GBASE

GBase 8s 应用开发指南

www.gbase.cn

GBase 8s应用开发指南，南大通用数据技术股份有限公司

通讯方式

南大通用数据技术股份有限公司

天津市高新区华苑产业园区工华道2号天百中心3层(300384)

电话：400-013-9696　　邮箱：info@gbase.cn

目　录

1 数据查询请求处理过程

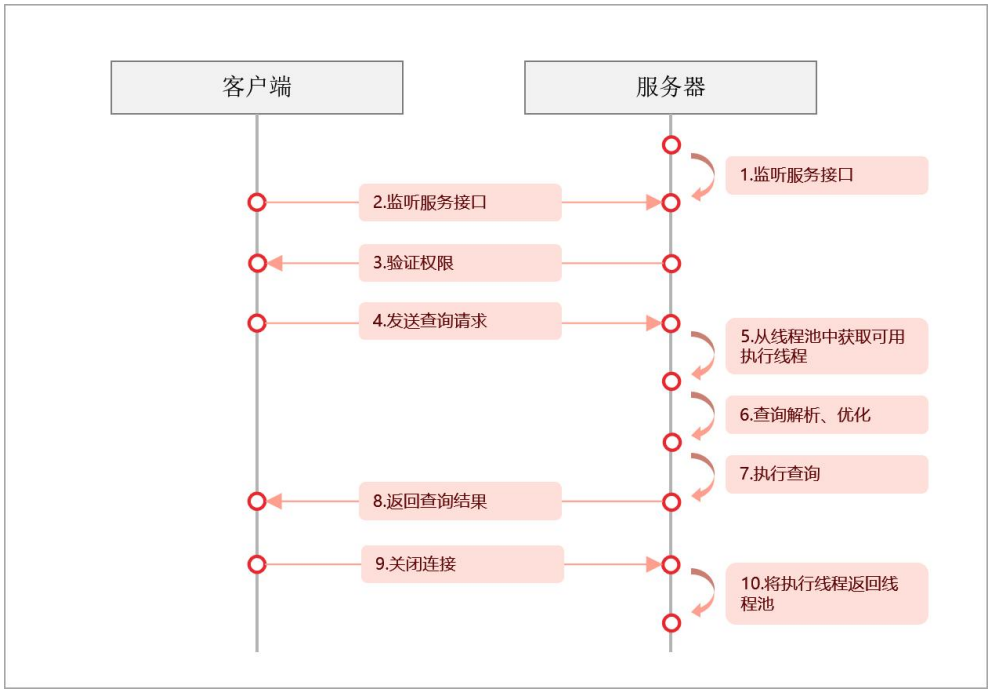

图 1-1 服务响应流程

2 开发设计规范

2.1 开发设计规范概述

本开发设计建议约定数据库建模和数据库应用程序开发过程中，应当遵守的设计规范。依据这些规范进行建模，能够更好地契合 GBase 8s 处理架构，输出更高效的业务 SQL 代码。

本开发设计建议中所陈述的“建议”和“关注”含义如下：

- **建议**：用户应当遵守的设计规则。遵守这些规则，能够保证业务的高效运行；违反这些规则，将导致业务性能的大幅下降或某些业务逻辑错误。
- **关注**：在业务开发过程中客户需要注意的细则。用于标识容易导致客户理解错误的知识点（实际上遵守 SQL 标准的 SQL 行为），或者程序中潜在的客户不易感知的默认行为。

2.2 数据库对象命名

数据库对象命名需要满足约束：非时序表长度不超过 63 个字节，时序表长度不超过 53 个字符，以字母或下划线开头，中间字符可以是字母、数字、下划线、$、#。

- 【建议】避免使用保留或者非保留关键字命名数据库对象。

  说明

  可以使用 select * from pg_get_keywords()查询数据库关键字，或者在《GBase 8s V8.8.5_5.0.0_SQL 参考手册》关键字章节中查看。

- 【建议】避免使用双引号括起来的字符串来定义数据库对象名称，除非需要限制数据库对象名称的大小写。数据库对象名称大小写敏感会使定位问题难度增加。
- 【建议】数据库对象命名风格务必保持统一。
  - 增量开发的业务系统或进行业务迁移的系统，建议遵守历史的命名风格。
  - 建议使用多个单词组成，以下划线分割。
  - 数据库对象名称建议能够望文知意，尽量避免使用自定义缩写（可以使用通用的术语缩写进行命名）。例如，在命名中可以使用具有实际业务含义的英文词汇或汉语拼音，但规则应该在数据库实例范围内保持一致。
  - 变量名的关键是要具有描述性，即变量名称要有一定的意义，变量名要有前缀标明该变量的类型。

- 【建议】表对象的命名应该可以表征该表的重要特征。例如，在表对象命名时区分该表是普通表、临时表还是非日志表：
  - 普通表名按照数据集的业务含义命名。
  - 临时表以“tmp_+后缀”命名。
  - 非日志表以“ul_+后缀”命名。
  - 外表以“f_+后缀”命名。
  - 不创建以 redis_为前缀的数据库对象。
  - 不创建以 mlog_和以 matviewmap_为前缀的数据库对象。
- 【建议】非时序表对象命名建议不要超过 63 字节。如果过该长度内核会对表名进行截断，从而造成和设置值不一致的现象。且在不同字符集下，可能造成字符被截断，出现预期外的字符。

2.3 数据库对象设计

2.3.1 Database 和 Schema 设计

GBase 8s 中可以使用 Database 和 Schema 实现业务的隔离，区别在于 Database 的隔离更加彻底，各个 Database 之间共享资源极少，可实现连接隔离、权限隔离等， Database 之间无法直接互访。Schema 隔离的方式共用资源较多，可以通过 grant 与 revoke 语法便捷地控制不同用户对各 Schema 及其下属对象的权限。

- 从便捷性和资源共享效率上考虑，推荐使用 Schema 进行业务隔离。
- 建议系统管理员创建 Schema 和 Database，再赋予相关用户对应的权限。

Database 设计建议

- 【规则】在实际业务中，根据需要创建新的 Database，不建议直接使用数据库实例默认的 postgres 数据库。
- 【建议】一个数据库实例内，用户自定义的 Database 数量建议不超过 3 个。
- 【建议】为了适应全球化的需求，使数据库编码能够存储与表示绝大多数的字符，建议创建 Database 的时候使用 UTF-8 编码。
- 【关注】创建 Database 时，需要重点关注字符集编码(ENCODING)和兼容性

(DBCOMPATIBILITY)两个配置项。GBase 8s 支持 A、B、C 和 PG 四种兼容模式，分别表示兼容 O 语法、MY 语法、TD 语法和 POSTGRES 语法，不同兼容模式下的语法行为存在一定差异，默认为 A 兼容模式。

- 【关注】Database 的 owner 默认拥有该 Database 下所有对象的所有权限，包括删除权限。删除权限影响较大，请谨慎使用。

Schema 设计建议

- 【关注】如果该用户不具有 sysadmin 权限或者不是该 Schema 的 owner，要访问 Schema 下的对象，需要同时给用户赋予 Schema 的 usage 权限和对象的相应权 限。
- 【关注】如果要在 Schema 下创建对象，需要授予操作用户该 Schema 的 create 权限。
- 【关注】Schema 的 owner 默认拥有该 Schema 下对象的所有权限，包括删除权限。删除权限影响较大，请谨慎使用。

2.3.2 表设计

总体上讲，良好的表设计需要遵循以下原则：

- 【关注】减少需要扫描的数据量。通过分区表的剪枝机制可以大幅减少数据的扫描量。
- 【关注】尽量减少随机 I/O。通过聚簇/局部聚簇可以实现热数据的连续存储，将 随机 I/O 转换为连续 I/O，从而减少扫描的 I/O 代价。

选择存储方案

【建议】表的存储类型是表定义设计的第一步，客户业务类型是决定表的存储类型的主要因素，表存储类型的选择依据请参考下表。

表 2-1 表的存储类型及场景

存储类型	适用场景
行存	● 点查询（返回记录少，基于索引的简单查询）。 ● 增、删、改操作较多的场景。
列存	● 统计分析类查询 （关联、分组操作较多的场景）。 ● 即席查询（查询条件不确定，行存表扫描难以使用索引）。

选择分区方案

当表中的数据量很大时，应当对表进行分区，一般需要遵循以下原则：

- 【建议】使用具有明显区间性的字段进行分区，比如日期、区域等字段上建立分区。
- 【建议】分区名称应当体现分区的数据特征。例如，关键字+区间特征。
- 【建议】将分区上边界的分区值定义为 MAXVALUE，以防止可能出现的数据溢出。

表 2-2 表的分区方式及使用场景

分区方式	描述
Range	表数据通过范围进行分区。
Interval	表数据通过范围进行分区，超出范围的会自动根据间隔创建新的分区。
List	表数据通过指定列按照具体值进行分区。
Hash	表数据通过 Hash 散列方式进行分区。

典型的分区表定义如下：

```
--创建 Range 分区表
CREATE TABLE staffS_p1 (
staff_ID   NUMBER(6) not null, FIRST_NAME          VARCHAR2(20), LAST_NAME
           VARCHAR2(25), EMAIL     VARCHAR2(25), PHONE_NUMBER VARCHAR2(20),
HIRE_DATE          DATE,
employment_ID VARCHAR2(10), SALARY NUMBER(8,2), COMMISSION_PCT NUMBER(4,2),
MANAGER_ID NUMBER(6),
section_ID NUMBER(4)
)
PARTITION BY RANGE (HIRE_DATE) (
PARTITION HIRE_19950501 VALUES LESS THAN ('1995-05-01 00:00:00'), PARTITION
HIRE_19950502 VALUES LESS THAN ('1995-05-02 00:00:00'), PARTITION HIRE_maxvalue
VALUES LESS THAN (MAXVALUE)
);
--创建 Interval 分区表，初始两个分区，插入分区范围外的数据会自动新增分区 CREATE
TABLE sales
(prod_id NUMBER(6), cust_id NUMBER, time_id DATE, channel_id CHAR(1), promo_id
NUMBER(6),
quantity_sold NUMBER(3), amount_sold NUMBER(10,2)
)
PARTITION BY RANGE (time_id) INTERVAL('1 day')
```

```
( PARTITION p1 VALUES LESS THAN ('2019-02-01 00:00:00'), PARTITION p2 VALUES LESS
THAN ('2019-02-02 00:00:00')
);
--创建 List 分区表
CREATE TABLE test_list (col1 int, col2 int) partition by list(col1)
(
partition p1 values (2000), partition p2 values (3000), partition p3 values (4000),
partition p4 values (5000)
);
--创建 Hash 分区表
CREATE TABLE test_hash (col1 int, col2 int) partition by hash(col1)
(
partition p1, partition p2
);
```

更多的表分区语法信息，参见《GBase 8s V8.8.5_5.0.0_SQL 参考手册》中 CREATE TABLE PARTITION 章节。

2.3.3 规划存储模型

GBase 8s 支持行列混合存储。行、列存储模型各有优劣，建议根据实际情况选择。通常 GBase 8s 用于 TP 场景的数据库，默认使用行存储，仅对执行复杂查询且数据量大的 AP 场景时，才使用列存储。

行存储是指将表按行存储到硬盘分区上，列存储是指将表按列存储到硬盘分区上。默认情况下，创建的表为行存储。行存储和列存储的差异请参见下图。

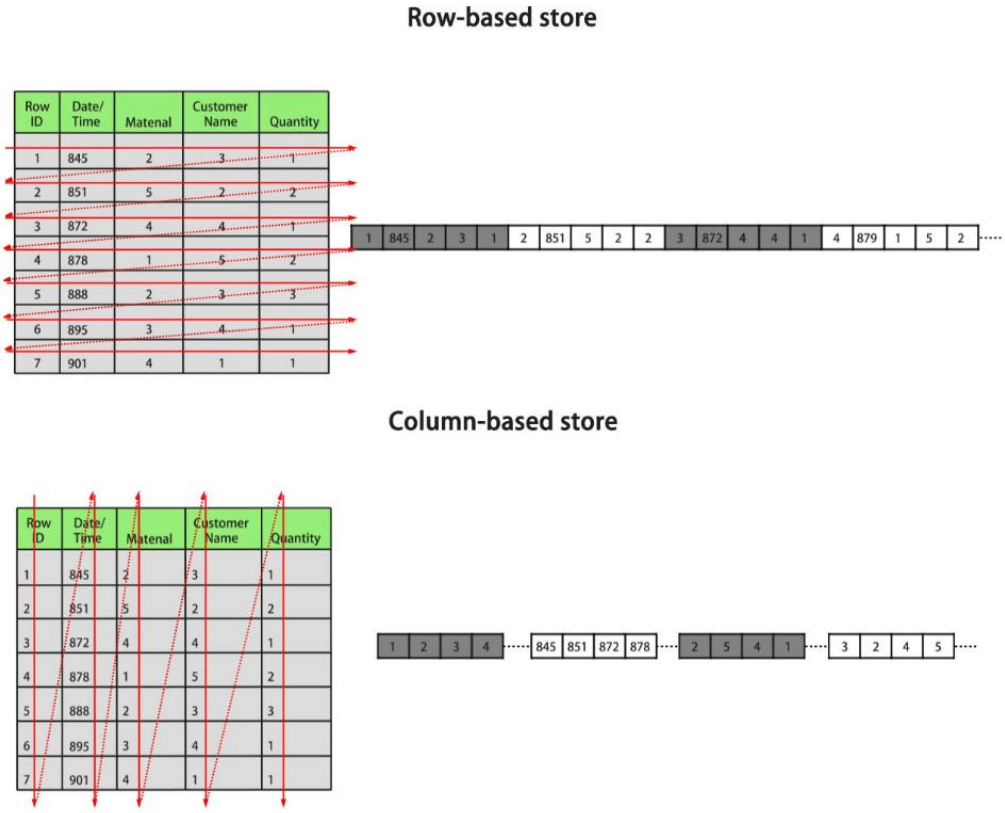

图 2-1 行存储和列存储的差异

上图中，左上为行存表，右上为行存表在硬盘上的存储方式。左下为列存表，右下为列存表在硬盘上的存储方式。

行、列存储有如下优缺点：

存储模型	优点	缺点
行存	数据被保存在一起。INSERT/UPDATE 容易。	选择（Selection）时即使只涉及某几列，所有数据也都会被读取。
列存	查询时只有涉及到的列会被读取。 投影（Projection）很高效。 任何列都能作为索引。	选择完成时，被选择的列要重新组装。 INSERT/UPDATE 比较麻烦。

一般情况下，如果表的字段比较多（大宽表），查询中涉及到的列不多的情况下，适合列存储。如果表的字段个数比较少，查询大部分字段，那么选择行存储比较好。

存储类型	适用场景
行存	点查询（返回记录少，基于索引的简单查询）。 增、删、改操作较多的场景。
列存	统计分析类查询（关联、分组操作较多的场景）。 即席查询（查询条件不确定，行存表扫描难以使用索引）。

行存表

默认创建表的类型。数据按行进行存储，即一行数据是连续存储。适用于对数据需要经常更新的场景。

```
postgres=#CREATE TABLE customer_t1
(
   state_ID      CHAR(2),
   state_NAME VARCHAR2(40),
   area_ID       NUMBER
);

--删除表
postgres=#DROP TABLE customer_t1;
```

列存表

数据按列进行存储，即一列所有数据是连续存储的。单列查询 IO 小，比行存表占用更少的存储空间。适合数据批量插入、更新较少和以查询为主统计分析类的场景。列存表不适合点查询。

```
postgres=#CREATE TABLE customer_t2
(
   state_ID      CHAR(2),
   state_NAME VARCHAR2(40),
   area_ID       NUMBER
)
WITH (ORIENTATION = COLUMN);

--删除表
postgres=#DROP TABLE customer_t2;
```

行存表和列存表的选择

- 更新频繁程度

  数据如果频繁更新，选择行存表。

- 插入频繁程度

  频繁的少量插入，选择行存表。一次插入大批量数据，选择列存表。

- 表的列数

  表的列数很多，选择列存表。

- 查询的列数

  如果每次查询时，只涉及了表的少数（<50%总列数）几个列，选择列存表。

- 压缩率

  列存表比行存表压缩率高。但高压缩率会消耗更多的 CPU 资源。

2.3.4 字段设计

选择数据类型

在字段设计时，基于查询效率的考虑，一般遵循以下原则：

- 【建议】尽量使用高效数据类型。
- 选择数值类型时，在满足业务精度的情况下，选择数据类型的优先级从高到低依次为整数、浮点数、NUMERIC。
- 【建议】当多个表存在逻辑关系时，表示同一含义的字段应该使用相同的数据类型。
- 【建议】对于字符串数据，建议使用变长字符串数据类型，并指定最大长度。请务必确保指定的最大长度大于需要存储的最大字符数，避免超出最大长度时出现字符截断现象。除非明确知道数据类型为固定长度字符串，否则，不建议使用 CHAR(n)、BPCHAR(n)、NCHAR(n)、CHARACTER(n)。

常用字符串类型介绍

在进行字段设计时，需要根据数据特征选择相应的数据类型。字符串类型在使用时比较容易混淆，下表列出了支持的常见字符串类型：

表 2-3 常用字符串类型

名称	描述	最大存储空间
CHAR(n)	定长字符串，n 描述了存储的字节长度，如果输入的字符串字节格式小于 n，那么后面会自动用空字符补齐至 n 个字节。	10MB
CHARACTER(n)	定长字符串，n 描述了存储的字节长度，如果输入的字符串字节格式小于 n，那么后面会自动用空字符补齐至 n 个字节。	10MB
NCHAR(n)	定长字符串，n 描述了存储的字节长度，如果输入的字符串字节格式小于 n，那么后面会自动用空字符补齐至 n 个字节。	10MB
BPCHAR(n)	定长字符串，n 描述了存储的字节长度，如果输入的字符串字节格式小于 n，那么后面会自动用空字符补齐至 n 个字节。	10MB
VARCHAR(n)	变长字符串，n 描述了可以存储的最大字节长度。	10MB
CHARACTER VARYING(n)	变长字符串，n 描述了可以存储的最大字节长度；此数据类型和 VARCHAR(n)是同一数据类型的不同表达形式。	10MB
VARCHAR2(n)	变长字符串，n 描述了可以存储的最大字节长度，此数据类型是为兼容 Oracle 类型新增的，行为和 VARCHAR(n)一致。	10MB
NVARCHAR2(n)	变长字符串，n 描述了可以存储的最大字节长度。	10MB
TEXT	不限长度(不超过 1GB-8203 字节)变长字符串。	1GB-8203 字节

2.3.5 约束设计

DEFAULT 和 NULL 约束

- 【建议】如果能够从业务层面补全字段值，那么，就不建议使用 DEFAULT 约束， 避

免数据加载时产生不符合预期的结果。

- 【建议】给明确不存在 NULL 值的字段加上 NOT NULL 约束，优化器会在特定场景下对其进行自动优化。
- 【建议】给可以显式命名的约束显式命名。除了 NOT NULL 和 DEFAULT 约束外， 其他约束都可以显式命名。

局部聚簇

Partial Cluster Key（局部聚簇，简称 PCK）是列存表的一种局部聚簇技术，在 GBase 8s 中，使用 PCK 可以通过 min/max 稀疏索引实现事实表快速过滤扫描。PCK 的选取遵循以下原则：

- 【关注】一张表上只能建立一个 PCK，一个 PCK 可以包含多列，但是一般不建议超过 2 列。
- 【建议】在查询中的简单表达式过滤条件上创建 PCK。这种过滤条件一般形如 col op const，其中 col 为列名，op 为操作符=、>、>=、<=、<，const 为常量值。
- 【建议】在满足上面条件的前提下，选择 distinct 值比较多的列上建 PCK。

唯一约束

- 【关注】行存表、列存表均支持唯一约束。
- 【建议】从命名上明确标识唯一约束，例如，命名为“UNI+构成字段”。

主键约束

- 【关注】行存表、列存表均支持主键约束。
- 【建议】从命名上明确标识主键约束，例如，将主键约束命名为“PK+字段名”。

检查约束

- 【关注】行存表支持检查约束，而列存表不支持。
- 【建议】从命名上明确标识检查约束，例如，将检查约束命名为“CK+字段名”。

2.3.6 视图和关联表设计

视图设计

- 【建议】除非视图之间存在强依赖关系，否则不建议视图嵌套。

- 【建议】视图定义中尽量避免排序操作。

关联表设计

- 【建议】表之间的关联字段应该尽量少。
- 【建议】关联字段的数据类型应该保持一致。
- 【建议】关联字段在命名上，应该可以明显体现出关联关系。例如，采用同样名称来命名。

2.4 SQL 编写

DDL

- 【建议】在 GBase 8s 中，建议 DDL（建表、comments 等）操作统一执行，在批处理作业中尽量避免 DDL 操作。避免大量并发事务对性能的影响。
- 【建议】在非日志表（unlogged table）使用完后，立即执行数据清理（truncate）操作。因为在异常场景下，GBase 8s 不保证非日志表（unlogged table）数据的安全性。
- 【建议】临时表和非日志表的存储方式建议和基表相同。当基表为行存（列存）表时，临时表和非日志表也推荐创建为行存（列存）表，可以避免行列混合关联带来的高计算代价。
- 【建议】索引字段的总长度不超过 50 字节。否则，索引大小会膨胀比较严重，带来较大的存储开销，同时索引性能也会下降。
- 【建议】不要使用 DROP…CASCADE 方式删除对象，除非已经明确对象间的依赖关系，以免误删。

数据加载和卸载

- 【建议】在 insert 语句中显式给出插入的字段列表。例如：

```
INSERT INTO task(name,id,comment) VALUES ('task1','100','第 100 个任务');
```

- 【建议】在批量数据入库之后，或者数据增量达到一定阈值后，建议对表进行 analyze 操作，防止统计信息不准确而导致的执行计划劣化。
- 【建议】如果要清理表中的所有数据，建议使用 truncate table 方式，不要使用 delete table 方式。delete table 方式删除性能差，且不会释放那些已经删除了的数据占用的磁盘空间。

类型转换

- 【建议】在需要数据类型转换（不同数据类型进行比较或转换）时，使用强制类型转换，以防隐式类型转换结果与预期不符。
- 【建议】在查询中，对常量要显式指定数据类型，不要试图依赖任何隐式的数据类型转换。
- 【关注】若 sql_compatibility 参数设置为 A，在导入数据时，空字符串会自动转化为 NULL。如果需要保留空字符串需要 sql_compatibility 参数设置为 C。

查询操作

- 【建议】除 ETL 程序外，应该尽量避免向客户端返回大量结果集的操作。如果结果集过大，应考虑业务设计是否合理。
- 【建议】使用事务方式执行 DDL 和 DML 操作。例如，truncate table、update table、delete table、drop table 等操作，一旦执行提交就无法恢复。对于这类操作，建议使用事务进行封装，必要时可以进行回滚。
- 【建议】在查询编写时，建议明确列出查询涉及的所有字段，不建议使用“SELECT *”这种写法。一方面基于性能考虑，尽量减少查询输出列；另一方面避免增删字段对前端业务兼容性的影响。
- 【建议】在访问表对象时带上 schema 前缀，可以避免因 schema 切换导致访问到非预期的表。
- 【建议】超过 3 张表或视图进行关联（特别是 FULL JOIN）时，执行代价难以估算。建议使用 WITH TABLE AS 语句创建中间临时表的方式增加 SQL 语句的可读性。
- 【建议】尽量避免使用笛卡尔积和 FULL JOIN。这些操作会造成结果集的急剧膨胀，同时其执行性能也很低。
- 【关注】NULL 值的比较只能使用 IS NULL 或者 IS NOT NULL 的方式判断，其他任何形式的逻辑判断都返回 NULL。例如：NULL<>NULL、NULL=NULL 和 NULL<>1 返回结果都是 NULL，而不是期望的布尔值。
- 【关注】需要统计表中所有记录数时，不要使用 count(col)来替代 count(*)。count(*)会统计 NULL 值（真实行数），而 count(col)不会统计。
- 【关注】在执行 count(col)时，将“值为 NULL”的记录行计数为 0。在执行 sum(col)时，当所有记录都为 NULL 时，最终将返回 NULL；当不全为 NULL 时，“值为 NULL”的记录行将被计数为 0。

- 【关注】count(多个字段)时，多个字段名必须用圆括号括起来。例如，count((col1,col2,col3))。注意：通过多字段统计行数时，即使所选字段都为 NULL，该行也被计数，效果与 count(*)一致。
- 【关注】count(distinct col)用来计算该列不重复的非 NULL 的数量，NULL 将不被计数。
- 【关注】count(distinct (col1,col2,…))用来统计多列的唯一值数量，当所有统计字段都为 NULL 时，也会被计数，同时这些记录被认为是相同的。
- 【建议】使用连接操作符“||”替换 concat 函数进行字符串连接。因为 concat 函数生成的执行计划不能下推，导致查询性能严重劣化。
- 【建议】使用下面时间相关的宏替换 now 函数来获取当前时间。因为 now 函数生成的执行计划无法下推，导致查询性能严重劣化。

表 2-4 时间相关的宏

宏名称	描述	示例
CURRENT_DATE	获取当前日期，不包含时分秒。	postgres=#select CURRENT_DATE; date \------------ 2018-02-02 (1 row)
CURRENT_TIME	获取当前时间，不包含年月日。	postgres=#select CURRENT_TIME; timetz \-------------------- 00:39:34.633938+08 (1 row)
CURRENT_TIMESTAMP(n)	获取当前日期和时间，包含年月日时分秒。 说明： n 表示存储的毫秒位数。	postgres=#select CURRENT_TIMESTAMP(6); timestamptz \------------------------------- 2018-02-02 00:39:55.231689+08 (1 row)

- 【建议】尽量避免标量子查询语句的出现。标量子查询是出现在 select 语句输出列表中的子查询，在下面例子中，下划线部分即为一个标量子查询语句：

```
SELECT id, (SELECT COUNT(*) FROM films f WHERE f.did = s.id) FROM staffs_p1 s;
```

标量子查询往往会导致查询性能急剧劣化，在应用开发过程中，应当根据业务逻辑，对标量子查询进行等价转换，将其写为表关联。

- 【建议】在 where 子句中，应当对过滤条件进行排序，把选择读较小（筛选出的记录数较少）的条件排在前面。
- 【建议】where 子句中的过滤条件，尽量符合单边规则。即把字段名放在比较条件的一边，优化器在某些场景下会自动进行剪枝优化。形如 col op expression，其中 col 为表的一个列，op 为‘=’、‘>’的等比较操作符，expression 为不含列名的表达式。例如，

```
SELECT id, from_image_id, from_person_id, from_video_id FROM face_data WHERE
current_timestamp(6) - time < '1 days'::interval;
```

改写为：

```
SELECT id, from_image_id, from_person_id, from_video_id FROM face_data where time >
current_timestamp(6) - '1 days'::interval;
```

- 【建议】尽量避免不必要的排序操作。排序需要耗费大量的内存及 CPU，如果业务逻辑许可，可以组合使用 ORDER BY 和 LIMIT，减小资源开销。GBase 8s 默认按照 ASC & NULL LAST 进行排序。
- 【建议】使用 ORDER BY 子句进行排序时，显式指定排序方式（ASC/DESC），NULL 的排序方式（NULL FIRST/NULL LAST）。
- 【建议】不要单独依赖 limit 子句返回特定顺序的结果集。如果部分特定结果集，可以将 ORDER BY 子句与 Limit 子句组合使用，必要时也可以使用 OFFSET 跳过特定结果。
- 【建议】在保障业务逻辑准确的情况下，建议尽量使用 UNION ALL 来代替 UNION。
- 【建议】如果过滤条件只有 OR 表达式，可以将 OR 表达式转化为 UNION ALL 以提升性能。使用 OR 的 SQL 语句经常无法优化，导致执行速度慢。例如，将下面语句

```
SELECT * FROM scdc.pub_menu
WHERE (cdp= 300 AND inline=301) OR (cdp= 301 AND inline=302) OR (cdp= 302
ANDinline=301);
```

转换为：

```
SELECT * FROM scdc.pub_menu    WHERE (cdp= 300 AND inline=301)    union all
SELECT * FROM scdc.pub_menu    WHERE (cdp= 301 AND inline=302)    union all
SELECT * FROM tablename    WHERE (cdp= 302 AND inline=301);
```

- 【建议】当 in(val1, val2, val3…)表达式中字段较多时，建议使用 in (values (val1),

(val2),(val3)…)语句进行替换。优化器会自动把 in 约束转换为非关联子查询，从而提升查询性能。

- 【建议】在关联字段不存在 NULL 值的情况下，使用(not) exist 代替(not) in。例如，在下面查询语句中，当 T1.C1 列不存在 NULL 值时，可以先为 T1.C1 字段添加 NOT NULL 约束，再进行如下改写。

```
SELECT * FROM T1 WHERE T1.C1 NOT IN (SELECT T2.C2 FROM T2);
```

可以改写为：

```
SELECT * FROM T1 WHERE NOT EXISTS (SELECT  * FROM T2 WHERE
T1.C1=T2.C2);
```

 说明：

- 如果不能保证 T1.C1 列的值为 NOT NULL 的情况下，就不能进行上述改写。
- 如果 T1.C1 为子查询的输出，要根据业务逻辑确认其输出是否为 NOT NULL。

- 【建议】通过游标进行翻页查询，而不是使用 LIMIT OFFSET 语法，避免多次执行带来的资源开销。游标必须在事务中使用，执行完后务必关闭游标并提交事务。

2.5 程序开发规范

2.5.1 开发规范

如果用户在 APP 的开发中，使用了连接池机制，那么需要遵循如下规范：

- 如果在连接中设置了 GUC 参数，那么在将连接归还连接池之前，必须使用“SET SESSION AUTHORIZATION DEFAULT;RESET ALL;”将连接的状态清空。
- 如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。

否则，连接池里面的连接就是有状态的，会对用户后续使用连接池进行操作的正确性带来影响。

兼容性原则：

- 新驱动前向兼容数据库，若需使用驱动与数据库同步增加的新特性，必须升级数据库。

2.5.2 工具对接

2.5.2.1 JDBC 配置

目前，GBase 8s 相关的第三方工具都是通过 JDBC 进行连接的，此部分将介绍工具配置时的注意事项。

连接参数

- 【关注】第三方工具通过 JDBC 连接 GBase 8s 时，JDBC 向 GBase 8s 发起连接请求，会默认添加以下配置参数，详见 JDBC 代码 ConnectionFactoryImpl 类的实现。

```
params = {
{ "user", user },
{ "database", database },
{ "client_encoding", "UTF8" },
{ "DateStyle", "ISO" },
{ "extra_float_digits", "3" },
{ "TimeZone", createPostgresTimeZone() },
};
```

这些参数可能会导致 JDBC 客户端的行为与 gsql 客户端的行为不一致，例如，Date 数据显示方式、浮点数精度表示、timezone 显示。

如果实际期望和这些配置不符，建议在 java 连接设置代码中显式设定这些参数。

- 【建议】通过 JDBC 连接数据库时，应该保证下面三个时区设置一致：
  - JDBC 客户端所在主机的时区。
  - 数据库实例所在主机的时区。
  - 数据库实例配置过程中时区。

  说明

  时区设置相关的操作，请参考《GBase 8s V8.8.5_5.0.0_安装部署手册》中“同步系统时间”章节内容。

fetchsize

- 【关注】在应用程序中，如果需要使用 fetchsize，必须关闭 autocommit。开启 autocommit，会令 fetchsize 配置失效。

autocommit

- 【建议】在 JDBC 向 GBase 8s 申请连接的代码中，建议显式打开 autocommit 开关。如果基于性能或者其它方面考虑，需要关闭 autocommit 时，需要应用程序自己来保证事务的提交。例如，在指定的业务 SQL 执行完之后做显式提交，特别是客户端退出之前 务必保证所有的事务已经提交。

释放连接

- 【建议】推荐使用连接池限制应用程序的连接数。每执行一条 SQL 就连接一次数据库，是一种不好 SQL 的编写习惯。
- 【建议】在应用程序完成作业任务之后，应当及时断开和 GBase 8s 的连接，释放资源。建议在任务中设置 session 超时时间参数。
- 【建议】使用 JDBC 连接池，在将连接释放给连接池前，需要执行以下操作，重置会话环境。否则，可能会因为历史会话信息导致的对象冲突。

  如果在连接中设置了 GUC 参数，那么在将连接归还连接池之前，必须使用“SET SESSION AUTHORIZATION DEFAULT;RESET ALL;”将连接的状态清空。

  如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。

CopyManager

- 【建议】在不使用 ETL 工具，数据入库实时性要求又比较高的情况下，建议在开发应用程序时，使用 GBase 8s JDBC 驱动的 copyManger 接口进行微批导入。

3 基于 JDBC 开发

JDBC（Java Database Connectivity，Java 数据库连接）是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问接口，应用程序可基于它操作数据。GBase 8s 数据库提供了对 JDBC 4.0 特性的支持，需要使用 JDK1.8 版本编译程序代码，不支持 JDBC 桥接 ODBC 方式。

