十年一日，深入成就深度
业精于专，专注成就专业

数字有机体工作库及大规模存储与管理系统

开发手册(V3.4.7)

成都天心悦高科技发展有限公司

2015 年 10 月

版权声明

数字有机体工作库（DosSQL）及其客户端驱动的版权属于成都天心悦高科技发展有限公司所有。任何组织和个人未经成都天心悦高科技发展有限公司许可与授权，不得擅自使用、复制、更改该软件的产品。

本软件受版权法和国际条约的保护。如未经授权而擅自使用、复制或传播本软件程序（或其中任何部分），将受到严厉的刑事及民事制裁，并将在法律许可的范围内受到最大可能的起诉！

成都天心悦高科技发展有限公司

地址：成都市武侯区棕南小区

电话：028-83318559

邮编：610054

目 录

1 前言

1.1 编写约定

非常感谢您使用成都天心悦高科技发展有限公司的产品，本公司将竭诚为您提供最好的服务。

本手册假定用户能够理解并使用 Linux 的基本 shell 命令。

文中出现的‘#’号表示数字有机体系统的命令行提示符。

文中没有特特说明是批量同步工作库时，默认表示实时同步工作库。

命令格式描述中的斜体字表示应由用户填充的部分，”[ ]”表示命令中可选的命令参数。

为了阅读方便，文档以灰底黑框的形式呈现某些重要的配置操作。不过，由于数字有机体系统和 Windows 采用不同的字符集和文本规范，请不要直接拷贝文档中的命令行或者配置行到数字有机体系统中，请重新输入。

因软件更新，本手册可能包含技术上不准确的地方或文字错误。

本手册的内容将做定期的更新，恕不另行通知；更新的内容将会在本手册的新版本中加入。

本公司随时会改进或更新本手册中描述的产品或程序。

1.2 如何使用本手册

本手册的阅读对象为数字有机体工作库的应用软件开发人员，建议首先阅读第二章了解本文的内容和开发注意事项，然后根据自己使用的需要选择章节进行阅读。若遇到问题请阅读第 5 章了解新增错误类型，如何解决请查阅第六章。

如果需要查找某个接口函数或者数据结构，可以直接通过目录查找。

1.3 相关文档说明

数字有机体系统包括数字有机体工作平台和数字有机体工作库，本文档是数字有机体工作库的开发手册，相关的其他文档还有：

- 有关数字有机体工作平台的使用请参阅《数字有机体工作平台及抗毁容灾系统用户手册》。
- 有关如何在数字有机体工作平台上开发应用程序，请参考《数字有机体工作平台及抗毁容灾系统开发手册》。
- 有关如何安装、管理、维护和使用数字有机体工作库请参考《数字有机体工作库及大规模存储与管理系统用户手册》。

1.4 术语

数字有机体工作库及大规模存储与管理系统: 我们有时将数字有机体工作库及大规模存储与管理系统简称为数字有机体工作库或工作库，英文缩写为 DosSQL。这种工作库含盖常规数据库系统但远高于常规数据库系统，是一个在 Mysql 之上的、面向很多应用的、统一的、人能化的应用数据平台。

1.5 如何获得技术支持

在您遇到问题时，请首先联系您的产品提供商。大多数问题都可以在产品提供商的技术支持人员的帮助下得以解决。

您也可以通过产品提供商致电本公司的技术服务热线：028-83318559，获得电话技术支持。您还可以发送邮件，邮件地址是：tianxinyue@126.com。如果您确实需要本公司提供上门服务，本公司将竭诚为您服务。

2 简介

2.1 数字有机体系统简介

数字有机体系统（英文名称为 Digital Oganism System, 缩写为 DOS）是在刘心松教授带领下，由成都天心悦高科技发展有限公司的研发人员前后千余人次，经过三十多年的技术积累，研发成功的基础系统。

研发这种系统的原始宗旨是向生物特别是人类个体和群体的结构、机理和特性逼近，是一种人能化的新的系统模式。这种系统集操作系统、数据库系统、大规模存储、抗毁容灾、高伸缩、高智能、高灵活、自搜索、自传播、自复制、自修复、自重构、自适应、系统间的兼容性、群体间的协作性、对资源的动态管理调度合理配置、大小新旧机器混合使用等特性为一体，是一个整体解决方案，是面向所有应用的统一的（应用）系统平台。

数字有机体系统主要由数字有机体工作平台及抗毁容灾系统、数字有机体工作库及大规模存储与管理系统和数字有机体安全系统组成。这是从底层作起的一个一体化平台，可以在此平台上开发任何应用，形成任何应用系统。例如现在已有的应用系统就有数字有机体流媒体系统、数字有机体监控系统、数字有机体会议系统、数字有机体网关、数字有机体管理系统、数字有机体控申系统、数字有机体侦查指挥系统等。

本文有时将数字有机体工作平台及抗毁容灾系统，数字有机体工作库及大规模存储与管理系统和数字有机体安全系统统称为数字有机体系统。数字有机体工作平台及抗毁容灾系统含盖常规操作系统但远高于常规操作系统，是一个在 Linux 之上的、面向很多应用的、统一的、人能化的应用系统平台。数字有机体工作库及大规模存储与管理系统含盖常规数据库系统但远高于常规数据库系统，是一个在 Mysql 之上的、面向很多应用的、统一的、人能化的应用数据平台。

有时将数字有机体工作平台及抗毁容灾系统简称为数字有机体工作平台甚至工作平台。

有时将数字有机体工作库及大规模存储与管理系统简称为数字有机体工作库甚至工作库。

2.2 数字有机体工作库简介

数字有机体工作库由接口子系统、通信子系统、执行子系统和管理子系统四个部分构成，各子系统相互协作实现相关功能。可以由大量的服务器通过高速网络连接，构成一个超大型数字有机体工作库。由于系统具有高存储容量、高动态伸缩性、高可用性等特点，它可以为电子政务、视频点播、校园网等对数据库可靠性要求高、存储容量大的多种应用场合提供数据（管理）平台。它具有如下主要的功能和特性：

1) 海量存储

系统支持很多库，单库支持至少 TB 级数据存储。系统存储容量可根据用户数据的存

储需求动态扩展，能够支持大量视频、音频等多媒体数据，同时能够存储多种文本数据以及二进制格式的软件，能够满足多种应用需求。

2) 分布性

系统的分布性主要体现在数据分布性、执行分布性以及管理分布性。数据分布性指系统的所有用户数据根据可靠性以及执行效率的要求分布在系统中的各个节点，它们之间没有主次之分，是完全对等的关系。执行分布性乃指用户的每一条库操作指令能够被系统自动解析，选择最佳的服务器节点完成，缩短响应时间，比单机数据库系统的操作效率有显著的提高，特别是复杂操作，效率可以成倍的提高。管理分布性指整个系统的维护和管理分布在所有的节点上，不存在核心节点或中心节点，也没有主次之分，这样既保证了高可靠性，又避免了系统瓶颈。

3) 并行性

系统的并行性主要体现在操作并行性和管理并行性。操作并行性指用户的不同操作在不同节点并行处理，同时根据需要用户的同一操作也可在不同的节点上并行处理，这样既提高了操作的效率，又加大了系统的吞吐量。管理并行性指对整个系统的维护在不同的节点上并行进行，通过特定的协议协调和维护它们之间的一致性，这样既提高了系统管理的效率，同时又保证了高可靠性。这是集中式系统或单机系统以及 Cluster 系统无法做到的，是数字有机体工作库区别于其它数据库系统的显著特征之一。

4) 动态伸缩性

动态伸缩性指系统的规模、服务能力能够根据需要进行动态配置，而不是象单机系统那样，服务能力从一开始就定死并受限于服务器的性能指标。系统能够动态地增加、减少服务器节点的数量，根据需要动态地改变服务能力，可以由大量服务器组成的系统提供服务，同时也可动态地更换或升级系统中的服务器，而不影响整个系统的正常使用。因此可以根据业务量的增长需求分期分批投资和建设。

5) 高可靠性及高可用性

系统根据用户对工作库的使用频率、可靠性要求以及配置情况，动态决定工作库的分布以及冗余度，从而保证用户数据的可用性。同时在一定环境下系统能够自动进行重构，这样在发生节点故障时，系统通过自动重构用户的数据分布和分离故障节点就能保证用户数据的高可靠性及高可用性。

6) 高安全性

系统除具有一般数据库系统的安全特性外还提供远程数据同步功能，能够通过广域网络将系统中的数据从本地导出到远地存储，也可以方便地将数据从远地导入，因此即使发生灾难性事故也能保证数据的安全。数据通信支持 SSL 安全通信，并采用证书的机制防止主机的伪冒，具备主机识别的功能等。

7) 使用方便性

系统提供多种接口，方便用户的二次开发。系统支持多种开发工具，如 ODBC、JDBC 等，支持 C/C＋＋、Perl、JAVA、PHP 等高级程序语言，能够开发动态网页等各种数据库应用程序。

8) 硬件通用性

系统使用的各种硬件设备均为通用设备，构建系统方便灵活，维护简单。

2.3 接口概述

为了兼容现有的大多数 MySQL 应用程序，并缩短编程人员学习新系统的时间，数字有机体工作库的接口通过 MySQL 的原有接口提供。为了兼容现有的大多数 Linux 程序，并缩短编程人员学习新系统的时间，数字有机体工作库的接口通过 MySQL 的原有接口提供。不过，根据数字有机体工作库的需要，扩充了 SQL 指令集，并修改了部分底层功能。因此，不要采用原有的 MySQL 接口来访问数字有机体工作库，而应当采用数字有机体工作库提供的接口库。

同样，数字有机体工作库和 MySQL 在体系结构上是完全不同的。因此，少量只适用于单机环境的 SQL 指令在数字有机体工作库上也不再有意义或者不可能在存在了。请查阅本手册中“不支持特性章节”。因此，少量只适用于单机环境的 SQL 指令在数字有机体工作库上也不再有意义或者不可能再存在了。请查阅《数字有机体用户手册》中“不支持特性章节”。

DosSQL 提供 C API、C++ API、JDBC（JAVA API）、Perl API、HPH API、Python API、Tcl API。本手册向用户提供数字有机体工作库（DosSQL 当前版本 3.1.16）的开发接口使用说明，将重点介绍常用的 C API、JDBC 和 Perl API。

基于数字有机体工作库开发大数据量、大并发、高性能的应用时，推荐采用分表、分库、读写连接分离等技术，这将大幅度提升应用的性能。

开发接口因计算机系统平台（如 Windows 和 Linux）和开发语言而有所差异，DosSQL 支持多种系统平台和语言的应用开发，将介绍的内容如下。

开发库名称	组件名称	组件类型	系统平台	编程语言
CAPI	libmysqlclient.so（或.dll） libmysqlclient_r.so（或.dll） libmysqlclient.a（或.lib） libmysqlclient_r.a（或.lib） mysql.h 等文件	开发库文件和头文件	Windows/Linux	c/c++
JDBC	dossql-connector-java-3.1.0.jar	库文件	Windows/Linux	Java
PERL	Perl DBI Perl DBD C API	DBI 是数据库的通用接口，DBD 是数据库驱动程序	Linux	perl
示例	位于“/usr/local/dossql/demo/”目录	示例程序	linux	c/c++

3 API 和库

本章将介绍 DosSQL 可使用的 API，以及如何使用它们。将详细介绍 C API，这是因为它是开发人员经常用到的，而且它也是大多数其他 API 的基础。

3.1 C API

3.1.1 C API 简介

C API 是数字有机体工作库提供的以 C 库形式存在的基础接口库，是许多其他 API 接口的基础，也是 C/C++语言常用的编程接口，它包含在 mysqlclient 库中。

大多数其他客户端 API（除 Connector/J 和 Connector/NET 外）采用 mysqlclient 库来与数字有机体工作库进行通信。这意味着你可以使用很多相同环境变量带来的好处，这是因为它们是从库中引用的。

客户端具有最大的通信缓冲区大小，初始分配的缓冲区大小（16KB）将自动增加到最大（最大为 16MB）。由于缓冲区大小将按需增加，简单地增加到默认的最大限制，从其本身来说不会增加资源使用。该限制检查主要是检查错误查询和通信信息包。

通信缓冲区必须足够大，足以包含 1 条 SQL 语句（用于客户端-服务器通信）以及 1 行返回的数据（用于服务器-客户端通信）。每个线程的通信缓冲区将动态增加，以处理直至最大限制的任何查询或行。例如，如果 BLOB 值包含高达 16MB 的数据，那么通信缓冲区的大小限制至少为 16MB（在服务器和客户端）。客户端的默认最大值为 16MB，服务器的默认最大值也为 16MB。可以在启动服务器时通过更改 max_allowed_packet 参数的值增加它。

每次查询后，数字有机体工作库服务器会将通信缓冲区的大小降至 net_buffer_length 字节。对于客户端，不会降低与连接相关缓冲区大小，直至连接关闭为止。此时，客户端内存将被收回。

3.1.2 C API 数据类型

- MYSQL

该结构代表 1 个数据库连接的句柄，几乎所有的 MySQL 函数均使用它。不应尝试拷贝 MYSQL 结构，不保证这类拷贝结果会有用。

- MYSQL_RES

该结构代表返回行的查询结果（SELECT, SHOW, DESCRIBE, EXPLAIN），在本节的剩余部分，将查询返回的信息称为“结果集”。

- MYSQL_ROW

这是 1 行数据的“类型安全”表示。它目前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）。行是通过调用 mysql_fetch_row()获得的。

- MYSQL_FIELD

该结构包含关于字段的信息，如字段名、类型和大小，这里详细介绍了其成员。通过重复调用 mysql_fetch_field()，可为每个字段获得 MYSQL_FIELD 结构。字段值不是该结构的组成部份，它们包含在 MYSQL_ROW 结构中。

- MYSQL_FIELD_OFFSET

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

- my_ulonglong

用于行数以及 mysql_affected_rows()、mysql_num_rows()和 mysql_insert_id()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 my_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。例如

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

下面列出了 MYSQL_FIELD 结构包含的成员：

· char * name

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。

· char * org_name

段名称，由 Null 终结的字符串，忽略别名。

· char * table

如果该字段不是计算出的字段的话，它包含该字段的表的名称。对于计算出的字段，它是空字符串。如果用 AS 子句为该表指定了别名，它的值是别名。

· char * org_table

表的名称，由 Null 终结的字符串，忽略别名。

· char * db

字段源自数据库的名称，由 Null 终结的字符串。如果该字段是计算出的字段，则 db 即为空的字符串。

· char * catalog

catalog 名称，该值总是"def"。

· char * def

该字段的默认值，由 Null 终结的字符串。仅当使用 mysql_list_fields()时才设置它。

· unsigned long length

字段的宽度。

· unsigned long max_length

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 mysql_store_result()或 mysql_list_fields()，它将包含字段的最大长度。如果使用 mysql_use_result()，该变量的值为 0。

· unsigned int name_length

名称的长度。

· unsigned int org_name_length

org_name 的长度。

· unsigned int table_length

表名的长度。

· unsigned int org_table_length

org_table 的长度。

· unsigned int db_length

db 的长度。

· unsigned int catalog_length

catalog 的长度。

· unsigned int def_length

def 的长度。

· unsigned int flags

用于字段的不同“位标志”，标志的值可以有 0 个或多个下述位的集合：

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）

不再重视 BLOB_FLAG、ENUM_FLAG、SET_FLAG 和 TIMESTAMP_FLAG 标志，原因在于，它们指出了字段的类型，而不是类型的属性。更可取的方式是使用 MYSQL_TYPE_BLOB、MYSQL_TYPE_ENUM、MYSQL_TYPE_SET 或 MYSQL_TYPE_TIMESTAMP 测试 field->type。

在下面的示例中，介绍了标志值的典型用法：

```
if (field->flags & NOT_NULL_FLAG)
        printf("Field can't be null\n");
```

可以使用下表的宏来测试标志值的布尔状态：

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）。

· unsigned int decimals

用于数值字段的十进制数数目。

· unsigned int charset_nr

用于字段的字符集编号。

· enum enum_field_types type

字段的类型，类型值可以是下表所列的 MYSQL_TYPE_符号之一：

类型值	类型描述
MYSQL_TYPE_TINY	TINYINT 字段
MYSQL_TYPE_SHORT	SMALLINT 字段
MYSQL_TYPE_LONG	INTEGER 字段
MYSQL_TYPE_INT24	MEDIUMINT 字段
MYSQL_TYPE_LONGLONG	BIGINT 字段
MYSQL_TYPE_DECIMAL	DECIMAL 或 NUMERIC 字段
MYSQL_TYPE_NEWDECIMAL	精度数学 DECIMAL 或 NUMERIC
MYSQL_TYPE_FLOAT	FLOAT 字段
MYSQL_TYPE_DOUBLE	DOUBLE 或 REAL 字段
MYSQL_TYPE_BIT	BIT 字段
MYSQL_TYPE_TIMESTAMP	TIMESTAMP 字段
MYSQL_TYPE_DATE	DATE 字段
MYSQL_TYPE_TIME	TIME 字段
MYSQL_TYPE_DATETIME	DATETIME 字段
MYSQL_TYPE_YEAR	YEAR 字段
MYSQL_TYPE_STRING	CHAR 字段
MYSQL_TYPE_VAR_STRING	VARCHAR 字段
MYSQL_TYPE_BLOB	BLOB 或 TEXT 字段（使用 max_length 来确定最大长度）
MYSQL_TYPE_SET	SET 字段
MYSQL_TYPE_ENUM	ENUM 字段
MYSQL_TYPE_GEOMETRY	Spatial 字段
MYSQL_TYPE_NULL	NULL-type 字段
MYSQL_TYPE_CHAR	不再重视，用 MYSQL_TYPE_TINY 取代

可以使用 IS_NUM()宏来测试字段是否具有数值类型。将类型值传递给 IS_NUM()，如果字段为数值类型，会将其评估为“真”：

```
if (IS_NUM(field->type))
        printf("Field is numeric\n");
```

3.1.3 C API 函数概述

这里归纳了 C API 可使用的函数，并在下一节详细介绍它们。

函数	描述
mysql_affected_rows()	返回上次 UPDATE、DELETE 或 INSERT 查询更改 / 删除 / 插入的行数。
mysql_autocommit()	切换 autocommit 模式，ON/OFF
mysql_change_user()	更改打开连接上的用户和数据库。
mysql_charset_name()	返回用于连接的默认字符集的名称。
mysql_close()	关闭服务器连接。
mysql_commit()	提交事务。
mysql_data_seek()	在查询结果集中查找属性行编号。
mysql_debug()	用给定的字符串执行 DBUG_PUSH。
mysql_dump_debug_info()	让服务器将调试信息写入日志。
mysql_errno()	返回上次调用的 MySQL 函数的错误编号。
mysql_error()	返回上次调用的 MySQL 函数的错误消息。
mysql_fetch_field()	返回下一个表字段的类型。
mysql_fetch_field_direct()	给定字段编号，返回表字段的类型。
mysql_fetch_fields()	返回所有字段结构的数组。
mysql_fetch_lengths()	返回当前行中所有列的长度。
mysql_fetch_row()	从结果集中获取下一行
mysql_field_seek()	将列光标置于指定的列。
mysql_field_count()	返回上次执行语句的结果列的数目。
mysql_field_tell()	返回上次 mysql_fetch_field()所使用字段光标的位置。
mysql_free_result()	释放结果集使用的内存。
mysql_get_host_info()	返回描述连接的字符串。
mysql_get_server_version()	以整数形式返回服务器的版本号。
mysql_get_proto_info()	返回连接所使用的协议版本。
mysql_get_server_info()	返回服务器的版本号。
mysql_info()	返回关于最近所执行查询的信息。
mysql_init()	获取或初始化 MYSQL 结构。
mysql_insert_id()	返回上一个查询为 AUTO_INCREMENT 列生成的 ID。
mysql_kill()	杀死给定的线程。

mysql_library_end()	最终确定 MySQL C API 库。
mysql_library_init()	初始化 MySQL C API 库。
mysql_list_dbs()	返回与简单正则表达式匹配的数据库名称。
mysql_list_fields()	返回与简单正则表达式匹配的字段名称。
mysql_list_processes()	返回当前服务器线程的列表。
mysql_list_tables()	返回与简单正则表达式匹配的表名。
mysql_more_results()	检查是否还存在其他结果。
mysql_next_result()	在多语句执行过程中返回/初始化下一个结果。
mysql_num_fields()	返回结果集中的列数。
mysql_num_rows()	返回结果集中的行数。
mysql_options()	为 mysql_connect()设置连接选项。
mysql_ping()	检查与服务器的连接是否工作，如有必要重新连接。
mysql_query()	执行指定为“以 Null 终结的字符串”的 SQL 查询。
mysql_real_connect()	连接到 MySQL 服务器。
mysql_real_escape_string()	考虑到连接的当前字符集，为了在 SQL 语句中使用，对字符串中的特殊字符进行转义处理。
mysql_real_query()	执行指定为计数字符串的 SQL 查询。
mysql_refresh()	刷新或复位表和高速缓冲。
mysql_reload()	通知服务器再次加载授权表。
mysql_rollback()	回滚事务。
mysql_row_seek()	使用从 mysql_row_tell()返回的值，查找结果集中的行偏移。
mysql_row_tell()	返回行光标位置。
mysql_select_db()	选择数据库。
mysql_server_end()	最终确定嵌入式服务器库。
mysql_server_init()	初始化嵌入式服务器库。
mysql_set_server_option()	为连接设置选项（如多语句）。
mysql_sqlstate()	返回关于上一个错误的 SQLSTATE 错误代码。
mysql_shutdown()	关闭数据库服务器。
mysql_stat()	以字符串形式返回服务器状态。
mysql_store_result()	检索完整的结果集至客户端。
mysql_thread_id()	返回当前线程 ID。
mysql_thread_safe()	如果客户端已编译为线程安全的，返回 1。
mysql_use_result()	初始化逐行的结果集检索。
mysql_warning_count()	返回上一个 SQL 语句的告警数。

与数字有机体工作库交互时，应用程序应使用以下一般性原则：

1. 通过调用 mysql_library_init()初始化接口库。

2. 通过调用 mysql_init()初始化连接处理句柄，并通过调用 mysql_real_connect()连接到服务器。

3. 发出 SQL 语句并处理其结果。（在下面的讨论中，详细介绍了使用它的方法）。

4. 通过调用 mysql_close()，关闭与 DosSQL 服务器的连接。

5. 通过调用 mysql_library_end()，结束接口库的使用。

调用 mysql_library_init()和 mysql_library_end()的目的在于，为接口库提供恰当的初始化和结束处理。对于与客户端库链接的应用程序，它们提供了改进的内存管理功能。如果不调用 mysql_library_end()，内存块仍将保持分配状态（这不会增加应用程序使用的内存量，但某些内存泄漏检测器将抗议它）。对于与嵌入式服务器链接的应用程序，这些调用会启动并停止服务器。

如果愿意，可省略对 mysql_library_init()的调用，这是因为必要时 mysql_init()会自动调用它。

要想连接到服务器，可调用 mysql_init()来初始化连接处理句柄，然后用该处理句柄（以及其他信息，如主机名、用户名和密码）调用 mysql_real_connect()。建立连接后，mysql_real_connect()会将再连接标志（MYSQL 结构的一部分）设置为 1。对于该标志，值“1”指明，如果因连接丢失而无法执行语句，放弃之前，会尝试再次连接到服务器。完成连接后，退出程序前必须调用 mysql_close()中止它。

当连接处于活动状态时，客户端或许会使用 mysql_query()或 mysql_real_query()向服务器发出 SQL 查询。两者的差别在于，mysql_query()预期的查询为指定的、由 Null 终结的字符串，而 mysql_real_query()预期的是计数字符串。如果字符串包含二进制数据（其中可能包含 Null 字节），就必须使用 mysql_real_query()。

对于每个非 SELECT 查询（例如 INSERT、UPDATE、DELETE），通过调用 mysql_affected_rows()，可发现有多少行已被改变（影响）。

对于 SELECT 查询，能够检索作为结果集的行。注意，某些语句因其返回行类似于 SELECT，包括 SHOW、DESCRIBE 和 EXPLAIN，应按照对待 SELECT 语句的方式处理它们。

客户端处理结果集的方式有两种。一种方式是，通过调用 mysql_store_result()，一次性地检索整个结果集。该函数能从服务器获得查询返回的所有行，并将它们保存在客户端。第二种方式是针对客户端的，通过调用 mysql_use_result()，对“按行”结果集检索进行初始化处理。该函数能初始化检索结果，但不能从服务器获得任何实际行。

在这两种情况下，均能通过调用 mysql_fetch_row()访问行。通过 mysql_store_result()，mysql_fetch_row()能够访问以前从服务器获得的行。通过 mysql_use_result()，mysql_fetch_row()能够实际地检索来自服务器的行。通过调用 mysql_fetch_lengths()，能获得关于各行中数据大小的信息。

完成结果集操作后，请调用 mysql_free_result()释放结果集使用的内存。

这两种检索机制是互补的。客户端程序应选择最能满足其要求的方法。实际上，客户端最常使用的是 mysql_store_result()。

mysql_store_result()的 1 个优点在于，由于将行全部提取到了客户端上，你不仅能连续访问行，还能使用 mysql_data_seek()或 mysql_row_seek()在结果集中向前或向后移动，以更改结果集内当前行的位置。通过调用 mysql_num_rows()，还能发现有多少行。但另一方面，对于大的结果集，mysql_store_result()所需的内存可能会很大，你很可能遇到内存溢出状况。

mysql_use_result()的 1 个优点在于，客户端所需的用于结果集的内存较少。原因在于，一次它仅维护一行（由于分配开销较低，mysql_use_result()能更快）。它的缺点在于，你必须快速处理每一行以避免妨碍服务器，你不能随机访问结果集中的行（只能连续访问行），你不知道结果集中有多少行，直至全部检索了它们为止。不仅如此，即使在检索过程中你判定已找到所寻找的信息，也必须检索所有的行。

通过 API，客户端能够恰当地对查询作出响应（仅在必要时检索行），而无需知道查询是否是 SELECT 查询。可以在每次 mysql_query()或 mysql_real_query()后，通过调用 mysql_store_result()完成该操作。如果结果集调用成功，查询为 SELECT，而且能够读取行。如果结果集调用失败，可调用 mysql_field_count()来判断结果是否的确是所预期的。如果 mysql_field_count()返回 0，查询不返回数据（表明它是 INSERT、UPDATE、DELETE 等），而且不返回行。如果 mysql_field_count()是非 0 值，查询应返回行，但没有返回行。这表明查询是失败了的 SELECT。关于如何实现该操作的示例，请参见关于 mysql_field_count()的介绍。

无论是 mysql_store_result()还是 mysql_use_result()，均允许你获取关于构成结果集的字段的信息（字段数目，它们的名称和类型等）。通过重复调用 mysql_fetch_field()，可以按顺序访问行内的字段信息；或者通过调用 mysql_fetch_field_direct()，能够在行内按字段编号访问字段信息。通过调用 mysql_field_seek()，可以改变当前字段的光标位置。对字段光标的设置将影响后续的 mysql_fetch_field()调用。此外，你也能通过调用 mysql_fetch_fields()，一次性地获得关于字段的所有信息。

