GBASE

GBase 8a 程序员手册 C API 篇

www.gbase.cn

GBase 8a 程序员手册 CAPI 篇，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

天津华苑产业区海泰发展六道 6 号海泰绿色产业基地 J 座(300384)

电话：400-013-9696　　邮箱：info@gbase.cn

目 录

前言

手册简介

GBase 8a 程序员手册从程序员进行数据库开发的角度对 GBase 8a 进行详细介绍。

本手册介绍供客户端连接 GBase 8a 服务器用的 GBase 8a C API 接口驱动程序。本部分内容通过大量示例为用户演示如何使用 GBase 8a C API 函数及使用 GBase C API 进行编程的示例。

公约

下面的文本约定用于本文档：

约 定	说 明
加粗字体	表示文档标题
大写英文（SELECT）	表示 GBase 8a 关键字
等宽字体	表示代码示例
…	表示被省略的内容。

1 GBase C API 版本

GBase C API 版本	说　明
8.3.81.53	增加集群 IP 路由，负载均衡功能
8.3.81.51	稳定版

2 安装文件

我们提供的 C API 接口的 bin 文件格式如下:

GBaseCAPI-<product version>-<build version>-<os version and archtecture>.bin。

例如: GBaseCAPI-8.3.81.53-build53.8-redhat6-x86_64.bin。

3 GBase C API

3.1数据类型

3.1.1GBASE

◆ 结构说明

该结构代表与数据库连接的句柄。不建议对 GBase 结构进行拷贝，不保证这样的拷贝会有用。

3.1.2GBASE_RES

◆ 结构说明

该结构用来保存 SELECT、SHOW、DESCRIBE、EXPLAIN 查询返回的结果集。

3.1.3GBASE_ROW

◆ 结构说明

该结构用来保存 1 行数据。它是按照计数字节字符串的数组实施的。(如果字段值可能包含二进制数据，不能将其当作由 NULL 终结的字符串对待，这是因为这类值可能会包含 NULL 字节)。它是通过 gbase_fetch_row()获取的。

3.1.4GBASE_FIELD

◆ 结构说明

该结构用来保存字段的信息（字段名、类型、大小)。通过重复调用

gbase_fetch_field()可以为每个字段获取 GBASE_FIELD 结构。

◆ 结构成员

名 称	类 型	描 述
name	char*	字段名称，由 NULL 终结的字符串。如果用 AS 子句为该字段指定了别名，名称的值也是别名。
org_name	char*	字段名称，由 NULL 终结的字符串。忽略别名。
table	char*	包含该字段的表的名称，如果该字段不是计算出的字段的话。对于计算出的字段，表值为空的字符串。如果用 AS 子句为该表指定了别名，表的值也是别名。
org_table	char*	表的名称，由 NULL 终结的字符串。忽略别名。
db	char*	字段源自的数据的名称，由 NULL 终结的字符串。如果该字段是计算出的字段，db 为空的字符串。
catalog	char*	catalog 名称。该值总是"def"。
def	char*	该字段的默认值，由 NULL 终结的字符串。仅当使用 GBASE_list_fields()时才设置它。
length	unsigned long	字段的宽度，如表定义中所指定的那样。
max_length	unsigned long	用于结果集的字段的最大宽度（对于实际位于结果集中的行，最长字段值的长度）。如果使用 gbase_store_result()或 gbase_list_fields()，它将包含字段的最大长度。如果使用 gbase_use_result()，该变量的值为0。
name_length	unsigned int	名称的长度。

名　称	类　型	描　述
org_name_length	unsigned int	org_name 的长度。
table_length	unsigned int	表的长度。
org_table_length	unsigned int	org_table 的长度。
db_length	unsigned int	db 的长度。
catalog_length	unsigned int	catalog 的长度。
def_length	unsigned int	def 的长度。
flags	unsigned int	用于字段的不同“位标志”。
decimals	unsigned int	用于数值字段的十进制数数目。
charsetnr	unsigned int	用于字段的字符集编号。
type	enum_field_types	字段的类型。类型值可以是下标所列的 GBASE_TYPE_符号之一：

- flags 字段值集合

标志值	标志描述
NOT_NULL_FLAG	字段不能为 NULL
PRI_KEY_FLAG	字段是主键的组成部分
UNIQUE_KEY_FLAG	字段是唯一键的组成部分
MULTIPLE_KEY_FLAG	字段是非唯一键的组成部分
UNSIGNED_FLAG	字段具有 UNSIGNED 属性
ZEROFILL_FLAG	字段具有 ZEROFILL 属性
BINARY_FLAG	字段具有 BINARY 属性
AUTO_INCREMENT_FLAG	字段具有 AUTO_INCREMENT 属性
ENUM_FLAG	字段是 ENUM（不再重视）
SET_FLAG	字段是 SET（不再重视）
BLOB_FLAG	字段是 BLOB 或 TEXT（不再重视）
TIMESTAMP_FLAG	字段是 TIMESTAMP（不再重视）

标志值的典型用法：

```
if (field->flags & NOT_NULL_FLAG)

printf("Field can't be null\n");
```

可以使用下述方面的宏来定义标志值的布尔状态：

标志状态	描　述
IS_NOT_NULL(flags)	如果该字段定义为 NOT NULL，为“真”。
IS_PRI_KEY(flags)	如果该字段是主键，为“真”。
IS_BLOB(flags)	如果该字段是 BLOB 或 TEXT，为“真”（不再重视，用测试 field->type 取而代之）。

- type 字段值集合

类型值	类型描述
GBASE_TYPE_TINY	TINYINT 字段
GBASE_TYPE_SHORT	SMALLINT 字段
GBASE_TYPE_LONG	INTEGER 字段
GBASE_TYPE_INT24	MEDIUMINT 字段
GBASE_TYPE_LONGLONG	BIGINT 字段
GBASE_TYPE_DECIMAL	DECIMAL 或 NUMERIC 字段
GBASE_TYPE_NEWDECIMAL	精度数 DECIMAL 或 NUMERIC
GBASE_TYPE_FLOAT	FLOAT 字段
GBASE_TYPE_DOUBLE	DOUBLE 或 REAL 字段
GBASE_TYPE_BIT	BIT 字段
GBASE_TYPE_TIMESTAMP	TIMESTAMP 字段
GBASE_TYPE_DATE	DATE 字段
GBASE_TYPE_TIME	TIME 字段
GBASE_TYPE_DATETIME	DATETIME 字段
GBASE_TYPE_YEAR	YEAR 字段
GBASE_TYPE_STRING	CHAR 字段
GBASE_TYPE_VAR_STRING	VARCHAR 字段
GBASE_TYPE_BLOB	BLOB 或 TEXT 字段（使用 max_length 来确定最大长度）
GBASE_TYPE_SET	SET 字段

类型值	类型描述
GBASE_TYPE_ENUM	ENUM 字段
GBASE_TYPE_GEOMETRY	Spatial 字段
GBASE_TYPE_NULL	NULL-type 字段
GBASE_TYPE_CHAR	不再重视，用 GBASE_TYPE_TINY 取代

3.1.5GBASE_FIELD_OFFSET

- 结构说明

这是 GBASE 字段列表偏移量的“类型安全”表示（由 gbase_field_seek()使用）。偏移量是行内的字段编号，从 0 开始。

3.1.6gs_ulonglong

- 结构说明

用于行数以及 gbase_affected_rows()、gbase_num_rows()的类型。该类型提供的范围为 0-1.84e19。

在某些系统上，不能打印类型 gs_ulonglong 的值。要想打印这类值，请将其转换为无符号长整数类型并使用%lu 打印格式，如：

```
printf ("Number of rows: %lu\n", (unsigned long)
gbase_num_rows(result))
```

3.2函数描述

3.2.1gbase_affected_rows

- 摘要：

返回上次 UPDATE 更改的行数，上次 DELETE 删除的行数，或上次 INSERT 语

句插入的行数。对于 UPDATE、DELETE 或 INSERT 语句，可在 gbase_query()后立刻调用。对于 SELECT 语句，gbase_affected_rows()的工作方式与 gbase_num_rows()类似。

◆ 语法：

```
gs_ulonglong gbase_affected_rows(GBASE *gbase);
```

◆ 参数：

gbase　数据库句柄

◆ 返回值：

大于 0 的整数表明受影响或检索的行数。“0”表示 UPDATE 语句未更新记录，在查询中没有与 WHERE 匹配的行，或未执行查询。“-1”表示查询返回错误，或者对于 SELECT 查询，在调用 gbase_store_result()之前调用了 gbase_affected_rows()。由于 gbase_affected_rows()返回无符号值，通过比较返回值和“(gs_ulonglong)-1”或等效的“(gs_ulonglong)~0”，检查是否为“-1”。.

3.2.2 gbase_autocommit

◆ 摘要：

如果模式为“1”，启用 autocommit 模式；如果模式为“0”，禁止 autocommit 模式。

◆ 语法：

```
gs_bool gbase_autocommit(GBASE * gbase, gs_bool auto_mode);
```

◆ 参数：

gbase 数据库句柄

auto_mode 值为 0 或 1，用来启用或禁止 autocommit 模式

◆ 返回值：

如果成功，返回 0，如果出现错误，返回非 0 值。

3.2.3 gbase_change_user

◆ 摘要:

更改用户，将参数 db 的值作为 gbase 连接句柄的当前默认数据库。在后续查询中，对于不包含显式数据库区分符的表引用，该数据库是默认数据库。

如果不能确定已连接的用户或用户不具有使用数据库的权限，gbase_change_user()将失败。在这种情况下，不会改变用户和数据库。

如果不打算拥有默认数据库，可将 db 参数设置为 NULL。

该命令总是会执行活动事务的 ROLLBACK 操作，关闭所有的临时表，解锁所有的锁定表，并复位状态，就像进行了新连接那样。即使未更改用户，也会出现该情况。

◆ 语法:

```
gs_bool gbase_change_user(GBASE *gbase,
const char *user,
const char *passwd,
const char *db);
```

◆ 参数:

◆ 返回值:

0 表示成功，非 0 值表示出现错误。

◆ 错误:

CR_COMMANDS_OUT_OF_SYNC 以不恰当的顺序执行了命令。

CR_SERVER_GONE_ERROR GBase 服务器不可用。

CR_SERVER_LOST 在查询过程中丢失了与服务器的连接。

CR_UNKNOWN_ERROR	出现未知错误。
ER_UNKNOWN_COM_ERROR	GBase 服务器未实施该命令
ER_ACCESS_DENIED_ERROR	用户或密码错误。
ER_BAD_DB_ERROR	数据库不存在。
ER_DBACCESS_DENIED_ERROR	用户没有访问数据库的权限。
ER_WRONG_DB_NAME	数据库名称过长。

3.2.4 gbase_character_set_name

- 摘要

为当前连接返回默认的字符集。

- 语法

```
const char * gbase_character_set_name(GBASE *gbase);
```

- 返回值
- 默认字符集。

3.2.5 gbase_close

- 摘要:

关闭前面打开的连接。如果句柄是由 gbase_init()或 gbase_connect()自动分配的，gbase_close()还将解除分配由 gbase 指向的连接句柄。

- 语法:

```
void gbase_close(GBASE *sock);
```

- 参数:
- 返回值:

无

3.2.6 gbase_commit

◆ 摘要:

提交当前事务。

◆ 语法:

```
gs_bool gbase_commit(GBASE * gbase);
```

◆ 参数:

◆ 返回值:

如果成功，返回 0，如果出现错误，返回非 0 值。

3.2.7 gbase_data_seek

◆ 摘要:

在查询结果集中寻找任意行。偏移值为行号，范围从 0 到 gbase_num_rows(result)-1。

该函数要求结果集结构包含查询的所有结果，因此，gbase_data_seek()仅应与 gbase_store_result()联合使用，而不是与 gbase_use_result()。

◆ 语法:

```
void gbase_data_seek(GBASE_RES *result,gs_ulonglong offset);
```

◆ 参数:

◆ 返回值:

无

3.2.8 gbase_errno

◆ 摘要:

返回最近调用的 API 函数的错误代码。errmsg.h 头文件中，列出了客户端错误消息编号。

◆ 语法:

```
unsigned int gbase_errno(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

如果失败，返回上次 API 函数调用的错误代码。“0”表示未出现错误。

3.2.9 gbase_error

◆ 摘要:

对于失败的最近调用的 API 函数，gbase_error()返回包含错误消息的、由 NULL 终结的字符串。

◆ 语法:

```
const char * gbase_error(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

返回描述错误的、由 NULL 终结的字符串。如果未出现错误，返回空字符串。

3.2.10 gbase_fetch_field

◆ 摘要:

返回采用 GBASE_FIELD 结构的结果集的列。重复调用该函数，以检索关于结果集中所有列的信息。未剩余字段时，gbase_fetch_field()返回 NULL。每次执行新的 SELECT 查询时，将复位 gbase_fetch_field()，以返回关于第 1 个字段的信息。调用 gbase_field_seek()也会影响 gbase_fetch_field()返回的字段。

- 语法：

GBASE_FIELD * gbase_fetch_field(GBASE_RES *result);

- 参数：
- 返回值：

当前列的 GBASE_FIELD 结构。如果未剩余任何列，返回 NULL。

3.2.11 gbase_fetch_field_direct

- 摘要：

给定结果集内某 1 列的字段编号，以 GBASE_FIELD 结构形式返回列的字段定义。可以使用该函数检索任意列的定义。字段编号的值应在从 0 到 gbase_num_fields(result)-1 的范围内。

- 语法：

GBASE_FIELD *gbase_fetch_field_direct(GBASE_RES *res, unsigned int fieldnr);

- 参数：
- 返回值：

对于指定列，返回 GBASE_FIELD 结构。

3.2.12 gbase_fetch_fields

◆ 摘要:

对于结果集，返回所有 GBASE_FIELD 结构的数组。每个结构提供了结果集中 1 列的字段定义。

◆ 语法:

```
GBASE_FIELD * gbase_fetch_fields(GBASE_RES *res);
```

◆ 参数:

◆ 返回值:

对于结果集，返回所有 GBASE_FIELD 结构的数组。每个结构提供了结果集中 1 列的字段定义。

3.2.13 gbase_fetch_lengths

◆ 摘要:

返回结果集内当前行的列的长度。如果打算复制字段值，该长度信息有助于优化，这是因为，你能避免调用 strlen()。此外，如果结果集包含二进制数据，必须使用该函数来确定数据的大小，原因在于，对于包含 NULL 字符的任何字段，strlen()将返回错误的结果。

◆ 语法:

```
unsigned long * gbase_fetch_lengths(GBASE_RES *result);
```

◆ 参数:

◆ 返回值:

无符号长整数的数组表示各列的大小（不包括任何终结 NULL 字符）。如果出现错误，返回 NULL。

3.2.14 gbase_fetch_row

◆ 摘要:

检索结果集的下一行。在 gbase_store_result()之后使用时，如果没有要检索的行，gbase_fetch_row()返回 NULL。在 gbase_use_result()之后使用时，如果没有要检索的行或出现了错误，gbase_fetch_row()返回 NULL。

行内值的数目由 gbase_num_fields(result)给出。如果行中保存了调用 gbase_fetch_row()返回的值，将按照 row[0]到 row[gbase_num_fields(result)-1],访问这些值的指针。行中的 NULL 值由 NULL 指针指明。

可以通过调用 gbase_fetch_lengths()来获得行中字段值的长度。对于空字段以及包含 NULL 的字段，长度为 0。通过检查字段值的指针，能够区分它们。如果指针为 NULL，字段为 NULL，否则字段为空。

◆ 语法:

```
GBASE_ROW  gbase_fetch_row(GBASE_RES *result);
```

◆ 参数:

◆ 返回值:

下一行的 GBASE_ROW 结构。如果没有更多要检索的行或出现了错误，返回 NULL。

3.2.15 gbase_field_seek

◆ 摘要:

将字段光标设置到给定的偏移处。对 gbase_fetch_field()的下一次调用将检索与该偏移相关的列定义。

要想查找行的开始，请传递值为 0 的偏移量。

◆ 语法：

GBASE_FIELD_OFFSET gbase_field_seek(GBASE_RES *result, GBASE_FIELD_OFFSET offset);

◆ 参数：

◆ 返回值：

字段光标的前一个值。

3.2.16 gbase_field_count

◆ 摘要：

返回作用在连接上的最近查询的列数。该函数的正常使用是在 gbase_store_result()返回 NULL（因而没有结果集指针）时。在这种情况下，可调用 gbase_field_count()来判定 gbase_store_result()是否应生成非空结果。这样，客户端就能采取恰当的动作，而无需知道查询是否是 SELECT（或类似 SELECT 的）语句。

◆ 语法：

unsigned int gbase_field_count(GBASE *gbase);

◆ 参数：

◆ 返回值：

表示结果集中列数的无符号整数。

3.2.17 gbase_field_tell

◆ 摘要：

返回上一个 gbase_fetch_field()所使用的字段光标的定义。该值可用作 gbase_field_seek()的参量。

◆ 语法：

```
GBASE_FIELD_OFFSET  gbase_field_tell(GBASE_RES *res);
```

◆ 参数：

◆ 返回值：

字段光标的当前偏移量。

3.2.18 gbase_free_result

◆ 摘要：

释放由 gbase_store_result()、gbase_use_result()、gbase_list_dbs()等为结果集分配的内存。完成对结果集的操作后，必须调用 gbase_free_result()释放结果集使用的内存。

释放完成后，不要尝试访问结果集。

◆ 语法：

```
void    gbase_free_result(GBASE_RES *result);
```

◆ 参数：

◆ 返回值：

无

3.2.19 gbase_get_character_set_info

◆ 摘要：

该函数提供了关于默认客户端字符集的信息。可以使用 gbase_set_character_set()函数更改默认的字符集。

◆ 语法：

void gbase_get_character_set_info(GBASE *gbase, GS_CHARSET_INFO *charset);

- 参数:
- 返回值:

无

3.2.20 gbase_get_client_info

- 摘要:

返回表示客户端库版本的字符串。

- 语法:

```
const char *    gbase_get_client_info(void);
```

- 参数:
- 返回值:

表示 GBASE 客户端库版本的字符串。

3.2.21 gbase_get_client_version

- 摘要:

返回表示客户端库版本的整数。该值的格式是 XYYZZ，其中 X 是主版本号，YY 是发布级别，ZZ 是发布级别内的版本号。

- 语法:

```
unsigned long   gbase_get_client_version(void);
```

- 参数:
- 返回值:

表示 GBase 客户端库版本的整数。

3.2.22 gbase_get_host_info

◆ 摘要:

返回描述了所使用连接类型的字符串，包括服务器主机名。

◆ 语法:

```
const char *gbase_get_host_info(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

代表服务器主机名和连接类型的字符串。

3.2.23 gbase_get_server_version

◆ 摘要:

以整数形式返回服务器的版本号。

◆ 语法:

```
unsigned long    gbase_get_server_version(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

以数据形式返回服务器版本

3.2.24 gbase_hex_string

◆ 摘要:

该函数用于创建可用在 SQL 语句中的合法 SQL 字符串。该字符串从形式上

编码为十六进制格式，每个字符编码为 2 个十六进制数。结果被置入其中，并添加 1 个终结 NULL 字节。

- 语法：

```
unsigned long  gbase_hex_string(char *to,const char *from, unsigned long from_length);
```

- 参数：
- 返回值：

置于“to”中的值的长度，不包括终结用 NULL 字符

3.2.25 gbase_get_proto_info

- 摘要：

返回当前连接所使用的协议版本。

- 语法：

```
unsigned int     gbase_get_proto_info(GBASE *gbase);
```

- 参数：
- 返回值：

代表当前连接所使用协议版本的无符号整数。

3.2.26 gbase_get_server_info

- 摘要：

返回代表服务器版本号的字符串。

- 语法：

```
const char *    gbase_get_server_info(GBASE *gbase);
```

- 参数:
- 返回值:

代表服务器版本号的字符串。

3.2.27 gbase_info

- 摘要:

检索字符串，该字符串提供了关于最近执行查询的信息。

- 语法:

```
const char * gbase_info(GBASE *gbase);
```

- 参数:
- 返回值:

字符串，它表示最近所执行查询的额外信息。如果该查询无可用信息，返回 NULL。

3.2.28 gbase_init

- 摘要:

分配或初始化与 gbase_real_connect()相适应的 GBASE 对象。如果 gbase 是 NULL 指针，该函数将分配、初始化、并返回新对象。否则，将初始化对象，并返回对象的地址。如果 gbase_init()分配了新的对象，当调用 gbase_close()来关闭连接时。将释放该对象。