3.1 JDBC 包、驱动类和环境类

JDBC 包

在 linux 服务器端源代码目录下执行 build.sh，获得驱动 jar 包 postgresql.jar。

驱动包与 PostgreSQL 保持兼容，其中类名、类结构与 PostgreSQL 驱动完全一致，曾经运行于 PostgreSQL 的应用程序可以直接移植到当前系统使用。

驱动类

在创建数据库连接之前，需要加载数据库驱动类“org.postgresql.Driver”。

由于 GBase 8s 在 JDBC 的使用上与 PG 的使用方法保持兼容，所以同时在同一进程内使用两个 JDBC 驱动的时候，可能会类名冲突。

相比于 PG 驱动，GBase 8s 的 JDBC 驱动主要做了以下特性的增强：

(1) 支持 SHA256 加密方式登录。

(2) 支持对接实现 sf4j 接口的第三方日志框架。

(3) 支持容灾切换。

环境类

客户端需配置 JDK1.8，配置方法如下：

(1) DOS 窗口输入“java -version”，查看 JDK 版本，确认为 JDK1.8 版本。如果未安装 JDK，请从官方网站下载安装包并安装。

(2) 根据如下步骤配置系统环境变量。

① 右键单击“我的电脑”，选择“属性”。

② 在“系统”页面左侧导航栏单击“高级系统设置”。

③ 在“系统属性”页面，“高级”页签上单击“环境变量”。

在“环境变量”页面上，“系统变量”区域单击“新建”或“编辑”配置系统变量。变量说明请参见下表。

表 3-1 变量说明

变量名	操作	变量值
JAVA_HO ME	若存在，则单击“编辑”。 若不存在，则单击“新建”。	JAVA 的安装目录。 例如：C:\Program Files\Java\jdk1.8.0_131
Path	编辑	若配置了 JAVA_HOME，则在变量值的最前面加上： %JAVA_HOME%\bin; 若未配置 JAVA_HOME，则在变量值的最前面加上 JAVA 安装的全路径： C:\Program Files\Java\jdk1.8.0_131\bin;
CLASSPAT H	新建	.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar;

3.2 开发流程

图 3-1 采用 JDBC 开发应用程序的流程

3.3 加载驱动

在创建数据库连接之前，需要先加载数据库驱动程序。

加载驱动有两种方法：

- 在代码中创建连接之前任意位置隐含装载：Class.forName("org.postgresql.Driver");
- 在 JVM 启动时参数传递：java -Djdbc.drivers=org.postgresql.Driver jdbctest

说明

上述 jdbctest 为测试用例程序的名称。

3.4 连接数据库

在创建数据库连接之后，才能使用它来执行 SQL 语句操作数据。

函数原型

JDBC 提供了三个方法，用于创建数据库连接。

- DriverManager.getConnection(String url);
- DriverManager.getConnection(String url, Properties info);
- DriverManager.getConnection(String url, String user, String password);

参数

表 3-2 数据库连接参数

参数	描述
info	数据库连接属性（所有属性大小写敏感）。常用的属性如下： ● PGDBNAME：String 类型。表示数据库名称。（URL 中无需配置该参数，自动从 URL 中解析） ● PGHOST：String 类型。主机 IP 地址。详细示例见下。 ● PGPORT：Integer 类型。主机端口号。详细示例见下。 ● user：String 类型。表示创建连接的数据库用户。 ● password：String 类型。表示数据库用户的密码。 ● enable_ce: String 类型。其中 enable_ce=1 表示 JDBC 支持密态等值查询。

参数	描述
	● refreshClientEncryption:String 类型。其中 refreshClientEncryption=1 表示密态数据库支持客户端缓存刷新（默认值为 1）。 ● loggerLevel：String 类型。目前支持 3 种级别：OFF、DEBUG、 TRACE。设置为 OFF 关闭日志，设置为 DEBUG 和 TRACE 记录的日志信息详细程度不同。 ● loggerFile：String 类型。Logger 输出的文件名。需要显示指定日志文件名，若未指定目录则生成在客户端运行程序目录。此参数已废弃， 不再生效，如需使用可通过 java.util.logging 属性文件或系统属性进行配置。 ● allowEncodingChanges：Boolean 类型。设置该参数值为“true”进行字符集类型更改，配合 characterEncoding=CHARSET 设置字符集， 二者使用“&”分隔；characterEncoding 取值范围为 UTF8、GBK、LATIN1。 ● currentSchema：String 类型。在 search-path 中指定要设置的 schema。 ● hostRecheckSeconds：Integer 类型。JDBC 尝试连接主机后会保存主机状态：连接成功或连接失败。在 hostRecheckSeconds 时间内保持可信，超过则状态失效。缺省值是 10 秒。 ● ssl：Boolean 类型。以 SSL 方式连接。 ● ssl=true 可支持 NonValidatingFactory 通道和使用证书的方式： 1、NonValidatingFactory 通道需要配置用户名和密码，同时将 SSL 设置为 true。 2、配置客户端证书、密钥、根证书，将 SSL 设置为 true。 ● sslmode：String 类型。SSL 认证方式。取值范围为：require、verify- ca、verify-full。 ■ require 只尝试 SSL 连接，如果存在 CA 文件，则应设置成 verify-ca 的方式验证。 ■ verify-ca 只尝试 SSL 连接，并且验证服务器是否具有由可信任的证书机构签发的证书。 ■ verify-full 只尝试 SSL 连接，并且验证服务器是否具有由可信任的证书机构签发的证书，以及验证服务器主机名是否与证书中的一致。 ● sslcert：String 类型。提供证书文件的完整路径。客户端和服务端证书的类型为 End Entity。 ● sslkey：String 类型。提供密钥文件的完整路径。使用时将客户端证书转换为 DER 格式： `openssl pkcs8 -topk8 -outform DER -in client.key -out client.key.pk8 -nocrypt`

参数	描述
	● sslrootcert：String 类型。SSL 根证书的文件名。根证书的类型为 CA。 ● sslpassword：String 类型。提供给 ConsoleCallbackHandler 使用。 ● sslpasswordcallback：String 类型。SSL 密码提供者的类名。缺省值：org.postgresql.ssl.jdbc4.LibPQFactory.ConsoleCallbackHandler。 ● sslfactory：String 类型。提供的值是 SSLSocketFactory 在建立 SSL 连接时用的类名。 ● sslfactoryarg：String 类型。此值是上面提供的 sslfactory 类的构造函数的可选参数（不推荐使用）。 ● sslhostnameverifier：String 类型。主机名验证程序的类名。接口实现 javax.net.ssl.HostnameVerifier，默认使用 org.postgresql.ssl.PGjdbcHostnameVerifier。 ● loginTimeout：Integer 类型。指建立数据库连接的等待时间。超时时间单位为秒。 ● connectTimeout：Integer 类型。用于连接服务器操作的超时值。如果连接到服务器花费的时间超过此值，则连接断开。超时时间单位为秒，值为 0 时表示已禁用，timeout 不发生。 ● socketTimeout：Integer 类型。用于 socket 读取操作的超时值。如果从服务器读取所花费的时间超过此值，则连接关闭。超时时间单位为秒，值为 0 时表示已禁用，timeout 不发生。 ● cancelSignalTimeout：Integer 类型。发送取消消息本身可能会阻塞，此属性控制用于取消命令的“connect 超时”和“socket 超时”。 超时时间单位为秒，默认值为 10 秒。 ● tcpKeepAlive：Boolean 类型。启用或禁用 TCP 保活探测功能。默认为 false。 ● logUnclosedConnections：Boolean 类型。客户端可能由于未调用 Connection 对象的 close()方法而泄漏 Connection 对象。最终这些对象将被垃圾回收，并且调用 finalize()方法。如果调用者自己忽略了此操作，该方法将关闭 Connection。 ● assumeMinServerVersion：String 类型。客户端会发送请求进行 float 精度设置。该参数设置要连接的服务器版本，如 assumeMinServerVersion=9.0，可以在建立时减少相关包的发送。 ● ApplicationName：String 类型。设置正在使用连接的 JDBC 驱动的名称。通过在数据库主节点上查询 pg_stat_activity 表可以看到正在连接的客户端信息，JDBC 驱动名称显示在 application_name 列。缺省值为 PostgreSQL JDBC

参数	描述
	Driver。 ● connectionExtraInfo：Boolean 类型。表示驱动是否上报当前驱动的部署路径、进程属主用户到数据库。 取值范围：true 或 false，默认值为 false。设置 connectionExtraInfo 为 true，JDBC 驱动会将当前驱动的部署路径、进程属主用户、url 连接配置信息上报到数据库中，记录在 connection_info 参数里；同时可以在 PG_STAT_ACTIVITY 中查询到。 ● autosave：String 类型。共有 3 种："always", "never", "conservative"。如果查询失败，指定驱动程序应该执行的操作。在 autosave=always 模式下，JDBC 驱动程序在每次查询之前设置一个保存点，并在失败时回滚到该保存点。在 autosave=never 模式（默认）下，无保存点。在 autosave=conservative 模式下，每次查询都会设置保存点，但是只会在“statement XXX 无效”等情况下回滚并重试。 ● protocolVersion：Integer 类型。连接协议版本号，目前仅支持 1 和 3。注意：设置 1 时仅代表连接的是 V1 服务端。设置 3 时将采用 md5 加密方式，需要同步修改数据库的加密方式：gs_guc set -N all -I all -c "password_encryption_type=1"，重启数据库生效后需要创建用 md5 方式加密口令的用户。同时修改 pg_hba.conf，将客户端连接方 式修改为 md5。用新建用户进行登录（不推荐）。 说明： MD5 加密算法安全性低，存在安全风险，建议使用更安全的加密算法。 ● prepareThreshold：Integer 类型。控制 parse 语句何时发送。默认值是 5。第一次 parse 一个 SQL 比较慢，后面再 parse 就会比较快，因为有缓存了。如果一个会话连续多次执行同一个 SQL，在达到 prepareThreshold 次数以上时，JDBC 将不再对这个 SQL 发送 parse 命 令。 ● preparedStatementCacheQueries：Integer 类型。确定每个连接中缓存的查询数，默认情况下是 256。若在 prepareStatement()调用中使用超过 256 个不同的查询，则最近最少使用的查询缓存将被丢弃。0 表示禁用缓存。 ● preparedStatementCacheSizeMiB：Integer 类型。确定每个连接可缓存的最大值（以兆字节为单位），默认情况下是 5。若缓存了超过 5MB 的查询，则最近最少使用的查询缓存将被丢弃。0 表示禁用缓存。 ● databaseMetadataCacheFields：Integer 类型。默认值是 65536。指定每个连接可缓存的最大值。“0”表示禁用缓存。 ● databaseMetadataCacheFieldsMiB：Integer 类型。默认值是 5。每个连接可缓

参数	描述
	存的最大值，单位是 MB。“0”表示禁用缓存。 ● stringtype：String 类型，可选字段为：false, "unspecified", "varchar"。设置通过 setString()方法使用的 PreparedStatement 参数的类型，如果 stringtype 设置为 VARCHAR（默认值），则这些参数将作为 varchar 参数发送给服务器。若 stringtype 设置为 unspecified，则参数将作为 untyped 值发送到服务器，服务器将尝试推断适当的类型。 ● batchMode：String 类型。用于确定是否使用 batch 模式连接。默认值为 on，表示开启 batch 模式。 ● fetchsize：Integer 类型。用于设置数据库连接所创建 statement 的默认 fetchsize。默认值为 0，表示一次获取所有结果。 ● reWriteBatchedInserts：Boolean 类型。批量导入时，该参数设置为 true，可将 N 条插入语句合并为一条：insert into TABLE_NAME values(values1, ..., valuesN), ..., (values1, ..., valuesN);使用该参数时，需设置 batchMode=off。 ● unknownLength：Integer 类型，默认为 Integer.MAX_VALUE。某些 postgresql 类型（例如 TEXT）没有明确定义的长度，当通过 ResultSetMetaData.getColumnDisplaySize 和 ResultSetMetaData.getPrecision 等函数返回关于这些类型的数据时，此参数指定未知长度类型的长度。 ● uppercaseAttributeName：Boolean 类型，默认值为 false 不开启，为 true 时开启。该参数开启后会将获取元数据的接口的查询结果转为大写。适用场景为数据库中存储元数据全为小写，但要使用大写的元数据作为出参和入参。 涉及到的接口：8.1.3 java.sql.DatabaseMetaData、8.1.7java.sql.ResultSetMetaData ● defaultRowFetchSize：Integer 类型。确定一次 fetch 在 ResultSet 中读取的行数。限制每次访问数据库时读取的行数可以避免不必要的内存消耗，从而避免 OutOfMemoryException。缺省值是 0，这意味着 ResultSet 中将一次获取所有行。没有负数。 ● binaryTransfer：Boolean 类型。使用二进制格式发送和接收数据，默认值为“false”。 ● binaryTransferEnable：String 类型。启用二进制传输的类型列表，以逗号分隔。OID 编号和名称二选一，例如 binaryTransferEnable=Integer4_ARRAY,Integer8_ARRAY。 比如：OID 名称为 BLOB，编号为 88，可以如下配置： binaryTransferEnable=BLOB 或 binaryTransferEnable=88 ● binaryTransferDisEnable：String 类型。禁用二进制传输的类型列表，以逗号

参数	描述
	分隔。OID 编号和名称二选一。覆盖 binaryTransferEnable 的设置。 ● blobMode：String 类型。用于设置 setBinaryStream 方法为不同类型 的数据赋值，设置为 on 时表示为 blob 类型数据赋值，设置为 off 时表示为 bytea 类型数据赋值，默认为 on。 ● socketFactory：String 类型。用于创建与服务器 socket 连接的类的名称。该类必须实现了接口 “javax.net.SocketFactory”，并定义无参 或单 String 参数的构造函数。 ● socketFactoryArg：String 类型。此值是上面提供的 socketFactory 类的构造函数的可选参数，不推荐使用。 ● receiveBufferSize：Integer 类型。该值用于设置连接流上的 SO_RCVBUF。 ● sendBufferSize：Integer 类型。该值用于设置连接流上的 SO_SNDBUF。 ● preferQueryMode：String 类型。共有 4 种："extended", "extendedForPrepared", "extendedCacheEverything", "simple"。用于指定执行查询的模式，simple 模式会 excute，不 parse 和 bind； extended 模式会 bind 和 excute；extendedForPrepared 模式为 prepared statement 扩展使用；extendedCacheEverything 模式会缓存每个 statement。 ● targetServerType： String 类型。该参数识别主备数据节点是通过查询 URL 连接串中，数据节点是否允许写操作来实现的，默认为"any"。共有四种："any"，"master"，"slave"，"preferSlave"： ■ master 则尝试连接到 URL 连接串中的主节点，如果找不到就抛出异常。 ■ slave 则尝试连接到 URL 连接串中的备节点，如果找不到就抛出异常。 ■ preferSlave 则尝试连接到 URL 连接串中的备数据节点（如果有可用的话），否则连接到主数据节点。 ■ any 则尝试连接 URL 连接串中的任何一个数据节点。 ● priorityServers：Integer 类型。此值用于指定 url 上配置的前 n 个节点作为主数据库实例被优先连接。默认值为 null。该值为数字，大于 0， 且小于 url 上配置的 DN 数量。 例如：jdbc:postgresql://host1:port1,host2:port2,host3:port3,host4:port4,/database? priorityServers=2。即表示 host1 与 host2 为主数据库实例节点，host3 与 host4 为容灾数据库实例节点。 ● forceTargetServerSlave：Boolean 类型。此值用于控制是否开启强制连接备机功能，并在数据库实例发生主备切换时，禁止已存在的连接在升主备机上继续使用。默认值为 false，表示不开启强制连接备机功能。true，表示开启强

参数	描述
	制连接备机功能。 ● traceInterfaceClass：String 类型。默认值为 null，用于获取 traceId 的实现类。值是实现获取 traceId 方法的接口 org.postgresql.log.Tracer 的 实现类的完整限定类名。 ● use_boolean：Boolean 类型。用于设置 extended 模式下 setBoolean 方法绑定的 oid 类型，默认为 false，绑定 int2 类型；设置为 true 则绑定 bool 类型。 ● allowReadOnly：Boolean 类型。用于设置是否允许只读模式，默认为 true，允许设置只读模式；设置为 false 则禁用只读模式。 ● TLSCiphersSupperted：String 类型。用于设置支持的 TLS 加密套件， 默认为 TLS_DHE_RSA_WITH_AES_128_GCM_SHA256,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
user	数据库用户。
password	数据库用户的密码。

说明

uppercaseAttributeName 参数开启后，如果数据库中有小写、大写和大小写混合的元数据，只能查询出小写部分的元数据，并以大写的形式输出，使用前请务必确认元数据的存储是否全为小写以避免数据出错。

示例

```
//以下代码将获取数据库连接操作封装为一个接口，可通过给定用户名和密码来连接数据库。
public static Connection getConnect(String username, String passwd)
{
//驱动类。
String driver = "org.postgresql.Driver";
//数据库连接描述符。
String sourceURL = "jdbc:postgresql://10.10.0.13:15432/postgres"; Connection conn = null;
try
{
//加载驱动。Class.forName(driver);
```

```
}
catch( Exception e )
{
e.printStackTrace(); return null;
}
try
{
//创建连接。
conn = DriverManager.getConnection(sourceURL, username, passwd);
System.out.println("Connection succeed!");
}
catch(Exception e)
{
e.printStackTrace(); return null;
}
return conn;
};
// 以下代码将使用 Properties 对象作为参数建立连接
public static Connection getConnectUseProp(String username, String passwd)
{
//驱动类。
String driver = "org.postgresql.Driver";
//数据库连接描述符。
String sourceURL = "jdbc:postgresql://10.10.0.13:15432/postgres?"; Connection conn = null;
Properties info = new Properties();
try
{
//加载驱动。Class.forName(driver);
}
catch( Exception e )
{
e.printStackTrace(); return null;
}
try
{
info.setProperty("user", username); info.setProperty("password", passwd);
//创建连接。
conn = DriverManager.getConnection(sourceURL, info); System.out.println("Connection
succeed!");
}
catch(Exception e)
```

```
{
e.printStackTrace(); return null;
}
return conn;
};
```

3.5 连接数据库（以 SSL 方式）

用户通过 JDBC 连接数据库服务器时，可以通过开启 SSL 加密客户端和服务器之间 的通讯，为敏感数据在 Internet 上的传输提供了一种安全保障手段。本小节主要介绍应用程序通过 JDBC 如何采用 SSL 的方式连接 GBase 8s 数据库。在使用本小节所描述的方法前，默认用户已经获取了服务端和客户端所需要的证书和私钥文件。

服务端配置

当开启 SSL 模式后，必须提供根证书、服务器证书和私钥。

配置步骤（假设用户的证书文件放在数据目录/opt/database/install/data/dn/下，且采用默认文件名）：

(1) 以操作系统用户 gbase 登录数据库主节点。

(2) 生成并配置证书。

生成 SSL 证书，具体请参见证书生成。将生成出的文件 server.crt，server.key，cacert.pem 拷贝到服务端数据目录下。

使用如下命令可以查询数据库节点的数据目录，instance 列为数据目录。

```
gs_om -t status --detail
```

在 Unix 系统上，server.crt、server.key 的权限设置必须禁止任何外部或组的访问，请执行如下命令实现这一点。

```
chmod 0600 server.key
```

(3) 开启 SSL 认证模式。

```
gs_guc set -D /opt/database/install/data/dn -c "ssl=on"
```

(4) 配置客户端接入认证参数，IP 为所要连接的主机 IP。

```
gs_guc reload -D /opt/database/install/data/dn -h "hostssl all     all    127.0.0.1/32      cert"
gs_guc reload -D /opt/database/install/data/dn -h "hostssl all     all    IP/32      cert"
```

表示允许 127.0.0.1/32 网段的客户端以 ssl 认证方式连接到数据库服务器。

须知

如果服务端 pg_hba.conf 文件中 METHOD 配置为 cert，则只有客户端使用证书（client.crt）中所设置的用户名（common name）才能够成功连接数据库。如果设置为 md5、sm3 或 sha256 则对连接数据库的用户没有限制。

MD5 加密算法安全性低，存在安全风险，建议使用更安全的加密算法。

(5) 配置 SSL 认证相关的数字证书参数。各命令后所附为设置成功的回显。

```
gs_guc set -D /opt/database/install/data/dn -c "ssl_cert_file='server.crt'"
gs_guc set: ssl_cert_file='server.crt'
gs_guc set -D /opt/database/install/data/dn -c "ssl_key_file='server.key'"
gs_guc set: ssl_key_file='server.key'
gs_guc set -D /opt/database/install/data/dn -c "ssl_ca_file='cacert.pem'"
gs_guc set: ssl_ca_file='cacert.pem'
```

(6) 重启数据库。

```
gs_om -t restart
```

客户端配置

上传证书文件，将在服务端配置章节生成出的文件 client.key.pk8，client.crt，cacert.pem 放置在客户端。

示例

注：示例 1 和示例 2 选择其一。

```
public class SSL{
public static void main(String[] args) { Properties urlProps = new Properties();
String urls = "jdbc:postgresql://10.29.37.136:15432/postgres";
/**
* ================== 示例 1 使用 NonValidatingFactory 通道
*/ urlProps.setProperty("sslfactory","org.postgresql.ssl.NonValidatingFactory");
urlProps.setProperty("user", "world");
urlProps.setProperty("password", "test@123"); urlProps.setProperty("ssl", "true");
/**
* ================== 示例 2 使用证书
*/
urlProps.setProperty("sslcert", "client.crt"); urlProps.setProperty("sslkey", "client.key.pk8");
urlProps.setProperty("sslrootcert", "cacert.pem"); urlProps.setProperty("user", "world");
urlProps.setProperty("ssl", "true");
/* sslmode 可配置为：require、verify-ca、verify-full，以下三个示例选择其一*/
```

```
/* ================== 示例 2.1 设置 sslmode 为 require，使用证书 */
urlProps.setProperty("sslmode", "require");
/* ================== 示例 2.2 设置 sslmode 为 verify-ca，使用证书 */
urlProps.setProperty("sslmode", "verify-ca");
/* ================== 示例 2.3 设置 sslmode 为 verify-full，使用证书（Linux 下验证）
*/ urls = "jdbc:postgresql://world:15432/postgres";
urlProps.setProperty("sslmode", "verify-full"); try {
Class.forName("org.postgresql.Driver").newInstance();
} catch (Exception e) { e.printStackTrace();
}
try {
Connection conn;
conn = DriverManager.getConnection(urls,urlProps); conn.close();
} catch (Exception e) { e.printStackTrace();
}
}
}
/**
注：将客户端密钥转化为 DER 格式:
openssl pkcs8 -topk8 -outform DER -in client.key -out client.key.pk8 -nocrypt
openssl pkcs8 -topk8 -inform PEM -in client.key -outform DER -out client.key.der -v1
PBE-MD5-DES
openssl pkcs8 -topk8 -inform PEM -in client.key -outform DER -out client.key.der -v1
PBE-SHA1-3DES
以上算法由于安全级别较低，不推荐使用。
如果客户需要采用更高级别的私钥加密算法，启用 bouncycastle 或者其他第三方私钥解密
密码包后可以使用的私钥加密算法如下：
openssl pkcs8 -in client.key -topk8 -outform DER -out client.key.der -v2 AES128
openssl pkcs8 -in client.key -topk8 -outform DER -out client.key.der -v2 aes-256-cbc -iter
1000000
openssl pkcs8 -in client.key -topk8 -out client.key.der -outform Der -v2 aes-256-cbc -v2prf
hmacWithSHA512
启用 bouncycastle：使用 jdbc 的项目引入依赖：bcpkix-jdk15on.jar 包，版本建议：1.65 以
上。
*/
```

3.6 执行 SQL 语句

3.6.1 执行普通 SQL

执行普通 SQL

应用程序通过执行 SQL 语句来操作数据库的数据（不用传递参数的语句），需要按以下步骤执行：

```
Connection conn = DriverManager.getConnection("url","user","password"); Statement stmt = conn.createStatement();
```

(1) 调用 Connection 的 createStatement 方法创建语句对象。

```
int rc = stmt.executeUpdate("CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name VARCHAR(32));");
```

(2) 调用 Statement 的 executeUpdate 方法执行 SQL 语句。

- 数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务，事务块中不支持 vacuum 操作。如果其中有一个语句失败，那么整个请求都将会被回滚。
- 使用 Statement 执行多语句时应以";”作为各语句间的分隔符，存储过程、函数、匿名块不支持多语句执行。
- "/”可用作创建单个存储过程、函数、匿名块的结束符。

(3) 关闭语句对象。

```
stmt.close();
```

3.6.2 执行预编译 SQL 语句

预编译语句是只编译和优化一次，然后可以通过设置不同的参数值多次使用。由于已经预先编译好，后续使用会减少执行时间。因此，如果多次执行一条语句，请选择使用预编译语句。可以按以下步骤执行：

(1) 调用 Connection 的 prepareStatement 方法创建预编译语句对象。

(2) 调用 PreparedStatement 的 setShort 设置参数。

```
pstmt.setShort(1, (short)2);
```

(3) 调用 PreparedStatement 的 executeUpdate 方法执行预编译 SQL 语句。

```
int rowcount = pstmt.executeUpdate();
```

(4) 调用 PreparedStatement 的 close 方法关闭预编译语句对象。

```
pstmt.close();
```

3.6.3 调用存储过程

GBase 8s 支持通过 JDBC 直接调用事先创建的存储过程，步骤如下：

(1) 调用 Connection 的 prepareCall 方法创建调用语句对象。

```
Connection myConn = DriverManager.getConnection("url","user","password");
CallableStatement cstmt = myConn.prepareCall("{? = CALL TESTPROC(?,?,?)}");
```

(2) 调用 CallableStatement 的 setInt 方法设置参数。

```
cstmt.setInt(2, 50);
cstmt.setInt(1, 20);
cstmt.setInt(3, 90);
```

(3) 调用 CallableStatement 的 registerOutParameter 方法注册输出参数。

```
cstmt.registerOutParameter(4, Types.INTEGER); //注册 out 类型的参数，类型为整型。
```

(4) 调用 CallableStatement 的 execute 执行方法调用。

```
cstmt.execute();
```

(5) 调用 CallableStatement 的 getInt 方法获取输出参数。

```
int out = cstmt.getInt(4); //获取 out 参数
```

示例：

```
//在数据库中已创建了如下存储过程，它带有 out 参数。
create or replace procedure testproc (
psv_in1 in integer, psv_in2 in integer, psv_inout in out integer
)
as begin
psv_inout := psv_in1 + psv_in2 + psv_inout; end;
/
```

(6) 调用 CallableStatement 的 close 方法关闭调用语句。

```
cstmt.close();
```

说明

很多的数据库类如 Connection、Statement 和 ResultSet 都有 close()方法，在使用完对象

后应把它们关闭。要注意的是，Connection 的关闭将间接关闭所有与它关联的 Statement，Statement 的关闭间接关闭了 ResultSet。

一些 JDBC 驱动程序还提供命名参数的方法来设置参数。命名参数的方法允许根据名称而不是顺序来设置参数，若参数有默认值，则可以不用指定参数值就可以使用此参数的默认值。即使存储过程中参数的顺序发生了变更，也不必修改应用程序。目前 GBase 8s 数据库的 JDBC 驱动程序不支持此方法。

GBase 8s 数据库不支持带有输出参数的函数，也不支持存储过程和函数参数默认值。

须知

- 当游标作为存储过程的返回值时，如果使用 JDBC 调用该存储过程，返回的游标将不可用。
- 存储过程不能和普通 SQL 在同一条语句中执行。
- 存储过程中 inout 类型参数必需注册出参。

3.6.4 Oracle 兼容模式启用重载时，调用存储过程

打开参数 behavior_compat_options='proc_outparam_override'后，JDBC 调用事先创建的存储过程。

(1) 调用 Connection 的 prepareCall 方法创建调用语句对象。

```
Connection conn = DriverManager.getConnection("url","user","password"); CallableStatement cs = conn.prepareCall("{ CALL TEST_PROC(?,?,?) }");
```

(2) 调用 CallableStatement 的 setInt 方法设置参数。

```
PGobject pGobject = new PGobject();
```

pGobject.setType("public.compfoo"); // 设置复合类型名，格式为"schema.typename”。pGobject.setValue("(1,demo)"); // 绑定复合类型值，格式为"(value1,value2)”。cs.setObject(1, pGobject);

(3) 调用 CallableStatement 的 registerOutParameter 方法注册输出参数。

// 注册 out 类型的参数，类型为复合类型,格式为"schema.typename”。

```
cs.registerOutParameter(2, Types.STRUCT, "public.compfoo");
```

(4) 调用 CallableStatement 的 execute 执行方法调用。

```
cs.execute();
```

(5) 调用 CallableStatement 的 getObject 方法获取输出参数。

```
PGobject result = (PGobject)cs.getObject(2);  // 获取 out 参数
result.getValue(); // 获取复合类型字符串形式值。
result.getArrayValue(); //获取复合类型数组形式值，以复合数据类型字段顺序排序。
result.getStruct(); //获取复合类型子类型名，按创建顺序排序。
```

(6) 调用 CallableStatement 的 close 方法关闭调用语句。

```
cs.close();
```

说明

oracle 兼容模式开启参数后，调用存储过程必须使用{call proc_name(?,?,?)}形式调用，调用函数必须使用{? = call func_name(?,?)}形式调用（等号左侧的" ? "为函数返回值的占位符，用于注册函数返回值）。

参数 behavior_compat_options='proc_outparam_override'行为变更后，业务需要重新建立连接，否则无法正确调用存储过程和函数。

函数和存储过程中包含复合类型时，参数的绑定与注册需要使用 schema.typename 形式。

```
// 在数据库创建复合数据类型。
CREATE TYPE compfoo AS (f1 int, f3 text);
// 在数据库中已创建了如下存储过程，它带有 out 参数。
create or replace procedure test_proc (
psv_in in compfoo, psv_out out compfoo
)
as begin
psv_out := psv_in; end;
/
```

3.6.5 执行批处理

用一条预处理语句处理多条相似的数据，数据库只创建一次执行计划，节省了语句的编译和优化时间。可以按如下步骤执行：

(1) 调用 Connection 的 prepareStatement 方法创建预编译语句对象。

```
Connection conn = DriverManager.getConnection("url","user","password"); PreparedStatement
pstmt = conn.prepareStatement("INSERT INTO customer_t1 VALUES (?)");
```

针对每条数据都要调用 setShort 设置参数，以及调用 addBatch 确认该条设置完毕。

```
pstmt.setShort(1, (short)2);
```

```
pstmt.addBatch();
```

(2) 调用 PreparedStatement 的 executeBatch 方法执行批处理。

```
int[] rowcount = pstmt.executeBatch();
```

(3) 调用 PreparedStatement 的 close 方法关闭预编译语句对象。

```
pstmt.close();
```

说明

- 在实际的批处理过程中，通常不终止批处理程序的执行，否则会降低数据库的性能。因此在批处理程序时，应该关闭自动提交功能，每几行提交一次。关闭自动提交功能的语句为：conn.setAutoCommit(false);

3.7 处理结果集

SQL 兼容性检查

GBase 8s 数据库支持接收外部发出的 SQL，并对其执行语法解析，无需重写或执行（因为数据库内当前不存在相应元数据）。若失败，返回失败原因。建议配合 GBase 8s 迁移工具使用，用于 SQL 兼容性检查。

设置结果集类型

不同类型的结果集有各自的应用场景，应用程序需要根据实际情况选择相应的结果集类型。在执行 SQL 语句过程中，都需要先创建相应的语句对象，而部分创建语句对象的方法提供了设置结果集类型的功能。具体的参数设置如表 6-3 所示。涉及的 Connection 的方法如下：

```
//创建一个 Statement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。
createStatement(int resultSetType, int resultSetConcurrency);
//创建一个 PreparedStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。
prepareStatement(String sql, int resultSetType, int resultSetConcurrency);
//创建一个 CallableStatement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。
prepareCall(String sql, int resultSetType, int resultSetConcurrency);
```

表 3-3 结果集类型

参数	描述
resultSetType	表示结果集的类型，具体有三种类型： ● ResultSet.TYPE_FORWARD_ONLY：ResultSet 只能向前移动。

	是缺省值。 ● ResultSet.TYPE_SCROLL_SENSITIVE：在修改后重新滚动到修改所在行，可以看到修改后的结果。 ● ResultSet.TYPE_SCROLL_INSENSITIVE：对可修改例程所做的编辑不进行显示。 说明： 结果集从数据库中读取了数据之后，即使类型是 ResultSet.TYPE_SCROLL_SENSITIVE，也不会看到由其他事务在这之后引起的改变。调用 ResultSet 的 refreshRow()方法，可进入数据库并从其中取得当前游标所指记录的最新数据。
resultSetConcurrenc y	表示结果集的并发，具体有两种类型： ● ResultSet.CONCUR_READ_ONLY：如果不从结果集中的数据建立一个新的更新语句，不能对结果集中的数据进行更新。 ● ResultSet.CONCUR_UPDATEABLE：可改变的结果集。对于可滚动的结果集，可对结果集进行适当的改变。

