第6章 MySQL C API

MySQL提供用C编程语言编写的客户机库，可以用它编写访问 MySQL的客户机程序。这个库定义了应用程序编程接口，包括下面的实用程序：

- 建立和终止与服务器会话的连接管理例程。
- 构造查询的例程，将例程发送到服务器，并处理结果。
- 当其他C API调用失败时，确定错误准确原因的状态和错误报告函数。

本章介绍如何用客户机库编写自己的程序。我们要记住的一些要点是，自己的程序与MySQL 分发包中已有的客户机程序的一致性，代码的模块性和可重用性。本章假设您知道用C编程的一些知识，但并不一定是专家。

本章从简单到复杂粗略地开发了一系列的客户机程序。这个过程是第一部分开发了一个程序框架，该框架除了与服务器连接和断开以外不能作任何事情。这样做的原因是，尽管MySQL 客户机程序是为不同的目的而编写的，但它们都有一个共同点：即创建与服务器的连接。

我们将用以下步骤来建立这个程序框架：

1. 编写一些连接和断开的简要代码（客户机程序 1）。
2. 增加一些错误检查（客户机程序 2）。
3. 使连接代码模块化和可重用化（客户机程序 3）。
4. 增加获取运行时连接参数的能力（主机，用户，口令）（客户机程序4）。

这个框架一般是合理的，可以使用它作为编写任意数量的客户机程序的基础。开发它以后，我们将暂不考虑如何处理各种问题。首先，我们将讨论如何处理特定的硬编码的 SQL语句，然后再开发用于处理任意语句的代码。在这之后，将查询处理的代码增加到客户机程序框架中，开发另一个程序（客户机程序 5），它类似于 mysql 客户机程序。

我们也将考虑（并解决）一些通用的问题，如“ 如何获取有关表的结构信息？ ”和“ 如何在数据库中插入图像？ ”

只有在需要时，本章才讨论客户机库的函数和数据类型。要想了解所有函数和类型的列表，请参阅附录 F“ C API 参考 ”。可以用这个附录作为使用客户机库任何部分的进一步的背景信息的参考。

样例程序可以由联机下载得到，可以直接使用，而不必再键入它们。有关的指导，请参阅附录 A“ 获得和安装软件 ”。

> **在哪里寻找样例**
>
> MySQL 邮件清单的一个共同问题就是“ 我在哪里可以找到一些用 C 写的客户机样例？ ”。当然，这个答案是“ 就在本书里！ ”。但是，许多人好像并没有考虑的是 MySQL 分发包中包括了若干客户机程序（例如 mysql、mysqladmin 和 mysqldump），这些大部分都是用C编写的。因为这个分发包可以很容易地以源程序形式使用，所以 MySQL 提供非常少的样例客户机代码。因此，如果您还没有这样做，找个时间找到源程序分发包，在

客户机目录中查看这些程序。MySQL 客户机程序为共享软件，从那里可以为自己的程序自由地借用代码。

在本章提供的样例和 MySQL 分发包中包括的客户机程序之间，可以找到与自己编写程序时想做的事情相类似的代码。如果是这样，可以通过拷贝和修改已有的程序来重新使用代码。应该阅读本章，了解客户机库是如何工作的。然而，请记住，并不总是需要自己编写琐碎的每件事情（您将注意到，在本章编写程序的讨论中，代码的可重用性是目的之一）。通过使用其他人编好的程序，可以避免许多工作，那是最好的。

6.1 建立客户机程序的一般过程

本节介绍使用MySQL客户机库编译和连接程序所包括的步骤。不同的系统建立客户机程序的命令也有所不同，可能需要稍微修改一下这里介绍的命令。然而，这里的说明是通用的，应该能够将它用于几乎您编写的任何客户机程序中。

6.1.1 基本的系统需求

当您用 C 编写 MySQL 客户程序时，显然将需要一个 C 编译程序。这里说明的样例使用 gcc。除了自己的源文件以外，还将需要下列程序：

- MySQL 头文件。
- MySQL 客户机库。

MySQL 头文件和客户机库组成客户机编程的支持程序。它们可能已经安装到您的系统上。如果没有，应获取它们。如果 MySQL 从源程序分发包或二进制分发包中安装，则客户机可编程的支持程序应该已经作为该处理的一部分安装了。如果 MySQL 是从 RPM 文件中安装的，则除非安装了开发程序 RPM，否则就没有这种支持。如果需要安装 MySQL 头文件和库，请参阅附录 A。

6.1.2 编译和连接客户机程序

要想编译和连接客户机程序，就必须指定 MySQL 头文件和客户机库的位置，因为它们通常不安装在编译程序和连接程序缺省搜索的位置。对于下面的样例，假定头文件和客户机库的位置为 /usr/local/include/mysql 和 /usr/local/lib/mysql。

要想告知编译程序如何寻找 MySQL 头文件，则当将源文件编译为目标文件时，传送给它一个 -I/usr/local/include/mysql 参数。例如，可以使用这样的命令：

```
% gcc -c -I/usr/local/include/mysql myclient.c
```

要想告知连接程序在哪，可以找到客户机库和它的名称，当连接目标文件产生一个可执行的二进制文件时，传送 -L/usr/local/lib/mysql 和 -lmysqlclient 参数，如下所示：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient
```

如果客户机程序包括多个文件，则所有目标文件的名称都要列在连接命令上。如果连接步骤导致不能找到必需的 floor() 函数的错误，则通过在命令行的后面增加 -lm，连接到数学库：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient -lm
```

可能还需要增加其他的库。例如，在 Solaris 上可能需要 -lsocket -lnsl。

如果没有使用 make 建立程序，则建议您了解一下如何进行，以便不必手工地键入许多建立程序的命令。假设有一个客户机程序 myclient，包括两个源文件 main.c 和 aux.c，及一个头文件 myclient.h。一个简单的 Makefile 建立这个程序的代码，如下所示：

```
CC = gcc
INCLUDES = -I/usr/local/include/mysql
LIBS = -L/usr/local/lib/mysql -lmysqlclient

all: myclient

main.o: main.c myclient.h
    $(CC) -c $(INCLUDES) main.c
aux.o: aux.c myclient.h
    $(CC) -c $(INCLUDES) aux.c

myclient: main.o aux.o
    $(CC) -o myclient main.o aux.o $(LIBS)

clean:
    rm -f myclient main.o aux.o
```

如果是一个需要连接到数学库的系统，则更改 LIBS 的值，并将 -lm 加到最后：

```
LIBS = -L/usr/local/lib/mysql -lmysqlclient -lm
```

如果需要其他的库，如 -lsocket 和 -lnsl，则也要将这些库加到 LIBS 中。

使用 Makefile，无论何时修改何源文件，只简单地键入“ make ”就可以重新建立程序代码。那比键入一句长的 gcc 命令更容易，发生错误更少。

6.2 客户机程序1 —— 连接到服务器

我们的第一个 MySQL 客户机程序很简单：连接到服务器、断开，并退出。它本身并不是非常有用，但是必须知道如何做它，因为实际上用 MySQL 数据库做任何事情都必须与服务器连接。这是一个公用的操作，开发创建连接的代码是编写每个客户机程序都将使用的代码。除此之外，这项任务带给我们一些简单开始的事情。以后，我们可以增加这个客户机来做一些更有用的事情。

我们第一个客户机程序的源代码，客户机程序 1，包括一个单独的文件， client.c：

```
/* client1.c */

#include <stdio.h>
#include <mysql.h>

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_db_name     NULL /* database to use (default = none) */

MYSQL   *conn;              /* pointer to connection handler */

int
main (int argc, char *argv[])
{

    conn = mysql_init (NULL);
    mysql_real_connect (
                conn,           /* pointer to connection handler */
```

```
                def_host_name,  /* host to connect to */
                def_user_name,  /* user name */
                def_password,   /* password */
                def_db_name,    /* database to use */
                0,              /* port (use default) */
                NULL,           /* socket (use default) */
                0);             /* flags (none) */
    mysql_close (conn);
    exit (0);
}
```

这个源文件以包括 stdio.h 和 mysql.h 开始。MySQL 客户机可能包括其他的头文件，但是一般来说至少有两个是最基本的。

主机名称、用户名称、口令和数据库名称的缺省值固定在代码内部，使事情变得简单。以后，我们将参数化这些值，以便可以在选项文件或命令行中指定它们。

程序的 main() 函数创建和终止与服务器的连接。建立连接使用如下两个步骤：

1) 调用 mysql_init() 来获取连接处理程序。MYSQL 数据类型是一个包括连接信息的结构。这种类型的变量称为连接处理程序。当我们将 NULL 传递给 mysql_init() 时，它分配一个 MYSQL 变量，初始化它，然后返回一个指向它的指针。

2) 调用 mysql_real_connect() 来创建与服务器的连接。mysql_real_connect() 可有任意数量的参数，例如：

- 连接处理程序的指针。这不能为 NULL；它应该是由 mysql_init() 返回的值。
- 服务器主机。如果指定 NULL 或主机“ localhost ”，则客户机连接到在本地主机使用 UNIX 套接字运行的服务器上。如果指定一个主机名称或主机的 IP 地址，则客户机连接到使用 TCP/IP 连接命名的主机上。

在 Windows 上，除了使用 TCP/IP 连接而不是用UNIX 套接字以外，这种操作是类似的（在 Windows NT 上，如果主机为 NULL，则在 TCP/IP 以前，先试着使用一个指定的管道来连接）。

- 用户名称和口令。如果名称为 NULL，则客户机库将逻辑名称发送给服务器。如果口令为 NULL，则不发送口令。
- 端口号和套接字文件。这些指定为 0 或 NULL，来告知客户机库使用它的缺省值。如果不指定端口和套接字，则根据希望连接到的主机确定这些缺省值。附录 F 中的 mysql_real_connect() 的描述给出有关这些的详细情况。
- 标志值。因为我们不使用任何特定的连接操作，因此它是 0。这个参数可用的选项在附录 F 中的 mysql_real_connect() 的项目中讨论详细情况。

要想终止这个连接，可将连接处理程序的指针传递给 mysql_close()。当将连接处理程序传递给 mysql_close() 来终止这个连接时，由 mysql_init() 自动分配的连接处理程序自动地释放。

要想测试客户机程序 1，可使用本章前面建立客户机程序时给出的指导来编译和连接，然后运行它：

```
% client1
```

程序连接到服务器、断开并退出。这一点都不令人兴奋，但它是一个开始。然而，它只是一个开始，因为有两个重要的缺点：

- 客户机没有错误检查，所以并不真正地知道实际上它是否在工作！
- 连接参数（主机名称，用户名称等）在源代码内部固定。如果允许用户通过指定选项文件或命令行中的参数来解决这个问题则更好一些。

这些问题的处理都不困难。我们将在下面专门解决它们。

6.3 客户机程序2 —— 增加错误检查

我们的第二个客户机程序将像第一个客户机程序一样，但是将修改它们，考虑错误出现的可能性。“ 将错误检查作为读者的练习 ” 这样的项目在编程文献中相当常见，这或许是因为检查错误相当令人讨厌。但是，我赞同这种观点，即 MySQL 客户机程序应该测试错误条件并适当地进行回应。由于某种原因，返回状态值的客户机库的调用做这些事情，而且您要承担忽略它们的后果。您最终还是要试图捕获由于没有错误检查而出现在程序中的错误，这些程序的用户会对程序运行如此不规律感到奇怪。

考虑我们的程序，客户机程序 1。如何知道它是否真正连接到服务器上？可以通过查看服务器的日志，找出与运行程序时间相应的 Connect和Quit事件：

```
030805 21:32:14       20 Connect    paul@localhost on
                      20 Quit
```

或者，可以查看Access denied 消息：

```
030805 22:31:47       21 Connect    Access denied for user: 'paul@localhost'
                                    (Using password: NO)
```

