第7章 Perl DBI API

本章介绍如何使用 Perl DBI 与 MySQL 接口。我们不讨论 DBI 的基本原理或体系结构。有关 DBI 这些方面的信息（特别是与 C 和 PHP API 的比较），请参阅第 5 章。

本章的举例动用了样例数据库 samp_db，使用了学分保存方案和历史同盟需要的表。想要从本章中取得最大收获，最好了解一些有关 Perl 的知识。如果不想这样，那么通过拷贝这里看到的样例代码，也能有所帮助，并可以编写自己的脚本，不过找一本好的 Perl 书，可能仍是一件非常有价值的投资。有这样一本书，名为《 Programming Perl》，第二版是由 Wall、Christiansen、Schwartz 和 Potter（O' Reilly出版社1996出版）撰写的。（机械工业出版社1999年已出版了《Perl 5编程详解》 —— 编者注。）

DBI 的当前版本为 1.13，但是此处的大部分介绍也可用于更早的 1.xx 版本。请注意，对所介绍的早期版本中没有出现的特性作了说明。

MySQL 的 DBI 需要至少为 5.004_05 的 Perl 版本。另外还必须安装 Msql-Mysql 模块和 Data-Dumper Perl 模块，以及 MySQL C 客户机库和一些头文件。如果计划编写基于 Web 的 DBI 脚本，则要使用 CGI.pm 模块。本章中，这个模块用于与 Apache Web 服务器的连接。如果需要获得这样的程序包，请参阅附录A。该附录中也给出了获得本章开发的样例脚本的说明。可以下载这些脚本，不必自己键入。

很大程度上，本章介绍 Perl DBI 的方法和变量只是出于讨论的需要。至于所有方法和变量的更全面的列表，请参阅附录 G。 如果要使用 DBI 的任何部分，可以用该附录作为进一步研究的背景材料。可通过运行下面的命令来得到联机文档：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

在数据库驱动程序（DBD）级，MySQL 的驱动程序建立在 MySQL C 客户机库的基础之上，因而具有它的某些特性。有关该库的详细信息，请参阅第 6 章。

7.1 Perl 脚本的特点

Perl 脚本为文本文件，可以利用任何文本编辑器来创建它们。本章所有的 Perl 脚本都遵从 UNIX 的约定，第一行以‘ #！ ’开始，接着是执行这个脚本要使用的程序路径名。第一行如下所示：

```
#! /usr/bin/perl
```

如果在您的系统中，路径名不是 Perl，如为 /usr/local/bin/perl5 或 /opt/bin/perl，则需要修改‘ #! ’行。否则，Perl 脚本不能在系统中正确运行。

在‘ #！ ’之后含有一个空格，这是因为有的系统会将‘ #! / ’解释为4 个字节的怪异数字，所以如果没有空格，则忽略这一行，这样，会将相应脚本作为外壳脚本来对待。

在 UNIX 系统中，应该使 Perl 脚本成为可执行文件，以便只要键入其名称就可执行。为使脚本成为可执行文件，对文件模式做如下更改即可：

```
% chmod +x script_name
```

如果在 Windows 下使用 ActiveState Perl，则不必使脚本成为可执行文件，可如下运行一个脚本：

```
C:\> perl script_name
```

7.2 Perl DBI 基础

本节提供 DBI 的背景信息——在编写自己的脚本和支持其他人编写的脚本时，需要这些信息。如果已经熟悉 DBI，则可以略过这节，直接跳到7.3节“运行 DBI”。

7.2.1 DBI 数据类型

从某些方面来说，使用 Perl DBI API 类似于使用第6章介绍的 C 客户机库。在使用 C 客户机库时，主要依靠指向结构或数组的指针来调用函数和访问与 MySQL 相关的数据。在使用 DBI API 时，除了函数称为方法，指针称为引用外，也调用函数和使用指向结构的指针。指针变量称为句柄，句柄指向的结构称为对象。

DBI 使用若干种句柄。它们往往通过表 7-1 所示的惯用名称在 DBI 文件中引用。而惯用的非句柄变量的名称如表 7-2所示。实际上，在本章中，我们并不使用每个变量名，但是，在阅读其他人编写的 DBI 脚本时，了解它们是有用的。

表7-1 惯用的 Perl DBI 句柄变量名

名 称	说 明
$dbh	数据库对象的句柄
$sth	语句（查询）对象的句柄
$fh	打开文件的句柄
$h	“通用”句柄；其意义取决于上下文

表7-2 惯用的 Perl DBI 非句柄变量的名称

名 称	说 明
$rc	从返回真或假的操作中返回的代码
$rv	从返回整数的操作中返回的值
$rows	从返回行数的操作中返回的值
@ary	查询返回的表示一行值的数组（列表）

7.2.2 一个简单的 DBI 脚本

让我们从一个简单脚本 dump_members开始，它举例说明了 DBI 程序设计中若干标准概念，如与 MySQL 服务器的连接和断开、检索数据等。此脚本产生的结果为以制表符分隔形式列出的历史同盟成员。这个格式本身并不让人感兴趣：在这里，了解如何使用 DBI 比产生漂亮的输出更为重要。

dump_members 如下：

```
#! /usr/bin/perl
```

```
# dump_members - dump Historical League's membership list

use DBI;
use strict;

my ($dsn) = "DBI:mysql:samp_db:localhost"; # data source name
my ($user_name) = "paul";  # user name
my ($password) = "secret"; # password
my ($dbh, $sth);           # database and statement handles
my (@ary);                 # array for rows returned by query

# connect to database
$dbh = DBI->connect ($dsn, $user_name, $password, { RaiseError => 1 });

# issue query
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();

# read results of query, then clean up
while (@ary = $sth->fetchrow_array ())
{
    print join ("\t", @ary), "\n";
}
$sth->finish ();

$dbh->disconnect ();
exit (0);
```

要想自己试验这个脚本，可以下载它（请参阅符录 A），或使用文本编辑器创建它，然后使之可执行，以便能运行。当然，可能至少需要更改一些连接参数（主机名、数据库名、用户名和口令）。本章中的其他 DBI 脚本也是这样。在参数缺省时，本章下载脚本的权限设置为只允许读。如果您将自己的 MySQL 用户名和口令放在它们之中，我建议将它们保留为这种方式，以便其他人不能读取这些值。以后，在 7.2.8节“指定连接参数”中，我们将看到如何从选项文件中获得这些参数，而不是将它们直接放在脚本中。

现在，让我们逐行看完这个脚本。第一行是标准行，指出哪里可以找到 Perl 的指示器：

```
#! /usr/bin/perl
```

在本章将要讨论的脚本中，每个脚本都包含这行；以后不再说明。此脚本中至少应该含有一个简短的目的说明，这是一个好主意，所以下一行是一个注释，给阅读此脚本的人提供一个关于它做什么的线索：

```
# dump_members - dump Historical League's membership list
```

从‘#’字符到行尾部的文本为注释。有必要做一些练习，就是在整个脚本中编写一些注释来解释它们如何工作。

接下来是两个 use 行：

```
use DBI;
use strict;
```

use DBI 告知 Perl 解释程序它需要引入 DBI 模块。如果没有这一行，试图在脚本中做与 DBI 相关的任何事，都将出现错误。不需要指出想要哪个 DBD 级别的模块。在连接数据库时，DBI 会激活相应的模块。

use strict 告知 Perl，在使用它们之前需要声明变量。如果没有 use strict 行，也可以编写脚本，但是，它有助于发现错误，所以建议始终要包括这行。例如，置为严格模式时，如果

声明变量 $my_var，但是之后错误地用 $mv_var 来访问，则在运行这个脚本时，将获得下面的消息：

```
Global symbol "$mv_var" requires explicit package name at line n
```

这个消息会使您想，“怎么了？$mv_var？我从未使用过这种名称的变量！”，然后，找到脚本中的第*n*行，看是什么问题，并改正它。如果不用严格模式，Perl 不会给出 $mv_var；将只是简单地按具有 undef（未定义的）值的该名称创建一个新的变量，并毫无动静地使用它，然后，您会莫名其妙脚本为什么不工作。

因为我们在严格模式下操作，所以我们将定义脚本使用的变量：

```
my ($dsn) = "DBI:mysql:samp_db:localhost"; # data source name
my ($user_name) = "paul";  # user name
my ($password) = "secret"; # password
my ($dbh, $sth);           # database and statement handles
my (@ary);                 # array for rows returned by query
```

现在我们准备连接数据库：

```
# connect to database
$dbh = DBI->connect ($dsn, $user_name, $password, { RaiseError => 1 });
```

connect( ) 调用作为 DBI->connect( ) 来调用，因为它是 DBI 类的方法。不必真正知道它是什么意思；它只是一个使人头痛的面向对象的行话（如果的确想知道，那么它意味着 connect( ) 是“属于”DBI 的一个函数）。connect( ) 有若干参数：

- **数据源**。（经常调用的数据源名称，或 DSN。）数据源格式由要使用的特定 DBD 模块需求来确定。对于 MySQL 驱动程序，允许的格式如下：

  ```
  "DBI:mysql:db_name"
  "DBI:mysql:db_name:host_name"
  ```

  对于第一种格式，主机名缺省为 localhost（实际上有其他允许的数据源格式，我们将在后面7.2.8节“指定连接参数”中讨论）。“DBI”大写没关系，但是“mysql”必须小写。

- 用户名和口令。
- 表示额外连接属性的可选参数。这个参数控制 DBI 的错误处理行为，我们指定的看起来有点奇怪的构造启用了 RaiseError 属性。这导致 DBI 检查与数据库相关的错误，并显示消息，而且只要它检测到错误就退出（这就是为什么在 dump_members 脚本中的任何地方都没有看到错误检查代码的原因；DBI 将它全部处理了）。7.2.3节“处理错误”包括了对错误响应的可选方法。

如果 connect( ) 调用成功，则它返回数据库句柄，我们分配给 $dbh（如果 connect( ) 失败，通常返回 undef。然而，因为我们在脚本中启用了 RaiseError，所以 connect( )不返回；但是，DBI 将显示一条错误消息，并且在出现错误时退出）。

连接到数据库后，dump_members 发布一条 SELECT 语句查询来检索全体成员列表，然后，执行一个循环来处理返回的每一行。这些行构成了结果集。

为了完成 SELECT语句，首先需要准备，然后再运行它：

```
# issue query
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();
```

利用数据库句柄调用 prepare( )；在执行前，它将 SQL 语句传递给预处理的驱动程序。实

际上，在这里某些驱动程序做了一些有关这条语句的事情。其他驱动程序只是记住它，直到调用 execute( ) 使这条语句被执行为止。从 prepare( ) 返回的值是一个语句句柄 $sth，如果出现错误，则为 undef。在进一步处理与这条语句相关的所有内容时，都使用这个语句句柄。

请注意，指定的这个查询没有分号结束符。您无疑有这样的（经过长时间使用 mysql 程序养成的）习惯，用 ‘ ; ’ 字符终止 SQL 语句。然而，在使用DBI时，最好打破这个习惯，因为分号经常导致查询出现语法错误而失败。向查询增加 ‘ \g ’ 也类似，使用 DBI 时不要这样。

在调用一个方法而不用向它传递任何参数时，可以没有这个圆括号。下列两个调用是等价的：

```
$sth->execute ();
$sth->execute;
```

我宁愿有圆括号，因为它使人感到这个调用看上去不像变量。您的选择就可能不同了。

调用 execute( ) 后，可以处理成员列表的行。在 dump_members 脚本中，提取行的循环简单地显示了每行的内容：

```
# read results of query, then clean up
while (@ary = $sth->fetchrow_array ())
{
    print join ("\t", @ary), "\n";
}
$sth->finish ();
```

fetchrow_array( ) 返回含有当前行的列值的数组，在没有剩余的行时，返回一个空数组。这样，此循环提取了由 SELECT 语句返回的连续行，并显示列值之间用制表符分隔的每一行。在数据库中 NULL 作为 undef 值返回到 Perl 脚本，但是将它们显示为空字符串，而不是单词 “NULL ”。

请注意，制表符和换行符（表示为 ‘ \t ’ 和 ‘ \n ’）括在双引号中。在 Perl 中，只解释出现在双引号内的转义符序列，不解释出现在单引号内的转义符序列。如果使用单引号，则输出将为字符串 “ \t ” 和 “ \n ”。

提取行的循环终止以后，调用 finish( ) 告知 DBI 不再需要语句句柄，并且释放分配给它的所有临时资源。实际上，除非只提取结果集的一部分（无论是设计的原因，还是因为出现一些问题），否则不需要调用 finish( )。然而，在提取循环之后，finish( ) 始终是很保险的，我认为调用并执行 finish( )，比区分何时需要，何时不需要更容易一些。

我们已经显示完了全部成员列表，所以我们可以从服务器上断开连接，并且退出：

```
$dbh->disconnect ();
exit (0);
```

dump_members 示出了许多 DBI 程序的大多数通用概念，而且不必了解更多的知识，就可以着手编写自己的 DBI 程序。例如，要想写出一些其他表的内容，所需要做的只是更改传递给 prepare( ) 方法的 SELECT 语句的文本。而且实际上，如果想了解这种技术的某些应用，可略过这部分，直接跳到 7.3节 “ 运行 DBI ” 中讨论如何生成历史同盟一年一度的宴会成员列表程序和 League 打印目录的部分。然而，DBI 提供许多其他有用的功能。下一节介绍了一些，以便能够在 Perl 脚本中看看如何完成比运行一条简单的 SELECT 语句更多的事情。

7.2.3 处理错误

在 dump_members 调用 connect( )方法时，应该启用 RaiseError 错误处理属性，以便这些

错误用一条错误消息就能自动地终止相应的脚本。也可以用其他方式处理这些错误。例如，可以自己检查错误而不必使用 DBI。

为了查看如何控制 DBI 的错误处理行为，我们来仔细查看一下 connect( ) 调用的最终参数。下面两个相关的属性是 RaiseError 和 PrintError：

- 如果启用 RaiseError（设为非零值），如果在 DBI 方法中出现错误，则 DBI 调用 die( ) 来显示一条消息并且退出。
- 如果启用 PrintError，在出现 DBI错误时，DBI 会调用 warn( ) 来显示一条消息，但是相应脚本会继续执行。

缺省时，RaiseError 是禁用的，而 PrintError 启用。在此情况下，如果 connect( )调用失败，则 DBI 显示一条消息，而且继续执行。这样，如果省略 connect( ) 的四个参数，则得到缺省的错误处理行为，可以如下检查错误：

```
$dbh = DBI->connect ($dsn, $user_name, $password)
    or exit (1);
```

如果出现错误，则 connect( ) 返回 undef 表示失败，并且触发对 exit( ) 的调用。因为 DBI 已经显示了错误消息，所以您就不一定要显示它了。

如果明确给出该错误检查属性的缺省值，可如下调用 connect( )。

```
$dbh = DBI->connect ($dsn, $user_name, $password,
                     { RaiseError => 0, PrintError => 1})
    or exit (1);
```

这就需要更多的编写工作，但是即使对不经意的读者，处理错误行为也会更为明显。

如果想自己检查错误，并显示自己的消息，应该禁用 RaiseError 和 PrintError：

```
$dbh = DBI->connect ($dsn, $user_name, $password,
                     { RaiseError => 0, PrintError => 0})
    or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";
```

变量 $DBI::err 和 $DBI::errstr，只用于所显示的 die( ) 调用中，有助于构造错误消息。它们含有 MySQL 错误代码和错误字符串，非常像 C API 函数中的 mysql_errno( )和 mysql_error( )。

如果仅仅要 DBI 处理错误，以便不必自己检查它们，则启用 RaiseError：

```
$dbh = DBI->connect ($dsn, $user_name, $password, { RaiseError => 1 });
```

到目前为止，这是最容易的方法，并且是 dump_members 带来的。如果在脚本退出时，想要执行某种类型的清除代码，启用 RaiseError 可能是不恰当的，尽管在这种情况下，可以重新定义 $SIG{_DIE_} 句柄，可以做想做的事情。

避免启用 RaiseError 属性的另一个原因是 DBI 在它的消息中显示技术信息，如下：

```
disconnect(DBI::db=HASH(0x197aae4)) invalidates 1 active statement. Either
destroy statement handles or call finish on them before disconnecting.
```

对于编程者来说，这是好的信息，但对普通用户可能没有什么意义。在此情形，最好自己检查错误，以便可以显示对期望使用这个脚本的人更有意义的消息。或者也可在这里考虑重新定义 $SIG{_DIE_} 句柄。这样可能很有用，因为它允许启用 RaiseError 来使错误处理简单化，而不是用自己的消息替换 DBI 给出的缺省错误消息。为了提供自己的 _DIE_ 句柄，可在执行任何 DBI 调用以前，进行下面的工作：

```
$SIG{__DIE__} = sub { die "Sorry, an error occurred\n"; };
```

也可以用普通的风格定义一个子例程，并利用这个子例程的引用来设置这个句柄值：

```
sub die_handler
{
    die "Sorry, an error occurred\n";
}

$SIG{__DIE__} = \&die_handler;
```

除了在 connect( ) 调用中逐字传递错误处理属性之外，还可以利用散列定义它们，并传递对这个散列的引用。有人发现以这种方式准备属性设置使脚本更容易阅读和编辑，但是在功能上这两种方法是相同的。下面是一个说明如何使用属性散列的样例：

```
%attr =
(
    PrintError => 0,
    RaiseError => 0
);
$dbh = DBI->connect ($dsn, $user_name, $password, \%attr)
    or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";
```