在结果集中定位

ResultSet 对象具有指向其当前数据行的光标。最初，光标被置于第一行之前。next 方法将光标移动到下一行；因为该方法在 ResultSet 对象没有下一行时返回 false，所以可

以在 while 循环中使用它来迭代结果集。但对于可滚动的结果集，JDBC 驱动程序提供更多的定位方法，使 ResultSet 指向特定的行。定位方法如下表所示。

表 3-4 在结果集中定位的方法

方法	描述
next()	把 ResultSet 向下移动一行。
previous()	把 ResultSet 向上移动一行。
beforeFirst()	把 ResultSet 定位到第一行之前。
afterLast()	把 ResultSet 定位到最后一行之后。
first()	把 ResultSet 定位到第一行。
last()	把 ResultSet 定位到最后一行。

absolute(int)	把 ResultSet 移动到参数指定的行数。
relative(int)	通过设置为 1 向前（设置为 1，相当于 next()）或者向后（设置为-1，相当于 previous()）移动参数指定的行。

获取结果集中光标的位置

对于可滚动的结果集，可能会调用定位方法来改变光标的位置。JDBC 驱动程序提供了获取结果集中光标所处位置的方法。获取光标位置的方法如下表所示。

表 3-5 获取结果集光标的位置

方法	描述
isFirst()	是否在一行。
isLast()	是否在最后一行。
isBeforeFirst()	是否在第一行之前。
isAfterLast()	是否在最后一行之后。
getRow()	获取当前在第几行。

获取结果集中的数据

ResultSet 对象提供了丰富的方法，以获取结果集中的数据。获取数据常用的方法如表 6-6 所示，其他方法请参考 JDK 官方文档。

表 3-6 ResultSet 对象的常用方法

方法	描述
int getInt(int columnIndex)	按列标获取 int 型数据。
int getInt(String columnLabel)	按列名获取 int 型数据。
String getString(int columnIndex)	按列标获取 String 型数据。
String getString(String columnLabel)	按列名获取 String 型数据。
Date getDate(int columnIndex)	按列标获取 Date 型数据
Date getDate(String columnLabel)	按列名获取 Date 型数据。

3.8 关闭连接

在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。

关闭数据库连接可以直接调用其 close 方法即可。如：Connection conn = DriverManager.getConnection("url","user","password") ; conn.close() ;

3.9 日志管理

GBase 8s JDBC 驱动程序支持使用日志记录，来帮助解决在应用程序中使用 JDBC 驱动程序时的问题。GBase 8s 的 JDBC 支持如下三种日志管理方式：

- 对接应用程序使用的 SLF4J 日志框架。
- 对接应用程序使用的 JdkLogger 日志框架。
- SLF4J 和 JdkLogger 是业界 Java 应用程序日志管理的主流框架，描述应用程序如何使用这些框架超出了本文范围，用户请参考对应的官方文档（SLF4J：http://www.slf4j.org/manual.html，JdkLogger：https://docs.oracle.com/javase/8/docs/technotes/guides/logging/overview.html）。

  方式一：对接应用程序的 SLF4J 日志框架。

  在建立连接时，url 配置 logger=Slf4JLogger。

  可采用 Log4j 或 Log4j2 来实现 SLF4J。当采用 Log4j 实现 SLF4J，需要添加如下 jar 包：

  log4j-*.jar、slf4j-api-*.jar、slf4j-log4*-*.jar，（*区分版本），和配置文件：

  log4j.properties。若采用 Log4j2 实现 SLF4J，需要添加如下 jar 包：log4j-api-*.jar、log4j-core-*.jar、log4j-slf4j18-impl-*.jar、slf4j-api-*-alpha1.jar（*区分版本），和配 置文件：log4j2.xml。

  此方式支持日志管控。SLF4J 可通过文件中的相关配置实现强大的日志管控功能，建议使用此方式进行日志管理。

  注意

  - 此方式依赖 slf4j 的通用 API 接口，如 org.slf4j.LoggerFactory.getLogger(String name)、org.slf4j.Logger.debug(String var1) 、 org.slf4j.Logger.info(String var1) 、org.slf4j.Logger.warn(String warn)、org.slf4j.Logger.warn(String warn)等，若以上接口发生变更，日志将无法打印。

示例：

```
public static Connection GetConnection(String username, String passwd){
String sourceURL = "jdbc:postgresql://10.10.0.13:15432/postgres?logger=Slf4JLogger";
Connection conn = null;
try{
//创建连接
conn = DriverManager.getConnection(sourceURL,username,passwd);
System.out.println("Connection succeed!");
}catch (Exception e){ e.printStackTrace(); return null;
}
return conn;
}
```

log4j.properties 示例：

```
log4j.logger.org.postgresql=ALL, log_gsjdbc
# 默认文件输出配置 log4j.appender.log_gsjdbc=org.apache.log4j.RollingFileAppender
log4j.appender.log_gsjdbc.Append=true log4j.appender.log_gsjdbc.File=gsjdbc.log
log4j.appender.log_gsjdbc.Threshold=TRACE log4j.appender.log_gsjdbc.MaxFileSize=10MB
log4j.appender.log_gsjdbc.MaxBackupIndex=5
log4j.appender.log_gsjdbc.layout=org.apache.log4j.PatternLayout
log4j.appender.log_gsjdbc.layout.ConversionPattern=%d %p %t %c - %m%n
log4j.appender.log_gsjdbc.File.Encoding = UTF-8
```

log4j2.xml 示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="OFF">
<appenders>
<Console name="Console" target="SYSTEM_OUT">
<PatternLayout pattern="%d %p %t %c - %m%n"/>
</Console>
<File name="FileTest" fileName="test.log">
<PatternLayout pattern="%d %p %t %c - %m%n"/>
</File>
<!--JDBC Driver 日志文件输出配置，支持日志回卷，设定日志大小超过 10MB 时，创建新的文件，新文件的命名格式为：年-月-日-文件编号-->
<RollingFile name="RollingFileJdbc" fileName="gsjdbc.log"
filePattern="%d{yyyy-MM-dd}-%i.log">
<PatternLayout pattern="%d %p %t %c - %m%n"/>
<Policies>
<SizeBasedTriggeringPolicy size="10 MB"/>
```

```
</Policies>
</RollingFile>
</appenders>
<loggers>
<root level="all">
<appender-ref ref="Console"/>
<appender-ref ref="FileTest"/>
</root>
<!--指定 JDBC Driver 日志，级别为：all，可查看所有日志，输出到 gsjdbc.log 文件中-->
<logger name="org.postgresql" level="all" additivity="false">
<appender-ref ref="RollingFileJdbc"/>
</logger>
</loggers>
</configuration>
```

方式二：对接应用程序使用的 JdkLogger 日志框架。

默认的 Java 日志记录框架将其配置存储在名为 logging.properties 的文件中。Java 会在 Java 安装目录的文件夹中安装全局配置文件。logging.properties 文件也可以创建并与单个项目一起存储。

logging.properties 配置示例：

```
# 指定处理程序为文件。
handlers= java.util.logging.FileHandler
# 指定默认全局日志级别
.level= ALL
# 指定日志输出管控标准 java.util.logging.FileHandler.level=ALL
java.util.logging.FileHandler.pattern = gsjdbc.log java.util.logging.FileHandler.limit = 500000
java.util.logging.FileHandler.count = 30 java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter java.util.logging.FileHandler.append=false
```

代码中使用示例：

```
System.setProperty("java.util.logging.FileHandler.pattern","jdbc.log");
FileHandler fileHandler = new
FileHandler(System.getProperty("java.util.logging.FileHandler.pattern")); Formatter formatter
= new SimpleFormatter();
fileHandler.setFormatter(formatter);
Logger logger = Logger.getLogger("org.postgresql"); logger.addHandler(fileHandler);
logger.setLevel(Level.ALL); logger.setUseParentHandlers(false);
```

链路跟踪功能

JDBC 驱动程序提供了应用到数据库的链路跟踪功能，用于将数据库端离散的 SQL 和应用程序的请求关联起来。该功能需要应用开发者实现 org.postgresql.log.Tracer 接口类，并在 url 中指定接口实现类的全限定名。

url 示例：

```
String URL = "jdbc:postgresql://127.0.0.1:15432/postgres?
traceInterfaceClass=xxx.xxx.xxx.GBase 8sTraceImpl";
```

org.postgresql.log.Tracer 接口类定义如下：

```
public interface Tracer {
// Retrieves the value of traceId. String getTraceId();
}
```

org.postgresql.log.Tracer 接口实现类示例：

```
import org.postgresql.log.Tracer;
public class GBase 8sTraceImpl implements Tracer {
private static MDC mdc = new MDC(); private final String TRACE_ID_KEY = "traceId";
public void set(String traceId) { mdc.put(TRACE_ID_KEY, traceId);
}
public void reset() { mdc.clear();
}
@Override
public String getTraceId() {
return mdc.get(TRACE_ID_KEY);
}
}
```

上下文映射示例，用于存放不同请求的生成的 traceId。

```
import java.util.HashMap;
public class MDC {
static final private ThreadLocal<HashMap<String, String>> threadLocal = new
ThreadLocal<>();
public void put(String key, String val) { if (key == null || val == null) {
throw new IllegalArgumentException("key or val cannot be null");
} else {
if (threadLocal.get() == null) { threadLocal.set(new HashMap<>());
}
threadLocal.get().put(key, val);
}
}
```

```
public String get(String key) { if (key == null) {
throw new IllegalArgumentException("key cannot be null");
} else if (threadLocal.get() == null) { return null;
} else {
return threadLocal.get().get(key);
}
}
public void clear() {
if (threadLocal.get() == null) { return;
} else {
threadLocal.get().clear();
}
}
}
```

业务使用 traceId 示例。

```
String traceId = UUID.randomUUID().toString().replaceAll("-", ""); GBase
8sTrace.set(traceId);
pstm = con.prepareStatement("select * from test_trace_id where id = ?"); pstm.setInt(1, 1);
pstm.execute();
pstm = con.prepareStatement("insert into test_trace_id values(?,?)"); pstm.setInt(1, 2);
pstm.setString(2, "test"); pstm.execute(); GBase 8sTrace.reset();
```

3.10 JDBC 接口参考

JDBC 接口是一套提供给用户的 API 方法，本节将对部分常用接口做具体描述。

3.10.1 java.sql.Connection

java.sql.Connection 是数据库连接接口。

表 3-7 对 java.sql.Connection 接口的支持情况

方法名	返回值类型	支持 JDBC 4
abort(Executor executor)	void	Yes
clearWarnings()	void	Yes
close()	void	Yes
commit()	void	Yes

createArrayOf(String typeName, Object[] elements)	Array	Yes
createBlob()	Blob	Yes
createClob()	Clob	Yes
createSQLXML()	SQLXML	Yes
createStatement()	Statement	Yes
createStatement(int resultSetType, int resultSetConcurrency)	Statement	Yes
createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Statement	Yes
getAutoCommit()	Boolean	Yes
getCatalog()	String	Yes
getClientInfo()	Properties	Yes
getClientInfo(String name)	String	Yes
getHoldability()	int	Yes
getMetaData()	DatabaseMetaData	Yes
getNetworkTimeout()	int	Yes
getSchema()	String	Yes
getTransactionIsolation()	int	Yes
getTypeMap()	Map<String,Class<?>>	Yes
getWarnings()	SQLWarning	Yes
isClosed()	Boolean	Yes
isReadOnly()	Boolean	Yes

isValid(int timeout)	boolean	Yes
nativeSQL(String sql)	String	Yes
prepareCall(String sql)	CallableStatement	Yes
prepareCall(String sql, int resultSetType, int resultSetConcurrency)	CallableStatement	Yes
prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	CallableStatement	Yes
prepareStatement(String sql)	PreparedStatement	Yes
prepareStatement(String sql, int autoGeneratedKeys)	PreparedStatement	Yes
prepareStatement(String sql, int[] columnIndexes)	PreparedStatement	Yes
prepareStatement(String sql, int resultSetType, int resultSetConcurrency)	PreparedStatement	Yes
prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	PreparedStatement	Yes
prepareStatement(String sql, String[] columnNames)	PreparedStatement	Yes
releaseSavepoint(Savepoint savepoint)	void	Yes
rollback()	void	Yes
rollback(Savepoint savepoint)	void	Yes
setAutoCommit(boolean autoCommit)	void	Yes

setClientInfo(Properties properties)	void	Yes
setClientInfo(String name,String value)	void	Yes
setHoldability(int holdability)	void	Yes
setNetworkTimeout (Executor executor, int milliseconds)	void	Yes
setReadOnly(boolean readOnly)	void	Yes
setSavepoint()	Savepoint	Yes
setSavepoint(String name)	Savepoint	Yes
setSchema(String schema)	void	Yes
setTransactionIsolation(int level)	void	Yes
setTypeMap(Map<String, Class<?>> map)	void	Yes

须知

- 接口内部默认使用自动提交模式，若通过 setAutoCommit(false)关闭自动提交，将会导致后面执行的语句都受到显式事务包裹，数据库中不支持事务中执行的语句不能在此模式下执行。

3.10.2 java.sql.CallableStatement

java.sql.CallableStatement 是存储过程执行接口。

表 3-8 对 java.sql.CallableStatement 的支持情况

方法名	返回值类型	支持 JDBC 4
getArray(int parameterIndex)	Array	Yes
getBigDecimal(int parameterIndex)	BigDecimal	Yes
getBlob(int parameterIndex)	Blob	Yes

getBoolean(int parameterIndex)	boolean	Yes
getByte(int parameterIndex)	byte	Yes
getBytes(int parameterIndex)	byte[]	Yes
getClob(int parameterIndex)	Clob	Yes
getDate(int parameterIndex)	Date	Yes
getDate(int parameterIndex, Calendar cal)	Date	Yes
getDouble(int parameterIndex)	double	Yes
getFloat(int parameterIndex)	float	Yes
getInt(int parameterIndex)	int	Yes
getLong(int parameterIndex)	long	Yes
getObject(int parameterIndex)	Object	Yes
getObject(int parameterIndex, Class<T> type)	Object	Yes
getShort(int parameterIndex)	short	Yes
getSQLXML(int parameterIndex)	SQLXML	Yes
getString(int parameterIndex)	String	Yes
getNString(int parameterIndex)	String	Yes
getTime(int parameterIndex)	Time	Yes
getTime(int parameterIndex, Calendar cal)	Time	Yes
getTimestamp(int parameterIndex)	Timestamp	Yes
getTimestamp(int	Timestamp	Yes

parameterIndex, Calendar cal)		
registerOutParameter(int parameterIndex, int type)	void	Yes
registerOutParameter(int parameterIndex, int sqlType, int type)	void	Yes
wasNull()	Boolean	Yes

说明

- 不允许含有 OUT 参数的语句执行批量操作。
- 以下方法是从 java.sql.Statement 继承而来：close，execute，executeQuery，executeUpdate, getConnection, getResultSet, getUpdateCount, isClosed，setMaxRows，setFetchSize。
- 以下方法是从 java.sql.PreparedStatement 继承而来：addBatch，clearParameters，execute，executeQuery，executeUpdate，getMetaData，setBigDecimal，setBoolean，setByte，setBytes，setDate，setDouble，setFloat，setInt，setLong，setNull，setObject，setString，setTime，setTimestamp 。
- registerOutParameter(int parameterIndex, int sqlType, int type)方法仅用于注册复合数据类型，其它类型不支持。

3.10.3 java.sql.DatabaseMetaData

java.sql.DatabaseMetaData 是数据库对象定义接口。

表 3-9 对 java.sql.DatabaseMetaData 的支持情况

方法名	返回值类型	支持 JDBC 4
allProceduresAreCallable()	boolean	Yes
allTablesAreSelectable()	boolean	Yes
autoCommitFailureClosesAll ResultSets()	boolean	Yes
dataDefinitionCausesTransac tionCommit()	boolean	Yes

dataDefinitionIgnoredInTran sactions()	boolean	Yes
deletesAreDetected(int type)	boolean	Yes
doesMaxRowSizeIncludeBlob s()	boolean	Yes
generatedKeyAlwaysReturne d()	boolean	Yes
getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean nullable)	ResultSet	Yes
getCatalogs()	ResultSet	Yes
getCatalogSeparator()	String	Yes
getCatalogTerm()	String	Yes
getClientInfoProperties()	ResultSet	Yes
getColumnPrivileges(String catalog, String schema, String table, String columnNamePattern)	ResultSet	Yes
getConnection()	Connection	Yes
getCrossReference(String parentCatalog, String parentSchema, String parentTable, String foreignCatalog, String foreignSchema, String foreignTable)	ResultSet	Yes
getDefaultTransactionIsolati on()	int	Yes
getExportedKeys(String catalog, String schema, String table)	ResultSet	Yes
getExtraNameCharacters()	String	Yes
getFunctionColumns(String catalog, String schemaPattern, String functionNamePattern, String columnNamePattern)	ResultSet	Yes
getFunctions(String catalog, String schemaPattern, String functionNamePattern)	ResultSet	Yes
getIdentifierQuoteString()	String	Yes

getImportedKeys(String catalog, String schema, String table)	ResultSet	Yes
getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate)	ResultSet	Yes
getMaxBinaryLiteralLength()	int	Yes
getMaxCatalogNameLengt h()	int	Yes
getMaxCharLiteralLength()	int	Yes
getMaxColumnNameLengt h()	int	Yes
getMaxColumnsInGroupBy()	int	Yes
getMaxColumnsInIndex()	int	Yes
getMaxColumnsInOrderBy()	int	Yes
getMaxColumnsInSelect()	int	Yes
getMaxColumnsInTable()	int	Yes
getMaxConnections()	int	Yes
getMaxCursorNameLength()	int	Yes
getMaxIndexLength()	int	Yes
getMaxLogicalLobSize()	default long	Yes
getMaxProcedureNameLeng th()	int	Yes
getMaxRowSize()	int	Yes
getMaxSchemaNameLengt h()	int	Yes
getMaxStatementLength()	int	Yes
getMaxStatements()	int	Yes
getMaxTableNameLength()	int	Yes
getMaxTablesInSelect()	int	Yes

getMaxUserNameLength()	int	Yes
getNumericFunctions()	String	Yes
getPrimaryKeys(String catalog, String schema, String table)	ResultSet	Yes
getPartitionTablePrimaryKey s(String catalog, String schema, String table)	ResultSet	Yes
getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)	ResultSet	Yes
getProcedures(String catalog, String schemaPattern, String procedureNamePattern)	ResultSet	Yes
getProcedureTerm()	String	Yes
getSchemas()	ResultSet	Yes
getSchemas(String catalog, String schemaPattern)	ResultSet	Yes
getSchemaTerm()	String	Yes
getSearchStringEscape()	String	Yes
getSQLKeywords()	String	Yes
getSQLStateType()	int	Yes
getStringFunctions()	String	Yes
getSystemFunctions()	String	Yes
getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern)	ResultSet	Yes
getTimeDateFunctions()	String	Yes
getTypeInfo()	ResultSet	Yes
getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)	ResultSet	Yes

getURL()	String	Yes
getVersionColumns(String catalog, String schema, String table)	ResultSet	Yes
insertsAreDetected(int type)	boolean	Yes
locatorsUpdateCopy()	boolean	Yes
othersDeletesAreVisible(int type)	boolean	Yes
othersInsertsAreVisible(int type)	boolean	Yes
othersUpdatesAreVisible(int type)	boolean	Yes
ownDeletesAreVisible(int type)	boolean	Yes
ownInsertsAreVisible(int type)	boolean	Yes
ownUpdatesAreVisible(int type)	boolean	Yes
storesLowerCaseIdentifiers()	boolean	Yes
storesMixedCaseIdentifiers()	boolean	Yes
storesUpperCaseIdentifiers()	boolean	Yes
supportsBatchUpdates()	boolean	Yes
supportsCatalogsInDataMani pulation()	boolean	Yes
supportsCatalogsInIndexDefi nitions()	boolean	Yes
supportsCatalogsInPrivilege Definitions()	boolean	Yes
supportsCatalogsInProcedure Calls()	boolean	Yes
supportsCatalogsInTableDefi nitions()	boolean	Yes
getTypeInfo()	ResultSet	Yes
getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)	ResultSet	Yes
getURL()	String	Yes

getVersionColumns(String catalog, String schema, String table)	ResultSet	Yes
insertsAreDetected(int type)	boolean	Yes
locatorsUpdateCopy()	boolean	Yes
othersDeletesAreVisible(int type)	boolean	Yes
othersInsertsAreVisible(int type)	boolean	Yes
othersUpdatesAreVisible(int type)	boolean	Yes
ownDeletesAreVisible(int type)	boolean	Yes
ownInsertsAreVisible(int type)	boolean	Yes
ownUpdatesAreVisible(int type)	boolean	Yes
storesLowerCaseIdentifiers()	boolean	Yes
storesMixedCaseIdentifiers()	boolean	Yes
storesUpperCaseIdentifiers()	boolean	Yes
supportsBatchUpdates()	boolean	Yes
supportsCatalogsInDataMani pulation()	boolean	Yes
supportsCatalogsInIndexDefi nitions()	boolean	Yes
supportsCatalogsInPrivilege Definitions()	boolean	Yes
supportsCatalogsInProcedure Calls()	boolean	Yes
supportsCatalogsInTableDefi nitions()	boolean	Yes
supportsCorrelatedSubquerie s()	boolean	Yes
supportsDataDefinitionAndD ataManipulationTransaction s()	boolean	Yes
supportsDataManipulationTr ansactionsOnly()	boolean	Yes
supportsGetGeneratedKeys()	boolean	Yes

supportsMixedCaseIdentifier s()	boolean	Yes
supportsMultipleOpenResult s()	boolean	Yes
supportsNamedParameters()	boolean	Yes
supportsOpenCursorsAcrossC ommit()	boolean	Yes
supportsOpenCursorsAcrossR ollback()	boolean	Yes
supportsOpenStatementsAcr ossCommit()	boolean	Yes
supportsOpenStatementsAcr ossRollback()	boolean	Yes
supportsPositionedDelete()	boolean	Yes
supportsPositionedUpdate()	boolean	Yes
supportsRefCursors()	boolean	Yes
supportsResultSetConcurrenc y(int type, int concurrency)	boolean	Yes
supportsResultSetType(int type)	boolean	Yes
supportsSchemasInIndexDefi nitions()	boolean	Yes
supportsSchemasInPrivilege Definitions()	boolean	Yes
supportsSchemasInProcedur eCalls()	boolean	Yes
supportsSchemasInTableDefi nitions()	boolean	Yes
supportsSelectForUpdate()	boolean	Yes
supportsStatementPooling()	boolean	Yes
supportsStoredFunctionsUsin gCallSyntax()	boolean	Yes
supportsStoredProcedures()	boolean	Yes
supportsSubqueriesInCompa risons()	boolean	Yes
supportsSubqueriesInExists()	boolean	Yes

supportsSubqueriesInIns()	boolean	Yes
supportsSubqueriesInQuantif ieds()	boolean	Yes
supportsTransactionIsolation Level(int level)	boolean	Yes
supportsTransactions()	boolean	Yes
supportsUnion()	boolean	Yes
supportsUnionAll()	boolean	Yes
updatesAreDetected(int type)	boolean	Yes
getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)	ResultSet	Yes
getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)	ResultSet	Yes
getTableTypes()	ResultSet	Yes
getUserName()	String	Yes
isReadOnly()	boolean	Yes
nullsAreSortedHigh()	boolean	Yes
nullsAreSortedLow()	boolean	Yes
nullsAreSortedAtStart()	boolean	Yes
nullsAreSortedAtEnd()	boolean	Yes
getDatabaseProductName()	String	Yes
getDatabaseProductVersion( )	String	Yes
getDriverName()	String	Yes
getDriverVersion()	String	Yes

getDriverMajorVersion()	int	Yes
getDriverMinorVersion()	int	Yes
usesLocalFiles()	boolean	Yes
usesLocalFilePerTable()	boolean	Yes
supportsMixedCaseIdentifier s()	boolean	Yes
storesUpperCaseIdentifiers()	boolean	Yes
storesLowerCaseIdentifiers()	boolean	Yes
supportsMixedCaseQuotedId entifiers()	boolean	Yes
storesUpperCaseQuotedIden tifiers()	boolean	Yes
storesLowerCaseQuotedIden tifiers()	boolean	Yes
storesMixedCaseQuotedIden tifiers()	boolean	Yes
supportsAlterTableWithAddC olumn()	boolean	Yes
supportsAlterTableWithDrop Column()	boolean	Yes
supportsColumnAliasing()	boolean	Yes
nullPlusNonNullIsNull()	boolean	Yes
supportsConvert()	boolean	Yes
supportsConvert(int fromType, int toType)	boolean	Yes
supportsTableCorrelationNa mes()	boolean	Yes
supportsDifferentTableCorrel ationNames()	boolean	Yes
supportsExpressionsInOrderB y()	boolean	Yes
supportsOrderByUnrelated()	boolean	Yes
supportsGroupBy()	boolean	Yes
supportsGroupByUnrelated()	boolean	Yes

supportsGroupByBeyondSele ct()	boolean	Yes
supportsLikeEscapeClause()	boolean	Yes
supportsMultipleResultSets()	boolean	Yes
supportsMultipleTransaction s()	boolean	Yes
supportsNonNullableColumn s()	boolean	Yes
supportsMinimumSQLGram mar()	boolean	Yes
supportsCoreSQLGrammar()	boolean	Yes
supportsExtendedSQLGram mar()	boolean	Yes
supportsANSI92EntryLevelS QL()	boolean	Yes
supportsANSI92Intermediate SQL()	boolean	Yes
supportsANSI92FullSQL()	boolean	Yes
supportsIntegrityEnhanceme ntFacility()	boolean	Yes
supportsOuterJoins()	boolean	Yes
supportsFullOuterJoins()	boolean	Yes
supportsLimitedOuterJoins()	boolean	Yes
isCatalogAtStart()	boolean	Yes
supportsSchemasInDataMan ipulation()	boolean	Yes
supportsSavepoints()	boolean	Yes
supportsResultSetHoldabilit y(int holdability)	boolean	Yes
getResultSetHoldability()	int	Yes
getDatabaseMajorVersion()	int	Yes
getDatabaseMinorVersion()	int	Yes
getJDBCMajorVersion()	int	Yes

getJDBCMinorVersion()	int	Yes

说明

uppercaseAttributeName 为 true 时，以下接口会将查询结果转为大写，可转换范围与 java 中的 toUpperCase 方法一致。

- public ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern)
- public ResultSet getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)
- public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)
- public ResultSet getSchemas(String catalog, String schemaPattern)
- public ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)
- public ResultSet getColumnPrivileges(String catalog, String schema, String table, String columnNamePattern)
- public ResultSet getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern)
- public ResultSet getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean nullable)
- public ResultSet getPrimaryKeys(String catalog, String schema, String table)
- protected ResultSet getImportedExportedKeys(String primaryCatalog, String primarySchema, String primaryTable, String foreignCatalog, String foreignSchema, String foreignTable)
- public ResultSet getIndexInfo(String catalog, String schema, String tableName, boolean unique, boolean approximate)
- public ResultSet getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)
- public ResultSet getFunctions(String catalog, String schemaPattern, String functionNamePattern)

⚠ 注意

getPartitionTablePrimaryKeys(String catalog, String schema, String table)接口用于获取分区表含全局索引的主键列，使用示例如下：

