第13章 MySQL高级编程

在本章中，我们将学习一些关于MySQL的高级话题，包括表格类型、事务和存储过程。

在本章中，我们将主要介绍以下内容：

- LOAD DATA INFILE语句
- 存储引擎
- 事务
- 外键
- 存储过程

13.1 LOAD DATA INFILE语句

到目前，我们还没有讨论的一个MySQL有用特性是LOAD DATA INFILE语句。可以使用这个语句从一个文件载入表数据。它的执行速度非常快。这个灵活的命令具有很多选项，但是常见用法如下所示：

```
LOAD DATA INFILE "newbooks.txt" INTO TABLE books;
```

该命令行从newbooks.txt文件将原始数据读入到表books。在默认情况下，文件中的数据字段必须通过Tab键进行间隔，而且必须包括在单引号内，同时每一行都必须由换行（\n）符进行间隔。特殊字符必须用“\”进行转义。所有这些特性都可以通过LOAD语句的不同选项进行配置；参阅MySQL手册，获得更详细信息。

要使用LOAD DATA INFILE语句，用户必须具有FILE权限，关于权限已经在第9章讨论过。

13.2 存储引擎

MySQL支持许多不同的存储引擎，有时候也称作*表格类型*。这就意味着对这些表的内部实现可以有选择。数据库每个表可以使用不同的存储引擎，而且可以轻松地对它们进行转换。

当使用如下所示的语句创建一个表时，可以选择一个表格类型：

```
CREATE TABLE table TYPE= type ....
```

常见的可用表格类型包括：

- **MyISAM**——这是默认类型，也是我们已经在本书中使用的类型。它是基于传统的ISAM类型，ISAM是Indexed Sequential Access Method（*有索引的顺序访问方法*）的缩写，它是存储记录和文件的标准方法。与其他存储引擎相比较，MyISAM具有检查和修复表格的大多数工具。MyISAM表格可以被压缩，而且它们支持全文搜索。它们不是事务安全的，而且也不支持外键。

- **MEMORY**（也就是以前的**HEAP**）——该类型的表存储在内存中，表的索引是哈希分布的。

这使得MEMORY表格运行得非常快，但是如果发生崩溃，数据将丢失。这些特性使MEMORY表非常适合保存临时数据或者派生的数据。应该在CREATE TABLE语句中指定MAX_ROWS，否则这些表可能会吞噬所有内存。同样，它们也不能具有BLOB、TEXT或AUTO INCREMENT列。

- **MERGE**——这些表允许你为了查询的目的，把MyISAM表的集合作为一个单个表。因此，你可在某些操作系统中避开最大文件大小限制。
- **ARCHIVE**——这些表保存了大量数据，但是只有少量的脚注（footprint）。这种类型的表只支持INSERT 和SELECT查询，不支持DELETE、UPDATE和REPLACE。此外，也不使用索引。
- **CSV**——这些表保存在服务器的单个文件中，它包含了用逗号间隔的数据。这种标类型的优点在于在需要查看的时候，否则，完全可以使用一种外部的表格应用程序来存储数据，例如Microsoft的Excel。
- **InnoDB**——这种类型的表是事务安全的；也就是说，它们提供了COMMIT和ROLLBACK功能。InnoDB表还支持外键。虽然比MyISAM表要慢，但是如果应用程序需要一个事务安全的存储引擎，我们建议使用它。

在大多数Web应用程序中，通常都会使用MyISAM或InnoDB表格或者二者的结合。

当对一个表格使用大量的SELECT或INSERT语句（或者二者的结合）时，应该使用MyISAM表格，因为在执行这两种命令时，MyISAM是最快的。对于许多Web应用程序（例如分类）来说，MyISAM是最佳选择。如果需要全文搜索功能，也应该使用MyISAM。当事务非常重要（例如存储财务数据的表格），或在INSERT和SELECT语句是交错执行的情况下（例如在线的消息栏或论坛系统），应该使用InnoDB。