- 语法:

```
GBASE * gbase_init(GBASE *gbase);
```

- 参数:
- 返回值:

初始化的 GBASE*句柄。如果无足够内存以分配新的对象，返回 NULL。

3.2.29 gbase_insert_id

◆ 摘要:

返回由以前的 INSERT 或 UPDATE 语句为 AUTO_INCREMENT 列生成的值。在包含 AUTO_INCREMENT 字段的表中执行了 INSERT 语句后，应使用该函数。

如果需要保存值，在生成值的语句后，务必立刻调用 gbase_insert_id()。

◆ 语法:

```
gs_ulonglong  gbase_insert_id(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

如果前面的语句未使用 AUTO_INCREMENT，gbase_insert_id()返回 0。

3.2.30 gbase_kill

◆ 摘要:

请求服务器杀死由 pid 指定的线程。

◆ 语法:

```
int     gbase_kill(GBASE *gbase,unsigned long pid);
```

◆ 参数:

◆ 返回值:

0 表示成功，非 0 值表示出现错误。

◆ 错误:

CR_COMMANDS_OUT_OF_SYNC　　以不恰当的顺序执行了命令。

CR_SERVER_GONE_ERROR	GBASE 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.31 gbase_list_dbs

- 摘要:

返回由服务器上的数据库名称组成的结果集，该服务器与由通配符参数指定的简单正则表达式匹配。通配符参数可以包含通配符“%”或“_”，也可以是 NULL 指针，以便与所有的数据库匹配。调用 gbase_list_dbs()的方法类似于执行查询 show database [LIKE wild]。

必须用 gbase_free_result()释放结果集。

- 语法:

```
GBASE_RES * gbase_list_dbs(GBASE *gbase,const char *wild);
```

- 参数:

- 返回值:

成功后返回 GBASE_RES 结果集。如果出现错误，返回 NULL。

3.2.32 gbase_list_fields

- 摘要:

返回由给定表中的字段名称组成的结果集，给定表与由通配符参数指定的简单正则表达式匹配。通配符参数可以包含通配符“%”或“_”，也可以是 NULL 指针，以便与所有的字段匹配。调用 gbase_list_fields()的方法类似于执行查询 SHOW COLUMNS FROM tbl_name [LIKE wild]。

必须用 gbase_free_result()释放结果集。

注意，建议使用 SHOW COLUMNS FROM tbl_name，而不是 gbase_list_fields()。

◆ 语法：

GBASE_RES * gbase_list_fields(GBASE *gbase, const char *table,const char *wild);

◆ 参数：

◆ 返回值：

如果成功，返回 GBASE_RES 结果集。如果出现错误，返回 NULL。

错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBASE 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.33 gbase_list_processes

◆ 摘要：

返回描述当前服务器线程的结果集。

必须用 gbase_free_result()释放结果集。

◆ 语法：

GBASE_RES * gbase_list_processes(GBASE *gbase);

◆ 参数：

◆ 返回值：

如果成功，返回 GBASE_RES 结果集。如果出现错误，返回 NULL。

3.2.34 gbase_list_tables

◆ 摘要:

返回由当前数据库内的表名组成的结果集，当前数据库与由通配符参数指定的简单正则表达式匹配。通配符参数可以包含通配符“%”或“_”，也可以是 NULL 指针，以便与所有的表匹配。调用 gbase_list_tables()的方法类似于执行查询 SHOW tables [LIKE wild]。

必须用 gbase_free_result()释放结果集。

◆ 语法:

```
GBASE_RES * gbase_list_tables(GBASE *gbase,const char *wild);
```

◆ 参数:

◆ 返回值:

如果成功，返回 GBASE_RES 结果集。如果出现错误，返回 NULL。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBASE 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.35 gbase_more_results

◆ 摘要:

如果当前执行的查询存在多个结果，返回“真”，而且应用程序必须调用 gbase_next_result()来获取结果。

◆ 语法:

gs_bool gbase_more_results(GBASE *gbase);

◆ 参数:

◆ 返回值:

如果存在多个结果，返回“真”(1)，如果不存在多个结果，返回“假”(0)。

3.2.36 gbase_next_result

◆ 摘要:

如果存在多个查询结果，gbase_next_result()将读取下一个查询结果，并将状态返回给应用程序。

如果前面的查询返回了结果集，必须为其调用 gbase_free_result()。

调用了 gbase_next_result()后，连接状态就像你已为下一查询调用了 gbase_real_query()或 gbase_query()时的一样。这意味着你能调用 gbase_store_result()、gbase_warning_count()、gbase_affected_rows()等等。

如果 gbase_next_result()返回错误，将不执行任何其他语句，也不会获取任何更多的结果，

◆ 语法:

int gbase_next_result(GBASE *gbase);

◆ 参数:

◆ 返回值:

0 成功并有多个结果。

-1 成功但没有多个结果。

>0 出错。

- 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBASE 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.37 gbase_num_fields

- 摘要:

返回结果集中的列数。

- 语法:

```
unsigned int gbase_num_fields(GBASE_RES *res);
```

- 参数:
- 返回值:

表示结果集中列数的无符号整数。

3.2.38 gbase_num_rows

- 摘要:

返回结果集中的行数。

- 语法:

```
gs_ulonglong  gbase_num_rows(GBASE_RES *res);
```

- 参数:
- 返回值:

结果集中的行数。

3.2.39 gbase_options

◆ 摘要:

可用于设置额外的连接选项，并影响连接的行为。可多次调用该函数来设置数个选项。

应在 gbase_init()之后、以及 gbase_connect()或 gbase_real_connect()之前调用 gbase_options()。

◆ 语法:

```
int gbase_options(GBASE *gbase, enum gbase_option  option,
const void  *arg);
```

◆ 参数:

gbase　数据库句柄。

option　将要设置的选项，option 值的集合如下表。

arg option 设置的选项的值。

option 选项	参量类型	功　能
GBASE_OPT_USE_SERVER_BALANCE	gs_bool*	开启 server 端负载均衡。
GBASE_OPT_KEEP_ALIVE	gs_bool*	值为1时使用 SOCKET 的 KEEP_ALIVE 功能 值为0时关闭 SOCKET 的 KEEP_ALIVE 功能
GBASE_OPT_TCP_KEEPIDLE	unsigned int *	SOCKET 空闲多长时间后启动 SOCKET 的 KEEP_ALIVE 功能。默认 300 s
GBASE_OPT_TCP_KEEPINTVL	unsigned int *	每隔多少秒发送一次连接检测。默认 10 s。
GBASE_OPT_TCP_KEEP	unsigned int *	连续进行多少次连接检测。默

option 选项	参量类型	功　能
CNT		认 5 次。
GBASE_INIT_COMMAND	char *	连接到 GBASE 服务器时将执行的命令。再次连接时将自动地再次执行。
GBASE_OPT_CONNECT_TIMEOUT	unsigned int *	以秒为单位的连接超时。
GBASE_OPT_LOCAL_INFILE	指向单元的可选指针	如果未给定指针，或指针指向“unsigned int != 0”，将允许命令 LOAD LOCAL INFILE。
GBASE_OPT_PROTOCOL	unsigned int *	要使用的协议类型。应是 gbase.h 中定义的 gbase_protocol_type 的枚举值之一。
GBASE_OPT_READ_TIMEOUT	unsigned int *	从服务器读取信息的超时（目前仅在 Windows 平台的 TCP/IP 连接上有效)。
GBASE_OPT_RECONNECT	gs_bool *	如果发现连接丢失，启动或禁止与服务器的自动再连接。
GBASE_OPT_SET_CLIENT_IP	char *	对于与 libgbased 链接的应用程序（具备鉴定支持特性的已编译 libgbased)，它意味着，出于鉴定目的，用户将被视为从指定的 IP 地址（指定为字符串）进行连接。对于与 libgbaseclient 链接的应用程序，该选项将被忽略。
GBASE_OPT_WRITE_TIMEOUT	unsigned int *	写入服务器的超时（目前仅在 Windows 平台的 TCP/IP 连接上有效)。
GBASE_READ_DEFAULT	char *	从命名选项文件而不是从

option 选项	参量类型	功 能
_FILE		gs.cnf 读取选项。
GBASE_READ_DEFAULT_GROUP	char *	从 gs.cnf 或用 GBASE_READ_DEFAULT_FILE 指定的文件中的命名组读取选项。
GBASE_REPORT_DATA_TRUNCATION	gs_bool *	通过 GBASE_BIND.error，对于预处理语句，允许或禁止通报数据截断错误（默认为禁止）。
GBASE_SECURE_AUTH	gs_bool*	是否连接到不支持密码混编功能的服务器。
GBASE_SET_CHARSET_DIR	char*	指向包含字符集定义文件的目录的路径名。
GBASE_SET_CHARSET_NAME	char*	用作默认字符集的字符集的名称。
GBASE_SHARED_MEMORY_BASE_NAME	char*	命名为与服务器进行通信的共享内存对象。

注意，如果使用了 GBASE_READ_DEFAULT_FILE 或 GBASE_READ_DEFAULT_GROUP，总会读取客户端组。

选项文件中指定的组可能包含下述选项：

选 项	描 述
connect-timeout	以秒为单位的连接超时。在 Linux 平台上，该超时也用作等待服务器首次回应的时间。
compress	使用压缩客户端／服务器协议。
database	如果在连接命令中未指定数据库，连接到该数据库。
debug	调试选项。
disable-local-infile	禁止使用 LOAD DATA LOCAL。
host	默认主机名。

选　项	描　述
init-command	连接到 GBase 服务器时将执行的命令。再次连接时将自动地再次执行。
interactive-timeout	等同于将 CLIENT_INTERACTIVE 指定为 gbase_real_connect()。
local-infile[=(0|1)]	如果无参量或参量!= 0，那么将允许使用 LOAD DATA LOCAL。
max_allowed_packet	客户端能够从服务器读取的最大信息包。
multi-results	允许多语句执行或存储程序的多个结果集。
multi-statements	允许客户端在 1 个字符串内发送多条语句。(由";"隔开)。
password	默认密码。
pipe	使用命名管道连接到 NT 平台上的 GBase 服务器。
protocol={TCP | SOCKET | PIPE | MEMORY}	连接到服务器时将使用的协议。
port	默认端口号。
return-found-rows	通知 gbase_info()返回发现的行，而不是使用 UPDATE 时更新的行。
shared-memory-base-name=name	共享内存名称，用于连接到服务器。
socket	默认的套接字文件。
user	默认用户。

◆ 返回值:

成功时返回 0。如果使用了未知选项，返回非 0 值。

3.2.40 gbase_ping

◆ 摘要:

检查与服务器的连接是否工作。如果连接丢失，将自动尝试再连接。

该函数可被闲置了较长时间的客户端使用，用以检查服务器是否已关闭了连接，并在必要时再次连接。

◆ 语法：