```
PgDatabaseMetaData dbmd = (PgDatabaseMetaData)conn.getMetaData();
dbmd.getPartitionTablePrimaryKeys("catalogName", "schemaName", "tableName");
```

3.10.4 java.sql.Driver

java.sql.Driver 是数据库驱动接口。

表 3-10 对 java.sql.Driver 的支持情况

方法名	返回值类型	支持 JDBC 4
acceptsURL(String url)	Boolean	Yes
connect(String url, Properties info)	Connection	Yes
jdbcCompliant()	Boolean	Yes
getMajorVersion()	int	Yes
getMinorVersion()	int	Yes
getParentLogger()	Logger	Yes

3.10.5 java.sql.PreparedStatement

java.sql.PreparedStatement 是预处理数据接口。

表 3-11 对 java.sql.PreparedStatement 的支持情况

方法名	返回值类型	支持 JDBC 4
clearParameters()	void	Yes
execute()	Boolean	Yes
executeQuery()	ResultSet	Yes
excuteUpdate()	int	Yes

executeLargeUpdate()	long	No
getMetaData()	ResultSetMetaData	Yes
getParameterMetaData()	ParameterMetaData	Yes
setArray(int parameterIndex, Array x)	void	Yes
setAsciiStream(int parameterIndex, InputStream x, int length)	void	Yes
setBinaryStream(int parameterIndex, InputStream x)	void	Yes
setBinaryStream(int parameterIndex, InputStream x, int length)	void	Yes
setBinaryStream(int parameterIndex, InputStream x, long length)	void	Yes
setBlob(int parameterIndex, InputStream inputStream)	void	Yes
setBlob(int parameterIndex, InputStream inputStream, long length)	void	Yes
setBlob(int parameterIndex, Blob x)	void	Yes
setCharacterStream(int parameterIndex, Reader reader)	void	Yes
setCharacterStream(int parameterIndex, Reader reader, int length)	void	Yes
setClob(int parameterIndex, Reader reader)	void	Yes
setClob(int parameterIndex, Reader reader, long length)	void	Yes
setClob(int parameterIndex, Clob x)	void	Yes
setDate(int parameterIndex, Date x, Calendar cal)	void	Yes

setNull(int parameterIndex, int sqlType)	void	Yes
setNull(int parameterIndex, int sqlType, String typeName)	void	Yes
setObject(int parameterIndex, Object x)	void	Yes
setObject(int parameterIndex, Object x, int targetSqlType)	void	Yes
setObject(int parameterIndex, Object x, int targetSqlType, int scaleOrLength)	void	Yes
setSQLXML(int parameterIndex, SQLXML xmlObject)	void	Yes
setTime(int parameterIndex, Time x)	void	Yes
setTime(int parameterIndex, Time x, Calendar cal)	void	Yes
setTimestamp(int parameterIndex, Timestamp x)	void	Yes
setTimestamp(int parameterIndex, Timestamp x, Calendar cal)	void	Yes
setUnicodeStream(int parameterIndex, InputStream x, int length)	void	Yes
setURL(int parameterIndex, URL x)	void	Yes
setBoolean(int parameterIndex, boolean x)	void	Yes
setBigDecimal(int parameterIndex, BigDecimal x)	void	Yes
setByte(int parameterIndex, byte x)	void	Yes
setBytes(int parameterIndex, byte[] x)	void	Yes
setDate(int parameterIndex, Date x)	void	Yes
setDouble(int parameterIndex, double x)	void	Yes

setFloat(int parameterIndex, float x)	void	Yes
setInt(int parameterIndex, int x)	void	Yes
setLong(int parameterIndex, long x)	void	Yes
setShort(int parameterIndex, short x)	void	Yes
setString(int parameterIndex, String x)	void	Yes
setNString(int parameterIndex, String x)	void	Yes
addBatch()	void	Yes
executeBatch()	int[]	Yes

说明

- addBatch()、execute()必须在 clearBatch()之后才能执行。
- 调用 executeBatch()方法并不会清除 batch。用户必须显式使用 clearBatch()清除 。
- 在添加了一个 batch 的绑定变量后，用户若想重用这些值(再次添加一个 batch)，无需再次使用 set*()方法 。
- 以下方法是从 java.sql.Statement 继承而来： close， execute， executeQuery，executeUpdate, getConnection, getResultSet, getUpdateCount, isClosed， setMaxRows, setFetchSize。
- executeLargeUpdate()方法必须在 JDBC4.2 及以上使用。

3.10.6java.sql.ResultSet

java.sql.ResultSet 是执行结果集接口。

表 3-12 对 java.sql.ResultSet 的支持情况

方法名	返回值类型	支持 JDBC 4
absolute(int row)	Boolean	Yes
afterLast()	void	Yes
beforeFirst()	void	Yes

cancelRowUpdates()	void	Yes
clearWarnings()	void	Yes
close()	void	Yes
deleteRow()	void	Yes
findColumn(String columnLabel)	int	Yes
first()	Boolean	Yes
getArray(int columnIndex)	Array	Yes
getArray(String columnLabel)	Array	Yes
getAsciiStream(int columnIndex)	InputStream	Yes
getAsciiStream(String columnLabel)	InputStream	Yes
getBigDecimal(int columnIndex)	BigDecimal	Yes
getBigDecimal(String columnLabel)	BigDecimal	Yes
getBinaryStream(int columnIndex)	InputStream	Yes
getBinaryStream(String columnLabel)	InputStream	Yes
getBlob(int columnIndex)	Blob	Yes
getBoolean(int columnIndex)	Boolean	Yes
getBoolean(String columnLabel)	Boolean	Yes
getByte(int columnIndex)	byte	Yes
getBytes(int columnIndex)	byte[]	Yes

getByte(String columnLabel)	byte	Yes
getBytes(String columnLabel)	byte[]	Yes
getCharacterStream(int columnIndex)	Reader	Yes
getCharacterStream (String columnLabel)	Reader	Yes
getClob(int columnIndex)	Clob	Yes
getClob(String columnLabel)	Clob	Yes
getConcurrency()	int	Yes
getCursorName()	String	Yes
getDate(int columnIndex)	Date	Yes
getDate(int columnIndex, Calendar cal)	Date	Yes
getDate(String columnLabel)	Date	Yes
getDate(String columnLabel, Calendar cal)	Date	Yes
getDouble(int columnIndex)	double	Yes
getDouble(String columnLabel)	double	Yes
getFetchDirection()	int	Yes
getFetchSize()	int	Yes
getFloat(int columnIndex)	float	Yes
getFloat(String columnLabel)	float	Yes
getInt(int columnIndex)	int	Yes
getInt(String columnLabel)	int	Yes

getLong(int columnIndex)	long	Yes
getLong(String columnLabel)	long	Yes
getMetaData()	ResultSetMetaData	Yes
getObject(int columnIndex)	Object	Yes
getObject(int columnIndex, Class<T> type)	<T> T	Yes
getObject(int columnIndex, Map<String, Class<?>> map)	Object	Yes
getObject(String columnLabel)	Object	Yes
getObject(String columnLabel, Class<T> type)	<T> T	Yes
getObject(String columnLabel, Map<String, Class<?>> map)	Object	Yes
getRow()	int	Yes
getShort(int columnIndex)	short	Yes
getShort(String columnLabel)	short	Yes
getSQLXML(int columnIndex)	SQLXML	Yes
getSQLXML(String columnLabel)	SQLXML	Yes
getStatement()	Statement	Yes
getString(int columnIndex)	String	Yes
getString(String columnLabel)	String	Yes
getNString(int columnIndex)	String	Yes
getNString(String columnLabel)	String	Yes
getTime(int columnIndex)	Time	Yes

getTime(int columnIndex, Calendar cal)	Time	Yes
getTime(String columnLabel)	Time	Yes
getTime(String columnLabel, Calendar cal)	Time	Yes
getTimestamp(int columnIndex)	Timestamp	Yes
getTimestamp(int columnIndex, Calendar cal)	Timestamp	Yes
getTimestamp(String columnLabel)	Timestamp	Yes
getTimestamp(String columnLabel, Calendar cal)	Timestamp	Yes
getType()	int	Yes
getWarnings()	SQLWarning	Yes
insertRow()	void	Yes
isAfterLast()	Boolean	Yes
isBeforeFirst()	Boolean	Yes
isClosed()	Boolean	Yes
isFirst()	Boolean	Yes
isLast()	Boolean	Yes
last()	Boolean	Yes
moveToCurrentRow()	void	Yes
moveToInsertRow()	void	Yes
next()	Boolean	Yes
previous()	Boolean	Yes

refreshRow()	void	Yes
relative(int rows)	Boolean	Yes
rowDeleted()	Boolean	Yes
rowInserted()	Boolean	Yes
rowUpdated()	Boolean	Yes
setFetchDirection(int direction)	void	Yes
setFetchSize(int rows)	void	Yes
updateArray(int columnIndex, Array x)	void	Yes
updateArray(String columnLabel, Array x)	void	Yes
updateAsciiStream(int columnIndex, InputStream x, int length)	void	Yes
updateAsciiStream(String columnLabel, InputStream x, int length)	void	Yes
updateBigDecimal(int columnIndex, BigDecimal x)	void	Yes
updateBigDecimal(String columnLabel, BigDecimal x)	void	Yes
updateBinaryStream(int columnIndex, InputStream x, int length)	void	Yes
updateBinaryStream (String columnLabel, InputStream x, int length)	void	Yes
updateBoolean(int columnIndex,	void	Yes

boolean x)		
updateBoolean(String columnLabel, boolean x)	void	Yes
updateByte(int columnIndex, byte x)	void	Yes
updateByte(String columnLabel, byte x)	void	Yes
updateBytes(int columnIndex, byte[] x)	void	Yes
updateBytes(String columnLabel, byte[] x)	void	Yes
updateCharacterStream (int columnIndex, Reader x, int length)	void	Yes
updateCharacterStream (String columnLabel, Reader reader, int length)	void	Yes
updateDate(int columnIndex, Date x)	void	Yes
updateDate(String columnLabel, Date x)	void	Yes
updateDouble(int columnIndex, double x)	void	Yes
updateDouble(String columnLabel, double x)	void	Yes
updateFloat(int columnIndex, float x)	void	Yes
updateFloat(String columnLabel, float x)	void	Yes

updateInt(int columnIndex, int x)	void	Yes
updateInt(String columnLabel, int x)	void	Yes
updateLong(int columnIndex, long x)	void	Yes
updateLong(String columnLabel, long x)	void	Yes
updateNull(int columnIndex)	void	Yes
updateNull(String columnLabel)	void	Yes
updateObject(int columnIndex, Object x)	void	Yes
updateObject(int columnIndex, Object x, int scaleOrLength)	void	Yes
updateObject(String columnLabel, Object x)	void	Yes
updateObject(String columnLabel, Object x, int scaleOrLength)	void	Yes
updateRow()	void	Yes
updateShort(int columnIndex, short x)	void	Yes
updateShort(String columnLabel, short x)	void	Yes
updateSQLXML(int columnIndex, SQLXML xmlObject)	void	Yes
updateSQLXML(String columnLabel, SQLXML	void	Yes

xmlObject)		
updateString(int columnIndex, String x)	void	Yes
updateString(String columnLabel, String x)	void	Yes
updateTime(int columnIndex, Time x)	void	Yes
updateTime(String columnLabel, Time x)	void	Yes
updateTimestamp(int columnIndex, Timestamp x)	void	Yes
updateTimestamp(String columnLabel, Timestamp x)	void	Yes
wasNull()	Boolean	Yes

说明

- 一个 Statement 不能有多个处于“open”状态的 ResultSet。
- 用于遍历结果集（ResultSet）的游标（Cursor）在被提交后不能保持“open”的状态。

3.10.7 java.sql.ResultSetMetData

java.sql.ResultSetMetaData 是对 ResultSet 对象相关信息的具体描述。

表 3-13 对 java.sql.ResultSetMetData 的支持情况

方法名	返回值类型	支持 JDBC 4
getCatalogName(int column)	String	Yes
getColumnClassName(int column)	String	Yes
getColumnCount()	int	Yes
getColumnDisplaySize(int column)	int	Yes
getColumnLabel(int column)	String	Yes

getColumnName(int column)	String	Yes
getColumnType(int column)	int	Yes
getColumnTypeName(int column)	String	Yes
getPrecision(int column)	int	Yes
getScale(int column)	int	Yes
getSchemaName(int column)	String	Yes
getTableName(int column)	String	Yes
isAutoIncrement(int column)	boolean	Yes
isCaseSensitive(int column)	boolean	Yes
isCurrency(int column)	boolean	Yes
isDefinitelyWritable(int column)	boolean	Yes
isNullable(int column)	int	Yes
isReadOnly(int column)	boolean	Yes
isSearchable(int column)	boolean	Yes
isSigned(int column)	boolean	Yes
isWritable(int column)	boolean	Yes

说明

uppercaseAttributeName 为 true 时，下面接口会将查询结果转为大写，可转换范围为 26 个英文字母。

- public String getColumnName(int column)
- public String getColumnLabel(int column)

3.10.8java.sql.Statement

java.sql.Statement 是 SQL 语句接口。

表 3-14 对 java.sql.Statement 的支持情况

方法名	返回值类型	支持 JDBC 4
addBatch(String sql)	void	Yes
clearBatch()	void	Yes
clearWarnings()	void	Yes
close()	void	Yes
closeOnCompletion()	void	Yes
execute(String sql)	Boolean	Yes
execute(String sql, int autoGeneratedKeys)	Boolean	Yes
execute(String sql, int[] columnIndexes)	Boolean	Yes
execute(String sql, String[] columnNames)	Boolean	Yes
executeBatch()	Boolean	Yes
executeQuery(String sql)	ResultSet	Yes
executeUpdate(String sql)	int	Yes
executeUpdate(String sql, int autoGeneratedKeys)	int	Yes
executeUpdate(String sql, int[] columnIndexes)	int	Yes
executeUpdate(String sql, String[] columnNames)	int	Yes
getConnection()	Connection	Yes
getFetchDirection()	int	Yes
getFetchSize()	int	Yes
getGeneratedKeys()	ResultSet	Yes

getMaxFieldSize()	int	Yes
getMaxRows()	int	Yes
getMoreResults()	boolean	Yes
getMoreResults(int current)	boolean	Yes
getResultSet()	ResultSet	Yes
getResultSetConcurrenc y()	int	Yes
getResultSetHoldability( )	int	Yes
getResultSetType()	int	Yes
getQueryTimeout()	int	Yes
getUpdateCount()	int	Yes
getWarnings()	SQLWarning	Yes
isClosed()	Boolean	Yes
isCloseOnCompletion()	Boolean	Yes
isPoolable()	Boolean	Yes
setCursorName(String name)	void	Yes
setEscapeProcessing (boolean enable)	void	Yes
setFetchDirection(int direction)	void	Yes
setMaxFieldSize(int max)	void	Yes
setMaxRows(int max)	void	Yes
setPoolable(boolean poolable)	void	Yes
setQueryTimeout(int seconds)	void	Yes
setFetchSize(int rows)	void	Yes

cancel()	void	Yes
executeLargeUpdate(Str ing sql)	long	No
getLargeUpdateCount()	long	No
executeLargeBatch()	long	No
executeLargeUpdate(Str ing sql, int autoGeneratedKeys)	long	No
executeLargeUpdate(Str ing sql, int[] columnIndexes)	long	No
executeLargeUpdate(Str ing sql, String[] columnNames)	long	No

说明

- 通过 setFetchSize 可以减少结果集在客户端的内存占用情况。它的原理是通过将结果集打包 成游标，然后分段处理，所以会加大数据库与客户端的通信量，会有性能损耗。
- 由于数据库游标是事务内有效，所以，在设置 setFetchSize 的同时，需要将连接设置为非自动提交模式，setAutoCommit(false)。同时在业务数据需要持久化到数据库中时，在连接上执行提交操作。
- LargeUpdate 相关方法必须在 JDBC4.2 及以上使用。

3.10.9 javax.sql.ConnectionPoolDataSource

javax.sql.ConnectionPoolDataSource 是数据源连接池接口。

表 3-15 对 javax.sql.ConnectionPoolDataSource 的支持情况

方法名	返回值类型	支持 JDBC 4
getPooledConnection()	PooledConnection	Yes
getPooledConnection(String user,String password)	PooledConnection	Yes

3.10.10javax.sql.DataSource

java.sql.DataSource 是数据源接口。

表 3-16 对 java.sql.DataSource 的支持情况

方法名	返回值类型	支持 JDBC 4
getConneciton()	Connection	Yes
getConnection(String username,String password)	Connection	Yes
getLoginTimeout()	int	Yes
getLogWriter()	PrintWriter	Yes
setLoginTimeout(int seconds)	void	Yes
setLogWriter(PrintWriter out)	void	Yes

3.10.11javax.sql.PooledConnection

javax.sql.PooledConnection 是由连接池创建的连接接口。

表 3-17 对 javax.sql.PooledConnection 的支持情况

方法名	返回值类型	支持 JDBC 4
addConnectionEventListener (ConnectionEventListener listener)	void	Yes
close()	void	Yes
getConnection()	Connection	Yes
removeConnectionEventListener (ConnectionEventListener listener)	void	Yes

3.10.12javax.naming.Context

javax.naming.Context 是连接配置的上下文接口。

表 3-18 对 javax.naming.Context 的支持情况

方法名	返回值类型	支持 JDBC 4
bind(Name name, Object obj)	void	Yes
bind(String name, Object obj)	void	Yes
lookup(Name name)	Object	Yes
lookup(String name)	Object	Yes
rebind(Name name, Object obj)	void	Yes
rebind(String name, Object obj)	void	Yes
rename(Name oldName, Name newName)	void	Yes
rename(String oldName, String newName)	void	Yes
unbind(Name name)	void	Yes
unbind(String name)	void	Yes

3.10.13javax.naming.spi.InitialContextFactory

javax.naming.spi.InitialContextFactory 是初始连接上下文工厂接口。

表 3-19 对 javax.naming.spi.InitialContextFactory 的支持情况

方法名	返回值类型	支持 JDBC 4
getInitialContext(Hashtable<?, ?> environment)	Context	Yes

3.10.14CopyManager

CopyManager 是 GBase 8s 的 JDBC 驱动中提供的一个 API 接口类，用于批量向 GBase 8s 数据库中导入数据。

CopyManager 的继承关系

CopyManager 类位于 org.postgresql.copy Package 中，继承自 java.lang.Object 类，该类的

声明如下：

```
public class CopyManager
extends Object
```

构造方法

```
public CopyManager(BaseConnection connection)
throws SQLException
```

常用方法

表 3-20 CopyManager 常用方法

返回值	方法	描述	throws
CopyIn	copyIn(String sql)	——	SQLException
long	copyIn(String sql, InputStream from)	使用 COPY FROM STDIN 从 InputStream 中快速向数据库中的表加载数据。	SQLException,IOException
long	copyIn(String sql, InputStream from, int bufferSize)	使用 COPY FROM STDIN 从 InputStream 中快速向数据库中的表加载数据。	SQLException,IOException
long	copyIn(String sql, Reader from)	使用 COPY FROM STDIN 从 Reader 中快速向数据库中的表加载数据。	SQLException,IOException
long	copyIn(String sql, Reader from, int bufferSize)	使用 COPY FROM STDIN 从 Reader 中快速向数据库中的表加载数据。	SQLException,IOException
CopyOu t	copyOut(String sql)	——	SQLException
long	copyOut(String sql, OutputStream to)	将一个 COPY TO TDOUT 的结果集从数据库发送到 OutputStream 类中。	SQLException,IOException
long	copyOut(String sql, Writer to)	将一个 COPY TO TDOUT 的结果集从数据库发送到 Writer 类中。	SQLException,IOException

3.10.15PGReplicationConnection

PGReplicationConnection 是 GBase 8s 的 JDBC 驱动中提供的一个 API 接口类，用于执行逻辑复制相关的功能。

PGReplicationConnection 的继承关系

PGReplicationConnection 是逻辑复制的接口，实现类是 PGReplicationConnectionImpl，该类位于 org.postgresql.replication Package 中，该类的声明如下：

```
public class PGReplicationConnection implements PGReplicationConnection
```

构造方法

```
public PGReplicationConnection(BaseConnection connection)
```

常用方法

返回值	方法	描述	throws
ChainedCreate ReplicationSlot Builder	createReplicationSlot( )	用于创建逻辑复制槽	——
void	dropReplicationSlot(S tring slotName)	用于删除逻辑复制槽	SQLException,IOE xception
ChainedStream Builder	replicationStream()	用户开启逻辑复制	——

3.11 JDBC 常用参数参考

- targetServerType

  原理：值为 master 时会依次尝试连接串中配置的 ip，直到能够连接到集群中的主机，

  值为 slave 时会依次尝试连接串中配置的 ip，直到能够连接到集群中的备机（查询语句为：select local_role, db_state from pg_stat_get_stream_replications();）。

  建议：有写操作的业务建议配置 master，以保证主备切换后能正常连接主机，但是要注意在主备倒换过程中备机没有完全升主的时候无法正常建连，导致业务语句无法正常执行。

- hostRecheckSeconds

  原理：JDBC 内部存储的 dn 列表保持可信的时间，未超过此时间时会从中直接读取存

储的主机地址，当超过此时间时或者在可信时间内连接主机失败时会通过更新 dn 列表中该节点状态，之后连接其他的 ip。

建议：默认值 10s，建议根据业务进行调整，配合参数 targetServerType 使用。

- allowReadOnly

原理：是否可以通过 setReadOnly 来修改事务访问模式，如果为 true 则可以修改，如果为 false 则无法通过此接口来修改，修改语句为 SET SESSION CHARACTERISTICS AS TRANSACTION + READ ONLY / READ WEITE。

建议：保持默认值为 true。

- fetchsize

原理：fetchsize 在设置为 n 后，数据库服务器端在执行查询后，调用者在执行 resultset.next()的时候，JDBC 会先与服务器端进行通信，取 n 条数据到 jdbc 的客户端中，然后返回第一条给调用者，当调用者取到第 n+1 条数据的时候，会再次到数据库服务端去拿数据。

作用：避免了数据库一下把所有结果全部传输到客户端来，将客户端的内存资源撑爆掉。

建议：建议根据自身的业务查询数据数量和客户端机器内存情况来配置此参数，设置 fetchsize 时要关闭自动提交(autocommit=false)，否则会导致 fetchsize 无法生效。

- defaultRowFetchSize

作用：fetchsize 默认值为 0，defaultRowFetchSize 会修改 fetchsize 的默认值。

- batchMode

作用：用于确定是否使用 batch 模式连接。默认值为 on，开启后可以提升批量更新的性能，同时批量更新的返回值会发生改变，例如，批量插入三条数据，在开启时返回值为[3,0,0]，在关闭后返回值为[1,1,1]。

建议：如果本身业务框架(例如 hibernate)在批量更新时会检测返回值，可以通过调整此参数来解决。

- loginTimeout

作用：控制与数据库建联时间，其中时间包括 connectiontimeout 和 sockettimeout，超过阈值则退出。计算方式为：loginTimeout=connectiontimeout*节点数量+连接认证时间+初始化语句执行时间。

建议：配置后会每次建连都会开启一个异步线程，在连接数较多的情况可能会导致客户端压力增大，如果业务确认需要此设置此参数，需要注意在集中式下建议调整为3*connectTimeout 防止在网络异常情况且第三个 IP 为主的情况下，无法连接。


须知： 此参数设置后对于多 ip 而言，时间是尝试连接 ip 的时间，可能会出现因为设置的值较小导致后面的 ip 无法连接的问题，例如设置了三个 ip，如果 logintimeout 为 5s，但前两个 ip 建连总共用了 5s，第三个 ip 会无法进行连接，在集中式环境下，此最后一个 ip 恰好为主机，可能会导致自动寻主失败。

- cancelSignalTimeout

作用：发送取消消息本身可能会阻塞，此属性控制用于取消命令的“connect 超时”和“socket 超时”。超时时间单位为秒。主要为了防止连接超时取消时本身执行超时检测。

建议：默认值为 10 秒，建议根据业务进行调整。

- connectTimeout

作用：控制建立连接时套接字超时阈值（此时是 jdbc 通过 socket 连接到数据的时间，并不是返回 connection 对象的时间），超过阈值查找下一个 IP。

建议：该参数决定了每个节点 TCP 连接建立的最大超时时间，如果某节点网络故障，与该节点建立连接时会等待 connectTimeout 超时，然后尝试连接下一个节点。考虑到网络抖动，时延等情况，默认建议设置为 3s。

- socketTimeout

作用：控制套接字操作超时值，如果业务语句执行或者从网络读取数据流超过此阈值，连接中断（即语句超过规定时间执行，没有数据返回的时候）。

建议：该参数限制单 SQL 最长的执行时间，单语句执行超过该值则会超时报错退出，建议根据业务特征进行配置。

- socketTimeoutInConnecting

作用： 控制建连阶段套接字操作超时值，在建连阶段，如果从网络中读取数据流超过此阈值，则尝试查找下一个节点建连。

建议： 该参数仅影响建连阶段的 socket 超时时间，如果未配置，默认为 5s。

- autosave

作用：值为 always 时可以在事务中每个语句前面设置一个 savepoint 点，在事务中语句执行报错时会返回到最近的上一个 savepoint 点，可以让事务中后续语句可以正常执行，最终可以正常提交。

建议：不建议设置此参数，性能劣化严重。

- currentSchema

作用：设置当前连接的 schema，如果未设置，则默认 schema 为连接使用的用户名。

建议：建议配置此参数，业务数据所在的 schema。

- prepareThreshold

作用：默认值为 5，如果一个会话连续多次执行同一个 SQL，在达到 prepareThreshold 次数以上时，JDBC 将不再对这个 SQL 发送 parse 命令，会将其缓存起来，提升执行速度。

建议：默认值为 5，根据业务需要进行调整。

- preparedStatementCacheQueries

作用：确定每个连接中缓存的查询数，默认情况下是 256。若在 prepareStatement()调用中使用超过 256 个不同的查询，则最近最少使用的查询缓存将被丢弃。

建议：默认值为 256，根据业务需要进行调整。配合 prepareThreshold 使用。

- blobMode

作用：setBinaryStream 方法为不同类型的数据赋值，设置为 on 时表示为 blob 类型数据赋值，设置为 off 时表示为 bytea 类型数据赋值，默认为 on。例如在 preparestatement 和 callablestatement 对象中对参数进行赋值操作。

建议：默认值为 true。

- setAutocommit 方法

作用：值为 true 时，执行每个语句都会自动开启事务，在执行结束后自动提交事务，即每个语句都是一个事务。值为 false 时，会自动开启一个事务，事务需要通过执行 SQL 手动提交。

建议：根据业务特征进行调整，如果基于性能或者其它方面考虑，需要关闭 autocommit 时，需要应用程序自己来保证事务的提交。例如，在指定的业务 SQL 执行完之后做显式提交，特别是客户端退出之前务必保证所有的事务已经提交。

4 基于 ODBC 开发

ODBC（Open Database Connectivity，开放数据库互连）是由 Microsoft 公司基于 X/ OPEN CLI 提出的用于访问数据库的应用程序编程接口。应用程序通过 ODBC 提供的 API 与数据库进行交互，增强了应用程序的可移植性、扩展性和可维护性。

ODBC 的系统结构参见图 4-1。

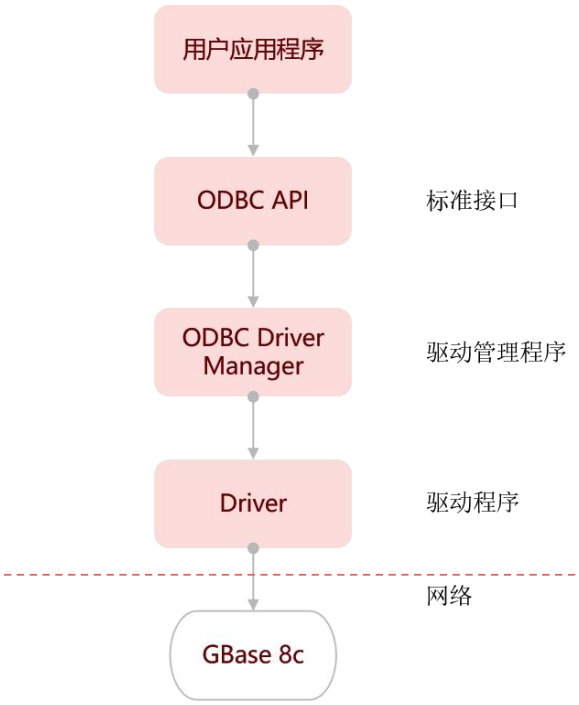

图 4-1 ODBC 系统机构

GBase 8s 目前在以下环境中提供对 ODBC 的支持。

表 4-1 ODBC 支持平台

操作系统	平台
CentOS 6.4/6.5/6.6/6.7/6.8/6.9/7.0/7.1/7.2/7.3/7.4	x86_64 位
CentOS 7.6	ARM64 位
EulerOS 2.0 SP2/SP3	x86_64 位
EulerOS 2.0 SP8	ARM64 位

UNIX/Linux 系统下的驱动程序管理器主要有 unixODBC 和 iODBC，在这选择驱动管理器 unixODBC-2.3.7 作为连接数据库的组件。

Windows 系统自带 ODBC 驱动程序管理器，在控制面板->管理工具中可以找到数据源（ODBC）选项。

说明

当前数据库 ODBC 驱动基于开源版本，对于 tinyint、smalldatetime、nvarchar、nvarchar2 类型，在获取数据类型的时候，可能会出现不兼容。

4.1 ODBC 包及依赖的库和头文件

Linux 下的 ODBC 包

获取发布包。Linux 环境下，开发应用程序要用到 unixODBC 提供的头文件（sql.h、sqlext.h 等）和库 libodbc.so。这些头文件和库可从 unixODBC-2.3.0 的安装包中获得。

4.2 Linux 下配置数据源

将 GBase 8s 提供的 ODBC DRIVER（psqlodbcw.so）配置到数据源中便可使用。配置数据源需要配置“odbc.ini”和“odbcinst.ini”两个文件（在编译安装 unixODBC 过程中生成且默认放在“/usr/local/etc”目录下），并在服务器端进行配置。

操作步骤

(1) 获取 unixODBC 源码包。

获取参考地址：https://sourceforge.net/projects/unixodbc/files/unixODBC/2.3.9/unixODBC-2.3.9pre.tar.gz/download

(2) 安装 unixODBC。如果机器上已经安装了其他版本的 unixODBC，可以直接覆盖安装。

目前不支持 unixODBC-2.2.1 版本。以 unixODBC-2.3.0 版本为例，在客户端执行如下命令安装 unixODBC。默认安装到“/usr/local”目录下，生成数据源文件到 “/usr/ local/etc”目录下，库文件生成在“/usr/local/lib”目录。

(3) 替换客户端驱动程序。

① 解压 tar 包。解压后会得到两个文件夹：lib 与 odbc，在 odbc 文件夹中还会有一个 lib 文件夹。/odbc/lib 中会有“psqlodbca.la”， “psqlodbca.so”，“psqlodbcw.la”和“psqlodbcw.so”四个文件，将这四个文件拷贝到“/usr/local/lib”目录下。

② 将 lib 目录中的库拷贝到“/usr/local/lib”目 录下。

(4) 配置数据源。

① 配置 ODBC 驱动文件。

在“/usr/local/etc/odbcinst.ini”文件中追加以下内容。

```
[GaussMPP] Driver64=/usr/local/lib/psqlodbcw.so setup=/usr/local/lib/psqlodbcw.so
```

odbcinst.ini 文件中的配置参数说明如下表所示。

表 4-2 odbcinst.ini 文件配置参数

参数	描述	示例
[DriverName]	驱动器名称，对应数据源 DSN 中的驱动名。	[DRIVER_N]
Driver64	驱动动态库的路径。	Driver64=/usr/local/lib/psqlodbcw.so
setup	驱动安装路径，与 Driver64 中动态库的路径一致。	setup=/usr/local/lib/psqlodbcw.so

② 配置数据源文件。

在“/usr/local/etc/odbc.ini”文件中追加以下内容。

```
[MPPODBC]
Driver=GaussMPP Servername=10.145.130.26（数据库 Server IP） Database=postgres （数据库名） Username=gbase （数据库用户名） Password= （数据库用户密码）
Port=15432 （数据库侦听端口）
Sslmode=allow
```

odbc.ini 文件配置参数说明如表 4-3 所示。

表 4-3 odbc.ini 文件配置参数

参数	描述	示例
[DSN]	数据源的名称。	[MPPODBC]
Driver	驱动名，对应 odbcinst.ini 中的 DriverName。	Driver=DRIVER_N

参数	描述	示例
Servername	服务器的 IP 地址。可配置多个 IP 地址。	Servername=10.145.130.2 6
Database	要连接的数据库的名称。	Database=postgres
Username	数据库用户名称。	Username=gbase
Password	数据库用户密码。	Password= 说明 ODBC 驱动本身已经对内存密码进行过清理，以保证用户密码在连接后不会再在内存中保留。 但是如果配置了此参数，由于 UnixODBC 对数据源文件等进行缓存，可能导致密码长期保留在内存中。 推荐在应用程序连接时，将密码传递给相应 API，而非写在数据源配置文件中。同时连接成功后，应当及时清理保存密码的内存段。
Port	服务器的端口号。	Port=15432
Sslmode	开启 SSL 模式	Sslmode=allow
Debug	设置为 1 时，将会打印 psqlodbc 驱动的 mylog，日志生成目录为 /tmp/。设置为 0 时则不会生成。	Debug=1
UseServerSidePrepare	是否开启数据库端扩展查询协议。 可选值 0 或 1，默认为 1， 表示打开扩展查询协议。	UseServerSidePrepare=1
UseBatchProtocol	是否开启批量查询协议（打开可提高 DML 性 能）；可选值 0 或者 1，默认为 1。 当此值为 0 时，不使用批量查询	UseBatchProtocol=1

参数	描述	示例
	协议（主要用于与早期数据库版本通信兼容）。 当此值为 1，并且数据库 support_batch_bind 参数存在且为 on 时，将打开批量查询协议。	
ForExtensionConnect or	这个开关控制着 savepoint 是否发送，savepoint 相关问题可以注意这个开关。	ForExtensionConnector=1
UnamedPrepStmtThr eshold	每次调用 SQLFreeHandle 释放 Stmt 时，ODBC 都会向 server 端发送一个 Deallocate plan_name 语句，业务中存在大量这类语句。为了减少这类语句的发送，我们将 stmt->plan_name 置空，从而使得数据库识别这个为 unamed stmt。增加这个参数对 unamed stmt 的阈值进行控制。	UnamedPrepStmtThresho ld=100
ConnectionExtraInfo	GUC 参数 connection_info（参见 connection_info）中显示驱动部署路径和进程属主用户的开关。	ConnectionExtraInfo=1 说明 默认值为 0。当设置为 1 时，ODBC 驱动会将当前驱动的部署路径、进程属主用户上报到数据库中，记录在 connection_info 参数（参见 connection_info）里；同时可以在 20.3.72 PG_STAT_ACTIVITY 中查询到。
BoolAsChar	设置为 Yes 是，Bools 值将会映射为 SQL_CHAR。如不设置将会映射为 SQL_BIT。	BoolsAsChar = Yes
RowVersioning	当尝试更新一行数据时，设置为 Yes 会允许应用检测数据有没有被其他用户进行修改。	RowVersioning=Yes
ShowSystemTables	驱动将会默认系统表格为普通	ShowSystemTables=Yes

参数	描述	示例
	SQL 表格。	

其中关于 Sslmode 的选项的允许值，具体信息见下表：

表 4-4 Sslmode 的可选项及其描述

Sslmode	是否会启用 SSL 加密	描述
disable	否	不使用 SSL 安全连接。
allow	可能	如果数据库服务器要求使用，则可以使用 SSL 安全加密连接，但不验证数据库服务器的真实性。
prefer	可能	如果数据库支持，那么建议使用 SSL 安全加密连接， 但不验证数据库服务器的真实性。
require	是	必须使用 SSL 安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。
verify-ca	是	必须使用 SSL 安全连接，并且验证数据库是否具有可信证书机构签发的证书。
verify- full	是	必须使用 SSL 安全连接，在 verify-ca 的验证范围之外，同时验证数据库所在主机的主机名是否与证书内容一致。GBase 8s 不支持此模式。

(5) （可选）生成 SSL 证书，具体请参见《GBase 8s V8.8.5_5.0.0_数据库管理指南》证书生成。在服务端与客户端通过 ssl 方式连接的情况下，需要执行步骤 5 或步骤 6。非 ssl 方式连接情况下可以跳过。

(6) （可选）替换 SSL 证书，具体请参见《GBase 8s V8.8.5_5.0.0_数据库管理指南》证书替换。

(7) SSL 模式：

声明如下环境变量，同时保证 client.key*系列文件为 600 权限：

```
退回根目录，创建.postgresql 目录，并将 root.crt，client.crt，client.key，client.key.cipher，client.key.rand，
client.req，server.crt，server.key，server.key.cipher，server.key.rand，server.req 放在此路径下。
```

Unix 系统下，server.crt、server.key 的权限设置必须禁止任何外部或组的访问，请执行如下命令实现这一点。chmod 0600 server.key
将 root.crt 以及 server 开头的证书相关文件全部拷贝进数据库 install/data 目录下（与 postgresql.conf 文件在同一路径）。
修改 postgresql.conf 文件：
ssl = on
ssl_cert_file = 'server.crt' ssl_key_file = 'server.key' ssl_ca_file = 'root.crt'
修改完参数后需重启数据库。
修改配置文件 odbc.ini 中的 sslmode 参数（require 或 verify-ca）。

(8) 配置数据库服务器。

① 以操作系统用户 gbase 登录数据库主节点。

② 执行如下命令增加对外提供服务的网卡 IP 或者主机名（英文逗号分隔），NODETYPE 指定节点类型，NodeName 为当前节点名称：