下面的脚本 dump_members2 举例说明了当要自己检查错误并显示自己的消息时，如何编写脚本。dump_member2 处理和 dump_members 一样的查询，但是明确地禁用 PrintError 和 RaiseError，然后测试每个 DBI 调用的结果。如果出现错误，在退出以前，脚本调用了子例程 bail_out( ) 显示消息及 $DBI::err 和 $DBI::errstr 的内容：

```
#! /usr/bin/perl

# dump_members2 - dump Historical League's membership list

use DBI;
use strict;

my ($dsn) = "DBI:mysql:samp_db:localhost"; # data source name
my ($user_name) = "paul";  # user name
my ($password) = "secret"; # password
my ($dbh, $sth);           # database and statement handles
my (@ary);                 # array for rows returned by query
my (%attr) =               # error-handling attributes
(
    PrintError => 0,
    RaiseError => 0
);

# connect to database
$dbh = DBI->connect ($dsn, $user_name, $password, \%attr)
    or bail_out ("Cannot connect to database");

# issue query
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name")
    or bail_out ("Cannot prepare query");
$sth->execute ()
    or bail_out ("Cannot execute query");

# read results of query
while (@ary = $sth->fetchrow_array ())
{
```

```
    print join ("\t", @ary), "\n";
}
$DBI::err == 0
    or bail_out ("Error during retrieval");

# clean up
$sth->finish ()
    or bail_out ("Cannot finish query");
$dbh->disconnect ()
    or bail_out ("Cannot disconnect from database");
exit (0);

# bail_out() subroutine - print error code and string, then exit

sub bail_out
{
my ($message) = shift;

    die "$message\nError $DBI::err ($DBI::errstr)\n";
}
```

除了 bail_out( ) 是退出而不是返回到调用者以外， bail_out( ) 类似于我们在第6章中为编写 C 程序使用的 print_error( ) 函数。每次想显示错误消息时， bail_out( )解除了写出 $DBI::err 和 $DBI::errstr 名称的麻烦。同样，通过封装显示到子例程的错误消息，可更改子例程使整个脚本中错误消息的格式一致。

dump_member2 脚本在提取行循环的后面有一个测试，这是 dump_members 所没有的。因为如果在 fetchrow_array( ) 中出现错误，dump_members2 不会自动地退出，所以人们判断循环是因为结果集读取完成而终止（正常终止），还是因为出现错误而终止做出确定是很困难的。当然，任何一种方式，循环都将终止，但是如果出现错误，则将删截脚本的输出。如果没有错误检查，运行该脚本的人将无法知道是否有错！如果自己检查错误，应该检查提取循环的结果。

7.2.4 处理不返回结果集的查询

DELETE、INSERT、REPLACE和UPDATE 等执行后不返回行的语句比 SELECT、DESCRIB、EXPLAIN 和 SHOW 等执行后返回行的语句的处理相对要容易一些。为处理一条非 SELECT 语句，利用数据库句柄，将它传递给 do( )。do( ) 方法在一个步骤内准备和执行该查询。例如，开始输入一个新的成员， Marcis Brown，终止日期为 2002 年 6 月 3 日，可以这样做：

```
$rows = $dbh->do ("INSERT member (last_name,first_name,expiration)"
                  . " VALUES('Brown','Marcia','2002-6-3')");
```

do( ) 方法返回涉及行的计数，如果出现错误，则返回 undef。因为各种原因，可能出现错误（例如，这个查询可能是畸形的，或可能没有访问这个表的权力）。对于非 undef 的返回，注意那些没有受到影响的行的情况。当这种情况发生时， do( ) 不返回数字 0；而是返回字符串“ 0E0 ”（0的Perl科学计数法形式）。“ 0E0 ”在数值上等价于 0，但是，在条件测试中将其视为真，以便可以将其与早期的 undef 区别。如果 do( ) 返回 0，则区分是出现了错误（undef）还是“ 没有受到影响的行 ”这两种情况将更困难。使用下面的两个测试之一可以检查错误：

```
if (!defined ($rows)) { # error }
if (!$rows) { # error }
```

在数值环境中，“0E0”与 0 等价。下面的代码将正确地显示 $rows 的任何非 undef 值的行数：

```
if (!$rows)
{
    print "error\n";
}
else
{
    $rows += 0; # force conversion to number if "0E0"
    print "$rows rows affected\n";
}
```

也可以用 printf( ) 使用‘%d’格式显示 $row 来强制进行隐含的数字转换：

```
if (!$rows)
{
    print "error\n";
}
else
{
    printf "%d rows affected\n", $rows;
}
```

do( ) 方法等价于后跟 execute( ) 的 prepare( )。前面的 INSERT 语句可以不调用 do( )，如下发布：

```
$sth = $dbh->prepare ("INSERT member (last_name,first_name,expiration)"
                    . " VALUES('Brown','Marcia','2002-6-3')");
$rows = $sth->execute ();
```

7.2.5 处理返回结果集的查询

本章提供了有关实现 SELECT 查询中提取行循环的若干选项的详细信息 (或其他类似于 SELECT 的返回行的查询,如 DESCRIBE、EXPLAIN 和 SHOW)。还讨论了如何获得结果中行数的计数值，如何处理不需要循环的结果集，以及如何一次检索整个结果集的全部内容等。

1. 编写提取行的循环

dump_members 脚本利用 DBI 方法的标准序列检索数据：prepare( ) 使驱动程序处理查询，execute( ) 开始执行这个查询，fetchrow_array( ) 提取结果集中的每一行，finish( ) 释放与这个查询相关的资源。

prepare( )、execute( ) 和 finish( ) 是处理返回行的查询中非常标准的部分。然而，对于提取的行，fetchrow_array( ) 实际上只是若干方法中的一种（请参阅表 7-3）。

表7-3 DBI 提取行的方法

方 法 名	返 回 值
fetchrow_array( )	行值的数组
fetchrow_arrayref( )	对行值数组的引用
fetch( )	与 fetchrow_arrayref( ) 相同
fetchrow_hashref( )	对行值的散列引用，列名键索引

下面的例子说示出了怎样使用每个提取行方法。这些例子在整个结果集的行中循环，对

于每一行，显示由逗号分隔的列值。在某些情况下，编写这些显示代码还有一些更有效的方法，但是这些例子是以能够说明访问单个列值的语法的方式编写的。

可如下使用 fetchrow_array( )：

```
while (@ary = $sth->fetchrow_array ())
{77
    $delim = "";
    for ($i = 0; $i < @ary; $i++)
    {
        print $delim . $ary[$i];
        $delim = ",";
    }
    print "\n";
}
```

对 fetchrow_array( ) 的每个调用都返回行值数组，不再有行时，返回一个空数组。

选择将返回值分配给数组变量，可以在一组标量变量中提取列值。如果想使用比 $ary[0]、$ary[1] 等更有意义的变量名，就可以这样做。假设要在变量中检索名称和电子邮件值，可使用 fetchrow_array( )，可以如下选择并提取行：

```
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email"
                      . " FROM member ORDER BY last_name");
$sth->execute ();
while (($last_name, $first_name, $suffix, $email)
                                        = $sth->fetchrow_array ())
{
    # do something with variables
}
```

当然，在以这种方式使用一列变量时，必须保证查询按正确的次序选择列。 DBI 不关心 SELECT 语句指定列的次序，所以正确地分配变量是您的职责。在提取行时，使用一种称为参数约束的技术，也可以使列值自动分配给单独的变量。

fetchrow_arrayref( ) 类似于 fetchrow_array( )，但不返回包含当前行的列值的数组，而是返回这个数组的引用，在没有乘余行时，返回 undef。如下使用：

```
while ($ary_ref = $sth->fetchrow_arrayref ())
{
    $delim = "";
    for ($i = 0; $i < @{$ary_ref}; $i++)
    {
        print $delim . $ary_ref->[$i];
        $delim = ",";
    }
    print "\n";
}
```

通过数组引用 $ary_ref 访问数组元素。这类似于引用指针，所以使用了 $ary_ref->[$i] 而不是 $ary[$i]。要想引用整个数组，就要使用 @{$ary_ref} 结构。

fetchow_arrayef( ) 不适合在列表中提取变量。例如，下面的循环不起作用：

```
while (($last_name, $first_name, $suffix, $email)
                                        = @{$sth->fetchrow_arrayref ()})
{
    # do something with variables
}
```

实际上，只要 fetchrow_arrayref( ) 提取一行，这个循环就能正确地运行。但是在没有更

多的行时，fetchrow_arrayref( ) 返回 undef，并且 @{undef} 不合法（它有些像在 C 程序中试图废弃一个 NULL 指针）。

提取行的第三个方法 fetchrow_hashref( )，如下使用：

```
{
    $delim = "";
    foreach $key (keys (%{$hashref}))
    {
        print $delim . $hashref->{$key};
        $delim = ",";
    }
    print "\n";
}
```

对 fetchrow_hashref( ) 的每个调用都返回一个按列名索引的行值散列的引用，在没有更多的行时，返回 undef。在此情况下，列值不按特定的次序出现；Perl 散列的成员是无序的。然而，散列元素是按列名索引的，所以 $hashref 提供了一个单独的变量，可通过它按名称访问任何列值。这使得能按任意需要的次序来提取值（或者它们中的任何子集），而且不必知道 SELECT 查询检索的列的次序。例如，如果想访问名称和电子邮件域，可以如下进行：

```
while ($hashref = $sth->fetchrow_hashref ())
{
    $delim = "";
    foreach $key ("last_name", "first_name", "suffix", "email")
    {
        print $delim . $hashref->{$key};
        $delim = ",";
    }
    print "\n";
}
```

如果希望将一行值传递给某个函数而又不需要这个函数知道 SELECT 语句中指定列的次序时，fetchrow_hashref( ) 是非常有用的。既然如此，可以调用 fetchrow_hashref( ) 来检索行，并且编写一个使用列名访问来自行散列值的函数。

如果使用 fetchrow_hashref( )，请记住下列警告：

- 如果性能很重要，则 fetchrow_hashref( ) 并不是最好的选择，因为它没有 fetchrow_array( ) 或 fetchrow_arrayref( ) 的效率高。
- 作为散列键值使用的列名具有与 SELECT 语句中写出时相同的字符。在 MySQL 中，列名不区分大小写，所以此查询也是这样，不管以大写字母还是小写字母给出列名，查询结果都是一样的。但是 Perl 散列索引名是区分大小写的，这可能会带来一些问题。为了避免潜在的大小写不匹配问题，可通过传递 NAME_lc 或 NAME_uc 属性，告知 fetchrow_hashref( ) 强迫列名为大写或小写：

```
$hash_ref = $sth->fetchrow_hashref ('NAME_lc'); # use lowercase names
$hash_ref = $sth->fetchrow_hashref ('NAME_uc'); # use uppercase names
```

- 散列对每个唯一的列名含有一个元素。如果正在执行从多个具有重叠名称的表中返回列的连接，则不能访问所有的列值。例如，如果发布下面的查询，fetchrow_hashref( ) 将返回只有一个元素的散列：

```
SELECT a.name, b.name FROM a, b WHERE a.name = b.name
```

2. 确定查询返回的行数

如何知道 SELECT 或类似于 SELECT 的查询返回的行数？一种方法是，当提取它们时，计算这些行的数量。实际上，这是知道 SELECT 查询返回多少行的唯一方便的方法。使用MySQL驱动程序，可以在调用 execute( ) 后利用语句句柄调用 rows( ) 方法，但是这对其他数据库引擎并不方便。而且即使就 MySQL 来说，如果已经设置了 mysql_use_result 属性，rows( ) 也不能返回正确的结果，直到提取了所有行（有关的详细信息，请参阅附录 G）。所以只能如提取行一样对它们进行计数。

3. 提取单行的结果

如果结果集只含单个行，则不需要运行循环来获得结果。假设要编写得出历史同盟成员当前数量的脚本 count_members。完成查询的代码如下所示：

```
# issue query
$sth = $dbh->prepare ("SELECT COUNT(*) FROM member");
$sth->execute ();

# read results of query, then clean up
$count = $sth->fetchrow_array ();
$sth->finish ();
$count = "can't tell" if !defined ($count);
print "$count\n";
```

SELECT 语句只返回一行，所以不需要循环；我们只调用 fetchrow_array( ) 一次。另外，因为我们只选择一列，所以甚至不需要将返回值分配给数组。当在标量环境中（单个值而不是所期望的一列）调用 fetchrow_array( ) 时，它返回这个行的第一列，如果没有更多的有效行，则返回 undef。

另一种期望最多有一个记录的查询是一个含有 LIMIT 1 来约束返回的行数的查询。其一般的用法是返回特定列含有最大或最小值的行。例如，下面的查询给出最近出生的总统姓名和出生日期：

```
$query = "SELECT last_name, first_name, birth"
       . " FROM president ORDER BY birth DESC LIMIT 1";
$sth = $dbh->prepare ($query);
$sth->execute ();

# read results of query, then clean up
($last_name, $first_name, $birth) = $sth->fetchrow_array ();
$sth->finish ();
if (!defined ($last_name))
{
    print "Query returned no result\n";
}
else
{
    print "Most recently born president: $first_name $last_name ($birth)\n";
}
```

必须无提取循环的其他类型的查询利用 MAX( ) 或 MIN( ) 来选择单个值。但是在所有这些情况下，获得单个行结果的一种更容易的方法就是使用数据库句柄方法 selectrow_array( )，它结合了 prepare( )、execute( ) 并在单个调用中提取行。它返回一个数组（而不是一个引用），如果出现错误，则返回一个空数组。前一例子可利用 selectrow_array( ) 编写如下：

```
$query = "SELECT last_name, first_name, birth"
       . " FROM president ORDER BY birth DESC LIMIT 1";
($last_name, $first_name, $birth) = $dbh->selectrow_array ($query);
if (!defined ($last_name))
```

```
{
    print "Query returned no result\n";
}
else
{
    print "Most recently born president: $first_name $last_name ($birth)\n";
}
```

4. 处理完整的结果集

在使用提取循环时，DBI 不提供在结果集中随意查找的方法，或以任何次序而不是以循环返回的次序来处理行。同样，提取行以后，如果没有保存，前一行会丢失。这种做法并不一定合适以下情况：

- 以非连续的次序处理行。考虑一种情况，想以历史同盟的 president 表中列出的美国总统为主体，进行一些测验。如果希望每次测验时都以不同的次序提出问题，则可以从 president 表中选择所有行。然后，可能以任意的次序提取行来改变与所问问题有关的总统的次序。要想任意地提取一行，就必须同时访问所有的行。
- 只使用返回行的子集，对其进行随机选择。例如，当问及总统出生地时，要想出现多个选择的问题，则可以随便地提取一行来选择总统（正确的答案），然后再从取来干扰的选择中提取若干其他行。
- 即使确实以连续的次序去处理，也想紧紧抓住整个结果集。如果想经过这些行进行多个传递，这可能是必需的。例如，在统计计算中，可能先浏览一遍结果集，来估计数据的一些通用数字属性，然后再次检查这些行，来实现更加明确的分析。

可以用几个不同的方式作为一个整体访问结果集。可以完成这个常见的提取循环，并在提取它时保存每一行，可以使用一次返回整个结果集的方法。无论哪种方法都以在结果集中包括一行一行的矩阵作为结束，和选择的列一样多。可以以任何次序任意多次地处理矩阵的元素。下面的讨论说明这两种方法。

使用提取循环来捕获结果集的一种方法是使用 fetchrow_array( ) 并保存对这些行引用的数组。除了保存所有的行，然后显示矩阵举例说明了如何确定矩阵中的行数和列数，及如何访问矩阵的个别成员以外，下面的代码和 dump_members 中提取和显示的循环作用是一样的。

```
my (@matrix) = (); # array of array references

while (my @ary = $sth->fetchrow_array ())   # fetch each row
{
    push (@matrix, [ @ary ]); # save reference to just-fetched row
}
$sth->finish ();

# determine dimensions of matrix
my ($rows) = scalar (@matrix);
my ($cols) = ($rows == 0 ? 0 : scalar (@{$matrix[0]}));

for (my $i = 0; $i < $rows; $i++)           # print each row
{
my ($delim) = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        print $delim . $matrix[$i][$j];
        $delim = ",";
    }
```

```
        print "\n";
    }
```

在确定矩阵的维数时，必须首先确定行数，因为无论这个矩阵是否为空，都可能计算列数。如果 $rows 为0，则这个矩阵为空，并且 $cols 也为0。否则，列数可能作为行数组中的元素数量来计算，用语法 @{$matrix[$i]} 来整体访问行 $i。

在前述的样例中，我们提取每一行，然后保存对它的引用。可以设想调用 fetchrow_arrayref( ) 而不是直接地检索行引用可能更有效率：

```
my (@matrix) = (); # array of array references

while (my $ary_ref = $sth->fetchrow_arrayref ())
{
    # this will not work!
    push (@matrix, $ary_ref); # save reference to just-fetched row
}
$sth->finish ();
```

它不能正常工作，因为 fetchrow_arrayref( ) 重新使用了引用指向的数组。结果矩阵是一个引用的数组，数组中的每个元素都指向相同行 —— 最后检索的行。因此，如果想一次提取一行，则要使用 fetchrow_array( ) 而不是 fetchrow_arrayref( )。

另一个选择是使用提取循环，可以使用返回整个结果集的 DBI 方法中的一个。例如，fetchall_arrayref( ) 返回对引用数组的引用，数组的每个元素都指向结果集中某行。这非常简单，但很有效，这个返回值是对矩阵的引用。要想使用 fetchall_arrayref( )，则调用 prepare( ) 和 execute( )，然后如下检索结果：

```
my ($matrix_ref);  # reference to array of references

$matrix_ref = $sth->fetchall_arrayref ();   # fetch all rows

# determine dimensions of matrix
my ($rows) = (!defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my ($cols) = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++)           # print each row
{
my ($delim) = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        print $delim . $matrix_ref->[$i][$j];
        $delim = ",";
    }
    print "\n";
}
```