这条消息表示根本没有创建连接。不幸的是，客户机程序 1没有告诉我们出现的这些结果。实际上它不能。它不能实现任何错误检查，所以它甚至不知道自己发生了什么事。无论如何，当然不一定必须查看日志来寻找是否能连接到服务器！让我们立刻改正它。

在 MySQL 客户机库中返回值的例程基本上以下列两种方式之一表示成功或失败：

- 成功时，值的指针函数返回一个非 NULL 指针，失败时返回 NULL（在这里 NULL 的意思是 “ C NULL 指针 ”，而不是 “ MySQL NULL 列值 ”）。

  迄今为止，我们使用的客户机库的例程 mysql_init() 和 mysql_real_connect() 都用返回连接处理程序的指针来表示成功， NULL 表示失败。
- 整型数值的函数一般成功返回 0，失败返回非 0。不要测试特定的非 0值，如 -1。因为当失败时，并不保证客户机库函数返回任何特定的值。有时，您可能会看到像如下的较旧的错误地测试返回值的代码：

```
if (mysql_XXX() == -1)          /* this test is incorrect */
    fprintf (stderr, "something bad happened\n");
```

这个测试可能工作，也可能不工作。 MySQL API 不将任何非0错误的返回指定为特定的值，而只判断它（显然地）是否为 0。这个测试应该写成下面两段之一：

```
if (mysql_XXX())            /* this test is correct */
    fprintf (stderr, "something bad happened\n");
```

或如下所示：

```
if (mysql_XXX() != 0)       /* this test is correct */
    fprintf (stderr, "something bad happened\n");
```

这两个测试是等价的。如果审核 MySQL 的源代码，则可以发现，它基本上用第一种形式测试，因为这编写起来更简短。

不是每个 API 调用都返回值。我们使用的另一个客户机例程 mysql_close() 就不返回值（它如何失败？失败了又如何？无论如何，都要进行连接）。

当客户机库调用失败，并且需要有关失败的详细信息时， API 中的两个调用都是有用的。mysql_error() 返回包括错误信息的字符串，而 mysql_errno() 返回数值代码。应该在错误出现以后立刻调用它们，因为如果发布另一个返回状态的 API 调用，则从 mysql_error() 或 mysql_errno() 获取的任何错误信息都将来自于后面的调用。

一般来说，程序的用户查看错误字符串比查看错误代码更有启发。如果只报告两者中的一个，则建议报告字符串。出于全面考虑，本章的这个样例报告两个值。

考虑前述的讨论，我们将编写第二个客户机程序，即客户机程序 2。它类似于客户机程序 1，但是适当地增加了错误检查代码。源文件 client2.c 如下所示：

```
/* client2.c */

#include <stdio.h>
#include <mysql.h>

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_db_name     NULL /* database to use (default = none) */

MYSQL   *conn;              /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    conn = mysql_init (NULL);
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
        exit (1);

    }
    if (mysql_real_connect (
                conn,           /* pointer to connection handler */
                def_host_name,  /* host to connect to */
                def_user_name,  /* user name */
                def_password,   /* password */
                def_db_name,    /* database to use */
                0,              /* port (use default) */
                NULL,           /* socket (use default) */
                0)              /* flags (none) */
            == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                         mysql_errno (conn), mysql_error (conn));
        exit (1);
    }
    mysql_close (conn);
    exit (0);
}
```

这个错误检查的逻辑是，如果失败，则 mysql_init() 和 mysql_real_connect() 都返回 NULL。请注意，尽管这个程序检查 mysql_init() 返回的值，但是，如果它失败，却不调用错误报告函数。这是因为当 mysql_init() 失败时，不能假设连接处理程序包括任何有意义的信息。

相反，如果 mysql_real_connect() 失败了，则连接处理程序并不反映有效的连接，但是的确包括传送给错误报告函数的错误信息（不要将该处理程序传送给任何其他的客户机例程！因为它们一般假设是一个有效连接，所以您的程序可能崩溃）。

编译和连接客户机程序2，然后试着运行它：

```
% client2
```

如果客户机程序2没有别输出，则连接成功。另一方面，可能会如下所示：

```
% client2
mysql_real_connect() failed:
Error 1045 (Access denied for user: 'paul@localhost' (Using password: NO))
```

这个输出表示没有创建连接，并说明为什么。或者，它还表示我们的第一个程序，即客户机程序1，没有成功地连接到服务器（毕竟客户机程序 1使用同样的连接参数）！而在那时我们不知道，因为客户机程序 1没有错误检查。而客户机程序 2做检查，所以当出问题时，它可以告知我们。这就是应该始终测试 API 函数返回值的原因。

MySQL 邮件清单问题经常是与错误检查有关的。典型的问题是“当发送这个查询时，为什么我的程序崩溃了？”或“我的程序怎么没有返回任何东西？”在许多情况下，在查询发布以前，有疑问的程序不检查在发布该查询前是否成功地建立了连接，或者不检查在试着检索结果前确保服务器成功执行该查询。不要假定每个客户机库都调用成功。

本章下面的例子完成错误检查，而且也应该这样。看起来它好像有更多的工作，但是从长远地运行来看，它的工作实际上是少的，因为您化费了更少的时间来捕获错综复杂的问题。在第7章“Perl DBI API”和第8章“PHP API”中，也使用这种检查错误的方法。

现在，当运行客户机2的程序时，假设看到拒绝访问（Access denied）的消息。如何改正这个问题呢？一种可能是将主机名称、用户名称和口令的 #define 行更改为允许访问服务器的值。这是有好处的，在这个意义上，至少应该能做一个连接。但是，这些值是程序中的固定编码。所以笔者建议不要用这种方法，特别是对口令值。当将自己的程序编译为二进制格式时，您可能认为口令隐藏起来了，但是，如果有人在程序上运行 strings，则它根本隐藏不住（更不用说明读取访问源文件的人根本不用做一点工作，就可以获取口令）。

在“客户机程序4——运行时获取连接参数”一节中我们将处理访问的问题。首先，笔者想说明编写连接代码的一些其他方法。

6.4 客户机程序3——产生连接代码模块

对于我们的第三个客户机程序，即客户机程序 3，通过将它封装到函数 do_connect() 和 do_disconnect() 中，将使连接和断开代码更加模块化，这样可以很容易地由多个客户机程序使用。这提供一种选择，可将连接代码精确地嵌入到 main() 函数中。无论如何，对在应用程序过程中套用老调的任何代码都是一个好主意。将它放在可以通过多个程序访问的函数中，而不是在每个程序中都编写一遍。如果修正这个函数中的一个错误或对这个函数作了一些改进，则可只更改一次，只要重新编译就可以使用这个函数的所有程序都被修正或利用这种改进。同样，编写一些客户机程序，以便在它们执行过程中可以若干次地连接和断开。如果将安装和卸载方法放在连接和断开的函数中，则编写这样一个客户机更加容易。

封装策略如下所示：

1) 将公用代码分离到一个独立的源文件（common.c）的包装函数中。

2) 提供一个头文件，common.h，其中包括该公共例程的原型。

3) 在使用公共例程的客户机源文件中包括 common.h。

4) 将公共源文件编译成目标文件。

5) 将公共目标文件连接到您的客户机程序中。

用这些策略，让我们构造 do_connect() 和 do_disconnect()。

do_connect() 代替对 mysql_init() 和 mysql_real_connect() 的调用，并替换错误打印的代码。除了不传递任何连接处理程序外，您可以像 mysql_real_connect() 一样调用它。do_connect() 分配并初始化这个处理程序，然后，在连接后返回一个指向它的指针。如果 do_ connect() 失败，则在打印一个错误消息以后，返回NULL（那就是说，调用 do_connect() 并获取返回值 NULL 的任何程序都可以简单地退出，而不用担心打印消息的本身）。do_ disconnect () 产生一个指向连接处理程序的指针，并调用 mysql_close ()。这里是 common.c 的代码：

```
#include <stdio.h>
#include <mysql.h>
#include "common.h"

MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
MYSQL   *conn;  /* pointer to connection handler */

    conn = mysql_init (NULL);   /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
    if (mysql_real_connect (conn, host_name, user_name, password,
                        db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                          mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    return (conn);          /* connection is established */
}

void
do_disconnect (MYSQL *conn)
{
    mysql_close (conn);
}
```

- common.h 声明common.c 中这些例程的原型：

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags);
void
do_disconnect (MYSQL *conn);
```

要想访问公共例程，应在源文件中包括 common.h。请注意，common.c 同样包括 common.h。那就是说，如果 common.c 中的函数定义与头文件中的声明不匹配，则立即得到一个编译程序警告。同样，如果更改 common.c 中的调用次序而没有相应地更改 common.h，

则当重新编译 common.c 时，编译程序将发出警告。

有人会问为什么要发明包装函数 do_disconnect()，而它使用得还这么少。do_disconnect() 和 mysql_close() 等价。但是假设在断开连接时，都有一些要执行的额外清除。则通过调用已经完全控制的包装函数，可以修改该包装函数来做需要的事情，对于所做的任何断开的操作，这种更改统一生效。如果直接调用 mysql_close()，则不能做到这点。

在前面，笔者声称对在多个程序中或在单个程序内部多处使用的函数中，将代码封装成模块化代码是有好处的。前面介绍一个理由，还有一些理由参见下面的两个样例。

■ **样例1** 在 MySQL3.22以前的版本中，mysql_real_connect() 调用与它现在稍微有些不同：即没有数据库名称参数。如果想利用旧的 MySQL 客户机库使用 do_connect()，则它不能工作。然而，可以修改 do_connect()，使它可在3.22版以前的版本上运行。这就意味着，通过修改 do_connect()，可以增加使用它的所有程序的可移植性。如果将这些连接代码直接嵌入到每个客户机中，则必须独立地修改它们中的每一个。

要想修正 do_connect()，使它可以处理 mysql_real_connect() 的旧格式，那么就可以使用包括当前 MySQL 版本的 MySQL_VERSION_ID 宏。更改了的 do_connect() 测试 MySQL_VERSION_ID 值，并使用 mysql_real_connect() 的正确格式：

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
MYSQL   *conn;  /* pointer to connection handler */

    conn = mysql_init (NULL);   /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        fprintf (stderr, "mysql_init() failed\n");
        return (NULL);
    }
#if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200  /* 3.22 and up */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    db_name, port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                         mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
#else                           /* pre-3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                    port_num, socket_name, flags) == NULL)
    {
        fprintf (stderr, "mysql_real_connect() failed:\nError %u (%s)\n",
                         mysql_errno (conn), mysql_error (conn));
        return (NULL);
    }
    if (db_name != NULL)        /* simulate effect of db_name parameter */
    {
        if (mysql_select_db (conn, db_name) != 0)
        {
            fprintf (stderr, "mysql_select_db() failed:\nError %u (%s)\n",
                         mysql_errno (conn), mysql_error (conn));
```

```
                    mysql_errno (conn), mysql_error (conn));
            mysql_close (conn);
            return (NULL);
        }
    }
#endif
    return (conn);          /* connection is established */
}
```

除了下述两点以外，do_connect() 的这个修改过的版本和前一个版本在外观上是完全一样的：

- 它不将 db_name 参数传递给 mysql_real_connect() 较早的格式，因为那个版本没有这样的参数。
- 如果数据库名称是非NULL的，则do_connect()调用mysql_select_db()使指定的数据库为当前数据库（这类似于没有 db_name 参数的效果）。如果没有选择这个数据库，则do_connect()打印一个错误消息，关闭连接，并返回NULL来表示失败。