```
gs_guc reload -N NodeName -I all -c "listen_addresses='XXX,XX'"
```

在 DR（Direct Routing，LVS 的直接路由 DR 模式）模式中需要将虚拟 IP 地址（10.11.12.13）加入到服务器的侦听地址列表中。

listen_addresses 也可以配置为“*”或“0.0.0.0”，此配置下将侦听所有网卡，但 存在安全风险，不推荐用户使用，推荐用户按照需要配置 IP 或者主机名，打开侦听。

③ 执行如下命令在数据库主节点配置文件中增加一条认证规则。例如，假设客户端 IP 地址为 10.11.12.13，即远程连接的机器的 IP 地址，远程连接主备式集群为例。

```
gs_guc reload -N all -I all -h "host all jack 10.11.12.13/32 sha256"
```

- -N all 表示 GBase 8s 中的所有主机。
- -I all 表示主机中的所有实例。
- -h 表示指定需要在“pg_hba.conf”增加的语句。
- all 表示允许客户端连接到任意的数据库。
- jack 表示连接数据库的用户。
- 10.11.12.13/32 表示只允许 IP 地址为 10.11.12.13 的主机连接。在使用过程中，请根据用户的网络进行配置修改。32 表示子网掩码为 1 的位数，即 255.255.255.255
- sha256 表示连接时 jack 用户的密码使用 sha256 算法加密。

如果将 ODBC 客户端配置在和要连接的数据库主节点在同一台机器上，则可使用 local

trust 认证方式，如下：

```
local all all trust
```

如果将ODBC客户端配置在和要连接的数据库主节点在不同机器上，则需要使用 sha256 认证方式，如下：

```
host all all xxx.xxx.xxx.xxx/32 sha256
```

④ 重启数据库。

```
gha_ctl stop all -l http://<dcs_ip>:2379
gha_ctl start all -l http://<dcs_ip>:2379
```

(9) 在客户端配置环境变量。

```
vim ~/.bashrc
```

在配置文件中追加以下内容。

```
export LD_LIBRARY_PATH=/usr/local/lib/:$LD_LIBRARY_PATH export
ODBCSYSINI=/usr/local/etc
export ODBCINI=/usr/local/etc/odbc.ini
```

(10) 执行如下命令使设置生效。

```
source ~/.bashrc
```

测试数据源配置

执行./isql -v MPPODBC（数据源名称）命令。

- 如果显示如下信息，表明配置正确，连接成功。

```
+---------------------------------------+
| Connected!
|
| sql-statement
| help [tablename]
| quit
|
+---------------------------------------+ SQL>
```

- 若显示 ERROR 信息，则表明配置错误。请检查上述配置是否正确。

常见问题处理

- [UnixODBC][Driver Manager]Can't open lib 'xxx/xxx/psqlodbcw.so' : file not found.

此问题的可能原因：

- odbcinst.ini 文件中配置的路径不正确

确认的方法：'ls'一下错误信息中的路径，以确保该 psqlodbcw.so 文件存在， 同时具有执行权限。

- psqlodbcw.so 的依赖库不存在，或者不在系统环境变量中

确认的办法：ldd 一下错误信息中的路径，如果是缺少 libodbc.so.1 等 UnixODBC 的库，那么按照“操作步骤”中的方法重新配置 UnixODBC，并确保它的安装路径下的 lib 目录添加到了 LD_LIBRARY_PATH 中；如果是缺少其他库，请将 ODBC 驱动包中的 lib 目录添加到 LD_LIBRARY_PATH 中。

- [UnixODBC]connect to server failed: no such file or directory

此问题可能的原因：

- 配置了错误的/不可达的数据库地址，或者端口

请检查数据源配置中的 Servername 及 Port 配置项。

- 服务器侦听不正确

如果确认 Servername 及 Port 配置正确，请根据“操作步骤”中数据库服务器的相关配置，确保数据库侦听了合适的网卡及端口。

- 防火墙及网闸设备
- 请确认防火墙设置，将数据库的通信端口添加到可信端口中。如果有网闸设备，请确认一下相关的设置。

- [unixODBC]The password-stored method is not supported.

此问题可能原因：

数据源中未配置 sslmode 配置项。

解决办法：

请配置该选项至 allow 或以上选项。此配置的更多信息，见表 6-10。

- Server common name "xxxx" does not match host name "xxxxx"

此问题的原因：使用了 SSL 加密的“verify-full”选项，驱动程序会验证证书中的主机名与实际部署数据库的主机名是否一致。

解决办法：碰到此问题可以使用“verify-ca”选项，不再校验主机名；或者重新生成一套与数据库所在主机名相同的 CA 证书。

- Driver's SQLAllocHandle on SQL_HANDLE_DBC failed

此问题的可能原因：可执行文件（比如 UnixODBC 的 isql，以下都以 isql 为例）与数据库驱动（psqlodbcw.so）依赖于不同的 odbc 的库版本：libodbc.so.1 或者 libodbc.so.2。此 问题可以通过如下方式确认：

```
ldd `which isql` | grep odbc ldd psqlodbcw.so | grep odbc
```

这时，如果输出的 libodbc.so 最后的后缀数字不同或者指向不同的磁盘物理文件， 那么基本就可以断定是此问题。isql 与 psqlodbcw.so 都会要求加载 libodbc.so，这时如果它们加载的是不同的物理文件，便会导致两套完全同名的函数列表，同时出现在同一个可见域里（UnixODBC 的 libodbc.so.*的函数导出列表完全一致）， 产生冲突，无法加载数据库驱动。

解决办法：确定一个要使用的 UnixODBC，然后卸载另外一个（比如卸载库版本号为.so.2 的 UnixODBC），然后将剩下的.so.1 的库，新建一个同名但是后缀为.so.2 的软链接，便可解决此问题。

- FATAL: Forbid remote connection with trust method!

由于安全原因，数据库主节点禁止 GBase 8s 内部其他节点无认证接入。

如果要在 GBase 8s 内部访问数据库主节点，请将 ODBC 程序部署在数据库主节点所在机器，服务器地址使用"127.0.0.1"。建议业务系统单独部署在 GBase 8s 外部，否则可能会影响数据库运行性能。

- [unixODBC][Driver Manager]Invalid attribute value

有可能是 unixODBC 的版本并非推荐版本，建议通过“odbcinst --version”命令排查环境中的 unixODBC 版本。

- authentication method 10 not supported.

使用开源客户端碰到此问题，可能原因：

数据库中存储的口令校验只存储了 SHA256 格式哈希，而开源客户端只识别 MD5 校验，双方校验方法不匹配报错。

说明

- 数据库并不存储用户口令，只存储用户口令的哈希码。

- 数据库当用户更新用户口令或者新建用户时，会同时存储两种格式的哈希码，这时将兼容开源的认证协议。
- 但是当老版本升级到新版本时，由于哈希的不可逆性，所以数据库无法还原用户口令， 进而生成新格式的哈希，所以仍然只保留了 SHA256 格式的哈希，导致仍然无法使用 MD5 做口令认证。
- MD5 加密算法安全性低，存在安全风险，建议使用更安全的加密算法。
- 要解决该问题，可以更新用户口令（参见《GBase 8s V8.8.5_5.0.0_SQL 参考手册》ALTER USER 章节）；或者新建一个用户（参见《GBase 8s V8.8.5_5.0.0_SQL 参考手册》CREATE USER 章节），赋于同等权限，使用新用户连接数据库。

- unsupported frontend protocol 3.51: server supports 1.0 to 3.0

目标数据库版本过低，或者目标数据库为开源数据库。请使用对应版本的数据库驱动连接目标数据库。

- FATAL: GSS authentication method is not allowed because XXXX user password is not disabled.

目标数据库主节点的 pg_hba.conf 里配置了当前客户端 IP 使用"gss"方式来做认证， 该认证算法不支持用作客户端的身份认证，请修改到"sha256"后再试。配置方法见步骤 8。

4.3 开发流程

图 4-2 ODBC 开发应用程序的流程

开发流程中涉及的 API

表 4-5 相关 API 说明

功能	API
申请句柄资源	SQLAllocHandle：申请句柄资源，可替代如下函数： SQLAllocEnv：申请环境句柄 SQLAllocConnect：申请连接句柄 SQLAllocStmt：申请语句句柄
设置环境属性	SQLSetEnvAttr
设置连接属性	SQLSetConnectAttr
设置语句属性	SQLSetStmtAttr
连接数据源	SQLConnect

绑定缓冲区到结果集的列中	SQLBindCol
绑定 SQL 语句的参数标志和缓冲区	SQLBindParameter
查看最近一次操作错误信息	SQLGetDiagRec
为执行 SQL 语句做准备	SQLPrepare
执行一条准备好的 SQL 语句	SQLExecute
直接执行 SQL 语句	SQLExecDirect
结果集中取行集	SQLFetch
返回结果集中某一列的数据	SQLGetData
获取结果集中列的描述信息	SQLColAttribute
断开与数据源的连接	SQLDisconnect
释放句柄资源	SQLFreeHandle：释放句柄资源，可替代如下函数： SQLFreeEnv：释放环境句柄 SQLFreeConnect：释放连接句柄 SQLFreeStmt：释放语句句柄

说明

- 数据库中收到的一次执行请求（不在事务块中），如果含有多条语句，将会被打包成一个事务， 同时如果其中有一个语句失败，那么整个请求都将会被回滚。

警告

- ODBC 为应用程序与数据库的中心层，负责把应用程序发出的 SQL 指令传到数据库当 中，自身并不解析 SQL 语法。故在应用程序中写入带有保密信息的 SQL 语句时（如明文密码），保密信息会被暴露在驱动日志中。

4.4 示例：常用功能和批量绑定

常用功能示例代码

```
// 此示例演示如何通过 ODBC 方式获取 GBase 8s 中的数据。
// DBtest.c (compile with: libodbc.so) #include <stdlib.h>
#include <stdio.h> #include <sqlext.h> #ifdef WIN32 #include <windows.h> #endif
SQLHENV V_OD_Env;      // Handle ODBC environment
SQLHSTMT    V_OD_hstmt    // Handle statement
SQLHDBC    V_OD_hdbc;    // Handle connection
char       typename[100];
SQLINTEGER value = 100;
SQLINTEGER V_OD_erg,V_OD_buffer,V_OD_err,V_OD_id; int main(int argc,char *argv[])
{
// 1. 申请环境句柄
V_OD_erg = SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&V_OD_Env);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
printf("Error AllocHandle\n"); exit(0);
}
// 2. 设置环境属性（版本信息）
SQLSetEnvAttr(V_OD_Env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
// 3. 申请连接句柄
V_OD_erg = SQLAllocHandle(SQL_HANDLE_DBC, V_OD_Env, &V_OD_hdbc);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env); exit(0);
}
// 4. 设置连接属性
SQLSetConnectAttr(V_OD_hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_ON,
0);
// 5. 连接数据源，这里的“userName”与“password”分别表示连接数据库的用户名和用户密码，请根据实际情况修改。
// 如果 odbc.ini 文件中已经配置了用户名密码，那么这里可以留空（""）；但是不建议这么做，因为一旦
odbc.ini 权限管理不善，将导致数据库用户密码泄露。
V_OD_erg = SQLConnect(V_OD_hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
(SQLCHAR*) "userName", SQL_NTS, (SQLCHAR*) "password", SQL_NTS); if
((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
```

```
printf("Error SQLConnect %d\n",V_OD_erg); SQLFreeHandle(SQL_HANDLE_ENV,
V_OD_Env); exit(0);
}
printf("Connected !\n");
// 6. 设置语句属性
SQLSetStmtAttr(V_OD_hstmt,SQL_ATTR_QUERY_TIMEOUT,(SQLPOINTER *)3,0);
// 7. 申请语句句柄
SQLAllocHandle(SQL_HANDLE_STMT, V_OD_hdbc, &V_OD_hstmt);
// 8. 直接执行 SQL 语句。
SQLExecDirect(V_OD_hstmt,"drop table IF EXISTS customer_t1",SQL_NTS);
SQLExecDirect(V_OD_hstmt,"CREATE TABLE customer_t1(c_customer_sk INTEGER,
c_customer_name
VARCHAR(32));",SQL_NTS);
SQLExecDirect(V_OD_hstmt,"insert into customer_t1 values(25,li)",SQL_NTS);
// 9. 准备执行
SQLPrepare(V_OD_hstmt,"insert into customer_t1 values(?)",SQL_NTS);
// 10. 绑定参数
SQLBindParameter(V_OD_hstmt,1,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_INTEGER,
0,0, &value,0,NULL);
// 11. 执行准备好的语句
SQLExecute(V_OD_hstmt);
SQLExecDirect(V_OD_hstmt,"select id from testtable",SQL_NTS);
// 12. 获取结果集某一列的属性
SQLColAttribute(V_OD_hstmt,1,SQL_DESC_TYPE,typename,100,NULL,NULL);
printf("SQLColAtrribute %s\n",typename);
// 13. 绑定结果集
SQLBindCol(V_OD_hstmt,1,SQL_C_SLONG, (SQLPOINTER)&V_OD_buffer,150,
(SQLLEN *)&V_OD_err);
// 14. 通过 SQLFetch 取结果集中数据
V_OD_erg=SQLFetch(V_OD_hstmt);
// 15. 通过 SQLGetData 获取并返回数据。
while(V_OD_erg != SQL_NO_DATA)
{
SQLGetData(V_OD_hstmt,1,SQL_C_SLONG,(SQLPOINTER)&V_OD_id,0,NULL);
printf("SQLGetData     ID = %d\n",V_OD_id);
V_OD_erg=SQLFetch(V_OD_hstmt);
};
printf("Done !\n");
// 16. 断开数据源连接并释放句柄资源
```

```
SQLFreeHandle(SQL_HANDLE_STMT,V_OD_hstmt); SQLDisconnect(V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_DBC,V_OD_hdbc); SQLFreeHandle(SQL_HANDLE_ENV,
V_OD_Env); return(0);
}
批量绑定示例代码
/**************************************************************************
请在数据源中打开 UseBatchProtocol，同时指定数据库中参数 support_batch_bind
为 on
CHECK_ERROR 的作用是检查并打印错误信息。
此示例将与用户交互式获取 DSN、模拟的数据量，忽略的数据量，并将最终数据入库到
test_odbc_batch_insert 中
**************************************************************************/
#include <stdio.h>
#include <stdlib.h> #include <sql.h> #include <sqlext.h> #include <string.h>
void Exec(SQLHDBC hdbc, SQLCHAR* sql)
{
SQLRETURN retcode; // Return status
SQLHSTMT hstmt = SQL_NULL_HSTMT; // Statement handle SQLCHAR  loginfo[2048];
// Allocate Statement Handle
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLAllocHandle(SQL_HANDLE_STMT) failed");
return;
}
// Prepare Statement
retcode = SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS); sprintf((char*)loginfo,
"SQLPrepare log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) {
printf("SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS) failed"); return;
}
// Execute Statement
retcode = SQLExecute(hstmt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute(hstmt) failed"); return;
}
// Free Handle
retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt); sprintf((char*)loginfo,
"SQLFreeHandle stmt log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLFreeHandle(SQL_HANDLE_STMT, hstmt)
failed"); return;
}
}
```

```
int main ()
{
SQLHENV henv = SQL_NULL_HENV; SQLHDBC hdbc = SQL_NULL_HDBC;
int  batchCount = 1000; //  批量绑定的数据量
SQLLEN rowsCount = 0;
int  ignoreCount = 0; //  批量绑定的数据中，不要入库的数据量
SQLRETURN retcode; SQLCHAR     dsn[1024] = {'\0'};
SQLCHAR    loginfo[2048];
do
{
if (ignoreCount > batchCount)
{
printf("ignoreCount(%d) should be less than batchCount(%d)\n", ignoreCount, batchCount);
}
}while(ignoreCount > batchCount);
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv); if
(!SQL_SUCCEEDED(retcode)) {
printf("SQLAllocHandle failed"); goto exit;
}
// Set ODBC Verion
retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER*)SQL_OV_ODBC3, 0);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetEnvAttr failed"); goto exit;
}
// Allocate Connection
retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLAllocHandle failed"); goto exit;
}
// Set Login Timeout
retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetConnectAttr failed"); goto exit;
}
// Set Auto Commit
retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT,
(SQLPOINTER)(1), 0);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetConnectAttr failed"); goto exit;
}
// Connect to DSN
// gaussdb 替换成用户所使用的数据源名称 sprintf(loginfo, "SQLConnect(DSN:%s)", dsn);
retcode = SQLConnect(hdbc, (SQLCHAR*) "gaussdb", SQL_NTS, (SQLCHAR*) NULL, 0,
NULL, 0);
```

```
if (!SQL_SUCCEEDED(retcode)) { printf("SQLConnect failed"); goto exit;
}
// init table info.
Exec(hdbc, "drop table if exists test_odbc_batch_insert");
Exec(hdbc, "create table test_odbc_batch_insert(id int primary key, col varchar2(50))");
// 下面的代码根据用户输入的数据量，构造出将要入库的数据:
{
SQLRETURN retcode;
SQLHSTMT hstmtinesrt = SQL_NULL_HSTMT; int       i;
SQLCHAR   *sql = NULL; SQLINTEGER *ids = NULL; SQLCHAR    *cols = NULL;
SQLLEN      *bufLenIds = NULL; SQLLEN     *bufLenCols = NULL; SQLUSMALLINT
*operptr = NULL; SQLUSMALLINT *statusptr = NULL; SQLULEN           process = 0;
// 这里是按列构造，每个字段的内存连续存放在一起。
ids = (SQLINTEGER*)malloc(sizeof(ids[0]) * batchCount); cols =
(SQLCHAR*)malloc(sizeof(cols[0]) * batchCount * 50);
// 这里是每个字段中，每一行数据的内存长度。
bufLenIds = (SQLLEN*)malloc(sizeof(bufLenIds[0]) * batchCount); bufLenCols =
(SQLLEN*)malloc(sizeof(bufLenCols[0]) * batchCount);
// 该行是否需要被处理，SQL_PARAM_IGNORE 或 SQL_PARAM_PROCEED operptr =
(SQLUSMALLINT*)malloc(sizeof(operptr[0]) * batchCount); memset(operptr, 0,
sizeof(operptr[0]) * batchCount);
// 该行的处理结果。
// 注：由于数据库中处理方式是同一语句隶属同一事务中，所以如果出错，那么待处理数
据都将是出错的，  并不会部分入库。
statusptr = (SQLUSMALLINT*)malloc(sizeof(statusptr[0]) * batchCount); memset(statusptr,
88, sizeof(statusptr[0]) * batchCount);
if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds)
{
fprintf(stderr, "FAILED:\tmalloc data memory failed\n"); goto exit;
}
for (int i = 0; i < batchCount; i++)
{
ids[i] = i;
sprintf(cols + 50 * i, "column test value %d", i); bufLenIds[i] = sizeof(ids[i]);
bufLenCols[i] = strlen(cols + 50 * i);
operptr[i] = (i < ignoreCount) ? SQL_PARAM_IGNORE : SQL_PARAM_PROCEED;
}
// Allocate Statement Handle
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtinesrt);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLAllocHandle failed"); goto exit;
}
```

```
// Prepare Statement
sql = (SQLCHAR*)"insert into test_odbc_batch_insert values(?, ?)"; retcode =
SQLPrepare(hstmtinesrt, (SQLCHAR*) sql, SQL_NTS); sprintf((char*)loginfo, "SQLPrepare
log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLPrepare failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMSET_SIZE,
(SQLPOINTER)batchCount, sizeof(batchCount));
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLBindParameter(hstmtinesrt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, sizeof(ids[0]), 0,&(ids[0]), 0, bufLenIds);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLBindParameter failed"); goto exit;
}
retcode = SQLBindParameter(hstmtinesrt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 50, 50,
cols, 50, bufLenCols);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLBindParameter failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMS_PROCESSED_PTR,
(SQLPOINTER)&process, sizeof(process));
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_STATUS_PTR,
(SQLPOINTER)statusptr, sizeof(statusptr[0]) * batchCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_OPERATION_PTR,
(SQLPOINTER)operptr, sizeof(operptr[0]) * batchCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLSetStmtAttr failed"); goto exit;
}
retcode = SQLExecute(hstmtinesrt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute(hstmtinesrt) failed"); goto exit;
retcode = SQLRowCount(hstmtinesrt, &rowsCount); if (!SQL_SUCCEEDED(retcode)) {
printf("SQLRowCount failed");
goto exit;
}
if (rowsCount != (batchCount - ignoreCount))
{
```

```
sprintf(loginfo, "(batchCount - ignoreCount)(%d) != rowsCount(%d)", (batchCount -
ignoreCount), rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
else
{
sprintf(loginfo, "(batchCount - ignoreCount)(%d) == rowsCount(%d)", (batchCount -
ignoreCount), rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
// check row number returned if (rowsCount != process)
{
sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
else
{
sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
for (int i = 0; i < batchCount; i++)
{
if (i < ignoreCount)
{
if (statusptr[i] != SQL_PARAM_UNUSED)
{
sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_UNUSED", i, statusptr[i]);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
}
else if (statusptr[i] != SQL_PARAM_SUCCESS)
{
sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_SUCCESS", i, statusptr[i]);
if (!SQL_SUCCEEDED(retcode)) { printf("SQLExecute failed"); goto exit;
}
}
```

```
}
retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmtinesrt); sprintf((char*)loginfo,
"SQLFreeHandle hstmtinesrt");
if (!SQL_SUCCEEDED(retcode)) { printf("SQLFreeHandle failed"); goto exit;
}
}
exit:
(void) printf ("\nComplete.\n");
// Connection
if (hdbc != SQL_NULL_HDBC) { SQLDisconnect(hdbc);
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}
// Environment
if (henv != SQL_NULL_HENV) SQLFreeHandle(SQL_HANDLE_ENV, henv);
return 0;
}
```

4.5 典型应用场景配置

日志诊断场景

ODBC 日志分为 unixODBC 驱动管理器日志和 psqlODBC 驱动端日志。前者可以用于追溯应用程序 API 的执行是否成功，后者是底层实现过程中的一些 DFX 日志，用来帮助定位问题。

```
unixODBC 日志需要在 odbcinst.ini 文件中配置：
[ODBC]
Trace=Yes TraceFile=/path/to/odbctrace.log
[GaussMPP] Driver64=/usr/local/lib/psqlodbcw.so setup=/usr/local/lib/psqlodbcw.so
```

psqlODBC 日志只需要在 odbc.ini 加上：

```
[gaussdb] Driver=GaussMPP
Servername=10.10.0.13（数据库 Server IP）
...
Debug=1（打开驱动端 debug 日志）
```

说明

unixODBC 日志将会生成在 TraceFile 配置的路径下，psqlODBC 会在系统/tmp/下生成 mylog_xxx.log。

高性能场景

进行大量数据插入时，建议如下：

- 需要设置批量绑定：odbc.ini 配置文件中设置 UseBatchProtocol=1、数据库设置
- support_batch_bind=on。
- ODBC 程序绑定类型要和数据库中类型一致。
- 客户端字符集和数据库字符集一致。
- 事务改成手动提交。

odbc.ini 配置文件：

```
[gaussdb] Driver=GaussMPP
Servername=10.10.0.13（数据库 Server IP）
...
UseBatchProtocol=1 （默认打开）
ConnSettings=set client_encoding=UTF8 （设置客户端字符编码，保证和 server 端一致）
```

绑定类型用例：

```
#include <stdio.h> #include <stdlib.h> #include <sql.h> #include <sqlext.h> #include
<string.h> #include <sys/time.h>
#define MESSAGE_BUFFER_LEN 128 SQLHANDLE h_env = NULL; SQLHANDLE
h_conn = NULL; SQLHANDLE h_stmt = NULL;
void print_error()
{
SQLCHAR Sqlstate[SQL_SQLSTATE_SIZE+1];
SQLINTEGER NativeError;
SQLCHAR MessageText[MESSAGE_BUFFER_LEN];
SQLSMALLINT TextLength; SQLRETURN ret = SQL_ERROR;
ret = SQLGetDiagRec(SQL_HANDLE_STMT, h_stmt, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
if ( SQL_SUCCESS == ret)
{
printf("\n STMT ERROR-%05d %s", NativeError, MessageText); return;
}
ret = SQLGetDiagRec(SQL_HANDLE_DBC, h_conn, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
if ( SQL_SUCCESS == ret)
{
printf("\n CONN ERROR-%05d %s", NativeError, MessageText); return;
}
```

```
ret = SQLGetDiagRec(SQL_HANDLE_ENV, h_env, 1, Sqlstate, &NativeError, MessageText,
MESSAGE_BUFFER_LEN, &TextLength);
if ( SQL_SUCCESS == ret)
{
printf("\n ENV ERROR-%05d %s", NativeError, MessageText); return;
}
return;
}
/*  期盼函数返回 SQL_SUCCESS */
#define RETURN_IF_NOT_SUCCESS(func) \
{\
SQLRETURN ret_value = (func);\ if (SQL_SUCCESS != ret_value)\
{\
print_error();\
printf("\n failed line = %u: expect SQL_SUCCESS, but ret = %d", LINE , ret_value);\ return
SQL_ERROR; \
}\
}
/*  期盼函数返回 SQL_SUCCESS */
#define RETURN_IF_NOT_SUCCESS_I(i, func) \
{\
SQLRETURN ret_value = (func);\ if (SQL_SUCCESS != ret_value)\
{\
print_error();\
printf("\n failed line = %u (i=%d): : expect SQL_SUCCESS, but ret = %d", LINE , (i),
ret_value);\ return SQL_ERROR; \
}\
}
/*  期盼函数返回 SQL_SUCCESS_WITH_INFO */ #define
RETURN_IF_NOT_SUCCESS_INFO(func) \
{\
SQLRETURN ret_value = (func);\
if (SQL_SUCCESS_WITH_INFO != ret_value)\
{\
print_error();\
printf("\n failed line = %u: expect SQL_SUCCESS_WITH_INFO, but ret = %d", LINE ,
ret_value);\ return SQL_ERROR; \
}\
}
/*  期盼数值相等  */
#define RETURN_IF_NOT(expect, value) \ if ((expect) != (value))\
```

```
{\
printf("\n failed line = %u: expect = %u, but value = %u", LINE , (expect), (value)); \ return
SQL_ERROR;\
}
/* 期盼字符串相同 */
#define RETURN_IF_NOT_STRCMP_I(i, expect, value) \ if (( NULL == (expect) ) || (NULL
== (value)))\
{\
printf("\n failed line = %u (i=%u): input NULL pointer !", LINE , (i)); \ return SQL_ERROR; \
}\
else if (0 != strcmp((expect), (value)))\
{\
printf("\n failed line = %u (i=%u): expect = %s, but value = %s", LINE , (i), (expect), (value)); \
return SQL_ERROR;\
}
// prepare + execute SQL 语句 int execute_cmd(SQLCHAR *sql)
{
if ( NULL == sql )
{
return SQL_ERROR;
}
if ( SQL_SUCCESS != SQLPrepare(h_stmt, sql, SQL_NTS))
{
return SQL_ERROR;
}
if ( SQL_SUCCESS != SQLExecute(h_stmt))
{
return SQL_ERROR;
}
return SQL_SUCCESS;
}
// execute + commit 句柄
int commit_exec()
{
if ( SQL_SUCCESS != SQLExecute(h_stmt))
{
return SQL_ERROR;
}
// 手动提交
if ( SQL_SUCCESS != SQLEndTran(SQL_HANDLE_DBC, h_conn, SQL_COMMIT))
{
```

```
return SQL_ERROR;
}
return SQL_SUCCESS;
}
int begin_unit_test()
{
SQLINTEGER ret;
/* 申请环境句柄 */
ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &h_env); if
((SQL_SUCCESS != ret) && (SQL_SUCCESS_WITH_INFO != ret))
{
printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_ENV failed ! ret = %d", ret);
return SQL_ERROR;
}
/* 进行连接前必须要先设置版本号 */
if (SQL_SUCCESS != SQLSetEnvAttr(h_env, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER)SQL_OV_ODBC3, 0))
{
print_error();
printf("\n begin_unit_test::SQLSetEnvAttr SQL_ATTR_ODBC_VERSION failed ! ret = %d",
ret); SQLFreeHandle(SQL_HANDLE_ENV, h_env);
return SQL_ERROR;
}
/* 申请连接句柄 */
ret = SQLAllocHandle(SQL_HANDLE_DBC, h_env, &h_conn); if (SQL_SUCCESS != ret)
{
print_error();
printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_DBC failed ! ret = %d", ret);
SQLFreeHandle(SQL_HANDLE_ENV, h_env);
return SQL_ERROR;
}
/* 建立连接 */
ret = SQLConnect(h_conn, (SQLCHAR*) "gaussdb", SQL_NTS,
(SQLCHAR*) NULL, 0, NULL, 0); if (SQL_SUCCESS != ret)
{
print_error();
printf("\n begin_unit_test::SQLConnect failed ! ret = %d", ret);
SQLFreeHandle(SQL_HANDLE_DBC, h_conn); SQLFreeHandle(SQL_HANDLE_ENV,
h_env);
return SQL_ERROR;
}
```

```
/* 申请语句句柄 */
ret = SQLAllocHandle(SQL_HANDLE_STMT, h_conn, &h_stmt); if (SQL_SUCCESS != ret)
{
print_error();
printf("\n begin_unit_test::SQLAllocHandle SQL_HANDLE_STMT failed ! ret = %d", ret);
SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
SQLFreeHandle(SQL_HANDLE_ENV, h_env); return SQL_ERROR;
}
return SQL_SUCCESS;
}
void end_unit_test()
{
/* 释放语句句柄 */ if (NULL != h_stmt)
{
SQLFreeHandle(SQL_HANDLE_STMT, h_stmt);
}
/* 释放连接句柄 */ if (NULL != h_conn)
{
SQLDisconnect(h_conn); SQLFreeHandle(SQL_HANDLE_DBC, h_conn);
}
/* 释放环境句柄 */ if (NULL != h_env)
{
SQLFreeHandle(SQL_HANDLE_ENV, h_env);
}
return;
}
int main()
{
// begin test
if (begin_unit_test() != SQL_SUCCESS)
{
printf("\n begin_test_unit failed."); return SQL_ERROR;
}
// 句柄配置同前面用例
int  i = 0;
SQLCHAR* sql_drop = "drop table if exists test_bindnumber_001"; SQLCHAR* sql_create =
"create table test_bindnumber_001("
"f4 number, f5 number(10, 2)" ")";
SQLCHAR* sql_insert = "insert into test_bindnumber_001 values(?, ?)"; SQLCHAR*
sql_select = "select * from test_bindnumber_001";
SQLLEN      RowCount; SQL_NUMERIC_STRUCT st_number;
```

```
SQLCHAR   getValue[2][MESSAGE_BUFFER_LEN];
/* step 1. 建 表 */ RETURN_IF_NOT_SUCCESS(execute_cmd(sql_drop));
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_create));
/* step 2.1 通 过 SQL_NUMERIC_STRUCT 结 构 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
//第一行: 1234.5678
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 8;
st_number.scale = 4;
st_number.sign = 1; st_number.val[0] = 0x4E; st_number.val[1] = 0x61; st_number.val[2] =
0xBC;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
// 关闭自动提交
SQLSetConnectAttr(h_conn, SQL_ATTR_AUTOCOMMIT,
(SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0);
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
//第二行: 12345678
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 8;
st_number.scale = 0;
st_number.sign = 1; st_number.val[0] = 0x4E; st_number.val[1] = 0x61; st_number.val[2] =
0xBC;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 0, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 0, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
//第三行: 12345678
memset(st_number.val, 0, SQL_MAX_NUMERIC_LEN);
st_number.precision = 0;
```

```
st_number.scale = 4;
st_number.sign = 1; st_number.val[0] = 0x4E; st_number.val[1] = 0x61; st_number.val[2] =
0xBC;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_NUMERIC, SQL_NUMERIC, sizeof(SQL_NUMERIC_STRUCT), 4, &st_number, 0,
NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.2 第 四 行 通 过 SQL_C_CHAR 字 符 串 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS)); SQLCHAR*
szNumber = "1234.5678";
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_NUMERIC, strlen(szNumber), 0, szNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_CHAR, SQL_NUMERIC, strlen(szNumber), 0, szNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.3 第 五 行 通 过 SQL_C_FLOAT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS)); SQLREAL
fNumber = 1234.5678;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_FLOAT, SQL_NUMERIC, sizeof(fNumber), 4, &fNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_FLOAT, SQL_NUMERIC, sizeof(fNumber), 4, &fNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.4 第 六 行 通 过 SQL_C_DOUBLE 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS)); SQLDOUBLE
dNumber = 1234.5678;
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_DOUBLE, SQL_NUMERIC, sizeof(dNumber), 4, &dNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_DOUBLE, SQL_NUMERIC, sizeof(dNumber), 4, &dNumber, 0, NULL));
```