为了检测和通报错误，MySQL 提供了使用 mysql_errno()和 mysql_error()函数访问错误信息。它们能返回关于最近调用的函数的错误代码或错误消息。最近调用的函数可能成功也可能失败。这样，你就能判断错误是在何时出现的，以及错误是什么。

3.1.4 C API 函数描述

在本节所作的介绍中，按照 C 编程语言，为 NULL 的参数或返回值表示 NULL，而不是 MySQL Null 值。

返回值的函数通常会返回指针或整数, 除非作了其他规定。 返回指针的函数将返回非 NULL 值，以指明成功，或返回 NULL 值以指明出错。返回整数的函数将返回 0 以指明成功，或返回非 0 值以指明出错。注意，非 0 值仅表明这点, 除非在函数描述中作了其他说明. 不要对非 0 值进行测试。正确的方式是

```
if (result) /* correct */
  ... error ...
```

以下两种方式都是错误的。

```
if (result < 0)    /* incorrect */
  ... error ...

if (result == -1)    /* incorrect */
  ... error ...
```

当函数返回错误时，在函数描述的“错误”部分将列出可能的错误类型。通过调用 mysql_errno()可发现出现的错误是什么。通过调用 mysql_error()，可获得错误的字符串表示。

3.1.4.1 mysql_affected_rows()

my_ulonglong mysql_affected_rows(MYSQL *mysql)

描述

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 mysql_query()后立刻调用。对于 SELECT 语句，mysql_affected_rows()的工作方式与 mysql_num_rows()类似。

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 mysql_store_result()之前调用了 mysql_affected_rows()。由于该函数返回无符号值，可通过比较返回值和“(my_ulonglong)-1”检查是否为“-1”。

错误

无。

示例：

```
mysql_query(&mysql,"UPDATE products SET cost=cost*1.25 WHERE group=10");
printf("%ld products updated",(long) mysql_affected_rows(&mysql));
```

如果在连接至 mysqld 时指定了标志 CLIENT_FOUND_ROWS，对于 UPDATE 语句，mysql_affected_rows()将返回 WHERE 语句匹配的行数。

注意，使用 REPLACE 命令时，如果新行替代了旧行，mysql_affected_rows()返回 2。这是因为，在该情况下删除了重复行后插入了 1 行。

如果使用“INSERT ... ON DUPLICATE KEY UPDATE”来插入行，如果行是作为新行插入的，mysql_affected_rows()返回 1，如果是更新了已有的行，返回 2。

3.1.4.2 mysql_autocommit()

my_bool mysql_autocommit(MYSQL *mysql, my_bool mode)

描述

如果模式为“1”，启用 autocommit 模式；如果模式为“0”，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。

错误

无。

3.1.4.3 mysql_change_user()

my_bool mysql_change_user(MYSQL *mysql, const char *user, const char *password, const char *db)

描述

更改用户，并使由 db 指定的数据库成为由 mysql 指定的连接上的默认数据库（当前数据库）。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，mysql_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。

返回值

0 表示成功，非 0 值表示出现错误。

错误

与从 mysql_real_connect()获得的相同。

• CR_COMMANDS_OUT_OF_SYNC

以不恰当的顺序执行了命令。

• CR_SERVER_GONE_ERROR

MySQL 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

MySQL 服务器未实施该命令（或许是较低版本的服务器）。

• 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。

示例：

```
if (mysql_change_user(&mysql, "user", "password", "new_database"))
{
    fprintf(stderr, "Failed to change user.    Error: %s\n",
    mysql_error(&mysql));
}
```

3.1.4.4 mysql_character_set_name()

const char *mysql_character_set_name(MYSQL *mysql)

描述

为当前连接返回默认的字符集。

返回值

默认字符集。

错误

无。

3.1.4.5 mysql_close()

void mysql_close(MYSQL *mysql)

描述

关闭前面打开的连接。如果句柄是由 mysql_init()或 mysql_connect()自动分配的，mysql_close()还将释放由 mysql 指向的连接句柄。

返回值

无。

错误

无。

3.1.4.6 mysql_commit()

my_bool mysql_commit(MYSQL *mysql)

描述

提交当前事务。

该函数的动作受 completion_type 系统变量的值控制。尤其是，如果 completion_type 的值为 2，终结事务并关闭客户端连接后，服务器将执行释放操作。客户端程序应调用 mysql_close()，从客户端一侧关闭连接。

返回值

如果成功，返回 0，如果出现错误，返回非 0 值。

错误

无。

3.1.4.7 mysql_data_seek()

void mysql_data_seek(MYSQL_RES *result, my_ulonglong offset)

描述

在查询结果集中寻找任意行。偏移值为行号，范围从 0 到 mysql_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，so mysql_data_seek()仅应与 mysql_store_result()联合使用，而不是与 mysql_use_result()联合使用。

返回值

无。

错误

无。

3.1.4.8 mysql_debug()

void mysql_debug(const char *debug)

描述

用给定的字符串执行 DBUG_PUSH。mysql_debug()采用 Fred Fish 调试库。要想使用该函数，必须编译客户端库，使之支持调试功能。

返回值

无。

错误

无。

示例：

这里给出的调用将使客户端库在客户端机器的/tmp/client.trace 中生成 1 个跟踪文件。

```
mysql_debug("d:t:O,/tmp/client.trace");
```

3.1.4.9 mysql_dump_debug_info()

int mysql_dump_debug_info(MYSQL *mysql)

描述

指示服务器将一些调试信息写入日志。要想使之工作，已连接的用户必须具有 SUPER 权限。

返回值

如果命令成功，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

MySQL 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。

3.1.4.10 mysql_errno()

unsigned int mysql_errno(MYSQL *mysql)

描述

对于由 mysql 指定的连接，mysql_errno()返回最近调用的 API 函数的错误代码，该函数调用可能成功也可能失败。“0”返回值表示未出现错误。在 MySQL errmsg.h 头文件中，列出了客户端错误消息编号。

注意，如果成功，某些函数，如 mysql_fetch_row()等，不会设置 mysql_errno()。

经验规则是，如果成功，所有向服务器请求信息的函数均会复位 mysql_errno()。

返回值

如果失败，返回上次 mysql_xxx()调用的错误代码。“0”表示未出现错误。

错误

无。

3.1.4.11 mysql_error()

const char *mysql_error(MYSQL *mysql)

描述

对于由 mysql 指定的连接，对于失败的最近调用的 API 函数，mysql_error()返回包含错误消息的、由 Null 终结的字符串。如果该函数未失败，mysql_error()的返回值可能是以前的错误，或指明无错误的空字符串。

经验规则是，如果成功，所有向服务器请求信息的函数均会复位 mysql_error()。

对于复位 mysql_errno()的函数，下述两个测试是等效的：

```
if(mysql_errno(&mysql))
{
  // an error occurred
}

if(mysql_error(&mysql)[0] != '\0')
{
  // an error occurred
}
```

通过重新编译 MySQL 客户端库，可以更改客户端错误消息的语言。

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。

错误

无。

3.1.4.12 mysql_fetch_field()

MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)

描述

返回采用 MYSQL_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，mysql_fetch_field()返回 NULL。

每次执行新的 SELECT 查询时，将复位 mysql_fetch_field()，以返回关于第 1 个字段的信息。调用 mysql_field_seek()也会影响 mysql_fetch_field()返回的字段。

如果调用了 mysql_query()以在表上执行 SELECT，但未调用 mysql_store_result()，如果调用了 mysql_fetch_field()以请求 BLOB 字段的长度，MySQL 将返回默认的 Blob 长度（8KB）。之所以选择 8KB 是因为 MySQL 不知道 BLOB 的最大长度。应在日后使其成为可配置的。一旦检索了结果集，field->max_length 将包含特定查询中该列的最大值的长度。

返回值

当前列的 MYSQL_FIELD 结构。如果未剩余任何列，返回 NULL。

错误

无。

示例：

```
MYSQL_FIELD *field;

while((field = mysql_fetch_field(result)))
{
    printf("field name %s\n", field->name);
}
```

3.1.4.13 mysql_fetch_field_direct()

MYSQL_FIELD *mysql_fetch_field_direct(MYSQL_RES *result, unsigned int fieldnr)

描述

给定结果集内某 1 列的字段编号 fieldnr，以 MYSQL_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。Fieldnr 的值应在从 0 到 mysql_num_fields(result)-1 的范围内。

返回值

对于指定列，返回 MYSQL_FIELD 结构。

错误

无。

示例：

```
unsigned int num_fields;
unsigned int i;
MYSQL_FIELD *field;

num_fields = mysql_num_fields(result);
for(i = 0; i < num_fields; i++)
{
    field = mysql_fetch_field_direct(result, i);
    printf("Field %u is %s\n", i, field->name);
}
```

3.1.4.14 mysql_fetch_fields()

MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES *result)

描述

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

返回值

关于结果集所有列的 MYSQL_FIELD 结构的数组。

错误

无。

示例：

```
unsigned int num_fields;
unsigned int i;
MYSQL_FIELD *fields;
num_fields = mysql_num_fields(result);
fields = mysql_fetch_fields(result);
for(i = 0; i < num_fields; i++)
{
    printf("Field %u is %s\n", i, fields[i].name);
}
```

3.1.4.15 mysql_fetch_lengths()

unsigned long *mysql_fetch_lengths(MYSQL_RES *result)

描述

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。

对于空列以及包含 NULL 值的列，其长度为 0。要想了解区分这两类情况的方法，请参见关于 mysql_fetch_row()的介绍。

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。

错误

mysql_fetch_lengths()仅对结果集的当前行有效。如果在调用 mysql_fetch_row()之前或检索了结果集中的所有行后调用了它，将返回 NULL。

示例：

```
MYSQL_ROW row;
unsigned long *lengths;
unsigned int num_fields;
unsigned int i;

row = mysql_fetch_row(result);
if (row)
{
    num_fields = mysql_num_fields(result);
    lengths = mysql_fetch_lengths(result);
    for(i = 0; i < num_fields; i++)
    {
        printf("Column %u is %lu bytes in length.\n", i, lengths[i]);
    }
}
```

3.1.4.16 mysql_fetch_row()

MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)

描述

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

行内值的数目由 mysql_num_fields(result)给出。如果行中保存了调用 mysql_fetch_row()返回的值，将按照 row[0]到 row[mysql_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 mysql_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。

返回值

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

错误

注意，在对 mysql_fetch_row()的两次调用之间，不会复位错误。

- 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。

示例：

```
MYSQL_ROW row;
unsigned int num_fields;
unsigned int i;

num_fields = mysql_num_fields(result);
while ((row = mysql_fetch_row(result)))
{
    unsigned long *lengths;
    lengths = mysql_fetch_lengths(result);
    for(i = 0; i < num_fields; i++)
    {
        printf("[%.*s] ", (int) lengths[i], row[i] ? row[i] : "NULL");
    }
    printf("\n");
}
```

3.1.4.17 mysql_field_count()

unsigned int mysql_field_count(MYSQL *mysql)

描述

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 mysql_store_result()返回 NULL（因而没有结果集指针）时。在这种情况下，可调用 mysql_field_count()来判定 mysql_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。在这里给出的示例中，演示了完成它的方法。

返回值

表示结果集中列数的无符号整数。

错误

无。

示例：

```
MYSQL_RES *result;
unsigned int num_fields;
unsigned int num_rows;

if (mysql_query(&mysql,query_string))
{
    // error
}
else // query succeeded, process any data returned by it
{
    result = mysql_store_result(&mysql);
    if (result) // there are rows
    {
        num_fields = mysql_num_fields(result);
        // retrieve rows, then call mysql_free_result(result)
    }
    else // mysql_store_result() returned nothing; should it have?
    {
        if(mysql_field_count(&mysql) == 0)
        {
            // query does not return data
            // (it was not a SELECT)
                num_rows = mysql_affected_rows(&mysql);
        }
        else // mysql_store_result() should have returned data
        {
            fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
        }
    }
}
```

另一种可选的方法是，用 mysql_errno(&mysql)替换 mysql_field_count(&mysql)调用。在该情况下，无论语句是否是 SELECT，你将直接从 mysql_store_result()查找错误，而不是从 mysql_field_count()的值进行推断。

3.1.4.18 mysql_field_seek()

MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset)

描述

将字段光标设置到给定的偏移处。对 mysql_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。

返回值

字段光标的前一个值。

错误

无。

3.1.4.19 mysql_field_tell()

MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)

描述

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

返回值

字段光标的当前偏移量。

错误

无。

3.1.4.20 mysql_free_result()

void mysql_free_result(MYSQL_RES *result)

描述

释放由 mysql_store_result()、mysql_use_result()、mysql_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 mysql_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。

返回值

无。

错误

无。

3.1.4.21 mysql_get_character_set_info()

void mysql_get_character_set_info(MYSQL *mysql, MY_CHARSET_INFO *cs)

描述

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

示例：

```
if (!mysql_set_character_set(&mysql, "utf8"))
{
    MY_CHARSET_INFO cs;
```

```
    mysql_get_character_set_info(&mysql, &cs);
    printf("character set information:\n");
    printf("character set name: %s\n", cs.name);
    printf("collation name: %s\n", cs.csname);
    printf("comment: %s\n", cs.comment);
    printf("directory: %s\n", cs.dir);
    printf("multi byte character min. length: %d\n", cs.mbminlen);
    printf("multi byte character max. length: %d\n", cs.mbmaxlen);
}
```

3.1.4.22 mysql_get_host_info()

char *mysql_get_host_info(MYSQL *mysql)

描述

返回描述了所使用连接类型的字符串，包括服务器主机名。

返回值

代表服务器主机名和连接类型的字符串。

错误

无。

3.1.4.23 mysql_get_proto_info()

unsigned int mysql_get_proto_info(MYSQL *mysql)

描述

返回当前连接所使用的协议版本。

返回值

代表当前连接所使用协议版本的无符号整数。

错误

无。

3.1.4.24 mysql_get_server_info()

char *mysql_get_server_info(MYSQL *mysql)

描述

返回代表服务器版本号的字符串。

返回值

代表服务器版本号的字符串。

错误

无。

3.1.4.25 mysql_get_server_version()

unsigned long mysql_get_server_version(MYSQL *mysql)

描述

以整数形式返回服务器的版本号。

返回值

表示 MySQL 服务器版本的数值，格式如下：

```
major_version*10000 + minor_version *100 + sub_version
```

例如，对于 5.0.12，返回 500012。

在客户端程序中，为了快速确定某些与版本相关的服务器功能是否存在，该函数很有用。

错误

无。

3.1.4.26 mysql_hex_string()

unsigned long mysql_hex_string(char *to, const char *from, unsigned long length)

描述

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。

“from”所指向的字符串必须是长度字节“long”。必须为“to”分配缓冲区，缓冲区至少为 length*2+1 字节长。当 mysql_hex_string()返回时，“to”的内容为由 Null 终结的字符串。返回值是编码字符串的长度，不包括终结用 Null 字符。

可采用 0xvalue 或 X'value'格式将返回值置于 SQL 语句中。但是，返回值不包括 0x 或 X'...'。调用者必须提供所希望的格式是何种。

示例：

```
char query[1000],*end;

end = strmov(query,"INSERT INTO test_table values(");
end = strmov(end,"0x");
end += mysql_hex_string(end,"What's this",11);
end = strmov(end,",0x");
end += mysql_hex_string(end,"binary data: \0\r\n",16);
*end++ = ')';

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
{
    fprintf(stderr, "Failed to insert row, Error: %s\n",
    mysql_error(&mysql));
```

```
}
```

示例中所使用的 strmov()函数包含在 mysqlclient 库中，它的工作方式类似于 strcpy()，但返回指向第 1 个参数终结 Null 的指针。

返回值

置于“to”中的值的长度，不包括终结用 Null 字符。

错误

无。

3.1.4.27 mysql_info()

char *mysql_info(MYSQL *mysql)

描述

检索字符串，该字符串提供了关于最近执行查询的信息，但仅对这里列出的语句有效。对于其他语句，mysql_info()返回 NULL。字符串的格式取决于查询的类型，如本节所述。数值仅是说明性的，字符串包含与查询相适应的值。

- INSERT INTO ... SELECT ...

字符串格式：记录，100；副本，0；警告，0

- INSERT INTO ... VALUES (...),(...),(...)...

字符串格式：记录，3；副本，0；警告，0

- LOAD DATA INFILE ...

字符串格式：记录，1；删除，0；跳过，0；警告，0

- ALTER TABLE

字符串格式：记录，3；副本，0；警告，0

- UPDATE

字符串格式：匹配行，40；更改，40；警告，0

注意，mysql_info()为 INSERT ... VALUES 返回非 NULL 值，INSERT ... VALUES 仅用于多行形式的语句（也就是说，仅当指定了多个值列表时）。

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。

错误

无。

3.1.4.28 mysql_init()

MYSQL *mysql_init(MYSQL *mysql)

描述

分配或初始化与 mysql_real_connect()相适应的 MYSQL 对象。如果 mysql 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 mysql_init()分配了新的对象，当调用 mysql_close()来关闭连接时。将释放该对象。

返回值

初始化的 MYSQL*句柄。如果无足够内存以分配新的对象，返回 NULL。

错误

在内存不足的情况下，返回 NULL。

3.1.4.29 mysql_insert_id()

my_ulonglong mysql_insert_id(MYSQL *mysql)

描述

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。

更准确地讲，将在下述条件下更新 mysql_insert_id()：

- 将值保存到 AUTO_INCREMENT 列中的 INSERT 语句。无论值是通过在列中存储特殊值 NULL 或 0 自动生成的，还是确切的非特殊值，都成立。
- 在有多行 INSERT 语句的情况下，mysql_insert_id()返回第 1 个自动生成的 AUTO_INCREMENT 值，如果未生成这类值，将返回插入在 AUTO_INCREMENT 列中的最后 1 个确切值。
- 通过将 LAST_INSERT_ID(expr)插入到任意列中以生成 AUTO_INCREMENT 值的 INSERT 语句。
- 通过更新任意列至 LAST_INSERT_ID(expr)以生成 AUTO_INCREMENT 值的 INSERT 语句。
- mysql_insert_id()的值不受诸如 SELECT 等返回结果集的语句的影响。
- 如果前面的语句返回了错误，mysql_insert_id()的值将是不确定的。

注意，如果前面的语句未使用 AUTO_INCREMENT，mysql_insert_id()返回 0。如果需要保存值，在生成值的语句后，务必立刻调用 mysql_insert_id()。

mysql_insert_id()的值仅受在当前客户端连接内发出的语句的影响。不受由其他客户端发出的语句的影响。

此外还应注意，SQL LAST_INSERT_ID()函数的值总包含最近生成的 AUTO_INCREMENT 值，而且在语句之间不会被复位，原因在于该函数的值是在服务器中维护的。另一个区别是，如果设置了 AUTO_INCREMENT 列来指定非特殊值，不会更新 LAST_INSERT_ID()。

LAST_INSERT_ID()不同于 mysql_insert_id()的原因在于，LAST_INSERT_ID()在脚本中很容易使用，而 mysql_insert_id()则试图提供关于在 AUTO_INCREMENT 列中出现情况的更准确信息。

返回值

在前面的讨论中已予以了介绍。

错误

无。

3.1.4.30 mysql_kill()

int mysql_kill(MYSQL *mysql, unsigned long pid)

描述

请求服务器杀死由 pid 指定的线程。

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

MySQL 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。

3.1.4.31 mysql_library_end()

void mysql_library_end(void)

描述

它是 mysql_server_end()函数的同义词。

3.1.4.32 mysql_library_init()

int mysql_library_init(int argc, char **argv, char **groups)

描述

这是 mysql_server_init()函数的同义词。

3.1.4.33 mysql_list_dbs()

MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wild)

描述

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 mysql_list_dbs()的方法类似于执行查询 SHOW database [LIKE wild]。

必须用 mysql_free_result()释放结果集。

返回值

成功后返回 MYSQL_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_OUT_OF_MEMORY

内存溢出。

• CR_SERVER_GONE_ERROR

MySQL 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。

3.1.4.34 mysql_list_fields()

MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const char *wild)

描述

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 mysql_list_fields()的方法类似于执行查询 SHOW COLUMNS FROM tbl_name [LIKE wild]。

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

必须用 mysql_free_result()释放结果集。

返回值

如果成功，返回 MYSQL_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

MySQL 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。

3.1.4.35 mysql_list_processes()

MYSQL_RES *mysql_list_processes(MYSQL *mysql)

描述

返回描述当前服务器线程的结果集。该类信息与 mysqladmin processlist 或 SHOW PROCESSLIST 查询给出的信息相同。

必须用 mysql_free_result()释放结果集。

返回值

如果成功，返回 MYSQL_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

MySQL 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。

3.1.4.36 mysql_list_tables()

MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild)

描述

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 mysql_list_tables()的方法类似于执行查询 HOW tables [LIKE wild]。

必须用 mysql_free_result()释放结果集。

返回值

如果成功，返回 MYSQL_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

MySQL 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。

3.1.4.37 mysql_more_results()

my_bool mysql_more_results(MYSQL *mysql)

描述

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mysql_next_result()来获取结果。

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）。

在大多数情况下，可调用 mysql_next_result()来测试是否存在多个结果，如果存在多个结果，对检索进行初始化操作。

错误

无。

3.1.4.38 mysql_next_result()

int mysql_next_result(MYSQL *mysql)

描述

如果存在多个查询结果，mysql_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 mysql_free_result()。

调用了 mysql_next_result()后，连接状态就像你已为下一查询调用了 mysql_real_query()或 mysql_query()时的一样。这意味着你能调用 mysql_store_result()、mysql_warning_count()、mysql_affected_rows()等等。

如果 mysql_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

返回值

返回值	描述
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。例如，没有为前面的结果集调用 mysql_use_result()。

- CR_SERVER_GONE_ERROR

MySQL 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。

3.1.4.39 mysql_num_fields()

unsigned int mysql_num_fields(MYSQL_RES *result)

要想传递 MYSQL*参量取而代之，请使用无符号整数 mysql_field_count(MYSQL *mysql)。

描述

返回结果集中的行数。

注意，你可以从指向结果集的指针或指向连接句柄的指针获得行数。如果 mysql_store_result()或 mysql_use_result()返回 NULL，应使用连接句柄（因而没有结果集指针）。在该情况下，可调用 mysql_field_count()来判断 mysql_store_result()是否生成了非空结果。这样，客户端程序就能采取恰当的行动，而不需要知道查询是否是 SELECT 语句（或类似 SELECT 的语句）。在下面的示例中，介绍了执行该操作的方式。

返回值

表示结果集中行数的无符号整数。

错误

无。

示例：

```
MYSQL_RES *result;
unsigned int num_fields;
unsigned int num_rows;

if (mysql_query(&mysql,query_string))
{
    // error
}
else // query succeeded, process any data returned by it
{
    result = mysql_store_result(&mysql);
    if (result) // there are rows
    {
        num_fields = mysql_num_fields(result);
        // retrieve rows, then call mysql_free_result(result)
    }
    else // mysql_store_result() returned nothing; should it have?
    {
        if (mysql_errno(&mysql))
        {
            fprintf(stderr, "Error: %s\n", mysql_error(&mysql));
        }
        else if (mysql_field_count(&mysql) == 0)
        {
            // query does not return data
            // (it was not a SELECT)
            num_rows = mysql_affected_rows(&mysql);
```

```
        }
    }
}
```

另一种可选方式是（如果你知道你的查询应返回结果集），使用检查“mysql_field_count(&mysql) is = 0”来替换 mysql_errno(&mysql)调用。仅当出错时才应使用它。

3.1.4.40 mysql_num_rows()

my_ulonglong mysql_num_rows(MYSQL_RES *result)

描述

返回结果集中的行数。

mysql_num_rows()的使用取决于是否采用了 mysql_store_result()或 mysql_use_result()来返回结果集。如果使用了 mysql_store_result()，可以立刻调用 mysql_num_rows()。如果使用了 mysql_use_result()，mysql_num_rows()不返回正确的值，直至检索了结果集中的所有行为止。

返回值

结果集中的行数。

错误

无。

3.1.4.41 mysql_options()

int mysql_options(MYSQL *mysql, enum mysql_option option, const char *arg)

描述

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 mysql_init()之后、以及 mysql_connect()或 mysql_real_connect()之前调用 mysql_options()。

选项参量指的是你打算设置的选项。Arg 参量是选项的值。如果选项是整数，那么 arg 应指向整数的值。

可能的选项值：

选项	参量类型	功能
MYSQL_INIT_COMMAND	char *	连接到 MySQL 服务器时将执行的命令, 再次连接时将自动地再次执行。
MYSQL_OPT_COMPRESS	未使用	使用压缩客户端 / 服务器协议
MYSQL_OPT_CONNECT_TIMEOUT	unsigned int *	以秒为单位的连接超时。

MYSQL_OPT_GUESS_CONNECTION	未使用	对于与 libmysqld 链接的应用程序，允许库“猜测”是否使用嵌入式服务器或远程服务器。“猜测”表示，如果设置了主机名但不是本地主机，将使用远程服务器。该行为是默认行为。 可使用 MYSQL_OPT_USE_EMBEDDED_CONNECTION 和 MYSQL_OPT_USE_REMOTE_CONNECTION 覆盖它。对于与 libmysqlclient 链接的应用程序，该选项将被忽略。
MYSQL_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。
MYSQL_OPT_NAMED_PIPE	未使用	使用命名管道连接到 NT 平台上的 MySQL 服务器。
MYSQL_OPT_PROTOCOL	unsigned int *	要使用的协议类型。应是 mysql.h 中定义的 mysql_protocol_type 的枚举值之一。
MYSQL_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）。
MYSQL_OPT_RECONNECT	my_bool *	如果发现连接丢失，启动或禁止与服务器的自动再连接。
MYSQL_OPT_SET_CLIENT_IP	char *	对于与 libmysqld 链接的应用程序（具备鉴定支持特性的已编译 libmysqld），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 libmysqlclient 链接的应用程序，该选项将被忽略。
MYSQL_OPT_USE_EMBEDDED_CONNECTION	未使用	对于与 libmysqld 链接的应用程序，就连接而言，它将强制使用嵌入式服务器。对于与 libmysqlclient 链接的应用程序，该选项将被忽略。
MYSQL_OPT_USE_REMOTE_CONNECTION	未使用	对于与 libmysqld 链接的应用程序，就连接而言，它将强制使用远程服务器。对于与 libmysqlclient 链接的应用程序，该选项将被忽略。
MYSQL_OPT_USE_RESULT	未使用	不使用该选项。
MYSQL_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）。
MYSQL_READ_DEFAULT_FILE	char *	从命名选项文件而不是从 my.cnf 读取选项。
MYSQL_READ_DEFAULT_GROUP	char *	从 my.cnf 或用 MYSQL_READ_DEFAULT_FILE 指定的文件中的命名组读取选项。
MYSQL_REPORT_DATA_TRUNCATION	my_bool *	通过 MYSQL_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）。
MYSQL_SECURE_AUTH	my_bool*	是否连接到不支持密码混编功能的服务器。
MYSQL_SET_CHARSET_DIR	char*	指向包含字符集定义文件的目录的路径名。
MYSQL_SET_CHARSET_NAME	char*	用作默认字符集的字符集的名称。