■ **样例2** 该样例是在对第一个样例的 do_connect() 做更改的基础上建立的。那些更改导致对错误函数 mysql_errno() 和 mysql_error() 的三组调用。每次都将报告问题的这些代码书写出来是非常讨厌的。除此之外，错误所打印出的代码看起来不舒服，读起来也困难。而读下面这样的代码就比较容易：

```
print_error (conn, " mysql_real_connect() failed " );
```

所以，让我们在 print_error() 函数中封装错误打印。即使 conn 为 NULL，也可以编写它来做一些明智的事情。也就是说，如果 mysql_init() 调用失败，可以使用 print_error()。而且没有混合调用（一些为 fprintf()，一些为 print_error()）。

我听到一些反对意见：“为了想报告一个错误而又不必每次都调用两个错误函数，所以使代码故意编写得难以阅读，以说明封装样例更好。其实不用真的写出所有的错误打印代码：只将它编写一次，然后当再次需要时就使用拷贝和粘贴即可。”这种观点是正确的，但我持反对意见，理由如下：

- 即使使用拷贝和粘贴，用较短的代码段进行起来也更容易。
- 每当报告错误时，无论是否愿意每次调用两种函数，将所有的错误报告代码书写得很长，会产生不一致性。将错误报告的代码放在容易调用的包装函数中，就可以减少这种想法并提高编码的一致性。
- 如果决定修改错误消息的格式，则只需要在一个地方而不是整个程序中做更改，这样就要容易许多。或者，如果决定将错误消息编写到日志文件中而不是（或除此以外还）编写到 stderr 中，则只须更改 print_error()，这就更容易。这种方法可能犯更少的错误，而且再一次减少了工作量和不一致的可能性。
- 当测试程序时，如果使用调试程序，将断点放在错误报告的函数中，则当它侦测出一个错误条件时，调试程序是使程序中断的一种便利方法。

以下是错误报告函数 print_error()的使用举例：

```
void
print_error (MYSQL *conn, char *message)
{
    fprintf (stderr, "%s\n", message);
    if (conn != NULL)
    {
```

```
        fprintf (stderr, "Error %u (%s)\n",
                mysql_errno (conn), mysql_error (conn));
    }
}
```

Print_error()在common中，所以为它增加一个原型到common.中：

```
/* common.h */
void print_error (MYSQL *conn, char *message);
```

现在，可以修改 do_connect() 来使用 print_error()：

```
MYSQL *
do_connect (char *host_name, char *user_name, char *password, char *db_name,
            unsigned int port_num, char *socket_name, unsigned int flags)
{
MYSQL   *conn;  /* pointer to connection handler */

    conn = mysql_init (NULL);  /* allocate, initialize connection handler */
    if (conn == NULL)
    {
        print_error (NULL, "mysql_init() failed (probably out of memory)");
        return (NULL);
    }
#if defined(MYSQL_VERSION_ID) && MYSQL_VERSION_ID >= 32200  /* 3.22 and up
*/
    if (mysql_real_connect (conn, host_name, user_name, password,
                db_name, port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        return (NULL);
    }
#else                           /* pre-3.22 */
    if (mysql_real_connect (conn, host_name, user_name, password,
                port_num, socket_name, flags) == NULL)
    {
        print_error (conn, "mysql_real_connect() failed");
        return (NULL);
    }
    if (db_name != NULL)        /* simulate effect of db_name parameter */
    {
        if (mysql_select_db (conn, db_name) < 0)
        {
            print_error (conn, "mysql_select_db() failed");
            mysql_close (conn);
            return (NULL);
        }
    }
#endif
    return (conn);          /* connection is established */
}
```

主源文件 client3.c 与 client2.c 一样，但是所有嵌入的连接和断开代码都利用调用包装函数来删除和替换了。如下所示：

```
/* client3.c */

#include <stdio.h>
```

```
#include <mysql.h>
#include "common.h"

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_port_num    0    /* use default port */
#define def_socket_name NULL /* use default socket name */
#define def_db_name     NULL /* database to use (default = none) */

MYSQL   *conn;  /* pointer to connection handler */

int
main (int argc, char *argv[])
{
    conn = do_connect (def_host_name, def_user_name, def_password, def_db_name,
                                    def_port_num, def_socket_name, 0);
    if (conn == NULL)
        exit (1);

    /* do the real work here */

    do_disconnect (conn);
    exit (0);
}
```

6.5 客户机程序4——在运行时获取连接参数

现在我们有了容易修改的防止出现错误的连接代码，我们要了解一些如何做某些比使用NULL 连接参数更灵巧的事情，如在运行时允许用户指定一些值。

客户机程序3由于固定连接参数方面的缺陷，要想更改那些值中的任何一个，都必须编辑源文件并重新编译。这十分不方便，特别是想使程序用于其他人时。

在运行时指定连接参数的一个通用的方法是使用命令行选项。 MySQL 分发包中的程序接受两种形式的连接参数，如表6-1所示。

表6-1 标准的 MySQL 命令行选项

参 数	短 格 式	长 格 式
主机名称	-h host_name	--host=host_name
用户名称	-u user_name	--user=user_name
口令	-p 或 -pyour_password	--password 或--password=your_password
端口号	-P port_num	--port=port_num
套接字名称	-S socket_name	--socket=socket_name

与标准的 MySQL 客户机程序一致，客户机程序将接受同样的格式。这很容易，那是因为客户机库包括了实现选项分析的函数。

除此之外，客户机程序具有从选项文件中抽取信息的能力。这允许将连接参数放在 -/. my.cnf（也就是主目录中的 .my.cnf 文件）中，以便不用在命令行中指定它们。客户机库使检查 MySQL 选项文件和从它们中抽取任何相关的值变得非常容易。只在程序中增加几行代码，就可以使选项文件识别它，并且通过编写自己的代码而不必重新改造这个框架来进行操作。附录 E “ MySQL 程序参考 ”中说明了选项文件的语法。

6.5.1 访问选项文件内容

使用 load_default() 函数为连接参数值读取选项文件，load_default() 寻找选项文件、分析任何感兴趣的可选组的内容，以及重新编写程序的参数向量（ argv[] 数组），以便把来自于那些组的信息以命令行选项的形式放置在 argv[] 的开头。这就是说，在命令行指定出现的选项。因此，当分析命令选项时，就得到了作为常规选项分析循环部分的连接参数。选项加到 argv[] 的开头而不是加到末尾，所以，如果连接参数真的在命令行指定，它们要比 load_defaults() 增加的任何选项晚一些出现（因而忽略）。

下面的小程序 show_argv 显示了如何使用 load_defaults()，并举例说明了对参数向量如何做出这样的修改：

```
/* show_argv.c */

#include <stdio.h>
#include <mysql.h>

char *groups[] = { "client", NULL };

int
  main (int argc, char *argv[])
  {
  int i;

      my_init ();

      printf ("Original argument vector:\n");
      for (i = 0; i < argc; i++)
          printf ("arg %d: %s\n", i, argv[i]);

      load_defaults ("my", groups, &argc, &argv);

      printf ("Modified argument vector:\n");
      for (i = 0; i < argc; i++)
          printf ("arg %d: %s\n", i, argv[i]);

      exit (0);
  }
```

该处理选项文件的代码包括：

- groups[] 是一个字符串数组，表示所感兴趣的选项文件组。对于客户机程序，始终至少指定“ client ”（[client] 组）。数组的最后一个元素必须是 NULL。
- my_init() 是 load_defaults() 所需的执行一些设置操作的初始化例程。
- load_defaults() 有四个参数：选项文件的前缀（这里应该始终是“ my ”），列出感兴趣的可选组的数组、程序参数的数目和向量的地址。不传数目和向量的值，而是传地址，因为load_defaults() 需要改变它们的值。特别注意的是，虽然 argv 是一个指针，但还是要传 &argv ，它是指针的地址。

show_argv打印参数两次，第一次是在命令行指定它们的时候，第二次是在 load_defaults() 修改它们的时候。为了查看 load_defaults() 的运行效果，应确信在主目录中有一个具有 [client] 组指定设置的 .my.cnf 文件。假设 .my.cnf 文件如下：

```
[client]
user=paul
password=secret
host=some_host
```

如果是这种情况，则执行 show_argv 产生的输出结果如下：

```
% show_argv a b
Original argument vector:
arg 0: show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: show_argv
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

有可能会从不在命令行或~/.my.cnf 文件中的 show_argv 所产生的输出结果中看到一些选项。如果是这样，它们或许是在系统范围的选项文件中指定的。在主目录中读取 .my.cnf 之前，load_defaults() 实际上是在 MySQL 数据目录中寻找 /etc/my.cnf 和 my.cnf 文件 （在 Windows 中，load_defaults() 在Windows 系统目录中寻找文件 C：my.cnf、C:\mysql\data\my.cnf 和 my.ini ）。

使用 load_defaults() 的客户机程序几乎始终是在选项组列表中指定“ client ”（以便从选项文件中获取任何通用的客户机设置），但是也可以为请求自己的程序请求特定值。可将下列代码：

```
char *groups[] = { "client", NULL };
```

修改为：

```
char *groups[] = { "show_argv", "client", NULL };
```

然后将 [show_argv] 组加到~/.my.cnf 文件中：

```
[client]
user=paul
password=secret
host=some_host

[show_argv]
host=other_host
```

有了这些改变，再次调用 show_argv 就得到了一个不同的结果，如下所示：

```
% show_argv a b
Original argument vector:
arg 0: show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: show_argv
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: --host=other_host
arg 5: a
arg 6: b
```

参数数组中选项值出现的顺序取决于它们在选项文件中列出的顺序，而不是选项组在 group[] 数组中列出的顺序。这意味着将可能在选项文件的 [client] 组之后指定程序专有的组。即如果在两个组中都指定了一个选项，程序专有的值将有更高的优先权。在这个例子中可以看到： 在组 [client] 和 [show_argv] 中都指定了 host 选项，但是因为组 [show_argv] 在选项文件的最后出现，所以 host 值将在参数向量中出现并取得优先权。

load_defaults() 不是从环境设置中提取值，如果想使用环境变量的值，例如 MYSQL_TCP_PORT 或者 MYSQL_UNIX_PORT，就必须使用 getenv() 来自己管理。我不想把这个管理能力增加到客户机中，但这里有一个例子，介绍了如何检查几个标准的与 MySQL 有关的环境变量值：

```
extern char *getenv();
char *p;
int port_num;
char *socket_name;

if ((p = getenv ("MYSQL_TCP_PORT")) != NULL)
    port_num = atoi (p);
if ((p = getenv ("MYSQL_UNIX_PORT")) != NULL)
    socket_name = p;
```

在标准 MySQL 客户机中，环境变量值的优先权比在选项文件或命令行指定值的优先权要低。如果检查环境变量的值，并要与约定保持一致，那么就要在调用 load_default() 或者处理命令行选项之前（不是之后）检查环境。

6.5.2 分析命令行参数

现在我们可以把所有的连接参数都放入参数向量，但需要一个分析该向量的方法。getopt_long() 函数就是为此目的设计的。

getopt_long() 设在 MySQL 客户机库的内部，因此，无论什么时候与库连接都可以访问它。源文件中要包含 getopt.h 头文件，可以把这个头文件从 MySQL 源分发包的 include 目录拷贝到正在开发的客户机程序所在的目录中。

load_defaults() 与安全

因为有些程序（如 ps）可以显示任何过程的参数列表， load_defaults() 将口令的文本放在参数列表中，所以您可能对它处理窥探的含意表示惊异。这没有问题，因为 ps 显示原始的 argv[] 内容，由 load_defaults() 创建的任何口令参数都指向为它自己分配的区域，这个区域并不是原始区域的一部分，所以 ps 看不见它。

另一方面，除非故意清除，否则在命令行指定的口令会在 ps 中出现。6.5.2节“分析命令行参数”介绍了如何去做。

下面的程序 show_param 使用 load_defaults() 读取选项文件，然后调用 getopt_long() 来分析参数向量。show_param 举例说明了通过执行以下操作参数处理的每个阶段发生了什么：

1) 建立主机名称、用户名称和口令的缺省值。
2) 打印原始连接参数和参数向量值。
3) 调用 load_defaults() 重新编写参数向量，反映选项文件内容，然后打印结果向量。
4) 调用 getopt_long() 处理参数向量，然后打印结果参数值和参数向量中的剩余部分。

show_param 允许使用各种指定的连接参数的方法进行试验（无论是在选项文件中还是在命令行中），并通过显示使用什么值进行连接来查看结果。当实际上我们把参数处理代码与连接函数 do_connect() 连到一起时，show_param 对于预知下一个客户机程序将要发生什么是很有用的。

以下是 show_param.c 的代码：