```
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1, RowCount);
SQLBIGINT bNumber1 = 0xFFFFFFFFFFFFFFFF; SQLBIGINT bNumber2 = 12345;
/* step 2.5 第七行通过 SQL_C_SBIGINT 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_SBIGINT, SQL_NUMERIC, sizeof(bNumber1), 4, &bNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_SBIGINT, SQL_NUMERIC, sizeof(bNumber2), 4, &bNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1, RowCount);
/* step 2.6 第八行通过 SQL_C_UBIGINT 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_UBIGINT,
SQL_NUMERIC, sizeof(bNumber1), 4, &bNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_UBIGINT,
SQL_NUMERIC, sizeof(bNumber2), 4, &bNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NO_UCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1, RowCount);
SQLLEN lNumber1 = 0xFFFFFFFFFFFFFFFF; SQLLEN lNumber2 = 12345;
/* step 2.7 第九行通过 SQL_C_LONG 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_LONG,
SQL_NUMERIC, sizeof(lNumber1), 0, &lNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_LONG,
SQL_NUMERIC, sizeof(lNumber2), 0, &lNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1, RowCount);
/* step 2.8 第十行通过 SQL_C_ULONG 绑定参数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_ULONG,
```

```
SQL_NUMERIC, sizeof(lNumber1), 0, &lNumber1, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_ULONG,
SQL_NUMERIC, sizeof(lNumber2), 0, &lNumber2, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
SQLSMALLINT sNumber = 0xFFFF;
/* step 2.9 第 十 一 行 通 过 SQL_C_SHORT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_SHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_SHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* step 2.10 第 十 二 行 通 过 SQL_C_USHORT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_USHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_USHORT,
SQL_NUMERIC, sizeof(sNumber), 0, &sNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
SQLCHAR cNumber = 0xFF;
/* step 2.11 第 十 三 行 通 过 SQL_C_TINYINT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 1, SQL_PARAM_INPUT,
SQL_C_TINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt, 2, SQL_PARAM_INPUT,
SQL_C_TINYINT, SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
```

```
/* step 2.12 第 十 四 行 通 过 SQL_C_UTINYINT 绑 定 参 数 */
RETURN_IF_NOT_SUCCESS(SQLPrepare(h_stmt, sql_insert, SQL_NTS));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  1,  SQL_PARAM_INPUT,
SQL_C_UTINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(SQLBindParameter(h_stmt,  2,  SQL_PARAM_INPUT,
SQL_C_UTINYINT,
SQL_NUMERIC, sizeof(cNumber), 0, &cNumber, 0, NULL));
RETURN_IF_NOT_SUCCESS(commit_exec());
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(1,
RowCount);
/* 用字符串类型统一进行期盼 */
SQLCHAR*       expectValue[14][2] = {{"1234.5678",    "1234.57"},
{"12345678","12345678"},
{"0", "0"},
{"1234.5678","1234.57"},
{"1234.5677","1234.57"},
{"1234.5678","1234.57"},
{"-1","12345"},
{"-1","12345"},
{"4294967295","12345"},
{"18446744073709551615", "12345"},
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_select)); while ( SQL_NO_DATA !=
SQLFetch(h_stmt))
{
RETURN_IF_NOT_SUCCESS_I(i, SQLGetData(h_stmt, 1, SQL_C_CHAR, &getValue[0],
MESSAGE_BUFFER_LEN, NULL));
RETURN_IF_NOT_SUCCESS_I(i, SQLGetData(h_stmt, 2, SQL_C_CHAR, &getValue[1],
MESSAGE_BUFFER_LEN, NULL));
//RETURN_IF_NOT_STRCMP_I(i, expectValue[i][0], getValue[0]);
//RETURN_IF_NOT_STRCMP_I(i, expectValue[i][1], getValue[1]); i++;
}
RETURN_IF_NOT_SUCCESS(SQLRowCount(h_stmt, &RowCount)); RETURN_IF_NOT(i,
RowCount);
SQLCloseCursor(h_stmt);
/* step final. 删 除 表 还 原 环 境 */
RETURN_IF_NOT_SUCCESS(execute_cmd(sql_drop));
end_unit_test();
}
```

说明

上述用例中定义了 number 列，调用 SQLBindParameter 接口时，绑定 SQL_NUMERIC 会比 SQL_LONG 性能高一些。因为如果是 char，在数据库服务端插入数据时需要进行数据类型转换， 从而引发性能瓶颈。

4.6 ODBC 接口参考

ODBC 接口是一套提供给用户的 API 函数，本节将对部分常用接口做具体描述，若涉及其他接口可参考 msdn （网址： https://msdn.microsoft.com/en-us/library/windows/desktop/ms714177(v=vs.85).aspx）中 ODBC Programmer's Reference 项的相关内容。

4.6.1 SQLAllocEnv

在 ODBC 3.x 版本中，ODBC 2.x 的函数 SQLAllocEnv 已被 SQLAllocHandle 代替。有关详细信息请参阅 SQLAllocHandle。

4.6.2 SQLAllocConnect

在 ODBC 3.x 版本中，ODBC 2.x 的函数 SQLAllocConnect 已被 SQLAllocHandle 代替。有关详细信息请参阅 SQLAllocHandle。

4.6.3 SQLAllocHandle

功能描述

分配环境、连接、语句或描述符的句柄，它替代了 ODBC 2.x 函数 SQLAllocEnv、SQLAllocConnect 及 SQLAllocStmt。

原型

```
SQLRETURN SQLAllocHandle(SQLSMALLINT HandleType,
SQLHANDLE     InputHandle,
SQLHANDLE     *OutputHandlePtr);
```

参数

表 4-6 SQLAllocHandle 参数

关键字	参数说明
HandleType	由 SQLAllocHandle 分配的句柄类型。必须为下列值之一： ● SQL_HANDLE_ENV（环境句柄） ● SQL_HANDLE_DBC（连接句柄）

	● SQL_HANDLE_STMT（语句句柄） ● SQL_HANDLE_DESC（描述句柄） 申请句柄顺序为，先申请环境句柄，再申请连接句柄，最后申请语句句柄，后申请的句柄都要依赖它前面申请的句柄。
InputHandle	将要分配的新句柄的类型。 ● 如果 HandleType 为 SQL_HANDLE_ENV，则这个值为 SQL_NULL_HANDLE。 ● 如果 HandleType 为 SQL_HANDLE_DBC，则这一定是一个环境句柄。 ● 如果 HandleType 为 SQL_HANDLE_STMT 或 SQL_HANDLE_DESC，则它一定是一个连接句柄。
OutputHandlePt r	输出参数：一个缓冲区的指针，此缓冲区以新分配的数据结构存放返回的句柄。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当分配的句柄并非环境句柄时，如果 SQLAllocHandle 返回的值为 SQL_ERROR，则它会将 OutputHandlePtr 的值设置为 SQL_NULL_HDBC、SQL_NULL_HSTMT 或 SQL_NULL_HDESC。之后，通过调用带有适当参数的 SQLGetDiagRec，其中 HandleType 和 Handle 被设置为 IntputHandle 的值，可得到相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.4 SQLAllocStmt

在 ODBC 3.x 版本中，ODBC 2.x 的函数 SQLAllocStmt 已被 SQLAllocHandle 代替。有关详细信息请参阅 SQLAllocHandle。

4.6.5 SQLBindCol

功能描述

将应用程序数据缓冲区绑定到结果集的列中。

原型

```
SQLRETURN SQLBindCol(SQLHSTMT  StatementHandle,
SQLUSMALLINT ColumnNumber,
SQLSMALLINT TargetType,
SQLPOINTER     TargetValuePtr,
SQLLEN      BufferLength,
SQLLEN      *StrLen_or_IndPtr);
```

参数

表 4-7 SQLBindCol 参数

关键字	参数说明
StatementHandl e	语句句柄。
ColumnNumber	要绑定结果集的列号。起始列号为 0，以递增的顺序计算列号，第 0 列是书签列。若未设置书签页，则起始列号为 1。
TargetType	缓冲区中 C 数据类型的标识符。
TargetValuePtr	输出参数：指向与列绑定的数据缓冲区的指针。SQLFetch 函数返回这个缓冲区中的数据。如果此参数为一个空指针，则 StrLen_or_IndPtr 是一个有效值。
BufferLength	TargetValuePtr 指向缓冲区的长度，以字节为单位。
StrLen_or_IndPtr	输出参数：缓冲区的长度或指示器指针。若为空值，则未使用任何长度或指示器值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLBindCol 返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用 此函数的具体信息。

示例

参见示例。

4.6.6 SQLBindParameter

功能描述

将一条 SQL 语句中的一个参数标志和一个缓冲区绑定起来。

原型

```
SQLRETURN SQLBindParameter(SQLHSTMT      StatementHandle,
SQLUSMALLINT ParameterNumber,
SQLSMALLINT     InputOutputType,
SQLSMALLINT ValuetType,
SQLSMALLINT ParameterType,
SQLULEN ColumnSize,
SQLSMALLINT DecimalDigits,
SQLPOINTER     ParameterValuePtr,
SQLLEN      BufferLength,
SQLLEN      *StrLen_or_IndPtr);
```

参数

表 4-8 SQLBindParameter 参数

关键字	参数说明
StatementHandle	语句句柄。
ParameterNumbe r	参数序号，起始为 1，依次递增。
InputOutputType	输入输出参数类型。
ValueType	参数的 C 数据类型。

ParameterType	参数的 SQL 数据类型。
ColumnSize	列的大小或相应参数标记的表达式。
DecimalDigits	列的十进制数字或相应参数标记的表达式。
ParameterValuePt r	指向存储参数数据缓冲区的指针。
BufferLength	ParameterValuePtr 指向缓冲区的长度，以字节为单位。
StrLen_or_IndPtr	缓冲区的长度或指示器指针。若为空值，则未使用任何长度或指示器值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLBindParameter 返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用 此函数的具体信息。

示例

参见示例。

4.6.7 SQLColAttribute

功能描述

返回结果集中一列的描述符信息。

原型

```
SQLRETURN SQLColAttibute(SQLHSTMT     StatementHandle,
SQLUSMALLINT ColumnNumber,
SQLUSMALLINT       FieldIdentifier,
SQLPOINTER     CharacterAtrriburePtr,
```

```
SQLSMALLINT  BufferLength,
SQLSMALLINT  *StringLengthPtr,
SQLLEN       *NumericAttributePtr);
```

参数

表 4-9 SQLColAttribute 参数

关键字	参数说明
SQLColAttribute	语句句柄。
SQLColAttribute	要检索字段的列号，起始为 1，依次递增。
SQLColAttribute	IRD 中 ColumnNumber 行的字段。
SQLColAttribute	输出参数：一个缓冲区指针，返回 FieldIdentifier 字段值。
SQLColAttribute	● 如果 FieldIdentifier 是一个 ODBC 定义的字段，而且 CharacterAttributePtr 指向一个字符串或二进制缓冲区，则此参数为该缓冲区的长度。 ● 如果 FieldIdentifier 是一个 ODBC 定义的字段，而且 CharacterAttributePtr 指向一个整数，则会忽略该字段。
SQLColAttribute	输出参数：缓冲区指针，存放*CharacterAttributePtr 中字符 类型数据的字节总数，对于非字符类型，忽略 BufferLength 的值。
NumericAttributePt r	输出参数：指向一个整型缓冲区的指针，返回 IRD 中 ColumnNumber 行 FieldIdentifier 字段的值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLColAttribute 返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_STMT 和

StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用 此函数的具体信息。

示例

参见示例。

4.6.8 SQLConnect

功能描述

在驱动程序和数据源之间建立连接。连接上数据源之后，可以通过连接句柄访问到所有有关连接数据源的信息，包括程序运行状态、事务处理状态和错误信息。

原型

```
SQLRETURN    SQLConnect(SQLHDBC  ConnectionHandle,
SQLCHAR    *ServerName,
SQLSMALLINT NameLength1,
SQLCHAR    *UserName,
SQLSMALLINT NameLength2,
SQLCHAR    *Authentication,
SQLSMALLINT NameLength3);
```

参数

表 4-10 SQLConnect 参数

关键字	参数说明
ConnectionHandle	连接句柄，通过 SQLAllocHandle 获得。
ServerName	要连接数据源的名称。
NameLength1	ServerName 的长度。
UserName	数据源中数据库用户名。
NameLength2	UserName 的长度。
Authentication	数据源中数据库用户密码。
NameLength3	Authentication 的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当调用 SQLConnect 函数返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_DBC 和 ConnectionHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.9 SQLDisconnect

功能描述

关闭一个与特定连接句柄相关的连接。

原型

```
SQLRETURN SQLDisconnect(SQLHDBC      ConnectionHandle);
```

参数

表 4-11 SQLDisconnect 参数

关键字	参数说明
ConnectionHandle	连接句柄，通过 SQLAllocHandle 获得。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当调用 SQLDisconnect 函数返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调 用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_DBC 和 ConnectionHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用 此函数的具体信息。

示例

参见示例。

4.6.10 SQLExecDirect

功能描述

使用参数的当前值，执行一条准备好的语句。对于一次只执行一条 SQL 语句，SQLExecDirect 是最快的执行方式。

原型

```
SQLRETURN SQLExecDirect(SQLHSTMT StatementHandle,
SQLCHAR   *StatementText,
SQLINTEGER          TextLength);
```

参数

表 4-12 SQLExecDirect 参数

关键字	参数说明
StatementHandl e	语句句柄，通过 SQLAllocHandle 获得。
StatementText	要执行的 SQL 语句。不支持一次执行多条语句。
TextLength	StatementText 的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

- SQL_STILL_EXECUTING：表示语句正在执行。
- SQL_NO_DATA：表示 SQL 语句不返回结果集。

注意事项

当调用 SQLExecDirect 函数返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调 用此函数的具体信息。

示例

参见示例。

4.6.11 SQLExecute

功能描述

如果语句中存在参数标记的话，SQLExecute 函数使用参数标记参数的当前值，执行一条准备好的 SQL 语句。

原型

```
SQLRETURN SQLExecute(SQLHSTMT     StatementHandle);
```

参数

表 4-13 SQLExecute 参数

关键字	参数说明
StatementHandle	要执行语句的语句句柄。
ColumnNumber	要绑定结果集的列号。起始列号为 0，以递增的顺序计算列号，第 0 列是书签列。若未设置书签页，则起始列号为 1。
TargetType	缓冲区中 C 数据类型的标识符。
TargetValuePtr	输出参数：指向与列绑定的数据缓冲区的指针。SQLFetch 函数返回这个缓冲区中的数据。如果此参数为一个空指针，则 StrLen_or_IndPtr 是一个有效值。
BufferLength	TargetValuePtr 指向缓冲区的长度，以字节为单位。

StrLen_or_IndPtr	输出参数：缓冲区的长度或指示器指针。若为空值，则未使用任何长度或指示器值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。
- SQL_NO_DATA：表示 SQL 语句不返回结果集。

注意事项

当 SQLExecute 函数返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，可通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.12 SQLFetch

功能描述

从结果集中取下一个行集的数据，并返回所有被绑定列的数据。

原型

```
SQLRETURN SQLFetch(SQLHSTMT      StatementHandle);
```

参数

表 4-14 SQLFetch 参数

关键字	参数说明
StatementHandle	语句句柄，通过 SQLAllocHandle 获得。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。
- SQL_NO_DATA：表示 SQL 语句不返回结果集。

注意事项

当调用 SQLFetch 函数返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用 此函数的具体信息。

示例

参见示例。

4.6.13 SQLFreeStmt

功能描述

在 ODBC 3.x 版本中，ODBC 2.x 的函数 SQLFreeStmt 已被 SQLFreeHandle 代替。有关详细信息请参阅 SQLFreeHandle。

4.6.14 SQLFreeConnect

在 ODBC 3.x 版本中，ODBC 2.x 的函数 SQLFreeConnect 已被 SQLFreeHandle 代替。有关详细信息请参阅 SQLFreeHandle。

4.6.15 SQLFreeHandle

功能描述

释放与指定环境、连接、语句或描述符相关联的资源，它替代了 ODBC 2.x 函数 SQLFreeEnv、SQLFreeConnect 及 SQLFreeStmt。

原型

```
SQLRETURN SQLFreeHandle(SQLSMALLINT HandleType,
SQLHANDLE     Handle);
```

参数

表 4-15 SQLFreeHandle 参数

关键字	参数说明
HandleType	SQLFreeHandle 要释放的句柄类型。必须为下列值之一： ● SQL_HANDLE_ENV ● SQL_HANDLE_DBC ● SQL_HANDLE_STMT ● SQL_HANDLE_DESC 如果 HandleType 不是这些值之一，SQLFreeHandle 返回 SQL_INVALID_HANDLE。
Handle	要释放的句柄。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

如果 SQLFreeHandle 返回 SQL_ERROR，句柄仍然有效。

示例

参见示例。

4.6.16 SQLFreeEnv

在 ODBC 3.x 版本中，ODBC 2.x 的函数 SQLFreeEnv 已被 SQLFreeHandle 代替。有关详细信息请参阅 SQLFreeHandle。

4.6.17 SQLPrepare

功能描述

准备一个将要进行的 SQL 语句。

原型

```
SQLRETURN SQLPrepare(SQLHSTMT    StatementHandle,
SQLCHAR    *StatementText,
SQLINTEGER TextLength);
```

参数

表 4-16 SQLPrepare 参数

关键字	参数说明
StatementHandle	语句句柄。
StatementText	SQL 文本串。
TextLength	StatementText 的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLPrepare 返回的值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数分别设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.18 SQLGetData

功能描述

SQLGetData 返回结果集中某一列的数据。可以多次调用该接口，以便检索部分不定长度的数据。

原型

```
SQLRETURN    SQLGetData(SQLHSTMT      StatementHandle,
SQLUSMALLINT Col_or_Param_Num,
SQLSMALLINT  TargetType,
SQLPOINTER           TargetValuePtr,
SQLLEN       BufferLength,
SQLLEN       *StrLen_or_IndPtr);
```

参数

表 4-17 SQLGetData 参数

关键字	参数说明
StatementHandle	语句句柄，通过 SQLAllocHandle 获得。
Col_or_Param_Nu m	要返回数据的列号。结果集的列按增序从 1 开始编号。书签列的列号为 0。
TargetType	TargetValuePtr 缓冲中的 C 数据类型的类型标识符。若 TargetType 为 SQL_ARD_TYPE，驱动使用 ARD 中 SQL_DESC_CONCISE_TYPE 字段的类型标识符。若为 SQL_C_DEFAULT，驱动根据源的 SQL 数据类型选择缺省的数据类型。
TargetValuePtr	输出参数：指向返回数据所在缓冲区的指针。
BufferLength	TargetValuePtr 所指向缓冲区的长度。
StrLen_or_IndPtr	输出参数：指向缓冲区的指针，在此缓冲区中返回长度或标识符的值。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_NO_DATA：表示 SQL 语句不返回结果集。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。
- SQL_STILL_EXECUTING：表示语句正在执行。

注意事项

当调用 SQLGetData 函数返回 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过调用 SQLGetDiagRec 函数，并将 HandleType 和 Handle 参数分别设置为 SQL_HANDLE_STMT 和 StatementHandle，可得到一个相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.19 SQLGetDiagRec

功能描述

返回诊断记录的多个字段的当前值，其中诊断记录包含错误、警告及状态信息。

原型

```
SQLRETURN SQLGetDiagRec(SQLSMALLINT HandleType,
SQLHANDLE     Handle,
SQLSMALLINT RecNumber,
SQLCHAR   *SQLState,
SQLINTEGER          *NativeErrorPtr,
SQLCHAR   *MessageText,
SQLSMALLINT  BufferLength,
SQLSMALLINT *TextLengthPtr);
```

参数

表 4-18 SQLGetDiagRec 参数

关键字	参数说明
HandleType	句柄类型标识符，它说明诊断所要求的句柄类型。必须为下列值之一： ● SQL_HANDLE_ENV ● SQL_HANDLE_DBC ● SQL_HANDLE_STMT ● SQL_HANDLE_DESC
Handle	诊断数据结构的句柄，其类型由 HandleType 来指出。如果 HandleType 是 SQL_HANDLE_ENV，Handle 可以是共享的或

	非共享 的环境句柄。
RecNumber	指出应用从查找信息的状态记录。状态记录从 1 开始编号。
NativeErrorPt r	输出参数：指向缓冲区的指针，该缓冲区存储着本地的错误码。
MessageText	指向缓冲区的指针，该缓冲区存储着诊断信息文本串。
BufferLength	MessageText 的长度。
TextLengthPt r	输出参数：指向缓冲区的指针，返回 MessageText 中的字节总数。如果返回字节数大于 BufferLength，则 MessageText 中的诊断信息文本被截断成 BufferLength 减去 NULL 结尾字符的长度。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

SQLGetDiagRec 不返回诊断记录，而是通过以上返回值，来报告执行结果。

当调用 ODBC 函数返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，可以通过调用 SQLGetDiagRec 返回的诊断信息值 SQLSTATE，查看失败或警告原因。SQLSTATE 值如下表所示。

表 8-29 SQLSTATE 值

SQLSATATE	错误	描述
HY000	一般错误	未定义特定的 SQLSTATE 所产生的一个错误。
HY001	内存分配错误	驱动程序不能分配所需要的内存来支持函数的执行或完成。
HY008	取消操作	调用 SQLCancel 取消执行语句后，依然在 StatementHandle 上调用函数。

HY010	函数系列错误	在为执行中的所有数据参数或列发送数据前就调用了执行函数。
HY013	内存管理错误	不能处理函数调用，可能由当前内存条件差引起。
HYT01	连接超时	数据源响应请求之前，连接超时。
IM001	驱动程序不支持此函数	调用了 StatementHandle 相关的驱动程序不支持的函数

示例

参见示例。

4.6.20 SQLSetConnectAttr

功能描述

设置控制连接各方面的属性。

原型

```
SQLRETURN SQLSetConnectAttr(SQLHDBC ConnectionHandle,
SQLINTEGER Attribute,
SQLPOINTER ValuePtr, SQLINTEGER StringLength);
```

参数

表 4-19 SQLSetConnectAttr 参数

关键字	参数说明
EnvironmentHan dle	环境句柄。
Attribute	需设置的环境属性，可为如下值： ● SQL_ATTR_ODBC_VERSION：指定 ODBC 版本。 ● SQL_CONNECTION_POOLING：连接池属性。 ● SQL_OUTPUT_NTS：指明驱动器返回字符串的形式。
ValuePtr	指向对应 Attribute 的值。依赖于 Attribute 的值，ValuePtr 可能是 32 位整型值，或为以空结束的字符串。
StringLength	如果 ValuePtr 指向字符串或二进制缓冲区，这个参数是

	*ValuePtr 长度，如果 ValuePtr 指向整型，忽略 StringLength。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLSetEnvAttr 的返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过借助 SQL_HANDLE_ENV 的 HandleType 和 EnvironmentHandle 的 Handle，调用 SQLGetDiagRec 可得到相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.21 SQLSetEnvAttr

功能描述

设置控制环境各方面的属性。

原型

```
SQLRETURN SQLSetEnvAttr(SQLHENV        EnvironmentHandle,
SQLINTEGER Attribute,
SQLPOINTER ValuePtr, SQLINTEGER StringLength);
```

参数

表 4-20 SQLSetEnvAttr 参数

关键字	参数说明
EnvironmentHan dle	环境句柄。
Attribute	需设置的环境属性，可为如下值： ● SQL_ATTR_ODBC_VERSION：指定 ODBC 版本。

	● SQL_CONNECTION_POOLING：连接池属性。 ● SQL_OUTPUT_NTS：指明驱动器返回字符串的形式。
ValuePtr	指向对应 Attribute 的值。依赖于 Attribute 的值，ValuePtr 可能是 32 位整型值，或为以空结束的字符串。
StringLength	如果 ValuePtr 指向字符串或二进制缓冲区，这个参数是 *ValuePtr 长度，如果 ValuePtr 指向整型，忽略 StringLength。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLSetEnvAttr 的返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过借助 SQL_HANDLE_ENV 的 HandleType 和 EnvironmentHandle 的 Handle，调用 SQLGetDiagRec 可得到相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.22 SQLSetStmtAttr

功能描述

设置相关语句的属性。

原型

```
SQLRETURN SQLSetStmtAttr(SQLHSTMT    StatementHandle,
SQLINTEGER Attribute,
SQLPOINTER ValuePtr, SQLINTEGER StringLength);
```

参数

表 4-21 SQLSetStmtAttr 参数

关键字	参数说明
StatementHandle	语句句柄。
Attribute	需设置的属性。
ValuePtr	指向对应 Attribute 的值。依赖于 Attribute 的值，ValuePtr 可能是 32 位无符号整型值，或指向以空结束的字符串，二进制缓冲区，或者驱动定义值。注意，如果 ValuePtr 参数是驱动程序指定值。ValuePtr 可能是有符号的整数。
StringLength	如果 ValuePtr 指向字符串或二进制缓冲区，这个参数是 *ValuePtr 长度，如果 ValuePtr 指向整型，忽略 StringLength。

返回值

- SQL_SUCCESS：表示调用正确。
- SQL_SUCCESS_WITH_INFO：表示会有一些警告信息。
- SQL_ERROR：表示比较严重的错误，如：内存分配失败、建立连接失败等。
- SQL_INVALID_HANDLE：表示调用无效句柄。其他 API 的返回值同理。

注意事项

当 SQLSetStmtAttr 的返回值为 SQL_ERROR 或 SQL_SUCCESS_WITH_INFO 时，通过借助 SQL_HANDLE_STMT 的 HandleType 和 StatementHandle 的 Handle，调用 SQLGetDiagRec 可得到相关的 SQLSTATE 值，通过 SQLSTATE 值可以查出调用此函数的具体信息。

示例

参见示例。

4.6.23 示例

常用功能示例代码

```
// 此示例演示如何通过 ODBC 方式获取 GBase 8s 数据库中的数据。
// DBtest.c (compile with: libodbc.so) #include <stdlib.h>
#include <stdio.h> #include <sqlext.h> #ifdef WIN32 #include <windows.h> #endif
```

```
SQLHENV   V_OD_Env;   // Handle ODBC environment SQLHSTMT        V_OD_hstmt;
          // Handle statement SQLHDBC     V_OD_hdbc;         // Handle connection
char      typename[100]; SQLINTEGER value = 100;
SQLINTEGER V_OD_erg,V_OD_buffer,V_OD_err,V_OD_id; int main(int argc,char *argv[])
{
// 1. 申请环境句柄
V_OD_erg = SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&V_OD_Env);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
printf("Error AllocHandle\n"); exit(0);
}
// 2. 设置环境属性（版本信息）
SQLSetEnvAttr(V_OD_Env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
// 3. 申请连接句柄
V_OD_erg = SQLAllocHandle(SQL_HANDLE_DBC, V_OD_Env, &V_OD_hdbc);
if ((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
SQLFreeHandle(SQL_HANDLE_ENV, V_OD_Env); exit(0);
}
// 4. 设置连接属性
SQLSetConnectAttr(V_OD_hdbc, SQL_ATTR_AUTOCOMMIT, SQL_AUTOCOMMIT_ON,
0);
// 5. 连接数据源，这里的“userName”与“password”分别表示连接数据库的用户名和用户密码，请根据实际情况修改。
// 如果 odbc.ini 文件中已经配置了用户名密码，那么这里可以留空（""）；但是不建议这么做，因为一旦
odbc.ini 权限管理不善，将导致数据库用户密码泄露。
V_OD_erg = SQLConnect(V_OD_hdbc, (SQLCHAR*) "gaussdb", SQL_NTS,
(SQLCHAR*) "userName", SQL_NTS, (SQLCHAR*) "password", SQL_NTS); if
((V_OD_erg != SQL_SUCCESS) && (V_OD_erg != SQL_SUCCESS_WITH_INFO))
{
printf("Error SQLConnect %d\n",V_OD_erg); SQLFreeHandle(SQL_HANDLE_ENV,
V_OD_Env); exit(0);
}
printf("Connected !\n");
// 6. 设置语句属性
SQLSetStmtAttr(V_OD_hstmt,SQL_ATTR_QUERY_TIMEOUT,(SQLPOINTER *)3,0);
// 7. 申请语句句柄
SQLAllocHandle(SQL_HANDLE_STMT, V_OD_hdbc, &V_OD_hstmt);
// 8. 直接执行 SQL 语句。
```

```
SQLExecDirect(V_OD_hstmt,"drop table IF EXISTS customer_t1",SQL_NTS);
SQLExecDirect(V_OD_hstmt,"CREATE TABLE customer_t1(c_customer_sk INTEGER,
c_customer_name
VARCHAR(32));",SQL_NTS);
SQLExecDirect(V_OD_hstmt,"insert into customer_t1 values(25,li)",SQL_NTS);
// 9. 准备执行
SQLPrepare(V_OD_hstmt,"insert into customer_t1 values(?)",SQL_NTS);
// 10. 绑定参数
SQLBindParameter(V_OD_hstmt,1,SQL_PARAM_INPUT,SQL_C_SLONG,SQL_INTEGER,
0,0, &value,0,NULL);
// 11. 执行准备好的语句
SQLExecute(V_OD_hstmt);
SQLExecDirect(V_OD_hstmt,"select id from testtable",SQL_NTS);
// 12. 获取结果集某一列的属性
SQLColAttribute(V_OD_hstmt,1,SQL_DESC_TYPE,typename,100,NULL,NULL);
printf("SQLColAtrribute %s\n",typename);
// 13. 绑定结果集
SQLBindCol(V_OD_hstmt,1,SQL_C_SLONG, (SQLPOINTER)&V_OD_buffer,150,
(SQLLEN *)&V_OD_err);
// 14. 通过 SQLFetch 取结果集中数据
V_OD_erg=SQLFetch(V_OD_hstmt);
// 15. 通过 SQLGetData 获取并返回数据。
while(V_OD_erg != SQL_NO_DATA)
{
SQLGetData(V_OD_hstmt,1,SQL_C_SLONG,(SQLPOINTER)&V_OD_id,0,NULL);
printf("SQLGetData     ID = %d\n",V_OD_id);
V_OD_erg=SQLFetch(V_OD_hstmt);
};
printf("Done !\n");
// 16. 断开数据源连接并释放句柄资源
SQLFreeHandle(SQL_HANDLE_STMT,V_OD_hstmt); SQLDisconnect(V_OD_hdbc);
SQLFreeHandle(SQL_HANDLE_DBC,V_OD_hdbc); SQLFreeHandle(SQL_HANDLE_ENV,
V_OD_Env); return(0);
}
```

批量绑定示例代码

```
/*****************************************************************
*请在数据源中打开 UseBatchProtocol，同时指定数据库中参数 support_batch_bind
*为 on
*CHECK_ERROR 的作用是检查并打印错误信息。
```

```
*此示例将与用户交互式获取 DSN、模拟的数据量，忽略的数据量，并将最终数据入库到 test_odbc_batch_insert 中
**********************************************************************/
#include <stdio.h>
#include <stdlib.h> #include <sql.h> #include <sqlext.h> #include <string.h>
#include "util.c"
void Exec(SQLHDBC hdbc, SQLCHAR* sql)
{
SQLRETURN retcode; // Return status
SQLHSTMT hstmt = SQL_NULL_HSTMT; // Statement handle SQLCHAR  loginfo[2048];
// Allocate Statement Handle
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt); CHECK_ERROR(retcode,
"SQLAllocHandle(SQL_HANDLE_STMT)",
hstmt, SQL_HANDLE_STMT);
// Prepare Statement
retcode = SQLPrepare(hstmt, (SQLCHAR*) sql, SQL_NTS); sprintf((char*)loginfo,
"SQLPrepare log: %s", (char*)sql); CHECK_ERROR(retcode, loginfo, hstmt,
SQL_HANDLE_STMT);
// Execute Statement
retcode = SQLExecute(hstmt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql); CHECK_ERROR(retcode,
loginfo, hstmt, SQL_HANDLE_STMT);
// Free Handle
retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmt); sprintf((char*)loginfo,
"SQLFreeHandle stmt log: %s", (char*)sql); CHECK_ERROR(retcode, loginfo, hstmt,
SQL_HANDLE_STMT);
}
int main ()
{
SQLHENV henv = SQL_NULL_HENV; SQLHDBC hdbc = SQL_NULL_HDBC;
int  batchCount = 1000; SQLLEN rowsCount = 0; int ignoreCount = 0;
SQLRETURN retcode; SQLCHAR    dsn[1024] = {'\0'};
SQLCHAR   loginfo[2048];
// 交互获取数据源名称
getStr("Please input your DSN", (char*)dsn, sizeof(dsn), 'N');
// 交互获取批量绑定的数据量 getInt("batchCount", &batchCount, 'N', 1); do
{
// 交互获取批量绑定的数据中，不要入库的数据量
getInt("ignoreCount", &ignoreCount, 'N', 1); if (ignoreCount > batchCount)
{
printf("ignoreCount(%d) should be less than batchCount(%d)\n", ignoreCount, batchCount);
```