对于临时表格或要是实现视图，可以使用MEMORY表格。如果需要处理大量的MyISAM表格，可以使用MERGE表格。

使用ALTER TABLE语句，可以在创建表格后修改表格的类型，如下所示：

```
alter table orders type=innodb;
alter table order_items type=innodb;
```

在贯穿本书的内容中，我们几乎都使用了MyISAM表格。下面，我们将花些时间集中介绍事务的使用，以及用InnoDB表格实现它们的方法。

13.3 事务

事务是确保数据库一致的机制，尤其是在发生错误或服务器崩溃情况下确保数据库一致的机制。在接下来的内容中，我们将学习事务的概念，以及如何使用InnoDB实现事务。

13.3.1 理解事务的定义

首先，让我们定义事务这个术语。事务是一个或一系列的查询，这些查询可以保证能够在

数据库中作为一个整体全部执行或者全部不执行。这样，数据库才能在无论事务是否完成的情况下保持一致状态。

要了解该功能的重要性原因，可以考虑一个银行数据库。假设希望将资金从一个账户转移到另一个账户的情况。这个动作将涉及从一个账户删除资金并且将这些资金放置在另一个账户内，这样至少涉及两个查询。这两个查询的同时执行或不执行都是至关重要的。如果从一个账户中取出资金，而在将这些资金存入到另一个账户之前，发生了停电，那会出现什么情况呢？资金会丢失么？

我们可能听说过ACID原则。ACID是描述事务安全性的4个需求：

- Atomicity（原子性）——一个事务必须是原子性的；也就是说，它必须是作为一个整体完全执行或者不执行。
- Consistency（一致性）——一个事务必须能够使数据库处于一致的状态。
- Isolation（孤立性）——未完全完成的事务不能被数据库的其他用户所见；也就是说，在事务完全完成之前，它们都是孤立的。
- Durability（持续性）——一旦写入到数据库后，事务必须是永久的而且持续的。

一个事务被永久地写入到数据库中称作该事务被提交了。一个没有写入到数据库中的事务（因此数据库将状态重置到事务开始之前的状态）称作事务被回滚了。

13.3.2 通过InnoDB使用事务

在默认的情况下，MySQL是以自动提交（autocommit）模式运行的。这就意味着所执行的每一个语句都将立即写入到数据库（提交）中。如果我们使用事务安全的表格类型，很可能不希望这种行为。

要在当前的会话中关闭自动提交，输入如下所示的命令：

```
set autocommit=0;
```

如果自动提交被打开了，必须使用如下所示语句开始一个事务：

```
start transaction;
```

如果自动提交是关闭的，不需要使用以上命令，因为当输入一个SQL语句时，一个事务将自动启动。

在完成了组成事务的语句输入后，可以使用如下所示语句将其提交给数据库：

```
commit;
```

如果改变主意，可以使用如下所示语句回到数据库以前的状态：

```
rollback;
```

只有提交了一个事务，该事务才能在其他会话中被其他用户所见。

下面，让我们来看一个例子。在books数据库中，执行本章上一节给出的ALTER TABLE语句。如果还没有执行此操作，请使用如下语句：

```
alter table orders type=innodb;
alter table order_items type=innodb;
```

这些语句可以将两个表格转换成InnoDB表格（如果希望使用type=MyISAM运行相同的语句，还可以将表格类性转换回来）。

现在，打开两个到books数据库的连接。在一个连接中，在数据库中添加一个新的订单记录：

```
insert into orders values (5, 2, 69.98, '2008-06-18');
insert into order_items values (5, '0-672-31697-8', 1);
```

现在检查一下能否看到新的订单：

```
select * from orders where orderid=5;
```

应该看到如下所示的订单记录：

```
+---------+------------+--------+------------+
| orderid | customerid | amount | date       |
+---------+------------+--------+------------+
|       5 |          2 |  69.98 | 2008-06-18 |
+---------+------------+--------+------------+
```

保持该连接的打开状态，进入到另一个连接，运行相同的SELECT查询。将无法看到该订单记录：

```
Empty set (0.00 sec)
```

（如果可以看到该订单记录，很可能是没有关闭自动提交。检查自动提交并且确认将表格类型转换成InnoDB格式）。其原因就是事务还没有被提交（这是事务孤立性的很好说明）。