```
/* show_param.c */

#include <stdio.h>
#include <stdlib.h> /* needed for atoi() */
#include "getopt.h"

char *groups[] = { "client", NULL };

struct option long_options[] =
{
    {"host",     required_argument, NULL, 'h'},
    {"user",     required_argument, NULL, 'u'},
    {"password", optional_argument, NULL, 'p'},
    {"port",     required_argument, NULL, 'P'},
    {"socket",   required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

int
main (int argc, char *argv[])
{
char *host_name = NULL;
char *user_name = NULL;
char *password = NULL;
unsigned int port_num = 0;
char *socket_name = NULL;
int i;
int c, option_index;
my_init ();

printf ("Original connection parameters:\n");
printf ("host name: %s\n", host_name ? host_name : "(null)");
printf ("user name: %s\n", user_name ? user_name : "(null)");
printf ("password: %s\n", password ? password : "(null)");
printf ("port number: %u\n", port_num);
printf ("socket name: %s\n", socket_name ? socket_name : "(null)");

printf ("Original argument vector:\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

load_defaults ("my", groups, &argc, &argv);

printf ("Modified argument vector after load_defaults():\n");
for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                &option_index)) != EOF)
{
    switch (c)
    {
```

```
        case 'h':
            host_name = optarg;
            break;
        case 'u':
            user_name = optarg;
            break;
        case 'p':
            password = optarg;
            break;
        case 'P':
            port_num = (unsigned int) atoi (optarg);
            break;
        case 'S':
            socket_name = optarg;
            break;
        }
}

argc -= optind; /* advance past the arguments that were processed */
argv += optind; /* by getopt_long() */

printf ("Connection parameters after getopt_long():\n");
printf ("host name: %s\n", host_name ? host_name : "(null)");
printf ("user name: %s\n", user_name ? user_name : "(null)");
printf ("password: %s\n", password ? password : "(null)");
printf ("port number: %u\n", port_num);
printf ("socket name: %s\n", socket_name ? socket_name : "(null)");
        printf ("Argument vector after getopt_long():\n");
        for (i = 0; i < argc; i++)
            printf ("arg %d: %s\n", i, argv[i]);

        exit (0);
    }
```

为了处理参数向量， show_argv() 使用 getopt_long() ， 它在循环中调用：

To process the argument vector, show_argv uses getopt_long(), which you typically call in a loop:

```
    while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                    &option_index)) != EOF)
    {
```

getopt_long() 的前两个参数是程序的计数参数和向量参数，第三个参数列出了要识别的选项字符。这些是程序选项的短名称形式。选项字符后可以有冒号、双冒号或者无冒号，表示选项值必须跟在选项后面、可以跟在选项后面或者不能跟在选项后面。第四个参数 long_options 是一个指向可选结构数组的指针，每个可选结构为程序需要支持的选项指定信息。它的目标与第三个参数的可选字符串相类似。每个 long_options[] 结构有四个元素，其描述如下：

- **选项的长名称。**
- **选项值**。这个值可以是 required_argument、optional_argument 或者 no_argument，表明选项值是必须跟在选项后面、可以跟在选项后面，还是不能跟在选项后面（它们与第三个参数选项字符串中的冒号、双冒号或无冒号的作用相同）。
- **标记参数。**可用它存储变量指针。如果找到这个选项， getopt_long() 则把第四个参数指定的值存储到变量中去。如果标记是 NULL，getopt_long() 就把 optarg 变量指向下一个选项的任何值，并返回选项的短名称。 long_options[] 数组为所有的选项指定了

NULL。那就是说，如果遇到 getopt_long()，就返回每个参数，以便我们可以在 switch 语句中来处理它。

- **选项的短（单个字符）名称。**在 long_options[] 数组中指定的短名称必须与作为第三个参数传递给 getopt_long() 的选项字符串所使用的字母相匹配，否则程序将不能正确处理命令行参数。

long_options[] 数组必须由一个所有元素都设为 0 的结构所终止。

getopt_long() 的第五个参数是一个指向 int 变量的指针。getopt_long() 把与最后遇到的选项相符合的 long_options[] 结构索引存储到变量中（show_param 不用这个值做任何事情）。

请注意，口令选项（指定为 --password 或者 -p ）可以获得一个选项值，那就是说，如果使用长选项形式可指定为 --password 或者 --password = your_pass，如果使用短选项形式则指定为 -p 或者 -pyour_pass。可选字符串中“ p ”后面的双冒号和 long_options[] 数组中的 optional_argument 表示了口令值的可选特性。特别是， MySQL 客户机允许在命令行省略口令值，然后提示输入。这样避免了在命令行给出口令，它防止其他人通过偷窃看到口令。在编写下一个客户机程序（客户机程序 4）时，将把口令检查性能添加进去。

下面是 show_param 的调用示例和结果输出（假设 ~ /.my.cnf 一直与 show_argv 示例有相同的内容）：

```
% show_param -h yet_another_host x
Original connection parameters:
host name: (null)
user name: (null)
password: (null)
port number: 0
socket name: (null)
Original argument vector:
arg 0: show_param
arg 1: -h
arg 2: yet_another_host
arg 3: x
Modified argument vector after load_defaults():
arg 0: show_param
arg 1: --user=paul
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: x
Connection parameters after getopt_long():
host name: yet_another_host
user name: paul
password: secret
port number: 0
socket name: (null)
Argument vector after getopt_long():
arg 0: x
```

输出结果说明从命令行得到主机名（忽略选项文件中的这个值），从选项文件中得到用户名和口令。getopt_long() 正确分析了选项是在短选项形式（ -h host_name ）中指定还是在长选项形式（ --user = paul ， --password = secret ）中指定。

现在让我们去掉纯粹说明选项处理例程是如何工作的这一部分，把剩余部分作为根据选项文件或命令行提供的任何选项而连接到服务器的客户机的基础。源文件 client4.c 的代码如下：

```
/* client4.c */

#include <stdio.h>
#include <stdlib.h> /* for atoi() */
#include <mysql.h>
#include "common.h"
#include "getopt.h"

#define def_host_name   NULL /* host to connect to (default = localhost) */
#define def_user_name   NULL /* user name (default = your login name) */
#define def_password    NULL /* password (default = none) */
#define def_port_num    0    /* use default port */
#define def_socket_name NULL /* use default socket name */
#define def_db_name     NULL /* database to use (default = none) */

char *groups[] = { "client", NULL };

struct option long_options[] =
{
    {"host",     required_argument, NULL, 'h'},
    {"user",     required_argument, NULL, 'u'},
    {"password", optional_argument, NULL, 'p'},
    {"port",     required_argument, NULL, 'P'},
    {"socket",   required_argument, NULL, 'S'},
    { 0, 0, 0, 0 }
};

MYSQL   *conn;  /* pointer to connection handler */

int
main (int argc, char *argv[])
{
char *host_name = def_host_name;
char *user_name = def_user_name;
char *password = def_password;
unsigned int port_num = def_port_num;
char *socket_name = def_socket_name;
char *db_name = def_db_name;
char    passbuf[100];
int ask_password = 0;
int c, option_index=0;
int i;

    my_init ();
load_defaults ("my", groups, &argc, &argv);

while ((c = getopt_long (argc, argv, "h:p::u:P:S:", long_options,
                &option_index)) != EOF)
{
    switch (c)
    {
    case 'h':
        host_name = optarg;
        break;
    case 'u':
        user_name = optarg;
        break;
    case 'p':
        if (!optarg)        /* no value given */
            ask_password = 1;
```

```
            else                /* copy password, wipe out original */
            {
                    (void) strncpy (passbuf, optarg, sizeof(passbuf)-1);
                    passbuf[sizeof(passbuf)-1] = '\0';
                    password = passbuf;
                    while (*optarg)
                        *optarg++ = ' ';
            }
            break;
        case 'P':
            port_num = (unsigned int) atoi (optarg);
            break;
        case 'S':
            socket_name = optarg;
            break;
        }
    }

    argc -= optind; /* advance past the arguments that were processed */
    argv += optind; /* by getopt_long() */

    if (argc > 0)
    {
        db_name = argv[0];
        --argc; ++argv;
    }

    if (ask_password)
        password = get_tty_password (NULL);

    conn = do_connect (host_name, user_name, password, db_name,
                                    port_num, socket_name, 0);
    if (conn == NULL)
        exit (1);
        /* do the real work here */

        do_disconnect (conn);
        exit (0);
    }
```

与前面开发的客户机程序 1、客户机程序 2和客户机程序 3比较一下，客户机程序 4具有一些以前没有的内容：

- 允许在命令行指定数据库名称，它紧跟在由 getopt_long() 分析的选项的后面。这与 MySQL 分发包中标准客户机的行为是一致的。
- 对口令值做了备份之后，删除参数向量中的任何口令值。这使时间窗口最小化，在时间窗口中命令行所指定的口令对于 ps 或其他系统状态程序是可见的（窗口缩到最小，但并没有删除。命令行指定的口令仍然不太安全）。
- 如果给出没有值的口令选项，则客户机程序提示用户用 get_tty_password() 输入口令。在客户机库中，这是一个实用程序，它提示输入口令而不在显示器上回应（客户机库充满了这样吸引人的东西。因为找到了相关的例程和使用它们的方法，所以有助于从 MySQL 客户机程序的源文件中的读取）。您可能会问：“为什么不只调用 getpass() 呢？”回答是，并不是所有的系统都有这个函数，如 Windows。get_tty_password() 可以在系统间移植，因为它被配置为适应各种不同系统。

客户机程序4按照指定的选项来响应。假设没有使事件复杂化的选项文件。如果无参数调用客户机程序4，则连接到 localhost，并把 UNIX 注册名和无口令传递到服务器中。相反，如果像介绍的那样调用客户机程序 4，则提示输入口令（没有直接以 -p 开头的口令值），连接到 some_host，并将用户名 some_user 和键入的口令都传递到服务器：