```
}
}while(ignoreCount > batchCount);
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
CHECK_ERROR(retcode, "SQLAllocHandle(SQL_HANDLE_ENV)",
henv, SQL_HANDLE_ENV);
// Set ODBC Verion
retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
(SQLPOINTER*)SQL_OV_ODBC3, 0);
CHECK_ERROR(retcode, "SQLSetEnvAttr(SQL_ATTR_ODBC_VERSION)", henv,
SQL_HANDLE_ENV);
// Allocate Connection
retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc); CHECK_ERROR(retcode,
"SQLAllocHandle(SQL_HANDLE_DBC)",
henv, SQL_HANDLE_DBC);
// Set Login Timeout
retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)5, 0);
CHECK_ERROR(retcode, "SQLSetConnectAttr(SQL_LOGIN_TIMEOUT)",
hdbc, SQL_HANDLE_DBC);
// Set Auto Commit
retcode = SQLSetConnectAttr(hdbc, SQL_ATTR_AUTOCOMMIT,
(SQLPOINTER)(1), 0);
CHECK_ERROR(retcode, "SQLSetConnectAttr(SQL_ATTR_AUTOCOMMIT)", hdbc,
SQL_HANDLE_DBC);
// Connect to DSN
sprintf(loginfo, "SQLConnect(DSN:%s)", dsn);
retcode = SQLConnect(hdbc, (SQLCHAR*) dsn, SQL_NTS,
(SQLCHAR*) NULL, 0, NULL, 0);
CHECK_ERROR(retcode, loginfo, hdbc, SQL_HANDLE_DBC);
// init table info.
Exec(hdbc, "drop table if exists test_odbc_batch_insert");
Exec(hdbc, "create table test_odbc_batch_insert(id int primary key, col varchar2(50))");
// 下面的代码根据用户输入的数据量，构造出将要入库的数据:
{
SQLRETURN retcode;
SQLHSTMT hstmtinesrt = SQL_NULL_HSTMT; int       i;
SQLCHAR    *sql = NULL; SQLINTEGER *ids = NULL; SQLCHAR     *cols = NULL;
SQLLEN      *bufLenIds = NULL; SQLLEN     *bufLenCols = NULL; SQLUSMALLINT
*operptr = NULL; SQLUSMALLINT *statusptr = NULL; SQLULEN           process = 0;
// 这里是按列构造，每个字段的内存连续存放在一起。
ids = (SQLINTEGER*)malloc(sizeof(ids[0]) * batchCount); cols =
(SQLCHAR*)malloc(sizeof(cols[0]) * batchCount * 50);
```

```
// 这里是每个字段中，每一行数据的内存长度。
bufLenIds = (SQLLEN*)malloc(sizeof(bufLenIds[0]) * batchCount); bufLenCols =
(SQLLEN*)malloc(sizeof(bufLenCols[0]) * batchCount);
// 该行是否需要被处理，SQL_PARAM_IGNORE 或 SQL_PARAM_PROCEED operptr =
(SQLUSMALLINT*)malloc(sizeof(operptr[0]) * batchCount); memset(operptr, 0,
sizeof(operptr[0]) * batchCount);
// 该行的处理结果。
// 注：由于数据库中处理方式是同一语句隶属同一事务中，所以如果出错，那么待处理数
据都将是出错的， 并不会部分入库。
statusptr = (SQLUSMALLINT*)malloc(sizeof(statusptr[0]) * batchCount); memset(statusptr,
88, sizeof(statusptr[0]) * batchCount);
if (NULL == ids || NULL == cols || NULL == bufLenCols || NULL == bufLenIds)
{
fprintf(stderr, "FAILED:\tmalloc data memory failed\n"); goto exit;
}
for (int i = 0; i < batchCount; i++)
{
ids[i] = i;
sprintf(cols + 50 * i, "column test value %d", i); bufLenIds[i] = sizeof(ids[i]);
bufLenCols[i] = strlen(cols + 50 * i);
operptr[i] = (i < ignoreCount) ? SQL_PARAM_IGNORE : SQL_PARAM_PROCEED;
}
// Allocate Statement Handle
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmtinesrt);
CHECK_ERROR(retcode, "SQLAllocHandle(SQL_HANDLE_STMT)",
hstmtinesrt, SQL_HANDLE_STMT);
// Prepare Statement
sql = (SQLCHAR*)"insert into test_odbc_batch_insert values(?, ?)"; retcode =
SQLPrepare(hstmtinesrt, (SQLCHAR*) sql, SQL_NTS); sprintf((char*)loginfo, "SQLPrepare
log: %s", (char*)sql); CHECK_ERROR(retcode, loginfo, hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMSET_SIZE,
(SQLPOINTER)batchCount, sizeof(batchCount));
CHECK_ERROR(retcode, "SQLSetStmtAttr", hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLBindParameter(hstmtinesrt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, sizeof(ids[0]), 0,&(ids[0]), 0, bufLenIds);
CHECK_ERROR(retcode, "SQLBindParameter for id", hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLBindParameter(hstmtinesrt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 50, 50,
cols, 50, bufLenCols);
CHECK_ERROR(retcode, "SQLBindParameter for cols", hstmtinesrt,
SQL_HANDLE_STMT);
```

```
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAMS_PROCESSED_PTR,
(SQLPOINTER)&process, sizeof(process));
CHECK_ERROR(retcode, "SQLSetStmtAttr for SQL_ATTR_PARAMS_PROCESSED_PTR",
hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_STATUS_PTR,
(SQLPOINTER)statusptr, sizeof(statusptr[0]) * batchCount);
CHECK_ERROR(retcode, "SQLSetStmtAttr for SQL_ATTR_PARAM_STATUS_PTR",
hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLSetStmtAttr(hstmtinesrt, SQL_ATTR_PARAM_OPERATION_PTR,
(SQLPOINTER)operptr, sizeof(operptr[0]) * batchCount);
CHECK_ERROR(retcode, "SQLSetStmtAttr for SQL_ATTR_PARAM_OPERATION_PTR",
hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLExecute(hstmtinesrt);
sprintf((char*)loginfo, "SQLExecute stmt log: %s", (char*)sql); CHECK_ERROR(retcode,
loginfo, hstmtinesrt, SQL_HANDLE_STMT);
retcode = SQLRowCount(hstmtinesrt, &rowsCount);
CHECK_ERROR(retcode, "SQLRowCount execution", hstmtinesrt, SQL_HANDLE_STMT);
if (rowsCount != (batchCount - ignoreCount))
{
sprintf(loginfo, "(batchCount - ignoreCount)(%d) != rowsCount(%d)", (batchCount -
ignoreCount), rowsCount);
CHECK_ERROR(SQL_ERROR, loginfo, NULL, SQL_HANDLE_STMT);
}
else
{
sprintf(loginfo, "(batchCount - ignoreCount)(%d) == rowsCount(%d)", (batchCount -
ignoreCount), rowsCount);
CHECK_ERROR(SQL_SUCCESS, loginfo, NULL, SQL_HANDLE_STMT);
}
// check row number returned if (rowsCount != process)
{
sprintf(loginfo, "process(%d) != rowsCount(%d)", process, rowsCount);
CHECK_ERROR(SQL_ERROR, loginfo, NULL, SQL_HANDLE_STMT);
}
else
{
sprintf(loginfo, "process(%d) == rowsCount(%d)", process, rowsCount);
CHECK_ERROR(SQL_SUCCESS, loginfo, NULL, SQL_HANDLE_STMT);
}
for (int i = 0; i < batchCount; i++)
{
```

```
if (i < ignoreCount)
{
if (statusptr[i] != SQL_PARAM_UNUSED)
{
sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_UNUSED", i, statusptr[i]);
CHECK_ERROR(SQL_ERROR, loginfo, NULL, SQL_HANDLE_STMT);
}
}
else if (statusptr[i] != SQL_PARAM_SUCCESS)
{
sprintf(loginfo, "statusptr[%d](%d) != SQL_PARAM_SUCCESS", i, statusptr[i]);
CHECK_ERROR(SQL_ERROR, loginfo, NULL, SQL_HANDLE_STMT);
}
}
retcode = SQLFreeHandle(SQL_HANDLE_STMT, hstmtinesrt); sprintf((char*)loginfo,
"SQLFreeHandle hstmtinesrt"); CHECK_ERROR(retcode, loginfo, hstmtinesrt,
SQL_HANDLE_STMT);
}
exit:
printf ("\nComplete.\n");
// Connection
if (hdbc != SQL_NULL_HDBC) { SQLDisconnect(hdbc);
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
}
// Environment
if (henv != SQL_NULL_HENV) SQLFreeHandle(SQL_HANDLE_ENV, henv);
return 0;
}
```

5 基于 libpq 开发

libpq 是 GBase 8s 的 C 应用程序接口。libpq 是一套允许客户程序向服务器服务进程发送查询并且获得查询返回的库函数。同时也是其他几个 GBase 8s 应用接口下面的引擎，如 ODBC 等依赖的库文件。本章给出两个示例，显示如何利用 libpq 编写代码。

5.1 libpq 使用依赖的头文件

使用 libpq 的前端程序必须包括头文件 libpq-fe.h，并且必须与 libpq 库链接。

5.2 开发流程

编译并且链接一个 libpq 的源程序，需要做下面的一些事情：

(1) 解压相应的发布包文件。其中 include 文件夹下的头文件为所需的头文件，lib 文件夹中为所需的 libpq 库文件。

说明

- 除 libpq-fe.h 外，include 文件夹下默认还存在头文件 postgres_ext.h，gs_thread.h，gs_threadlocal.h，这三个头文件是 libpq-fe.h 的依赖文件。

(2) 包含 libpq-fe.h 头文件：

```
#include <libpq-fe.h>
```

(3) 通过-I directory 选项，提供头文件的安装位置（有些时候编译器会查找缺省的目录，因此可以忽略这些选项）。如：

```
gcc -I (头文件所在目录) -L (libpq 库所在目录) testprog.c -lpq
```

(4) 如果要使用 makefile 命令，向 CPPFLAGS、LDFLAGS、LIBS 变量中增加如下选项：

```
CPPFLAGS += -I (头文件所在目录) LDFLAGS += -L (libpq 库所在目录) LIBS += -lpq
```

5.3 示例

示例 1

```
/*
* testlibpq.c
*/
#include <stdio.h> #include <stdlib.h> #include <libpq-fe.h>
static void exit_nicely(PGconn *conn)
```

```
{
PQfinish(conn); exit(1);
}
int
main(int argc, char **argv)
{
const char *conninfo;
PGconn       *conn; PGresult    *res; int     nFields;
int  i,j;
/*
用户在命令行上提供了 conninfo 字符串的值时使用该值
否则环境变量或者所有其它连接参数
都使用缺省值。
*/
if (argc > 1)
conninfo = argv[1]; else
conninfo = "dbname=postgres port=42121 host='10.44.133.171' application_name=test
connect_timeout=5 sslmode=allow user='test' password='test_1234'";
/*  连接数据库  */
conn = PQconnectdb(conninfo);
/*  检查后端连接成功建立  */
if (PQstatus(conn) != CONNECTION_OK)
{
fprintf(stderr, "Connection to database failed: %s", PQerrorMessage(conn));
exit_nicely(conn);
}
/*
测试实例涉及游标的使用时候必须使用事务块
*把全部放在一个  "select * from pg_database"
PQexec()  里，过于简单，不推荐使用
*/
/*  开始一个事务块  */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn)); PQclear(res);
exit_nicely(conn);
}
/*
在结果不需要的时候 PQclear PGresult，以避免内存泄漏
*/ PQclear(res);
```

```
/*
从系统表 pg_database（数据库的系统目录）里抓取数据
*/
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database"); if
(PQresultStatus(res) != PGRES_COMMAND_OK)
{
fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn)); PQclear(res);
exit_nicely(conn);
}
PQclear(res);
res = PQexec(conn, "FETCH ALL in myportal"); if (PQresultStatus(res) !=
PGRES_TUPLES_OK)
{
fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn)); PQclear(res);
exit_nicely(conn);
}
/* 打 印 属 性 名 称 */ nFields = PQnfields(res);
for (i = 0; i < nFields; i++)
printf("%-15s", PQfname(res, i)); printf("\n\n");
/* 打印行 */
for (i = 0; i < PQntuples(res); i++)
{
for (j = 0; j < nFields; j++)
printf("%-15s", PQgetvalue(res, i, j)); printf("\n");
}
PQclear(res);
/* 关闭入口 ... 不用检查错误 ... */
res = PQexec(conn, "CLOSE myportal"); PQclear(res);
/* 结束事务 */
res = PQexec(conn, "END"); PQclear(res);
/* 关闭数据库连接并清理 */ PQfinish(conn);
return 0;
}
```

示例 2

```
/*
testlibpq2.c
测试外联参数和二进制 I/O。
*
在运行这个例子之前，用下面的命令填充一个数据库
*
```

```
*
CREATE TABLE test1 (i int4, t text);
*
INSERT INTO test1 values (2, 'ho there');
*
期望的输出如下
*
*
tuple 0: got
i = (4 bytes) 2
t = (8 bytes) 'ho there'
*
*/
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <sys/types.h> #include
<libpq-fe.h>
/* for ntohl/htonl */ #include <netinet/in.h> #include <arpa/inet.h>
static void exit_nicely(PGconn *conn)
{
PQfinish(conn); exit(1);
}
/*
这个函数打印查询结果，这些结果是二进制格式，从上面的
注释里面创建的表中抓取出来的
*/
static void show_binary_results(PGresult *res)
{
int  i;
int         i_fnum, t_fnum;
/* 使用 PQfnumber 来避免对结果中的字段顺序进行假设 */ i_fnum = PQfnumber(res,
"i");
t_fnum = PQfnumber(res, "t");
for (i = 0; i < PQntuples(res); i++)
{
char       *iptr;
char       *tptr;
int  ival;
/* 获取字段值（忽略可能为空的可能） */ iptr = PQgetvalue(res, i, i_fnum);
tptr = PQgetvalue(res, i, t_fnum);
/*
INT4 的二进制表现形式是网络字节序
建议转换成本地字节序
```

```
*/
ival = ntohl(*((uint32_t *) iptr));
/*
TEXT 的二进制表现形式是文本，因此 libpq 能够给它附加一个字节零
把它看做 C 字符串
*
*/
printf("tuple %d: got\n", i); printf(" i = (%d bytes) %d\n",
PQgetlength(res, i, i_fnum), ival); printf(" t = (%d bytes) '%s'\n",
PQgetlength(res, i, t_fnum), tptr); printf("\n\n");
}
}
int
main(int argc, char **argv)
{
const char *conninfo;
PGconn *conn;
PGresult *res;
const char *paramValues[1]; int   paramLengths[1];
int  paramFormats[1]; uint32_t binaryIntVal;
/*
如果用户在命令行上提供了参数，
那么使用该值为 conninfo 字符串；否则
使用环境变量或者缺省值。
*/
if (argc > 1)
conninfo = argv[1]; else
conninfo = "dbname=postgres port=42121 host='10.44.133.171' application_name=test
connect_timeout=5 sslmode=allow user='test' password='test_1234'";
/* 和数据库建立连接 */
conn = PQconnectdb(conninfo);
/* 检查与服务器的连接是否成功建立 */
if (PQstatus(conn) != CONNECTION_OK)
{
fprintf(stderr, "Connection to database failed: %s", PQerrorMessage(conn));
exit_nicely(conn);
}
/* 把整数值 "2" 转换成网络字节序 */ binaryIntVal = htonl((uint32_t) 2);
/* 为 PQexecParams 设置参数数组 */ paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal); paramFormats[0] = 1; /* 二进制 */
res = PQexecParams(conn,
```

```
"SELECT * FROM test1 WHERE i = $1::int4",
1,  /*  一个参数  */
NULL, /*  让后端推导参数类型  */ paramValues,
paramLengths, paramFormats,
1); /*  要求二进制结果  */
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn)); PQclear(res);
exit_nicely(conn);
}
show_binary_results(res);
PQclear(res);
/*  关闭与数据库的连接并清理  */ PQfinish(conn);
return 0;
}
```

5.4 libpq 接口参数

5.4.1 数据库连接控制函数

数据库连接控制函数控制与数据库服务器链接的事情。一个应用程序一次可以与多个服务器建立链接，如一个客户端链接多个数据库的场景。每个链接都是用一个从函数 PQconnectdb、PQconnectdbParams 或 PQsetdbLogin 获得的 PGconn 对象表示。注意，这些函数总是返回一个非空的对象指针，除非内存分配失败，会返回一个空的指针。链接建立的接口保存在 PGconn 对象中，可以调用 PQstatus 函数来检查一下返回值看看连接是否成功。

5.4.1.1 PQconnectdbParams

功能描述

与数据库服务器建立一个新的连接。

原型

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

参数

表 5-1 PQconnectdbParams 参数

关键字	参数说明
keywords	定义为一个字符串的数组，每个都成为一个关键字。
values	给每个关键字一个值。
expand_dbname	当 expand_dbname 是非零的时，允许将 dbname 的关键字值看做一个连接字符串。只有第一个出现的 dbname 是这样展开的， 任何随后的 dbname 值作为纯数据库名处理。

返回值

PGconn *：指向包含链接的对象指针，内存在函数内部申请。

注意事项

这个函数用从两个 NULL 结束的数组中来的参数，打开一个新的数据库连接。与 PQsetdbLogin 不同的是，不必更换函数名就可扩展参数集。因此，建议应用程序中使用这个函数或类似变种函数 PQconnectStartParams 和 PQconnectPoll。

示例

参见示例。

5.4.1.2 PQconnectdb

功能描述

与数据库服务器建立一个新的连接。

原型

```
PGconn *PQconnectdb(const char *conninfo);
```

参数

表 5-2 PQconnectdb 参数

关键字	参数说明
conninfo	链接字符串，字符串中的字段见 6.4.5 链接参数章节。

返回值

PGconn *：指向包含链接的对象指针，内存在函数内部申请。

注意事项

- 这个函数用从一个字符串 conninfo 来的参数与数据库打开一个新的链接。
- 传入的参数可以为空，表明使用所有缺省的参数，或者可以包含一个或更多个用空白间隔的参数设置，或者它可以包含一个 URL。

示例

参见示例。

5.4.1.3 PQconninfoParse

功能描述

根据连接，返回已解析的连接选项。

原型

```
PQconninfoOption* PQconninfoParse(const char* conninfo, char**    errmsg);
```

参数

表 5-3 PQconninfoParse 参数

关键字	参数说明
conninfo	被传递的字符串。可以为空，这样将会使用默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个 URI。
errmsg	错误信息。

返回值

PQconninfoOption 类型指针。

5.4.1.4 PQconnectStart

功能描述

与数据库服务器建立一次非阻塞的连接。

原型

```
PGconn* PQconnectStart(const char* conninfo);
```

参数

表 5-4 PQconnectStart 参数

关键字	参数说明
conninfo	连接信息字符串。可以为空，这样将会使用默认参数。也可以包含由空格分隔的一个或多个参数设置，还可以包含一个 URL。

返回值

PGconn 类型指针。

5.4.1.5 PQerrorMessage

功能描述

返回连接上的错误信息。

原型

```
char* PQerrorMessage(const PGconn*    conn);
```

参数

表 5-5 PQerrorMessage 参数

关键字	参数说明
conn	连接句柄

返回值

char 类型指针。

示例

参见示例。

5.4.1.6 PQsetdbLogin

功能描述

与数据库服务器建立一个新的链接。

原型

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
```

```
                const char *login,
                const char *pwd);
```

参数

表 5-6 PQsetdbLogin 参数

关键字	参数说明
pghost	要链接的主机名，详见链接参数章节描述的 host 字段。
pgport	主机服务器的端口号，详见链接参数描述的 port 字段。
pgoptions	添加命令行选项以在运行时发送到服务器，详见链接参数描述的 options 字段。
pgtty	忽略（以前，这个选项声明服务器日志的输出方向）。
dbName	要链接的数据库名，详见链接参数描述的 dbname 字段。
login	要链接的用户名，详见链接参数章节描述的 user 字段。
pwd	如果服务器要求口令认证，所用的口令，详见链接参数描述的 password 字段。

返回值

PGconn *：指向包含链接的对象指针，内存在函数内部申请。

注意事项

- 该函数为 PQconnectdb 前身，参数个数固定，未定义参数被调用时使用缺省值， 若需要给固定参数设置缺省值，则可赋值 NULL 或者空字符串。
- 若 dbName 中包含“=”或链接 URL 的有效前缀，则该 dbName 被看做一个 conninfo 字符串并传递至 PQconnectdb 中，其余参数与 PQconnectdbParams 保持一致。

示例

参见示例。

5.4.1.7 PQfinish

功能描述

关闭与服务器的连接，同时释放被 PGconn 对象使用的存储器。

原型

```
void    PQfinish(PGconn *conn);
```

参数

表 5-7 PQfinish 参数

关键字	参数说明
conn	指向包含链接的对象指针。

注意事项

若 PQstatus 判断服务器链接失败，应用程序调用 PQfinish，释放被 PGconn 对象使用的存储器。当调用 PQfinish 后，PGconn 指针不可重复使用。

示例

参见示例。

5.4.1.8 PQreset

功能描述

重置与服务器的通讯端口。

原型

```
void PQreset(PGconn *conn);
```

参数

表 5-8 PQreset 参数

关键字	参数说明
conn	指向包含链接的对象指针

注意事项

使用此函数，关闭与服务器的连接，并使用之前使用过的参数，重建与该服务器的连接。该函数适用于在链接异常后进行故障恢复的场景。

示例

参见示例。

5.4.1.9 PQstatus

功能描述

返回链接的状态。

原型

```
ConnStatusType PQstatus(const PGconn *conn);
```

参数

表 5-9 PQstatus 参数

关键字	参数说明
conn	指向包含链接的对象指针。

返回值

ConnStatusType：链接状态的枚举，包括：

```
CONNECTION_STARTED
等待进行连接。
CONNECTION_MADE
连接成功；等待发送。
CONNECTION_AWAITING_RESPONSE
等待来自服务器的响应。
CONNECTION_AUTH_OK
已收到认证；等待后端启动结束。
CONNECTION_SSL_STARTUP
协商 SSL 加密。
CONNECTION_SETENV
协商环境驱动的参数设置。
CONNECTION_OK
链接正常。
CONNECTION_BAD
链接故障。
```

注意事项

状态可以是多个值之一。但是，在异步连接过程之外只能看到其中两个：CONNECTION_OK 和 CONNECTION_BAD。与数据库的良好连接状态为 CONNECTION_OK。状态表示连接尝试失败 CONNECTION_BAD。通常，“正常”状态将

一直保持到 PQfinish，但通信失败可能会导致状态 CONNECTION_BAD 过早变为。在这种情况下，应用程序可以尝试通过调用进行恢复 PQreset。

示例

参见示例。

5.4.2 数据库执行语句函数

在成功建立与数据库服务器的连接后，可以使用数据库执行语句函数，用以执行 SQL 查询和命令。

5.4.2.1 PQclear

功能描述

释放与 PGresult 相关联的存储空间，任何不再需要的查询结果都应该用 PQclear 释放。

原型

```
void PQclear(PGresult *res);
```

参数

表 5-10 PQclear 参数

关键字	参数说明
res	包含查询结果的对象指针。

注意事项

PGresult 不会自动释放，当提交新的查询时它并不消失，甚至断开连接后也不会。要删除它，必须调用 PQclear，否则则会有内存泄漏。

示例

参见示例。

5.4.2.2 PQexec

功能描述

向服务器提交一条命令并等待结果。

原型

```
PGresult *PQexec(PGconn *conn, const char *command);
```

参数

表 5-11 PQexec 参数

关键字	参数说明
conn	指向包含链接的对象指针。
command	需要执行的查询字符串。

返回值

PGresult：包含查询结果的对象指针。

注意事项

应该调用 PQresultStatus 函数来检查任何错误的返回值（包括空指针的值，在这种情况下它将返回 PGRES_FATAL_ERROR）。使用 PQerrorMessage 获取有关错误的更多信息。

须知

命令字符串可以包括多个 SQL 命令（用分号分隔）。在一个 PQexec 调用中发送的多个查询是在一个事务里处理的，除非在查询字符串里有明确的 BEGIN/COMMIT 命令把整个字符串分隔成多个事务。请注意，返回的 PGresult 结构只描述字符串里执行的最后一条命令的结果，如果有一个命令失败，那么字符串处理的过程就会停止，并且返回的 PGresult 会描述错误条件。

示例

参见示例。

5.4.2.3 PQexecParams

功能描述

执行一个绑定参数的命令。

原型

```
PGresult* PQexecParams(PGconn* conn,
                       const char* command,
                       int nParams,
                       const Oid* paramTypes,
                       const char* const* paramValues,
```

```
                const int* paramLengths,
                const int* paramFormats,
                int resultFormat);
```

参数

表 5-12 PQexecParams 参数

关键字	参数说明
conn	连接句柄。
command	SQL 文本串。
nParams	绑定参数的个数
paramTypes	绑定参数类型。
paramValues	绑定参数的值。
paramLengths	参数长度。
paramFormats	参数格式（文本或二进制）。
resultFormat	返回结果格式（文本或二进制）。

返回值

PGresult 类型指针。

5.4.2.4 PQexecParamsBatch

功能描述

执行一个批量绑定参数的命令。

原型

```
PGresult* PQexecParamsBatch(PGconn* conn,
                const char* command,
                int nParams,
                int nBatch,
                const Oid* paramTypes,
                const char* const* paramValues,
                const int* paramLengths,
                const int* paramFormats,
```

```
                int resultFormat);
```

参数

表 5-13 PQexecParamsBatch 参数

关键字	参数说明
conn	连接句柄。
command	SQL 文本串。
nParams	绑定参数的个数
nBatch	批量操作数。
paramTypes	绑定参数类型。
paramValues	绑定参数的值。
paramLengths	参数长度。
paramFormats	参数格式（文本或二进制）。
resultFormat	返回结果格式（文本或二进制）。

返回值

PGresult 类型指针。

5.4.2.5 PQexecPrepared

功能描述

发送一个请求，以用给定参数执行预备语句并等待结果。

原型

```
PGresult* PQexecPrepared(PGconn* conn,
                         const char* stmtName,
                         int nParams,
                         const char* const* paramValues,
                         const int* paramLengths,
                         const int* paramFormats,
                         int resultFormat);
```

参数

表 5-14 PQexecPrepared 参数

关键字	参数说明
conn	连接句柄。
stmtName	stmt 名称，可以用""或者 NULL 来引用未命名语句，否则它必须是一个现有预备语句的名字。
nParams	参数个数。
paramValues	参数的实际值。
paramLengths	参数的实际数据长度。
paramFormats	参数的格式（文本或二进制）。
resultFormat	结果的格式（文本或二进制）。

返回值

PGresult 类型指针。

5.4.2.6 PQexecPreparedBatch

功能描述

发送一个请求，以用给定的批量参数执行预备语句并等待结果。

原型

```
PGresult* PQexecPreparedBatch(PGconn* conn,
                              const char* stmtName,
                              int nParams,
                              int nBatchCount,
                              const char* const* paramValues,
                              const int* paramLengths,
                              const int* paramFormats,
                              int resultFormat);
```

参数

表 5-15 PQexecPrepareBatch 参数

关键字	参数说明

conn	连接句柄。
stmtName	stmt 名称，可以用""或者 NULL 来引用未命名语句，否则它必须是一个现有预备语句的名字。
nParams	参数个数。
nBatchCount	批量数。
paramValues	参数的实际值。
paramLengths	参数的实际数据长度。
paramFormats	参数的格式（文本或二进制）。
resultFormat	结果的格式（文本或二进制）。

返回值

PGresult 类型指针。

5.4.2.7 PQfname

功能描述

返回与给定列号相关联的列名。列号从 0 开始。调用者不应该直接释放该结果。它将在相关的 PGresult 句柄被传递给 PQclear 之后被释放。

原型

```
char *PQfname(const PGresult *res, int column_number);
```

参数

表 5-16 PQfname 参数

关键字	参数说明
res	操作结果句柄
column_number	列数

返回值

char 类型指针。

示例

参见示例。

5.4.2.8 PQgetvalue

功能描述

返回一个 PGresult 的一行的单一域值。行和列号从 0 开始。调用者不应该直接释放该结果。它将在相关的 PGresult 句柄被传递给 PQclear 之后被释放。

原型

```
char *PQgetvalue(const PGresult *res,
int row_number,
int column_number);
```

参数

表 5-17 PQgetvalue 参数

关键字	参数说明
res	操作结果句柄
row_number	行数
column_number	列数

返回值

对于文本格式的数据，PQgetvalue 返回的值是该域值的一种空值结束的字符串表示。

对于二进制格式的数据，该值是由该数据类型的 typsend 和 typreceive 函数决定的二进制表示。

如果该域值为空，则返回一个空串。

示例

参见示例。

5.4.2.9 PQnfields

功能描述

返回查询结果中每一行的列数。

原型

```
int PQnfields(const PGresult *res);
```

参数

表 5-18 PQnfields 参数

关键字	参数说明
res	操作结果句柄

返回值

int 类型数字。

示例

参见示例。

5.4.2.10 PQntuples

功能描述

返回查询结果中的行（元组）数。由于返回的是整数结果，在 32 位操作系统上大数据量结果集，可能使返回值溢出。

原型

```
int PQntuples(const PGresult *res);
```

参数

表 5-19 PQntuples 参数

关键字	参数说明
res	操作结果句柄

返回值

int 类型数字

示例

参见示例。

5.4.2.11 PQprepare

功能描述

用给定的参数提交请求，创建一个预备语句，然后等待结束。

原型

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

参数

表 5-20 PQprepare 参数

关键字	参数说明
conn	指向包含链接的对象指针。
stmtName	需要执行的 stmt 名称。
query	需要执行的查询字符串。
nParams	参数个数。
paramTypes	声明参数类型的数组。

返回值

PGresult：包含查询结果的对象指针。

注意事项

- PQprepare 创建一个为 PQexecPrepared 执行用的预备语句，本特性支持命令的重复执行，不需要每次都进行解析和规划。PQprepare 仅在协议 3.0 及以后的连接中支持，使用协议 2.0 时，PQprepare 将失败。
- 该函数从查询字符串创建一个名为 stmtName 的预备语句，该查询字符串必须包 含一个 SQL 命令。stmtName 可以是""来创建一个未命名的语句，在这种情况下， 任何预先存在的未命名的语句都将被自动替换；否则，如果在当前会话中已经定义了语句名称，则这是一个错误。如果使用了任何参数，那么在查询中将它们称为$1,$2 等。nParams

是在 paramTypes[]数组中预先指定类型的参数的数量。(当 nParams 为 0 时，数组指针可以为 NULL) paramTypes[]通过 OID 指定要分配给参数符号的数据类型。如果 paramTypes 为 NULL ，或者数组中的任何特定元素为零，服务器将按照对非类型化字面字符串的相同方式为参数符号分配数据类型。另外，查询可以使用数字高于 nParams 的参数符号；还将推断这些符号的数据类型。

须知

通过执行 SQLPREPARE 语句，还可以创建与 PQexecPrepared 一起使用的预备语句。此外，虽然没有用于删除预备语句的 libpq 函数，但是 SQL DEALLOCATE 语句可用于此目的。

示例

参见示例。

5.4.2.12 PQresultStatus

功能描述

返回命令的结果状态。

原型

```
ExecStatusType PQresultStatus(const PGresult *res);
```

参数

表 5-21 PQresultStatus 参数-

关键字	参数说明
res	包含查询结果的对象指针。

返回值

PQresultStatus：命令执行结果的枚举，包括：

```
PQresultStatus 可以返回下面数值之一：
PGRES_EMPTY_QUERY
发送给服务器的字串是空的。
PGRES_COMMAND_OK
成功完成一个不返回数据的命令。
PGRES_TUPLES_OK
成功执行一个返回数据的查询（比如 SELECT 或者 SHOW）。
```

PGRES_COPY_OUT
（从服务器）Copy Out （拷贝出）数据传输开始。
PGRES_COPY_IN
Copy In（拷贝入）（到服务器）数据传输开始。
PGRES_BAD_RESPONSE
服务器的响应无法理解。
PGRES_NONFATAL_ERROR
发生了一个非致命错误（通知或者警告）。
PGRES_FATAL_ERROR
发生了一个致命错误。
PGRES_COPY_BOTH
拷贝入/出（到和从服务器）数据传输开始。这个特性当前只用于流复制， 所以这个状态不会在普通应用中发生。
PGRES_SINGLE_TUPLE
PGresult 包含一个来自当前命令的结果元组。 这个状态只在查询选择了单行模式时发生

注意事项

- 恰好检索到零行的 SELECT 命令仍然显示 PGRES_TUPLES_OK 。PGRES_COMMAND_OK 用于永远不能返回行的命令 (插入或更新，不带返回子句等)。PGRES_EMPTY_QUERY 响应可能表明客户端软件存在 bug。
- 状态为 PGRES_NONFATAL_ERROR 的结果永远不会由 PQexec 或其他查询执行函数直接返回，此类结果将传递给通知处理程序。

