MySQL 5.0 存储过程

MySQL 5.0　新特性系列 第一部分

MySQL 技术白皮书

Peter Gulutzan
March, 2005
翻译：陈朋奕
西安电子科技大学
2005-5-6

Table of Contents

目录(目录不做翻译了，因为基本都是专有名词)

Introduction

本书是为需要了解5.0版本新特性的MySQL老用户而写的。简单的来说是介绍了“存储过程、触发器、视图、信息架构视图”，这是介绍MySQL 5.0新特性丛书的第一集。希望这本书能像内行专家那样与您进行对话，用简单的问题、例子让你学到需要的知识。
为了达到这样的目的，我会从每一个细节开始慢慢的为大家建立概念，最后会给大家展示较大的实用用例，在学习之前也许大家会认为这个用例很难，但是只要跟着课程去学，相信很快就能掌握。

Conventions and Styles约定和编程风格

每次我想要演示实际代码时，我会对mysql客户端的屏幕就出现的代码进行调整，将字体改成Courier，使他们看起来与普通文本不一样。在这里举个例子：

```
mysql> DROP FUNCTION f;
Query OK, 0 rows affected (0.00 sec)
```

如果实例比较大，则需要在某些行和段落间加注释，同时我会用将“<--”符号放在页面的右边以表示强调。例如：

```
mysql> CREATE PROCEDURE p ()
    -> BEGIN
    ->    /* This procedure does nothing */                    <--
    -> END;//
Query OK, 0 rows affected (0.00 sec)
```

有时候我会将例子中的"mysql>"和"->"这些系统显示去掉，你可以直接将代码复制到mysql客户端程序中（如果你现在所读的不是电子版的，可以在mysql.com网站下载相关脚本）

所以的例子都已经在Suse 9.2 Linux、Mysql 5.0.3公共版上测试通过。在您阅读本书的时候，Mysql已经有更高的版本，同时能支持更多OS了，包括Windows，Sparc，HP-UX。因此这里的例子将能正常的运行在您的电脑上。但如果运行仍然出现故障，可以咨询你认识的资深Mysql用户，以得到长久的支持和帮助。

A Definition and an Example 定义及实例

存储过程是一种存储在书库库中的程序（就像正规语言里的子程序一样），准确的来说，MySQL支持的“routines（例程）”有两种：一是我们说的存储过程， 二是在其他SQL语句中可以返回值的函数（使用起来和Mysql预装载的函数一样，如pi()）。我在本书里面会更经常使用存储过程，因为这是我们过去的习惯，相信大家也会接受。

一个存储过程包括名字，参数列表，以及可以包括很多SQL语句的SQL语句集。
在这里对局部变量，异常处理，循环控制和IF条件句有新的语法定义。
下面是一个包括存储过程的实例声明：
（译注：为了方便阅读，此后的程序不添任何中文注释）

```
CREATE PROCEDURE procedure1              /* name 存储过程名*/
(IN parameter1 INTEGER)                  /* parameters 参数*/
BEGIN                                    /* start of block 语句块头*/
  DECLARE variable1 CHAR(10);            /* variables变量声明 */
  IF parameter1 = 17 THEN                /* start of IF IF条件开始*/
    SET variable1 = 'birds';             /* assignment 赋值*/
  ELSE
    SET variable1 = 'beasts';            /* assignment 赋值*/
  END IF;                                /* end of IF IF结束*/
  INSERT INTO table1 VALUES (variable1);/* statement SQL语句*/
END                                      /* end of block 语句块结束*/
```

下面我将会介绍你可以利用存储过程做的工作的所有细节。同时我们将介绍新的数据库对象——触发器，因为触发器和存储过程的关联是必然的。

Why Stored Procedures 为什么要用存储过程

由于存储过程对于MySQL来说是新的功能，很自然的在使用时你需要更加注意。 毕竟，在此之前没有任何人使用过，也没有很多大量的有经验的用户来带你走他们走过的路。然而你应该开始考虑把现有程序（可能在服务器应用程序中， 用户自定义函数（UDF）中，或是脚本中）转移到存储过程中来。这样做不需要原因， 你不得不去做。