```
% client4 -h some_host -u some_user -p some_db
```

客户机程序4也把数据库名 some_db 传递给 do_connect()，成为当前数据库。如果没有选项文件，则处理它的内容并用来改变参数连接。

早期，我们曾热衷于封装代码，创建包装函数，目的是断开与服务器的连接和从服务器的连接断开。询问是否把分析选项部分放置到包装函数中也是合理的。我想这是可能的，但并不想去做。选项分析代码与连接代码在程序间并不一致：程序经常支持除了标准选项之外的其他选项，不同的程序很可能支持其他选项的不同设置。这就使选项处理循环标准化的函数很难编写。而且，与连接的建立不同，在它的执行过程中程序可以希望进行多次（因而是好的封装候选者），而选项分析只在程序开始时执行一次。

迄今为止，我们所做的工作完成了每个 MySQL 客户机程序所必须做的事情：用适当的参数与服务器相连接。当然应该知道如何连接，现在知道怎么做了，并且处理的细节由客户机程序框架（client4.c ）来实现，因此就不必再去考虑了。这就是说可以集中精力干真正感兴趣的事情 —— 访问数据库的内容。应用程序中所有的真正功能将在 do_connect() 调用和 do_disconnect() 调用之间发生，但是我们现在所拥有的是用于建立可为许多不同客户机程序使用的基本框架。编写一个新程序，要做到以下几点：

1) 制作一个 client4.c 的备份。

2) 如果接受其他选项而不是 client4.c 支持的标准选项，那么修改处理选项循环。

3) 在连接和断开调用之间加上自己的应用程序代码。

这样就算完成了。

构造客户机程序框架的目的是，很容易地建立和断开连接，以便集中精力干真正想做的事情。

6.6 处理查询

我们已经知道了如何开始和结束与服务器的会话，现在应该看看如何控制会话。本节介绍了如何与服务器通信以处理查询。

执行的每个查询应包括以下几步：

1) **构造查询**。查询的构造取决于查询的内容 —— 特别要看是否含有二进制数据。

2) **通过将查询发送到服务器执行来发布查询。**

3) **处理查询结果。**这取决于发布查询的类型。例如， SELECT 语句返回数据行等待处理，INSERT 语句就不这样。

构造查询的一个要素就是使用哪个函数将查询发送到服务器。较通用的发布查询例程是 mysql_real_query()。该例程给查询提供了一个计数串（字符串加上长度）。必须了解查询串的长度，并将它们连同串本身一起传递给 mysql_real_query() 。因为查询是一个计数的字符串，所以它的内容可能是任何东西，其中包括二进制数据或者空字节。查询不能是空终结串。

另一个发布查询的函数， mysql_query()，在查询字符串允许的内容上有更多的限制，但

更容易使用一些。传递到 mysql_query() 的查询应该是空终结串，这说明查询内部不能含有空字节（查询里含有空字节会导致错误地中断，这比实际的查询内容要短）。一般说来，如果查询包含任意的二进制数据，就可能包含空字节，因此不要使用 mysql_query()。另一方面，当处理空终结串时，使用熟悉的标准 C 库字符串函数构造查询是很耗费资源的，例如 strcpy() 和 sprintf()。

构造查询的另一个要素就是是否要执行溢出字符的操作。如果在构造查询时使用含有二进制数据或者其他复杂字符的值时，如引号、反斜线等，就需要使用这个操作。这些将在6.8.2节“对查询中有疑问的数据进行编码”中讨论。

下面是处理查询的简单轮廓：

```
if (mysql_query (conn, query) != 0)
{
    /* failure; report error */
}
else
{
    /* success; find out what effect the query had */
}
```

mysql_query() 和 mysql_real_query() 的查询成功都会返回零值，查询失败返回非零值。查询成功指服务器认为该查询有效并接受，而且能够执行，并不是指有关该查询结果。例如，它不是指 SELECT 查询所选择的行，或 DELETE 语句所删除的行。检查查询的实际结果要包括其他的处理。

查询失败可能有多种原因，有一些常见的原因如下：

- 含有语法错误。
- 语义上是非法的——例如涉及对表中不存在的列的查询。
- 没有足够的权利访问查询所引用的数据。

查询可以分成两大类：不返回结果的查询和返回结果的查询。 INSERT、DELETE和UPDATE等语句属于“不返回结果”类的查询，即使对修改数据库的查询，它们也不返回任何行。可返回的唯一信息就是有关受作用的行数。

SELECT 语句和 SHOW 语句属于“返回结果”类的查询；发布这些语句的目的就是要返回某些信息。返回数据的查询所生成的行集合称为结果集，在 MySQL 中表示为 MYSQL_RES 数据类型，这是一个包含行的数据值及有关这些值的元数据（如列名和数据值的长度）的结构。空的结果集（就是包含零行的结果）要与“没有结果”区分开。

6.6.1 处理不返回结果集的查询

处理不返回结果集的查询，用 mysql_query() 或 mysql_real_query() 发布查询。如果查询成功，可以通过调用 mysql_affected_rows() 找出有多少行需要插入、删除或修改。

下面的样例说明如何处理不返回结果集的查询：

```
if (mysql_query (conn, "INSERT INTO my_tbl SET name = 'My Name'") != 0)
{
    print_error ("INSERT statement failed");
}
else
{
```

```
        printf ("INSERT statement succeeded: %lu rows affected\n",
                (unsigned long) mysql_affected_rows (conn));
}
```

请注意在打印时 mysql_affected_rows() 的结果是如何转换为 unsigned long 类型的，这个函数返回一个 my_ulonglong 类型的值，但在一些系统上无法直接打印这个类型的值（例如，笔者观察到它可在 FreeBSD 下工作，但不能在 Solaris 下工作）。把值转换为 unsigned long 类型并使用‘ %lu ’打印格式可以解决这个问题。同样也要考虑返回 my_ulonglong 值的其他函数，如 mysql_num_rows() 和 mysql_insert_id()。如果想使客户机程序能跨系统地移植，就要谨记这一点。

mysql_rows_affected() 返回查询所作用的行数，但是“ 受作用的行 ”的含义取决于查询的类型。对于 INSERT、 DELETE 和 UPDATE，是指插入、删除或者更新的行数，也就是 MySQL 实际修改的行数。如果行的内容与所要更新的内容相同，则 MySQL 就不再更新行。这就是说虽然可能选择行来更新（通过 UPDATE 语句的 WHERE 子句），但实际上该行可能并未改变。

对于 UPDATE，“ 受作用的行 ”的意义实际上是个争论点，因为人们想把它当成“ 被匹配的行 ”——即选择要更新的行数，即使更新操作实际上并未改变其中的值也是如此。如果应用程序需要这个信息，则当与服务器连接时可以用它来请求以实现这个功能。将 CLIENT_FOUND_ROWS 的 flags 值传递给 mysql_real_connect()。也可以将 CLIENT_FOUND_ROWS 作为 flags 参数传递给 do_connect()；它将把值传递给 mysql_real_connect()。

6.6.2 处理返回结果集的查询

通过调用 mysql_query() 和 mysql_real_query() 发布查询之后，返回数据的查询以结果集形式进行。在 MySQL 中实现它非常重要， SELECT 不是返回行的唯一语句， SHOW、DESCRIBE 和 EXPLAIN 都需要返回行。对所有这些语句，都必须在发布查询后执行另外的处理行操作。

处理结果集包括下面几个步骤：

- 通过调用 mysql_store_result() 或 mysql_use_result() 产生结果集。这些函数如果成功则返回 MYSQL_RES 指针，失败则返回 NULL。稍后我们将查看 mysql_store_result() 与 mysql_use_result() 的不同，以及选择其中一个而不选另一个时的情况。我们的样例使用 mysql_store_result()，它能立即从服务器返回行，并将它们存储到客户机中。
- 对结果集的每一行调用 mysql_fetch_rows()。这个函数返回 MYSQL_ROW 值，它是一个指向字符串数组的指针，字符串数组表示行中每列的值。要根据应用程序对行进行操作。可以只打印出列值，执行有关的统计计算，或者做些其他操作。当结果集中不再有行时， mysql_fetch_rows() 返回 NULL。
- 处理结果集时，调用 mysql_free_result() 释放所使用的内存。如果忽略了这一点，则应用程序就会泄露出内存（对于长期运行的应用程序，适当地解决结果集是极其重要的；否则，会注意到系统将由一些过程所取代，这些过程消耗着经常增长的系统资源量）。

下面的样例轮廓介绍了如何处理返回结果集的查询：

```
MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM mysql") != 0)
    print_error (conn, "mysql_query() failed");
else
{
    res_set = mysql_store_result (conn);    /* generate result set */
    if (res_set == NULL)
            print_error (conn, "mysql_store_result() failed");
    else
    {
        /* process result set, then deallocate it */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

我们通过调用函数 process_result_set() 来处理每一行，这里有个窍门，因为我们并没有定义这个函数，所以需要这样做。通常，结果的处理集函数是基于下面的循环：

```
MYSQL_ROW row;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* do something with row contents */
}
```

从 mysql_fetch_row() 返回的 MYSQL_ROW 值是一个指向数值数组的指针，因此，访问每个值就是访问 row[i]，这里 i 的范围是从0到该行的列数减1。

这里有几个关于 MYSQL_ROW 数据类型的要点需要注意：

- MYSQL_ROW 是一个指针类型，因此，必须声明类型变量为 MYSQL_ROW row，而不是 MYSQL_ROW *row。
- MYSQL_ROW 数组中的字符串是空终结的。但是，列可能含有二进制数据，这样，数据中就可能含有空字节，因此，不应该把值看成是空终结的。由列的长度可知列值有多长。
- 所有数据类型的值都是作为字符串返回的，即使是数字型的也是如此。如果需要该值为数字型，就必须自己对该字符串进行转换。
- 在 MYSQL_ROW 数组中，NULL 指针代表 NULL，除非声明列为 NOT NULL，否则应该经常检查列值是否为 NULL 指针。

应用程序可以利用每行的内容做任何想做的事，为了举例说明这一点，我们只打印由制表符隔开列值的行，为此还需要另外一个函数， mysql_num_fields() ,它来自于客户机库；这个函数告知我们该行包括多少个值（列）。

下面就是 process_result_set() 的代码：

```
void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_ROW       row;
unsigned int    i;

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
```

```
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        if (i > 0)
            fputc ('\t', stdout);
        printf ("%s", row[i] != NULL ? row[i] : "NULL");


        }
        fputc ('\n', stdout);
    }
    if (mysql_errno (conn) != 0)
        print_error (conn, "mysql_fetch_row() failed");
    else
        printf ("%lu rows returned\n", (unsigned long) mysql_num_rows (res_set));
}
```

process_result_set() 以制表符分隔的形式打印每一行（将 NULL值显示为单词“ NULL ”），它跟在被检索的行计数的后面，该计数通过调用 mysql_num_rows() 来计算。像 mysql_affected_rows() 一样，mysql_num_rows() 返回 my_ulonglong 值，因此，将值转换为 unsigned long 型，并用‘ %lu ’格式打印。

提取行的循环紧接在一个错误检验的后面，如果要用 mysql_store_result() 创建结果集，mysql_fetch_row() 返回的NULL值通常意味着“ 不再有行 ”。然而，如果用 mysql_use_result() 创建结果集，则 mysql_fetch_row() 返回的 NULL 值通常意味着“ 不再有行 ”或者发生了错误。无论怎样创建结果集，这个测试只允许 process_result_set() 检测错误。

process_result_set() 的这个版本是打印列值要求条件最低的方法，每种方法都有一定的缺点，例如假设执行下面的查询：

```
SELECT last_name, first_name, city, state FROM president
```

会得到下面的输出：

```
Adams    John     Braintree    MA
Adams    John Quincy Braintree    MA
Arthur   Chester A.  Fairfield    VT
Buchanan     James    Mercersburg PA
Bush     George W.    Milton   MA
Carter   James E. Jr Plains   GA
Cleveland    Grover   Caldwell     NJ
```

我们可以通过提供一些信息如列标签，及通过使这些值垂直排列，而使输出结果漂亮一点。为此，我们需要标签和每列所需的最宽的值。这个信息是有效的，但不是列数据值的一部分，而是结果集的元数据的一部分（有关数据的数据）。简单归纳了一下查询处理程序后，我们将在6.6.6节“ 使用结果集元数据 ”中给出较漂亮的显示格式。

打印二进制数据

对包含可能含有空字节的二进制数据的列值，使用‘ %s ’printf() 格式标识符不能将它正确地打印；printf() 希望一个空终结串，并且直到第一个空字节才打印列值。对于二进制数据，最好用列的长度，以便打印完整的值，如可以用 fwrite() 或 putc()。