MYSQL_SHARED_MEMORY_BASE_NAME	char*	命名为与服务器进行通信的共享内存对象。应与你打算连接的 mysqld 服务器使用的选项“-shared-memory-base-name”相同。

注意，如果使用了 MYSQL_READ_DEFAULT_FILE 或 MYSQL_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。
init-command	连接到 MySQL 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 mysql_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 MySQL 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 mysql_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（默认为 "MYSQL"）。
socket	默认的套接字文件。
user	默认用户。

注意，“timeout”（超时）已被“connect-timeout”（连接超时）取代。

返回值

成功时返回 0。如果使用了未知选项，返回非 0 值。

示例：

```
MYSQL mysql;
```

```
mysql_init(&mysql);
mysql_options(&mysql,MYSQL_OPT_COMPRESS,0);
mysql_options(&mysql,MYSQL_READ_DEFAULT_GROUP,"odbc");
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
    mysql_error(&mysql));
}
```

该代码请求客户端使用压缩客户端／服务器协议，并从 my.cnf 文件的 obdc 部分读取额外选项。

3.1.4.42 mysql_ping()

int mysql_ping(MYSQL *mysql)

描述

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。

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 MySQL 服务器本身是否已关闭，连接失效可能有其他原因，如网络故障等。

错误

- CR_COMMANDS_OUT_OF_SYNC

以不恰当的顺序执行了命令。

- CR_SERVER_GONE_ERROR

MySQL 服务器不可用。

- CR_UNKNOWN_ERROR

出现未知错误。

3.1.4.43 mysql_query()

int mysql_query(MYSQL *mysql, const char *query)

描述

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。

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

如果希望了解查询是否应返回结果集，可使用 mysql_field_count()进行检查。

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

MySQL 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。

3.1.4.44 mysql_real_connect()

MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd, const char *db, unsigned int port, const char *unix_socket, unsigned long client_flag)

描述

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

参数的指定方式如下：

• 第 1 个参数应是已有 MYSQL 结构的地址。调用 mysql_real_connect()之前，必须调用 mysql_init()来初始化 MYSQL 结构。通过 mysql_options()调用，可更改多种连接选项。

• “host”的值必须是主机名或 IP 地址。如果“host”是 NULL 或字符串"localhost"，其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。

• “user”参数包含用户的 DosSQL 登录 ID。如果“user”是 NULL 或空字符串""，用户将被视为当前用户。在 UNIX 环境下，它是当前的登录名。在 Windows ODBC 下，必须明确指定当前用户名。

• “passwd”参数包含用户的密码。如果“passwd”是 NULL，仅会对该用户的（拥有 1 个空密码字段的）用户表中的条目进行匹配检查。这样，数据库管理员就能按特定的方式设置 DosSQL 权限系统，根据用户是否拥有指定的密码，用户将获得不同的权限。

注释：调用 mysql_real_connect()之前，不要尝试加密密码，密码加密将由客户端 API 自动处理。

• “db”是数据库名称。如果 db 为 NULL，连接会将默认的数据库设为该值。

• 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。

• 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。

• client_flag 的值通常为 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。
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 mysqld 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。

对于某些参数，能够从选项文件获得取值，而不是取得 mysql_real_connect()调用中的确切值。为此，在调用 mysql_real_connect()之前，应与 MYSQL_READ_DEFAULT_FILE 或 MYSQL_READ_DEFAULT_GROUP 选项一起调用 mysql_options()。随后，在 mysql_real_connect()调用中，为准备从选项文件读取值的每个参数指定“无值”值：

• 对于 host，指定 NULL 值或空字符串("")。

• 对于 user，指定 NULL 值或空字符串。

• 对于 passwd，指定 NULL 值。（对于密码，mysql_real_connect()调用中的空字符串的值不能被选项文件中的字符串覆盖，这是因为空字符串明确指明 MySQL 账户必须有空密码）。

• 对于 db，指定 NULL 值或空字符串

• 对于 port，指定“0”值。

• 对于 unix_socket，指定 NULL 值。

对于某一参数，如果在选项文件中未发现值，将使用它的默认值，如本节前面介绍的那样。

返回值

如果连接成功，返回 MYSQL*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 MySQL 服务器。

• CR_CONNECTION_ERROR

无法连接到本地 MySQL 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，起因于试图连接到具有特定客户端库（该客户端库使用了不同的协议版本）的服务器。如果使用很早的客户端库来建立与较新的服务器（未使用“--old-protocol”选项开始的）的连接，就会出现该情况。

• 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。

示例：

```
MYSQL mysql;

mysql_init(&mysql);
mysql_options(&mysql,MYSQL_READ_DEFAULT_GROUP,"your_prog_name");
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
    mysql_error(&mysql));
}
```

通过使用 mysql_options()，接口库将读取 my.cnf 文件的[client]和[your_prog_name]部分，以确保程序工作，即使某人以某种非标准的方式设置 DosSQL 也同样。

注意，一旦建立了连接，mysql_real_connect()将设置再连接标志（MYSQL 结构的组成部份）的值。值“1”表示，如果因连接丢失而无法执行语句，放弃前将尝试再次连接到服务器。

3.1.4.45 mysql_real_escape_string()

unsigned long mysql_real_escape_string(MYSQL *mysql, char *to, const char *from, unsigned long length)

注意，mysql 必须是有效的开放式连接。之所以需要它是因为，转义功能取决于服务器使用的字符集。

描述

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。编码的字符为 NUL (ASCII 0)、‘\n’、‘\r’、‘\’、‘'’、‘"’、以及 Control-Z。（严格地讲，MySQL 仅需要反斜杠和引号字符，用于引用转义查询中的字符串。该函数能引用其他字符，从而使得它们在日志文件中具有更好的可读性）。

“from”指向的字符串必须是长度字节“long”。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。当 mysql_real_escape_string()返回时，“to”的内容是由 Null 终结的字符串。返回值是编码字符串的长度，不包括终结用 Null 字符。

如果需要更改连接的字符集，应使用 mysql_set_character_set()函数，而不是执行 SET NAMES (或 SET CHARACTER SET)语句。mysql_set_character_set()的工作方式类似于 SET NAMES，但它还能影响 mysql_real_escape_string()所使用的字符集，而 SET NAMES 则不能。

示例：

```
char query[1000],*end;

end = strmov(query,"INSERT INTO test_table values(");
*end++ = '\'';
end += mysql_real_escape_string(&mysql, end,"What's this",11);
*end++ = '\'';
*end++ = ',';
*end++ = '\'';
end += mysql_real_escape_string(&mysql, end,"binary data: \0\r\n",16);
*end++ = '\'';
*end++ = ')';

if (mysql_real_query(&mysql,query,(unsigned int) (end - query)))
```

```
{
    fprintf(stderr, "Failed to insert row, Error: %s\n",
    mysql_error(&mysql));
}
```

该示例中使用的 strmov()函数包含在 mysqlclient 库中，工作方式与 strcpy()类似，但会返回指向第 1 个参数终结用 Null 的指针。

返回值

置于“to”中的值的长度，不包括终结用 Null 字符。

错误

无。

3.1.4.46 mysql_real_query()

int mysql_real_query(MYSQL *mysql, const char *query, unsigned long length)

描述

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。

对于包含二进制数据的查询，必须使用 mysql_real_query()而不是 mysql_query()，这是因为二进制数据可能会包含‘\0’字符。此外，mysql_real_query()比 mysql_query()快，这是因为它不会在查询字符串上调用 strlen()。

如果希望知道查询是否应返回结果集，可使 mysql_field_count()进行检查。

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

MySQL 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。

3.1.4.47 mysql_refresh()

int mysql_refresh(MYSQL *mysql, unsigned int options)

描述

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。

“options”参量是一种位掩码，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

MySQL 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。

3.1.4.48 mysql_reload()

int mysql_reload(MYSQL *mysql)

描述

请求 MySQL 服务器重新加载授权表。连接的用户必须具有 RELOAD 权限。

该函数已过时。最好使用 mysql_query()来发出 SQL FLUSH PRIVILEGES 语句。

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

MySQL 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。

3.1.4.49 mysql_rollback()

my_bool mysql_rollback(MYSQL *mysql)

描述

回滚当前事务。

该函数的动作取决于 completion_type 系统变量的值。尤其是，如果 completion_type 的值为“2”，终结事务后，服务器将执行释放操作，并关闭客户端连接。客户端程序应调用 mysql_close()，从客户端一侧关闭连接。

返回值

如果成功，返回 0，如果出现错误，返回非 0 值。

错误

无。

3.1.4.50 mysql_row_seek()

MYSQL_ROW_OFFSET mysql_row_seek(MYSQL_RES *result,
MYSQL_ROW_OFFSET offset)

描述

将行光标置于查询结果集中的任意行。“offset”值是行偏移量，它应是从 mysql_row_tell()或 mysql_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 mysql_data_seek()。

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

返回值

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

错误

无。

3.1.4.51 mysql_row_tell()

MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)

描述

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

仅应在 mysql_store_result()之后，而不是 mysql_use_result()之后使用 mysql_row_tell()。

返回值

行光标的当前偏移量。

错误

无。

3.1.4.52 mysql_select_db()

int mysql_select_db(MYSQL *mysql, const char *db)

描述

使由 db 指定的数据库成为由 mysql 指定的连接上的默认数据库（当前数据库）。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 mysql_select_db()将失败。

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

MySQL 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。

3.1.4.53 mysql_set_character_set()

int mysql_set_character_set(MYSQL *mysql, char *csname)

描述

该函数用于为当前连接设置默认的字符集。字符串 csname 指定了 1 个有效的字符集名称。连接校对成为字符集的默认校对。该函数的工作方式与 SET NAMES 语句类似，但它还能设置 mysql->charset 的值，从而影响了由 mysql_real_escape_string()设置的字符集。

返回值

0 表示成功，非 0 值表示出现错误。

示例：

```
MYSQL mysql;

mysql_init(&mysql);
if (!mysql_real_connect(&mysql,"host","user","passwd","database",0,NULL,0))
{
    fprintf(stderr, "Failed to connect to database: Error: %s\n",
    mysql_error(&mysql));
}

if (!mysql_set_charset_name(&mysql, "utf8"))
{
    printf("New client character set: %s\n", mysql_character_set_name(&mysql));
}
```

3.1.4.54 mysql_set_server_option()

int mysql_set_server_option(MYSQL *mysql, enum enum_mysql_set_option option)

描述

允许或禁止连接的选项。选项可以取下述值之一：

MYSQL_OPTION_MULTI_STATEMENTS_ON	允许多语句支持。
MYSQL_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

MySQL 服务器不可用。

• CR_SERVER_LOST

在查询过程中，与服务器的连接丢失。

• ER_UNKNOWN_COM_ERROR

服务器不支持 mysql_set_server_option()或服务器不支持试图设置的选项。

3.1.4.55 mysql_shutdown()

int mysql_shutdown(MYSQL *mysql, enum enum_shutdown_level shutdown_level)

描述

请求数据库服务器关闭。已连接的用户必须具有 SHUTDOWN 权限。DosSQL 服务器仅支持 1 种关闭类型，shutdown_level 必须等效于 SHUTDOWN_DEFAULT。设计规划了额外的关闭级别，以便能够选择所需的级别。对于用旧版本 libmysqlclient 头文件编译并调用 mysql_shutdown()的动态链接可执行程序，需要与旧版本的 libmysqlclient 动态库一起使用。

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

MySQL 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。

3.1.4.56 mysql_sqlstate()

const char *mysql_sqlstate(MYSQL *mysql)

描述

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 DosSQL 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。

返回值

包含 SQLSTATE 错误码的、由 Null 终结的字符串。

3.1.4.57 mysql_ssl_set()

int mysql_ssl_set(MYSQL *mysql, const char *key, const char *cert, const char *ca, const char *capath, const char *cipher)

描述

使用 mysql_ssl_set()，可采用 SSL 建立安全连接。必须在 mysql_real_connect()之前调用它。

除非在客户端库中允许了 OpenSSL 支持，否则 mysql_ssl_set()不作任何事。

Mysql 是从 mysql_init()返回的连接处理程序。其他参数的指定如下：

- key 是 key 文件的路径名。
- cert 是证书文件的路径名。
- ca 是证书授权文件的路径名。
- capath 是指向目录的路径名，该目录中包含以 pem 格式给出的受信任 SSL CA 证书。
- cipher 是允许密码的列表，用于 SSL 加密。

对于任何未使用的 SSL 参数，可为其给定 NULL。

返回值

该函数总返回 0。如果 SSL 设置不正确，当你尝试连接时，mysql_real_connect()将返回错误。

3.1.4.58 mysql_stat()

char *mysql_stat(MYSQL *mysql)

描述

返回包含特定信息的字符串，该信息与 `mysqladmin status` 命令提供的信息类似。包括以秒为单位的正常运行时间，以及运行线程的数目，问题数，再加载次数，以及打开的表数目。

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

MySQL 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。

3.1.4.59 mysql_store_result()

MYSQL_RES *mysql_store_result(MYSQL *mysql)

描述

对于成功检索了数据的每个查询（SELECT、SHOW、DESCRIBE、EXPLAIN、CHECK TABLE 等），必须调用 mysql_store_result()或 mysql_use_result() 。

对于其他查询，不需要调用 mysql_store_result()或 mysql_use_result()，但是如果在任何情况下均调用了 mysql_store_result()，它也不会导致任何伤害或性能降低。通过检查 mysql_store_result()是否返回 0，可检测查询是否没有结果集（以后会更多）。

如果希望了解查询是否应返回结果集，可使 mysql_field_count()进行检查。

mysql_store_result()将查询的全部结果读取到客户端，分配 1 个 MYSQL_RES 结构，并将结果置于该结构中。

如果查询未返回结果集，mysql_store_result()将返回 Null 指针（例如，如果查询是 INSERT 语句）。

如果读取结果集失败，mysql_store_result()还会返回 Null 指针。通过检查 mysql_error()是否返回非空字符串，mysql_errno()是否返回非 0 值，或 mysql_field_count()是否返回 0，可以检查是否出现了错误。

如果未返回行，将返回空的结果集。（空结果集设置不同于作为返回值的空指针）。

一旦调用了 mysql_store_result()并获得了不是 Null 指针的结果，可调用 mysql_num_rows()来找出结果集中的行数。

可以调用 mysql_fetch_row()来获取结果集中的行，或调用 mysql_row_seek()和 mysql_row_tell()来获取或设置结果集中的当前行位置。

一旦完成了对结果集的操作，必须调用 mysql_free_result()。

返回值

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

错误

如果成功，mysql_store_result()将复位 mysql_error()和 mysql_errno()。

• 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

MySQL 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。

3.1.4.60 mysql_thread_id()

unsigned long mysql_thread_id(MYSQL *mysql)

描述

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

如果连接丢失，并使用 mysql_ping()进行了再连接，线程 ID 将改变。这意味着你不应获取线程 ID 并保存它供以后使用，应在需要时获取它。

返回值

当前连接的线程 ID。

错误

无。

3.1.4.61 mysql_use_result()

MYSQL_RES *mysql_use_result(MYSQL *mysql)

描述

对于成功检索数据的每个查询（SELECT、SHOW、DESCRIBE、EXPLAIN），必须调用 mysql_store_result()或 mysql_use_result()。

mysql_use_result()将初始化结果集检索，但并不像 mysql_store_result()那样将结果集实际读取到客户端。它必须通过对 mysql_fetch_row()的调用，对每一行分别进行检索。这将直接从服务器读取结果，而不会将其保存在临时表或本地缓冲区内，与 mysql_store_result()相比，速度更快而且使用的内存也更少。客户端仅为当前行和通信缓冲区分配内存，分配的内存可增加到 max_allowed_packet 字节。

另一方面，如果你正在客户端一侧为各行进行大量的处理操作，或者将输出发送到了用户可能会键入“^S”（停止滚动）的屏幕，就不应使用 mysql_use_result()。这会绑定服务器，并阻止其他线程更新任何表（数据从这类表获得）。

使用 mysql_use_result()时，必须执行 mysql_fetch_row()，直至返回 NULL 值，否则，未获取的行将作为下一个检索的一部分返回。C API 给出命令不同步错误，如果忘记了执行该操作，将不能运行该命令。

不应与从 mysql_use_result()返回的结果一起使用 mysql_data_seek()、mysql_row_seek()、mysql_row_tell()、mysql_num_rows()或 mysql_affected_rows()，也不应发出其他查询，直至 mysql_use_result()完成为止。（但是，提取了所有行后，mysql_num_rows()将准确返回提取的行数）。

一旦完成了对结果集的操作，必须调用 mysql_free_result()。

使用 libmysqld 嵌入式服务器时，由于在调用 mysql_free_result()之前，内存使用将随着每个检索的行增加，内存效益将基本丧失。

返回值

MYSQL_RES 结果结构。如果出现错误，返回 NULL。

错误

如果成功，mysql_use_result()将复位 mysql_error()和 mysql_errno()。

- 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

MySQL 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。

3.1.4.62 mysql_warning_count()

unsigned int mysql_warning_count(MYSQL *mysql)

错误

返回执行前一个 SQL 语句期间生成的告警数目。

返回值

告警计数。

错误

无。

3.1.5 C API 预处理语句

DosSQL 客户端 / 服务器协议提供了预处理语句。该功能采用了由 mysql_stmt_init()初始化函数返回的 MYSQL_STMT 语句处理程序数据结构。对于多次执行的语句，预处理执行是一种有效的方式。首先对语句进行解析，为执行作好准备。接下来，在需要时使用初始化函数返回的语句句柄执行一次或多次。

对于多次执行的语句，预处理执行比直接执行快。主要原因在于，仅对查询执行一次解析操作。在直接执行的情况下，每次执行语句时，均将进行查询。此外，由于每次执行预处理语句时仅需发送参数的数据，从而减少了网络通信量。

预处理语句的另一个优点是，它采用了二进制协议，从而使得客户端和服务器之间的数据传输更有效率。

下述语句可用作预处理语句：CREATE TABLE、DELETE、DO、INSERT、REPLACE、SELECT、SET、UPDATE、以及大多数 SHOW 语句。在 MySQL 5.1 中，不支持其他语句。

3.1.6 C API 预处理语句的数据类型

预处理语句主要使用 MYSQL_STMT 和 MYSQL_BIND 数据结构。第 3 种结构 MYSQL_TIME 用于传输暂时性数据。

• MYSQL_STMT

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

MYSQL_STMT 结构没有供应用程序使用的参数。此外，不应尝试复制 MYSQL_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。

• MYSQL_BIND

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 mysql_stmt_bind_param()一起使用，用于将参数数据值绑定到缓冲区上，以供 mysql_stmt_execute()使用。对于输出，它与 mysql_stmt_bind_result()一起使用，用于绑定结果缓冲区，以便用于 with mysql_stmt_fetch()以获取行。

MYSQL_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。

enum enum_field_types buffer_type

缓冲的类型。在本节后面列出了允许的 buffer_type 值。对于输入，buffer_type 指明了与语句参数捆绑的值类型。对于输出，它指明了你希望从结果缓冲区收到的值类型。

void *buffer

对于输入，这是指向存储语句参数数据值的缓冲的指针。对于输出，它是指向返回结果集列值的缓冲的指针。对于数值列类型，缓冲应指向恰当的 C 类型变量（如果将该变量与具有 UNSIGNED 属性的列关联起来，则为变量 unsigned C 类型。通过使用 is_unsigned 成员，指明变量是 signed 或 unsigned 类型，详情请参见本节后面的介绍）。对于日期和时间列类型，缓冲应指向 MYSQL_TIME 结构。对于字符和二进制字符串列类型，缓冲应指向字符缓冲区。

unsigned long buffer_length

*buffer 的实际大小，单位为字节，它指明了可保存在缓冲区内的最大数据。对于字符和二进制 C 数据，buffer_length 值指定了与 mysql_stmt_bind_param()一起使用时的*buffer 长度，或与 mysql_stmt_bind_result()一起使用时能够提取到缓冲区内的最大数据。

unsigned long *length

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。“length”用于字符或二进制 C 数据。对于输入参数数据绑定，“length”指向 unsigned long 变量，该变量指明了存储在*buffer 中参数值的长度，供 mysql_stmt_execute()使用。对于输出值绑定，mysql_stmt_fetch()会将返回的列值保存到“length”指向的变量中。

对于数值和临时数据类型，“length”将被忽略，原因在于该数据值的长度是由 buffer_type 值决定的。

my_bool *is_null

该成员指向 my_bool 变量，如果值为 NULL，该变量为“真”，如果值为非 Null，该变量为“假”。对于输入，将*is_null 设置为“真”，指明以语句参数的形式传递 NULL 值。对于输出，如果从语句返回的结果集列值为 NULL，当获取了行后，该值将被设为“真”。

“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：

♦ 如果数据值总是 NULL，使用 MYSQL_TYPE_NULL 绑定列。

- 如果数据值总是 NOT NULL，设置 is_null = (my_bool*) 0。
- 在所有其他情况下，应将 is_null 设置为 my_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。

my_bool is_unsigned

该成员用于整数类型。（对应于 MYSQL_TYPE_TINY、MYSQL_TYPE_SHORT、MYSQL_TYPE_LONG、以及 MYSQL_TYPE_LONGLONG 类型的代码）。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”。

my_bool error

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 MYSQL_REPORT_DATA_TRUNCATION 选项的 mysql_options()，启用截短通报功能。允许该功能后，mysql_stmt_fetch()返回 MYSQL_DATA_TRUNCATED，而且对于出现截短情况的参数，在 MYSQL_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。

要想使用 MYSQL_BIND 结构，应将其内容置为 0 以便初始化它，然后对其进行设置，恰当地描述它。例如，要想声明并初始化三个 MYSQL_BIND 结构的数组，可使用下述代码：

```
MYSQL_BIND bind[3];.0
memset(bind, 0, sizeof(bind));
```

• MYSQL_TIME

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。将 MYSQL_BIND 结构的 buffer_type 成员设置为临时值之一，并将 buffer 成员设置为指向 MYSQL_TIME 结构，即可实现该点。

MYSQL_TIME 结构包含下述成员：

unsigned int year

年份

unsigned int month

月份

unsigned int day

天

unsigned int hour

小时

unsigned int minute

分钟

unsigned int second

秒

my_bool neg

布尔标志，用于指明时间是否为负数。

unsigned long second_part

秒的分数部分，该成员目前不使用。

仅使用施加在给定临时类型值上的 MYSQL_TIME 结构的部分：用于 DATE、DATETIME 和 TIMESTAMP 的年、月、日部分。用于 TIME、DATETIME 和 TIMESTAMP 值的小时、分钟、秒部分。

在下面的表格中，给出了可在 MYSQL_BIND 结构的 buffer_type 成员中指定的允许值。在该表中，还给出了与每个 buffer_type 值最接近的对应 SQL 类型，对于数值和临时类型，给出了对应的 C 类型。

buffer_type 值	SQL 类型	C 类型
MYSQL_TYPE_TINY	TINYINT	char
MYSQL_TYPE_SHORT	SMALLINT	short int
MYSQL_TYPE_LONG	INT	int
MYSQL_TYPE_LONGLONG	BIGINT	long long int
MYSQL_TYPE_FLOAT	FLOAT	float
MYSQL_TYPE_DOUBLE	DOUBLE	double
MYSQL_TYPE_TIME	TIME	MYSQL_TIME
MYSQL_TYPE_DATE	DATE	MYSQL_TIME
MYSQL_TYPE_DATETIME	DATETIME	MYSQL_TIME
MYSQL_TYPE_TIMESTAMP	TIMESTAMP	MYSQL_TIME
MYSQL_TYPE_STRING	CHAR	
MYSQL_TYPE_VAR_STRING	VARCHAR	
MYSQL_TYPE_TINY_BLOB	TINYBLOB/TINYTEXT	
MYSQL_TYPE_BLOB	BLOB/TEXT	
MYSQL_TYPE_MEDIUM_BLOB	MEDIUMBLOB/MEDIUMTEXT	
MYSQL_TYPE_LONG_BLOB	LONGBLOB/LONGTEXT	

隐式类型转换可沿两个方向执行。

3.1.7 C API 预处理语句函数概述

3.1.7.1 函数列表

在此归纳了预处理语句处理功能可使用的函数，并在后面的章节中作了详细的介绍。

函数	描述
mysql_stmt_affected_rows()	返回由预处理语句 UPDATE、DELETE 或 INSERT 变更、删除或插入的行数目。
mysql_stmt_attr_get()	获取预处理语句属性的值。
mysql_stmt_attr_set()	设置预处理语句的属性。
mysql_stmt_bind_param()	将应用程序数据缓冲并与预处理 SQL 语句中的参数标记符关联起来。
mysql_stmt_bind_result()	将应用程序数据缓冲并与结果集中的列关联起来。

mysql_stmt_close()	释放预处理语句使用的内存。
mysql_stmt_data_seek()	寻找语句结果集中的任意行编号。
mysql_stmt_errno()	返回上次语句执行的错误编号。
mysql_stmt_error()	返回上次语句执行的错误消息。
mysql_stmt_execute()	执行预处理语句。
mysql_stmt_fetch()	从结果集获取数据的下一行，并返回所有绑定列的数据。
mysql_stmt_fetch_column()	获取结果集当前行中某列的数据。
mysql_stmt_field_count()	对于最近的语句，返回结果行的数目。
mysql_stmt_free_result()	释放分配给语句句柄的资源。
mysql_stmt_init()	为 MYSQL_STMT 结构分配内存并初始化它。
mysql_stmt_insert_id()	对于预处理语句的 AUTO_INCREMENT 列，返回生成的 ID。
mysql_stmt_num_rows()	从语句缓冲结果集返回总行数。
mysql_stmt_param_count()	返回预处理 SQL 语句中的参数数目。
mysql_stmt_param_metadata()	返回结果集的参数元数据。
mysql_stmt_prepare()	为执行操作准备 SQL 字符串。
mysql_stmt_reset()	复位服务器中的语句缓冲区。
mysql_stmt_result_metadata()	以结果集形式返回预处理语句元数据。
mysql_stmt_row_seek()	使用从 mysql_stmt_row_tell()返回的值，查找语句结果集中的行偏移。
mysql_stmt_row_tell()	返回语句行光标位置。
mysql_stmt_send_long_data()	将程序块中的长数据发送到服务器。
mysql_stmt_sqlstate()	返回关于上次语句执行的 SQLSTATE 错误代码。
mysql_stmt_store_result()	将完整的结果集检索到客户端。

