING
- RETURNS
- REUSE
- REVERSE
- REVOKE
- RIGHT
- ROBIN
- ROLE
- ROLLBACK
- ROLLFORWARD
- ROLLING
- ROOT
- ROUND
- ROUTINE
- ROW
- ROWID
- ROWIDS
- ROWNUMBER
- ROWS
- ROW_COUNT
- ROW_NUMBER
- RPAD
- RTRIM
- RULE

**S**

- SAMEAS
- SAMPLES
- SAMPLING
- SAVE
- SAVEPOINT
- SCHEMA
- SCALE
- SCROLL
- SECLABEL_BY_COMP
- SECLABEL_BY_NAME
- SECLABEL_TO_CHAR
- SECOND
- SECONDARY
- SECURED
- SECURITY
- SECTION
- SELCONST
- SELECT
- SELECTING
- SELECT_GRID
- SELECT_GRID_ALL
- SELFUNC
- SELFUNCARGS
- SENSITIVE
- SEQUENCE
- SERIAL
- SERIAL8
- SERIALIZABLE
- SERVER
- SERVER_NAME
- SERVERUUID
- SESSION
- SET
- SETSESSIONAUTH
- SHARE
- SHORT
- SIBLINGS
- SIGNED
- SIN
- SITENAME
- SIZE
- SKIP
- SMALLFLOAT
- SMALLINT

- SOME
- SOURCEID
- SOURCETYPE
- SPACE
- SPECIFIC
- SQL
- SQLCODE
- SQLCONTEXT
- SQLERROR
- SQLSTATE
- SQLWARNING
- SQRT
- STABILITY
- STACK
- STANDARD
- START
- STAR_JOIN
- STATCHANGE
- STATEMENT
- STATIC
- STATISTICS
- STATLEVEL
- STATUS
- STDDEV
- STEP
- STOP
- STORAGE
- STORE
- STRATEGIES
- STRING
- STRINGTOLABEL
- STRUCT
- STYLE
- SUBCLASS_ORIGIN
- SUBSTR
- SUBSTRING
- SUBSTRING_INDEX
- SUM
- SUPPORT
- SYNC
- SYNONYM
- SYS*

**T**

- T
- TABLE

- TABLES
- TAN
- TASK
- TB
- TEMP
- TEMPLATE
- TEST
- TEXT
- THEN
- TIB
- TIME
- TO
- TODAY
- TO_CHAR
- TO_DATE
- TO_DSINTERVAL
- TO_NUMBER
- TO_YMINTERVAL
- TRACE
- TRAILING
- TRANSACTION
- TRANSITION
- TREE
- TRIGGER
- TRIGGERS
- TRIM
- TRUE
- TRUNC
- TRUNCATE
- TRUSTED
- TYPE
- TYPEDEF
- TYPEID
- TYPENAME
- TYPEOF

**U**

- UID
- UNBOUNDED
- UNCOMMITTED
- UNDER
- UNION
- UNIQUE
- UNITS
- UNKNOWN
- UNLOAD

- UNLOCK
- UNSIGNED
- UPDATE
- UPDATING
- UPON
- UPPER
- USAGE
- USE
- USELASTCOMMITTED
- USER
- USE_HASH
- USE_NL
- USING
- USTLOW_SAMPLE

**V**

- VALUE
- VALUES
- VAR
- VARCHAR
- VARIABLE
- VARIANCE
- VARIANT
- VARYING
- VERCOLS
- VIEW
- VIOLATIONS
- VOID
- VOLATILE

**W - Z**

- WAIT
- WARNING
- WEEKDAY
- WHEN
- WHENEVER
- WHERE
- WHILE
- WITH
- WITHOUT
- WORK
- WRITE
- WRITEDOWN
- WRITEUP
- XADATASOURCE
- XID

- XLOAD
- XUNLOAD
- YEAR

GBASE®

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微信二维码

技术支持热线：400-013-9696