6.6.3 通用目标查询处理程序

前面介绍的处理查询样例应用了语句是否应该返回一些数据的知识来编写的。这是可能

的，因为查询固定在代码内部：使用 INSERT 语句时，它不返回结果，使用 SHOW TABLES 语句时，才返回结果。

然而，不可能始终知道查询用的是哪一种语句，例如，如果执行一个从键盘键入或来源于文件的查询，则它可能是任何的语句。不可能提前知道它是否会返回行。当然不想对查询做语法分析来决定它是哪类语句，总之，并不像看上去那样简单。只看第一个单词是不够的，因为查询也可能以注释语句开始，例如：

```
/* comment */ SELECT ...
```

幸运的是不必过早地知道查询类型就能够正确地处理它。用 MySQL C API 可编写一个能很好地处理任何类型语句的通用目标查询处理程序，无论它是否会返回结果。

在编写查询处理程序的代码之前，让我们简述一下它是如何工作的：

- 发布查询，如果失败，则结束。
- 如果查询成功，调用 mysql_store_result() 从服务器检索行，并创建结果集。
- 如果 mysql_store_result() 失败，则查询不返回结果集，或者在检索这个结果集时发生错误。可以通过把连接处理程序传递到 mysql_field_count() 中，并检测其值来区别这两种情况，如下：
  - 如果 mysql_field_count() 非零，说明有错误，因为查询应该返回结果集，但却没有。这种情况发生有多种原因。例如：结果集可能太大，内存分配失败，或者在提取行时客户机和服务器之间发生网络中断。

    这种过程稍微有点复杂之处就在于，MySQL 3.22.24 之前的早期版本中不存在mysql_field_count()，它们使用的是 mysql_num_fields()。为编写 MySQL 任何版本都能运行的程序，在调用 mysql_field_count() 的文件中都包含下面的代码块：

    ```
    #if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID<32224
    #define mysql_field_count mysql_num_fields
    #endif
    ```

    这就将对 mysql_field_count() 的一些调用看作是比 MySQL 3.22.24 更早版本中的 mysql_num_fields() 的调用。
  - 如果 mysql_field_count() 返回0，就意味着查询不返回结果（这说明查询是类似于 INSERT、DELETE、或 UPDATE 的语句）。
- 如果 mysql_store_result() 成功，查询返回一个结果集，通过调用 mysql_fetch_row() 来处理行，直到它返回 NULL 为止。

下面的列表说明了处理任意查询的函数，给出了连接处理程序和空终结查询字符串：

```
#if !defined(MYSQL_VERSION_ID) || MYSQL_VERSION_ID<32224
#define mysql_field_count mysql_num_fields
#endif

void
process_query (MYSQL *conn, char *query)
{
MYSQL_RES *res_set;
unsigned int field_count;

    if (mysql_query (conn, query) != 0) /* the query failed */
    {
        print_error (conn, "process_query() failed");
```

```
    return;
}

/* the query succeeded; determine whether or not it returns data */

res_set = mysql_store_result (conn);
if (res_set == NULL)    /* no result set was returned */
{
    /*
     * does the lack of a result set mean that an error
     * occurred or that no result set was returned?
     */
    if (mysql_field_count (conn) > 0)
    {
        /*
         * a result set was expected, but mysql_store_result()
         * did not return one; this means an error occurred
         */
        print_error (conn, "Problem processing result set");
    }
    else
    {
        /*
         * no result set was returned; query returned no data
         * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
         * so just report number of rows affected by query
         */
        printf ("%lu rows affected\n",
                (unsigned long) mysql_affected_rows (conn));
        }
    }
    else    /* a result set was returned */
    {
        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

6.6.4 可选择的查询处理方法

process_query() 的这个版本有三个特性：

- 用 mysql_query() 发布查询。
- 用 mysql_store_query() 检索结果集。
- 没有得到结果集时，用 mysql_field_count() 把错误事件和不需要的结果集区别开来。

针对查询处理的这些特点，有如下三种方法：

- 可以用计数查询字符串和 mysql_real_query()，而不使用空终结查询字符串和 mysql_query()。
- 可以通过调用 mysql_use_result() 而不是调用 mysql_store_result() 来创建结果集。
- 可以调用 mysql_error() 而不是调用 mysql_field_count() 来确定结果集是检索失败还是仅仅没有设置检索。

可用以上部分或全部方法代替 process_query()。以下是一个 process_real_query() 函数，它与 process_query() 类似，但使用了所有三种方法：

```
void
process_real_query (MYSQL *conn, char *query, unsigned int len)
{
MYSQL_RES *res_set;
unsigned int field_count;

    if (mysql_real_query (conn, query, len) != 0)   /* the query failed */
    {
        print_error (conn, "process_real_query () failed");
        return;
    }

    /* the query succeeded; determine whether or not it returns data */

    res_set = mysql_use_result (conn);
    if (res_set == NULL)    /* no result set was returned */
    {
       /*
        * does the lack of a result set mean that an error
        * occurred or that no result set was returned?
        */
       if (mysql_errno (conn) != 0) /* an error occurred */
           print_error (conn, "Problem processing result set");
       else
       {
           /*
            * no result set was returned; query returned no data
            * (it was not a SELECT, SHOW, DESCRIBE, or EXPLAIN),
            * so just report number of rows affected by query
            */
           printf ("%lu rows affected\n",
                       (unsigned long) mysql_affected_rows (conn));
       }
    }
    else    /* a result set was returned */
    {
        /* process rows, then free the result set */
        process_result_set (conn, res_set);
        mysql_free_result (res_set);
    }
}
```

6.6.5 mysql_store_result() 与 mysql_use_result() 的比较

函数 mysql_store_result() 与 mysql_use_result() 类似，它们都有连接处理程序参数，并返回结果集。但实际上两者间的区别还是很大的。两个函数之间首要的区别在于从服务器上检索结果集的行。当调用时，mysql_store_result() 立即检索所有的行，而 mysql_use_result() 启动查询，但实际上并未获取任何行，mysql_store_result() 假设随后会调用 mysql_fetch_row() 检索记录。这些行检索的不同方法引起两者在其他方面的不同。本节加以比较，以便了解如何选择最适合应用程序的方法。

当 mysql_store_result() 从服务器上检索结果集时，就提取了行，并为之分配内存，存储到客户机中，随后调用 mysql_fetch_row() 就再也不会返回错误，因为它仅仅是把行脱离了已经保留结果集的数据结构。mysql_fetch_row() 返回 NULL 始终表示已经到达结果集的末端。

相反，mysql_use_result() 本身不检索任何行，而只是启动一个逐行的检索，就是说必须

对每行调用 mysql_fetch_row() 来自己完成。既然如此，虽然正常情况下， mysql_fetch_row() 返回NULL 仍然表示此时已到达结果集的末端，但也可能表示在与服务器通信时发生错误。可通过调用 mysql_errno() 和 mysql_error() 将两者区分开来。

与 mysql_use_result() 相比，mysql_store_result() 有着较高的内存和处理需求，因为是在客户机上维护整个结果集，所以内存分配和创建数据结构的耗费是非常巨大的，要冒着溢出内存的危险来检索大型结果集，如果想一次检索多个行，可用 mysql_use_result()。

mysql_use_result() 有着较低的内存需求，因为只需给每次处理的单行分配足够的空间。这样速度就较快，因为不必为结果集建立复杂的数据结构。另一方面， mysql_use_result() 把较大的负载加到了服务器上，它必须保留结果集中的行，直到客户机看起来适合检索所有的行。这就使某些类型的客户机程序不适用 mysql_use_result()：

- 在用户的请求下提前逐行进行的交互式客户机程序（不必仅仅因为用户需要喝杯咖啡而让服务器等待发送下一行）。
- 在行检索之间做了许多处理的客户机程序。

在所有这些情况下，客户机程序都不能很快检索结果集的所有行，它限制了服务器，并对其他客户机程序产生负面的影响，因为检索数据的表在查询过程中是读锁定的。要更新表的客户机或要插入行的任何客户机程序都被阻塞。

偏移由 mysql_store_result() 引起的额外内存需求对一次访问整个结果集带来相当的好处。结果集中的所有行都是有效的，因此，可以任意访问： mysql_data_seek()、mysql_rowseek() 和 mysql_row_tell() 函数允许以任意次序访问行。而 mysql_use_result() 只能以 mysql_fetch_row() 检索的顺序访问行。如果想要以任意次序而不是从服务器返回的次序来处理行，就必须使用 mysql_store_result()。例如，如果允许用户来回地浏览查询所选的行，最好使用 mysql_store_result()。

使用 mysql_store_result() 可以获得在使用 mysql_use_result() 时是无效的某些类型的列信息。通过调用 mysql_num_rows() 来获得结果集的行数，每列中的这些值的最大宽度值存储在 MYSQL_FIELD 列信息结构的 max_width 成员中。使用 mysql_use_result()，直到提取完所有的行，mysql_num_rows() 才会返回正确值，而且 max_width 无效，因为只有在每行的数据都显示后才能计算。

由于 mysql_use_result() 比 mysql_store_result() 执行更少的操作，所以 mysql_use_result() 就强加了一个 mysql_store_result() 没有的需求：即客户机对结果集中的每一行都必须调用 mysql_fetch_row()，否则，结果集中剩余的记录就会成为下一个查询结果集中的一部分，并且发生“不同步”的错误。这种情形在使用 mysql_store_result() 时不会发生，因为当函数返回时，所有的行就已被获取。事实上，使用 mysql_store_result() 就不必再自己调用 mysql_fetch_row()。对于所有感兴趣的事情就是是否得到一个非空的结果，而不是结果所包含的内容的查询来说，它是很有用的。例如，要知道表 my_tbl 是否存在，可以执行下面的查询：

```
SHOW TABLES LIKE "my_tbl"
```

如果在调用 mysql_store_result() 之后，mysql_num_rows() 的值为非零，这个表就存在，就不必再调用 mysql_fetch_row() （当然仍需调用 mysql_free_result()）。

如果要提供最大的灵活性，就给用户选择使用任一结果集处理方法的选项。 mysql 和

mysqldump 是执行这个操作的两个程序，缺省时，使用 mysql_store_result()，但是如果指定 --quick 选项，则使用 mysql_use_result()。

6.6.6 使用结果集元数据

结果集不仅包括数据行的列值，而且还包括数据信息，这些信息成为元数据结果集，包括：

- 结果集中的行数和列数，通过调用 mysql_num_rows() 和 mysql_num_fields() 实现。
- 行中每列值的长度，通过调用 mysql_fetch_lengths() 实现。
- 有关每列的信息，例如列名和类型、每列值的最大宽度和列来源的表等。MYSQL_FIELD 结构存储这些信息，通过调用 mysql_fetch_fields() 来获得它。附录 F 详细地描述了 MYSQL_FIELD 结构，并列出了提供访问列信息的所有函数。

元数据的有效性部分决定于结果集的处理方法，如在上节中提到的，如果要使用行计数或者列长度的最大值，就必须用 mysql_store_result() 而不是 mysql_use_result() 创建结果集。

结果集元数据对确定有关如何处理结果集非常有帮助：

- 列名和宽度信息对漂亮地生成带有列标题并垂直排列的格式化输出是非常有用的。
- 使用列计数来确定处理数据行的连续列值的循环所迭代的次数。如果要分配取决于结果集中已知的行数或列数的数据结构，就可以使用行或列计数。
- 可以确定列的数据类型。可以看出列是否是数字的，是否可能包括二进制数据等等。

在前面的6.6.1节“处理返回结果集的查询”中，我们编写了从结果集的行中以制表符分隔的形式打印出结果的 process_result_set() 程序。这对某些目的是很好的（例如要把数据输入到电子制表软件中），但对于可视化检查或打印输出，就不是一个漂亮的显示格式。回忆前面的 process_result_set() 版本，产生过这样的输出：