调用 mysql_stmt_init()以创建语句句柄，然后调用 mysql_stmt_prepare 准备语句，调用 mysql_stmt_bind_param()提供参数数据，并调用 mysql_stmt_execute()执行语句。通过更改 mysql_stmt_bind_param()提供的相应缓冲区中的参数值，可重复执行 mysql_stmt_execute()。

如果语句是 SELECT 或任何其他能生成结果集的语句，mysql_stmt_prepare()也会通过 mysql_stmt_result_metadata()以 MYSQL_RES 结果集的形式返回结果集元数据信息。

你可以使用 mysql_stmt_bind_result()提供结果缓冲，以便 mysql_stmt_fetch()能自动将数据返回给这些缓冲。这是一种按行获取方式。

此外，你也能使用 mysql_stmt_send_long_data()将程序块中的文本或二进制数据发送到服务器。

完成语句执行后，必须使用 mysql_stmt_close()关闭语句句柄，以便与之相关的所有资源均能被释放。

如果通过调用 mysql_stmt_result_metadata()获得了 SELECT 语句的结果集元数据，也应使用 mysql_free_result()释放元数据。

3.1.7.2 执行步骤

要想准备和执行语句，应用程序必须采取下述步骤：

1. 用 msyql_stmt_init()创建预处理语句句柄。要想在服务器上准备预处理语句，可调用 mysql_stmt_prepare()，并为其传递包含 SQL 语句的字符串。

2. 如果语句生成了结果集，调用 mysql_stmt_result_metadata()以获得结果集元数据。与包含查询返回列的结果集不同，该元数据本身也采用了结果集的形式。元数据结果集指明了结果中包含多少列，并包含每一列的信息。

3. 使用 mysql_stmt_bind_param()设置任何参数的值。必须设置所有参数，否则语句执行将返回错误，或生成无法预料的结果。

4. 调用 mysql_stmt_execute()执行语句。

5. 如果语句生成了结果集，捆绑数据缓冲，通过调用 mysql_stmt_bind_result()，检索行值。

6. 通过重复调用 mysql_stmt_fetch()，按行将数据提取到缓冲区，直至未发现更多行为止。

7. 通过更改参数值并再次执行语句，重复步骤 3 到步骤 6。

调用 mysql_stmt_prepare()时，MySQL 客户端／服务器协议将执行下述操作：

- 服务器解析语句，并通过赋值语句 ID 将 OK 状态发回客户端。此外，如果它是面向结果集的语句，还将发送总的参数数目，列计数和元数据。在此调用过程中，服务器将检查语句的所有语法和语义。
- 客户端采用该语句 ID 用于进一步操作，以便服务器能从其语句池中识别语句。

调用 mysql_stmt_execute()时，MySQL 客户端／服务器协议将执行下述操作：

- 客户端使用语句句柄，并将参数数据发送到服务器。
- 服务器使用由客户端提供的 ID 来识别语句，用新提供的数据替换参数标记符，并执行语句。如果语句生成了结果集，服务器将数据发回客户端。否则，服务器将会发送 OK 状态，以及总的变更、删除和插入行数。

调用 mysql_stmt_fetch()时，MySQL 客户端／服务器协议将执行下述操作：

- 客户端按行从信息包读取数据，并通过执行必要的转换操作将其放入应用程序数据缓冲中。如果应用程序的缓冲类型与服务器返回的字段类型相同，转换十分简明。

如果出现了错误，可分别使用 mysql_stmt_errno()、mysql_stmt_error()和 mysql_stmt_sqlstate()获取语句错误代码、错误消息和 SQLSTATE 值。

3.1.7.3 预处理语句日志功能

对于与 mysql_stmt_prepare()和 mysql_stmt_execute() C API 函数一起执行的预处理语句，服务器会将“准备”和“执行”行写入一般查询日志，以便你能了解语句是在何时准备和执行的。

假定按下述方式准备和执行了语句：

1. 调用 mysql_stmt_prepare()以准备语句字符串"SELECT ?"。

2. 调用 mysql_stmt_bind_param()将值“3”绑定到预处理语句中的参数。

3. 调用 mysql_stmt_execute()，执行预处理语句。

上述调用的结果是，服务器将下述行写入一般查询日志：

Prepare [1] SELECT ?

Execute [1] SELECT 3

日志中的每个“准备”和“执行”行均具有[n]语句 ID 标识，这样你就能跟踪已记录的预处理语句。n 是正整数。对于客户端，如果同时有多个活动的预处理语句，n 可能会大于 1。替换了“?”参数的数据值后，每个“执行”行将显示一条预处理语句。

3.1.8 C API 预处理语句函数描述

为了准备和执行查询，请使用下述部分详细介绍的函数。

注意，与 MYSQL_STMT 结构一起使用的所有函数均以前缀 mysql_stmt_开始。

要想创建 MYSQL_STMT 句柄，请使用 mysql_stmt_init()函数。

3.1.8.1 mysql_stmt_affected_rows()

my_ulonglong mysql_stmt_affected_rows(MYSQL_STMT *stmt)

描述

返回上次执行语句更改、删除或插入的总行数。对于 UPDATE、DELETE 或 INSERT 语句，可在 mysql_stmt_execute()之后立刻调用它们。对于 SELECT 语句，mysql_stmt_affected_rows()的工作方式类似于 mysql_num_rows()。

返回值

大于 0 的整数指明了受影响或检索的行数。对于 UPDATE 语句，“0”表明未更新任何记录，在查询中没有与 WHERE 子句匹配的行，或尚未执行任何查询。“-1”表明返回了错误，或对 SELECT 查询，在调用 mysql_stmt_store_result()之前调用了 mysql_stmt_affected_rows()。由于 mysql_stmt_affected_rows()返回无符号值，可通过比较返回值和“(my_ulonglong)-1”（或等效的“(my_ulonglong)~0”），检查“-1”。

错误

无。

3.1.8.2 mysql_stmt_attr_get()

int mysql_stmt_attr_get(MYSQL_STMT *stmt, enum enum_stmt_attr_type option, void *arg)

描述

可用于获得语句属性的当前值。

“option”参量是希望获取的选项，“arg”应指向包含选项值的变量。如果“option”是整数，那么“arg”应指向整数的值。

返回值

如果是 OK，返回 0。如果选项未知，返回非 0 值。

错误

无。

3.1.8.3 mysql_stmt_attr_set()

int mysql_stmt_attr_set(MYSQL_STMT *stmt, enum enum_stmt_attr_type option, const void *arg)

描述

可用于影响预处理语句的行为。可多次调用该函数来设置多个选项。

“option”参量是希望设置的选项，“arg”参量是选项的值。如果“option”是整数，那么“arg”应指向整数的值。

可能的选项值：

选项	参量类型	功能
STMT_ATTR_UPDATE_MAX_LENGTH	my_bool *	如果设为 1：更新 mysql_stmt_store_result()中的元数据 MYSQL_FIELD->max_length。
STMT_ATTR_CURSOR_TYPE	unsigned long *	调用 mysql_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。

如果与 CURSOR_TYPE_READ_ONLY 一起使用了 STMT_ATTR_CURSOR_TYPE 选项，当调用了 mysql_stmt_execute()时，将为语句打开光标。如果存在由前一个 mysql_stmt_execute()调用打开的光标，在打开新的光标前，将关闭该光标。此外，为再执行而准备语句之前，mysql_stmt_reset()还将关闭任何打开的光标。mysql_stmt_free_result()将关闭任何打开的光标。

如果为预处理语句打开了光标，没必要调用 mysql_stmt_store_result()，这是因为，该函数会导致在客户端一侧对结果集进行缓冲处理。

在 MySQL 5.0.2 中增加了 STMT_ATTR_CURSOR_TYPE 选项。在 MySQL 5.0.6 中，增加了 STMT_ATTR_PREFETCH_ROWS 选项。

返回值

如果是 OK，返回 0。如果选项未知，返回非 0 值。

错误

无。

示例：

在下述示例中，为预处理语句打开了 1 个光标，并将每次获取的行数设为 5：

```
MYSQL_STMT *stmt;
int rc;
unsigned long type;
unsigned long prefetch_rows = 5;

stmt = mysql_stmt_init(mysql);
type = (unsigned long) CURSOR_TYPE_READ_ONLY;
rc = mysql_stmt_attr_set(stmt, STMT_ATTR_CURSOR_TYPE, (void*) &type);
/* ... check return value ... */
rc = mysql_stmt_attr_set(stmt, STMT_ATTR_PREFETCH_ROWS,
(void*) &prefetch_rows);
/* ... check return value ... */
```

3.1.8.4 mysql_stmt_bind_param()

my_bool mysql_stmt_bind_param(MYSQL_STMT *stmt, MYSQL_BIND *bind)

描述

mysql_stmt_bind_param()用于为 SQL 语句中的参数标记符绑定数据，以传递给 mysql_stmt_prepare()。它使用 MYSQL_BIND 结构来提供数据。“bind”是 MYSQL_BIND 结构的某一数组的地址。按照客户端库的预期，对于查询中出现的每个“?”参数标记符，数组中均包含 1 个元素。

假定你准备了下述语句：

INSERT INTO mytbl VALUES(?,?,?)

绑定参数时，MYSQL_BIND 结构的数组包含 3 个元素，并能声明如下：

MYSQL_BIND bind[3];

在“C API 预处理语句的数据类型”中，介绍了应设置的每个 MYSQL_BIND 元素的成员。

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。

示例：

关于 mysql_stmt_bind_param()的用法，请参见“mysql_stmt_execute()”给出的示例。

3.1.8.5 mysql_stmt_bind_result()

my_bool mysql_stmt_bind_result(MYSQL_STMT *stmt, MYSQL_BIND *bind)

描述

mysql_stmt_bind_result()用于将结果集中的列与数据缓冲和长度缓冲关联（绑定）起来。当调用 mysql_stmt_fetch()以获取数据时，MySQL 客户端／服务器协议会将绑定列的数据置于指定的缓冲区内。

调用 mysql_stmt_fetch()之前，必须将所有列绑定到缓冲。“bind”是 MYSQL_BIND 结构某一数组的地址。按照客户端库的预期，对于结果集中的每一列，数组应包含相应的元素。如果未将列绑定到 MYSQL_BIND 结构，mysql_stmt_fetch()将简单地忽略数据获取操作。缓冲区应足够大，足以容纳数据值，这是因为协议不返回成块的数据值。

可以在任何时候绑定或再绑定列，即使已部分检索了结果集后也同样。新的绑定将在下一次调用 mysql_stmt_fetch()时起作用。假定某一应用程序绑定了结果集中的列，并调用了 mysql_stmt_fetch()。客户端／服务器协议将返回绑定缓冲区中的数据。接下来，假定应用程序将多个列绑定到不同的缓冲，该协议不会将数据置于新绑定的缓冲区，直至下次调用 mysql_stmt_fetch()为止。

要想绑定列，应用程序将调用 mysql_stmt_bind_result()，并传递类型、地址、以及长度缓冲的地址。在“C API 预处理语句的数据类型”中，介绍了应设置的各 MYSQL_BIND 元素的成员。

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。

示例：

关于 mysql_stmt_bind_result()的用法，请见“mysql_stmt_fetch()”中给出的示例。

3.1.8.6 mysql_stmt_close()

my_bool mysql_stmt_close(MYSQL_STMT *)

描述

关闭预处理语句。此外，mysql_stmt_close()还会取消由“stmt”指向的语句句柄分配。

如果当前语句已挂起或未读取结果，该函数将取消它们，以便能执行下一个查询，

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

MySQL 服务器不可用。

• CR_UNKNOWN_ERROR

出现未知错误。

示例：

关于 mysql_stmt_close()的用法，请见“mysql_stmt_execute()”中给出的示例。

3.1.8.7 mysql_stmt_data_seek()

void mysql_stmt_data_seek(MYSQL_STMT *stmt, my_ulonglong offset)

描述

查找语句结果集中的任意行。偏移量为行编号，应位于从 0 到 mysql_stmt_num_rows(stmt)-1 的范围内。

该函数要求语句结果集结构包含上次执行查询的全部结果，这样，mysql_stmt_data_seek()就能与 mysql_stmt_store_result()一起使用。

返回值

无。

错误

无。

3.1.8.8 mysql_stmt_errno()

unsigned int mysql_stmt_errno(MYSQL_STMT *stmt)

描述

对于由 stmt 指定的语句，mysql_stmt_errno()将返回最近调用的语句 API 函数的错误代码，该函数或成功或失败。“0”返回值表示未出现错误。在 MySQL errmsg.h 头文件中列出了客户端错误消息编号。在 mysqld_error.h 中，列出了服务器错误消息。

返回值

错误代码值。如果未出现错误，返回 0。

错误

无。

3.1.8.9 mysql_stmt_error()

const char *mysql_stmt_error(MYSQL_STMT *stmt)

描述

对于由 stmt 指定的语句，mysql_stmt_error()返回由 Null 终结的字符串，该字符串包含最近调用的语句 API 函数的错误消息，该函数或成功或失败。如果未出现错误，返回空字符串("")。这意味着下述两个测试是等效的：

```
if (mysql_stmt_errno(stmt))
{
  // an error occurred
}
```

```
if (mysql_stmt_error(stmt)[0])
{
  // an error occurred
}
```

通过重新编译 MySQL 客户端库，可更改客户端错误消息的语言。目前，能够选择数种语言之一显示错误消息。

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。

错误

无。

3.1.8.10 mysql_stmt_execute()

int mysql_stmt_execute(MYSQL_STMT *stmt)

描述

mysql_stmt_execute()执行与语句句柄相关的预处理查询。在该调用期间，将当前绑定的参数标记符的值发送到服务器，服务器用新提供的数据替换标记符。

如果语句是 UPDATE、DELETE 或 INSERT，通过调用 mysql_stmt_affected_rows()，可发现更改、删除或插入的总行数。如果这是诸如 SELECT 等能生成结果集的语句，调用任何其他能导致查询处理的函数之前，必须调用 mysql_stmt_fetch()来获取数据。

对于生成结果集的语句，执行语句之前，可通过调用 mysql_stmt_attr_set()，请求 mysql_stmt_execute()为语句打开光标。如果多次执行某一语句，在打开新的光标前，mysql_stmt_execute()将关闭任何已打开的光标。

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

MySQL 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。

示例：

在下面的示例中，介绍了使用 mysql_stmt_init()、mysql_stmt_prepare()、mysql_stmt_param_count()、mysql_stmt_bind_param()、mysql_stmt_execute()、以及 mysql_stmt_affected_rows()创建和填充表的方法。假定 mysql 变量具有有效的连接句柄。

```
#define STRING_SIZE 50

#define DROP_SAMPLE_TABLE "DROP TABLE IF EXISTS test_table"
#define CREATE_SAMPLE_TABLE "CREATE TABLE test_table(col1 INT,\
  col2 VARCHAR(40),\
  col3 SMALLINT,\
  col4 TIMESTAMP)"
#define INSERT_SAMPLE "INSERT INTO test_table(col1,col2,col3) VALUES(?,?,?)"

MYSQL_STMT *stmt;
MYSQL_BIND bind[3];
my_ulonglong affected_rows;
int param_count;
short small_data;
int int_data;
char str_data[STRING_SIZE];
unsigned long str_length;
my_bool    is_null;

if (mysql_query(mysql, DROP_SAMPLE_TABLE))
{
     fprintf(stderr, " DROP TABLE failed\n");
     fprintf(stderr, " %s\n", mysql_error(mysql));
     exit(0);
}

if (mysql_query(mysql, CREATE_SAMPLE_TABLE))
{
       fprintf(stderr, " CREATE TABLE failed\n");
       fprintf(stderr, " %s\n", mysql_error(mysql));
       exit(0);
```

```
}

/* Prepare an INSERT query with 3 parameters */
/* (the TIMESTAMP column is not named; the server */
/* sets it to the current date and time) */
stmt = mysql_stmt_init(mysql);
if (!stmt)
{
      fprintf(stderr, " mysql_stmt_init(), out of memory\n");
      exit(0);
}
if (mysql_stmt_prepare(stmt, INSERT_SAMPLE, strlen(INSERT_SAMPLE)))
{
      fprintf(stderr, " mysql_stmt_prepare(), INSERT failed\n");
      fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
      exit(0);
}
fprintf(stdout, " prepare, INSERT successful\n");

/* Get the parameter count from the statement */
param_count= mysql_stmt_param_count(stmt);
fprintf(stdout, " total parameters in INSERT: %d\n", param_count);

if (param_count != 3) /* validate parameter count */
{
      fprintf(stderr, " invalid parameter count returned by MySQL\n");
      exit(0);
}

/* Bind the data for all 3 parameters */

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

/* INTEGER PARAM */
/* This is a number type, so there is no need to specify buffer_length */
bind[0].buffer_type= MYSQL_TYPE_LONG;
bind[0].buffer= (char *)&int_data;
bind[0].is_null= 0;
bind[0].length= 0;

/* STRING PARAM */
bind[1].buffer_type= MYSQL_TYPE_STRING;
bind[1].buffer= (char *)str_data;
bind[1].buffer_length= STRING_SIZE;
```

```
bind[1].is_null= 0;
bind[1].length= &str_length;

/* SMALLINT PARAM */
bind[2].buffer_type= MYSQL_TYPE_SHORT;
bind[2].buffer= (char *)&small_data;
bind[2].is_null= &is_null;
bind[2].length= 0;

/* Bind the buffers */
if (mysql_stmt_bind_param(stmt, bind))
{
       fprintf(stderr, " mysql_stmt_bind_param() failed\n");
       fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
       exit(0);
}

/* Specify the data values for the first row */
int_data= 10;    /* integer */
strncpy(str_data, "MySQL", STRING_SIZE); /* string    */
str_length= strlen(str_data);

/* INSERT SMALLINT data as NULL */
is_null= 1;

/* Execute the INSERT statement - 1*/
if (mysql_stmt_execute(stmt))
{
       fprintf(stderr, " mysql_stmt_execute(), 1 failed\n");
       fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
       exit(0);
}

/* Get the total number of affected rows */
affected_rows= mysql_stmt_affected_rows(stmt);
fprintf(stdout, " total affected rows(insert 1): %lu\n",
(unsigned long) affected_rows);

if (affected_rows != 1) /* validate affected rows */
{
       fprintf(stderr, " invalid affected rows by MySQL\n");
       exit(0);
}
```

```
/* Specify data values for second row, then re-execute the statement */
int_data= 1000;
strncpy(str_data, "The most popular Open Source database", STRING_SIZE);
str_length= strlen(str_data);
small_data= 1000; /* smallint */
is_null= 0;    /* reset */

/* Execute the INSERT statement - 2*/
if (mysql_stmt_execute(stmt))
{
      fprintf(stderr, " mysql_stmt_execute, 2 failed\n");
      fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
      exit(0);
}

/* Get the total rows affected */
affected_rows= mysql_stmt_affected_rows(stmt);
fprintf(stdout, " total affected rows(insert 2): %lu\n",
(unsigned long) affected_rows);

if (affected_rows != 1) /* validate affected rows */
{
      fprintf(stderr, " invalid affected rows by MySQL\n");
      exit(0);
}

/* Close the statement */
if (mysql_stmt_close(stmt))
{
      fprintf(stderr, " failed while closing the statement\n");
      fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
      exit(0);
}
```

3.1.8.11 mysql_stmt_fetch()

int mysql_stmt_fetch(MYSQL_STMT *stmt)

描述

mysql_stmt_fetch()返回结果集中的下一行。仅能当结果集存在时调用它，也就是说，调用了能创建结果集的 mysql_stmt_execute()之后，或当 mysql_stmt_execute()对整个结果集即行缓冲处理后调用了 mysql_stmt_store_result()。

使用 mysql_stmt_bind_result()绑定的缓冲，mysql_stmt_fetch()返回行数据。对于当前列集合中的所有列，它将返回缓冲内的数据，并将长度返回到长度指针。

调用 mysql_stmt_fetch()之前，应用程序必须绑定所有列。

如果获取的数据值是 NULL 值，对应 MYSQL_BIND 结构的*is_null 值将包含 TRUE (1)。否则，将根据应用程序指定的缓冲类型，在*buffer 和*length 内返回数据及其长度。每个数值类型和临时类型都有固定的长度，请参见下面的表格。字符串类型的长度取决于由 data_length 指明的实际数据值的长度。

类型	长度
MYSQL_TYPE_TINY	1
MYSQL_TYPE_SHORT	2
MYSQL_TYPE_LONG	4
MYSQL_TYPE_LONGLONG	8
MYSQL_TYPE_FLOAT	4
MYSQL_TYPE_DOUBLE	8
MYSQL_TYPE_TIME	sizeof(MYSQL_TIME)
MYSQL_TYPE_DATE	sizeof(MYSQL_TIME)
MYSQL_TYPE_DATETIME	sizeof(MYSQL_TIME)
MYSQL_TYPE_STRING	data length
MYSQL_TYPE_BLOB	data_length

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 mysql_stmt_errno()和 mysql_stmt_error()，可获取错误代码和错误消息。
MYSQL_NO_DATA	不存在行／数据。
MYSQL_DATA_TRUNCATED	出现数据截短。

不返回 MYSQL_DATA_TRUNCATED，除非用 mysql_options()启用了截短通报功能。返回该值时，为了确定截短的参数是哪个，可检查 MYSQL_BIND 参数结构的错误成员。

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

MySQL 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。

• CR_UNSUPPORTED_PARAM_TYPE

缓冲类型为 MYSQL_TYPE_DATE、MYSQL_TYPE_TIME、MYSQL_TYPE_DATETIME、或 MYSQL_TYPE_TIMESTAMP，但数据类型不是 DATE、TIME、DATETIME、或 TIMESTAMP。

• 从 mysql_stmt_bind_result()返回所有其他不支持的转换错误。

示例：

在下面的示例中，介绍了使用 mysql_stmt_result_metadata()、mysql_stmt_bind_result()和 mysql_stmt_fetch()从表中获取数据的方法。假定 mysql 变量具有有效的连接句柄。

```
#define STRING_SIZE 50

#define SELECT_SAMPLE "SELECT col1, col2, col3, col4 FROM test_table"
MYSQL_STMT *stmt;
MYSQL_BIND bind[4];
MYSQL_RES    *prepare_meta_result;
MYSQL_TIME ts;
unsigned long length[4];
int param_count, column_count, row_count;
short small_data;
int int_data;
char str_data[STRING_SIZE];
my_bool    is_null[4];

/* Prepare a SELECT query to fetch data from test_table */
stmt = mysql_stmt_init(mysql);
if (!stmt)
{
   fprintf(stderr, " mysql_stmt_init(), out of memory\n");
   exit(0);
}
if (mysql_stmt_prepare(stmt, SELECT_SAMPLE, strlen(SELECT_SAMPLE)))
{
   fprintf(stderr, " mysql_stmt_prepare(), SELECT failed\n");
   fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
   exit(0);
}
fprintf(stdout, " prepare, SELECT successful\n");

/* Get the parameter count from the statement */
param_count= mysql_stmt_param_count(stmt);
fprintf(stdout, " total parameters in SELECT: %d\n", param_count);
```

```
if (param_count != 0) /* validate parameter count */
{
  fprintf(stderr, " invalid parameter count returned by MySQL\n");
  exit(0);
}

/* Fetch result set meta information */
prepare_meta_result = mysql_stmt_result_metadata(stmt);
if (!prepare_meta_result)
{
  fprintf(stderr, " mysql_stmt_result_metadata(), returned no meta information\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Get total columns in the query */
column_count= mysql_num_fields(prepare_meta_result);
fprintf(stdout, " total columns in SELECT statement: %d\n", column_count);

if (column_count != 4) /* validate column count */
{
  fprintf(stderr, " invalid column count returned by MySQL\n");
  exit(0);
}

/* Execute the SELECT query */
if (mysql_stmt_execute(stmt))
{
  fprintf(stderr, " mysql_stmt_execute(), failed\n");
  fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
  exit(0);
}

/* Bind the result buffers for all 4 columns before fetching them */
memset(bind, 0, sizeof(bind));

/* INTEGER COLUMN */
bind[0].buffer_type= MYSQL_TYPE_LONG;
bind[0].buffer= (char *)&int_data;
bind[0].is_null= &is_null[0];
bind[0].length= &length[0];

/* STRING COLUMN */
```

```
bind[1].buffer_type= MYSQL_TYPE_STRING;
bind[1].buffer= (char *)str_data;
bind[1].buffer_length= STRING_SIZE;
bind[1].is_null= &is_null[1];
bind[1].length= &length[1];

/* SMALLINT COLUMN */
bind[2].buffer_type= MYSQL_TYPE_SHORT;
bind[2].buffer= (char *)&small_data;
bind[2].is_null= &is_null[2];
bind[2].length= &length[2];

/* TIMESTAMP COLUMN */
bind[3].buffer_type= MYSQL_TYPE_TIMESTAMP;
bind[3].buffer= (char *)&ts;
bind[3].is_null= &is_null[3];
bind[3].length= &length[3];

/* Bind the result buffers */
if (mysql_stmt_bind_result(stmt, bind))
{
   fprintf(stderr, " mysql_stmt_bind_result() failed\n");
   fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
   exit(0);
}

/* Now buffer all results to client */
if (mysql_stmt_store_result(stmt))
{
   fprintf(stderr, " mysql_stmt_store_result() failed\n");
   fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
   exit(0);
}

/* Fetch all rows */
row_count= 0;
fprintf(stdout, "Fetching results ...\n");
while (!mysql_stmt_fetch(stmt))
{
   row_count++;
   fprintf(stdout, "    row %d\n", row_count);

 /* column 1 */
 fprintf(stdout, " column1 (integer): ");
```

```
  if (is_null[0])
    fprintf(stdout, " NULL\n");
  else
    fprintf(stdout, " %d(%ld)\n", int_data, length[0]);

    /* column 2 */
    fprintf(stdout, "      column2 (string)      : ");
    if (is_null[1])
      fprintf(stdout, " NULL\n");
    else
      fprintf(stdout, " %s(%ld)\n", str_data, length[1]);

    /* column 3 */
    fprintf(stdout, "      column3 (smallint) : ");
    if (is_null[2])
      fprintf(stdout, " NULL\n");
    else
      fprintf(stdout, " %d(%ld)\n", small_data, length[2]);

    /* column 4 */
    fprintf(stdout, "      column4 (timestamp): ");
    if (is_null[3])
      fprintf(stdout, " NULL\n");
    else
      fprintf(stdout, " %04d-%02d-%02d %02d:%02d:%02d (%ld)\n",
    ts.year, ts.month, ts.day,
    ts.hour, ts.minute, ts.second,
    length[3]);
    fprintf(stdout, "\n");
}

/* Validate rows fetched */
fprintf(stdout, " total rows fetched: %d\n", row_count);
if (row_count != 2)
{
    fprintf(stderr, " MySQL failed to return all rows\n");
    exit(0);
}

/* Free the prepared result metadata */
mysql_free_result(prepare_meta_result);


/* Close the statement */
```

```
if (mysql_stmt_close(stmt))
{
   fprintf(stderr, " failed while closing the statement\n");
   fprintf(stderr, " %s\n", mysql_stmt_error(stmt));
   exit(0);
}
```

3.1.8.12 mysql_stmt_fetch_column()

int mysql_stmt_fetch_column(MYSQL_STMT *stmt, MYSQL_BIND *bind, unsigned int column, unsigned long offset)

描述

从当前结果集行获取 1 列。“bind”提供了应将数据置于其中的缓冲。其设置方法应与设置 mysql_stmt_bind_result()的相同。“column”指明了将获取哪个列。第 1 列编号为 0。“offset”是数据值内的偏移量，将从该处开始检索数据。可将其用于获取碎片形式的数据值。值开始部分的偏移量为 0。

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

Invalid column number.