如果结果集为空，则 fetchall_arrayref( ) 返回一个对空数组的引用。如果出现错误，则结果为 undef，所以如果没有启用 RaiseError，则在开始使用它以前，要确保检查返回值。

行数和列数由矩阵是否为空来确定。如果想作为一个数组访问这个矩阵的整个行 $i，应该使用语法 @{$matrix_ref->[$i]}。

使用 fetchall_arrayref( ) 来检索结果集当然比编写一个提取行的循环要更简单一些，尽管访问数组元素的语法是一个小技巧。一个与 fetchall_arrayref( ) 方法相类似，但却做了更多工作的方法是selectall_arrayref( )。这个方法为您完成了整个 prepare( )、execute( )、提取循环、finish( ) 序列。为了使用 selectall_arrayref( )，应该利用数据库句柄直接将查询传递给它：

```
my ($matrix_ref);  # reference to array of references

$matrix_ref =
    $dbh->selectall_arrayref ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");

# determine dimensions of matrix
my ($rows) = (!defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my ($cols) = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++)           # print each row
{
my ($delim) = "";
    for (my $j = 0; $j < $cols; $j++)
    {
        print $delim . $matrix_ref->[$i][$j];
        $delim = ",";
    }
    print "\n";
}
```

5. 检查 NULL 值

从数据库中检索数据时，可能需要区分值为 NULL 、为 0 或者为空字符串的列。因为 DBI 返回 NULL 列值作为 undef，所以这种区分是容易的。然而，必须确保使用正确的测试。如果试用下面的代码段，则它所有三次显示都为“ false! ”：

```
$col_val = undef; if (!$col_val) { print "false!\n"; }
$col_val = 0;     if (!$col_val) { print "false!\n"; }
$col_val = "";    if (!$col_val) { print "false!\n"; }
```

而且，对于这两个测试，这段代码都显示“ false! ”：

```
$col_val = undef; if ($col_val eq "") { print "false!\n"; }
$col_val = "";    if ($col_val eq "") { print "false!\n"; }
```

下面这段代码效果相同：

```
$col_val = "";
if ($col_val eq "") { print "false!\n"; }
if ($col_val == 0)  { print "false!\n"; }
```

要想区分 NULL 列值和非 NULL 列值，则使用 defined( )。知道了没有出现 NULL 值之后，使用适当的测试可以在其他类型值之间加以区分。例如：

```
if (!defined ($col_val)) { print "NULL\n"; }
elsif ($col_val eq "")   { print "empty string\n"; }.
elsif ($col_val == 0)    { print "zero\n"; }
else                     { print "other\n"; }
```

以适当的次序完成这些测试是很重要的，因为如果 $col_val 为空字符串，则第二个和第三个比较就都为真。如果颠倒比较的次序，则会错误地将空字符串标识为 0。

7.2.6 引用问题

迄今为止，我们已经利用引用字符串以最基本的方式构造了查询。在引用的字符串含有引用值时，会在 Perl 词汇一级产生问题。在插入或者选择含有引号、反斜杠或二进制数据的值时，在 SQL 中也可能出问题。如果指定一个查询作为 Perl 引用的字符串，则必须避免在查询字符串本身中出现引用字符：

```
$query = 'INSERT absence VALUES(14,\'1999-9-16\')';
$query = "INSERT absence VALUES(14,\"1999-9-16\")";
```

Perl 和 MySQL 都允许用单引号或双引号引用字符串，所以混合使用引用字符有时可以避免这种无法引用引用字符自身的情况：

```
$query = 'INSERT absence VALUES(14,"1999-9-16")';
$query = "INSERT absence VALUES(14,'1999-9-16')";
```

然而，在 Perl 中，这两种类型的引号并不等价。只有在双引号内部才解释为变量引用。因此，当想通过在查询字符串中嵌入变量引用来构造查询时，单引号并不是非常有用的。例如，如果 $var 的值为 14，则下面的两个字符串并不等价：

```
"SELECT * FROM member WHERE id = $var"
'SELECT * FROM member WHERE id = $var'
```

两个字符串的解释如下所示；显然，第一个字符串与希望传递给 MySQL 服务器的内容更为相像：

```
"SELECT * FROM member WHERE id = 14"
'SELECT * FROM member WHERE id = $var'
```

用双引号来引用字符串的另一个选择是使用 qq{} 结构，它告诉 Perl 在‘ qq{ ’和‘ } ’之间的每个字符都要看作为双引号括起的字符串（两个 q 表示“双引号”）。例如，下列两行是等价的：

```
$date = "1999-9-16";
$date = qq{1999-9-16};
```

使用 qq{} 时，构造查询不用过多考虑引号的问题，因为可以在这个查询字符串内自由地使用引号（单引号或双引号），而不用避开它们。此外，还解释了变量引用。 qq{} 的这两种特性可用下面的 INSERT 语句来说明：

```
$id = 14;
$date = "1999-9-16";
$query = qq{INSERT absence VALUES($id,"$date")};
```

不一定使用‘ { ’和‘ } ’作为 qq 的分隔符。其他格式，如 qq( ) 和 qq//，也可以使用，只要封闭的分隔符不出现在字符串内即可。我喜欢用 qq{}，因为‘ { ’不像‘) ’或‘ / ’会出现在查询的文本内，并且在查询字符串的结尾也可能有问题。例如，‘) ’出现在所显示的 INSERT 语句的内部，所以 qq( ) 对于引用查询字符串来说不是一个有用的结构。

qq{} 结构能跨行，如果想让查询字符串在 Perl 代码中醒目，这很有用：

```
$id = 14;
$date = "1999-9-16";
$query = qq{
    INSERT absence VALUES($id,"$date")
};
```

如果希望将查询格式分为多个行，从而使它的可读性更强，这样也很有用。例如，dump_members 脚本中的 SELECT 语句如下：

```
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
    . "street, city, state, zip, phone FROM member ORDER BY last_name");
```

用 qq{}编写如下：

```
$sth = $dbh->prepare (qq{
            SELECT
                last_name, first_name, suffix, email,
```

```
        street, city, state, zip, phone
    FROM member
    ORDER BY last_name
});
```

双引号字符串也可以跨行。但是，对于编写多行的字符串，我更喜欢用 qq{}。我发现当在一行中看到不匹配的引号时，我就自然地去想，“这会是语法错误吗？”，然后，我就会浪费时间去寻找相匹配的引号。

qq{} 结构在 Perl 词汇级注意了引用的问题，因此可将引号容易地放到字符串内，而不会使 Perl 搞混它们。然而，还必须考虑 SQL 级的语法。考虑向 member 表中插入一条记录：

```
$last = "O'Malley";
$first = "Brian";
$expiration = "2002-9-1";
$rows = $dbh->do (qq{
    INSERT member (last_name,first_name,expiration)
    VALUES('$last','$first','$expiration')
});
```

do( ) 发送给 MySQL 的字符串如下所示：

```
INSERT member (last_name,first_name,expiration)
VALUES('O'Malley','Brian','2002-9-1')
```

这是不合法的 SQL 语句，因为在单引号字符串内出现了单引号。在第 6章中，我们遇到过类似的引用问题。在那里，我们使用 mysql_escape_string( ) 来处理这个问题。DBI 提供了一个类似的机制 —— 在一条语句中，对想按字面使用的每个引用值，都调用 quote( ) 方法，并使用它的返回值。

前面的例子可编写如下：

```
$last = $dbh->quote ("O'Malley");
$first = $dbh->quote ("Brian");
$expiration = $dbh->quote ("2002-9-1");
$rows = $dbh->do (qq{
    INSERT member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
});
```

现在，do( ) 发送给 MySQL 的字符串如下所示，具有出现在引用字符串内的可能对服务器转义的引号：

```
INSERT member (last_name,first_name,expiration)
VALUES('O\'Malley','Brian','2002-9-1')
```

请注意，在查询字符串中引用 $last 和 $first 时，不要增加括起来的引号；quote( ) 方法支持它们。如果增加了引号，则查询将出现过多的引号，如下面的例子所示：

```
$value = "paul";
$quoted_value = $dbh->quote ($value);

print "... WHERE name = $quoted_value\n";
print "... WHERE name = '$quoted_value'\n";
```

这些语句产生下面的输出：

```
... WHERE name = 'paul'
... WHERE name = ''paul''
```

7.2.7 占位符和参数约束

在前面各节中，我们通过把要插入或选择的值作为选择标准，直接放在查询字符串中构造了查询。不一定非要这样做。DBI允许在查询字符串内部放置一些称为占位符的特殊标记符，然后，在执行该查询时，将这些值代替那些标识符来使用。这样做的主要原因是提高性能，特别是在循环中反复执行某个查询的时候。

为了说明占位符如何工作，举例说明。假设学校新学期刚开始，打算清理学分薄的student 表，然后利用包含在文件中的一列学生姓名将其初始化，使其包含新学生。不用占位符，可以如下这样删除现有表的内容，并装入新的姓名：

```
$dbh->do (qq{ DELETE FROM student } );  # delete existing rows
while (<>)                              # add new rows
{
    chomp;
    $_ = $dbh->quote ($_);
    $dbh->do (qq{ INSERT student SET name = $_ });
}
```

这样做效率很低，因为 INSERT 查询的基本格式每次都是相同的，并且在整个循环中，do( ) 每次都调用 prepare( ) 和 execute( )。在进入这个循环以前，只调用一次 prepare( ) 来设置 INSERT 语句，并且在这个循环内部只调用 execute( )，这样做效率更高一些。只调用一次prepare( )，可避免其他多次调用。DBI 允许我们这样做：

```
$dbh->do (qq{ DELETE FROM student } );  # delete existing rows
$sth = $dbh->prepare (qq{ INSERT student SET name = ? });
while (<>)                              # add new rows
{
    chomp;
    $sth->execute ($_);
}
$sth->finish();
```

请注意这个 INSERT 查询中的‘？’就是一个占位符。调用 execute( ) 时，将查询发送给服务器，传递这个值来代替占位符。一般来说，如果发现在循环内部调用了 do( )，应该在循环前调用 prepare( )，并在这个循环内部调用 execute( ) 更好一些。

有关占位符的一些注意事项：

- 在查询字符串内，不要在引号中封装占位符字符。如果这样做，不能识别为占位符。
- 不要使用 quote( ) 方法来指定占位符的值，否则将在插入的值中得到额外的引号。
- 在查询字符串中可以有一个以上的占位符，但是要确保占位符的标记符与传递给execute( ) 的值一样多。
- 每个占位符都必须指定一个单独的值，而不是一列值。例如，不能运行这样的语句：

```
$sth = $dbh->prepare (qq{{
            INSERT member last_name, first_name VALUES(?)
});
$sth->execute ("Adams,Bill,2003-09-19");
```

必须这样：

```
$sth = $dbh->prepare (qq{{
            INSERT member last_name, first_name VALUES(?,?,?)
});
$sth->execute ("Adams","Bill","2003-09-19");
```

- 为了将 NULL 指定为占位符，应该使用 undef。
- 不要对关键字使用占位符。这样会出问题，因为占位符的值是由 quote( ) 自动处理的。关键字将被放在引号括起来的查询中，因此，这个查询会由于语法错误而失败。

除了在循环中提高效率以外，对于某些数据库引擎，可以从占位符的使用中获得其他的性能好处。某些引擎高速缓存了准备好的查询，以及为有效地运行这个查询所生成的计划。也就是说，如果以后这个服务器收到同样的查询，则它可以再次使用相应的计划而不用生成。查询高速缓存特别有助于复杂的 SELECT 语句，因为可能需要花费时间生成较好的执行计划。占位符提供了一个在高速缓存中寻找查询的好机会，因为它们使查询比直接在查询字符串中嵌入指定的列值来构造查询更通用。对于 MySQL，在这种方式下，占位符并不提高性能，因为没有高速缓存查询。然而，可能仍想使用占位符编写自己的查询；如果偶然将 DBI 脚本传递给支持查询高速缓存的引擎，则这个脚本比没有占位符时运行效率更高。

在查询运行时，允许在查询字符串中用占位符代替这些值。换句话说，可以参数化这个查询的“输入”。在提取行而不必将值赋给变量时，DBI 也提供一个称为参数约束的输出操作，允许通过检索自动进入这些变量的列值使“输出”参数化。

假设有一个查询，检索 member 表中的成员姓名。可以告诉 DBI 将选定列的值赋给 Perl 变量。在提取行时，变量利用相应的列值自动进行更新。下面是一个例子，说明如何将这些列约束到变量上，然后在提取循环中访问它们：

```
$sth = $dbh->prepare (qq{
            SELECT last_name, first_name, suffix
            FROM member
            ORDER BY last_name, first_name
});
$sth->execute ();
$sth->bind_col (1, \$last_name);
$sth->bind_col (2, \$first_name);
$sth->bind_col (3, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch;
```

bind_col( ) 的每个调用都应该指定一个列号和一个希望与该列相联的变量的引用。列号从 1 开始。bind_col( ) 应该在 execute( ) 之后调用。

还有一种选择，就是单独调用 bind_col( )，可以在 bind_columns( ) 的单个调用中传递全部变量引用：

```
$sth = $dbh->prepare (qq{
            SELECT last_name, first_name, suffix
            FROM member
            ORDER BY last_name, first_name
});
$sth->execute ();
$sth->bind_columns (\$last_name, \$first_name, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch;
```

7.2.8 指定连接参数

建立服务器的连接的最直接的方法为，调用 connect( ) 方法时指定所有连接参数：

```
$data_source = "DBI:mysql:db_name:host_name";
$dbh->connect ($data_source, user_name, password);
```

如果遗漏连接参数，则 DBI 做下面的事情：

- 如果未定义数据源或未定义空字符串，则使用 DBI_DSN 环境变量。如果未定义用户名和口令，则使用 DBI_USER 和 DBI_PASS 环境变量（但如果它们为空字符串则不使用）。在 Windows 下，如果未定义用户名，则使用 user 变量。
- 如果遗漏了主机名，其缺省值为 localhost。
- 如果将用户名指定为 undef 或空字符串，则其缺省为 UNIX 的登录名称。在 Windows 下，用户名缺省为 ODBC。
- 如果将口令指定为 undef 或空字符串，则不传送口令。

通过将某些选项添加到字符串的初始部分，每个都在分号前面，可以在数据源中指定这些选项。例如，可以使用 mysql_read_default_file 选项来指定一个选项文件的路径名：

```
$data_source =
    "DBI:mysql:samp_db;mysql_read_default_file=/u/paul/.my.cnf";
```

当执行这个脚本时，它将从这个文件中读取连接参数。假设 /u/paul/.my.cnf 含有下面的内容：

```
[client]
host=pit-viper.snake.net
user=paul
password=secret
```

然后 connect( ) 调用试图连接到 pit-viper-snake-net 上的 MySQL 服务器，并且用口令 secret 及用户名 paul 连接。如果想允许具有正确地设置选项文件的任何人使用您的脚本，则像这样指定数据源：

```
$data_source =
    "DBI:mysql:samp_db;mysql_read_default_file=$ENV{HOME}/.my.cnf";
```

$ENV{HOME} 含有用户运行这个脚本的主目录的路径名，所以这个脚本使用的主机名、用户名和口令将会从每个用户自己的选项文件中抽取出来。以这种方式编写脚本，不必在这个脚本中逐字地嵌入连接参数。

还可以使用 mysql_read_default_group 选项，来指定一个选项文件组。这自动地导致读取用户的 .my.cnf 文件，并且除了 [client] 组以外，还允许读取一个指定的选项组。例如，如果在 DBI 脚本中具有指定的选项，则可以将它们列在 [dbi] 组中，然后以如下方式使用数据源值：

```
$data_source =
    "DBI:mysql:samp_db;mysql_read_default_group=dbi";
```

mysql_read_default_file 和 mysql_read_default_group 需要 MySQL 3.22.10 或更新的版本，以及 DBD::mysql 1.21.06 或更新的版本。有关指定的数据源字符串的选项的详细信息，请参阅附录G。有关 MySQL 选项文件格式的详细信息，请参阅附录 E。

使用选项文件并不防碍在 connect( ) 调用中指定连接参数（例如，如果想这个脚本作为特殊的用户来连接）。在 connect( ) 调用中指定的任何明确的主机名、用户名和口令值都将覆盖在选项文件中找到的连接参数。例如，想要脚本从命令行中分析 --host、--user 和 --password 选项，并使用那些值，如果给定，则优先于在选项文件中发现的任何内容。这是有用的，因为它是标准的 MySQL 客户机操作的方式。DBI 脚本将因此符合它的行为。

对于在本章中我们开发的保留在命令行中的脚本，我将使用一些标准的连接设置代码及卸载代码。我只在这里说明它一次，以便我们可以将精力集中在每个脚本的主体上，我们编写如下代码：

```
#! /usr/bin/perl

use DBI;
use strict;

# parse connection parameters from command line if given

use Getopt::Long;
$Getopt::Long::ignorecase = 0; # options are case sensitive

# default parameters - all missing
my ($host_name, $user_name, $password) = (undef, undef, undef);

# GetOptions doesn't seem to allow -uuser_name form, only -u user_name?
GetOptions(
    # =s means a string argument is required after the option
    "host|h=s"      => \$host_name
    ,"user|u=s"      => \$user_name
    # :s means a string argument is optional after the option
    ,"password|p:s"  => \$password

) or exit (1);

# solicit password if option specified without option value
if (defined ($password) && !$password)
{

    # turn off echoing but don't interfere with STDIN
    open (TTY, "/dev/tty") or die "Cannot open terminal\n";
    system ("stty -echo < /dev/tty");
    print STDERR "Enter password: ";
    chomp ($password = <TTY>);
    system ("stty echo < /dev/tty");
    close (TTY);
    print STDERR "\n";
}

# construct data source

my ($dsn) = "DBI:mysql:samp_db";
$dsn .= ":hostname=$host_name" if $host_name;
$dsn .= ";mysql_read_default_file=$ENV{HOME}/.my.cnf";

# connect to server
my (%attr) = ( RaiseError => 1 );
my ($dbh) = DBI->connect ($dsn, $user_name, $password, \%attr);
```