```
Adams   John    Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A.  Fairfield   VT
Buchanan    James   Mercersburg PA
Bush    George W.   Milton  MA
Carter  James E. Jr Plains  GA
Cleveland   Grover  Caldwell    NJ
...
```

让我们在每列加上标题和边框来对 process_result_set() 做些修改，以生成表格式的输出。这种修正版看上去更美观，输出的结果是相同的，如下所示：

```
+-----------+-------------+------------+-------+
| last_name | first_name  | city       | state |
+-----------+-------------+------------+-------+
| Adams     | John        | Braintree  | MA    |
| Adams     | John Quincy | Braintree  | MA    |
| Arthur    | Chester A.  | Fairfield  | VT    |
| Buchanan  | James       | Mercersburg| PA    |
| Bush      | George W.   | Milton     | MA    |
| Carter    | James E., Jr.| Plains    | GA    |
| Cleveland | Grover      | Caldwell   | NJ    |
| ...       | ...         | ...        | ...   |
+-----------+-------------+------------+-------+
```

显示算法的基本要点是这样的：

1) 确定每列的显示宽度。

2) 打印一列带有边框的列标题（由垂直竖线和前后的虚线分隔）。

3) 打印结果集每行的值、带边框的列（由垂直竖线分隔），并垂直排列，除此之外，打印正确的数字，将 NULL 值打印为单词“ NULL ”。

4) 最后，打印检索的行的计数。

该练习为结果集元数据的使用提供了一个很好的示范。为了显示所描述的输出，除了行所包含的数据值之外，我们还需了解许多有关结果集的内容。

您可能想，“ 这个描述听起来与 mysql 显示的输出惊人地相似 ”。是的，欢迎把 mysql 源代码和修正版的 process_result_set() 代码比较一下，它们是不同的，可以发现对同一问题使用两种方法是有指导作用的。

首先，我们需要确定每列的显示宽度，下面列出如何做这件事情。可观察到这些计算完全基于结果集元数据，无论行值是什么，它们都没有引用：

```
MYSQL_FIELD     *field;
unsigned int    i, col_len;

/* determine column display widths */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
        col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
        col_len = 4;    /* 4 = length of the word "NULL" */
    field->max_length = col_len;    /* reset column info */
}
```

列宽度通过结果集中列的 MYSQL_FIELD 结构的迭代来计算，调用 mysql_fetch_seek() 定位第一个结构，后续的 mysql_fetch_field() 调用返回指向连续列的结构的指针。显示出来的列宽度是下面三个值中的最大值，其中每一个都取决于列信息结构中的元数据：

- field->name的长度，也就是列标题的长度。
- field->max_length，列中最长的数据值的长度。
- 如果列中可能包括 NULL值，则为字符串“ NULL ”的长度，field->flag 表明列是否包含 NULL。

请注意，已知要显示的列的宽度后，我们将这个值赋给 max_length，max_length 是从客户机库获取的结构中的一个成员。这种获取是允许的吗？或者 MYSQL_FIELD 结构的内容应该为只读？一般来说，是“ 只读的 ”，但是 MySQL 分发包中的一些客户机程序以同样的方式改变了 max_length 的值，因此，假设这也是正确的（如果更喜欢不改变 max_length 值的方法，则分配一个 unsigned int 值的数组，将计算的宽度存储到这个数组中）。

显示宽度的计算包括一个说明，回想当使用 mysql_use_result() 创建结果集时，max_length 没有意义。因为我们需要 max_length 来确定列值的显示宽度，所以该算法的正确操作需要使用 mysql_store_result() 产生的结果集 (MYSQL_FIELD结构的 length 成员告知列值可以取得的最大值，如果使用 mysql_store_result() 而不是mysql_use_result()的话，这可能是个有用的工作环境)。

一旦知道了列的宽度，就可以准备打印，处理标题很容易；对于给定的列，只需使用由 field 指向的列信息结构，用已计算过的宽度打印出 name 成员。

```
printf (" %-*s |", field->max_length, field->name);
```

对于数据，我们对结果集中的行进行循环，在每次迭代时打印当前行的列值。从行中打印列值有些技巧，因为值可能是 NULL，也可能代表一个数（无论哪种情况都如实打印）。列值的打印如下，这里 row[i] 包括数据值和指向列信息的 field 指针：

```
if (row[i] == NULL)
    printf (" %-*s |", field->max_length, "NULL");
else if (IS_NUM (field->type))
    printf (" %*s |", field->max_length, row[i]);
else
    printf (" %-*s |", field->max_length, row[i]);
```

如果 field->type 指明的列类型是数字型，如 INT、FLOAT或者DECIMAL，那么宏IS_NUM的值为真。

显示该结果集的最终的代码如下所示。注意，因为我们需要多次打印虚线，所以这段代码封装在它自己的函数中，函数 print_dashes() 是这样的：

```
void
print_dashes (MYSQL_RES *res_set)
{
MYSQL_FIELD     *field;
unsigned int    i, j;

    mysql_field_seek (res_set, 0);
    fputc ('+', stdout);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        for (j = 0; j < field->max_length + 2; j++)
            fputc ('-', stdout);
        fputc ('+', stdout);
    }
    fputc ('\n', stdout);
}
void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_FIELD     *field;
MYSQL_ROW       row;
unsigned int    i, col_len;

    /* determine column display widths */
    mysql_field_seek (res_set, 0);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        col_len = strlen (field->name);
        if (col_len < field->max_length)
            col_len = field->max_length;
        if (col_len < 4 && !IS_NOT_NULL (field->flags))
            col_len = 4;    /* 4 = length of the word "NULL" */
        field->max_length = col_len;    /* reset column info */
    }

    print_dashes (res_set);
    fputc ('|', stdout);
    mysql_field_seek (res_set, 0);
```

```
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
        field = mysql_fetch_field (res_set);
        printf (" %-*s |", field->max_length, field->name);
    }
    fputc ('\n', stdout);
    print_dashes (res_set);

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
        mysql_field_seek (res_set, 0);
        fputc ('|', stdout);
        for (i = 0; i < mysql_num_fields (res_set); i++)
        {
            field = mysql_fetch_field (res_set);
            if (row[i] == NULL)
                printf (" %-*s |", field->max_length, "NULL");
            else if (IS_NUM (field->type))
                printf (" %*s |", field->max_length, row[i]);
            else
                printf (" %-*s |", field->max_length, row[i]);
        }
        fputc ('\n', stdout);
    }
    print_dashes (res_set);
    printf ("%lu rows returned\n", (unsigned long) mysql_num_rows (res_set));
}
```

MySQL 客户机库提供了访问列信息结构的几种方法，例如，前面样例的代码多次使用如下形式的循环访问这些结构：

```
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    field = mysql_fetch_field (res_set);
    ...
}
```

然而，mysql_field_seek() 与 mysql_fetch_field() 的结合是获得 MYSQL_FIELD 结构的唯一途径，可在附录 F 中查看 mysql_fetch_field() 函数和 mysql_fetch_field_direct() 函数，寻找其他获得列信息结构的方法。

6.7 客户机程序5 —— 交互式查询程序

让我们把迄今为止研究的诸多内容整理一下，编写一个简单的交互式客户机程序。它的功能包括可以进入查询，用通用目标查询处理程序 process_query 执行查询，并用前面研究过的显示格式 process_result_set() 显示查询结果。

客户机程序5在某些方面与 mysql 类似，虽然在几个特征上还是有所不同。客户机程序 5 在输入上有几个约束条件：

- 每个输入行必须包括一个完整的查询。
- 查询不会以分号或 ‘ \g ’ 为终止。
- 不识别类似 quit 的命令；而是用 Control-D 结束程序。

客户机程序 5的编写几乎是完全微不足道的（不到 10行的新代码）。客户机程序框架（client4.c）和写过的其他代码几乎提供了所需的每一件事，我们唯一要增加的是搜集输入行

并执行它们的循环。

为了建造客户机程序5，首先把客户机程序框架 client4.c 拷贝到 client5.c 中，然后把代码增加到 process_query()、process_result_set() 和 print_dashes() 中，最后在 client5.c 的 main() 中寻找标有下列字符的行：

```
/* do the real work here */
```

然后用下面的循环替换它：

```
while (1)
{
char    buf[1024];
    fprintf (stderr, "query> ");                    /* print prompt */
    if (fgets (buf, sizeof (buf), stdin) == NULL)    /* read query */
          break;
      process_query (conn, buf);                        /* execute query */
  }
```

编译 client5.c 产生client5.o,将 client5.c 与 common.o和客户机库连接，生成客户机程序 5，到此就全部完成了。您就拥有了一个可执行任意查询并显示结果的交互式 MySQL 客户机程序。

6.8 其他主题

本节包括几个主题，这些主题不完全适合于本章从 client1 到 client5 的开发中的任一小节的内容：

- 在使用结果集元数据帮助验证这些数据适合于计算之后，使用结果集数据计算结果。
- 如何处理很难插入到查询中的数据。
- 如何处理图形数据。
- 如何获得表结构的信息。
- 常见的 MySQL 程序设计错误及如何避免。

6.8.1 在结果集上执行计算

迄今为止，我们集中而主要地使用了结果集元数据来打印行数据，但很明显，除打印之外，还有需要使用数据做其他事情的时候。例如，计算基于数据值的统计信息，应用元数据确保数据适合它们要满足的需求。哪种类型的需求？对于启动程序来说，可能要校验一下正要执行数字计算的列实际上是否包含着数字！

下面的列表显示了一个简单函数 summary_stats(),它获取结果集和列索引，并产生列值的汇总统计。该函数还列出缺少数值的数量，它是通过检查 NULL 来检测的。这些计算包括两个数据所必须满足的需求，summary_stats() 用结果集元数据来校验：

- 指定的列必须存在（也就是说，列索引必须在结果集列值的范围内）。
- 此列必须包括数字值。

如果这些条件不满足，则 summary_stats() 只打印出错误消息并返回。代码如下：

```
void
summary_stats (MYSQL_RES *res_set, unsigned int col_num)
{
MYSQL_FIELD     *field;
MYSQL_ROW       row;
```

```
    unsigned int    n, missing;
    double  val, sum, sum_squares, var;

        /* verify data requirements */
        if (mysql_num_fields (res_set) < col_num)
        {
            print_error (NULL, "illegal column number");
            return;
        }
        mysql_field_seek (res_set, 0);
        field = mysql_fetch_field (res_set);
        if (!IS_NUM (field->type))
        {
            print_error (NULL, "column is not numeric");
            return;
        }

        /* calculate summary statistics */

        n = 0;
        missing = 0;
        sum = 0;
        sum_squares = 0;

        mysql_data_seek (res_set, 0);
        while ((row = mysql_fetch_row (res_set)) != NULL)
        {
            if (row[col_num] == NULL)
                missing++;
            else
            {
                n++;
                val = atof (row[col_num]);  /* convert string to number */
                sum += val;
                sum_squares += val * val;
            }
        }
        if (n == 0)
            printf ("No observations\n");
        else
        {
            printf ("Number of observations: %lu\n", n);
            printf ("Missing observations: %lu\n", missing);
            printf ("Sum: %g\n", sum);
            printf ("Mean: %g\n", sum / n);
            printf ("Sum of squares: %g\n", sum_squares);
            var = ((n * sum_squares) - (sum * sum)) / (n * (n - 1));
            printf ("Variance: %g\n", var);
            printf ("Standard deviation: %g\n", sqrt (var));
        }
    }
```

请注意在 mysql_fetch_row() 循环前面调用的 mysql_data_seek()。为获得同样的结果集，它允许多次调用 summary_stats()（假设要计算几列的统计值的话）。每次调用summary_stats()，都要“重新回到”到结果集的开始（这里假设用 mysql_store_result() 创建结果集，如果用 mysql_use_result() 创建结果集就只能按顺序处理行，而且只能处理一次）。

summary_stats() 是个相对简单的函数，但它给我们一个提示，就是如何编写一个比较复杂的计算程序，如两个列的最小二乘回归或者标准统计，如 t-检验。

6.8.2 对查询中有疑问的数据进行编码

包括引号、空值和反斜线的数据值，如果把它们插入到查询中，在执行查询时就会产生一些问题。下面的讨论论述了这些难点，并介绍了解决的办法。

假设要建造一个 SELECT 查询，它基于由 name 指向的空终结串的内容：