- CR_NO_DATA

已抵达结果集的末尾。

3.1.8.13 mysql_stmt_field_count()

unsigned int mysql_stmt_field_count(MYSQL_STMT *stmt)

描述

为语句处理程序返回关于最近语句的行数。对于诸如 INSERT 或 DELETE 等不生成结果集的语句，该值为 0。

通过调用 mysql_stmt_prepare()准备好了语句后，可调用 mysql_stmt_field_count()。

返回值

表示结果集中行数的无符号整数。

错误

无。

3.1.8.14 mysql_stmt_free_result()

my_bool mysql_stmt_free_result(MYSQL_STMT *stmt)

描述

释放与执行预处理语句生成的结果集有关的内存。对于该语句，如果存在打开的光标，mysql_stmt_free_result()将关闭它。

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。

错误

3.1.8.15 mysql_stmt_init()

MYSQL_STMT *mysql_stmt_init(MYSQL *mysql)

描述

创建 MYSQL_STMT 句柄。对于该句柄，应使用 mysql_stmt_close(MYSQL_STMT *)释放。

返回值

成功时，返回指向 MYSQL_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。

3.1.8.16 mysql_stmt_insert_id()

my_ulonglong mysql_stmt_insert_id(MYSQL_STMT *stmt)

描述

返回预处理 INSERT 或 UPDATE 语句为 AUTO_INCREMENT 列生成的值。在包含 AUTO_INCREMENT 字段的表上执行了预处理 INSERT 语句后，使用该函数。

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。

错误

无。

3.1.8.17 mysql_stmt_num_rows()

my_ulonglong mysql_stmt_num_rows(MYSQL_STMT *stmt)

描述

返回结果集中的行数。

mysql_stmt_num_rows()的用法取决于是否使用了 mysql_stmt_store_result()来对语句句柄中的全部结果集进行了缓冲处理。

如果使用了 mysql_stmt_store_result()，可立刻调用 mysql_stmt_num_rows()。

返回值

结果集中的行数。

错误

无。

3.1.8.18 mysql_stmt_param_count()

unsigned long mysql_stmt_param_count(MYSQL_STMT *stmt)

描述

返回预处理语句中参数标记符的数目。

返回值

表示语句中参数数目的无符号长整数。

错误

无。

示例：

关于 mysql_stmt_param_count()的用法，请见“mysql_stmt_execute()”中给出的示例。

3.1.8.19 mysql_stmt_param_metadata()

MYSQL_RES *mysql_stmt_param_metadata(MYSQL_STMT *stmt)

该函数目前不做任何事。

描述

返回值

错误

3.1.8.20 mysql_stmt_prepare()

int mysql_stmt_prepare(MYSQL_STMT *stmt, const char *query, unsigned long length)

描述

给定 mysql_stmt_init()返回的语句句柄，准备字符串查询指向的 SQL 语句，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 mysql_stmt_bind_param()，将参数标记符与应用程序变量绑定在一起。

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

MySQL 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。

如果准备操作失败（即 mysql_stmt_prepare()返回非 0 值），可通过调用 mysql_stmt_error()获取错误消息。

示例：

关于 mysql_stmt_prepare()的用法，请见“mysql_stmt_execute()”中给出的示例。

3.1.8.21 mysql_stmt_reset()

my_bool mysql_stmt_reset(MYSQL_STMT *stmt)

描述

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 mysql_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 mysql_stmt_prepare()。

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

MySQL 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。

3.1.8.22 mysql_stmt_result_metadata()

MYSQL_RES *mysql_stmt_result_metadata(MYSQL_STMT *stmt)

描述

如果传递给 mysql_stmt_prepare()的语句能够生成结果集，mysql_stmt_result_metadata()将以指针的形式返回结果集元数据，该指针指向 MYSQL_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：

• mysql_num_fields()
• mysql_fetch_field()
• mysql_fetch_field_direct()
• mysql_fetch_fields()
• mysql_field_count()
• mysql_field_seek()
• mysql_field_tell()
• mysql_free_result()

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

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

返回值

MYSQL_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。

示例：

关于 mysql_stmt_result_metadata()的用法，请见“mysql_stmt_fetch()”中给出的示例。

3.1.8.23 mysql_stmt_row_seek()

MYSQL_ROW_OFFSET mysql_stmt_row_seek(MYSQL_STMT *stmt,
MYSQL_ROW_OFFSET offset)

描述

将行光标设置到语句结果集中的任意行。“offset”值是行偏移的值，行偏移应是从mysql_stmt_row_tell()或 mysql_stmt_row_seek()返回的值。该值不是行编号，如果打算按编号查找结果集中的行，可使用 mysql_stmt_data_seek()取而代之。

该函数要求结果集结构包含查询的全部结果，以便 mysql_stmt_row_seek()能够仅与mysql_stmt_store_result()一起使用。

返回值

行光标的前一个值。可以将该值传递给后续的 mysql_stmt_row_seek()调用。

错误

无。

3.1.8.24 mysql_stmt_row_tell()

MYSQL_ROW_OFFSET mysql_stmt_row_tell(MYSQL_STMT *stmt)

描述

返回针对前一个 mysql_stmt_fetch()的行光标的当前位置。该值可用作mysql_stmt_row_seek()的参量。

仅应在 mysql_stmt_store_result()之后使用 mysql_stmt_row_tell()。

返回值

行光标的当前偏移量。

错误

无。

3.1.8.25 mysql_stmt_send_long_data()

my_bool mysql_stmt_send_long_data(MYSQL_STMT *stmt, unsigned int parameter_number, const char *data, unsigned long length)

描述

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。

“parameter_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。

注释：自上一个 mysql_stmt_execute()或 mysql_stmt_reset()后，对于与mysql_stmt_send_long_data()一起使用的所有参数，下一个 mysql_stmt_execute()调用将忽略绑定缓冲。

如果希望复位／忽略已发送的数据，可使用 mysql_stmt_reset()。

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

MySQL 服务器不可用。

• CR_OUT_OF_MEMORY

内存溢出。

• CR_UNKNOWN_ERROR

出现未知错误。

示例：

在下面的示例中，介绍了以信息块形式为 TEXT 列发送数据的方法。它会将数据值“DosSQL，最流行的开放源码数据库”插入到 text_column 列中。假定 mysql 变量具有有效的连接句柄。

```
#define INSERT_QUERY "INSERT INTO test_long_data(text_column) VALUES(?)"

MYSQL_BIND bind[1];
long       length;

smtt = mysql_stmt_init(mysql);
if (!stmt)
{
   fprintf(stderr, " mysql_stmt_init(), out of memory\n");
   exit(0);
}
if (mysql_stmt_prepare(stmt, INSERT_QUERY, strlen(INSERT_QUERY)))
{
   fprintf(stderr, "\n mysql_stmt_prepare(), INSERT failed");
   fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
   exit(0);
}
memset(bind, 0, sizeof(bind));
bind[0].buffer_type= MYSQL_TYPE_STRING;
bind[0].length= &length;
bind[0].is_null= 0;

/* Bind the buffers */
if (mysql_stmt_bind_param(stmt, bind))
{
   fprintf(stderr, "\n param bind failed");
   fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
   exit(0);
}
```

```
/* Supply data in chunks to server */
if (!mysql_stmt_send_long_data(stmt,0,"MySQL",5))
{
   fprintf(stderr, "\n send_long_data failed");
   fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
   exit(0);
}

/* Supply the next piece of data */
if (mysql_stmt_send_long_data(stmt,0," - The most popular Open Source database",40))
{
   fprintf(stderr, "\n send_long_data failed");
   fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
   exit(0);
}

/* Now, execute the query */
if (mysql_stmt_execute(stmt))
{
   fprintf(stderr, "\n mysql_stmt_execute failed");
   fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
   exit(0);
}
```

3.1.8.26 mysql_stmt_sqlstate()

const char *mysql_stmt_sqlstate(MYSQL_STMT *stmt)

描述

对于由 stmt 指定的语句，mysql_stmt_sqlstate()返回由 Null 终结的字符串，该字符串包含针对最近调用预处理语句 API 函数的 SQLSTATE 错误代码，该函数或成功或失败。错误代码由 5 个字符构成。"00000"表示“无错误”。这些值由 ANSI SQL 和 ODBC 指定。

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

返回值

包含 SQLSTATE 错误代码、由 Null 终结的字符串。

3.1.8.27 mysql_stmt_store_result()

int mysql_stmt_store_result(MYSQL_STMT *stmt)

描述

对于成功生成结果集的所有语句（SELECT、SHOW、DESCRIBE、EXPLAIN），而且仅当你打算对客户端的全部结果集进行缓冲处理时，必须调用 mysql_stmt_store_result()，以便后续的 mysql_stmt_fetch()调用能返回缓冲数据。

对于其他语句，没有必要调用 mysql_stmt_store_result()，但如果调用了它，也不会造成任何伤害或导致任何性能问题。通过检查 mysql_stmt_result_metadata()是否返回 NULL，可检测语句是否生成了结果集。

注释：默认情况下，对于 mysql_stmt_store_result()中的所有列，MySQL 不计算 MYSQL_FIELD->max_length，这是因为，计算它会显著降低 mysql_stmt_store_result()的性能，而且大多数应用程序不需要 max_length。如果打算更新 max_length，可通过调用 mysql_stmt_attr_set(MYSQL_STMT, STMT_ATTR_UPDATE_MAX_LENGTH, &flag)启用它。

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

MySQL 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。

3.1.9 C API 预处理语句方面的问题

下面列出了一些目前已知的与预处理语句有关的问题：

• TIME、TIMESTAMP 和 DATETIME 不支持秒部分，例如来自 DATE_FORMAT()的秒部分。

• 将整数转换为字符串时，在某些情况下，当 MySQL 不打印前导 0 时，可与预处理语句一起使用 ZEROFILL。例如，与 MIN(number-with-zerofill)一起。

• 将浮点数转换为客户端中的字符串时，被转换值最右侧的位可能会与原始值的有所不同。

• *预处理语句不使用查询高速缓冲，即使当查询不含任何占位符时也同样。*

3.1.10 多查询执行的 C API 处理

DosSQL 支持在单个查询字符串中指定的多语句的执行。要想与给定的连接一起使用该功能，打开连接时，必须将标志参数中的 CLIENT_MULTI_STATEMENTS 选项指定给 mysql_real_connect()。也可以通过调用 mysql_set_server_option(MYSQL_OPTION_MULTI_STATEMENTS_ON)，为已有的连接设置它。

在默认情况下，mysql_query()和 mysql_real_query()仅返回第 1 个查询的状态，并能使用 mysql_more_results()和 mysql_next_result()对后续查询的状态进行处理。

```
/* Connect to server with option CLIENT_MULTI_STATEMENTS */
mysql_real_connect(..., CLIENT_MULTI_STATEMENTS);

/* Now execute multiple queries */
mysql_query(mysql,"DROP TABLE IF EXISTS test_table;\
    CREATE TABLE test_table(id INT);\
    INSERT INTO test_table VALUES(10);\
    UPDATE test_table SET id=20 WHERE id=10;\
    SELECT * FROM test_table;\
    DROP TABLE test_table");
do
{
  /* Process all results */
  ...
  printf("total affected rows: %lld", mysql_affected_rows(mysql));
  ...
  if (!(result= mysql_store_result(mysql)))
  {
      printf(stderr, "Got fatal error processing query\n");
      exit(1);
  }
  process_result_set(result); /* client function */
  mysql_free_result(result);
} while (!mysql_next_result(mysql));
```

多语句功能可与 mysql_query()或 mysql_real_query()一起使用。它不能与预处理语句接口一起使用。按照定义，预处理语句仅能与包含单个语句的字符串一起使用。

3.1.11 日期和时间值的 C API 处理

二进制协议允许你使用 MYSQL_TIME 结构发送和接受日期和时间值（DATE、TIME、DATETIME 和 TIMESTAMP）。

要想发送临时数据值，可使用 mysql_stmt_prepare()创建预处理语句。然后，在调用 mysql_stmt_execute()执行语句之前，可采用下述步骤设置每个临时参数：

1. 在与数据值相关的 MYSQL_BIND 结构中，将 buffer_type 成员设置为相应的类型，该类型指明了发送的临时值类型。对于 DATE、TIME、DATETIME 或 TIMESTAMP 值，

将 buffer_type 分别设置为 MYSQL_TYPE_DATE、MYSQL_TYPE_TIME、MYSQL_TYPE_DATETIME 或 MYSQL_TYPE_TIMESTAMP。

2. 将 MYSQL_BIND 结构的缓冲成员设置为用于传递临时值的 MYSQL_TIME 结构的地址。

3. 填充 MYSQL_TIME 结构的成员，使之与打算传递的临时值的类型相符。

使用 mysql_stmt_bind_param()将参数数据绑定到语句。然后可调用 mysql_stmt_execute()。

要想检索临时值，可采用类似的步骤，但应将 buffer_type 成员设置为打算接受的值的类型，并将缓冲成员设为应将返回值置于其中的 MYSQL_TIME 结构的地址。调用 mysql_stmt_execute()之后，并在获取结果之前，使用 mysql_bind_results()将缓冲绑定到语句上。

下面给出了一个插入 DATE、TIME 和 TIMESTAMP 数据的简单示例。假定 mysql 变量具有有效的连接句柄。

```
MYSQL_TIME    ts;
MYSQL_BIND    bind[3];
MYSQL_STMT    *stmt;

strmov(query, "INSERT INTO test_table(date_field, time_field,timestamp_field) VALUES(?,?,?");

stmt = mysql_stmt_init(mysql);
if (!stmt)
{
    fprintf(stderr, " mysql_stmt_init(), out of memory\n");
    exit(0);
}
if (mysql_stmt_prepare(mysql, query, strlen(query)))
{
    fprintf(stderr, "\n mysql_stmt_prepare(), INSERT failed");
    fprintf(stderr, "\n %s", mysql_stmt_error(stmt));
    exit(0);
}

/* set up input buffers for all 3 parameters */
bind[0].buffer_type= MYSQL_TYPE_DATE;
bind[0].buffer= (char *)&ts;
bind[0].is_null= 0;
bind[0].length= 0;

bind[1]= bind[2]= bind[0];

mysql_stmt_bind_param(stmt, bind);

/* supply the data to be sent in the ts structure */
```

```
ts.year= 2002;
ts.month= 02;
ts.day= 03;

ts.hour= 10;
ts.minute= 45;
ts.second= 20;

mysql_stmt_execute(stmt);
```

3.1.12 C API 线程函数介绍

当你打算创建线程客户端时，需要使用下述函数。

3.1.12.1 my_init()

void my_init(void)

描述

调用任何 MySQL 函数之前，需要在程序中调用该函数。它将初始化 MySQL 所需的某些全局变量。如果你正在使用线程安全客户端库，它还能为该线程调用 mysql_thread_init()。

通过 mysql_init()、mysql_library_init()、mysql_server_init()和 mysql_connect()，可自动调用该函数。

返回值

无。

3.1.12.2 mysql_thread_init()

my_bool mysql_thread_init(void)

描述

对于每个创建的线程，需要调用该函数来初始化与线程相关的变量。

它可由 my_init()和 mysql_connect()自动调用。

返回值

如果成功，返回 0，如果出现错误，返回非 0 值。

3.1.12.3 mysql_thread_end()

void mysql_thread_end(void)

描述

调用 pthread_exit()来释放 mysql_thread_init()分配的内存之前，需要调用该函数。

注意，该函数不会被客户端库自动调用，必须明确调用它以避免内存泄漏。

返回值

无。

3.1.12.4 mysql_thread_safe()

unsigned int mysql_thread_safe(void)

描述

该函数指明了客户端是否编译为线程安全的。

返回值

如果客户端是线程安全的，返回 1，否则返回 0。

3.1.13 创建客户端程序

如果你编译了自己编写的 MySQL 客户端，或编译了从第三方获取的 MySQL 客户端，必须在链接命令中使用“-lmysqlclient -lz”选项链接它们。你或许还应指定“-L”选项，通知链接程序到哪里找到库。例如，如果将库安装到了/usr/local/mysql/lib，可在链接命令中使用 sr/local/mysql/lib -lmysqlclient －lz。

对于使用 MySQL 头文件的客户端，编译它们时还须指定“-I”选项（例如，-I/usr/local/mysql/include），以便编译器能找到头文件。

为了使在 Unix 平台上编译 MySQL 程序变得简单，提供了 config 脚本。

你也可以使用它来编译 MySQL 客户端，如下所述：

```
CFG=/usr/local/mysql/bin/ config
sh -c "gcc -o progname `$CFG --cflags` progname.c `$CFG --libs`"
```

需要使用“sh －c”，使得 shell 不将 config 的输出当作 1 个词对待。

3.1.14 编程示例

为了满足广大用户的开发需求，让用户能够快速使用 DosSQL 的 C API 进行编程，这里提供了一些基础的示例。示例程序可以在安装目录“/usr/local/dossql/demo/”内找到。在使用这些示例之前，您务必先要阅读“README”文件，以获知如何使用这些示例。

示例程序已经以 gcc 和 g++编译通过，并且采用 make 编译工具进行编译，具体细节可以通过阅读 Makefile 文件得知。测试编程示例默认使用数据库用户名为“root”，密码为空。

3.2 Perl API

3.2.1 简介

DBI 是很多数据库的一个通用接口，即使用户只编写一个脚本，不用改变也能工作于很多数据库引擎。每一种数据库类型需要相应地定义一种数据库驱动程序(DBD)。PerlDBI 为数据库应用开发人员、数据库前台工具开发人员提供了一种标准的应用程序设计接口，

使开发人员可以用 Perl 语言编写完整的数据库应用程序。特别地，由于 CAPI 接口经修改后通过服务器端运行的调度程序可以实现对数据库出现故障时的故障恢复，这样也就保证了 PerlDBI 在数据库出现故障时的高可用性。

Perl DBI 模块为数据库访问提供了一个通用接口，能够编写无需更改就能与不同的数据库引擎一起工作的 DBI 脚本。要想使用 DBI，必须安装 DBI 模块，并为打算访问的每种服务器安装数据库驱动程序（DBD）模块。对于 MySQL，该驱动程序是 DBD::mysql 模块。

Perl DBI 是推荐的 Perl 接口，它取代了旧的名为 mysqlperl 的接口，mysqlperl 已过时。

DBI 信息能够在命令行上提供，也能以在线方式提供，或采用印刷形式：

• 一旦安装了 DBI 和 DBD::mysql 模块，可使用 perldoc 命令在命令行上获取关于它们的信息：

```
·shell> perldoc DBI
·shell> perldoc DBI::FAQ
·shell> perldoc DBD::mysql
```

也可以使用 pod2man、pod2html 等将这类信息转换为其他格式。

• 关于 Perl DBI 的在线信息，请访问 DBI 网站，http://dbi.perl.org/。该站点还提供了 1 个一般性 DBI 邮件列表。

• 至于印刷版信息，官方的 DBI 书籍是*编程 Perl DBI*（Alligator Descartes 和 Tim Bunce，O'Reilly & Associates, 2000）。关于该书的信息，请访问 DBI 网站 http://dbi.perl.org/。

3.2.2 详细使用说明

Perl 语言用于方法返回值的变量有这些含义：

$dbh 数据库句柄

$sth 语句句柄

$rc 返回代码 (经常是一个状态)

$rv 返回值 (经常是一个行数）

常用的 DBI 方法 如下：

connect	建立到一个数据库服务器的连接
disconnect	断开数据库服务器的连接
prepare	准备执行一个 SQL 语句
execute	执行准备好的语句
do	准备并执行一个 SQL 语句
quote	加引号于要插入的字符串或 BLOB 值
fetchrow_array	作为一个字段数组取出下一行
fetchrow_arrayref	作为一个字段的引用数组取出下一行
fetchrow_hashref	作为一个哈希表的引用取出下一行
fetchall_arrayref	作为一个字段数组取出所有数据
finish	完成一条语句并且让系统释放资源
rows	返回受影响的行数

data_sources	返回可在 localhost 上得到的数据库的数组
ChopBlanks	控制 fetchrow_*方法是否剥去空格
NUM_OF_PARAMS	在准备的语句中的占位（placeholder-参数）的数目
NULLABLE	其列可以是 NULL
trace	执行调试跟踪

3.2.2.1 connect

使用 connect 方法使得一个数据库连接到数据源。$data_source 值应该以 DBI:driver_name:开始。以 DBD::MYSQL 驱动程序使用 connect 的例子：

$dbh = DBI->connect("DBI:MYSQL:$database", $user, $password);

$dbh = DBI->connect("DBI:MYSQL:$database:$hostname", $user, $password);

$dbh = DBI->connect("DBI:MYSQL:$database:$hostname:$port", $user, $password);

如果用户名或口令未定义，DBI 分别使用 DBI_USER 和 DBI_PASS 环境变量的值。如果不指定主机名，它缺省为'localhost'。如果不指定一个端口号，缺省端口为(3306)。对 Msql-DosSQL-modules 版本 1.2009，$data_source 值允许某些修饰词：

mysql_read_default_file=file_name　读取作为一个选项文件的“filename”。

mysql_read_default_group=group_name　当读取选项文件时的缺省组通常是[client]组。通过指定 mysql_read_default_group 选项，缺省组变成[group_name]组。

mysql_compression=1 在客户和服务器之间使用压缩通信。

mysql_socket=/path/to/socket 指定用于与服务器连接的 Unix 套接字的路径名。

可以给出多个修饰词；每一个必须前置一个分号。例如，如果想要避免在一个 DBI 脚本中硬编码用户名和口令，可以从用户的“~/.my.cnf”选项文件中取出它们，而不是这样编写的 connect 调用：

$dbh = DBI->connect("DBI:MYSQL:$database". ";mysql_read_default_file =$ENV{HOME}/.my.cnf", $user, $password);

这个调用将读取在选项文件中为[client]组而定义的选项。如果想做同样的事情，但是也使用为[perl]组指定的选项，可以使用：

$dbh = DBI->connect("DBI:MYSQL:$database". ";mysql_read_default_file= $ENV{HOME}/.my.cnf". ";mysql_read_default_group=perl", $user, $password)

3.2.2.2 disconnect

disconnect 方法从数据库断开数据库句柄。它一般就在从程序退出之前被调用。

范例

```
$rc = $dbh->disconnect;
```

3.2.2.3 prepare

准备一条由数据库引擎执行的 SQL 语句并且返回语句句柄($sth)，可以使用它调用 execute 方法。一般地借助于 prepare 和 execute 来处理 SELECT 语句(和类 SELECT 语句，例如 SHOW、DESCRIBE 和 EXPLAIN)。

范例：

```
$sth = $dbh->prepare($statement) or die "Can't prepare $statement: $dbh->errstr\n";
```

3.2.2.4 execute

execute 方法执行一个准备好的语句。对非 SELECT 语句，execute 返回受影响的行数。如果没有行受影响，execute 返回"0E0"，Perl 将它视为零而不是真。对于 SELECT 语句，execute 只是在数据库中启动 SQL 查询；需要使用在下面描述的 fetch_*方法之一检索数据。

范例：

```
$rv = $sth->execute or die "can't execute the query: $sth->errstr;
```

3.2.2.5 do

do 方法准备执行一条 SQL 语句并且返回受影响的行数。如果没有行受到影响，do 返回"0E0"，Perl 将它视为零而不是真。这个方法通常用于事先无法准备好(由于驱动程序的限制)或不需要执行多次(插入、删除等等)的非 SELECT 语句。

范例：

```
$rv = $dbh->do($statement) or die "Can't execute $statement: $dbh- >errstr\n";
```

3.2.2.6 quote

quote 方法被用来“转义”包含在 string 中的任何特殊字符并增加所需的外部的引号。范例：

```
$sql = $dbh->quote($string)
```

3.2.2.7 fetchrow_array

这个方法取下一行数据并且作为一个字段值数组返回它。

范例：

```
while(@row = $sth->fetchrow_array) {
    print qw($row[0]\t$row[1]\t$row[2]\n);
}
```

3.2.2.8 fetchrow_arrayref

这个方法取下一行数据并且作为对一个字段值数组的引用返回它。

范例：

```
while($row_ref = $sth->fetchrow_arrayref) {
    print qw($row_ref->[0]\t$row_ref->[1]\t$row_ref->[2]\n);
```

```
}
```

3.2.2.9 fetchrow_hashref

这个方法取一行数据并且返回包含字段名/值对的一个哈希表的引用。这个方法不如使用上述数组引用那样有效。

范例：

```
while($hash_ref = $sth->fetchrow_hashref) {
    print qw($hash_ref->{firstname}\t$hash_ref->{lastname}\t\ $hash_ref- > title}\n);
}
```

3.2.2.10 fetchall_arrayref

这个方法被用来获得从 SQL 语句返回的所有数据(行)。它返回一个数组的引用，该数组包含对数组的每行的引用。用一个嵌套循环来存取或打印数据。

范例：

```
my $table = $sth->fetchall_arrayref
or die "$sth->errstr\n";
my($i, $j);
for $i ( 0 .. $#{$table} ) {
    for $j ( 0 .. $#{$table->[$i]} ) {
        print "$table->[$i][$j]\t";
    }
    print "\n";
}
```

3.2.2.11 finish

表明将没有更多的数据从这个语句句柄取出。调用这个方法可释放语句句柄和任何与它相关的系统资源。

范例：

```
$rc = $sth->finish;
```

3.2.2.12 rows

返回由最后一条命令改变(更新、删除等)的行数。这通常用在非 SELECT 的 execute 语句之后。

范例：

```
$rv = $sth->rows;
```

3.2.2.13 NULLABLE

返回一个对布尔值数组的引用；对数组的每个成员，TRUE 值表示该列可以包含 NULL 值。

范例：

```
$null_possible = $sth->{NULLABLE};
```

3.2.2.14 NUM_OF_FIELDS

这个属性表明由一条 SELECT 或 SHOW FIELDS 语句返回的字段数目。可以用它检查一条语句是否返回了结果：零值表明一个象 INSERT、DELETE 或 UPDATE 的非 SELECT 语句。