现在回到第一个连接并且提交该事务：

```
commit;
```

应该可以在另一个连接中查询新的订单记录。

13.4 外键

InnoDB也支持外键。回忆一下，在第8章中，我们已经介绍了外键的概念。当使用MyISAM表格时，无法强制使用外键。

例如，假设在order_items表格中插入一行。必须包括一个有效的orderid。使用MyISAM表格，必须确认插入到程序逻辑任何位置的orderid具有有效性。在InnoDB中使用外键，可以让数据库完成检查操作。

如何设置外键呢？要创建一个使用外键的表格，可以改变该表格的DDL（数据定义语言）语句，如下所示：

```
create table order_items (
  orderid int unsigned not null references orders(orderid),
  isbn char(13) not null,
  quantity tinyint unsigned,
  primary key (orderid, isbn)
) type=InnoDB;
```

我们在orderid后添加了references orders(orderid)。这就意味着该列是一个外键，必须包含orders表格中的orderid列值。

最后，我们在声明的末尾添加了type=InnoDB的表格类型。这是外键所要求的。

使用ALTER TABLE语句，也可以对已有的表格进行以上修改，如下所示：

```
alter table order_items type=InnoDB;
alter table order_items
add foreign key (orderid) references orders(orderid);
```

要了解以上修改的工作原理，可以尝试插入一个数据行，而在orders表中，并没有与该行orderid相匹配的行：

```
insert into order_items values (77, '0-672-31697-8', 7);
```

将看到类似如下所示的错误：

```
ERROR 1452 (23000): Cannot add or update a child row:
a foreign key constraint fails
```

13.5　存储过程

一个存储过程是一个可编程的函数，它在MySQL中创建并保存。它可以由SQL语句和一些特殊的控制结构组成。当希望在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看作是对编程中面向对象方法的模拟。它们允许控制数据的访问方式。

首先，让我们来了解一个简单的示例。

13.5.1　基本示例

程序清单13-1显示了一个存储过程的声明。

程序清单13-1　basic_stored_procedure.sql——声明一个存储过程

```
# Basic stored procedure example
delimiter //

create procedure total_orders (out total float)
BEGIN
  select sum(amount) into total from orders;
END
//

delimiter ;
```

下面，让我们逐行分析以上代码。

第一行语句：

```
delimiter //
```

将语句末尾的分隔符从当前值（这个分隔符通常是分号，除非以前改变了分隔符）改为双斜杠字符。这样做的目的是可以在存储过程中使用分号分隔符，这样MySQL就会将分号当作是存储过程的代码，不会执行这些代码。

接下来的语句：

```
create procedure total_orders (out total float)
```

创建了实际的存储过程。该存储过程的名称是total_orders。它只有一个total参数，该参数是需要计算的值。OUT表示该参数将被传出或返回。

参数也可以声明为IN，表示该值必须传入到存储过程，或者INOUT，表示该值必须传入但是可以被存储过程修改。

float表示参数的类型。在这个例子中，将返回orders表中的所有订单的总数。orders列的类型为float，因此该返回类型也必须是float。可接受的数据类型映射到可供使用的列类型。

如果希望使用多个参数，可以提供一个由逗号间隔的参数列表，就像在PHP中的一样。过程体必须封闭在BEGIN和END语句中。它们都是对PHP中的括号（{}）的模拟，因为它们可以标识一个语句块。

在过程体中，只需运行一个SELECT语句。与常规SELECT语句的唯一差别在于使用into total子句将查询结果载入到total参数。

在声明了过程后，可以将分隔符重新设置为分号，如下语句所示：

```
delimiter ;
```

在过程声明之后，可以使用call关键字调用该过程，如下所示：

```
call total_orders(@t);
```

这个语句将调用total_orders过程并且传入一个用来保存结果的变量。要查看该结果，需要查看改变量，如下语句所示：

```
select @t;
```

其结果应该类似于如下所示：

```
+------------------+
| @t               |
+------------------+
| 289.92001152039  |
+------------------+
```