这个代码初始化 DBI，在命令行中查找连接参数，然后使用命令行中的或者在用户运行这个脚本的 -/.my.cnf 文件中所找到的参数，连接到 MySQL 服务器。如果在主目录中设置 .my.cnf 文件，则当运行这个脚本时，不一定要输入任何连接参数（请记住，设置这种方式，以便没有其他人读取这个文件。有关的指导请参阅附录 E）。

我们脚本的最后部分也类似于从脚本到脚本；它简单地终止这个连接并退出：

```
$dbh->disconnect ();
exit (0);
```

当我们读到 Web 程序设计的部分，即7.4节“在 Web 应用程序中使用 DBI”时，将修改一些这个连接设置代码，但是基本的思想是类似的。

7.2.9 调试

当想调试有故障的 DBI 脚本时，通常使用两项技术，即单独使用一个或一前一后地配合使用。首先，在脚本的整个过程中编写显示语句。它允许将自己调试的输出设计为想要的方式，但必须手工地增加语句。其次，可以使用 DBI 的内建跟踪能力。这更加通用，但也更加系统，而且它在打开以后，则会自动地出现。DBI 跟踪也说明一些除此以外就无法获得的有关驱动程序的操作信息。

1. 使用显示语句调试

在 MySQL 邮件清单中，常见问题之一是：“有一个查询，当我在 mysql 中执行它时运行得很好，但是它不能在我的 DBI 脚本中工作，怎么回事？”寻找发布不同查询的 DBI 脚本和这个发问者所期望的一样是很平常的。如果在执行它之前显示查询，则可能会惊异地看到真正发送到这个服务器上的内容。假设将一个查询键入到 mysql 中（没有终止的分号）：

```
INSERT member (last_name,first_name,expiration)
VALUES("Brown","Marcia","2002-6-3")
```

然后，在 DBI 脚本中试着做相同的事情：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2002-6-3";
$query =
qq{
    INSERT member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
};
$rows = $dbh->do ($query);
```

尽管它是同样的查询，但它不能工作。不是吗？试着显示：

```
print "$query\n";
```

结果如下：

```
INSERT member (last_name,first_name,expiration)
VALUES(Brown,Marcia,2002-6-3)
```

从这个输出中，可以看到是您忘记了 VALUES( ) 列表中这些列值前后的引号。指定查询的正确方法如下：

```
$last = $dbh->quote ("Brown");
$first = $dbh->quote ("Marcia");
$expiration = $dbh->quote ("2002-6-3");
$query =
qq{
    INSERT member (last_name,first_name,expiration)
    VALUES($last,$first,$expiration)
};
```

或者，可以使用占位符指定查询，并传递这些值，直接插入到 do( ) 方法中：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2002-6-3";
$query =
qq{
    INSERT member (last_name,first_name,expiration)
    VALUES(?,?,?)
};
$rows = $dbh->do ($query, undef, $last, $first, $expiration);
```

不幸的是，当做这些的时候，使用显示语句不能看到完整查询的样子，因为直到调用 do( )才能估计占位符的值。当使用占位符时，跟踪可能对调试方法更有帮助。

2. 使用跟踪调试

当试图查出脚本不能正确工作的原因时，可以告知 DBI 来生成跟踪（调试）信息。跟踪级别范围从 0（关闭）到 9（最多信息）。一般来说，跟踪级别 1 和 2 是最有用的。级别 2 跟踪说明正在执行的查询文本（包括占位符替换的结果）、调用 quote( ) 的结果等等。这可能对捕获问题有极大的帮助。

使用 trace( ) 方法，可以从独立的脚本内部控制跟踪，或者可以设置 DBI_TRACE 环境变量来影响所运行的所有 DBI 脚本的跟踪。

要想使用 trace( ) 调用，则传递一个跟踪级别参数，并可以有选择地再传递一个文件名。如果没有指定文件名，则所有的跟踪输出到 STDERR 中；否则，它就转到这个命名的文件中。一些样例如下：

```
DBI->trace (1)                  Level 1 trace to STDERR
DBI->trace (2, "trace.out")     Level 2 trace to "trace.out"
DBI->trace (0)                  Turn trace output off
```

当调用 DBI->trace( ) 时，跟踪所有的 DBI 操作。一个更精细的方法是，可以用独立的处理级别启用跟踪。当没想好脚本中问题的位置，并对在那点出现的每件事的跟踪输出不想插手时，这是有帮助的。例如，如果特定的 SELECT 查询有问题，则可以跟踪与这个查询相关的语句句柄：

```
$sth = $dbh->prepare (qq{ SELECT ... }); # create the statement handle
$sth->trace (1);                         # enable tracing on the statement
$sth->execute ();
```

如果对任何 trace( ) 调用指定一个文件名参数，则无论对 DBI 作为整体还是单独的句柄，所有的跟踪输出都要到那个文件中。

要想对运行的所有 DBI 脚本全部都打开跟踪，则从命令解释程序中设置 DBI_TRACE 环境变量。它的语法取决于使用的命令解释程序：

```
% setenv DBI_TRACE value        For csh, tcsh
$ DBI_TRACE=value               For sh, ksh, bash
$ export DBI_TRACE
C:\> set DBI_TRACE=value        For Windows
```

value 的模式和所有命令解释程序的模式一样：数字 n表示在级别n打开跟踪到 STDERR 中；文件名打开级别 2 跟踪到这个命名的文件，或 n=file_name 打开级别n跟踪到这个命名的文件中。下面的样例使用了 csh 语法：

```
% setenv DBI_TRACE 1                级别1跟踪到 STDERR 中
% setenv DBI_TRACE 1=trace.out      级别1跟踪到 " trace.out " 中
% setenv DBI_TRACE trace.out        级别2跟踪到 " trace.out " 中
```

如果打开跟踪到命令解释程序中的文件，则确保一旦解决了这个问题，就将它关闭。将调试输出增加到这个跟踪文件中，而不用重写它，所以如果不小心，则这个文件可能变得非常大。极其不好的想法是在命令解释程序的启动文件（如 .cshrc、.lonin 或 .profile）中定义 DBI_TRACE！在 UNIX 下，可以使用下面两个命令（csh 语法）之一关闭跟踪：

```
% setenv DBI_TRACE 0
% unsetenv DBI_TRACE
```

对于 sh、ksh 或 bash，这样做：

```
$ DBI_TRACE=0
$ export DBI_TRACE
```

在 Windows 操作系统中，可以使用下面两个命令之一关闭跟踪：

```
C:\> unset DBI_TRACE
C:\> set DBI_TRACE=0
```

7.2.10 使用结果集元数据

可以使用 DBI 来获得访问结果集元数据——也就是有关由查询选择行的描述信息。访问与结果集生成的查询所相关的语句句柄的属性来获得这个信息。提供这些属性中有一些是作为可用于横跨所有数据库驱动程序的标准 DBI 属性（如 NUM_OF_FIELDS，结果集中列的数量）。另外一些是MySQL 特定的，由 DBD::mysql 所提供的 DBI 的 MySQL 驱动程序。这些属性，如 mysql_max_length 告知了每列值的最大宽度，不能用于其他数据库引擎。要想使用任何 MySQL 特定的属性，都必须冒着使脚本不可移植到其他数据库的危险。另一方面，它们可以使它更容易地获得想要的信息。

必须在适当时候请求元数据。一般来说，直到调用 prepare( ) 和 execute( ) 之后，结果集属性才能用于 SELECT 语句。除此之外，在调用 finish( ) 之后，属性可能变为无效。

让我们来看看如何使用 MySQL 的一个元数据属性 mysql_max_length，与保留查询列名的 DBI 级别的 NAME 属性一起使用。我们可以将这些属性提供的信息合并起来，编写一个脚本 box_out，它以交互模式运行 mysql 客户机程序时获得的相同边框风格，从 SELECT 查询产生输出。box_out 的主体如下（可以用任何其他的语句替换 SELECT 语句；编写输出的例程独立于特定的查询）：

```
my ($sth) = $dbh->prepare (qq{
    SELECT last_name, first_name, city, state
    FROM president ORDER BY last_name, first_name
});
$sth->execute (); # attributes should be available after this call

# actual maximum widths of column values in result set
my (@wid) = @{$sth->{mysql_max_length}};
# number of columns
my ($ncols) = scalar (@wid);

# adjust column widths if column headings are wider than data values
for (my $i = 0; $i < $ncols; $i++)
{
my ($name_wid) = length ($sth->{NAME}->[$i]);

    $wid[$i] = $name_wid if $wid[$i] < $name_wid;
}

# print output
print_dashes (\@wid, $ncols);               # row of dashes
print_row ($sth->{NAME}, \@wid, $ncols);    # column headings
print_dashes (\@wid, $ncols);               # row of dashes
while (my $ary_ref = $sth->fetch)
{
```

```
    print_row ($ary_ref, \@wid, $ncols);    # row data values
}
print_dashes (\@wid, $ncols);               # row of dashes
$sth->finish ();
```

用execute( ) 将这个查询初始化之后，我们获得了所需的元数据。 $sth->{NAME} 和 $sth->{mysql_max_length} 给出了列名和每列值的最大宽度。为了在这个查询中为列命名，每个属性值都引用了一个数组，这个数组含有结果集每列中的一个值。

剩余的计算非常类似于在第 6章中开发的客户机程序 5中所使用的那些内容。例如，为避免偏离输出，如果列的名比该列中任何数据值都宽，则我们要向上调整列的宽度值。

输出函数 print_dashes( ) 和 print_row( ) 代码编写如下，它们也类似于客户机程序 5中相应的代码：

```
sub print_dashes
{
my ($wid_ary_ref) = shift;  # column widths
my ($cols) = shift;         # number of columns

    print "+";
    for (my $i = 0; $i < $cols; $i++)
        print "-" x ($wid_ary_ref->[$i]+2) . "+";
    }
    print "\n";
}

# print row of data.  (doesn't right-align numeric columns)

sub print_row
{
my ($val_ary_ref) = shift;  # column values
my ($wid_ary_ref) = shift;  # column widths
my ($cols) = shift;         # number of columns

    print "|";
    for (my $i = 0; $i < $cols; $i++)
    {
        printf " %-*s |", $wid_ary_ref->[$i],
                defined ($val_ary_ref->[$i]) ? $val_ary_ref->[$i] : "NULL";
    }
    print "\n";
}
```

box_out 的输出如下：

```
+-----------+--------------+--------------------+-------+
| 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    |
+-----------+--------------+--------------------+-------+
```

我们的下一个脚本使用了列元数据来产生不同格式的输出。这个脚本 show_member，允许快速浏览历史同盟成员项目，而不用输入任何查询。给出成员的姓，它就这样显示所选择的项目：

```
% show_member artel
```

```
last_name:  Artel
first_name: Mike
suffix:
expiration: 2003-04-16
email:      mike_artel@venus.org
street:     4264 Lovering Rd.
city:       Miami
state:      FL
zip:        12777
phone:      075-961-0712
interests:  Civil Rights,Education,Revolutionary War
member_id:  63
```

使用成员资格号码，或者使用与若干姓相匹配的模式也可以调用 show_members。下面的命令说明成员号码为23的项目，和以字母“ C ”开始的姓的成员项：

```
% show_member 23
% show_member c%
```

show_member 脚本的主体如下所示。它使用了 NAME 属性来确定输出的每行所使用的标号和 NUM_OF_FIELDS 属性，找出这个结果集含有的列数：

```
my ($count) = 0;    # number of entries printed so far
my (@label) = ();   # column label array
my ($label_wid) = 0;

while (@ARGV)       # run query for each argument on command line
{
my ($arg) = shift (@ARGV);
my ($sth, $clause, $address);

    # default is last name search; do id search if argument is numeric
    $clause = "last_name LIKE " . $dbh->quote ($arg);
    $clause = "member_id = " . $dbh->quote ($arg) if $arg =~ /^\d/;

    # issue query
    $sth = $dbh->prepare (qq{
        SELECT * FROM member
        WHERE $clause
        ORDER BY last_name, first_name
    });
    $sth->execute ();

    # get column names to use for labels and
    # determine max column name width for formatting
    # (only do this the first time through the loop, though)
    if ($label_wid == 0)
    {
        @label = @{$sth->{NAME}};
        foreach my $label (@label)
        {
            $label_wid = length ($label) if $label_wid < length ($label);
        }
    }

    # read and print query results, then clean up
    while (my @ary = $sth->fetchrow_array ())
    {
        # print newline before 2nd and subsequent entries
        print "\n" if ++$count > 1;
        foreach (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
```

```
    {
        # print label, and value if there is one
            printf "%-*s", $label_wid+1, $label[$i] . ":";
            print " " . $ary[$i] if $ary[$i];
            print "\n";
        }
    }
    $sth->finish ();
}
```

无论区域是什么， show_member 的目的都是说明一个项目的全部内容。通过使用 SELECT * 来检索所有的列和 NAME 属性来看看它们是什么，即使从 member 表中增加或删除列，这个脚本也会工作而不用做修改。

如果不检索任何行就想知道一个表含有哪些列，则可以发布下面这条查询：

```
SELECT * FROM tbl_name WHERE 1 = 0
```

以正常方式调用 prepare( ) 和 execute( ) 之后，可以从 @{$sth->{NAME}} 中得到列名。然而，请注意，尽管使用“空”查询的这个小窍门可以在 MySQL 下运行，但是它不可移植，而且并不是对所有的数据库引擎都可以工作的。

有关 DBI 和 DBD::mysql 所提供属性的详细信息，请参见附录 G。它完全可以使您确定是想通过避免 MySQL 特定的属性而为可移植性花费努力，还是在可移植性的开销方面利用它们。

7.3 运行 DBI

在这里，已经看到许多涉及 DBI 程序设计的概念，所以让我们继续做一些样例数据库能处理事情。最初，第 1章简述了我们的目标。本章通过编写 DBI 脚本，我们将处理的那些问题在这里列出。

对于学分保存方案，我们想能够检索任何给定的测验或测试的分数。

对于历史联盟，我们想做下面的事情：

- 以不同格式产生成员目录。我们想在年度宴会程序中，以可以用于生成显示目录的格式使用一个只有名称的列表。
- 寻找不久就要更新其成员资格的 League 成员，然后发送电子邮件通知他们。
- 编辑成员项目（毕竟，在更新成员资格时，我们将要更新他们的终止日期）。
- 寻找分享共同兴趣的成员。
- 使这个目录联机。

对于这样一些任务，我们将编写从命令行上运行的脚本。其他任务，我们将在 7.4节“在 Web 应用程序中使用 DBI”中创建脚本，可以与 Web 服务器配合使用。在本章的最后，我们将仍有许多有待完成的目标。将在第 8章“PHP API”中，完成剩余的目标。

7.3.1 生成历史同盟目录

我们的目标之一是能以不同格式产生历史同盟目录的信息。我们将生成的最简单格式是一个年度宴会程序的成员名列表。那可能是一个简单的无格式文本列表。它将成为创建这个程序的一部分较大文档，所以，我们所需要的就是可以粘贴到文档中的一些内容。

对于可显示的目录，则需要一种比无格式文本更好的表示方法，原因是我们想把一些内

容更精细地格式化。这里一个合理的选择为 RTF（丰富的文本格式 Rich Text Format），它是由 Microsoft 开发的一种格式，可以由许多字处理程序来识别。当然， Word 就是这种程序之一，但是许多其他的软件，如 WordPerfect 和 AppleWork 也是可以识别的。不同的字处理程序对 RTF 的支持程度也有所不同，但是我们将使用由即使对最低级别 RTF都确信的任何字处理程序所支持的全部 RTF 规定的一个基本子集。

生成宴会列表和 RTF 目录格式的过程本质上是一样的：发布查询来检索这些项目，然后运行将每个项目提取和格式化的循环。给出了基本的相似之处，就能很好地避免编写两个分开的脚本。所以，我们编写一个单独的脚本 gen_dir，它可以以不同的格式从这个目录生成输出。我们可以这样组织这个脚本：

1) 在编写出项目内容之前，完成这个输出格式可能需要的任何初始化。宴会程序成员列表不需要任何特殊的初始化，但是我们需要为这个 RTF 版本编写一些初始的控制语言。

2) 提取和显示每个项目，将我们要输出的类型适当地格式化。

3) 处理完所有的项目之后，还要完成任何必需的清除和终止。除了这个 RTF 版本需要的一些关闭控制语言以外，宴会列表不需要特殊的处理。

将来，我们可能想使用这个脚本以其他格式编写输出，所以我们通过设置“转换盒”——每个输出格式都有一个元素的散列，使它成为可扩展的。每个元素都指定对给定格式生成适当输出的函数：初始化函数、编写项目函数和清除函数如下所示：

```
# switchbox containing formatting functions for each output format
my (%switchbox) =
(
    "banquet" =>                        # functions for banquet list
    {
        "init"      => undef,           # no initialization needed
        "entry"     => \&format_banquet_entry,
        "cleanup"   => undef            # no cleanup needed
    },
    "rtf" =>                            # functions for RTF format
    {
        "init"      => \&rtf_init,
        "entry"     => \&format_rtf_entry,
        "cleanup"   => \&rtf_cleanup
    }
);
```