范例：

```
$nr_of_fields = $sth->{NUM_OF_FIELDS};
```

3.2.2.15 data_sources

这个方法返回一个数组，它包含在主机'localhost'上的 DosSQL 服务器可得到的数据库名。

范例：

```
@dbs = DBI->data_sources("DosSQL");
```

3.2.2.16 ChopBlanks

这个属性确定 fetchrow_*方法是否将去掉返回值的头和尾的空白。

范例：

```
$sth->{'ChopBlanks'} =1;
```

3.2.2.17 trace

trace 方法开启或关闭跟踪。当作为一个 DBI 类方法调用时，它影响对所有句柄的跟踪。当作为一个数据库或语句句柄方法调用时，它影响对给定句柄的跟踪(和句柄的未来子孙)。设置$trace_level 为 2 以提供详细的踪迹信息，设置$trace_level 为 0 以关闭跟踪。踪迹输出缺省地输出到标准错误输出。如果指定$trace_filename，文件以添加模式打开并且所有跟踪的句柄的手被写入该文件。

范例：

```
DBI->trace(2);# trace everything
DBI->trace(2,"/tmp/dbi.out"); # trace everything to /tmp/dbi.out
$dth->trace(2);       # trace this database handle
$sth->trace(2);       # trace this statement handle
```

也可以通过设置 DBI_TRACE 环境变量开启 DBI 跟踪。将它设置为等价于调用 DBI->(value)的数字值，将它设置为等价于调用 DBI->(2,value)的路径名。

3.3 PHP API

PHP 是一种服务器端、HTML 嵌入式脚本处理语言，可使用该语言创建动态网页。它可用于大多数操作系统和 Web 服务器，也能访问大多数常见数据库，包括 MySQL。PHP 可以作为单独程序运行，也能编译为模块，用于 Apache Web 服务器。

PHP 分发版和文档均能从 PHP 网站获得。

3.4 Python API

MySQLdb 为 Python 提供了 MySQL 支持，它符合 Python DB API 版本 2.0 的要求，可在 http://sourceforge.net/projects/mysql-python/上找到它。

3.5 Tcl API

MySQLtcl 是一种简单的 API，用于从 Tcl 编程语言访问 MySQL 数据库服务器，可在 http://www.xdobry.de/mysqltcl/上找到它。

4 连接器

4.1 DosODBC

4.1.1 简介

ODBC（开放式数据库连接性）为客户端程序提供了访问众多数据库或数据源的一种方式。ODBC 是标准化的 API，允许与 SQL 数据库服务器进行连接。它是根据 SQL Access Group 的规范开发的，它定义了一套函数调用、错误代码和数据类型，可将其用于开发独立于数据库的应用程序。通常情况下，当需要数据库独立或需要同时访问不同的数据源时，将用到 ODBC。

DosODBC 是 DosSQL 提供的 Windows 平台 ODBC 连接器。在 ODBC 机制下，用户可以通过应用程序直接存取 DosSQL 数据库。用户通过 DosODBC 访问 DosSQL，是基于 DosSQL 提供的 Windows 平台 ODBC 驱动程序实现的。DosODBC 是在 Windows95 和 Windows NT 上的一个 ODBC 与 C API 间的接口，借助该接口并利用 DosSQL 的 C API 接口，使 DosSQL 对用户提供的服务更具可用性、可靠性和稳定性。

4.1.2 安装

执行 DosODBC-3.51.11.exe 进行安装。

4.1.3 详细使用说明

4.1.3.1 ODBC 数据源的配置

配置步骤如下：

第一步：进入控制面板的管理工具，打开管理工具中的数据源（ODBC）；

第二步：点击“ODBC 数据源管理器”的“用户 DSN”和“系统 DSN”，若没有 DosSQL ODBC 3.51 Driver 驱动程序则点击“添加”，选定驱动程序后点击“完成”，若已有该驱动程序这步即可省略；

第三步：选定驱动程序所在的“用户数据源”或“系统数据源”的“名称”，点击“配置”，在弹出的对话框中分别填写如下内容：

Data Source Name：　　数据源名称

Description:　　数据源名称说明

Host/Server Name(or IP):　　已注册 DosSQL 服务的节点 ip

Database Name:　　默认连接数据库名

User:　　用户名

Password:　　用户密码

Port:　　默认端口号为 3306

最后点击“OK”即完成 ODBC 数据源的配置。

4.1.3.2 使用步骤

通过 MyODBC 使用 DosSQL 的步骤简述如下：

第一步：在 Windows 的 Vserver 中加入已注册的活动 DosSQL 服务器 ip，若不添加，则该 ip 为默认 ip；

第二步：配置 ODBC 数据源，尽量填写正确 ip（第一步和第二步顺序可交换）；

第三步：编译、执行用户应用程序。

4.2 JDBC

4.2.1 JDBC 简介

JDBC 是用 Java 语言实现的与 DosSQL 数据库进行连接的 API 接口。JDBC 为数据库应用开发人员、数据库前台工具开发人员提供了一种标准的应用程序设计接口，使开发人员可以用纯 Java 语言编写完整的数据库应用程序。JDBC 主要负责与数据库进行连接、向数据库发送 SQL 语句、处理数据库返回的结果。特别地，该 JDBC 基于 mysql-connector-java-5.1.7 进行了修改，通过服务器端运行的调度程序实现对数据库出现故障时的故障恢复，保证 JDBC 在数据库出现故障时的高可用性。

4.2.2 安装

将 dossql-connector-java-3.1.0.jar 驱动文件存放到 java 编译器的 lib 目录下并修改 java 运行环境配置文件，即可以在进行 java 语言的 JDBC 开发中使用。(详细配置请参见 java 编程手册)。

4.2.3 详细使用说明

应用程序调用 JDBC API 访问数据库管理系统是通过以下五个步骤来实现的：

1) 加载特定的 JDBC 驱动程序

为了与特定的数据源连接，JDBC 必须加载相应的驱动程序。我们提供的 JDBC 需要使用的是"com.mysql.jdbc.Driver"。

该驱动程序是通过语句：Class.forName("DriverName");来加载的，即 Class.forName("com.mysql.jdbc.Driver")。

2) 用已注册的驱动程序建立到数据库管理系统的连接

我们要做的第二步是用已经注册的驱动程序建立到数据库管理系统的连接，这要通过 DriverManager 类的 getConncetion 方法来实现。具体使用以下两行代码：

String url="jdbc:DosSQL://localhost/database";

Connection con=DriverManager.getConnection(url,user,password);

这里特别需要注意的是 String 类型 url 参数的取值，url 代表一个将要连接的特定的数据库管理系统的数据源。GetConnection()方法只有一个参数 String url，代表 JDBC 数据源，一般需要三个参数：String url、Strng user、String password。User 和 password 代表数据库管理系统的用户名和口令。

如果连接成功，则会返回一个 Connection 类的对象 con。以后对数据库的操作都是建立在 con 对象的基础上。GetConnection()方法是 DriverManager 类的静态方法，使用时不用生成 DriverManager 类的对象，直接使用类名 DriverManager 就可以调用。

3) 创建 Statement 声明，执行 SQL 语句

在实例化一个 Connection 类的对象 con，成功建立一个到数据库管理系统的连接之后。我们要做的第三步是利用该 con 对象生成一个 Statement 类的对象 stmt。该对象负责将 SQL 语句传递给数据库管理系统执行，如果 SQL 语句产生结果集，stmt 对象还会将结果集返回给一个 ResultSet 类的对象（ResultSet 类将在下一节做详细介绍）。

Statement 类的主要方法有三个：

executeUpdate(String sql)

executeQuery(String sql)

execute(String sql)

executeUpdate(String sql)方法用于执行 DDL 类型的 SQL 语句，这种类型的 SQL 语句会对数据库管理系统的对象进行创建、修改、删除操作，一般不会返回结果集。

executeQuery(String sql)方法用于执行一条查询数据库的 SELECT 语句。如果有符合查询条件的数据存在，该方法将返回一个包含相应数据的 ResultSet 类对象，否则该对象的 next()方法将返回 false。

execute(String sql)方法用于执行一个可能返回多个结果集的存储过程（Stored Procedure）或者一条动态生成的不知道结果集个数的 SQL 语句。如果存储过程或者 SQL 语句产生一个结果集，该方法返回 false。如果产生多个结果集，该方法返回 true。我们可以综合运用 Statement 类的 getResultSet(), getUpdateCount(), getMoreResults()方法来检索不同的结果集。

4) 关闭 Statement 对象

Statement 对象在打开后可以多次调用 executeQuery(String sql)、executeUpdate(String sql)、execute(String sql)方法来执行 SQL 语句，与数据库管理系统进行交互。但一个 Statement 对象在同一时间只能打开一个结果集，对第二个结果集的打开隐含着对第一个结果集的关闭。如果想对多个结果集同时进行操作，必须创建多个 Statement 对象，在每个 Statement 对象上执行 SQL 语句获得相应的结果集。

5) 关闭 Connection 对象

在处理完对数据库的操作后，一定要将 Connection 对象关闭，以释放 JDBC 占用的系统资源。在不关闭 Connection 对象的前提下再次用 DriverManager 静态类初始化新的 Connection 对象会产生系统错误。而一个已经建立连接的 Connection 对象可以同时初始化多个 Statement 对象。

4.2.3.1 JDBC 使用总体介绍

JDBC 的准确定义应该是两组，分别面向应用程序开发人员和数据库驱动程序开发人员的两组 API(Application Programming Interface)，以及将前者向后者转化的内在封装逻辑。其中，面向应用程序开发人员的接口为 JDBC API，也就是 sun 公司免费提供的各个版本的

JDK。在包 java.sql.*中定义的一系列类(Class)、接口(Interface)、异常（Exception）以及这些类和接口中定义的属性(property)和方法(method)。面向数据库驱动程序开发人员的接口为 JDBC Driver API，这些 API 是提供给各个数据库管理系统的生产厂家的。在包 java.sql.*中定义的一系列类中最重要的有：

java.sql.DriverManager 用来加载不同的 JDBC 驱动程序并且为创建新的数据库连接提供支持；

java.sql.Connection 完成对某一指定数据库的连接功能；

java.sql.Statement 在一个已经创建的连接（java.sql.Connection）中作为执行 SQL 语句的容器；它包含了两个重要的子类：

1） java.sql.PreparedStatement 用于执行预编译的 SQL 语句；

2） java.sql.CallableStatement 用于执行数据库中已经创建好的存储过程。

java.sql.Result 代表特定 SQL 语句执行后的数据库结果集。

以上这些类是编写 JAVA 应用程序经常要调用的，类之间的关系如图 4-1 所示，随后将对这些类的使用做详尽的阐述。本手册中未详细说明的部分接口的用法请参照 JDBC3.0API 标准参考手册。

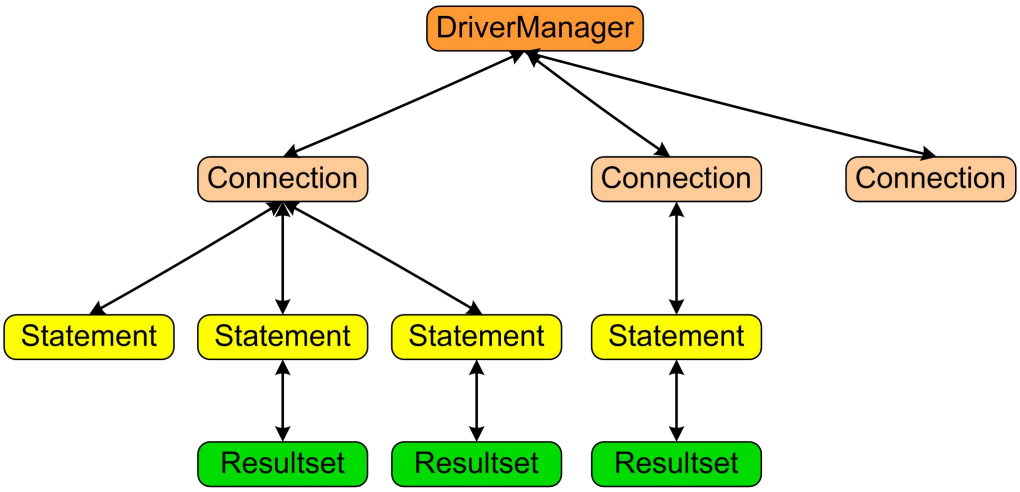

图 4-1　JDBC API 结构关系图

4.2.3.2　基本操作

1） 对表中记录的操作

对一个表中的记录可以进行修改、插入、和删除操作，分别对应于 SQL 的 UPDATE、INSERT、DELETE 操作。同 SELECT 操作类似，executeUpdate 方法的参数也是一个 String 对象，代表将要执行的进行更新操作的 SQL 语句。ExecuteUpdate()方法的返回值不是一个 ResultSet 对象，而是一个整数。对于 UPDATE、INSERT、DELETE 操作，这个整数是操作所有有影响的记录的行数。对于其他不返回值的 SQL 语句，例如，CREATE TABLE,ALTER TABLE,DROP TABLE 等，executeUpdate()的返回值是零。我们看下面的语句：

```
String sql=”UPDATE Sales SET total=’6000’ WHERE id=’09980’”;
```

```
stmt.executeUpdate(sql);
```

该语句将 Sales 表中 id 号为’09980’的记录的销售总额修改为 6000。

我们用下面的语句来添加和删除记录：

```
String    sqlInsert=” INSERT INTO Sales (total, id) VALUES(8000，’09974’)”;
stmt.executeUpdate(sqlInset);
String    sqlDelete=”DELETE FROM Sales WHERE id=’09981’;
stmt.executeUpdate(sqlDelete);
```

增加记录时，如果指出列名，则 VALUES 中的值赋予相应的列，对于没有写出列名的 SQL 语句，又分两种情况：如果创建该表时为该列指定了默认值，则将该默认值赋予该列。如果创建该表时没有为该列指定默认值，则为该列赋值为 NULL。如果 INSERT 语句中不写列名，则 VALUES 中的值按顺序赋予各列。下面的这段程序对 Sales 表进行更新操作，并显示操作前和操作后的结果：

该程序所操作的数据库表的表结构如表 3 所示：

表 3 例中的表结构

列名	数据类型
Id	CHAR
Name	CHAR
Sale_total	INTEGER

```
try
{
    Class.forName(“com.mysql.jdbc.Driver”);
    //加载驱动程序。
    Connection con=DriverManager.getConnection(“jdbc:DosSQL://localhost/database”);
    //建立连接。
    Statement statement=con.createStatement();
    //用 Connection 对象创建 statement 对象。
    String sqlquery=”SELCT * FROM    sales”;
    ResultSet resultset=statement.executeQuery(sqlquery);
    //执行查询操作
    System.out.println(“更新前表中的记录信息：”);
    while(resultset.next())
    {
        System.out.println(resultset.getString(“id”));
        System.out.println(resultset.getString(“name”));
        System.out.println(resultset.getInt(“sale_total”));
    }//while
    //显示更新前表中的记录信息。
    String sqlupdate=”UPDATE sales SET sale_total=9900 WHERE id=’001’”;
    statement.executeUpdate(sqlupdate);
    //更新表中的数据
    String sqldelete=”Delete FROM sales WHERE id=’002’”;
```

```
    statement.executeUpdate(sqldelete);
    //删除表中的数据
    String sqlinsert=”INSERT INTO sales “+ “(id, name,sale_total)    VALUES(‘009’,’goods9’ 7870)”;
    statement.executeUpdate(sqlinsert);
    //向表中插入数据
    resultset=statement.executeQuery(sqlquery);
    while(resultset.next())
    {
        System.out.println(resultset.getString(“id”));
        System.out.println(resultset.getString(“name”));
        System.out.println(resultset.getInt(“sale_total”));
    }//while
    //显示更新后表中的记录信息。
    rs.close();
    statement.close();
    con.close();
    //关闭与数据库相关的各种对象。
}catch(SQLException e)
{
    System.err.println(“SQLExcepton meet in record operation:”+e.getMessage();
}//catch
//声明捕获例外
```

2） 创建和删除表

创建和删除一个表对应于 SQL 的 CREATE TABLE 和 DROP TABLE 语句。也是使用 Statement 对象的 executeUpdate()方法来完成。

下面的语句创建一个表，该表有三列。数据类型分别是 INTERGER、CHAR、DATE。

```
 String sqlstr="CREATE TABLE    tablename“
      "("+
    "int_column       INTEGER,"+
    "char_column      CHAR,"+
    "date_culumn       DATE"
    ")";
System.out.println(sqlstr);
try
{
    Statement statement=con.createStatement();
    statement.executeUpdate(sqlstr);
}catch(SQLException e)
{
     System.err.println("in createTable:"+e.getMessage());
}
```

删除一个表，要用到 DROP TABLE 语句，例如，

```
String sqldroptable=”DROP TABLE    tablename”;
try
{
        Statement statement=con.getConnection();
        statement.executeUpdate(sqldroptable);
}catch(SQLExecptin e)
{
        System.err.println("in createTable:"+e.getMessage());
}
```

3） 增加和删除表中的列

对一个表的列进行更新操作是使用 SQL 的 ALTER TABLE 语句。对列进行的更新操作要影响到表中所有的行。

下面的语句在 t1 表中增加一列 Address，数据类型为字符串：

```
String sqlAddColumn=”ALTER TABLE T1 ADD COLUMN Addres VARCHAR(50)”;
statement.executeUpdate(sqlAddColumn);
```

在增加一列后，表中以前存在的行的次列值为 NULL。可以使用 UPDATE 语句设置此列的值，也可以使用 INSERT 语句增加新的记录。

删除表中一行的语句如下：

```
String sqlDelColumn=”ALTER TABLE T2 DROP COLUM Address”;
Statement.executeUpdate(sqlDelColumn);
```

下面一段程序实现了创建表，加入记录，增加和删除表中的列和删除表等功能：

```
Class.forName(“com.mysql.jdbc.Driver”);
Connection con=DriverManager.getConnection(“jdbc:DosSQL://localhost/database”);
CheckForWarning(conn.getWarnings());
Statement statement=conn.createStatement();
//创建表 t1,并加入两行记录
statement.executeUpdate(“CREATE TABLE t1 (c1 INTEGER, c2    VARCHAR(20))”);
statement.executeUpdate(“INSERT INTO t1 VALUES（100，’aaa’）”);
statement.executeUpdate(“INSERT INTO t1 VALUES (200, ‘bbb’)”);
ResultSet resultset=statement.executeQuery(“SELECT *    FROM    t1”);
System.out.println(“Result    after Create it”);
DisResultSet(resultset);
//在表中增加一列，并加入第三个记录
statement.executeUpdate(“ALTER TABLE t1 ADD COLUMN c3 VARCHAR(10)”);
statement.executeUpdate(“INSERT INTO t1 (c1,c2,c3) VALUES(300,’ccc’,’zzz’)”);
result=statement.executeQuery(“SELECT *    FROM t1”);
System.out.println(“Result after add column c3”);
dispResultSet(resultset);
//从表中删除一列
statement.executeUpdate(“ALTER TABLE t1 DROP COLUMN c2”);
resultset=statement.executeQuery(“SELECT * FROM t1”);
dispResultSet(resultset);
```

```
//删除这个表
statement.executeUpdate(“DROP TABLE tt “);
rs.close();
statemetn.close();
conn.close();
```

4.2.3.3 DatabaseMetaData 对象

Java 程序通过 JDBC 与数据库管理系统进行连接以后，得到一个 Connection 对象，可以从这个对象获得有关数据库管理系统的各种信息，包括数据库中的各个表，表中的各个列，数据类型，触发器，存储过程等各方面的信息。根据这些信息，JDBC 可以访问一个事先并不了解的数据库。获取这些信息的方法都是在 DatabaseMetaData 类的对象上实现的，而 DataBaseMetaData 对象是在 Connection 对象上获得的。

下面的语句可以在一个已经初始化的 Connection 对象的基础上创建一个 DatabaseMetaData 对象：

DatabaseMetaData datameta=con.getMetaData();

DatabaseMetaData 类中提供了许多方法用于获得数据源的各种信息，通过这些方法可以非常详细的了解数据库的信息。下面给出一些常用的方法，其他的方法请读者参考 JDK API 中的 DatabaseMetaData 类。

这些方法中，有一些要用字符串搜索匹配模式作为参数，这些匹配模式与 ODBC 中的一样。字符’_’匹配任何单个字符，’%’匹配零个或者多个字符。一个值为 null 的 Java String 可以与任何字符串相匹配。

1) 获取数据库的基本信息

①getURL()

返回一个 String 类对象，代表数据库的 URL。

②getUserName()

返回连接当前数据库管理系统的用户名。

③isReadOnly()

返回一个 boolean 值，指示数据库是否只允许读操作。

④getDatabaseProductName()

返回数据库的产品名称。

⑤getDatabaseProductVersion()

返回数据库的版本号。

⑥getDriverName()

返回驱动程序的名称。

⑦getDriverVersion()

返回驱动程序的版本号。

2) 获取数据库中各个表的情况

ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])

getTables 方法返回一个 ResultSet 对象，每一条记录是对一个表的描述。只有那些符合参数要求的表才被返回。

3) 获取表中各列的信息

ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String types[])

getColumns 返回一个 ResultSet 类的对象，其中每一行是对一列的描述，只有符合参数要求的列才被返回。

4) 获取关于索引的信息

ResultSet getIndexInfo(String catalog, String schema, Boolean unique, boolean approximate)

getIndexInfo 方法返回一个 ResultSet 类的对象，其中每一行是对一个索引的描述，只有符合参数要求的索引才被返回。

4.2.3.4 Driver 对象

(1) Driver 概述

Java.sql.Driver 接口提供了 6 种方法。其中一种方法用于在驱动程序和数据库之间建立连接；其他方法给出与驱动程序相关的信息，或获取到数据库的连接的必要信息。JDBC 提供 Driver 类来实现 Driver 接口。

Driver类中的大多数方法通常在后台工作，被通用的java.sql.DriverManager类所调用。例如，使用 DriverManger 类与数据库建立连接的应用程序调用 DriverManger.getConnection 方法。驱动程序管理器轮流在每一个注册的驱动程序上调用 Driver 中定义的 connect 方法，然后使用能够建立连接的第一个驱动程序来建立连接。为获取更详细的解释，请参见 DriverManager 的用法。然而，如果用户希望指定使用一个特定驱动程序，可以越过 DriverManger 类而直接调用 connect 方法。同样，虽然任何人都可以调用 getPropertyInfo 方法，但是大多数用户都不会用这个方法。该方法的本来目的是为了建立连接，允许 GUI 工具发现应该对用户显示的提示信息。值得注意的是，JDBC 驱动程序已经提供了一个单独的 DataSource 类，开发者可直接调用该类的方法建立到数据库的连接。推荐使用这个机制来建立连接，而不是直接使用 Driver 类。

(2) 加载和注册驱动程序

Driver 类应该使用静态部分来实现，这个静态部分称为静态初始化器，当加载这个静态初始化器时自动完成两项工作：（1）创建当前驱动程序自身的一个实例；（2）调用 DriverManager.registerDriver 方法来注册该实例表示的驱动程序。若 Driver 类包含这个静态部分，用户将能简单的调用 Class。forName 方法来加载和注册驱动程序，以字符串的方式向 Class.forName 方法提供驱动程序的类名。例如，下面的代码加载和注册 Driver 类 foo.bah.Driver：Class.forName(“foo.bah.Driver”);

4.2.3.5 DriverManager 对象

(1) DriverManager 概述

DriverManager 类是 JDBC 中传统的管理层，在用户和 JDBC 驱动程序之间协同工作。DriverManager 跟踪可用的驱动程序，并在数据库和相关的驱动程序之间建立连接。除此之外，DriverManager 类还需要掌握驱动程序的登陆时间限制，并将日志和追踪消息写入到日志文件中。

对一些简单的应用程序而言，在 DriverManager 类中，普通程序员通常需要直接调用的唯一方法是 DriverManager.gerConnection。顾名思义，该方法建立到数据库的连接。应用程序可以调用 DriverManager 方法 getDriver， getDrivers 和 registerDriver，以及 Driver 中的 connect 方法。但是大多数情况下，应用程序最好让 Drivermanager 类来管理建立连接。

(2) 跟踪可用驱动程序

DrvierManager 类维持一个已经注册的 Driver 类列表，这些 Driver 类通过自己调用 Drivermanager.registerDriver 来注册。对所用 Driver 类都应该设计有静态部分（也就是净化的初始化器），在这个初始化器中创建 Driver 自身的实例，并在加载时向 DriverManager 类注册这个 Driver，因此用户通常不需要直接调用。当一个 Drivermanager.registerDriver 方法，这个方法将在 Driver 类加载时被自动调用。当一个 Driver 类被加载后，将自动的以下列两种方式之一向 DriverManager 注册：

调用 Class.forName 方法，显示的加载这个驱动程序类。因为这种方法不需要依赖任何外部安装，所以建议用户使用这种方法来加载驱动程序（这种方法使用 Drivermanager 结构）。下面的代码加载了 类 acme.db.Driver：class.forName(“acme.db.Driver”);

如果编写的 acme.db.Driver 能够在加载该类时创建该类的实例，并用创建的实例作为参数调用 DriverManager.registerDriver 方法（这正是驱动程序应该做的），那么在调用 class.forName 方法后，这个 acme.db.Driver 类将位于 Drivermanager 的驱动程序列表中，并可以用来建立连接。

将 Driver 类添加到 java.lang.System 的属性 jdbc.drivers 中。该属性是一个驱动程序类名的列表，类名之间用冒号隔开，这些驱动程序类由 Drivermanager 类加载。当 Drivermanger 类被初始化时，它查找系统属性 “jdbc.drviers ”，如果用户已经拥有一个或多个驱动程序，则 DriverManager 类就试图加载这些驱动程序。下面的代码解释了如何在 ~/.hotjava/properties 中加载三个驱动程序类（ HotJava 在启动时将这些类装载入系统属性中）： jdbc.drivers = foo.bah.Driver: wombat.sql.Driver:bad.test.ourDriver 在第一次调用 DriverManager 方法时将自动加载这些驱动程序类。

值得注意的是这里加载驱动程序的第二种方法需要一个永久存储环境。如果这个条件不具备，则选择第一种方式调用 Class.forName 方法来显示加载的每一个驱动程序会更加安全。 Class.forName 也是用来引导特定驱动程序的正确方法，因为 DriverManager 类初始化后，它将永远不会重新检测 jdbc.drivers 属性列表。

在上述的两种情况下，新加载的 Driver 类需要通过调用 DriverManager.registerDriver 方法向 DriverManager 注册。就像前面所提的那样，Driver 在加载时自动完成注册。

考虑安全方面的原因，JDBC 驱动程序管理层将跟踪哪一个类加载器提供了哪些驱动

程序。然后，当 DriverManager 类打开连接时， 它只是用来自本地文件系统的驱动程序，或来自与发出连接请求的代码相同类加载器。

(3) 建立连接

加载 Driver 类并在 DriverManager 类中注册之后，该 Driver 类就能用来建立与数据库的连接。当调用 DriverManager.getConnection 方法发出连接请求时，DriverManager 类轮流检测每一个驱动程序，看哪一个驱动程序能建立连接。