与创建过程的方法类似，可以创建一个函数。函数接收输入参数并且返回一个唯一值。创建函数的基本语法几乎相同。程序清单13-2显示了一个简单的函数。

程序清单13-2 basic_function.sql——声明一个存储函数

```
# Basic syntax to create a function

delimiter //

create function add_tax (price float) returns float
```

```
return price*1.1;
//

delimiter ;
```

可以看到，该示例使用了function关键字，而不是procedure关键字。此外，二者还存在一些其他差异。

参数不必通过IN或OUT来指定，因为所有参数都是IN，或输入参数。在参数列表之后是returns float子句，它指定了返回值的类型。需要再次提到的是，该值可以是任何有效的MySQL类型。

使用return语句，可以返回一个值，就像在PHP中所介绍的。

请注意，这个示例并没有使用BEGIN和END语句。可以使用它们，但是它们并不是必需的。就像在PHP中，如果一个语句块只包含了一个语句，就不需要标注该语句块的开始和结束。

调用函数与调用过程存在一些差异。可以以调用内置函数的相同方式调用一个存储函数。例如，

```
select add_tax(100);
```

该语句应该返回如下所示的输出：

```
+--------------+
| add_tax(100) |
+--------------+
|          110 |
+--------------+
```

在定义了过程和函数之后，可以使用如下所示的语句来查看定义这些过程和函数的代码：

```
show create procedure total_orders;
```

或者

```
show create function addtax;
```

也可以使用如下所示的语句来删除它们：

```
drop procedure total_orders;
```

或者

```
drop function add_tax;
```

存储过程提供了使用控制结构、变量、DECLARE句柄（就像异常）的功能，以及游标这个重要的概念。在接下来的内容中，我们将简单介绍这些概念。

13.5.2　局部变量

使用declare语句，可以在begin...end语句块中声明局部变量。例如，可以对add_tax函数进行修改，使其使用一个局部变量来保存税率，如程序清单13-3所示。

程序清单13-3 basic_function_with_variables.sql——声明一个具有变量的存储函数

```
# Basic syntax to create a function

delimiter //

create function add_tax (price float) returns float
begin
  declare tax float default 0.10;
  return price*(1+tax);
end
//
delimiter ;
```

正如你可以看到的，我们使用declare关键字以及变量名称和变量类型声明了该变量。

默认的子句是可选的，它指定了该变量的初始值。现在可以开始使用这个变量了。

13.5.3 游标和控制结构

现在，让我们来分析一个更复杂的例子。在这个例子中，我们将编写一个存储过程，该存储过程将计算出最大金额的订单，并且返回该订单的orderid（很明显，通过一个简单的查询，就可以计算出该数目，但是这个简单的示例只是说明了如何使用游标和控制结构）。该存储过程的代码如程序清单13-4所示。

程序清单13-4 control_structures_cursors.sql——使用游标和循环来处理一个结果集

```
# Procedure to find the orderid with the largest amount
# could be done with max, but just to illustrate stored procedure principles

delimiter //
create procedure largest_order(out largest_id int)
begin
  declare this_id int;
  declare this_amount float;
  declare l_amount float default 0.0;
  declare l_id int;

  declare done int default 0;
  declare continue handler for sqlstate '02000' set done = 1;
  declare c1 cursor for select orderid, amount from orders;

  open c1;
  repeat
    fetch c1 into this_id, this_amount;
    if not done then
      if this_amount > l_amount then
        set l_amount=this_amount;
        set l_id=this_id;
```

```
        end if;
      end if;
    until done end repeat;
  close c1;

  set largest_id=l_id;

end
//

delimiter ;
```

以上代码使用了控制结构（条件语句和循环语句）、游标和声明句柄。下面，我们逐行分析以上代码。

在该存储过程的开始处，声明了一些在该存储过程中使用的局部变量。this_id和this_amount变量保存了当前行的orderid和amount值。l_amount和l_id变量用来存储最大的订单金额和与之对应的ID。由于需要将每一个值与当前最大值进行比较，可以将当前最大值初始化为0。