```
int     gbase_ping(GBASE *gbase);
```

◆ 参数：

◆ 返回值：

如果与服务器的连接有效返回 0。如果出现错误，返回非 0 值。返回的非 0 值不表示 GBase 服务器本身是否已关闭，连接可能因其他原因终端，如网络问题等。

3.2.41 gbase_query

◆ 摘要：

执行由“NULL 终结的字符串”查询指向的 SQL 查询。正常情况下，字符串必须包含 1 条 SQL 语句，而且不应为语句添加终结分号（‘;’）或“\g”。如果允许多语句执行，字符串可包含多条由分号隔开的语句。

gbase_query()不能用于包含二进制数据的查询，应使用 gbase_real_query()取而代之(二进制数据可能包含字符‘\0’，gbase_query()会将该字符解释为查询字符串结束)。

◆ 语法：

```
int gbase_query(GBASE *gbase, const char *q);
```

◆ 参数：

◆ 返回值：

如果查询成功，返回 0。如果出现错误，返回非 0 值。

◆ 错误：

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.42 gbase_real_connect

◆ 摘要：

gbase_real_connect()尝试与运行在主机上的 GBase 数据库引擎建立连接。在你能够执行需要有效 GBase 连接句柄结构的任何其他 API 函数之前，gbase_real_connect()必须成功完成。

◆ 语法：

```
GBASE * gbase_real_connect(GBASE *gbase,
   const char *host,
                        const char *user,
                        const char *passwd,
                        const char *db,
                        unsigned int port,
                        const char *unix_socket,
                        unsigned long clientflag);
```

◆ 参数：

gbase	调用 gbase_real_connect()之前，必须调用 gbase_init()来初始化 GBASE 结构。
host	必须是主机名或 IP 地址。如果“host”是 NULL 或字符串"localhost"，连接将被视为与本地主机的连接。如果操作系统支持套接字（Unix）或命名管道（Windows），将使用它们而不是 TCP/IP 连接到服务器。如果使用支持集群 IP 路由的 GBase C API 连接 GBase 8a 数据库集群时，host 可以是包含几个集群节点 IP 的字符串，IP 之间以“;”进行分割。这样可以使用 GBase C API 的集群 IP 路由功能。
user	用户的 GBase 登录 ID。如果“user”是 NULL 或空字符串""，用户将被视为当前用户。
passwd	用户的密码
db	数据库名称。如果 db 为 NULL，连接会将默认的数据库设为该值。
port	如果“port”不是 0，其值将用作 TCP/IP 连接的端口号。注意，“host”参数决定了连接的类型。
unix_socket	如果 unix_socket 不是 NULL，该字符串描述了应使用的套接字或命名管道。注意，“host”参数决定了连接的类型。
clientflag	其值通常为 0，但是，也能将其设置为下述标志的组合，以允许特定功能：

标志名称	标志描述
CLIENT_COMPRESS	使用压缩协议。

标志名称	标志描述
CLIENT_FOUND_ROWS	返回发现的行数（匹配的），而不是受影响的行数。
CLIENT_IGNORE_SPACE	允许在函数名后使用空格。使所有的函数名成为保留字。
CLIENT_INTERACTIVE	关闭连接之前，允许 interactive_timeout（取代了 wait_timeout）秒的不活动时间。客户端的会话 wait_timeout 变量被设为会话 interactive_timeout 变量的值。
CLIENT_LOCAL_FILES	允许 LOAD DATA LOCAL 处理功能。
CLIENT_MULTI_STATEMENTS	通知服务器，客户端可能在单个字符串内发送多条语句（由';'隔开）。如果未设置该标志，将禁止多语句执行。
CLIENT_MULTI_RESULTS	通知服务器，客户端能够处理来自多语句执行或存储程序的多个结果集。如果设置了 CLIENT_MULTI_STATEMENTS，将自动设置它。
CLIENT_NO_SCHEMA	禁止 db_name.tbl_name.col_name 语法。它用于 ODBC。如果使用了该语法，它会使分析程序生成错误，在捕获某些 ODBC 程序中的缺陷时，它很有用。
CLIENT_ODBC	客户端是 ODBC 客户端。它将 gbased 变得更为 ODBC 友好。
CLIENT_SSL	使用 SSL（加密协议）。该选项不应由应用程序设置，它是在客户端库内部设置的。

◆ 返回值：

如果连接成功，返回 GBASE*连接句柄。如果连接失败，返回 NULL。对于成功的连接，返回值与第 1 个参数的值相同。

◆ 错误

CR_CONN_HOST_ERROR	无法连接到 GBase 服务器。
CR_CONNECTION_ERROR	无法连接到本地 GBase 服务器。
CR_IPSOCK_ERROR	无法创建 IP 套接字。
CR_OUT_OF_MEMORY	内存溢出。
CR_SOCKET_CREATE_ERROR	无法创建 Unix 套接字。
CR_UNKNOWN_HOST	无法找到主机名的 IP 地址。
CR_VERSION_ERROR	协议不匹配
CR_NAMEDPIPEOPEN_ERROR	无法在 Windows 平台下创建命名管道。
CR_NAMEDPIPEWAIT_ERROR	在 Windows 平台下等待命名管道失败。
CR_NAMEDPIPESETSTATE_ERROR	在 Windows 平台下获取管道处理程序失败。
CR_SERVER_LOST	如果 connect_timeout > 0，而且在连接服务器时所用时间长于 connect_timeout 秒，或在执行 init-command 时服务器消失。

3.2.43 gbase_real_escape_string

◆ 摘要：

该函数用于创建可在 SQL 语句中使用的合法 SQL 字符串。按照连接的当前字符集，将“from”中的字符串编码为转义 SQL 字符串。将结果置于“to”中。并添加 1 个终结用 NULL 字节。

如果需要更改连接的字符集，应使用 gbase_set_character_set()函数。

◆ 语法：

```
unsigned long gbase_real_escape_string(GBASE *gbase,
char *to,
```

```
const char *from,
 unsigned long length);
```

◆ 参数:

gbase	数据库句柄。
to	必须为“to”缓冲区分配至少 length*2+1 字节。在最坏的情况下，每个字符或许需要使用 2 个字节进行编码，而且还需要终结 NULL 字节。
from	“from”指向的字符串必须是长度字节“long”。
length	from 字符串的长。

◆ 返回值:

置于“to”中的值的长度，不包括终结用 NULL 字符。

3.2.44 gbase_real_query

◆ 摘要:

执行由“query”指向的 SQL 查询，它应是字符串长度字节“long”。正常情况下，字符串必须包含 1 条 SQL 语句，而且不应为语句添加终结分号（‘;’）或“\g”。如果允许多语句执行，字符串可包含由分号隔开的多条语句。

◆ 语法:

```
int     gbase_real_query(GBASE *gbase, const char *q,   unsigned long length);
```

◆ 参数:

◆ 返回值:

如果查询成功，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.45 gbase_refresh

◆ 摘要:

该函数用于刷新表或高速缓冲，或复位复制服务器信息。连接的用户必须具有 RELOAD 权限。

◆ 语法:

```
int     STDCALL gbase_refresh(GBASE *gbase, unsigned int
refresh_options);
```

◆ 参数:

gbase	数据库句柄
refresh_options	这是是一种位掩码，由下述值的任意组合构成。能够以“or”(或) 方式将多个值组合在一起，用一次调用执行多项操作。
REFRESH_GRANT	刷新授权表，与 FLUSH PRIVILEGES 类似。
REFRESH_LOG	刷新日志，与 FLUSH LOGS 类似。
REFRESH_TABLES	刷新表高速缓冲，与 FLUSH TABLES 类似。
REFRESH_HOSTS	刷新主机高速缓冲，与 FLUSH HOSTS 类似。
REFRESH_STATUS	复位状态变量，与 FLUSH STATUS 类似。

REFRESH_THREADS	刷新线程高速缓冲。
REFRESH_SLAVE	在从复制服务器上，复位主服务器信息，并重新启动从服务器，与 RESET SLAVE 类似。
REFRESH_MASTER	在主复制服务器上，删除二进制日志索引中列出的二进制日志文件，并截短索引文件，与 RESET MASTER 类似。

◆ 返回值：

0 表示成功，非 0 值表示出现错误。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.46 gbase_rollback

◆ 摘要：

回滚当前事务。

◆ 语法：

```
gs_bool gbase_rollback(GBASE * gbase);
```

◆ 参数：

◆ 返回值：

如果成功，返回 0，如果出现错误，返回非 0 值。

3.2.47 gbase_row_seek

◆ 摘要:

将行光标置于查询结果集中的任意行。

该函数要求在结果集的结构中包含查询的全部结果，因此，gbase_row_seek()仅应与 gbase_store_result()一起使用，而不是与 gbase_use_result()。

◆ 语法:

```
GBASE_ROW_OFFSET  gbase_row_seek(GBASE_RES *result,
GBASE_ROW_OFFSET offset);
```

◆ 参数:

result 结果集。

offset offset 值是行偏移量，它应是从 gbase_row_tell()或 gbase_row_seek()返回的值。该值不是行编号，如果你打算按编号查找结果集中的行，请使用 gbase_data_seek()。

◆ 返回值:

行光标的前一个值。该值可传递给对 gbase_row_seek()的后续调用。

3.2.48 gbase_row_tell

◆ 摘要:

对于上一个 gbase_fetch_row()，返回行光标的当前位置。该值可用作 gbase_row_seek()的参量。

仅应在 gbase_store_result()之后，而不是 gbase_use_result()之后使用 gbase_row_tell()。

◆ 语法：

```
GBASE_ROW_OFFSET  gbase_row_tell(GBASE_RES *res);
```

◆ 参数：

◆ 返回值：

行光标的当前偏移量。

3.2.49 gbase_select_db

◆ 摘要：

将参数 db 的值作为当前连接的默认数据库。在后续查询中，该数据库将是未包含明确数据库区分符的表引用的默认数据库。

除非已连接的用户具有使用数据库的权限，否则 gbase_select_db()将失败。

◆ 语法：