有时候会有多个 JDBC 驱动程序能够连接到给定的 URL。例如，当连接到一个给定的远程数据库时，可以使用 JDBC-ODBC 桥驱动程序， JDBC 通用网络协议驱动程序或由数据库厂商提供的驱动程序。因为 DriverManager 类将使用第一次查找到的能够成功连接到给定 url 的驱动程序，所以在这种情况下，检查驱动程序的顺序非常重要。

首先，DrivaerManager 类通过对每一个驱动程序的注册顺序使用每一个已经注册的驱动程序 (jdbc.drivers 类表中的驱动程序总是最先注册)。 DriverManager 将跳过代码不可信任的任何驱动程序，除非加载它们的源与试图打开这个连接的代码的源相同。

DriverManger 类通过对每一个驱动程序调用 Driver.connect 方法，并向它们传递用户在 DriverManager.getConnection 方法中输入的 URL 来对它们进行轮流测试，在第一个识别出这个 url 的驱动程序间建立实际的连接。

这种方法看似效率不高，但是因为不可能同时加载许多驱动程序，因此每次连接实际只需要几个过程调用和字符比较操作。

下面的代码实例就是建立连接通常所需的全部代码，使用的驱动程序为 JDBC_ODBC 桥驱动程序。

```
Class.forName("jdbc.odbc.JdbcOdbcDriver"); //loads the driver
Strign url "jdbc:odbc:fred";
Connection con =DriverManager.getConnection(url, "userId","passwd");
```

变量 con 表示到数据源“fred”的连接，它能用来创建和执行 SQL 语句。

[2.0] 在 JDBC 2.0 Optional PackageApI 中，DataSource 对象能够用来建立与数据源的连接，这时仍可以使用 DriverManager，但是 DataSiource 对象比 DriverManager 提供更多的优点，因此它是一种更好的选择。然而，编写 Enterprise JavaBEans 组件（EJB 组件）的开发者应该使用 DataSource 对象，而不是使用 DriverManager。使用适合实现 DataSource 对象是获取连接池中的连接或能够参加与分布式事务的连接的唯一机会。

因为 Drivermanager 的方法是静态的，所以 DriverManager 方法被声明为 static，这意味着这些方法是在整个类上而不是在特定实例上进行操作。事实上， DriverManager 的构造函数通过声明为 private 来阻止实例化 DriverManager 类。从逻辑上看，DriverManager 类只有一个实例。这意味着要使用 DrvierManager 类来限定 DriverManager 方法的调用，正如下面代码行所表示：

```
DriverManager.setLogWrite(out)";
```

4.2.3.6 Connection 对象

(1) 概述

Connection 对象表示与数据库的连接。连接会话包括已执行的 SQL 语句以及在该连接上返回的结果。一个应用程序可以与一个数据库建立一条或多条连接，并且应用程序还可以与多个不同的数据库建立连接。

通过调用 Connetion.getMetaData 方法用户可以获取 Connection 对象的数据库信息。Connection.getMetaData 方法返回一个 DatabaseMetaData 对象，该对象包括数据库中的数据表、数据库支持的 SQL 语法、以及连接性能等信息。

(2) 打开连接

数据库建立连接的传统方式就是调用 DriverManager.getConnection 方法，该方法以一个包含 URL 的字符串作为参数。DriverManager 类属于 JDBC 管理层，它会尝试定位一个驱动程序，该驱动程序能够连接到给定的 URL 表示的数据库。DriverManager 类维护一个由已经注册的 Driver 类组成的列表，当调用 getConnection 方法时，DriverManager 类与列表中每一个驱动程序进行协商，直到发现一个驱动程序能连接到 URL 中指定的数据库为止。

用户可以绕过 JDBC 管理层而直接调用 Driver 方法。若有两个驱动程序能连接到同一个数据库，而用户又想显式的选择一个特定的驱动程序，在这种情况下直接调用 Driver 方法是比较有用的。

如

```
String url = “jdbc:odbc:wombat”;
Connection con = DriverManaget.getConnection(url,”oboy”,”12Java”);
```

与使用 DriverManager 对象相比，DataSource 接口是建立连接的更好的选择。

应用程序使用 DataSource 对象产生的 Connection 对象的方式本质上与它使用 DriverManager 产生的 Connection 对象的方式相同。然而也存在着差异，详细信息请参见 PooledConnection 用法。

(3) JDBC URL

JDBC URL 提供了一种标识数据源的方式，因此适当的驱动程序能够辨认该 URL 并建立与该 url 的连接。因为 JDBC URL 供不同类型的驱动程序使用，所以十分灵活。首先协定允许与给定的驱动程序使用不同的模式命名数据库。其次允许间接层次，如可以使用逻辑主机或数据库名等。第三，JDBC 允许在 URL 中包括所有必要的连接信息。

(4) 发送 SQL 语句

一旦建立连接，就可以使用该连接将 SQL 语句发送到它的底层数据库。JDBC API 提供了两个接口向数据库发送 SQL 语句：

Statement——由 Connection.createStatement 方法创建。Statement 对象用来发送没有参数的 SQL 语句。

PreparedStatemment——由 Connection.PreparedStatemment 方法创建。该方法可以携带一个或多个参数作为输入参数。PreparedStatemment 有一组方法来设置 IN 参数的值，执行语句后，这些参数将发送到数据库，PreparedStatemment 扩展了 Statement，因此包含

Statement 的方法。PreparedStatemment 对象可能比 Statement 对象效率更高，因为 PreparedStatemment 已通过预编译存储。多次使用的 SQL 语句通常使用 PreparedStatemment 对象。

添加的新版本的 Connection 方法可以与 ResultSet 接口的常数一起创建语句，这就能够指定新创建的语句产生的结果集的可保持性。

应用程序能够通过调用 Connection 对象的 setHoldbility 方法来设置 ResultSet 对象的可保持性，这些 ResultSet 对象是由连接创建的语句生成的。如

```
Statement stmt = con.createStatement(
                ResultSet.YEPE_SCROLL_SENDSITIVE,
                ResultSet.CONCUR_UPDATABLE,
                ResultSet.HOLD_CURSORS_OVER_COMMIT);

Con.setHoldbility(ResultSet.CLOSE_CURSORS_AT_COMMIT);
Statement stmt2 = con..createStatement();
Int holdability = con.getHoldability();
```

(5) 事务

事务由一个或多个已经执行、完成、要么提交要么回滚的语句组成。当调用 commit 方法或 rollback 方法时，当前事务结束，另一个事务开始。

默认情况下，一个新的 connection 对象处于自动提交模式。如果禁用自动提交模式，在显示的调用提交或者回滚之前事务不会结束。在禁用自动提交模式下，事务中的所有语句成批的进行提交或回滚。 事务的开始不需要显示的调用，它在禁用自动提交模式或调用提交方法或者回滚方法之后进行隐式的初始化。Commit 方法使得语句对数据库的所有修改得以永久保存，同时释放事务所有的锁资源，rollback 方法将放弃对数据库的改变。

4.2.3.7 ConnectionPoolDataSource 对象

(1) 概述

ConnectionPoolDataSource 对象是 PoolConnection 对象的工厂。PoolConnection 对象表示缓存在内存中的物理连接；因此 PoolConnection 对象可以被重用，当应用程序需要一个新的连接时，这种方式节省了创建一个新的物理连接的开销。ConnectionPoolDataSource 对象上的操作完全是内部的，因此 ConnectionPoolDataSource 接口不是应用程序员使用的 API 的组成部分, 驱动程序厂商在连接池的实现中使用该接口。

(2) Connection 对象和 PooledConnection 对象描述

当应用程序调用 DataSource .getConnection 方法时，返回值总是 Connetion 对象。然而，Connetion 对象是可以变化的，这依赖于 DataSource 类的实现。若 DataSource 类是一个基本实现对象，则是一个到数据源的物理连接。若 DataSource 方法实现连接池，则 getConnection 方法返回给应用程序的 Connetion 对象实际上是 PooledConnection 对象的句柄。

若 DataSource 类实现连接池，调用 getConnection 方法时就会产生不同的时间序列。连接池模块不是立即创建一个新的 Connection 对象，而是检查在连接池中是否存在可重用的 PooledConnection 对象。如果存在，则那个 PooledConnection 对象就用来创建一个新的 Connection 对象返回给应用程序。如果没有可供重用的 PooledConnection 对象，将使用 ConnectionPoolDataSource 对象创建一个 PooledConnection 对象。在任何情况下 PooledConnection. getConnection 方法都创建 Connection 对象返回给应用程序。

(3) 语句重用

不管是连接池还是语句池，重点是池机制只在后台操作，并不以任何方式影响应用程序代码。实际上，规范要求通过 PooledConnection 对象 PreparedStatement 对象的重用完全对应用程序透明。这就意味着使用 PreparedStatement 对象的应用程序在每次使用时，必须创建和关闭 PreparedStatement 对象。如果已经实现了语句池，PreparedStatement 对象的关闭只是把该对象返回给语句池而不是实际上关闭它。下次应用程序创建 PreparedStatement 对象时，池中的 PreparedStatement 对象可以被重用。PooledConnection 对象的标准属性集规定了连接池被关闭以前它们该放入多少条物理连接、多少物理连接可用、在连接被关闭以前它们可以空闲的时间等。应用于语句池的属性 maxStatements 指定了一个 PooledConnection 对象在它的语句池中应该保持打开的 PreparedStatement 对象的数目。相关标准请参见 JDBC 3.0API 标准。

(4) 连接池和语句池的属性

ConnectionPoolDataSource 实现中的属性集应用于 ConnectionPoolDataSource 实例创建的 PooledConnection 对象。这些属性值的集合决定了连接池和语句池的配置。

PooledConnection 属性是 ConnectionPoolDataSource 实现的组成部分；应用程序从来不直接使用这些属性。实现可以定义的附加属性，如果它定义这种附加属性的话，就必须确保所有属性名唯一。不管是特定于实现的属性还是标准属性，实现都必须对它所使用的所有属性提供 getter 和 setter 方法。以下是连接池属性描述：

maxStatements:池应该保持打开的语句总数，0 表示禁用语句缓存。

initialPoolSize:创建连接池时应该包含的物理连接数。

minPoolSize:连接池中物理连接的最小数目。

maxPoolSize:连接池中应该包含的物理连接的最大数目，0 表示没有最大数目限制。

maxIdleTime:物理连接被关闭以前它在连接池中保持空闲状态的秒数，0 表示没有时间限制。

propertyCycle:当前这些连接池属性值的定义策略，连接池在强加这些策略之前应该等待的以秒为单位的时间间隔。

管理 PooledConnection 对象池的应用服务器利用这些属性来确定怎样管理连接池。属性的 getter 和 setter 方法在 ConnectionPoolDataSource 的实现中定义。

(5) 关闭池中的语句

使用池中的语句的应用程序必须创建 PreparedStatement 对象并关闭这些对象，应用程

序就好像在使用没有置入语句池的语句一样。如果应用程序创建了几个 PreparedStatement 对象但是忘记关闭它们，即使已经实现了语句池，这些对象也不能被重复使用。换句话说，为了使语句可以重用，应用程序必须关闭池中的这个语句。应用程序可以使用两种方式来关闭 Statement 对象：在语句上调用 close 方法或者在连接上调用 close 方法。当应用程序在池中的语句上调用 PreparedStatement.close 方法时，应用程序所使用的逻辑语句将被关闭，但是物理语句在语句池中可以被重用。

关闭连接时关闭的不仅仅是连接，而且是关闭连接所创建的所有语句。因此，当应用程序在连接池中的连接上调用 connection.close 方法时，它不仅关闭了该应用程序所使用的 connection 对象，而且也关闭了应用程序所使用的逻辑 PreparedStatement 对象。然而底层的物理连接，也就是 PooledConnection 对象，将返回给连接池。

4.2.3.8 DataSource 对象

(1) 概述

DataSource 对象表示 Java 编程语言中的数据源。在基本术语中，数据源是存储数据的一种工具。数据源可以象复杂数据库一样复杂（对大公司而言），或者像只具有行和列的文件一样简单。数据源可以驻留在远程服务器上，也可以驻留在本地桌面计算机上。应用程序使用连接访问数据源，DataSource 对象可以被看作一个连接工厂，该工厂连接到 DataSource 实例便是特定数据源。DataSource 接口提供两种方法来建立到数据源的连接。

DataSource 对象使用 JNDI 命名服务进行注册，这一点使得 DataSource 对象比 DriverManger 多了两个主要的优点。首先，应用程序不需要硬编码驱动程序信息，而 DriverManger 需要硬编码驱动程序信息。程序员可以对数据源选择逻辑名，并使用 JNDI 命名服务注册该逻辑名，并且 JNDI 命名服务将提供与该逻辑名相关联的 DataSource 对象。然后利用 DataSource 对象创建到该对象所表示的数据源的连接。关于怎样使用 JNDI 以及使用 JNDI 的优点的更多信息将在以后介绍。

(2) 属性

DataSource 对象具有用来标识和描述它所表示的真实世界数据源的属性集。这些属性包括诸如数据库服务器的位置、数据库的名字以及与服务器进行通信的网络协议等信息。DataSource 属性遵循 JavaBeans 设计模式，并且通常在部署 DataSource 对象时进行属性设置。

为了鼓励不同厂商的 DataSource 实现保持一致性，JDBC 2.0 API 规定了标准的属性集，并且为每个属性值定了标准名。下表给出了标准名、数据类型以及每一个标准属性的描述。值得注意的是，DataSource 实现并不一定要支持所有这些属性；该表只是说明当实现支持某一属性时，这种实现应当使用的标准名。

属性名字	类型	描述
databaseName	String	服务器上特定数据库的名字
dataSourceName	String	底层 XADataSource 或 ConnectionPoolDataSource 对象的逻辑名，只有当实现连接池或分布式事务时才使用该属性。

description	String	该数据源的描述
networkProtocol	String	用来与服务器进行通信的网络协议
password	String	用户数据库密码
portNumber	int	服务器监听请求的端口
roleName	String	初始的 SQL 角色名
serverName	String	数据库服务器名
user	String	用户账号名

当然，DataSource 对象必须支持它所表示的数据源建立连接所需的所有属性，但是所有 DataSource 实现都需要支持的属性只有 description 属性。对于属性的这种标准化式的编写用于列出可用数据源的工具成为可能，从而能够给出每个数据源的描述以及其它有用的属性信息。

JDBC 2.0 API 并不限制 DataSource 对象只使用商标所指定的属性。厂商可以增加自己的属性，在这种情况下，厂商应该给每一个新属性提供一个特定于厂商的名字。

如果 DataSource 对象支持某一个属性，它必须提供对该属性的 getter 和 Setter 方法。下面的代码段解释了 DataSource 接口（在这个示例中为 vds）的厂商实现如果支持某个属性（例如属性 serverName）所应当包含的方法。

```
vds.setServerName(“my_database_server”);
String serverName = vds.getServerName();
```

属性最好通过开发者或系统管理员使用 GUI 工具进行设置，GUI 工具是数据源安装的组成部分。连接到数据源的用户不需要获取或设置属性。这通过在 DataSource 接口中不包括属性的 getter 和 setter 方法来强制做到这一点；这些方法只在实现中提供。getter 和 setter 方法包含在实现中而不是在公共接口中的作用就是在管理 DataSource 对象的 API 和用户使用的 API 之间形成一定的独立性。管理工具通过使用自省机制（introspection）能够获取属性。

(3) 使用 JNDI

JNDI 为应用程序提供统一的方式来发现和访问网络上的远程服务。远程服务可以是任何企业服务，包括消息服务和特定应用服务。当然，JDBC 应用程序主要对数据库服务感兴趣，一旦创建了 DataSource 对象并且已经使用 JNDI 命名服务进行过注册，应用程序就能够使用 JNDI API 来访问 DataSource 对象，然后就可以利用该 DataSource 对象连接到它所表示的数据源。

(4) 创建和注册 DataSource 对象

DataSource 对象的创建、部署和管理通常和使用该对象的 java 应用程序相分离。特定的数据源的 DataSource 对象是由开发者或系统管理员而不是由用户来创建和部署。

```
Context initContex = new InitialContext();
Context ctx = (Context)initContext.lookup(“java:/comp/env”);
DataSource ds = (DataSource_envContext.lookup(“jdbc/sample_db”);
Con = ds.getConnnection();
```

以上代码说明了创建 DataSource 的方法。JNDI 命名空间由初始名字上下文(context)以及该名字上下文中任意数目的子上下文组成。JNDI 层次的根是初始上下文，这里通过变量 envContext 来表示。在初始上下文中有许多子上下文，其中之一就是 jdbc，JNDI 子上下文留给 JDBC 数据源使用。层次中的最后一级元素是注册的对象。下面说明应用程序如何使用这种关联连接到数据源。

(5) 连接到数据源

实例代码：

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.loopup(“jdbc/sample_db”);
Connection con = ds.getConnection(“genius”,”abracadabre”);
Con.setAutoCommit(false);
PrepareStatement pstmt = con.preparestatement(“SELECT NAME,TITLE FROM PERSONNEL WHERE DEPT = ?”);
Pstmt.setString(1,”SALES”);
ResultSet rs = pstmt.executeQuery();

System.out.println(“Sales Department:”);
Whie(rs.next())
{
        String name = rs.getString(“NAME”);
        String title = rs.getString(“TITLE”);
        System.out.println(name+“” + title);
}
Pstmt.setString(1,”CUST_SERVICE”);
ResultSet rs = pstmt.executeQuery();

System.out.println(“Customer Service Department:”);
While(rs.next()){
    String name = rs.getString(“NAME”);
    String title = rs.getString(“TITLE”);
    System.out.println(name+””+title);
}
Con.commit();
Pstmt.close();
Con.close();
```

前两行代码使用 JNDI API；第三行代码使用 DataSourde API。在第一行代码为初始名上下文创建了 javax.naming.Context 实例后，第二行代码在该实例上调用方法 lookup 以获取与 jdbc/sample_db 相关联的 DataSource 对象。

(6) 日志

DataSource 接口提供了一些方法，允许用户获取和设置字符流，追踪和错误日志将写

入到这些字符流中。用户可以在给定的流上追踪特定的数据源。一个新的 DataSource 对象默认情况下将禁用日志，因此如果要使用日志就必须设置日志书写器。下面的代码行设置了 DataSource 对象的日志书写器。

```
Java.io.printwriter out = new java.io.printwrite();
Ds.setlogwrite(out);
```

使用下面的代码行能够禁用日志书写器。

```
Ds.setlogwrite(null);
```

下面的代码行获取字符输出流，日志和追踪消息将写入到该字符输出流中。

```
Java.io.printwriter write = ds.getlogwrite();
```

4.2.3.9 Statement 对象

(1) 概述

Statement 对象用于向数据库发送 sql 语句。Statement 接口为语句的执行和查询结果的读取提供了基本方法。它不同于 preparestatement 对象，Statement 对象用于执行没有参数的简单语句。

(2) 创建 Statement 对象

对某个特定的数据库的连接一旦建立，这个连接就可以向该数据库发送 SQL 语句。Statement 对象是通过 Connection 的 createStatement 方法来创建，就像下面的代码所示：

```
Connection con = DriverManager.getConnection(url,"sunny","");
Statement stmt = con.createStatement();
```

向数据库发送的 SQL 语句将会作为 Statement 对象的某个 execute 方法的参数，下面的例子说明了这一点，此例使用的是 executeQuery 方法：

```
ResultSet rs = stmt.executeQuery("SELECT a,b,c FROM table2");
```

变量 rs 指向一个结果集，该结果不能被更新并且其中的游标只能向前移动这些是 ResultSet 对象的默认行为。但现在新的 API 增加了一个新版本的 Connection.create Staetment 方法，这个方法创建了 Statement 对象可以产生可滚动、可更新的结果集。

(3) 使用 Statement 对象执行语句

Statement 接口为 SQL 语句的执行提供了 3 个方法：exexuteQeruy,executeUpdate,execute 方法。使用方法是否正确取决于 SQL 语句产生什么样的结果。

executeQuery 方法是为执行产生简单结果集的语句而设计的，这样的语句如 SELECT 语句。

exuecteUpdate 方法是用来执行 INSERT，UPDATE，或者 DELETE 语句的，也可以用来执行 SQL　DLL　。该方法返回更新计数，如果是表等不是对单独的数据行进行操作，所以 executeUpdate 方法返回值始终是 0。

Exeucte 方法用于可以返回多个结果集的语句、返回多个更新计数的语句或者二者都返回的语句。这是一个高级特性，后面我们将单独说明。

```
Statement stmt = con..createStatement();
ResultSet rs = stmt.executeQuery('SELECT *FROM TABLE 1");
Rs = stmt.executeQeruy("SELECT *FROM TABLE2");
```

注意，继承了 Statement 接口所有方法的 Prepared Statement 接口拥有自己的 exuectQuery 方法,executeUpdate 方法和 execute 方法。Statement 对象本身并不包含 SQL 语句。因此必须有一个 SQL 语句作为 Statement.execute 方法的参数。

(4) 关闭语句

Statement 对象会被 JAVA 的无用单元收集器自动关闭。然而，我们推荐当这些 Statement 对象不再需要时，使用显示的方法关闭该对象，这是一个良好的编程习惯。

当应用程序将 Connection 对象关闭时，所有使用该 Connection 对象创建的 Statement 对象也随之关闭。另外，所有由这些 Statement 对象生成的 ResultSet 对象也将随着 Statement 对象的关闭而关闭。

(5) 发送批量更新

支持批量更新的机制允许 Statement 对象将多个更新操作命令作为一个单元，或者作为一批一起提交给底层数据库管理系统。

```
Statement stmt = con.createStatement();
Con.setAutoCommit(false);

Stmt.addbatch("INSERT INTO employees VALUES (1000,'JOE')");
Stmt.addbatch("INSERT INTO departments    VALUES (100,'SHOE')");
Stmt.addbatch("INSERT INTO emp_dept VALUES (1000,'260')");
Int [ ] updateCounts = stmt.executeBatch();
```

在此例中，为了向公司的数据库中添加一名新员工，要将一行新的数据插入到三个不同的数据表中。首先将 Connction 对象 stmt 之后，又通过 addBatch 方法向批处理命令中添加 3 个 INSERT INTO 命令，随后又调用 executeBatch 方法将批处理命令发送给数据库。

因为关闭了数据库连接的自动提交事务功能，当错误发生时应用程序可以自由决定是否提交该事务。Statement 对象是和一个与其相关联的命令列表一同被创建的。该列表开始时为空，命令是通过 addBatch 方法向该列表中添加的。所有添加到该列表的命令只能是返回一个简单的更新计数的命令。该命令列表可以通过 clearBatch 方法清空。

(6) 执行特殊类型的语句

只有当语句可能的返回值不只是一个 resultSet 对象、不只是一个更新计数或者是 ResultSet 对象和更新计数的混合体的时候，才应该使用 execute 方法。当执行某些存储过程或者当动态的执行某个未知的 SQL 字符串时就可能导致返回多个结果。

因为 execute 方法用于处理那些超乎寻常的情况，那么需要使用一些特殊的手段来读取它们的返回结果。但是 execute 方法返回的既不是 ResultSet 对象也不是更新计数，而是一个 boolean 值，这个值指明了 execute 方法所生成的结果的类型：TURE 代表 ResultSet 对象，FALSE 代表更新计数。应用程序需要调用其它的方法来读取返回结果本身。所以不

能简单的用 executeQuery 与 executeUpdate 方法来替代 exuecte 方法。当知道语句的执行会返回一个 ResultSet 对象时，应用程序使用 executeQuery 方法，反之如果是一个更新计数，则用 executeUpdate 方法。否则未知情况就使用 execute 方法。

如果 execute 方法返回 True，则应用程序必须调用 getRsultSet 方法来获取 ResultSet 对象。然后使用 ResultSet 中的获取方法来读取 ResultSet 对象的各个值。如果 execute 方法返回 FALSE，则应用程序需要调用 getUpdateCount 方法。

4.2.3.10 PoolConnection 对象

PooledConnection 对象是提供数据源连接池机制的关键组件。DataSource 对象表示一个特定的真实数据源，实现 DataSource 可使连接重用。例如，DataSource 对象 ds 表示 XYZ 数据库，当调用 ds.getConnection 方法时，ds 返回的每一个连接就是到数据库 XYZ 的连接。如果 ds 的 DataSource 类已经实现了连接池，那么 ds.getConnction 方法的每一个连接就是可以使用并且可以重用的到 XYZ 的连接。当应用程序关闭连接时，该连接可以进入连接池而不是从物理上关闭这个连接。当下次应用程序从连接池中取连接时，还能获取到连接，事实上如果只建立一个连接，那么将反复使用同一个连接。

因为连接池完全在后台进行操作，所以它只有两种方式来影响应用程序代码。

首先，为了获取池中的连接，应用程序必须使用 DataSource 对象而不是 DriverManager 类来获取连接。任何 DataSource 对象，包括支持连接池的 DataSource 对象，都是使用 JNDI 命名服务进行注册，这由担任系统管理员角色的人提供。

```
Javax.naming.Context ctx = new InitialContext();
Javax.sql.DataSource ds = (DataSource)ctx.lookup(“jdbc/xyz”);
```

变量 ds 现在表示 DataSource 对象，应用程序可以在该对象上调用 getConnection 方法来获取到数据源的连接。

连接池对应用程序的第二个影响是应用程序应该显示的关闭连接。关闭连接总是一种好的编程习惯。对连接池而言，除非在连接池中的连接上调用 close 方法，否则该连接就不会返回到连接池中。因此为保证打开的连接的重用，程序员应该用 finally 关闭连接。

4.2.3.11 PreparedStatement 对象

在上一节中，我们讨论了用连接对象 Connection 产生 Statement 对象，然后用 Statement 与数据库管理系统进行交互。Statement 对象在每次执行 SQL 语句时都将该语句传递给数据库。在多次执行同一语句时，这样做效率较低。解决这个问题的办法是使用 PreparedStatement 对象。如果数据库支持预编译，可以在创建 PreparedStatement 对象时将 SQL 语句传递给数据库做预编译，以后每次执行这个 SQL 语句时，速度就可以提高很多。如果数据库不支持预编译，则在语句执行时，才将其传给数据库。这对于用户来说是完全透明的。

PreparedStatement 对象的 SQL 语句还可以接收参数。在语句中指出需要接收哪些参数，然后进行预编译。在每一次执行时，可以将不同的参数传递给 SQL 语句，大大提高了程序的效率与灵活性。一般情况下，使用 PreparedStatement 对象都是带输入参数的。

(1) 创建 PreparedStatement 对象

PreparedStatement 类是 Statement 类的子类。同 Statemetn 类一样，PreparedStatement 类的对象也是建立在 Connection 对象之上的。例如，为在 COFFES 表中更新数据创建一个 PreparedStatement 对象：