```
char query[1024];

sprintf (query, "SELECT * FROM my_tbl WHERE name='%s'", name);
```

如果 name 的值类似于 “ 0 ’ Malley,Brian ”，这时进行的查询就是非法的，因为引号在引用的字符串里出现：

```
SELECT * FROM my_tbl WHERE name='O'Malley, Brian'
```

需要特别注意这个引号，以便使服务器不将它解释为 name 的结尾。一种方法是在字符串内使用双引号，这就是 ANSI SQL 约定。SQL 支持这个约定，也允许引号在反斜线后使用：

```
SELECT * FROM my_tbl WHERE name='O''Malley, Brian'
SELECT * FROM my_tbl WHERE name='O\'Malley, Brian'
```

另一个有问题之处是查询中任意二进制数据的使用，例如，在把图形存储到数据库这样的应用程序中会发生这种情况。因为二进制数值含有一些字符，把它放到查询中是不安全的。

为了解决这个问题，可使用 mysql_escape_string()，它可以对特殊字符进行编码，使其在引用的字符串中可以使用。mysql_escape_string() 认为的特殊字符是指空字符、单引号、双引号、反斜线、换行符、回车符和 Control-Z（最后一个在 Windows 语言环境中出现）。

什么时候使用 mysql_escape_string() 呢？最保险的回答是 “ 始终 ”。然而，如果确信数据的形式并且知道它是正确的 —— 可能因为预先执行了确认检查 —— 就不必编码了。例如，如果处理电话号码的字符串，它完全由数字和短线组成，那么就不必调用 mysql_escape_string() 了，否则还是要调用。

mysql_escape_string() 对有问题的字符进行编码是将它们转换为以反斜线开头的 2个字符的序列。例如，空字符转换为 ‘ \0 ’,这里的 0 是可打印的 ASCII 码 0，而不是空。反斜线、单引号和双引号分别转换为 ‘ \\ ’、‘ \ ’ ’ 和 ‘ \ ”’。

调用 mysql_escape_string() 的过程如下：

```
to_len = mysql_escape_string (to_str, from_str, from_len);
```

mysql_escape_string() 对 from_str 进行编码，并把结果写入 to_str中，还添加了空终结值，这样很方便，因为可以利用像 strcpy() 和 strlen() 这样的函数使用该结果串。

from_str 指向包括将要编码的字符串的 char 缓冲区，这个字符串可能包含任何内容，其中包括二进制数据。to_str 指向一个存在的 char 缓冲区，在这个缓冲区里，可以写入编码的字符串；不要传递未初始化的指针或 NULL 指针，希望由 mysql_escape_string() 分配空间。由 to_str 指向的缓冲区的长度至少是 (from_len*2)+1 个字节（很可能 from_str 中的每个字符都需要用 2 个字符来编码；额外的字节是空终结值）。

from_len 和 to_len 都是 unsigned int 值，from_len 表示 from_str 中数据的长度；提供这个长度是非常必要的，因为 from_str 可能包含空值字节，不能把它当作空终结串。从 mysql_escape_string() 返回的 to_len 值是作为结果的编码字符串的实际长度，没有对空终结值进行计数。

当 mysql_escape_string() 返回时，to_str中编码的结果就可看作是空终结串，因为

from_str 中的空值都被编码为‘ \0 ’。

为了重新编写构造 SELECT 的代码，使名称的值即使包含引号也能工作，我们进行下面的操作：

```
char query[1024], *p;

p = strcpy (query, "SELECT * FROM my_tbl WHERE name='");
p += strlen (p);
p += mysql_escape_string (p, name, strlen (name));
p = strcpy (p, "'");
```

是的，这很难看，如要简化一点，就要使用第二个缓冲区，如替换成下列内容：

```
char query[1024], buf[1024];

(void) mysql_escape_string (buf, name, strlen (name));
sprintf (query, "SELECT * FROM my_tbl WHERE name='%s'", buf);
```

6.8.3 图像数据的处理

mysql_escape_string() 的基本功能之一就是把图像数据加载到一个表中。本节介绍如何进行这项工作（这个讨论也适用于二进制数据的其他形式）。

假设想从文件中读取图像，并将它们连同唯一的标识符存储到表中。 BLOB 类型对二进制数据来讲是个很好的选择，因此可以使用下面的表说明：

```
CREATE TABLE images
(
    image_id INT NOT NULL PRIMARY KEY,
    image_data BLOB
)
```

实际上，要想从文件中获取图像并放入 images 表，利用下面的函数 load_image() 可以实现，给出一个标识符号码和一个指向包括这个图像数据的打开文件的指针：

```
int
load_image (MYSQL *conn, int id, FILE *f)
{
char            query[1024*100], buf[1024*10], *p;
unsigned int    from_len;
int             status;

    sprintf (query, "INSERT INTO images VALUES (%d,'", id);
    p = query + strlen (query);
    while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
    {
        /* don't overrun end of query buffer! */
        if (p + (2*from_len) + 3 > query + sizeof (query))
        {
            print_error (NULL, "image too big");
            return (1);
        }
        p += mysql_escape_string (p, buf, from_len);
    }
    (void) strcpy (p, "')");
    status = mysql_query (conn, query);
    return (status);
}
```

load_image() 不会分配非常大的查询缓冲区（ 100K），因此它只能处理相对较小的图形。

在实际的应用程序中，可以根据图形文件的大小动态地分配缓冲区。

处理从数据库中恢复的图形数据（或任何二进制数据）并不像开始把它放入时那样问题重重，因为在变量 MYSQL_ROW 中数据值的原始形式是有效的，通过调用 mysql_fetch_length()，这个长度也是有效的。必须将值看作是计数串，而不是空终结串。

6.8.4 获取表信息

MySQL 允许使用下面的查询获取有关表结构的信息（下面两者是等价的）：

```
DESCRIBE tbl_name
SHOW FIELDS FROM tbl_name
```

与 SELECT 相类似，两个语句都返回结果集。为了在表中找出有关列，所需做的就是处理结果集中的行，从中获取有用的信息。例如，如果从 mysql 客户机上发布 DESCRIBE images 语句，就会返回这样的信息：

```
+------------+---------+------+-----+---------+-------+
| Field      | Type    | Null | Key | Default | Extra |
+------------+---------+------+-----+---------+-------+
| image_id   | int(11) |      | PRI | 0       |       |
| image_data | blob    | YES  |     | NULL    |       |
+------------+---------+------+-----+---------+-------+
```

如果从自己的客户机上执行同样的查询，可以得到相同的信息（没有边框）。

如果只想要单个列的信息，则使用如下这个查询：

SHOW FIELDS FROM *tbl_name* LIKE “*col_name*”

此查询会返回相同的列，但只是一行（如果列不存在就不返回行）。

6.8.5 需要避免的客户机程序设计错误

本节讨论一些常见的 MySQL C API 程序设计错误，以及如何避免其发生（这些问题在 MySQL 邮件清单中会周期性地突然出现）。

1. 错误1——使用未初始化的连接处理程序指针

在本章的样例中，我们已经通过传递 NULL 参数调用了 mysql_init()，这就是让它分配并且初始化 MYSQL 结构，然后返回一个指针。另外一种方法是将指针传递到一个已有的 MYSQL 结构中。在这种情况下，mysql_init() 会将结构初始化并返回一个指针，而不必自己分配结构。如果要使用第二种方法，则要小心会出现一些微妙的问题。下面的讨论指出了需要注意的一些问题。

如果将一个指针传递给 mysql_init()，它应该实际指向某些东西。看下面的代码段：

```
main ()
{
MYSQL    *conn;
    mysql_init (conn);
    ...
}
```

这个问题是，mysql_init() 得到了一个指针，但指针没有指向所知的任何地方。conn 是一个局部变量，因此在 main() 开始执行时它是一个能指向任何地方的未初始化的存储器，这就是说 mysql_init() 将使用指针，并可在内存的一些任意区域滥写。如果幸运的话，conn 将指向您的程序地址空间的外部，这样，系统将立即终止，使您能尽早意识到代码中出现的问题。

如果不幸的话，conn 将指向程序中以后才使用的一些数据的内部，直到再次使用那个数据时才发现问题。因此实际出现问题的地方远比执行程序时出现的问题多，也更难捕捉到。

下面是一段有问题的代码：

```
MYSQL    *conn;

main ()
{
    mysql_init (conn);
    mysql_real_connect (conn, ...)
    mysql_query(conn, "SHOW DATABASES");
    ...
}
```

此时，conn 是一个全局变量，因此在程序启动前，将它初始化为 0（就是 NULL）。mysql_init() 遇到 NULL 参数，因此初始化并分配一个新的连接处理程序。只要将 conn 传递给需要非NULL 连接处理程序的 MySQL C API 函数，系统就会崩溃。这些代码段的修改就是确保 conn 有一个可知的值。例如，可以将它初始化到已经分配的 MYSQL 结构地址中去：

```
MYSQL conn_struct, *conn = &conn_struct;
...
mysql_init (conn);
```

然而，推荐的（较容易的！）解决方案仅仅是将 NULL 显式地传递给 mysql_init()，让该函数分配 MYSQL 结构，并将返回值赋值给 conn：

```
MYSQL *conn;
...
conn = mysql_init (NULL);
```

无论如何不要忘记检验 mysql_init() 的返回值，以确保它不是 NULL。

2. 错误2——有效结果集检验的失败

请记住检查希望得到的结果集的调用状态。下面的代码没有做到这一点：

```
MYSQL_RES *res_set;
MYSQL_ROW row;

res_set = mysql_store_result (conn);
while ((row = mysql_fetch_row (res_set)) != NULL)
{
    /* process row */
}
```

不幸地是，如果 mysql_store_result() 失败，res_set 为 NULL，while 循环也不执行了，应测试返回结果集函数的返回值，以确保实际上在进行工作。

3. 错误3—— NULL 列值引起的失败

不要忘记检查mysql_fetch_row() 返回的数组 MYSQL_ROW 中列值是否为 NULL 指针。如果 row[i] 为 NULL，则在一些机器上，下面的代码就会引起崩溃：

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i]);
}
fputc ('\n', stdout);
```

该错误危害最大的部分是，有些 printf() 的版本很宽容地对NULL指针输出了 “ (null) ”，这就使错误很容易逃脱而没有把错误定位。如果把程序给了朋友，而他只有不太宽容 printf() 版本，程序就会崩溃，您的朋友会认为您是个无用的程序员。循环应该写成下面这样：

```
for (i = 0; i < mysql_num_fields (res_set); i++)
{
    if (i > 0)
        fputc ('\t', stdout);
    printf ("%s", row[i] != NULL ? row[i] : "NULL");
}
fputc ('\n', stdout);
```

不需要检查列值是否为 NULL 的惟一一次是当已经从列信息结构确定 IS_NOT_NULL() 为真时。

4. 错误4——传递无意义的结果缓冲区

需要您提供缓冲区的客户机库函数通常要使这些缓冲区真正存在，下面的代码违反了这个规则：

```
char *from_str = "some string";
char *to_str;
unsigned int len;

len = mysql_escape_string (to_str, from_str, strlen (from_str));
```

问题是什么呢？to_str 必须指向一个存在的缓冲区，而在这个样例中没有，因此，它指向了随意的位置。不要向 mysql_escape_string 传递无意义的指针作为 to_str 参数，否则它会恣意践踏内存。