```
int     gbase_select_db(GBASE *gbase, const char *db);
```

◆ 参数：

◆ 返回值：

0 表示成功，非 0 值表示出现错误。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.50 gbase_library_init

◆ 摘要:

该函数用于初始化 GBase CAPI 库。在使用 GBase CAPI 之前必须调用该函数。gbase_init 函数内部会调用 gbase_library_init，以确保使用 GBase CAPI 库使用时调用了 gbase_library_init。

◆ 语法:

```
int gbase_library_init(int argc, char **argv, char **groups);
```

◆ 参数:

argc 取值 0

argv 取值 NULL

groups 取值 NULL

◆ 返回值:

0 表示成功，非 0 值表示出现错误。

3.2.51 gbase_library_end

◆ 摘要:

在结束使用 GBase CAPI 库时，释放 GBase CAPI 库申请的资源。

◆ 语法:

```
void gbase_library_end()
```

◆ 参数:

◆ 返回值:

3.2.52 gbase_set_character_set

◆ 摘要:

该函数用于为当前连接设置默认的字符集。

◆ 语法:

```
int gbase_set_character_set(GBASE*gbase, const char *csname)
```

◆ 参数:

gbase 数据库句柄

csname 字符集名称，取值'gbk', 'utf8'。

◆ 返回值:

0 表示成功，非 0 值表示出现错误。

3.2.53 gbase_set_server_option

◆ 摘要:

允许或禁止连接的选项

◆ 语法:

```
int gbase_set_server_option(GBASE *gbase, enum
enum_gbase_set_option option);
```

◆ 参数:

gbase 数据库句柄

option GBASE_OPTION_MULTI_STATEMENTS_ON 允许多语句支持

GBASE_OPTION_MULTI_STATEMENTS_OFF 禁止多语句支持

◆ 返回值：

0 表示成功，非 0 值表示出现错误。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。

3.2.54 gbase_sqlstate

◆ 摘要：

返回由 NULL 终结的字符串，该字符串包含关于上次错误的 SQLSTATE 错误代码。错误代码包含 5 个字符。'00000'表示无错误。其值由 ANSI SQL 和 ODBC 指定

注意，并非所有的 GBase 错误均会被映射到 SQLSTATE 错误代码。值'HY000'（一般错误）用于未映射的错误。

◆ 语法：

```
const char *gbase_sqlstate(GBASE *gbase);
```

◆ 参数：

◆ 返回值：

包含 SQLSTATE 错误码的、由 NULL 终结的字符串。

3.2.55 gbase_shutdown

◆ 摘要：

请求数据库服务器关闭。已连接的用户必须具有 SHUTDOWN 权限。

◆ 语法：

```
int      gbase_shutdown(GBASE *gbase,  enum
gbase_enum_shutdown_level shutdown_level);
```

◆ 参数：

shutdown_level	必须等效于 SHUTDOWN_DEFAULT

◆ 返回值：

0 表示成功，非 0 值表示出现错误。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.56 gbase_stat

◆ 摘要：

返回包含特定信息的字符串。

◆ 语法：

```
const char *    gbase_stat(GBASE *gbase);
```

◆ 参数：

◆ 返回值：

描述服务器状态的字符集。如果出现错误，返回 NULL。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。

CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.57 gbase_store_result

◆ 摘要:

检索完整的结果集至客户端。如果成功，gbase_store_result()将复位 gbase_error()和 gbase_errno()。

◆ 语法:

```
GBASE_RES *  gbase_store_result(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

具有多个结果的 GBASE_RES 结果集合。如果出现错误，返回 NULL。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_OUT_OF_MEMORY	内存溢出。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.58 gbase_thread_init

◆ 摘要:

在线程里使用 GBase CAPI 时，需要首先调用该函数，然后再调用其它 GBase

CAPI 函数。

- 语法：

gs_bool gbase_thread_init()

- 参数：
- 返回值：

0 表示成功，非 0 值表示出现错误。

3.2.59 gbase_thread_end

- 摘要：

在线程里结束使用 GBase CAPI 时，需要调用该函数以清理 GBase CAPI 申请的资源。

- 语法：

void gbase_thread_end(void)

- 参数：
- 返回值：

3.2.60 gbase_thread_id

- 摘要：

返回当前连接的线程 ID。该值可用作 gbase_kill()的参量以杀死线程。

- 语法：

unsigned long gbase_thread_id(GBASE *gbase);

- 参数：
- 返回值：

当前连接的线程 ID。

3.2.61 gbase_use_result

◆ 摘要:

初始化逐行的结果集检索。如果成功，gbase_use_result()将复位 gbase_error()和 gbase_errno()。

◆ 语法:

```
GBASE_RES *  gbase_use_result(GBASE *gbase);
```

◆ 参数:

◆ 返回值:

GBASE_RES 结果结构。如果出现错误，返回 NULL。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_OUT_OF_MEMORY	内存溢出。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

3.2.62 gbase_warning_count

◆ 摘要:

返回上一个 SQL 语句的告警数。

◆ 语法:

```
unsigned int  gbase_warning_count(GBASE *gbase);
```

- 参数:
- 返回值:

告警计数。

4 C API 预处理

4.1数据类型

4.1.1GBASE_STMT

◆ 结构说明

该结构表示预处理语句。通过调用 gbase_stmt_init()创建语句，返回语句句柄，即指向 GBASE_STMT 的指针。该句柄用户所有后续的与语句有关的函数，直至使用 gbase_stmt_close()关闭了它为止。

GBASE_STMT 结构没有供应用程序使用的参数。此外，不应尝试复制 GBASE_STMT 结构。不保证这类复制物会有用。

多个语句句柄能够与单个连接关联起来。对句柄数目的限制取决于系统资源。

4.1.2GBASE_BIND

◆ 结构说明

该结构用于语句输入（发送给服务器的数据值）和输出（从服务器返回的结果值)。对于输入，它与 gbase_stmt_bind_param()一起使用，用于将参数数据值绑定到缓冲区上，以供 gbase_stmt_execute()使用。对于输出，它与 gbase_stmt_bind_result()一起使用，用于绑定结果缓冲区，以便用于用 gbase_stmt_fetch()以获取行。

GBASE_BIND 结构包含下述供应用程序使用的成员。每个成员用于输入和输出，但在某些时候，也能用于不同的目的，具体情况取决于数据传输的方向。

◆ 结构成员

名 称	类 型	描 述
buffer_type	enum_field_types	缓冲的类型。
buffer	void *	指向缓冲区的指针。
buffer_length	unsigned long	缓冲区的实现大小，单位为字节。
length	unsigned long*	指向 unsigned long 变量的指针，该变量指明了存储在*buffer 中数据的实际字节数。
is_null	gs_bool*	该成员指向 gs_bool 变量，如果值为 NULL，该变量为“真”，如果值为非 NULL，该变量为“假”。“is_null”是指向布尔类型的指针，而不是布尔标量，以便能以下述方式使用它： 1、如果数据值总是 NULL，使用 GBASE_TYPE_NULL 绑定列。 2、如果数据值总是 NOT NULL，设置 is_null = (gs_bool*) 0。 3、在所有其他情况下，应将 is_null 设置为 gs_bool 变量的地址，并在各次执行之间恰当地更改变量的值，以指明数据值是 NULL 或 NOT NULL。
is_unsigned	gs_bool	对于无符号类型，应将“is_unsigned”设置为“真”，对于带符号类型，应将其设置为“假”。

名 称	类 型	描 述
error	gs_bool	对于输出，该成员用于通报数据截短错误。必须通过调用带有 GBASE_REPORT_DATA_TRUNCATION 选项的 gbase_options()，启用截短通报功能。允许该功能后，gbase_stmt_fetch()返回 GBASE_DATA_TRUNCATED，而且对于出现截短情况的参数，在 GBASE_BIND 结构中，错误标志为“真”。截短指明丢失了符号或有效位数，或字符串过长以至于无法容纳在1列中。

4.1.3 GBASE_TIME

◆ 结构说明

该结构用于将 DATE、TIME、DATETIME 和 TIMESTAMP 数据直接发送到服务器，或从服务器直接接收这类数据。

◆ 结构成员

名 称	类 型	描 述
year	unsigned int	年份
month	unsigned int	月份
day	unsigned int	天
hour	unsigned int	小时
minute	unsigned int	分钟
second	unsigned int	秒
neg	unsigned int	布尔标志，用于指明时间是否为负数

4.2函数描述

4.2.1gbase_stmt_affected_rows

- 摘要:

返回由预处理语句 UPDATE、DELETE 或 INSERT 变更、删除或插入的行数目。

- 语法:

```
gs_ulonglong  gbase_stmt_affected_rows(GBASE_STMT *stmt);
```

- 参数:
- 返回值:

大于 0 的整数指明了受影响或检索的行数

4.2.2gbase_stmt_attr_get

- 摘要:

获取预处理语句属性的值。

- 语法:

```
gs_bool gbase_stmt_attr_get(GBASE_STMT *stmt, enum
enum_stmt_attr_type attr_type, void *attr);
```

- 参数:

attr_type 和 atttr 类型的清单参考 gbase_stmt_attr_set()。

- 返回值:

如果 OK，返回 0。如果选项未知，返回非 0 值。

4.2.3 gbase_stmt_attr_set

◆ 摘要:

设置预处理语句的属性。

◆ 语法:

gs_bool gbase_stmt_attr_set(GBASE_STMT *stmt, enum enum_stmt_attr_type attr_type, const void *attr)

◆ 参数:

“attr_type”参量是希望设置的选项，“attr”参量是选项的值。

选 项	参量类型	功 能
STMT_ATTR_UPDATE_MAX_LENGTH	gs_bool *	如果设为 1: 更新 gbase_stmt_store_result()中的元数据 GBASE_FIELD->max_length。
STMT_ATTR_CURSOR_TYPE	unsigned long *	调用 gbase_stmt_execute()时，语句将打开的光标类型。*arg 可以是 CURSOR_TYPE_NO_CURSOR（默认值）或 CURSOR_TYPE_READ_ONLY。
STMT_ATTR_PREFETCH_ROWS	unsigned long *	使用光标时，一次从服务器获取的行数。*arg 的范围从 1 到 unsigned long 的最大值。默认值为 1。

◆ 返回值:

如果 OK，返回 0。如果选项未知，返回非 0 值。

4.2.4 gbase_stmt_bind_param

◆ 摘要:

将应用程序数据缓冲与预处理 SQL 语句中的参数标记符关联起来。

◆ 语法:

gs_bool gbase_stmt_bind_param(GBASE_STMT * stmt, GBASE_BIND * bnd);

◆ 参数:

◆ 返回值:

如果绑定成功，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_INVALID_BUFFER_USE	指明“bind”(绑定) 是否将提供程序块中的长数据，以及缓冲类型是否为非字符串或二进制类型。
CR_UNSUPPORTED_PARAM_TYPE	不支持该转换。或许 buffer_type 值是非法的，或不是所支持的类型之一。
CR_OUT_OF_MEMORY	内存溢出。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.5 gbase_stmt_bind_result

◆ 摘要:

将应用程序数据缓冲与结果集中的列关联起来。

◆ 语法:

gs_bool STDCALL gbase_stmt_bind_result(GBASE_STMT * stmt, GBASE_BIND * bnd);

◆ 参数:

◆ 返回值：

如果绑定成功，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_UNSUPPORTED_PARAM_TYPE	不支持该转换。或许 buffer_type 值是非法的，或不是所支持的类型之一。
CR_OUT_OF_MEMORY	内存溢出。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.6 gbase_stmt_close

◆ 摘要：

释放预处理语句使用的内存。

◆ 语法：

```
gs_bool  gbase_stmt_close(GBASE_STMT * stmt);
```

◆ 参数：

◆ 返回值：

如果成功释放了语句，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.7 gbase_stmt_data_seek

◆ 摘要：

寻找语句结果集中的任意行编号。

- 语法：

```
void gbase_stmt_data_seek(GBASE_STMT *stmt, gs_ulonglong offset);
```

- 参数：
- 返回值：

无

4.2.8 gbase_stmt_errno

- 摘要：

返回上次语句执行的错误编号。

- 语法：

```
unsigned int  gbase_stmt_errno(GBASE_STMT * stmt);
```

- 参数：
- 返回值：

错误代码值。如果未出现错误，返回 0。

4.2.9 gbase_stmt_error

- 摘要：

返回上次语句执行的错误消息。

- 语法：

```
const char *gbase_stmt_error(GBASE_STMT * stmt);
```

- 参数：
- 返回值：

描述了错误的字符串。如果未出现错误，返回空字符串。

4.2.10 gbase_stmt_execute

◆ 摘要:

执行预处理语句。

◆ 语法:

```
int  gbase_stmt_execute(GBASE_STMT *stmt);
```

◆ 参数:

◆ 返回值:

如果执行成功，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_OUT_OF_MEMORY	内存溢出。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误

4.2.11 gbase_stmt_fetch

◆ 摘要:

从结果集获取数据的下一行，并返回所有绑定列的数据。

◆ 语法:

```
int  gbase_stmt_fetch(GBASE_STMT *stmt);
```

◆ 参数:

◆ 返回值:

返回值	描 述
0	成功，数据被提取到应用程序数据缓冲区。
1	出现错误。通过调用 gbase_stmt_errno() 和 gbase_stmt_error()，可获取错误代码和错误消息。
GBASE_NO_DATA	不存在行／数据。
GBASE_DATA_TRUNCATED	出现数据截短。

4.2.12 gbase_stmt_fetch_column

◆ 摘要:

获取结果集当前行中某列的数据。

◆ 语法:

```
int  gbase_stmt_fetch_column(GBASE_STMT *stmt,
  GBASE_BIND *bind_arg,,
  unsigned int column,
  unsigned long offset);
```

◆ 参数:

◆ 返回值:

如果成功获取了值，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_INVALID_PARAMETER_NO　　无效的列号。

CR_NO_DATA　　已抵达结果集的末尾。

4.2.13 gbase_stmt_field_count

- 摘要:

对于最近的语句，返回结果行的数目。

- 语法:

```
unsigned int  gbase_stmt_field_count(GBASE_STMT *stmt);
```

- 参数:
- 返回值:

表示结果集中行数的无符号整数。

4.2.14 gbase_stmt_free_result

- 摘要:

释放分配给语句句柄的资源。

- 语法:

```
gs_bool  gbase_stmt_free_result(GBASE_STMT *stmt);
```

- 参数:
- 返回值:

如果成功释放了结果集，返回 0。如果出现错误，返回非 0 值。

4.2.15 gbase_stmt_init

- 摘要:

为 GBASE_STMT 结构分配内存并初始化它。应使用 gbase_stmt_close(GBASE_STMT *)释放。

◆ 语法：

```
GBASE_STMT *  gbase_stmt_init(GBASE *gbase);
```

◆ 参数：

◆ 返回值：

成功时，返回指向 GBASE_STMT 结构的指针。如果内存溢出，返回 NULL。

◆ 错误

CR_OUT_OF_MEMORY　　内存溢出。

4.2.16 gbase_stmt_insert_id

◆ 摘要：

对于预处理语句的 AUTO_INCREMENT 列，返回生成的 ID。

◆ 语法：

```
gs_ulonglong  gbase_stmt_insert_id(GBASE_STMT *stmt);
```

◆ 参数：

◆ 返回值：

为在执行预处理语句期间自动生成或明确设置的 AUTO_INCREMENT 列返回值，或由 LAST_INSERT_ID(expr)函数生成的值。如果语句未设置 AUTO_INCREMENT 值，返回值不确定。

4.2.17 gbase_stmt_num_rows

◆ 摘要：

从语句缓冲结果集返回总行数。

◆ 语法：

```
gs_ulonglong  gbase_stmt_num_rows(GBASE_STMT *stmt);
```

- 参数:
- 返回值:

结果集中的行数。

4.2.18 gbase_stmt_param_count

- 摘要:

返回预处理 SQL 语句中的参数数目。

- 语法:

```
unsigned long  gbase_stmt_param_count(GBASE_STMT * stmt);
```

- 参数:
- 返回值:

表示语句中参数数目的无符号长整数。

4.2.19 gbase_stmt_prepare

- 摘要:

为执行操作准备 SQL 字符串。并返回状态值。字符串长度应由“length”参量给出。字符串必须包含 1 条 SQL 语句。不应为语句添加终结用分号(‘;’)或\g。

通过将问号字符“?”嵌入到 SQL 字符串的恰当位置，应用程序可包含 SQL 语句中的一个或多个参数标记符。

标记符仅在 SQL 语句中的特定位置时才是合法的。例如，它可以在 INSERT 语句的 VALUES()列表中(为行指定列值),或与 WHERE 子句中某列的比较部分(用以指定比较值)。但是，对于 ID (例如表名或列名)，不允许使用它们，不允许

指定二进制操作符（如等于号“=”）的操作数。后一个限制是有必要的，原因在于，无法确定参数类型。一般而言，参数仅在 DML（数据操作语言）语句中才是合法的，在 DDL（数据定义语言）语句中不合法。

执行语句之前，必须使用 gbase_stmt_bind_param()，将参数标记符与应用程序变量绑定在一起。

◆ 语法：

```
int  gbase_stmt_prepare(GBASE_STMT *stmt, const char *query,
unsigned long length);
```

◆ 参数：

◆ 返回值：

如果成功处理了语句，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_OUT_OF_MEMORY	内存溢出。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.20 gbase_stmt_reset

◆ 摘要：

在客户端和服务器上，将预处理语句复位为完成准备后的状态。主要用于复位用 gbase_stmt_send_long_data()发出的数据。对于语句，任何已打开的光标将被关闭。

要想重新准备用于另一查询的语句，可使用 gbase_stmt_prepare()。

◆ 语法：

```
gs_bool  gbase_stmt_reset(GBASE_STMT * stmt);
```

◆ 参数：

◆ 返回值：

如果语句成功复位，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.21 gbase_stmt_result_metadata

◆ 摘要：

以结果集形式返回预处理语句元数据。

如果传递给 gbase_stmt_prepare()的语句能够成生结果集，gbase_stmt_result_metadata()将以指针的形式返回结果集元数据，该指针指向 GBASE_RES 结构，可用于处理元信息，如总的字段数以及单独的字段信息。该结果集指针可作为参量传递给任何基于字段且用于处理结果集元数据的 API 函数，如：

gbase_num_fields()

gbase_fetch_field()

gbase_fetch_field_direct()

gbase_fetch_fields()

gbase_field_count()

gbase_field_seek()

gbase_field_tell()

gbase_free_result()

完成操作后，应释放结果集结构，可通过将其传递给 gbase_free_result()完成。它与释放通过 gbase_store_result()调用获得的结果集的方法类似。

gbase_stmt_result_metadata()返回的结果集仅包含元数据。不含任何行结果。与 gbase_stmt_fetch()一起使用语句句柄，可获取行。

◆ 语法:

```
GBASE_RES * gbase_stmt_result_metadata(GBASE_STMT *stmt);
```

◆ 参数:

◆ 返回值:

GBASE_RES 结果结构。如果不存在关于预处理查询的任何元信息，返回 NULL。

◆ 错误

CR_OUT_OF_MEMORY	内存溢出。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.22 gbase_stmt_row_seek

◆ 摘要:

使用从 gbase_stmt_row_tell()返回的值，查找语句结果集中的行偏移。

◆ 语法:

```
GBASE_ROW_OFFSET  gbase_stmt_row_seek(GBASE_STMT *stmt,
GBASE_ROW_OFFSET offset);
```

◆ 参数:

◆ 返回值：

行光标的前一个值。可以将该值换递给后续的 gbase_stmt_row_seek()调用。

4.2.23 gbase_stmt_row_tell

◆ 摘要：

返回语句行光标位置。仅应在 gbase_stmt_store_result()之后使用 gbase_stmt_row_tell()。

◆ 语法：

```
GBASE_ROW_OFFSET  gbase_stmt_row_tell(GBASE_STMT *stmt);
```

◆ 参数：

◆ 返回值：

行光标的当前偏移量。

4.2.24 gbase_stmt_send_long_data

◆ 摘要：

将程序块中的长数据发送到服务器。允许应用程序分段地（分块）将参数数据发送到服务器。可以多次调用该函数，以便发送关于某一列的字符或二进制数据的不同部分，列必须是 TEXT 或 BLOB 数据类型之一。

◆ 语法：

```
gs_bool  gbase_stmt_send_long_data(GBASE_STMT *stmt,
unsigned int  param_number,
const char *  data,
unsigned long length);
```