```
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE GOODS SET SALES = ? WHERE GOODS_NAME LIKE ?");
```

创建该 PreparedStatement 对象时，相应的更新记录的 SQL 语句已经被传递到数据库管理系统中进行预编译。在这里应该提醒读者，PreparedStatement 类是 Statement 类的子类，Statemet 类的方法 PreparedStatement 的对象都可以调用。另外，在建立 PreparedStatement 对象之前，应该保证 Connection 对象已经被正确的建立。

(2) 初始化 PreparedStatement 对象

如果以带输入参数的 SQL 语句形式创建了一个 PreparedStatement 对象（绝大多数情况下都是如此）。在 SQL 语句被数据库管理系统正确执行之前，必须为参数（也就是 SQL 语句中是'?'的地方）进行初始化。初始化的方法是调用 PreparedStatement 类的一系列 setXXX()方法。如果输入参数的数据类型是 int 型，则调用 setInt()方法；如果输入参数是 String 型，则调用 setString()方法。一般说来，Java 中提供的简单和复合数据类型，都可以找到相应的 setXXX()方法。例如上面举到的例子中，可以按如下语句进行初始化：

```
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
```

setXXX()方法一般有两个参数，第一个参数是 int 型，该参数指示 JDBC PreparedStatement 对象的第几个参数将要被初始化。第二个参数的值就是 PreparedStatemetn 将要被初始化的参数取值，数据类型自然也就相同。“updateSales.setInt(1,75);”语句使 updateSales 对象所对应的 SQL 语句中参数 SALES 的值被初始化为 75;“updateSales. setString(2, "goods1");”语句使 updateSales 对象所对应的 SQL 语句中参数 GOODS_NAME 的值被初始化为 goods1。

当 PreparedStatement 的一个对象的参数被初始化以后，该参数的值一直保持不变，直到它被再一次赋值为止。下面的代码可以很好的说明这个问题：

```
updateSales.setInt(1, 100);
updateSales.setString(2, "goods2");
updateSales.executeUpdate();
// 将商品名为”goods2”的记录的销售额设置为 100
updateSales.setString(2, "good3")
updateSales.executeUpdate();
// 将商品名为”goods3”的记录的销售额设置为 100
```

可以看到，这段代码的前一部分分别对 GOODS_NAME 和 SALES 的值进行了设置。然后才调用 executeUpdate()方法执行 SQL 语句。而在这段代码的后一部分，只设置了 GOODS_NAME 的值，Sales 的值没有进行更新，Sales 的值将维持 100 不变。再次调用 updateSales 对象执行 SQL 语句时，会把商品名为”goods3”的记录的销售额也设置为 100。

在这里顺便提醒读者，PreparedStatement 对象的 executeUpdate（）方法不同于 Statement 对象的 executeUpdate()方法，前者是不带参数的，其所需要的 SQL 语句型的参数已经在实例化该对象时提供了。

(3) 设置 PreparedStatement 对象参数

在 JAVA 的应用程序中，PreparedStament 对象要求输入大量的参数值，这就需要使用循环结构来解决烦琐的参数赋值问题。我们继续前面的例子：

```
int [] salesForWeek = {100, 85, 95, 125, 90};
String [] goodsname = {"goods1", "goods2", "goods3", "good4", "goods5"};
int len = goodsname.length;
for(int i = 0; i < len; i++)
{
updateSales.setInt(1, salesForWeek[i]);
updateSales.setString(2, goodsname[i]);
updateSales.executeUpdate();
}
```

在这段代码中，用 int 型数组来存储将要被更新记录的销售额，用 String 型数组存储将要被更新记录的商品名。这两个数组的值一一对应，销售额 100 对应着商品名为 goods1 的记录；销售额 85 对应着商品名为 goods2 的商品，依此类推。以后想再次更新数据库中的数据时，可以将这段代码当作模版，只需将数组中的数据进行修改，就可以完成要求的功能了。

执行 executeQuery()方法的返回值是一个包含符合特定要求的数据库的结果集，而执行 executeUpdate()方法并非如此，该方法返回一个整数，这个整数代表 executeUpdate()方法执行后所更新的数据库中记录的行数。例如

```
updateSales.setInt(1, 85);
updateSales.setString(2, "goods2");
int n = updateSales.executeUpdate();
// n = 1
```

这段代码执行后，GOODS 表中商品名是”goods2”的记录，销售额被更新为 85，有一条记录受到了影响。因此，n 的值是 1。

当 PreparedStatement 对象的 executeUpdate()方法所执行的 SQL 语句是诸如建立表格、修改表格、删除表格的 DDL 类型时，SQL 语句并不是直接对表中的记录进行操作，这时 executeUpdate()方法的返回值是 0。例如下面的语句：

```
String dropTable=”Drop Table GOODS”;
int n = updateSales.executeUpdate(createTableCoffees);
// n = 0
```

这里应该注意一点，当 PreparedStatement 对象的 executeUpdate()方法的返回值是 0 时，有两种可能：一种是所执行的 SQL 语句是对数据库管理系统的记录进行操作，并且还没有记录被更新；另一种是所执行的 SQL 语句是对数据库管理系统的表、视图等对象进行操作的 DDL 语言，没有数据记录被直接修改。

4.2.3.12 ResultSet 对象

在前面几小节中,我们介绍了用 Statement 类及其子类传递 SQL 语句，对数据库管理系统进行访问。一般说来，对数据库的操作 80%以上都是执行查询语句。这种语句执行的结果是返回一个 ResultSet 类的对象。要想把查询结果最后显示给用户，必须对 ResultSet 对象进行处理，本小节将介绍对结果集的处理方法。

(1) ResultSet 类的基本处理方法

一个 ResultSet 对象对应着一个由查询语句返回的一个表, 这个表中包含所有的查询结果。实际上, 我们就可以将一个 ResultSet 对象看成一个表，对 ResultSet 对象的处理必须逐行进行, 而对每一行中的各个列, 可以按任何顺序进行处理。

ResultSet 对象维持一个指向当前行的指针。最初, 这个指针指向第一行之前。Result 类的 next()方法使这个指针向下移动一行。因此, 第一次使用 next()方法将指针指向结果集的第一行，这时可以对第一行的数据进行处理。处理完毕后，使用 next()方法，将指针移向下一行，继续处理第二行数据。next()方法的返回值是一个 boolean 型的值，该值若为 true, 说明结果集中还存在下一条记录，并且指针已经成功指向该记录，可以对其进行处理；若返回值是 false, 则说明没有下一行记录，结果集已经处理完毕。

在对每一行进行处理时，可以对各个列按任意顺序进行处理。不过，按从左到右的顺序对各列进行处理可以获得较高的执行效率. ResultSet 类的 getXXX()方法可以从某一列中获得检索结果。其中 XXX 是 JDBC 中的 Java 数据类型，如 int, String , Date 等，这与 PreparedStatement 类和 CallableStatement 类设置 SQL 语句参数值相类似。 Sun 公司提供的 getXXX() API 提供两种方法来指定列名进行检索：一种是以一个 int 值作为列的索引，另一种是以一个 String 对象作为列名来索引。

下面是一段简单的显示检索结果的源代码：

```
Statement statement=con.createStatement();
//在 Connection 对象的基础上创建 Statement 对象。
String sql=”SELECT int_colmn, string_colmn,date_colmn,”+
”byte_colmn FROM    table_name”;
ResultSet result=statement.executeQuery(sql);
//用 Statement 对象执行 SQL 语句，返回结果集。
While(result.next())
{
    int int_value=result.getInt(1);
    String string_value=result.getString(“colmn2”);
    Date    date_value=result.getInt(3);
    Byte    byte_value[]=result.getString(“colmn4”);
    //从数据库中以两种不同的方式取得数据。
    System.out.println(int_value+” “+string_value+” “+date_value+” “);
}
```

(2) JAVA 和 SQL 之间数据类型的转化

Java 和 SQL 各自有一套数据类型（Java 应用程序的数据类型实际上就是 Java 的数据类型），我们要在应用程序和数据库管理系统之间正确的交换数据，必然要将二者的数据类型进行转换。但这种转换并不是要求绝对相同，比如说 Java 只提供变长度的数组，而不提供固定长度的，SQL 语言却二者都支持，我们的解决方法是将 SQL 中的定长数组也看成是变长的。再比如说，SQL 语言中没有 Java 中的 String 类型，却可以用各种 CHAR,VARCHAR 数据类型来代替。表 1 说明了 SQL 语言中的各种数据类型在 Java 中的默认表示。

表 1　SQL 到 Java 数据类型影射表

SQL 数据类型	JAVA 数据类型
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Int
BIGINT	Long
REAL	Float
FLOAT	Double
DOUBLE	Double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

表 2　Java 到 SQL 数据类型影射表

JAVA 数据类型	SQL 数据类型
String	VARCHAR or LONGVARCHAR
java.math.BigDecimal	NUMERIC
Boolean	BIT
Byte	TINYINT
Short	SMALLINT
Int	INTEGER
Long	BIGINT
Float	REAL

Double	DOUBLE
byte[]	VARBINARY or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

表 2 说明了 Java 语言中的各种数据类型在 SQL 中的默认表示。

下面，我们就几种常用的数据类型之间的转化进行说明。

①CHAR, VARCHAR, 和 LONGVARCHAR

在 SQL 语言中，有三种分别表示不同长度的字符类型 CHAR, VARCHAR, 和 LONGVARCHAR，在 Java 中并没有相应的三种不同的数据类型与之一一对应，JDBC 的处理方法是将其与 String 或者 char[]对应起来。在实际编程中不必对这三种 SQL 数据类型进行区分，全部将它们转化为 Sting 或者 char[]就可以了。通常使用应用得非常普遍的 String 类型。我们还可以利用 String 类提供的方法将一个 String 对象转化为 char[]，或者用 char[]为参数构造一个 String 对象。

对于定长度的 SQL 数据类型 CHAR(n)，当从数据库管理系统中获得的结果集提取该类型的数据时，JDBC 会为其构造一个长度为 n 的 String 对象来代表它，如果实际的字符个数不足'n', 系统会自动为 String 对象补上空格。当向数据库管理系统写入的数据类型应该是 CHAR(n)时，JDBC 也会将该 String 对象的末尾补上相应数量的空格。

一般情况下，CHAR, VARCHAR, LONGVARCHAR 和 String 之间可以无差错的进行转换。但值得注意的是，LONGVARCHAR 这种 SQL 的数据类型有时在数据库中代表的数据可能有几兆字节的大小，超过了 String 对象的承受范围。JDBC 解决的办法是用 Java 的 Input Stream 来接受这种类型的数据。Input Stream 不仅支持 ASCII，而且支持 Unicode，我们可以根据需要进行选择。

②DECIMAL 和 NUMERIC

SQL 的 DECIMAL 和 NUMERIC 通常用来表示需要一定精度的定点数。在 Java 的简单数据类型中，没有一种类型与之相对应。但从 JDK1.1 开始，Sun 公司在 java.math.*包中加入了一个新的类 BigDecimal，该类的对象可以与 DECIMAL 、NUMERIC 进行转换。

另外，当从数据库管理系统中读取数据时，还可以用 getString()方法来获取 DECIMAL 和 NUMERIC。

③BINARY, VARBINARY, 和 LONGVARBINARY

在编程时无须精确区分这三种 SQL 数据类型，JDBC 将它们统一影射为 byte[]。其中 LONGVARBINARY 和 LONGVARCHAR 相似，可以代表几兆字节的数据，超出数组的承受范围。解决的办法依然是用 Input Stream 来接受数据。

④BIT

代表一个二进制位的 BIT 类型被 JDBC 映射为 boolean 型。

⑤TINYINT, SMALLINT, INTEGER, 和 BIGINT

SQL 语言的 TINYINT, SMALLINT, INTEGER, 和 BIGINT 分别代表 8 位、16 位、32 位、64 位的数据。它们分别被映射为 Java 的 byte、 short、 int 和 long。

⑥REAL, FLOAT, 和 DOUBLE

SQL 定义了 REAL, FLOAT, DOUBLE 来支持浮点数。JDBC 将 REAL 映射到 Java 的 float,将 FLOAT，DOUBLE 映射到 java 的 double。

⑦DATE, TIME, 和 TIMESTAMP

SQL 定义了三种和日期相关的数据类型。 DATE 代表年、月、日，TIME 代表时、分、秒，TIMESTAMP 结合了 DATE 和 TIME 的全部信息，而且增加了更加精确的时间计量单位。

在 java 的标准类库中，java.util.*包中的 Date 类用来表示日期和时间。但是该类和 SQL 中的 DATE, TIME, 和 TIMESTAMP 直接映射关系并不清晰。并且，该类也不支持 TIMESTAMP 的精确时间计量单位。因此，Sun 公司在 java.sql.*中为 java.util.Date 增加了三个子类：java.sql.Date，java.sql.Time ，java.sql.Timestamp，分别与 SQL 中的三个日期数据类型相对应。

(3) 利用 InputStream 对象读取较长的值

在上一部分,介绍了用 getBytes()和 getString()方法来读取 LONGVARCHAR 和 LONGVARBINARYDE 的值,但长度有一个上限。这个上限就是 Statement 对象中的 getMaxFieldSize()所返回的值，并且可以用 Statement 类的 setMaxFieldSize()方法来重新设置这个上限值。但是，有时要在上限较小的情况下读取较长的数据。这时 ResultSet 类可以返回 java.io.InputStream 对象，用户可以从这个输入流对象中读取数据。这个流在返回后应该立即读取，在执行 ResultSet 对象的 next()方法后，这个流将被关闭。

从 Java 流中读出的是无类型的字节，可以作为 ASCII 和 Unicode 来使用。JDBC 提供了三种独立的方法来获得流：getBinaryStream 返回的流仅提供从数据库读取的原始字节，而不作任何转换。GetAsciiStream 返回的流提供一个字节的 ASCII 字符。GetUnicode 返回的流提供两个字节的 Unicode 字符。

下面是一个使用 ASCII 流的例子：

```
Statement statement=con.createStatement();
//在 Connection 对象的基础上创建 Statement 对象。
String sql=”SELECT * FROM table_name”;
ResultSet resultset=statement.execute(sql);
//用 Statement 对象执行 SQL 语句，返回 ResultSet 对象。
byte   buffer=new byte[2048];
//构造 byte 数组用于接收数据。
while(rusultset.next())
{
    InputStream in=resultset.getAsciiStream(1);
```

```
    boolean flag=false;
    while(flag)
    {
        int size=in.read(buffer);
        if(size==0)
        {
            flag=false;
        }//if
        for(int i=0; i<buffer.size(); i++)
        {
            out.println(buffer[I]);
        }
    }//while
}//while
```

(4) 获取结果集的信息

在对数据库的表结构已经了解的前提下，可以知道返回的结果集中各个列的一些情况，如列名、数据类型等。有时并不知道结果集中各个列的情况，这时可以使用 Resultset 类的 getMetaData 方法来获取结果集的信息：

```
ResultsetMetaData rsdata=resultset.getMetaData();
```

GetMetaData()方法返回一个 ResultSetMetaData 类的对象，使用该类的方法，得到许多关于结果集的信息，下面给出几个常用的方法：

①getColumnCount()返回一个 int 值，指出结果集中的列数。

②getTableName(int column)返回一个字符串，指出参数中所代表列的表名称。

③getColumnLabel(int column)返回一个 String 对象，该对象是 column 所指的列的显示标题。

④getColumnName（int column）返回的是该列在数据库中的名称。可以把此方法返回的 String 对象作为 Resultset 类的 getXXX()方法的参数。不过，并没有太大的实际意义。

⑤getColumnType(int comlumn)返回指定列的 SQL 数据类型。它的返回值是一个 int 值。在 java.sql.Types 类中有关于各种 SQL 数据类型的定义。

⑥getColumnTypeName(int comlumn)返回指定列的数据类型在数据源中的名称。它的返回值是一个 String 对象。

⑦isReadOnly(int column) 返回一个 boolean 值，指出该列是否是只读的。

⑧isWriteable(int column) 返回一个 boolean 值，指出该列是否可写。

⑨isNullable（int column）返回一个 boolean 值，指出该列是否允许存入一个 NULL 值。

前面几小节涉及到了如何在数据库中建表、插入数据，在此基础之上，本小节将细致地讲述如何对数据库进行更新操作，包括修改、插入和删除记录，创建和删除表、以及增加和删除某列。这些操作对应于 SQL 语句中的 INSERT、UPDATE、DELETE 和 CREATE、DROP 等操作。对于数据库更新操作也是在一个 Statement, PreparedStatement,或 CallableStatement 对象上完成的，但不是使用 executeQuery()方法，而是使用 executeUpdate()

方法。

4.2.3.13 SQLException 对象

SQLException 类是用来通知那些没有执行成功的或者没有得到完全执行的 SQL 语句。当一个方法抛出一个 SQLException 对象时，就意味着该方法在访问数据库时出现了错误。当方法的注释部分没有抛出异常这一条时，那么该方法所抛出的 SQLException 就是访问数据时出错而引发的。

警告是由一个单独的处理机制来处理的，在这种处理机制中，警告和提出 SQL 请求的对象相联。SQLWarning 类继承于 SQLException 类，它代表了 JAVA 语言中的一种警告。

SQLException 对象

每一个 SQLException 对象中都包含了如下的几类信息：

A：对错误的描述。这是一个 String 对象，它用来作为 JAVA 异常消息，可以通过调用 getMessage 方法来获取这个 String 对象，getMessage 方法由继承 Throwable 的相应方法而来。

B：识别该异常的 SQLstate 的字符串。它的值取决于底层的数据源。SQL99 定义了 SQLstate 和对应于这些状态值的相应条件。

应用程序通过调用 SQLException.getSQLstate 方法来获取 SQLstate 字符串，可以通过调用 DatabaseMetaData 类中的 getSQLStateType 方法来查看所返回的 SQLstate 的数据源标准。

C：识别导致抛出 SQLException 对象的错误编号。这个错误代号是一个整数，它根据数据库的错误码返回值确定。应用程序可以通过调用 SQLException 的 getErrorCode 方法来获取错误代号。

D：指向下一个 SQLException 对象的引用。如果存在异常链，应用程序可以通过调用 SQLException 方法的 getNextException 来获取异常链中的下一个异常。当异常链中没有异常时，返回 null。

获取 SQLException 信息

SQLException 的相关信息是通过获取 SQLException 类的每一个组件来获取的。getMessage 方法返回该异常的描述信息。SQLException 对象中定义了三个方法来获取其它组件：通过 getSQLState 方法获取 SQLState 值；通过 getErrorCode 方法来获取数据库所使用的错误编码；通过 getNextException 方法来获取连接到该异常的异常。

如下所示：

```
Catch(SQLException ex){
    System.out.printf("\n----------SQLException caught------\n");
    Do{
        System.out.println("SQLState:"+ex.getSQLState());
        System.out.println("Message:"+ex.getMessage());
        System.out.println("Vendor code:"+ex.getErrorCode());
        Ex = ex.getNextException();
    }while(ex != NULL)
```

```
}
```

程序抛出一个 SQLException 并不一定意味着产生该异常的方法没有被执行。所以程序员不应该根据异常来假定一个事务所处的状态。一旦有不能确定的状态的情况，最安全的方法就是调用 ROLLback 方法并重新执行语句。

5 新增错误类型

DosSQL 支持原有的 MySQL 错误类型，在此基础上增加了一些错误类型。

5.1 SQL 语句中使用多个数据库

DosSQL 中禁止同时操作多个数据库，当 SQL 语句中包含了多个数据库时，语句将会执行失败，并将错误消息返回给用户。

错误消息：“Sorry, Current version refuse to operate other or multi databases”

错误号：1105

5.2 创建伪线程失败

当本地节点不是数据库的控制节点时，需要创建伪线程来执行，如果由于网络通信失败、消息包丢失，或是分配内存失败造成的创建失败，造成的创建伪线程失败，这时将返回错误消息给用户。

错误消息：“Request to create disguise thread got a error”

错误号：1290

5.3 伪线程执行失败

伪线程执行事务是通过分布式方式执行的, 因此可能由于通信和节点发生故障的原因，造成执行的失败。

错误消息：“Disguise execute command got a error”

错误号：1290

5.4 数据库名长度不合法

用户输入的数据库名称超过了最大长度的限制。

错误消息：“Database name too long!”

错误号：1047

5.5 修改数据库属性

禁止用户在数据库为非 OK 的情况下去修改该数据库的属性。

错误消息：“Sorry, There is some dups not on DB_OK_STATE!”

错误号：1105

5.6 目录重构

目录在故障重构的过程中，服务器暂时拒绝提供服务，目录重构的时间大概为 1 秒到几秒。

错误消息：“database can`t execute because recoverying”

错误号：1290

5.7 执行不支持的命令

错误消息：“this version not supported it”

错误号：1235

5.8 备份数据库

备份数据库时，不能提供服务，停止服务的时间根据需要备份的数据量而定，一般 500M 的数据库需要 10 多秒钟。

错误消息：“database are stop execute because recovery”

错误号：1290

6 常见问题解决

6.1 注意事项

- 编译应用程序时加载库文件

使用 API 的应用程序，编译时需要指明用到的动态链接库或者静态库，否则编译时会报错。

- 与 MySQL 的兼容问题

一般基于 MySQL 的 API 开发的应用程序可以直接在 DosSQL 运行，但是有些特性 DosSQL 尚不支持，遇到在 DosSQL 上无法正常运行的 MySQL 应用程序时，请先检查其是否使用了 DosSQL 不支持的特性。

- 释放内存资源

一些接口在使用结束后需要释放内存资源，否则将造成内存溢出。

- WEB 应用

开发和部署 WEB 应用时，需要使用由成都天心悦高科技发展有限公司提供的对应版本的 JDBC 链接库。

6.2 常见问题与解答

(1) 问题：调用 mysql_connect()函数时，不能连接数据库，可能有哪些原因？

答案：有以下几种常见的原因：

1) 服务端异常，常见的是没有开启服务器，或者是没有开启 DosSQL 程序；
2) 函数的参数错误，没有需要连接的数据库，或者是给出了错误的用户名和密码；
3) 连接超时，可能是设置的超时时间过短，也可能是名字解析时间过长而导致超时。

解决的办法就是根据以上的可能原因逐步检查，并找到连接失败的原因，如果是服务端异常，则需要开启服务器并开启 DosSQL；如果是开发者使用 API 时给出了错误的参数，就改正为正确的参数；如果是超时的原因，则需要加大连接超时的时间限制，也可以在 Linux 系统的“/etc/hosts”文件中添加客户主机的地址来加快名字解析。

(2) 问题：PHP 编程时超出了最大执行时间，怎么处理？

答案：这是一种 PHP 限制，如果需要就进入文件 php.ini，并设置最大执行时间（开始为 30 秒）。此外，还可以将每脚本允许使用的 RAM 增加一倍，从 8MB 变为 16MB，这也是个不错的主意。

(3) 问题：对'uncompress'的未定义引用，怎么处理？

答案：这意味着所编译的客户端库支持压缩客户端 / 服务器协议。更正方法是，用“-lmysqlclient”进行链接时，在最后添加“-lz”。

(4) 问题：为什么在 mysql_query()返回成功后，mysql_store_result()有时会返回 NULL?

答案：成功调用 mysql_query()后，mysql_store_result()能够返回 NULL。出现该情况时，表明出现了下述条件之一：

- 出现了 malloc()故障（例如，如果结果集过大）。
- 无法读取数据（在连接上出现了错误）。
- 查询未返回数据（例如，它是 INSERT、UPDATE 或 DELETE）。

通过调用 mysql_field_count()，一定能检查出语句是否应生成非空结果。如果上一个查询是未返回值的语句（例如 INSERT 或 DELETE），当 mysql_field_count()返回 0，则结果为空，如果 mysql_field_count()返回非 0 值，则语句应生成非空结果。关于这方面的示例，请参见 mysql_field_count()函数介绍。

通过调用 mysql_error()或 mysql_errno()，可测试是否出现了错误。

(5) 问题：从查询 SQl 中能得到什么结果？

答案：除了查询返回的结果集外，还能获取下述信息：

- 执行 INSERT、UPDATE 或 DELETE 时，mysql_affected_rows()返回上次查询影响的行数。对于快速创建，请使用 TRUNCATE TABLE。
- mysql_num_rows()返回结果集中的行数。使用 mysql_store_result()，一旦 mysql_store_result()返回，就能调用 mysql_num_rows()。使用 mysql_use_result()，仅当用 mysql_fetch_row()获取了所有行后，才能调用 mysql_num_rows()。
- mysql_insert_id()返回上次查询生成的 ID，该查询使用 AUTO_INCREMENT 索引将行插入到表内。
- 某些查询（LOAD DATA INFILE ...、INSERT INTO ... SELECT ...、UPDATE）将返回额外信息。结果由 mysql_info()返回。关于它返回的字符串格式，请参见关于 mysql_info()的介绍。如果没有额外信息，mysql_info()将返回 NULL 指针。

(6) 问题：如何获得上次插入行的唯一 ID?

答案：如果将记录插入包含 AUTO_INCREMENT 列的表中，通过调用 mysql_insert_id()函数，可获取保存在该列中的值。

通过执行下述代码，可从 C 应用程序检查某一值是否保存在 AUTO_INCREMENT 列中（假定该语句已成功执行）。它能确定查询是否是具有 AUTO_INCREMENT 索引的 INSERT：

```
if ((result = mysql_store_result(&mysql)) == 0 &&
 mysql_field_count(&mysql) == 0 &&
 mysql_insert_id(&mysql) != 0)
{
 used_id = mysql_insert_id(&mysql);
```

}

生成新的 AUTO_INCREMENT 值时，也能与 mysql_query()一起通过执行 SELECT LAST_INSERT_ID()语句获得它，并从该语句返回的结果集检索该值。

对于 LAST_INSERT_ID()，最近生成的 ID 是在服务器上按连接维护的，它不会被另一个客户端改变。即使用 non-magic 值（即非 Null 非 0 值）更新了另一个 AUTO_INCREMENT 列，也不会更改它。

如果打算使用从某一表生成的 ID，并将其插入到第 2 个表中，可使用如下所示的 SQL 语句：

```
INSERT INTO foo (auto,text)
 VALUES(NULL,'text');   # generate ID by inserting NULL
INSERT INTO foo2 (id,text)
 VALUES(LAST_INSERT_ID(),'text');  # use ID in second table
```

注意，mysql_insert_id()返回保存在 AUTO_INCREMENT 列中的值，无论该值是因存储 NULL 或 0 而自动生成的，或是明确指定的，均如此。LAST_INSERT_ID()仅返回自动生成的 AUTO_INCREMENT 值。如果你保存了除 NULL 或 0 之外的确切值,不会影响 LAST_INSERT_ID()返回的值。

(7) 问题：与 C API 链接时，在某些系统上可能出现下述错误：

```
gcc -g -o client test.o -L/usr/local/lib/mysql -lmysqlclient -lsocket -lnsl

Undefinedfirst referenced
 symbol  in file
floor /usr/local/lib/mysql/libmysqlclient.a(password.o)
ld: fatal: Symbol referencing errors. No output written to client
```

答案：如果在你的系统上出现了该情况，必须在编译/链接行的末尾增加“-lm”，通过该方式包含数据库。