由一个格式名（在这种情况下的“ banquet ”和“ rtf ”）标识转换盒的每个元素。我们将编写这个脚本，以便在运行它时可以在命令行中指定想要的格式：

```
% gen_dir banquet
% gen_dir rtf
```

通过以这种方式设置转换盒，我们可以很容易地增加新格式的性能：

1) 编写三个格式化函数。

2) 向转换盒增加一个指向那些函数的新元素。

3) 为了以新的格式产生输出，调用 gen_dir，并在命令行中指定这个格式名。

按照命令行中的第一个参数所选择的适当转换盒项目的代码如下所示。它是由于输出格式的名称为 %switchbox 散列中关键字。如果在转换盒中不存在这样的关键字，则这个格式是无效的。不需要这个代码中的硬连线格式；如果向转换盒增加新的格式，则自动地检测它。

如果在命令行中没有指定格式名，或者指定了一个无效的名称，则这个脚本产生错误消息，并显示一列允许的名称：

```
# make sure one argument was specified on the command line
@ARGV == 1
    or die "Usage: gen_dir format_type\nAllowable formats: "
            . join (" ", sort (keys (%switchbox))) . "\n";

# determine proper switchbox entry from argument on command line
# if no entry was found, the format type was invalid
my ($func_hashref) = $switchbox{$ARGV[0]};

defined ($func_hashref)
    or die "Unknown format: $ARGV[0]\nAllowable formats: "
            . join (" ", sort (keys (%switchbox))) . "\n";
```

如果在命令行指定了一个有效的格式名，则前述的代码设置 $func_hashref。它的值将是指向选择了格式输出的编写函数的散列引用。然后我们可以运行这个选择项目的查询。之后，我们调用初始化函数、提取和显示这些项目，并激活清除函数：

```
# issue query
my ($sth) = $dbh->prepare (qq{
    SELECT * FROM member ORDER BY last_name, first_name
});
$sth->execute ();

# invoke initialization function if there is one
&{$func_hashref->{init}} if defined ($func_hashref->{init});

# fetch and print entries if there is an entry formatting function
if (defined ($func_hashref->{entry}))
{
    while (my $entry_ref = $sth->fetchrow_hashref ("NAME_lc"))
    {
        # pass entry reference to formatting function
        &{$func_hashref->{entry}} ($entry_ref);
    }
}
$sth->finish ();

# invoke cleanup function if there is one
&{$func_hashref->{cleanup}} if defined ($func_hashref->{cleanup});
```

因为某种原因，提取项目的循环使用了 fetchrow_hashref( )。如果这个循环提取数组，则这个格式化函数必须知道列的次序。它可能通过访问 $sth->{NAME} 属性（它含有返回次序的列名）来得到，但为什么烦扰呢？通过使用散列引用，格式化函数将只能命名那些想使用 $entry_ref->{col_name} 的列值。那样效率就非常低，但它容易做到，并可用于想生成的任何格式，因为我们知道我们需要的任何域都在散列中。

剩余的工作就是为每种输出格式编写这些函数（也就是说，通过转换盒项目为这些函数命名）。

1. 生成宴会程序成员列表

对于这种输出格式，我们只想要成员的姓名。不需要初始化或清除调用。只需要一个项目格式化函数：

```
sub format_banquet_entry
```

```
    {
    my ($entry_ref) = shift;
    my ($name);

    $name = $entry_ref->{first_name} . " " . $entry_ref->{last_name};
    if ($entry_ref->{suffix})           # there is a name suffix
    {
        # no comma for suffixes of I, II, III, etc.
        $name .= "," unless $entry_ref->{suffix} =~ /^[IVX]+$/;
        $name .= " " . $entry_ref->{suffix} if $entry_ref->{suffix};
    }
    print "$name\n";
}
```

format_banquet_entry( ) 的参数是行的列值的散列引用。这个函数将名和姓连在一起，加上可能出现的任何后缀。这里的窍门是如“ Jr. ”或“ Sr. ”后缀的前面应该有一个逗号或空格，但是如“ II ”或“ III ”后缀的前面只能为一个空格：

```
Michael Alvis IV
Clarence Elgar, Jr.
Bill Matthews, Sr.
Mark York II
```

因为字母‘ I ’、‘ V ’和‘ X ’覆盖了所有生成的数字，从第 1到第39，所以我们可以使用下面的测试来确定是否增加一个逗号：

```
$name .= "," unless $hash_ref->{suffix} =~ /^[IVX]+$/;
```

和名称放在一起的 format_banquet_entry( ) 的代码也是这个目录的 RTF 版本将需要的一些内容。然而，并不是复制 format_rtf_entry( ) 中的代码，让我们将它填入函数中：

```
sub format_name
{
my ($entry_ref) = shift;
my ($name);

    $name = $entry_ref->{first_name} . " " . $entry_ref->{last_name};
    if ($entry_ref->{suffix})           # there is a name suffix
    {
        # no comma for suffixes of I, II, III, etc.
        $name .= "," unless $entry_ref->{suffix} =~ /^[IVX]+$/;
        $name .= " " . $entry_ref->{suffix} if $entry_ref->{suffix};
    }
    return ($name);
}
```

将确定名称的字符串放在 format_name( ) 函数中，将把 format_banquet_entry( ) 函数减少到几乎没有：

```
sub format_banquet_entry
{
    printf "%s\n", format_name ($_[0]);
}
```

2. 生成显示格式的目录

生成这个目录的 RTF 版本比生成宴会程序成员列表更要棘手一些。首先，我们需要从每个项目中显示更多的信息。其次，我们需要用每个项目产生一些 RTF 控制语言来完成我们想要的作用。RTF 文档的最小框架是这样的：

```
{\rtf0
{\fonttbl {\f0 Times;}}
\plain \f0 \fs24
```

```
    ...document content goes here...
}
```

这个文档用花括号‘ { ’和‘ } ’作为开始和结束。RTF 关键字用反斜线符号开始，并且文档的第一个关键字必须为 \rtfn，n为这个文档对应的 RTF 规定的版本号。如果按我们的目的，0就比较合适。

在这个文档的内部，我们指定字体表来说明这些项目所使用的字体。字体表信息列在组中，由含有前导的 \fonttbl 关键字和一些字体信息的花括号组成。在框架中说明的这个字体表把字体号 0 定义为Times（我们只需要一个字体，但是如果想显示得更好一些，可以使用多种字体）。

下面的一些指示设置了缺省格式风格：\plain 选择无格式的格式，\f0 选择字体0（我们已经在字体表中定义为 Times ），\fs24 设置字体大小为12个点阵（\fs 后面的数量表示半个点阵的大小）。设置页边空白并不是必需的；大多数的字处理程序将提供合理的缺省值。

要想得到一个非常简单的方法，可以将每个项目显示为一系列的行，每行上都有一个标号。如果对应于特定输出行的信息缺失，则忽略这个行（例如，没有电子邮件地址的成员没有显示“ Email: ”行）。一些行（如“ Address: ”行）由多个列（街道、城市、州、邮政编码）中的信息构成，所以这个脚本必须能够处理缺失值的各种组合。这里是我们将使用的输出格式的样例：

```
Name: Mike Artel
Address: 4264 Lovering Rd., Miami, FL 12777
Telephone: 075-961-0712
Email: mike_artel@venus.org
Interests: Civil Rights,Education,Revolutionary War
```

对于显示的格式化项目， RTF 的表示方法如下所示：

```
\b Name: Mike Artel\b0\par
Address: 4264 Lovering Rd., Miami, FL 12777\par
Telephone: 075-961-0712\par
Email: mike_artel@venus.org\par
Interests: Civil Rights,Education,Revolutionary War\par
```

要想使“ Name: ”行为粗体，则在它的前面加 \b （后面有个空格）来打开粗体，并用 \b0 来关闭粗体。每行在末端都有一个段标记符（ \par）来告诉字处理程序移到下一行——没有太复杂的事情。

初始化函数产生前导 RTF 控制语言（请注意，两个反斜线符号获得输出中的一个反斜线符号）：

```
sub rtf_init
{
    print "{\\rtf0\n";
    print "{\\fonttbl {\\f0 Times;}}\n";
    print "\\plain \\f0 \\fs24\n";
}
```

类似地，清除函数产生终止控制语言（并不太多！）：

```
sub rtf_cleanup
{
    print "}\n";
}
```

真正的工作与格式化这个项目有关，即使这个任务相对简单。主要复杂点是将地址字符

串格式化，并确定应该显示哪个输出行：

```
sub format_rtf_entry
{
my ($entry_ref) = shift;
my ($address);

    printf "\\b Name: %s\\b0\\par\n", format_name ($entry_ref);
    $address = "";
    $address .= $entry_ref->{street} if $entry_ref->{street};
    $address .= ", " . $entry_ref->{city} if $entry_ref->{city};
    $address .= ", " . $entry_ref->{state} if $entry_ref->{state};
    $address .= " " . $entry_ref->{zip} if $entry_ref->{zip};
    print "Address: $address\\par\n" if $address;
    print "Telephone: $entry_ref->{phone}\\par\n" if $entry_ref->{phone};
    print "Email: $entry_ref->{email}\\par\n" if $entry_ref->{email};
    print "Interests: $entry_ref->{interests}\\par\n" if $entry_ref->{interests};
    print "\\par\n";
}
```

当然，不用限于这种特殊的格式化风格。可以更改如何显示任何域的方法，所以通过简单地更改 format_rtf_entry( )，可以几乎任意地更改显示的目录。用它原始格式的目录（一个字处理文档），是多么不容易做的事情！

gen_dir 脚本现在完成了。通过运行以下这些命令，我们可以以任意一种输出格式生成这个目录：

```
% gen_dir banquet > names.txt
% gen_dir rtf > directory.rtf
```

在 Windows 中，我可以运行 gen_dir，则这些文件准备从基于 Windows 字处理程序的内部使用。在 UNIX 中，我可就以运行上面那些命令，然后将这些输出文件以邮件形式发给自己作为附件，以便可以从我的 Macintosh 中获取它们，并将它们加载到字处理程序中。我偶尔使用 mutt 邮寄程序，它允许使用 -a 选项从命令行指定附件。可以如下发送给自己一个具有这两个附加文件的消息：

```
% mutt -a names.txt -a directory.rtf paul@snake.net
```

其他邮寄程序可能也允许创建附件。或者，可以以其他意思传输这些文件，如 FTP。无论如何，在这些文件被放到想放的地方之后，读取这个名称列表，并将它粘贴到年度程序文档，或者在可识别 RTF 的任何字处理程序中读取 RTF 文件，这都是较容易的。DBI 使我们从 MySQL 中抽取想要的信息很容易，Perl 的文本处理能力使我们将这些信息放在指定的格式中很容易。MySQL 不提供信息输出的任何特殊方式，但没有关系，因为将 MySQL 的数据库处理能力集成到如 Perl 的语言中并不费力，而这些语言具有极好的文本处理能力。

7.3.2 发送成员资格更新通知

当作为字处理文档维护历史同盟目录时，确定需要通知哪个成员其成员资格应该更新，这是件耗费时间并且容易出现错误的事情。既然我们在数据库中有信息，那么让我们看看如何自动地处理更新通知。我们想标识需要经过电子邮件更新的成员，这样我们就不必通过电话或邮件与他们联系了。

我们需要做的事情就是确定哪个成员在某些天以内快到更新的时间了。这个的查询涉及一个相对简单的日期计算：

```
SELECT ... FROM member
WHERE expiration < DATE_ADD(CURRENT_DATE, INTERVAL cutoff DAY)
```

cutoff 表示我们同意的可允许误差的天数。这个查询选择在几天之内快到更新时间的成员项目。作为特殊情况，终止点值为 0，寻找终止日期已过的成员（也就是说，实际上已经终止了的那些成员）。

我们标识了限制通知的这些记录之后，我们对它们应该怎么办呢？一个选择是直接从同样的脚本中发送邮件，但是，首先审阅不发送任何消息的列表可能有用。由于这个原因，我们将使用一个两阶段的方法：

- 阶段1：运行脚本 need_renewal 来标识需要更新的成员。可检查这个列表，或者可以使用它作为将更新通知发送到第 2 阶段的输入。
- 阶段2：运行脚本 renewal_notify，它通过电子邮件向成员发送“请更新”的通知。这个脚本应该通知您不具有电子邮件地址的成员，以便可以用其他方式与他们联系。

在此任务的第一部分中，need_renewal 脚本必须标识哪个成员需要更新。它的操作如下所示：

```
# default cutoff is 30 days; reset if numeric argument given
my ($cutoff) = 30;
$cutoff = shift (@ARGV) if @ARGV && $ARGV[0] =~ /^\d+$/;

warn "Using cutoff of $cutoff days\n";

my ($sth) = $dbh->prepare (qq{
        SELECT
            member_id, email, last_name, first_name, expiration,
            TO_DAYS(expiration) - TO_DAYS(CURRENT_DATE) AS days
        FROM member
        WHERE expiration < DATE_ADD(CURRENT_DATE, INTERVAL ? DAY)
        ORDER BY expiration, last_name, first_name
});
$sth->execute ($cutoff);    # pass cutoff as placeholder value

while (my $hash_ref = $sth->fetchrow_hashref ())
{
    print join ("\t",
                $hash_ref->{member_id},
                $hash_ref->{email},
                $hash_ref->{last_name},
                $hash_ref->{first_name},
                $hash_ref->{expiration},
                $hash_ref->{days} . " days")
          . "\n";
}
$sth->finish ();
```

need_renewal 脚本的输出如下所示（因为是针对当前日期确定的结果，而您读这本书的时间和我书写它的时间将是不同的，所以将获得不同的输出）：

```
89  g.steve@pluto.com       Garner  Steve   1999-08-03  -32 days
18  york_mark@earth.com     York    Mark    1999-08-24  -11 days
82  john edwards@venus.org  Edwards John    1999-09-12  8 days
```

可以观察到，处于负数天数的那些成员资格需要更新。负数意味着我们已经过期了（当手工地维护记录时，就可能发生这种情况；有些人从缝隙中滑掉了。既然我们在数据库中有了这些信息，那么我们要寻找在前面丢失的几个人）！

更新通知任务的第二部分涉及了通过电子邮件发送通知的脚本 renewal_notify。要想使renewal_notify 更容易使用，则我们可以使它支持三类命令行参数：成员关系 ID 号码，电子邮件地址和文件名。数值的参数表示成员资格 ID 值，带有字符‘@’的参数表示电子邮件的地址。其他任何事情都解释为应该读取的文件名，以便找到他们的 ID 号码或电子邮件地址，可以直接在命令行中这样做，或者通过将它们在文件中列出来去做（特别是，可以使用need_renewal 的输出作为 renewal_notify 的输入）。

对于要发送通知的每个成员，此脚本查找相应的 member 表项目，抽取电子邮件地址，并向那个地址发送一条消息。如果此项中没有电子邮件地址，则 renewal_notify 生成一条消息，通知您需要以一些其他方式与这些成员联系。

要想发送电子邮件，renewal_notify 打开与 sendmail 程序的管道，并将这封邮件推入此管道中（在 Windows 下不能这样操作，Windows 中没有 sendmail。可能需要寻找发送邮件的模块来代替它使用）。在此脚本开头附近，将到 sendmail 的路径名设置为参数。可能需要更改该路径，因为 sendmail 的位置随系统的变化而变化：

```
# change path to match your system
my ($sendmail) = "/usr/lib/sendmail -t -oi";
```

主要参数处理循环的操作如下所示。如果在命令行没有指定参数，则我们读取标准的输出作为输入。否则，我们通过将参数传递给 interpret_argument( )，将它分类为 ID 号、电子邮件地址或者文件名来处理每个参数：

```
if (@ARGV == 0)     # no arguments, read STDIN for values
{
    read_file (\*STDIN);
}
else
{
    while (my $arg = shift (@ARGV))
    {
        # interpret argument, with filename recursion
        interpret_argument ($arg, 1);
    }
}
```

函数 read_file( ) 读取了文件的内容（假设已经打开），并查看每行的第一个域（如果我们将 need_renewal 的输出作为 renewal_notify 的输入，则每行都有若干域，但是我们只想查看第一个域）。

```
sub read_file
{
my ($fh) = shift;
my ($arg);

    while (defined ($arg = <$fh>))
    {
        # strip off everything past column 1, including newline
        $arg =~ s/\s.*//s;
        # interpret argument, without filename recursion
        interpret_argument ($arg, 0);
    }
}
```

interpret_argument( ) 函数将每个参数分类，以便确定它是 ID 号码、电子邮件地址还是文

件名。对于 ID 号码和电子邮件地址，它查找适当的成员项目，并将它传递给 notify_member( )。我们必须注意由电子邮件所指定的成员。两个成员具有同样的地址是可能的（例如，丈夫和妻子），并且我们不想将一条消息发送给不能用这条消息的人。为了避免这一点，我们查找了与电子邮件地址相对应的成员的 ID 号码，来确保内容的正确。如果此地址和一个以上的 ID 号码匹配，则它是不确定的，我们在显示一条警告消息后忽略它。