◆ 参数:

Stmt	预处理语句句柄
param_number	指明了与数据关联的参数。参数从 0 开始编号。
data	是指向包含将要发送的数据的缓冲区的指针。
length	指明了缓冲区内的字节数。

◆ 返回值:

如果成功地将数据发送到服务器，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_OUT_OF_MEMORY	内存溢出。
CR_UNKNOWN_ERROR	出现未知错误。

4.2.25 gbase_stmt_sqlstate

◆ 摘要:

返回关于上次语句执行的 SQLSTATE 错误代码。

◆ 语法:

```
const char * gbase_stmt_sqlstate(GBASE_STMT * stmt);
```

◆ 参数:

◆ 返回值:

包含 SQLSTATE 错误代码、由 NULL 终结的字符串。

4.2.26 gbase_stmt_store_result

◆ 摘要:

将完整的结果集检索到客户端。以便后续的 gbase_stmt_fetch()调用能返回缓冲数据。

◆ 语法:

```
int  gbase_stmt_store_result(GBASE_STMT *stmt);
```

◆ 参数:

◆ 返回值:

如果成功完成了对结果的缓冲处理，返回 0。如果出现错误，返回非 0 值。

◆ 错误

CR_COMMANDS_OUT_OF_SYNC	以不恰当的顺序执行了命令。
CR_OUT_OF_MEMORY	内存溢出。
CR_SERVER_GONE_ERROR	GBase 服务器不可用。
CR_SERVER_LOST	在查询过程中，与服务器的连接丢失。
CR_UNKNOWN_ERROR	出现未知错误。

5 GBase C API 应用示例

5.1使用 GBase C API 创建连接

```
GBASE* gbase=NULL;
/*初始化 GBASE 结构体*/
if(!(gbase = gbase_init(0)))
{
    fprintf(stderr, "不能初始化 GBASE 结构体! \n");
    exit(1);
}
/*数据库连接*/
if(!gbase_real_connect(gbase, host, user, passwd, db, port, NULL,
0))
{
    fprintf(stderr, "\n%s\n", gbase_error(gbase));
    exit(1);
}
/*释放数据库连接句柄*/
gbase_close(gbase);
```

5.2使用 GBase C API 连接集群

```
GBASE* gbase=NULL;
CHAR* host=" 192.168.5.64;192.168.5.67;" ; // 集群各节点的 IP 地址
/*初始化 GBASE 结构体*/
if(!(gbase = gbase_init(0)))
{
    fprintf(stderr, "不能初始化 GBASE 结构体! \n");
    exit(1);
}
```

```
    /*数据库连接*/
    if(!gbase_real_connect(gbase, host, user, passwd, db, port, NULL,
0))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }
    /*释放数据库连接句柄*/
    gbase_close(gbase);
```

5.3使用 GBase C API 执行 SQL 语句

```
    /*打印表中的数据，示例 demo_1，demo_2，demo_3 调用此函数*/
    void print_table(GBASE* gbase, char* table_name)
    {
        GBASE_RES* res = NULL;
        GBASE_FIELD* fields = NULL;
        GBASE_ROW row;
        unsigned int num_fields;
        char sql_select[500] = "select * from ";
        int i=0;

        while(table_name[i]){sql_select[14+i] = table_name[i]; i++;}
        sql_select[14+i] = '\0';
        if(gbase_query(gbase, sql_select))
        {
            printf("%d\t%s\n", gbase_errno(gbase),
gbase_error(gbase));
            exit(0);
        }
        res = gbase_store_result(gbase);
        num_fields = gbase_num_fields(res);
        while(fields = gbase_fetch_field(res))
        {
            printf("|%s\t", fields->name);
```

```
        }
        printf("|\n");
        for(i=0;i<num_fields; i++)
        {
            printf("--------");
        }
        printf("\n");
        while((row = gbase_fetch_row(res)) != NULL)
        {
            for(i=0; i < num_fields; i++)
            {
                printf("|%s\t", row[i]);
            }
            printf("|\n");
        }
        gbase_free_result(res);
    }
    /*
    使用 sql 文创建表，并进行数据增、删、查、改操作
    */
    void demo_1(void)
    {
        GBASE* gbase=NULL;
        const char* sql_drop_table  = "drop table if exists g_demo1";
        const char* sql_create_table = "create table g_demo1(id int,
name varchar(20))";
        const char* insert_mode = "insert into g_demo1(id,
name)values(%d, '%s')";
        int id[NUM_INSERT];
        char name[NUM_INSERT][20];
        char sql_insert[200];
        const char* delete_mode = "delete from g_demo1 where id=%d";
        char sql_delete[200];
        const char* update_mode = "update g_demo1 set name='udpate'
where id=%d";
        char sql_update[200];
```

```
        const char* sql_select  = "select * from g_demo1";
        GBASE_RES* res  = NULL;
        GBASE_FIELD* fields = NULL;
        GBASE_ROW row;

        unsigned int i = 0;
        unsigned int num_fields = 0;

        /*初始化 GBASE 结构体*/
        if(!(gbase = gbase_init(0)))
        {
            fprintf(stderr, "不能初始化 GBASE 结构体! \n");
            exit(1);
        }

        /*数据库连接*/
        if(!gbase_real_connect(gbase, host, user, passwd, db, port,
NULL, 0))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
        }
        gbase->reconnect = 1;

        /*在数据库中创建表*/
        if(gbase_query(gbase, sql_drop_table))
        {
            fprintf(stderr, "%s", gbase_error(gbase));
            exit(1);
        }
        if(gbase_query(gbase, sql_create_table))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
        }
        /*向数据库表中插入数据*/
        for(i=0; i<NUM_INSERT; i++)
```

```
{
    id[i] = i;
    sprintf(name[i], "name_%d", i);
}
for(i=0; i<NUM_INSERT; i++)
{
    sprintf(sql_insert, insert_mode, id[i], name[i]);
    if(gbase_query(gbase, sql_insert))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }
}
/*从数据库表中选择出数据，并显示*/
if(gbase_query(gbase, sql_select))
{
    fprintf(stderr, "\n%s\n", gbase_error(gbase));
    exit(1);
}
res = gbase_store_result(gbase);
printf("插入操作后表的内容\n");
if(res)
{
    num_fields = gbase_num_fields(res);
    while(fields = gbase_fetch_field(res))
    {
        printf("\t| %s ", fields->name);
    }
    printf("\t|\n");
    while((row = gbase_fetch_row(res)) != NULL)
    {
        for(i=0; i < num_fields; i++)
        {
            printf("\t| %s ", row[i]);
        }
        printf("\t|\n");
    }
```

```
    }

    /*更新数据库表中数据*/
    sprintf(sql_update, update_mode, id[0]);
    if(gbase_query(gbase, sql_update))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }
    printf("更新操作后表的内容\n");
    print_table(gbase, "g_demo1");

    /*删除数据库表中数据*/
    sprintf(sql_delete, delete_mode, id[1]);
    if(gbase_query(gbase, sql_delete))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }
    printf("删除操作后表的内容\n");
    print_table(gbase, "g_demo1");
    /*释放结果集*/
    gbase_free_result(res);
    /*释放数据库连接句柄*/
    gbase_close(gbase);
}

/*
使用预处理语句进行数据插入和选择操作
*/
void demo_2(void)
{
    GBASE* gbase=NULL;
    GBASE_STMT* stmt;
    GBASE_BIND bind[2];
```

```
        const char* sql_drop_table  = "drop table if exists g_demo2";
        const char* sql_create_table = "create table g_demo2(id int,
name varchar(20))";
        const char* insert_mode  = "insert into g_demo2(id,
name)values(?, ?)";
        int in_id[NUM_INSERT];
        char in_name[NUM_INSERT][20];
        const char* update_mode = "update g_demo2 set name='update'
where id=?";
        const char* delete_mode = "delete from g_demo2 where id=?";
        const char* select_mode = "select id, name from g_demo2";
        int out_id[NUM_INSERT];
        char out_name[NUM_INSERT][20];

        unsigned int i = 0;
        int temp = 0;
        unsigned int num_fields = 0;

        /*初始化 GBASE 结构体*/
        if(!(gbase = gbase_init(0)))
        {
            fprintf(stderr, "不能初始化 GBASE 结构体! \n");
            exit(1);
        }
        /*数据库连接*/
        if(!gbase_real_connect(gbase, host, user, passwd, db, port,
NULL, 0))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
        }
        gbase->reconnect = 1;
        /*在数据库中创建表*/
        if(gbase_query(gbase, sql_drop_table))
        {
            fprintf(stderr, "%s", gbase_error(gbase));
            exit(1);
```

```
        }
        if(gbase_query(gbase, sql_create_table))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
        }
        /*向数据库表中插入数据*/
        for(i=0; i<NUM_INSERT; i++)
        {
            in_id[i] = i;
            sprintf(in_name[i], "name_%d", i);
        }
        if(!(stmt = gbase_stmt_init(gbase)))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }
        if(gbase_stmt_prepare(stmt, insert_mode,
strlen(insert_mode)))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }

        memset(bind, 0, sizeof(bind));

        bind[0].buffer_type = GBASE_TYPE_LONG;
        bind[1].buffer_type = GBASE_TYPE_STRING;
        bind[1].buffer_length = 20;
        printf("插入操作后表的内容\n");
        for(i=0; i<NUM_INSERT; i++)
        {
            bind[0].buffer = (void*)&in_id[i];
            bind[1].buffer = (void*)in_name[i];

            if(gbase_stmt_bind_param(stmt, bind))
            {
```

```
                fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
                exit(1);
            }
            if(gbase_stmt_execute(stmt))
            {
                fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
                exit(1);
            }
        }
        /*从数据库表中选择数据*/
        if(gbase_stmt_prepare(stmt, select_mode,
strlen(select_mode)))
        {
            fprintf(stderr, "\n%s'n", gbase_stmt_error(stmt));
            exit(1);
        }
        memset(bind, 0, sizeof(bind));

        bind[0].buffer_type = GBASE_TYPE_LONG;
        bind[1].buffer_type = GBASE_TYPE_STRING;
        bind[1].buffer_length = 20;

        if(gbase_stmt_execute(stmt))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }

        for(i=0; i<NUM_INSERT; i++)
        {
            bind[0].buffer = (void*)&out_id[i];
            bind[1].buffer = (void*)out_name[i];

            if(gbase_stmt_bind_result(stmt, bind))
            {
                fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
                exit(1);
```

```
            }
            if(!gbase_stmt_fetch(stmt))
            {
                printf("\t|%d\t|%s\t|\n", out_id[i] , out_name[i]);
            }
            else
            {
                fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
                exit(1);
            }
        }
        /*更新数据库表中数据*/
        if(gbase_stmt_prepare(stmt, update_mode,
strlen(update_mode)))
        {
            fprintf(stderr, "\n%s'n", gbase_stmt_error(stmt));
            exit(1);
        }
        memset(bind, 0, sizeof(bind));

        temp = 0;
        bind[0].buffer_type = GBASE_TYPE_LONG;
        bind[0].buffer = (void*)&temp;

        if(gbase_stmt_bind_param(stmt, bind))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }

        if(gbase_stmt_execute(stmt))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }
        printf("更新操作后表的内容\n");
        print_table(gbase, "g_demo2");
```

```
        /*删除数据库表中数据*/
        if(gbase_stmt_prepare(stmt, delete_mode,
strlen(delete_mode)))
        {
            fprintf(stderr, "\n%s'n", gbase_stmt_error(stmt));
            exit(1);
        }
        memset(bind, 0, sizeof(bind));

        temp = 1;
        bind[0].buffer_type = GBASE_TYPE_LONG;
        bind[0].buffer = (void*)&temp;

        if(gbase_stmt_bind_param(stmt, bind))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }

        if(gbase_stmt_execute(stmt))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }
        printf("删除操作后表的内容\n");
        print_table(gbase, "g_demo2");

        /*关闭 STMT 语句句柄*/
        if(gbase_stmt_close(stmt))
        {
            fprintf(stderr, "%s\n", gbase_stmt_error(stmt));
            exit(1);
        }
        /*释放数据库连接句柄*/
        gbase_close(gbase);
    }
```