示例

参见示例。

5.4.3 异步命令处理

PQexec 函数对普通的同步应用里提交命令已经足够使用。但是它却有几个缺陷，而这些缺陷可能对某些用户很重要：

- PQexec 等待命令结束，而应用可能还有其它的工作要做（比如维护用户界面等），此时 PQexec 可不想阻塞在这里等待响应。
- 因为客户端应用在等待结果的时候是处于挂起状态的，所以应用很难判断它是否该尝试结束正在进行的命令。
- PQexec 只能返回一个 PGresult 结构。如果提交的命令字符串包含多个 SQL 命令， 除了最后一个 PGresult 以外都会被 PQexec 丢弃。

- PQexec 总是收集命令的整个结果，将其缓存在一个 PGresult 中。虽然这为应用简化了错误处理逻辑，但是对于包含多行的结果是不切实际的。

不想受到这些限制的应用可以改用下面的函数，这些函数也是构造 PQexec 的函数：PQsendQuery 和 PQgetResult 。PQsendQueryParams，PQsendPrepare， PQsendQueryPrepared 也可以和 PQgetResult 一起使用。

5.4.3.1 PQsendQuery

功能描述

向服务器提交一个命令而不等待结果。如果查询成功发送则返回 1，否则返回 0。

原型

```
int PQsendQuery(PGconn *conn, const char    *command);
```

参数

表 5-22 PQsendQuery 参数

关键字	参数说明
conn	指向包含链接的对象指针。
command	需要执行的查询字符串。

返回值

int：执行结果为 1 表示成功，0 表示失败，失败原因存到 conn->errorMessage 中。

注意事项

在成功调用 PQsendQuery 后，调用 PQgetResult 一次或者多次获取结果。PQgetResult 返回空指针表示命令已执行完成，否则不能再次调用 PQsendQuery（在同一连接上）。

示例

参见示例。

5.4.3.2 PQsendQueryParams

功能描述

向服务器提交与命令分隔的参数，不等待结果。

原型

```
int PQsendQueryParams(PGconn *conn,
                      const char *command, int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

参数

表 5-23 PQsendQueryParams 参数

关键字	参数说明
conn	指向包含链接的对象指针。
command	需要执行的查询字符串。
nParams	参数个数。
paramTypes	参数类型。
paramValues	参数值。
paramLengths	参数长度。
paramFormats	参数格式。
resultFormat	结果的格式。

返回值

int：执行结果为 1 表示成功，0 表示失败，失败原因存到 conn->errorMessage 中。

注意事项

该函数等效于 PQsendQuery，只是查询参数可以和查询字符串分开声明。函数的参数处理和 PQexecParams 一样，和 PQexecParams 类似，它不能在 2.0 版本的协议连接上工作，并且它只允许在查询字符串里出现一条命令。

示例

参见示例。

5.4.3.3 PQsendPrepare

功能描述

发送一个请求，创建一个给定参数的预备语句，而不等待结束。

原型

```
int PQsendPrepare(PGconn *conn,
const char *stmtName,
const char *query,
int nParams,
const Oid *paramTypes);
```

参数

表 5-24 PQsendPrepare 参数

关键字	参数说明
conn	指向包含链接的对象指针。
stmtName	需要执行的 stmt 名称。
query	需要执行的查询字符串。
nParams	参数个数。
paramTypes	声明参数类型的数组。

返回值

int：执行结果为 1 表示成功，0 表示失败，失败原因存到 conn->errorMessage 中。

注意事项

该函数为 PQprepare 的异步版本：如果能够分派请求，则返回 1，否则返回 0。调用成功后，调用 PQgetResult 判断服务端是否成功创建了 preparedStatement。函数的参数与 PQprepare 一样处理。与 PQprepare 一样，它也不能在 2.0 协议的连接上工作。

示例

参见示例。

5.4.3.4 PQsendQueryPrepared

功能描述

发送一个请求执行带有给出参数的预备语句，不等待结果。

原型

```
int PQsendQueryPrepared(PGconn *conn,
const char *stmtName,
int nParams,
const char * const *paramValues,
const int *paramLengths,
const int *paramFormats,
int resultFormat);
```

参数

表 5-25 PQsendQueryPrepared 参数

关键字	参数说明
conn	指向包含链接信息的对象指针。
stmtName	需要执行的 stmt 名称。
nParams	参数个数。
paramValues	参数值。
paramLengths	参数长度。
paramFormats	参数格式。
resultFormat	结果的格式。

返回值

int：执行结果为 1 表示成功，0 表示失败，失败原因存到 conn->error_message 中。

注意事项

该函数类似于 PQsendQueryParams，但是要执行的命令是通过命名一个预先准备的语句来指定的，而不是提供一个查询字符串。该函数的参数与 PQexecPrepared 一样处 理。和 PQexecPrepared 一样，它也不能在 2.0 协议的连接上工作。

示例

参见示例。

5.4.3.5 PQflush

功能描述

尝试将任何排队的输出数据刷新到服务器。

原型

```
int    PQflush(PGconn *conn);
```

参数

表 5-26 PQflush 参数

关键字	参数说明
conn	指向包含链接信息的对象指针。

返回值

int：如果成功（或者如果发送队列为空），则返回 0；如果由于某种原因失败，则返回 -1；如果发送队列中的所有数据都发送失败，则返回 1。(此情况只有在连接为非阻塞时才能发生)，失败原因存到 conn->error_message 中。

注意事项

在非阻塞连接上发送任何命令或数据之后，调用 PQflush。如果返回 1，则等待套接字变为读或写就绪。如果为写就绪状态，则再次调用 PQflush。如果已经读到，调用 PQconsumeInput，然后再次调用 PQflush。重复，直到 PQflush 返回 0。(必须检查读 就绪，并用 PQconsumeInput 排出输入，因为服务器可以阻止试图向我们发送数据， 例如。通知信息，直到我们读完它才会读我们的数据。)一旦 PQflush 返回 0，就等待套接字准备好，然后按照上面描述读取响应。

示例

参见示例。

取消正在处理的查询

客户端应用可以使用本节描述的函数，要求取消一个仍在被服务器处理的命令。

5.4.3.6 PQgetCancel

功能描述

创建一个数据结构，其中包含取消通过特定数据库连接发出的命令所需的信息。

原型

```
PGcancel *PQgetCancel(PGconn *conn);
```

参数

表 5-27 PQgetCancel 参数-

关键字	参数说明
conn	指向包含链接信息的对象指针。

返回值

PGcancel：指向包含 cancel 信息对象的指针。

注意事项

PQgetCancel 创建一个给定 PGconn 连接对象的 PGcancel 对象。如果给定的 conn 是 NULL 或无效连接，它将返回 NULL。PGcancel 对象是一个不透明的结构，应用程序不能直接访问它；它只能传递给 PQcancel 或 PQfreeCancel。

示例

参见示例。

5.4.3.7 PQfreeCancel

功能描述

释放 PQgetCancel 创建的数据结构。

原型

```
void PQfreeCancel(PGcancel *cancel);
```

参数

表 5-28 PQfreeCancel 参数

关键字	参数说明
cancel	指向包含 cancel 信息的对象指针。

注意事项

PQfreeCancel 释放一个由前面的 PQgetCancel 创建的数据对象。

示例

参见示例。

5.4.3.8 PQcancel

功能描述

要求服务器放弃处理当前命令。

原型

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

参数

表 5-29 PQcancel 参数

关键字	参数说明
cancel	指向包含 cancel 信息的对象指针。
errbuf	出错保存错误信息的 buffer。
errbufsize	保存错误信息的 buffer 大小。

返回值

int：执行结果为 1 表示成功，0 表示失败，失败原因存到 errbuf 中。

注意事项

- 成功发送并不保证请求将产生任何效果。如果取消有效，当前命令将提前终止并返回错误结果。如果取消失败（例如，因为服务器已经处理完命令），无返回结果。
- 如果 errbuf 是信号处理程序中的局部变量，则可以安全地从信号处理程序中调用 PQcancel。就 PQcancel 而言，PGcancel 对象是只读的，因此它也可以从一个线程中调用，这个线程与操作 PGconn 对象线程是分离的。

示例

参见示例。

5.4.4 取消正在处理的查询

客户端应用可以使用本节描述的函数，要求取消一个仍在被服务器处理的命令。

5.4.4.1 PQgetCancel

功能描述

创建一个数据结构，其中包含取消通过特定数据库连接发出的命令所需的信息。

原型

PGcancel *PQgetCancel(PGconn *conn);

参数

表 5-30 PQsendQuery 参数

关键字	参数说明
conn	指向包含链接信息的对象指针。

返回值

PGcancel：指向包含 cancel 信息对象的指针。

注意事项

PQgetCancel 创建一个给定 PGconn 连接对象的 PGcancel 对象。如果给定的 conn 是 NULL 或无效连接，它将返回 NULL。PGcancel 对象是一个不透明的结构，应用程序不能直接访问它；它只能传递给 PQcancel 或 PQfreeCancel。

示例

请参见示例章节。

5.4.4.2 PQfreeCancel

功能描述

释放 PQgetCancel 创建的数据结构。

原型

void PQfreeCancel(PGcancel *cancel);

参数

表 5-31 PQ 参数

关键字	参数说明
cancel	指向包含 cancel 信息的对象指针。

注意事项

PQfreeCancel 释放一个由前面的 PQgetCancel 创建的数据对象。

示例

请参见示例章节。

5.4.4.3 PQcancel

功能描述

要求服务器放弃处理当前命令。

原型

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

参数

表 5-32 PQcancel 参数

关键字	参数说明
cancel	指向包含 cancel 信息的对象指针。
errbuf	出错保存错误信息的 buffer。
errbufsize	保存错误信息的 buffer 大小。

返回值

int：执行结果为 1 表示成功，0 表示失败，失败原因存到 errbuf 中。

注意事项

成功发送并不保证请求将产生任何效果。如果取消有效，当前命令将提前终止并返回错误结果。如果取消失败（例如，因为服务器已经处理完命令），无返回结果。

如果 errbuf 是信号处理程序中的局部变量，则可以安全地从信号处理程序中调用

PQcancel。就 PQcancel 而言，PGcancel 对象是只读的，因此它也可以从一个线程中调用，这个线程与操作 PGconn 对象线程是分离的。

示例

请参见示例章节。

5.5 链接参数

表 5-33 链接参数

参数	描述
host	要链接的主机名。如果主机名以斜杠开头，则它声明使用 Unix 域套接字通讯而不是 TCP/IP 通讯；该值就是套接字文件所存储的目录。如果没有声明 host，那么默认是与位于/tmp 目录（或者安装数据库的时候声明的套接字目录）里面的 Unix-域套接字链接。在没有 Unix 域套接字的机器上，默认与 localhost 链接。 接受以‘,’分割的字符串来指定多个主机名，支持指定多个主机名。
hostaddr	与之链接的主机的 IP 地址，是标准的 IPv4 地址格式，比如，172.28.40.9。如果机器支持 IPv6，那么也可以使用 IPv6 的地址。如果声明了一个非空的字符串，那么使用 TCP/IP 通讯机制。 接受以‘,’分割的字符串来指定多个 IP 地址，支持指定多个 IP 地址。 使用 hostaddr 取代 host 可以让应用避免一次主机名查找，这一点对于那些有时间约束的应用来说可能是非常重要的。不过，GSSAPI 或 SSPI 认证方法要求主机名（host）。因此，应用下面的规则： ● 如果声明了不带 hostaddr 的 host 那么就强制进行主机名查找。 ● 如果声明中没有 host，hostaddr 的值给出服务器网络地址；如果认证方法要求主机名，那么链接尝试将失败。 ● 如果同时声明了 host 和 hostaddr，那么 hostaddr 的值作为服务器网络地址。host 的值将被忽略，除非认证方法需要它，在这种情况下它将被用作主机名。 须知 要注意如果 host 不是网络地址 hostaddr 处的服务器名，那么认证很有可能失败。 如果主机名（host）和主机地址都没有，那么 libpq 将使用一个本地的 Unix 域套接字进行链接；或者是在没有 Unix 域套接字的机器上，它将

	尝试与 localhost 链接。
port	主机服务器的端口号，或者在 Unix 域套接字链接时的套接字扩展文件名。 接受以‘,’分割的字符串来指定多个端口号，支持指定多个端口号。
user	要链接的用户名，缺省是与运行该应用的用户操作系统名同名的用户。
dbname	数据库名，缺省和用户名相同。
password	如果服务器要求口令认证，所用的口令。
connect_tim eout	链接的最大等待时间，以秒计（用十进制整数字符串书写），0 或者不声明表示无穷。不建议把链接超时的值设置得小于 2 秒。
client_enco ding	为这个链接设置 client_encoding 配置参数。除了对应的服务器选项接受的值，你可以使用 auto 从客户端中的当前环境中确定正确的编码（Unix 系统上是 LC_CTYPE 环境变量）。
tty	忽略（以前，该参数指定了发送服务器调试输出的位置）。
options	添加命令行选项以在运行时发送到服务器。
application_ name	为 application_name 配置参数指定一个值，表明当前用户身份。
fallback_ap plication_na me	为 application_name 配置参数指定一个后补值。如果通过一个连接参数或 PGAPPNAME 环境变量没有为 application_name 给定一个值， 将使用这个值。在希望设置一个默认应用名但不希望它被用户覆盖的一般工具程序中指定一个后补值很有用。
keepalives	控制客户端侧的 TCP 保持激活是否使用。缺省值是 1，意思为打开， 但是如果不想要保持激活，你可以更改为 0，意思为关闭。通过 Unix 域套接字做的链接忽略这个参数。
keepalives_i dle	在 TCP 应该发送一个保持激活的信息给服务器之后，控制不活动的秒数。0 值表示使用系统缺省。通过 Unix 域套接字做的链接或者如果禁用了保持激活则忽略这个参数。
keepalives_i nterval	在 TCP 保持激活信息没有被应该传播的服务器承认之后，控制秒数。0 值表示使用系统缺省。通过 Unix 域套接字做的链接或者如果禁用了保持激活则忽略这个参数。
keepalives_c ount	添加命令行选项以在运行时发送到服务器。例如，设置为 -c

	comm_debug_mode=off 设置 guc 参数 comm_debug_mode 参数的会话的值为 off。
rw_timeout	设置客户端连接读写超时时间。
sslmode	启用 SSL 加密的方式： ● disable：不使用 SSL 安全连接。 ● allow：如果数据库服务器要求使用，则可以使用 SSL 安全加密连接，但不验证数据库服务器的真实性。 ● prefer：如果数据库支持，那么首选使用 SSL 安全加密连接，但不验证数据库服务器的真实性。 ● require：必须使用 SSL 安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。 ● verify-ca：必须使用 SSL 安全连接，当前 windows odbc 不支持 cert ● 方式认证。 ● verify-full：必须使用 SSL 安全连接，当前 windows odbc 不支持 cert 方式认证。
sslcompress ion	如果设置为 1（默认），SSL 连接之上传送的数据将被压缩（这要求 OpenSSL 版本为 0.9.8 或更高）。如果设置为 0，压缩将被禁用（这要求 OpenSSL 版本为 1.0.0 或更高）。如果建立的是一个没有 SSL 的连接，这个参数会被忽略。如果使用的 OpenSSL 版本不支持该参数，它也会被忽略。压缩会占用 CUP 时间，但是当瓶颈为网络时可以提高吞吐量。如果 CPU 性能是限制因素，禁用压缩能够改进响应时间和吞吐量。
sslcert	这个参数指定客户端 SSL 证书的文件名，它替换默认的~/.postgresql/postgresql.crt。如果没有建立 SSL 连接，这个参数会被忽略。
sslkey	这个参数指定用于客户端证书的密钥位置。它能指定一个会被用来替代默认的~/.postgresql/postgresql.key 的文件名，或者它能够指定一个从外部“引擎”（引擎是 OpenSSL 的可载入模块）得到的密钥。一个外部引擎说明应该由一个冒号分隔的引擎名称以及一个引擎相关的关键标识符组成。如果没有建立 SSL 连接，这个参数会被忽略。
sslrootcert	这个参数指定一个包含 SSL 证书机构（CA）证书的文件名称。如果该文件存在，服务器的证书将被验证是由这些机构之一签发。默认值是~/.postgresql/root.crt。
sslcrl	这个参数指定 SSL 证书撤销列表（CRL）的文件名。列在这个文件中

	的证书如果存在，在尝试认证该服务器证书时会被拒绝。默认值是~/.postgresql/root.crl。
requirepeer	这个参数指定服务器的操作系统用户，例如 requirepeer=postgres。当建立一个 Unix 域套接字连接时，如果设置了这个参数，客户端在连接开始时检查服务器进程是否运行在指定的用户名之下。如果发现不是，该连接会被一个错误中断。这个参数能被用来提供与 TCP/IP 连接上 SSL 证书相似的服务器认证（注意，如果 Unix 域套接字在/tmp 或另一个公共可写的位置，任何用户能启动一个在那里侦听的服务器。使用这个参数来保证你连接的是一个由可信用户运行的服务器）。这个选项只在实现了 peer 认证方法的平台上受支持。
krbsrvname	当用 GSSAPI 认证时，要使用的 Kerberos 服务名。为了让 Kerberos 认证成功，这必须匹配在服务器配置中指定的服务名。
gsslib	用于 GSSAPI 认证的 GSS 库。只用在 Windows 上。设置为 gssapi 可强制 libpq 用 GSSAPI 库来代替默认的 SSPI 进行认证。
service	用于附加参数的服务名。它指定保持附加连接参数的 pg_service.conf 中的一个服务名。这允许应用只指定一个服务名，这样连接参数能被集中维护。
authtype	不再使用“authtype”，因此将其标记为“不显示”。我们将其保留在数组中，以免拒绝旧应用程序中的 conninfo 字符串，这些应用程序可能仍在尝试设置它。
remote_nod ename	指定连接本地节点的远端节点名称。
localhost	指定在一个连接通道中的本地地址。
localport	指定在一个连接通道中的本地端口。
fencedUdfRPCMode	控制 fenced UDF RPC 协议是使用 unix 域套接字或特殊套接字文件名。缺省值是 0，意思为关闭，使用 unix domain socket 模式，文件类型为“.s.PGSQL.%d”，但是要使用 fenced udf，文件类型为.s.fencedMaster_unixdomain，可以更改为 1，意思为开启。
replication	这个选项决定是否该连接应该使用复制协议而不是普通协议。这是 PostgreSQL 的复制连接以及 pg_basebackup 之类的工具在内部使用的协议，但也可以被第三方应用使用。支持下列值，大小写无关： ● true、on、yes、1：连接进入到物理复制模式。 ● database：连接进入到逻辑复制模式，连接到 dbname 参数中指定

	的数据库。 ● false、off、no、0：该连接是一个常规连接，这是默认行为。在物理或者逻辑复制模式中，仅能使用简单查询协议。
backend_ve rsion	传递到远端的后端版本号。
prototype	设置当前协议级别，默认：PROTO_TCP。
enable_ce	控制是否允许客户端连接全密态数据库。默认 0，如果需要开启，则修改为 1。
connection_ info	Connection_info 是一个包含 driver_name、driver_version、driver_path 和 os_user 的 json 字符串。 如果不为 NULL，使用 connection_info 忽略 connectionExtraInf 如果为 NULL，生成与 libpq 相关的连接信息字符串，当 connectionExtraInf 为 false 时 connection_info 只有 driver_name 和 driver_version。
connectionE xtraInf	设置 connection_info 是否存在扩展信息，默认值为 0，如果包含其他信息，则需要设置为 1。
target_sessi on_attrs	设定连接的主机的类型。主机的类型和设定的值一致时才能连接成功。target_session_attrs 的设置规则如下： ● any(默认值)：可以对所有类型的主机进行连接。 ● read-write：当连接的主机允许可读可写时，才进行连接。 ● read-only：仅对可读的主机进行连接。 ● primary：仅对主备系统中的主机能进行连接。 ● standby：仅对主备系统中的备机进行连接。 ● prefer-standby：首先尝试找到一个备机进行连接。如果对 hosts 列表的所有机器都连接失败，那么尝试“any”模式进行连接。

6 基于 Psycopg 开发

Psycopg 是一种用于执行 SQL 语句的 Python API，可以为数据库提供统一访问接口。基于此，应用程序可进行数据操作。Psycopg2 是对 libpq 的封装，主要使用 C 语言实现，既高效又安全。它具有客户端游标和服务器端游标、异步通信和通知、支持 COPY TO/COPY FROM 功能。支持多种类型 Python 开箱即用，适配 PostgreSQL 数据类型；可以通过灵活的对象适配系统进行扩展和定制适配。Psycopg2 兼容 Unicode 和 Python 3。

GBase 8s 数据库支持 Psycopg2 特性，并且支持通过 SSL 模式链接到 Psycopg2。

6.1 Psycopg 包

获取发布包，并解压。解压后有两个文件夹：

- psycopg2：psycopg2 库文件。
- lib：lib 库文件。

图 6-1 采用 Psycopg2 开发应用程序的流程

6.2 加载驱动

- 在使用驱动之前，需要做如下操作：
  - 先解压版本对应驱动包，使用 root 用户将 psycopg2 拷贝到 python 安装目录下的 site-packages 文件夹下。
  - 修改 psycopg2 目录权限为 755。
  - 将 psycopg2 目录添加到环境变量$PYTHONPATH，并使之生效。
  - 对于非数据库用户，需要将解压后的 lib 目录，配置在 LD_LIBRARY_PATH 中。
- 在创建数据库连接之前，需要先加载如下数据库驱动程序：

```
import psycopg2
```

6.3 连接数据库

- 使用 psycopg2.connect 函数获得 connection 对象。
- 使用 connection 对象创建 cursor 对象。

6.4 执行 SQL 语句

- 构造操作语句，使用%s 作为占位符，执行时 psycopg2 会用参数值智能替换掉占位符。可以添加 RETURNING 子句，来得到自动生成的字段值。
- 使用 cursor.execute 方法来操作一行，使用 cursor.executemany 方法来操作多行。

6.5 处理结果集

- cursor.fetchone()：这种方法提取的查询结果集的下一行，返回一个序列，没有数　据可用时则返回空。
- cursor.fetchall()：这个例程获取所有查询结果(剩余)行，返回一个列表。空行时　则返回空列表。

6.6 关闭连接

- 在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。关闭数据库连接可以直接调用其 close 方法，如 connection.close()。

⚠ 注意

此方法关闭数据库连接，并不自动调用 commit()。如果只是关闭数据库连接而不调用 commit()方法，那么所有更改将会丢失。

6.7 连接数据库（SSL 方式）

用户通过 psycopy2 连接 Kernel 服务器时，可以通过开启 SSL 加密客户端和服务器之间的通讯。在使用 SSL 时，默认用户已经获取了服务端和客户端所需要的证书和私钥文件，关于证书等文件的获取请参考 Openssl 相关文档和命令。

- 使用*.ini 文件（python 的 configparser 包可以解析这种类型的配置文件）保存数据库连接的配置信息。
- 在连接选项中添加 SSL 连接相关参数：sslmode，sslcert，sslkey，sslrootcert。
  - sslmode：可选项见表 6-14。
  - sslcert：客户端证书路径。
  - sslkey：客户端密钥路径。
  - sslrootcert：根证书路径。
- 使用 psycopg2.connect 函数获得 connection 对象。
- 使用 connection 对象创建 cursor 对象。

表 6-1 sslmode 的可选项及其描述

sslmode	是否会启用 SSL 加密	描述
disable	否	不适用 SSL 安全连接。
allow	可能	如果数据库服务器要求使用，则可以使用 SSL 安全加密连接，但不验证数据库服务器的真实性。
prefer	可能	如果数据库支持，那么首选使用 SSL 连接， 但不验证数据库服务器的真实性。
require	是	必须使用 SSL 安全连接，但是只做了数据加密，而并不验证数据库服务器的真实性。
verify-ca	是	必须使用 SSL 安全连接。

verify-full	是	必须使用 SSL 安全连接，目前暂不支持。

6.8 示例：常用操作

```
import psycopg2
#创建连接对象
conn=psycopg2.connect(database="postgres",user="user",password="password",host="localhost",port=port) cur=conn.cursor() #创建指针对象
#创建连接对象（SSl 连接）
conn = psycopg2.connect(dbname="postgres", user="user", password="password",
host="localhost", port=port,
sslmode="verify-ca", sslcert="client.crt",sslkey="client.key",sslrootcert="cacert.pem")
注意： 如果 sslcert, sslkey,sslrootcert 没有填写，默认取当前用户.postgresql 目录下对应的 client.crt， client.key， root.crt
# 创建表
cur.execute("CREATE TABLE student(id integer,name varchar,sex varchar);")
#插入数据
cur.execute("INSERT INTO student(id,name,sex) VALUES(%s,%s,%s)",(1,'Aspirin','M'))
cur.execute("INSERT INTO student(id,name,sex) VALUES(%s,%s,%s)",(2,'Taxol','F'))
cur.execute("INSERT INTO student(id,name,sex) VALUES(%s,%s,%s)",(3,'Dixheral','M'))
# 获取结果
cur.execute('SELECT * FROM student') results=cur.fetchall()
print (results)
# 关闭连接 conn.commit() cur.close() conn.close()
psycopg2 常用链接方式
conn = psycopg2.connect(dbname="postgres", user="user", password="password",
host="localhost", port=port)
conn = psycopg2.connect("dbname=postgres user=user password=password host=localhost port=port")
使用日志 import logging import psycopg2
from psycopg2.extras import LoggingConnection
logging.basicConfig(level=logging.DEBUG) # 日志级别
logger = logging.getLogger( name )
db_settings = { "user": "user",
"password": "password", "host": "localhost", "database": "postgres", "port": port
}
conn = psycopg2.connect(connection_factory=LoggingConnection, **db_settings)
conn.initialize(logger)
```

6.9 Psycopg 接口参考

Psycopg 接口是一套提供给用户的 API 方法，本节将对部分常用接口做具体描述。

6.9.1 psycopg2.connect()

功能描述

此方法创建新的数据库会话并返回新的 connection 对象。

原型

```
conn=psycopg2.connect(dbname="test",user="postgres",password="secret",host="127.0.0.1",port="5432")
```

参数

表 8-62 psycopg2.connect 参数

关键字	参数说明
dbname	数据库名称
user	用户名
password	密码
host	数据库 IP 地址，默认为 UNIX socket 类型。
port	连接端口号，默认为 5432。
sslmode	ssl 模式，ssl 连接时用。
sslcert	客户端证书路径，ssl 连接时用。
sslkey	客户端密钥路径，ssl 连接时用。
sslrootcert	根证书路径，ssl 连接时用。

返回值

connection 对象（连接 GBase 8s 数据库实例的对象）。

示例

参见示例：常用操作。

6.9.2 connection.cursor()

功能描述

此方法用于返回新的 cursor 对象。

原型

```
cursor(name=None,  cursor_factory=None,  scrollable=None, withhold=False)
```

参数

表 8-63 connection.cursor 参数

关键字	参数说明
name	cursor 名称，默认为 None。
cursor_factory	用于创造非标准 cursor，默认为 None。
scrollable	设置 SCROLL 选项，默认为 None。
withhold	设置 HOLD 选项，默认为 False。

返回值

cursor 对象（用于整个数据库使用 Python 编程的 cursor）。

示例

参见示例：常用操作。

6.9.3 cursor.execute(query,vars_list)

功能描述

此方法执行被参数化的 SQL 语句（即占位符，而不是 SQL 文字）。psycopg2 模块支持用%s 标志的占位符。

原型

```
curosr.execute(query,vars_list)
```

参数

表 8-64 curosr.execute 参数

关键字	参数说明

query	待执行的 SQL 语句。
vars_list	变量列表，匹配 query 中%s 占位符。

返回值

无

示例

参见示例：常用操作。

6.9.4 curosr.executemany(query,vars_list)

功能描述

此方法执行 SQL 命令所有参数序列或序列中的 SQL 映射。

原型

```
curosr.executemany(query,vars_list)
```

参数

表 8-65 curosr.executemany 参数

关键字	参数说明
query	待执行的 SQL 语句。
vars_list	变量列表，匹配 query 中%s 占位符。

返回值

无

示例

参见示例：常用操作。

6.9.5 connection.commit()

功能描述

此方法将当前挂起的事务提交到数据库。

注意

默认情况下，Psycopg 在执行第一个命令之前打开一个事务：如果不调用 commit()，任何数据操作的效果都将丢失。

原型

```
connection.commit()
```

示例

参见示例：常用操作。

6.9.6 connection.rollback()

功能描述

此方法将当前挂起事务回滚。

⚠ 注意

执行关闭连接“close()”而不先提交更改“commit()”将导致执行隐式回滚。

原型

```
connection.rollback()
```

示例

参见示例：常用操作。

6.9.7 cursor.fetchone()

功能描述

此方法提取查询结果集的下一行，并返回一个元组。

原型

```
cursor.fetchone()
```

返回值

单个元组，为结果集的第一条结果，当没有更多数据可用时，返回为“None”。

示例

参见示例：常用操作。

6.9.8 cursor.fetchall()

功能描述

此方法获取查询结果的所有（剩余）行，并将它们作为元组列表返回。

原型

```
cursor.fetchall()
```

返回值

元组列表，为结果集的所有结果。空行时则返回空列表。

示例

参见示例：常用操作。

6.9.9 cursor.close()

功能描述

此方法关闭当前连接的游标。

原型

```
cursor.close()
```

示例

参见示例：常用操作。

6.9.10 connection.close()

功能描述

此方法关闭数据库连接。

注意

此方法关闭数据库连接，并不自动调用 commit()。如果只是关闭数据库连接而不调用 commit()方法，那么所有更改将会丢失。

原型

```
connection.close()
```

示例

参见示例：常用操作。

7 编译与调试

用户可以根据自己的需要，通过修改实例数据目录下的 postgresql.conf 文件中特定的配置参数来控制日志的输出，从而更好地了解数据库的运行状态。可调整的配置参数请参见表 7-1。

表 7-1 配置参数

参数名称	描述	取值范围	备注
client_min_messages	配置发送到客户端信息的级别。	DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 LOG NOTICE WARNING ERROR FATAL PANIC 默认值：NOTICE 。	设置级别后，发送到客户端的信息包含所设级别及以下所有低级别会发送的信息。级别越低，发送的信息越少。
log_min_messages	配置写到服务器日志里信息的级别。	DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR	指定某一级别后，写到日志的信息包含所有更高级别会输出的信息。级别越高，服务器日志的信息越少。

参数名称	描述	取值范围	备注
		LOG FATAL PANIC 默认值：WARNING。	
log_min_error_statement	配置写到服务器日志中错误 SQL 语句的级别。	DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1 INFO NOTICE WARNING ERROR FATAL PANIC 缺省值：ERROR。	所有导致一个特定级别（或者更高级别）错误的 SQL 语句都将记录在服务器日志中。 只有系统管理员可以修改该参数。
log_min_duration_statement	配置语句执行持续的最短时间。如果某个语句的持续时间大于或者等于设置的毫秒数，则会在日志中记录该语句及其持续时间。打开这个选项可以方便地跟踪需要优化的查询。	INT 类型。 默认值：30min。 单位：毫秒。	设置为-1 表示关闭这个功能。 只有系统管理员可以修改该参数。
log_connections/log_discon	配置是否在每次会话连接或结束时向服务器日志里打印	on：每次会话连接或结束时向日志里	-

参数名称	描述	取值范围	备注
nections	一条信息。	打印一条信息。 off：每次会话连接或结束时不向日志里打印信息。 默认值：off。	
log_duration	配置是否记录每个已完成语句的持续时间。	on：记录每个已完成语句的持续时间。 off：不记录已完成语句的持续时间。 默认值：off 。	只有系统管理员可以修改该参数。
log_statement	配置日志中记录哪些 SQL 语句。	none：不记录任何 SQL 语句。 ddl：记录数据定义语句。 mod:记录数据定义语句和数据操作语句。 all ：记录所有语句。 默认值： none。	只有系统管理员可以修改该参数。
log_hostname	配置是否记录主机名。	on：记录主机名。 off:不记录主机名。 默认值：off。	缺省时，连接日志只记录所连接主机的 IP 地址。打开这个选项会同时记录主机名。 该参数同时影响查看审计结果、GS_WLM_SESSION_HISTORY 、 PG_STAT_ACTIVITY 和 log_line_prefix 参数。

上表有关参数级别的说明请参见表 7-2。

表 7-2 日志级别参数说明

级别	说明
DEBUG[1-5]	提供开发人员使用的信息。5 级为最高级别，依次类推，1 级为最低级别。
INFO	提供用户隐含要求的信息。如在 VACUUM VERBOSE 过程中的信息。
NOTICE	提供可能对用户有用的信息。如长标识符的截断，作为主键一部分创建的索引。
WARNING	提供给用户的警告。如在事务块范围之外的 COMMIT。
ERROR	报告导致当前命令退出的错误。
LOG	报告一些管理员感兴趣的信息。如检查点活跃性。
FATAL	报告导致当前会话终止的原因。
PANIC	报告导致所有会话退出的原因。

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