如果参数看起来不像 ID号码或电子邮件地址，则将它作为文件名读取为进一步的输入。在这里，我们也必须小心——为了避免无穷循环的可能性，如果我们已经读取一个文件，则我们不想再读取文件：

```
sub interpret_argument
{
my ($arg, $recurse) = @_;
my ($query, $ary_ref);

    if ($arg =~ /^\d+$/)        # numeric membership ID
    {
        notify_member ($arg);
    }
    elsif ($arg =~ /@/)         # email address
    {
        # get member_id associated with address
        # (there should be exactly one)
        $query = qq{ SELECT member_id FROM member WHERE email = ? };
        $ary_ref = $dbh->selectcol_arrayref ($query, undef, $arg);
        if (scalar (@{$ary_ref}) == 0)
        {
            warn "Email address $arg matches no entry: ignored\n";
        }
        elsif (scalar (@{$ary_ref}) > 1)
        {
            warn "Email address $arg matches multiple entries: ignored\n";
        }
        else
        {
            notify_member ($ary_ref->[0]);
        }
    }
    else                        # filename
    {
        if (!$recurse)
        {
            warn "filename $arg inside file: ignored\n";
        }
        else
        {
            open (IN, $arg) or die "Cannot open $arg: $!\n";
            read_file (\*IN);
            close (IN);
        }
    }
}
```

实际上，发送更新通知的 notify_member( ) 函数的代码如下所示。如果得出这个成员没有电子邮件地址，则什么也不做，但是 notify_member( ) 显示一条警告消息，以便知道需要以其他某种方式与该成员联系。可以调用具有这条消息中所显示的这个成员资格 ID 号码的 show_member，来查看全部项目——例如，找出这个成员的电话号码和通信地址。

```
# notify member that membership will soon be in arrears

sub notify_member
{
my ($member_id) = shift;
my ($query, $sth, $entry_ref, @col_name);

    warn "Notifying $member_id...\n";
    $query = qq{ SELECT * FROM member WHERE member_id = ? };
    $sth = $dbh->prepare ($query);
    $sth->execute ($member_id);
    @col_name = @{$sth->{NAME}};
    $entry_ref = $sth->fetchrow_hashref ();
    $sth->finish ();
    if (!$entry_ref)        # no member found!
    {
        warn "NO ENTRY found for member $member_id!\n";
        return;
    }
    open (OUT, "| $sendmail") or die "Cannot open mailer\n";
    print OUT <<EOF;
To: $entry_ref->{email}
Subject: Your USHL membership is in need of renewal
 Greetings.  Your membership in the US Historical League is
 due to expire soon.  We hope that you'll take a few minutes to
 contact the League office to renew your membership.  The
 contents of your member entry are shown below.  Please note
 particularly the expiration date.

 Thank you.

 EOF
    foreach my $col_name (@col_name)
    {
        printf OUT "%s: %s\n", $col_name, $entry_ref->{$col_name};
    }
    close (OUT);
}
```

用它可能获得更好的内容——例如，通过向 member 表中增加一列来记录最近更新的提示是何时发送出去的。这样做将有助于避免过于频繁地发送通知。实际上，我们只需假设不存在大约每月运行一次以上的程序。

现在运行这两个脚本，从而可以这样使用它们：

```
% need_renewal > junk
% （看一看 junk，检查它是否合理）
% renewal_notify junk
```

要想通知单个的成员，可以通过 ID 号码或电子邮件地址指定它们：

```
% need_renewal 18 g.steve@pluto.com
```

7.3.3 历史同盟成员项目编辑

我们开始发送更新通知之后，假设我们通知的一些人将更新他们的成员资格是个安全的措施。当这种情况发生时，我们将需要一种更新其所具有的新的终止日期项的方法。下一章中，我们将开发一种方法，在 Web 浏览器上编辑成员记录，但是在这里，我们将建立一个命令行脚本 edit_member，允许用提示项的各部分新值的方法来更新项目。其操作如下：

- 如果在命令行上无参数调用，则 edit_member 假设您想输入一个新的号码，提示放在成员项目中的初始信息，并创建新的项目。

■ 如果在命令行上调用时带有成员 ID 号码，则edit_member 查找这个项目的已有内容，然后提示更新每一列。如果输入一列的值，则其替换当前的值。如果按 Enter 键，这列并不更改（如果不知道成员的 ID 号码，可以运行 show_member last_name 来查找其内容）。

如果只想更新成员的终止日期，则允许编辑全部项目的这种方式可能是不必要的过度行动。另一方面，类似这样的脚本也提供了一种简单的通用目的方式，来更新一个项目的任何部分而不必了解 SQL 的任何知识（一种特殊的情况为 edit_member 不允许更改 member_id 域，因为当创建一个项目时，自动地分配这个域，并且在以后不能更改）。

edit_member 需要了解的第一件事为 member 表中这些列的名称：

```
# get member table column names
my ($sth) = $dbh->prepare (qq{ SELECT * FROM member WHERE 1 = 0 });
$sth->execute ();
my (@col_name) = @{$sth->{NAME}};
$sth->finish ();
```

然后我们可以输入主体循环：

```
if (@ARGV == 0) # if no arguments, create new entry
{
        new_member (\@col_name);
}
else            # otherwise edit entries using arguments as member IDs
{
my (@id);

    # save @ARGV, then empty it so that reads from STDIN
    # don't use the arguments as filenames
    @id = @ARGV;
    @ARGV = ();
    # for each ID value, look up the entry, then edit it
    while (my $id = shift (@id))
    {
    my ($entry_ref);

        $sth = $dbh->prepare (qq{
                    SELECT * FROM member WHERE member_id = ?
                });
        $sth->execute ($id);
        $entry_ref = $sth->fetchrow_hashref ();
        $sth->finish ();
        if (!$entry_ref)
        {
            warn "No member with member ID = $id\n";
            next;
        }
        edit_member (\@col_name, $entry_ref);
    }
}
```

创建新成员项目的代码如下所示。它请求每个 member 表列，然后发布一条 INSERT 语句以增加一条新记录：

```
# create new member entry

sub new_member
{
my ($col_name_ref) = shift;
my ($entry_ref);
my ($col_val, $query, $delim);
```

```
    return unless prompt ("Create new entry? ") =~ /^y/i;
    # prompt for new values; user types in new value,
    # "null" to enter a NULL value, "exit" to exit
    # early.
    foreach my $col_name (@{$col_name_ref})
    {
        next if $col_name eq "member_id";   # skip key field
        $col_val = col_prompt ($col_name, "", 0);
        next if $col_val eq "";             # user pressed Enter
        return if $col_val =~ /^exit$/i;    # early exit
        $col_val = undef if $col_val =~ /^null$/i;
        $entry_ref->{$col_name} = $col_val;
    }
    # show values, ask for confirmation before inserting
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nInsert this entry? ") =~ /^y/i;

    # construct an INSERT query, then issue it.
    $query = "INSERT INTO member";
    $delim = " SET "; # put "SET" before first column, "," before others
    foreach my $col_name (@{$col_name_ref})
    {
        # only specify values for columns that were given one
        next if !defined ($entry_ref->{$col_name});
        # quote() quotes undef as the word NULL (without quotes),
        # which is what we want.
        $query .= sprintf ("%s %s=%s", $delim, $col_name,
                    $dbh->quote ($entry_ref->{$col_name}));
        $delim = ",";
    }
    warn "Warning: entry not inserted?\n"
        unless $dbh->do ($query) == 1;
}
```

new_member( )所用的提示例程如下所示：

```
# ask a question, prompt for an answer

sub prompt
{
my ($str) = shift;

    print STDERR $str;
    chomp ($str = <STDIN>);
    return ($str);
}

# prompt for a column value; show current value in prompt if
# $show_current is true

sub col_prompt
{
my ($name, $val, $show_current) = @_;
my ($prompt, $str);

    $prompt = $name;
    $prompt .= " [$val]" if $show_current;
    $prompt .= ": ";
    print STDERR $prompt;
    chomp ($str = <STDIN>);
    return ($str ? $str : $val);
}
```

col_prompt( ) 带有 $show_current 参数的原因是，当这个脚本用于更新项目时，我们也对已有成员项目请求的列值使用这个函数。当创建新的项目时， $show_current 将为 0，因为当

前没有值可以显示。在编辑一个已有项目时，它将为非零。后一种情况中的提示将显示当前的值，用户可以简单地通过按 Enter 键来接受。

编辑已有成员的代码类似于创建新成员的代码。然而，我们有一个可操作的项目，所以提示例程显示当前项目的值，并且edit_member( ) 函数发布一条 UPDATE 语句，而不是 INSERT语句：

```
# edit existing contents of an entry

sub edit_member
{
my ($col_name_ref, $entry_ref) = @_;
my ($col_val, $query, $delim);

    # show initial values, ask for okay to go ahead and edit
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nEdit this entry? ") =~ /^y/i;
    # prompt for new values; user types in new value to replace
    # existing value, "null" to enter a NULL value, "exit" to exit
    # early, or Enter to accept existing value.
    foreach my $col_name (@{$col_name_ref})
    {
        next if $col_name eq "member_id";   # skip key field
        $col_val = $entry_ref->{$col_name};
        $col_val = "NULL" unless defined ($col_val);
        $col_val = col_prompt ($col_name, $col_val, 1);
      return if $col_val =~ /^exit$/i;    # early exit
      $col_val = undef if $col_val =~ /^null$/i;
      $entry_ref->{$col_name} = $col_val;
    }
    # show new values, ask for confirmation before updating
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nUpdate this entry? ") =~ /^y/i;

    # construct an UPDATE query, then issue it.
    $query = "UPDATE member";
    $delim = " SET "; # put "SET" before first column, "," before others
    foreach my $col_name (@{$col_name_ref})
    {
        next if $col_name eq "member_id";   # skip key field
        # quote() quotes undef as the word NULL (without quotes),
        # which is what we want.
        $query .= sprintf ("%s %s=%s", $delim, $col_name,
                           $dbh->quote ($entry_ref->{$col_name}));
        $delim = ",";
    }
    $query .= " WHERE member_id = ?";
    warn "Warning: entry not updated?\n"
        unless $dbh->do ($query, undef, $entry_ref->{member_id}) == 1;
}
```

edit_member 的问题为它不进行任何输入值校验。对于 member 表中的大多数域，都没有什么校验——它们只是字符串域。但是对于 expiration 列，实际上应该检查输入值，以便确保它们看起来像日期。在一般目标的数据输入应用程序中，可能想抽取有关表的信息，以便确定它的所有列的类型。然后，可能按照那些类型上的约束条件来校验。那就比我在这里想探求的内容涉及得更多，所以我只在 col_prompt( ) 函数中增加一个快速方法，以便如果列名为“ expiration ”，则检查输入的格式。最低限度的日期值检查可以这样来做：

```
sub col_prompt
{
my ($name, $val, $show_current) = @_;
my ($prompt, $str);
```

```
loop:
    $prompt = $name;
    $prompt .= " [$val]" if $show_current;
    $prompt .= ": ";
    print STDERR $prompt;
    chomp ($str = <STDIN>);
    # perform rudimentary check on the expiration date
    if ($str && $name eq "expiration")  # check expiration date format
    {
        $str =~ /^\d+[^\d]\d+[^\d]\d+$/
            or goto loop;
    }
    return ($str ? $str : $val);
}
```

这个模板测试了非数字字符分隔的三个序列的数字。这只是检查的一部分，因为它没有侦测如“ 1999-14-22 ”的值为无效。要想使脚本更好，则应该给它更严格的日期检查以及其他检查，如需要名和姓的域，就应该给非空值。

一些其他的改进可能是，如果没有更改列，则跳过这个更新，当用户正在编辑它时，如果其他一些人已经更改了这条记录，则通知这个用户。可以通过保存成员项目列的原始数据来做到这一点，然后，编写 UPDATE 语句来只更新那些已经更改的列。如果没有，则甚至不需要发布这条语句。同样，对于每个原始列值，可以编写 WHERE 子句来包括 AND col_name = col_val。如果其他一些人已经更改了这条记录，则这可能导致 UPDATE 失败，此时它的反馈为，两个人要同时更改这个项目。

7.3.4 寻找共同兴趣的历史同盟成员

历史同盟秘书的责任之一就是处理成员的请求，这些成员可能要求对美国历史领域内特殊时期或特殊人物（如在大萧条中或者亚伯拉罕·林肯的生命）感兴趣的其他人清单。当在字处理程序文档中维护这个目录时，使用字处理程序的“ Find ”功能，可以非常容易地找到这样的成员。然而，产生一列只含有合格成员的项就要困难一些，因为它涉及大量的拷贝和粘贴。使用 MySQL，工作就变得容易得多，因为我们可以只运行如下这样的查询：

```
SELECT * FROM member WHERE interests LIKE "%lincoln%"
ORDER BY last_name, first_name
```

不幸的是，如果在 mysql 客户机程序运行这个查询，则结果看上去并不是非常好。让我们把少量的 DBI 脚本和生成较漂亮的输出的 interests 放在一起。首先，检查一下脚本，确保在命令行至少有一个命名的参数，因为如果没有一个命名的参数就没有内容可以搜索。然后，对于每个参数，脚本在 member 表的 interests 列上运行一个查询：

```
@ARGV or die "Usage: interests keyword\n";
search_members (shift (@ARGV)) while @ARGV;
```

为了搜索关键字字符串，我们在每一边都放了通配符‘ % ’，以便可以在 interests 列的任何地方都可以找到这个字符串。然后，我们显示相匹配的项：

```
sub search_members
{
my ($interest) = shift;
my ($sth, $count);
print "Search results for keyword: $interest\n\n";
$sth = $dbh->prepare (qq{
            SELECT * FROM member WHERE interests LIKE ?
            ORDER BY last_name, first_name
        });
# look for string anywhere in interest field
```

```
    $sth->execute ("%" . $interest . "%");
    $count = 0;
    while (my $hash_ref = $sth->fetchrow_hashref ())
    {
        format_entry ($hash_ref);
        ++$count;
    }
    print "$count entries found\n\n";
}
```

这里没有出现 format_entry( ) 函数。它与 gen_dir 脚本的函数 format_rtf_entry( ) 在本质上是相同的，但 format_entry( ) 函数去掉了 RTF 控制字。

7.3.5 联机历史同盟目录

在7.4节中，我们将开始编写连接到 MySQL 服务器并抽取信息的脚本，还要编写以 Web 页面形式在客户机的 Web 浏览器中出现的信息。那些脚本按照客户机请求动态地生成了 HTML。在我们到达那一点之前，让我们通过编写生成能装载到 Web 服务器文档树中的静态 HTML 文档的 DBI 代码，开始考虑有关的 HTML。以 HTML 格式创建的历史同盟目录是最好的选择，因为我们的目标之一就是无论如何要使目录联机。

一般来说，HTML 文档有点像下面这样的结构：

```
<HTML>                          ← beginning of document
<HEAD>                          ← beginning of document head
<TITLE>My Page Title</TITLE>    ← title of document
</HEAD>                         ← beginning of document head
<BODY>                          ← beginning of document body
<H1>My Level 1 Heading</H1>     ← a level 1 heading

... content of document body ...

</BODY>                         ← end of document body
</HTML>                         ← end of document
```

为了以这种格式生成目录，编写完整的脚本对于你来讲并不必要。回想一下，当我们编写 gen_dir 脚本时，我们使用了可扩展的框架，因此，为了以其他格式产生目录而插入了代码。这意味着假如代码生成了 HTML 输出，我们则需要编写文档初始化和清除的函数，和格式化单独项一样。然后我们需要创建转换盒元素来指向这些函数。

只显示出的联机文档非常容易地分解为可以由初始化函数和清除函数处理的序言和收尾部分，以及由项目格式化函数生成的中间部分。HTML 初始化函数生成级别1标题的每一部分，而清除函数生成关闭 </BODY> 和 </HTML> 标记的部分：

```
sub html_init
{
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>US Historical League Member Directory</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
    print "<H1>US Historical League Member Directory</H1>\n";
}

sub html_cleanup
{
    print "</BODY>\n";
    print "</HTML>\n";
}
```

一般来说，真正的工作在于格式化项目。但即使这样也不太困难。我们可以拷贝 format_rtf_entry( ) 函数，确保项目中的任何特殊字符都被编码，并且用 HTML 标出的标志替换 RTF 控制字：

```
sub format_html_entry
{
my ($entry_ref) = shift;
my ($address);

    # encode characters that are special in HTML
    foreach my $key (keys (%{$entry_ref}))
    {
        $entry_ref->{$key} =~ s/&/&amp;/g;
        $entry_ref->{$key} =~ s/\"/&quot;/g;
        $entry_ref->{$key} =~ s/>/&gt;/g;
        $entry_ref->{$key} =~ s/</&lt;/g;
    }
    printf "<STRONG>Name: %s</STRONG><BR>\n", format_name ($entry_ref);
    $address = "";
    $address .= $entry_ref->{street} if $entry_ref->{street};
    $address .= ", " . $entry_ref->{city} if $entry_ref->{city};
    $address .= ", " . $entry_ref->{state} if $entry_ref->{state};
    $address .= " " . $entry_ref->{zip} if $entry_ref->{zip};
    print "Address: $address<BR>\n" if $address;
    print "Telephone: $entry_ref->{phone}<BR>\n" if $entry_ref->{phone};
    print "Email: $entry_ref->{email}<BR>\n" if $entry_ref->{email};
    print "Interests: $entry_ref->{interests}<BR>\n"
                            if $entry_ref->{interests};
       print "<BR>\n";
   }
```

现在我们把另一个元素加到转换盒中，指出编写 HTML 的函数，并且完成对 gen_dir的更正：

```
# switchbox containing formatting functions for each output format
my (%switchbox) =
(
    "banquet" =>                        # functions for banquet list
    {
        "init"      => undef,           # no initialization needed
        "entry"     => \&format_banquet_entry,
        "cleanup"   => undef            # no cleanup needed
    },
    "rtf" =>                            # functions for RTF format
    {
        "init"      => \&rtf_init,
        "entry"     => \&format_rtf_entry,
        "cleanup"   => \&rtf_cleanup
    },
    "html" =>                           # functions for HTML format
    {
        "init"      => \&html_init,
        "entry"     => \&format_html_entry,
        "cleanup"   => \&html_cleanup
    }
);
```