5.4使用 GBase C API 执行存储过程

```
    /*执行存储过程*/
    void demo_3(void)
    {
        GBASE* gbase = NULL;
        char* sql_drop = "drop table if exists g_demo3";
        char* sql = "create table g_demo3(a varchar(10), b int)";
        char* p_drop = "DROP PROCEDURE IF EXISTS demo_p";
        char* p = "create procedure demo_p(in a varchar(10), \
                                           in b int) \
                                           begin \
                                             insert into g_demo3(a, b)
values (a, b); \
                                           end;";
        char* call_p = "CALL demo_p('call_p', 1)";

        if(!(gbase = gbase_init(0)))
        {
            fprintf(stderr, "不能初始化 GBASE 结构体! \n");
            exit(1);
        }

        if(!gbase_real_connect(gbase, host, user, passwd, db, port,
NULL, 0))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
        }

        /*在数据库中创建表*/
        if(gbase_query(gbase, sql_drop))
        {
            fprintf(stderr, "%s", gbase_error(gbase));
            exit(1);
        }
```

```
    if(gbase_query(gbase, sql))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }

    if(gbase_query(gbase, p_drop))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }

    if(gbase_query(gbase, p))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }

    if(gbase_query(gbase, call_p))
    {
        fprintf(stderr, "\n%s\n", gbase_error(gbase));
        exit(1);
    }

    printf("PROCEDURE 执行结果\n");
    print_table(gbase, "g_demo3");

    gbase_close(gbase);
}
```

5.5在多线程环境下使用 GBase C API

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "gbase.h"
```

```
#include <pthread.h>

static pthread_mutex_t lock_thread_num;
static pthread_cond_t cond_thread_num;
static int thread_num;

void* th_new(void* arg)
{
    GBASE* gbase;
    GBASE_RES *res;
    gbase_thread_init();
    gbase = gbase_init(0);
    if(!gbase_real_connect(gbase,"192.168.111.96","root","1111",
"test",5258, NULL, 0))
    {
        fprintf(stderr, "%s\n", gbase_error(gbase));
        goto exit;
    }

    exit:
    gbase_close(gbase);
    gbase_thread_end();
    pthread_mutex_lock(&lock_thread_num);
    thread_num--;
    pthread_cond_signal(&cond_thread_num);
    pthread_mutex_unlock(&lock_thread_num);
    pthread_exit(0);
    return 0;
}

int main()
{
    pthread_t t;
    unsigned int i=10;
    int error;
```

```
        pthread_attr_t thr_attr;

        if(gbase_library_init(0, NULL, NULL))
        {
            exit(1);
        }

        thread_num = i;
        if ((error=pthread_cond_init(&cond_thread_num,NULL)))
        {
            exit(1);
        }
        pthread_mutex_init(&lock_thread_num, NULL);

        if ((error=pthread_attr_init(&thr_attr)))
        {
            exit(1);
        }
        if
((error=pthread_attr_setdetachstate(&thr_attr,PTHREAD_CREATE_DETACHE
D)))
        {
            exit(1);
        }
        while(i--)
        {
            pthread_create(&t, &thr_attr, th_new, NULL);
        }
        pthread_mutex_lock(&lock_thread_num);
        while(thread_num > 0)
        {
            if
((error=pthread_cond_wait(&cond_thread_num,&lock_thread_num)))
                fprintf(stderr,"\nGot error: %d from
pthread_cond_wait\n",error);
        }
        pthread_mutex_unlock(&lock_thread_num);
```

```
        pthread_mutex_destroy(&lock_thread_num);
        gbase_library_end();
        return 0;
    }
```

5.6使用预处理快速插入数据

```
#include <windows.h>
#include <time.h>
#include <stdio.h>
#include "gbase.h"

#define  NUM_INSERT 1000

void main(void)
{
        GBASE* gbase=NULL;
        GBASE_STMT* stmt;
        GBASE_BIND bind[2];

        const char* sql_drop_table      = "drop table if exists
g_demo2";
        const char* sql_create_table    = "create table g_demo2(id int,
name varchar(20))";
        const char* insert_mode     = "insert into g_demo2(id,
name)values(?, ?)";
        int in_id[NUM_INSERT];
        char in_name[NUM_INSERT][20];

        unsigned int i = 0;
        char* host = "192.168.111.96";
        char* user = "gbase";
        char* passwd = "gbase20110531";
        char* db = "test";
        int port = 5258;
```

```
        /*初始化 GBASE 结构体*/
        if(!(gbase = gbase_init(0)))
        {
            fprintf(stderr, "不能初始化 GBASE 结构体! \n");
            exit(1);
        }
        /*数据库连接*/
        if(!gbase_real_connect(gbase, host, user, passwd, db, port,
NULL, 0))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
            }
        gbase->reconnect = 1;
        /*在数据库中创建表*/
        if(gbase_query(gbase, sql_drop_table))
        {
            fprintf(stderr, "%s", gbase_error(gbase));
            exit(1);
        }
        if(gbase_query(gbase, sql_create_table))
        {
            fprintf(stderr, "\n%s\n", gbase_error(gbase));
            exit(1);
        }
        gbase_autocommit(gbase, 0);
        /*向数据库表中插入数据*/
        for(i=0; i<NUM_INSERT; i++)
        {
            in_id[i] = i;
            sprintf(in_name[i], "name_%d", i);
        }
        if(!(stmt = gbase_stmt_init(gbase)))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
```

```
        }
        if(gbase_stmt_prepare(stmt, insert_mode,
strlen(insert_mode)))
        {
            fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
            exit(1);
        }

        memset(bind, 0, sizeof(bind));

        bind[0].buffer_type = GBASE_TYPE_LONG;
        bind[1].buffer_type = GBASE_TYPE_STRING;

        printf("开始插入%d 条数据\n", NUM_INSERT);
        for(i=0; i<NUM_INSERT; i++)
        {
            bind[0].buffer = (void*)&in_id[i];
            bind[1].buffer = (void*)in_name[i];
            bind[1].buffer_length = strlen(in_name[i]);

            if(gbase_stmt_bind_param(stmt, bind))
            {
                fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
                exit(1);
            }
            if(gbase_stmt_execute(stmt))
            {
                fprintf(stderr, "\n%s\n", gbase_stmt_error(stmt));
                exit(1);
            }
        }
        gbase_commit(gbase);

        /*关闭 STMT 语句句柄*/
        if(gbase_stmt_close(stmt))
        {
            fprintf(stderr, "%s\n", gbase_stmt_error(stmt));
```

```
            exit(1);
        }
        /*释放数据库连接句柄*/
        gbase_close(gbase);
}
```

5.7使用 GBase C API 负载均衡

使用 GBase CAPI 负载均衡创建到 8a 集群的连接时，客户端最少应该把 8a 集群一个安全组的节点 ip 地址传给 GBase CAPI。如 host="192.168.1.1;192.168.1.2"。如果 8a 集群一个安全组只有一个节点，那么客户端应该这样传给 GBase CAPI 集群的节点：host="192.168.1.1;"。字符串中 ip 地址后的分号是必须的。

使用 GBase CAPI 负载均衡创建到 8a 集群的连接时，如果没有设置 "GBASE_OPT_USE_SERVER_BALANCE"选项的值（该值默认为 0），那么 GBase CAPI 将在客户端传入的 ip 地址间进行负载均衡。如果设置 "GBASE_OPT_USE_SERVER_BALANCE"选项的值为 1，那么 GBase CAPI 将在 8a 集群所有节点之间进行负载均衡。如下是使用 GBase CAPI 负载均衡的代码样例。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "gbase.h"

int test_banalce()
{
        GBASE* gbase = NULL;
        char* host = "192.168.1.1;";
        char* user = "gbase";
        char* pwd = "gbase20110531";
        char* db = "test";
        int  port = 5258;
        int rc = 0;
```

```
        int use_server_balance = 1;

        gbase = gbase_init(NULL);
        gbase_options(gbase,  GBASE_OPT_USE_SERVER_BALANCE,
(void*)&use_server_balance);
        if(!gbase_real_connect(gbase, host, user, pwd, db, port, NULL,
0))
        {
            fprintf(stderr, "%d\n%s\n",gbase_errno(gbase),
gbase_error(gbase));
            rc = 1;
        }
        else
        {
            printf("%s\n", gbase_get_host_info(gbase));
        }
        gbase_close(gbase);
        return rc;
}

int main()
{
        int i = 0;
        for(;i < 20; i++)
        {
            test_banalce();
            usleep(2*1000);
        }
        return 0;
}
```

GBASE

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微博二维码　微信二维码

技术支持热线：400-013-9696