下一个变量被声明为done，初始化为0（false）。这个变量是循环标记。当遍历了所有需要查看的行，可以将该变量设置为1（true）。

以下代码行：

```
declare continue handler for sqlstate '02000' set done = 1;
```

是一个声明句柄。它类似于存储过程中的一个异常。在continue句柄和exit句柄中，也可以使用它。就像以上代码所显示的，continue句柄执行了指定的动作，并且继续存储过程的执行。exit句柄将从最近的begin...end代码块中退出。

声明句柄的下一个部分指定了句柄被调用的时间。在这个例子中，该句柄将在sqlstate '02000'语句被执行时调用。你可能会奇怪，这是什么意思，因为该语句非常神秘。这意味着，该句柄将在无法再找到记录行后被调用。我们将逐行处理一个结果集，而且当遍历了所有需要处理的记录行时，这个句柄将被调用。也可以指定等价的FOR NOT FOUND语句。其他选项还包括SQLWARNING和SQLEXCEPTION。

接下来就是游标。一个游标类似于一个数组；它将从一个查询获得结果集（例如，mysqli_query()所返回的），并且允许一次只处理一行（例如，我们可能会使用mysqli_fetch_row()函数）。分析以下游标：

```
declare c1 cursor for select orderid, amount from orders;
```

这个游标名称为c1。这只是它将要保存内容的定义。该查询还不会被执行。

接下来一行代码：

```
open c1;
```

真正运行这个查询。要获得每一个数据行，必须运行一个fetch语句。可以在一个repeat循环中完成此操作。在这个例子中，循环语句如下所示：

```
repeat
...
until done end repeat;
```

请注意，只有在循环语句块的末尾才会检查循环条件。存储过程还支持while循环，如下形式所示：

```
while condition do
...
end while;
```

此外，还支持loop循环语句，如下形式所示：

```
loop
...
end loop
```

这些循环没有内置的循环条件，但是可以通过leave语句退出循环。

请注意，存储过程不支持for循环。

继续这个例子，以下代码将获得一个数据行：

```
fetch c1 into this_id, this_amount;
```

以上代码行将从游标查询中获得一个数据行。该查询所获得两个属性保存在两个指定的局部变量中。

我们可以检查一个数据行是否被获得，然后再将当前循环量与最大的存储值进行比较，通过两个IF语句的方式，如下所示：

```
if not done then
  if this_amount > l_amount then
    set l_amount-this_amount;
    set l_id=this_id;
  end if;
end if;
```

请注意，变量值将通过set语句进行设置。

除了if...then语句外，存储过程还支持if...then...else语句结构，如下形式所示：

```
if condition then
    ...
    [elseif condition then]
    ...
    [else]
    ...
end if
```

此外，也可以使用case语句，如下形式所示：

```
case value
    when value then statement
    [when value then statement ...]
    [else statement]
```

```
    end case
```

回到这个例子，在循环语句末尾，将执行一些清除操作：

```
    close c1;
    set largest_id=l_id;
```

close语句将关闭这个游标。

最后，将所计算出的最大值赋值给OUT参数。不能将该参数作为临时变量，只能用来保存最终值（这种用法类似于其他一些编程语言，例如Ada）。

如果按照以上方式创建了这个存储过程，可以像调用其他存储过程一样调用这个存储过程：

```
call largest_order(@l);
select @l;
```

将获得类似于如下所示的输出：

```
+------+
| @l   |
+------+
| 3    |
+------+
```

你可以自己检查计算结果是否正确。

13.6　进一步学习

在本章中，我们简单地介绍了存储过程的功能。在MySQL手册中，可以找到更多关于存储过程的介绍。

关于LOAD DATA INFILE、不同的存储引擎和存储过程的更多信息，请参阅MySQL手册。

如果希望找到更多关于事务和数据库一致性的信息，我们推荐一本关于关系数据库基本介绍的图书——《An Introduction to Database Systems》，由C.J.Date编写。

13.7　下一章

到这里，我们已经介绍了PHP和MySQL的基础内容。在第14章中，我们将介绍电子商务和设置一个基于数据库的Web站点安全性问题。