为了产生 HTML 格式的目录，运行下面的命令并在 Web 服务器的文档树中安装结果输出文件：

```
% gen_dir html > directory.html
```

当更新目录时，可以再次运行命令来更新联机版本。另一个方案是建立周期性执行的 cron 作业。那就是说，联机目录将被自动地更新。例如，我可能使用类似于这个的 crontab 项在每天早晨4点运行 gen_dir：

```
0 4 * * * /u/paul/samp_db/gen_dir > /usr/local/apache/htdocs/directory.html
```

这个 cron 作业所运行的用户必须允许它们都执行位于 samp_db 目录中的脚本，并将文件编写到 Web 服务器的文档树中。

7.4 在 Web 应用程序中使用 DBI

迄今为止，我们编写的 DBI 脚本用于命令行环境中的命令解释程序，但 DBI 在其他环境下也是有用的，例如在基于 Web 的应用程序的开发中。当编写能从 Web 浏览器调用的 DBI 脚本时，就打开了新鲜而有趣的与数据库交互的性能。

例如，如果以表格的形式显示数据，则可以很容易地把每个列标题转换为可以选择的连接，以便将该列的数据重新排序。它允许单击一次就可以以不同的方式查看数据，而又不必键入任何查询。或者可以提供一种用户可以为数据库搜索而键入的标准格式，然后，显示含有搜索结果的页面。像这种简单的能力能够特别地改变为访问数据库内容而提供的交互性的水平。除此之外，Web 浏览器的显示能力比在终端窗口获得的能力要明显地更好一些，所以，输出也经常看起来更漂亮。

在这部分，我们将创建下面的基于 Web 的脚本：

- samp_db **数据库中表的通用浏览器。**这与我们想对这个数据库完成的任何特定的任务无关，但是它举例说明了若干 Web 程序设计概念，并提供了一种查看这些表所含有的信息的方便方式。
- **允许我们查看任何给定的测验或测试分数的分数浏览器。**它作为回顾评分事件结果的快速方式是很方便的，并且当我们需要创建测试的等级曲线时，它是有用的，所以我们可以以字母等级来标记试卷。
- **寻找分享共同兴趣的历史同盟成员的脚本。**通过允许用户输入搜索短语来完成它，然后在 member 表的 interests 域来搜索短语。我们已经编写了一个行命令脚本来做这些，但是，基于 Web 的版本提供了有指导意义的参考观点，允许对同一任务比较两种方法。

我们将使用 CGI.pm Perl 模块来编写这些脚本，这个模块是将 DBI 脚本连接到 Web 上最容易的方法（有关获得 CGI.pm 模块的说明，请参阅附录 A）。之所以称为 CGI.pm，是因为它有助于编写使用公共网关协议的脚本，这个协议定义了 Web 服务器如何与其他程序通信。CGI.pm 处理涉及了许多通用内务处理的任务细节，如收集通过 Web 服务器传递到脚本的作为输出的参数值。CGI.pm 也提供了生成 HTML 输出的便利方法，与编写自己原始的 HTML 标记相比，它减少了编写难看的 HTML 的机会。

在本章中，您将学到足够有关 CGI.pm 的知识来编写自己的 Web 应用程序，但是，当然不是它所包括的所有性能。要想学习有关这个模块的更多知识，请参阅 Lincoln Stein (John Wiley 1998 出版) 撰写的《Official Guide to Programming with CGI.pm》，或在以下网址查阅联机文档：

http://stein.cshl.org/www/software/CGI/

7.4.1 设置 CGI 脚本的 Apache

除了 DBI 和 CGI.pm 之外，编写基于 Web 的脚本还需要有一个以上的组件：Web 服务器。这里的说明适合Apache 服务器使用脚本，但是，如果愿意，稍微改编一点这些说明，就可以使用不同的服务器。

一般来说，Apache 装置的各个部分位于 /usr/local/apache 目录。对我们的目的来讲，这个目录中最重要的子目录为 htdocs（HTML 文档树）、cgi-bin （可执行的脚本和Web服务器调用的程序），和 conf（配置文件）。这些目录也可能放在系统中的其他地方。如果是这样，则要对下面的注意事项做适当的调整。

应该验证 cgi-bin 目录不在 Apache 文档树的内部，以便它内部的这些脚本不能作为无格式文本来请求。这是个安全的防范方法。您也不愿意让怀有恶意的客户机程序检查您的脚本，通过提取这些脚本的文本并研究它们来作为安全的突破口。

要想安装以 Apache 方式使用的 CGI 脚本，则将它放在 cgi-bin 目录下，然后将这个脚本的所有权更改为运行 Apache 的用户，并将它的模式更改为对该用户为可执行的和只读的模式。例如，如果 Apache 以名称为 www 的用户方式运行，则使用下面的命令：

```
% chown www script_name
% chmod 500 script_name
```

可能需要用 www 或 root 运行这些命令。如果不允许在 cgi-bin 目录下安装脚本，则可以请求系统管理员代表您来这样做。

安装这个脚本之后，通过向 Web 服务器发送适当的 URL，可以请求浏览器上的这个脚本。典型的 URL 是这样的：

http://your.host.*name*/cgi-bin/script_*name*

从 Web 浏览器请求脚本会导致 Web 服务器执行它。返回脚本的输出，结果作为 Web 页面出现在浏览器中。

如果为寻求更好的性能而使用具有 mod_perl 的 CGI 脚本，则可以这样做：

1) 确保至少有以下版本的必需软件：Perl 5.004、CGI.pm 2.36和mod_perl 1.07。

2) 确保将 mod_perl 编译为 Apache 可执行的文件。

3) 建立一个存储脚本的目录。我使用了 /usr/local/apache/cgi-perl。cgi-bin 不应该位于Apache文档树的内部，出于同样的安全原因，cgi-perl目录也不应该在那里。

4) 告知 Apache，与位于 cgi-perl 目录中的脚本 mod_perl 相关联：

```
Alias /cgi-perl/ /usr/local/apache/cgi-perl

<Location /cgi-perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
</Location>
```

如果正在使用 Apache 的当前版本，这个版本使用单个的配置文件，则将所有这些指示放在 httpd.conf 中。如果 Apache 的版本使用三个旧文件的方法来配置信息，则将 Alias指示放入 srm.conf 中，将Location 行放入 access.conf 中。对于 cgi-perl 目录，不要启用 mod_perl、PerlSendHeader 或 PerlSetupEnv 指示。这些由 CGI.pm 自动地处理，启用它们可能导致处理冲突。

mod_perl 脚本的 URL 与标准的 CGI 脚本的 URL 相类似。唯一的不同之处在于指定 cgi-perl 而不是 cgi-bin。

```
http://your.host.name/cgi-perl/script_name
```

有关的详细信息，请参阅下面地址的 Apache Web 站点的 mod_perl 区域：

```
http://perl.apache.org/
```

7.4.2 CGI.pm 的简要介绍

为了编写使用 CGI.pm 模块的 Perl 脚本，将 use 行放在这个脚本的开头附近，然后创建让您访问 CGI.pm 方法和变量的 CGI 对象：

```
use CGI;
my ($cgi) = new CGI;
```

我们的 CGI 脚本使用了 CGI.pm 的性能，它通过使用 $cgi 变量调用方法来实现。例如，为了生成级别1标题，我们将这样使用 h1( ) 方法：

```
print $cgi->h1 ("My Heading");
```

CGI.pm 也支持允许以函数调用它的方法的使用风格，而不用前导的 ‘ $cgi-> ’。在这里，我没有使用这个语法，是因为 ‘ $cgi-> ’ 符号更类似于使用 DBI 的方式，还因为它防止 CGI.pm函数名与可以定义的任何函数名产生冲突。

1. 检查输入参数，并编写输出

CGI.pm 所做的事情之一就是照看所有丑陋的细节，这些细节涉及到收集由 Web服务器向脚本提供的输入信息。为了获得那些信息，所需做的就是调用 param( ) 方法。可以如下获得所有可用的参数名：

```
my (@param) = $cgi->param ();
```

为了检索特定参数的值，只命名感兴趣的参数：

```
if (!$cgi->param ("my_param"))
{
    print "my_param is not set\n";
}
else
{
    printf "my_param value: %s\n", $cgi->param ("my_param");
}
```

CGI.pm还提供生成传送给客户机浏览器的输出方法。考虑下面的 HTML文档：

```
<HTML>
<HEAD>
<TITLE>My Page Title</TITLE>
</HEAD>
<BODY>
<H1>My Level-1 Heading</H1>
<P>Paragraph 1.
<P>Paragraph 2.
</BODY>
</HTML>
```

这个代码使用 $cgi 来产生等价的文档：

```
print $cgi->header ();
print $cgi->start_html (-title => "My Page Title");
```

```
print $cgi->h1 ("My Page Heading");
print $cgi->p ();
print "Paragraph 1.\n";
print $cgi->p ();
print "Paragraph 2.\n";
print $cgi->end_html ();
```

使用 CGI.pm 生成输出，而不是编写自己原始的 HTML，这样做的一些优点是，可以按逻辑单元考虑，而不是按单独的组成标识来考虑，而且 HTML 不太可能含有错误（我说“不太可能”的原因是 CGI.pm 不禁止做古怪的事情，如含有一列内部的标题）。除此之外，对于编写的非标记文本，CGI.pm 提供自动的字符转义，如 HTML 中指定的‘ < ’和‘ > ’。

如果愿意，CGI.pm 生成输出方法的使用并不排斥编写自己原始的 HTML。可以将这两种方法混合起来，组合调用具有生成文字标识的显示语句的 CGI.pm 方法。

2. 转义的 HTML 和 URL 文本

如果经 CGI.pm 方法，如 start_html( ) 或 h1( )，编写非标记的文本，则自动地转义文本中的特定字符。例如，如果使用下面的语句生成标题，则标题文本中的‘ & ’字符将由CGI.pm转换为‘ & ’：

```
print $cgi->start_html (-title => "A, B & C");
```

如果不使用 CGI.pm 生成输出的方法编写非标记的文本，则可能应该先让它经过 escapeHTML( )，以便确保可以正确地转义任何指定的字符。当构造可能含有特定字符的 URL 时也是这样，尽管在那种情况下应该使用 escape( ) 方法来代替它。使用适当的编码方法是很重要的，因为每种方法都将不同的字符集作为特殊的字符来对待，并使用彼此不同的格式来对待特殊的字符编码。考虑下面简短的 Perl 脚本：

```
#! /usr/bin/perl
use CGI;
$cgi = new CGI;

$s = "x<=y, right?";
print $cgi->escapeHTML ($s) . "\n"; # encode for use as HTML text
print $cgi->escape ($s) . "\n";     # encode for use in a URL
```

如果运行这个脚本，则它生成下面的输出，从这里可以看到 HTML 文本的编码不同于 URL 的编码：

```
x&lt;=y, right
x%3C%3Dy%2C%20right%3F
```

3. 编写多目的页面

编写基于 Web 的脚本来生成 HTML，而不是编写静态的 HTML 文档的主要原因之一是，根据调用方式，脚本可以产生不同类型的页面。我们将要编写的所有 CGI 脚本都有这种特性。每一个都像下面这样操作：

1) 当从浏览器第一次请求这个脚本时，它生成一个初始页面，允许选择想要的信息类型。

2) 当做了选择以后，重新调用这个脚本，但是，这次它在第二页检索，并显示请求的特定信息。

这里的主要问题是想从第一页的选择中确定第二页的内容，但是，通常 Web 页面是彼此独立的，除非安排某些特定排列的次序。这个窍门是让脚本生成页面，这个页面给参数设置一个值，告诉这个脚本的下一个调用想要的内容。当第一次调用这个脚本时，这个参数没有值；告诉这个脚本给出它的初始页面。当指出想看的信息内容时，这个页面再一次调用这个

脚本，但是，将参数设为指示这个脚本做什么的一个值。

将说明从页面传送回脚本有不同的方式。一种方式是提供一种用户填写的表格。当用户提交这张表格时，将它的内容提交给 Web 服务器。服务器将信息传递给脚本，这个脚本通过调用 param( ) 方法，能够找出提交的内容。这就是我们对第三个 CGI 脚本所做的事情（允许用户输入搜索历史同盟目录的关键字）。

对脚本指定说明的另外一种方法是，当请求脚本时，将信息作为发送到 Web服务器的URL的一部分来传递。这就是我们对于 samp_db 表浏览器和分数浏览器脚本要做的事情。这种工作方式是脚本生成含有超链接的页面。选择一个连接，再次调用这个脚本，但是，这次指定参数值，这个参数值指示这个脚本做什么。实际上，这个脚本以不同的方式调用它本身，来提供不同类型的结果，这取决于用户所选择的连接。

脚本可以允许通过向浏览器向它自己的 URL 传送一个含有超链接的页面来调用它本身。例如，脚本 my_script 可以编写含有如下这样连接的页面：

```
<A HREF="/cgi-bin/my_script">Click Me!</A>
```

当用户敲入文本“ Click Me! ”时，用户浏览器就请求将 my_script 发送回 Web 服务器。当然，所有这些会导致脚本再次发送出同一个页面，因为它不支持其他信息。然而，如果将一个参数附加到 URL 上，则当用户选择这个连接时，将这个参数送回 Web 浏览器。服务器调用这个脚本，这个脚本可以调用 param ( ) 来侦测设置的参数，并根据它的值采取行动。

为了把参数附到 URL 的末尾，加一个“ ? ”字符放到名称/值的前面。为了附上多个参数，用字符“ & ”分隔。例如：

```
/cgi-bin/my_script?name=value
/cgi-bin/my_script?name=value&name2=value2
```

为了构造带有附加参数的自引用的 URL，CGI脚本应该通过调用script_name ( ) 方法获得自己的URL来开始，然后像按照如下方法添加参数：

```
$url = $cgi->script_name ();    # get URL for script
$url .= "?name=value";          # add first parameter
$url .= "&name2=value2";        # add second parameter
```

在构造URL之后，通过使用 CGI.pm 的 a( ) 方法，可以生成一个包括它的超链接 <A> 标记：

```
print $cgi->a ({-href => $url}, "Click Me!");
```

通过检查一个简短的 CGI 脚本来查看如何工作会更容易。第一次调用时，下面的脚本 flip_flop，给出了一个含有单个超链接的称为页面 A 的页面。选择这个连接再次调用这个脚本，但是设置 page 参数，告诉它显示页面 B。页面 B也包括对脚本的连接，但是 page 参数没有值。因此，在页面 B中选择这个连接导致重新显示原始页面。随后的脚本调用将页面在脚本 A和脚本 B之间来回切换：

```
use CGI;

my ($cgi) = new CGI;
my ($url) = $cgi->script_name ();   # this script's own URL

print $cgi->header ();
if ($cgi->param ("page") ne "b")    # display page A
{
    print $cgi->start_html (-title => "Flip-Flop: Page A");
```

```
    print "This is Page A.<BR>To select Page B, ";
    $url .= "?page=b";      # attach parameter to select page B
    print $cgi->a ({-href => $url}, "click here");
}
else                                # display page B
{
    print $cgi->start_html (-title => "Flip-Flop: Page B");
    print "This is Page B.<BR>To select Page A, ";
    print $cgi->a ({-href => $url}, "click here");
}
print $cgi->end_html ();
```

如果另一个客户机程序出现并请求 flip_flop，就给出初始页面，因为不同客户机的浏览器并不互相影响。

实际上，$url 的值被前面的样例设置成漂亮的风格。在把它们放在 URL 之后以免包括特殊字符时，使用 escape( ) 方法对参数名和值进行编码是比较好的。这里有一个较好的方法来用附加的参数值来构造 URL：

```
$url = $cgi->script_name ();    # get URL for script
$url .= sprintf ("?%s=%s",      # add first parameter
            $cgi->escape ("name"), $cgi->escape ("value"));
$url .= sprintf ("&%s=%s",      # add second parameter
            $cgi->escape ("name2"), $cgi->escape ("value2"));
```

7.4.3 从 Web 脚本连接到 MySQL 服务器

我们在前一节“运行 DBI”中开发的命令行脚本，为建立到 MySQL 服务器的连接共享了一个通用的前文。CGI 脚本也共享了一些代码，但是有一些不同：

```
use DBI;
use CGI;
use strict;

# default connection  parameters - all missing
my ($host_name, $user_name, $password) = (undef, undef, undef);
my ($db_name) = "samp_db";

# construct data source
my ($dsn) = "DBI:mysql:$db_name";
$dsn .= ":hostname=$host_name" if $host_name;
$dsn .= ";mysql_read_default_file=/usr/local/apache/conf/samp_db.cnf";

# connect to server
my (%attr) = ( RaiseError => 1 );
my ($dbh) = DBI->connect ($dsn, $user_name, $password, \%attr);
```

这个前文与命令行脚本使用的前文的不同之处在于以下几个方面：

- 第一部分现在含有一条 use CGI 语句。
- 不再分析命令行的参数。
- 代码仍然在可选文件中寻找连接参数，但是，在用户执行脚本的主目录中不使用 .my.cnf 文件（也就 Web 服务器用户的主目录）。Web 服务器可能运行访问其他数据库的脚本，没有理由假设所有脚本会使用同一连接参数。相反，我们寻找不同位置存放的可选文件（/usr/local/apache/conf/samp_db.cnf）。如果想使用不同的文件，应该修改可选文件的路径名。

通过 Web 服务器调用的脚本作为 Web 服务器用户，而不是作为您来运行。这就提出了一些安全问题，因为在 Web 服务器接管之后您就不再控制了。应该把可选文件的所有权交给运行 Web 服务器的用户（可能是www 或者nobody 或者一些类似的用户），并将模式设置为400或600，以便其他用户不能读取。不幸的是，可以安装这个 Web 服务器的脚本来执行的任何人仍然能够读取这个文件。他们要做的所有事情就是编写一个脚本，显式地打开可选文件，并在Web页面上显示它的内容。因为他们的脚本作为 Web服务器用户来运行，所以它将有足够的权利来读取这个文件。

由于这个原因，创建一个对 samp_db 数据库具有只读（SELECT）权限的 MySQL 用户，然后在 samp_db.cnf 文件中列出这个用户的名称和口令，而不是您自己的名称和口令，这种行为是很谨慎的。作为有权修改数据库的表的用户，这种方式不会冒险允许脚本连接到数据库。第11章“常规的 MySQL 管理”，讨论了如何创建具有严格权限的 MySQL 用户账户。

另一种选择，可以在 Apache 的 suEXEC 机制下安排执行脚本。这就允许作为特殊权限的用户执行脚本，然后编写脚本，从只对那个用户为只读的可选文件中获得连接参数。例如，需要编写访问数据库的脚本，就可以这样做。

还有另外一种方法就是编写脚本，从客户机用户请求用户姓名和口令，并使用这些值建立到 MySQL 服务器的连接。这种方法对于为管理目的而创建脚本比对于为一般使用提供脚本更适合。无论如何，应该警惕用户名和口令请求的一些方法受到一些人的攻击，这些人可能在您和服务器之间的网络上安放窃听器。

因为可以从前面的段落中搜集，所以 Web 脚本的安全性是个棘手的问题。很明显，应该多读一些有关安全的主题，因为它是一个大的主题，所以在这里我不能真正做得很全面。查看 Apache 手册中有关安全性的资料是一种好的方法。您也可以查找 WWW 安全性的 FAQ 说明，例如可以使用下面的网址：

http://www.w3.org./Security/Faq/

7.4.4 samp_db数据库浏览器

对于第一个基于 Web 的应用程序，我们将开发一个简单的脚本 —— samp_db —— 允许查看 samp_db 数据库中存在的表，并从 Web 浏览器中交互式地检查这些表中的内容。samp_db 的工作方式如下：

- 当首次从浏览器中请求 samp_db 时，它连接到 MySQL 服务器，在 samp_db 数据库中检索一列表，并向浏览器发送一个页面，在这个页面中出现的每个表都作为可选择的连接。当选择这个页面中的一个表名时，浏览器就向 Web 服务器发送一个请求，请求 samp_browse 显示那个表的内容。
- 当调用 samp_browse 时，如果它收到从 Web 服务器发来的一个表名，则它就检索这个表的内容，并将信息显示在 Web 浏览器上。数据每列的标题就是表中列的名称。标题作为连接出现；如果选择它们中的一个，则浏览器就向 Web 服务器发送一个请求，显示同样的表，但按选择的列排序。

注意，这里有个警告：samp_db 表中的这些表相对较小，因此向浏览器发送表的全部内容并不是大问题。如果编辑 samp_db，显示包含大型表的不同数据库中的表，则应该考虑向行检索语句中增加一个LIMIT子句。

在 samp_browse 脚本的主体中，我们创建了 CGI 对象，并取消了 Web 页面的初始部分。然后检查是否按我们的假设，根据 tbl_name 参数值显示了一些特定的表：

```
my ($cgi) = new CGI;

# put out initial part of page
my ($title) = "$db_name Database Browser";
print $cgi->header ();
print $cgi->start_html (-title => $title);
print $cgi->h1 ($title);

# parameters to look for in URL
my ($tbl_name) = $cgi->param ("tbl_name");
my ($sort_column) = $cgi->param ("sort_column");

# if $tbl_name has no value, display a clickable list of tables.
# Otherwise, display contents of the given table.  $sort_column, if
# set, indicates which column to sort by.
if (!$tbl_name)
{
    display_table_list ()
}
else
{
    display_table ($tbl_name, $sort_column);
}

print $cgi->end_html ();
```

很容易找出参数的值，因为 CGI.pm 做了找出 Web 服务器传递给这个脚本信息的全部工作。我们只需调用具有我们感兴趣的参数名的 param( )，在samp_browse的主体中，这个参数为 tbl_name。如果它没有定义或者为空，则它就是这个脚本的初始调用，我们显示这个表列。否则，就显示由 tbl_name 参数命名的表的内容，由 sort_column 参数命名的列值排序。显示适当的信息之后，我们调用 end_html( ) 消除结束的 HTML 标志。

display_table_list( ) 函数生成初始页面。display_table_list( ) 检索这个表列并写出在每个单元中都含有一个数据库表名的单列的 HTML表：

```
sub display_table_list
{
my ($ary_ref, $url);

    print "Select a table by clicking on its name:<BR><BR>\n";

    # retrieve reference to single-column array of table names
    $ary_ref =
        $dbh->selectcol_arrayref (qq{ SHOW TABLES FROM $db_name });

    # display table with a border
    print "<TABLE BORDER>\n";
    print "<TR>\n";
    display_cell ("TH", "Table Name", 1);
    print "</TR>\n";
    foreach my $tbl_name (@{$ary_ref})
    {
        $url = $cgi->script_name ();
        $url .= sprintf ("?tbl_name=%s", $cgi->escape ($tbl_name));
        print "<TR>\n";
        display_cell ("TD", $cgi->a ({-href => $url}, $tbl_name), 0);
```

```
        print "</TR>\n";
        }
        print "</TABLE>\n";
    }
```

display_table_list( ) 生成的页面含有如下连接：

```
/cgi-bin/samp_browse?tbl_name=absence
/cgi-bin/samp_browse?tbl_name=event
/cgi-bin/samp_browse?tbl_name=member
...
```

当调用 samp_browse 时，如果 tbl_name 参数有值，则这个脚本将这个值传递给 display_table( )，连同按名称排序后的列名。如果没有命名的列，则我们按第一列排序（我们可以通过位置引用列，因而很容易地使用 ORDER BY 1子句来完成）：

```
sub display_table
{
my ($tbl_name, $sort_column) = @_;
my ($sth, $url);

    # if sort column not specified, use first column
    $sort_column = "1" unless $sort_column;

    # present a link that returns user to table list page
    print $cgi->a ({-href => $cgi->script_name ()}, "Show Table List");
    print "<BR><BR>\n";

    $sth = $dbh->prepare (qq{
                SELECT * FROM $tbl_name ORDER BY $sort_column
            });
    $sth->execute ();

    print "<B>Contents of $tbl_name table:</B><BR>\n";

    # display table with a border
    print "<TABLE BORDER>\n";
    # use column names for table headings; make each heading a link
    # that sorts output on the corresponding column
    print "<TR>\n";
    foreach my $col_name (@{$sth->{NAME}})
    {
        $url = $cgi->script_name ();
        $url .= sprintf ("?tbl_name=%s", $cgi->escape ($tbl_name));
        $url .= sprintf ("&sort_column=%s", $cgi->escape ($col_name));
        display_cell ("TH", $cgi->a ({-href => $url}, $col_name), 0);
    }
    print "</TR>\n";

    # display table rows
    while (my @ary = $sth->fetchrow_array ())
    {
       print "<TR>\n";
       foreach my $val (@ary)
       {
           display_cell ("TD", $val, 1);
       }
       print "</TR>\n";
    }

    $sth->finish ();
    print "</TABLE>\n";
}
```

表显示了与重新显示该表的连接相关的列标题的页面；这些连接包括 sort_column参数，它显式地指定排序的列。例如，对于显示 event 表内容的页面，列标题连接看起来如下所示：

```
/cgi-bin/samp_browse?tbl_name=event&sort_column=date
/cgi-bin/samp_browse?tbl_name=event&sort_column=type
/cgi-bin/samp_browse?tbl_name=event&sort_column=event_id
```

display_table_list( ) 和 display_table( ) 都使用了 display_cell( )，HTML表中作为单元显示值的实用程序函数。这个函数使用了一个小窍门，就是将空值转换为不可分的空格（   ），因为在带有边框的表中，空单元不会正确地显示边框。将不可分的空格放入这个单元中解决了这个问题。display_cell( ) 还具有控制是否将单元值编码的第三个参数。这是必需的，因为调用display_cell( )，显示了一些已经编码的单元值，如含有 URL 信息的列标题：

```
# display a value in a table cell; put non-breaking
# space in "empty" cells so borders show up

sub display_cell
{
my ($tag, $value, $encode) = @_;

    $value = $cgi->escapeHTML ($value) if $encode;
    $value = " " unless $value;
    print "<$tag>$value</$tag>\n";
}
```

如果想编写更通用的脚本，可以将 samp_browse 更改为浏览多个数据库。例如，在脚本开始时，可以显示服务器上的一列数据库，而不是一个特定数据库中的一列表。然后，选出一个数据库，获得它的一列表，再从那里继续。

7.4.5 学分保存方案分数浏览器

每当我们输入测试的分数时，都需要生成一个有序的分数列表，以便确定等级曲线，并分配字母等级。请注意，对于这个列表我们将做的所有事情就是显示它，以便能确定每个字母等级终止的位置。然后在返回给学生之前，在测试卷上标出等级分数。我们不在数据库中继续记录这个字母等级，因为在等级周期末尾的等级取决于数字分数，而不是字母等级。再请注意，严格地说，在创建检索分数的方法之前，就应该有一个输入分数的方法。我将输入分数的脚本一直保存到下一章。在这期间，在数据库中，我们已经从早期的等级周期部分中得到了几组分数。即使没有方便的分数输入方法，我们也可以使用具有那些分数的脚本。

我们浏览分数的脚本 score_browse 与 samp_browse 有些类似，但是希望查看给定测试或测验的分数这种更特定的目标。初始页面给出一列可以从中选择的可能的等级事件，允许用户选择它们中的任何一个，来查看与事件相关的分数。给定事件的分数按照高分在前的顺序按分数排序，因此可以显示出结果，并用它确定等级曲线。

score_browse只需要检查一个参数 event_id，查看是否指定了特定事件。如果不是，则 score_browse 就显示 event 表中的行，以便用户可以选择其中的一个。否则，就显示与所选事件相关的分数：

```
# parameter that tells us which event to display scores for
my ($event_id) = $cgi->param ("event_id");

# if $event_id has a value, display the event list.
# otherwise display the scores for the given event.
```

```
if (!$event_id)
{
    display_events ()
}
else
{
    display_scores ($event_id);
}
```

使用请求表列标题的列名，函数 display_events( ) 从 event 表中抽取信息，并以表格形式显示它。在每行的内部，显示 event_id 值，作为可以选择的连接，以触发检索相应事件分数的查询。每个事件的 URL 都只是到具有附加参数 score_browse的路径，这个参数指定事件号码：

```
/cgi-bin/score_browse?event_id=number
```

display_events( ) 函数编写如下：

```
sub display_events
{
my ($sth, $url);
    print "Select an event by clicking on its number:<BR><BR>\n";

    # get list of events
    $sth = $dbh->prepare (qq{
        SELECT event_id, date, type
        FROM event
        ORDER BY event_id
    });
    $sth->execute ();

    # display table with a border
    print "<TABLE BORDER>\n";
    # use column names for table column headings
    print "<TR>\n";
    foreach my $col_name (@{$sth->{NAME}})
    {
        display_cell ("TH", $col_name, 1);
    }
    print "</TR>\n";

    # associate each event id with a link that will show the
    # scores for the event; return rows using a hash to make
    # it easy to refer to the event_id column value by name.
    while (my $hash_ref = $sth->fetchrow_hashref ())
    {
        print "<TR>\n";
        $url = $cgi->script_name ();
        $url .= sprintf ("?event_id=%s",
                        $cgi->escape ($hash_ref->{event_id}));
        display_cell ("TD",
                    $cgi->a ({-href => $url}, $hash_ref->{event_id}), 0);
        display_cell ("TD", $hash_ref->{date}, 1);
        display_cell ("TD", $hash_ref->{type}, 1);
        print "</TR>\n";
    }
    $sth->finish ();
    print "</TABLE>\n";
}
```

当用户选择事件时，浏览器发送一个具有附加事件 ID 值的 score_browse 请求。

score_browse 找到 event_id 参数集，并调用 display_scores( ) 列出所有特定事件的分数。这个页面也显示了文本“ Show Event List ”，作为返回初始页面的连接，以便用户能很容易地返回事件列表页面。这个连接的 URL 引用了 score_browse 脚本，但不指定 event_id 参数的任何值。display_scores( )子程序如下所示：

```
sub display_scores
{
my ($event_id) = shift;
```

continued

```
    my ($sth);

        # a URL without any event_id parameter will
        # cause the event list to be displayed.
        print $cgi->a ({-href => $cgi->script_name ()}, "Show Event List");
        print "<BR><BR>\n";

        # select scores for the given event
        $sth = $dbh->prepare (qq{
            SELECT
                student.name, event.date, score.score, event.type
            FROM
                student, score, event
            WHERE
                student.student_id = score.student_id
                AND score.event_id = event.event_id
                AND event.event_id = ?
            ORDER BY
                event.date ASC, event.type ASC, score.score DESC
        });
        $sth->execute ($event_id);  # pass event ID as placeholder value

        print "<B>Scores for event $event_id</B><BR>\n";

        # display table with a border
        print "<TABLE BORDER>\n";
        # use column names for table column headings
        print "<TR>\n";
        foreach my $col_name (@{$sth->{NAME}})
        {
            display_cell ("TH", $col_name, 1);
        }
        print "</TR>\n";

        while (my @ary = $sth->fetchrow_array ())
        {
            print "<TR>\n";
            display_cell ("TD", shift (@ary), 1) while @ary;
            print "</TR>\n";
        }

        $sth->finish ();
        print "</TABLE>\n";
    }
```

display_scores( ) 运行的查询与我们以前在第 1章的1.4.8节中的“ 从多个表中检索信息 ”小节中开发的说明如何编写连接的查询极为类似。在那一章中，我们请求给定日期的分数，因为日期比事件的ID值更有意义。相反，当我们使用 score_browse 时，知道了精确的事件ID。那不是因为我们按照事件 ID 考虑（我们没有），而是因为脚本给了我们一列可从中选择的事

件ID。可以看到这种类型的接口减少了了解特定细节的需要。我们不必了解事件的 ID；只需要识别出想要的事件。

7.4.6 历史同盟共同兴趣的搜索

samp_browse 和 score_browse 脚本通过在初始页面给出一列选择而允许用户做出选择，那个页面中的每个选择都是用特定参数值再次调用这个脚本的一个连接。允许用户做出选择的另外一种方法是，将含有可编辑域的表格放在页面中。当可选范围没有约束到一些容易确定的值的集合时，这种方法更加合适。我们的下一个脚本举例说明了请求用户输入的这种方法。

在7.3节“运行 DBI”中，我们为寻找共享特定兴趣的历史联盟成员构造了一个命令行脚本。然而，那个脚本并不是联盟成员已经访问的脚本；联盟秘书必须运行这个脚本，然后把结果邮寄给请求这个列表的成员。最好使这个搜寻性能更广泛，以便成员可以自己使用。编写 Web 脚本就是进行这种事的一种方法。

脚本 interests 把少量表格放到用户能够输入关键字的地方，然后搜索 member 表，寻找有满足条件的成员并显示结果。通过将通配符“ %”加到关键字的两端，来执行这个搜索，以便在 interests 列值的任何地方都能找到。

在每个页面都显示关键字表格，以便用户可以立即输入新的搜索，甚至在显示搜索结果的页面中也可以。除此之外，还在关键字表格中显示前面页面的搜索字符串，以便如果用户想要运行类似的搜索，可以编辑这个字符串。这样就不必重新键入许多内容了：

```
# parameter to look for
my ($interest) = $cgi->param ("interest");

# Display a keyword entry form.  In addition, if $interest is defined,
# search for and display a list of members who have that interest.
# Note that the current $interest value is displayed as the default
# value of the interest field in the form.

print $cgi->start_form (-method => "POST");
print $cgi->textfield (-name => "interest",
                       -value => $interest,
                       -size => 40);
print $cgi->submit (-name => "button", -value => "Search");
print $cgi->end_form ();

# run a search if a keyword was specified
search_members ($interest) if $interest;
```

脚本和自己交流信息与 samp_browse 或 score_browse 略有不同。interest 参数没有加到 URL 的末尾，而表格中的信息由浏览器编码，并作为 POST 请求的一部分发送出去。然而，CGI.pm 使如何发送信息成为不相关的；参数值仍然通过调用 param( ) 来获得。

实现搜索和显示结果的函数如下所示，没有显示格式化项的函数 format_html_entry( )，因为它与 gen_dir 脚本中的相同：

```
sub search_members
{
my ($interest) = shift;
my ($sth, $count);

    printf "Search results for keyword: %s<BR><BR>\n",
```

```
                                    $cgi->escapeHTML ($interest);
    $sth = $dbh->prepare (qq{
                SELECT * FROM member WHERE interests LIKE ?
                ORDER BY last_name, first_name
            });
    # look for string anywhere in interest field
    $sth->execute ("%" . $interest . "%");
    $count = 0;
    while (my $hash_ref = $sth->fetchrow_hashref ())
    {
        format_html_entry ($hash_ref);
        ++$count;
    }
    print $cgi->p ("$count entries found");
}
```