存储过程是已经被认证的技术！虽然在Mysql中它是新的，但是相同功能的函数在其他DBMS中早已存在，而它们的语法往往是相同的。因此你可以从 其他人那里获得这些概念，也有很多你可以咨询或者雇用的经验用户，还有许多第三方的文档可供你阅读。

存储过程会使系统运行更快！虽然我们暂时不能在Mysql上证明这个优势，用户得到的体验也不一样。我们可以说的就是Mysql服务器在缓存机制上做了改进，就像Prepared statements（预处理语句）所做的那样。由于没有编译器，因此SQL存储过程不会像外部语言（如C）编写的程序运行起来那么快。但是提升速度的主要方法却在于能否降低网络信息流量。如果你需要处理的是需要检查、循环、多语句但没有用户交互的重复性任务，你就可以使用保存在服务器上的存储过程来完成。这样在执行任务的每一步时服务器和客户端之间就没那么多的信息来往了。

存储过程是可复用的组件！想象一下如果你改变了主机的语言，这对存储过程不会产生影响，因为它是数据库逻辑而不是应用程序。存储过程是可以移植的！当你用SQL编写存储过程时，你就知道它可以运行在Mysql支持的任何平台上，不需要你额外添加运行环境包，也不需要为程序在操作系统中执行设置许可，或者为你的不同型号的电脑配置不同的包。这就是与Java、C或PHP等外部语言相比使用SQL语句的优势。不过，使用外部语言例程的好处还是很好的选择，它们只是没有以上的优点而已。

存储过程将被保存！如果你编写好了一个程序，例如显示银行事物处理中的支票撤消，那想要了解支票的人就可以找到你的程序。 它会以源代码的形式保存在数据库中。这将使数据和处理数据的进程有意义的关联这可能跟你在课上听到的规划论中说的一样。

存储过程可以移植！Mysql完全支持SQL 2003标准。某些数据库（如DB2、Mimer）同样支持。但也有部分不支持的，如Oracle、SQL Server不支持。我们将会给予足够帮助和工具，使为其他DBMS编写的代码能更容易转移到Mysql上。

Setting up with MySQL 5.0　设置并开始MySQL 5.0服务

通过mysql_fix_privilege_tables或者~/mysql-5.0/scripts/mysql_install_db来开始MySQL服务

作为我们练习的准备工作的一部分，我假定MySQL 5.0已经安装。如果没有数据库管理员为你安装好数据库以及其他软件，你就需要自己去安装了。不过你很容易忘掉一件事，那就是你需要有一个名为mysql.proc的表。

在安装了最新版本后，你必须运行mysql_fix_privilege_tables或者mysql_install_db（只需要运行其中一个就够了）——不然存储过程将不能工作。我同时启用在root身份后运行一个非正式的SQL脚本，如下：

```
mysql>source/home/pgulutzan/mysql-
5.0/scripts/mysql_prepare_privilege_tables_for_5.sql
```

Starting the MySQL Client 启动MySQL客户端

这是我启动mysql客户端的方式。你也许会使用其他方式，如果你使用的是二进制版本或者是 Windows系统的电脑，你可能会在其他子目录下运行以下程序：

```
pgulutzan@mysqlcom:~> /usr/local/mysql/bin/mysql --user=root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.3-alpha-debug

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

在演示中，我将会展示以root身份登陆后的mysql客户端返回的结果，这样意味着我有极大的特权。

Check for the Correct Version 核对版本

为了确认使用的MySQL的版本是正确的，我们要查询版本。我有两种方法确认我使用的是 5.0版本：

```
SHOW VARIABLES LIKE 'version';
or
SELECT VERSION();
```

例如：

```
mysql> SHOW VARIABLES LIKE 'version';
+---------------+------------------+
| Variable_name | Value            |
+---------------+------------------+
| version       | 5.0.3-alpha-debug |
+---------------+------------------+
1 row in set (0.00 sec)

mysql> SELECT VERSION();
+------------------+
| VERSION()        |
+------------------+
| 5.0.3-alpha-debug |
+------------------+
1 row in set (0.00 sec)
```

当看见数字'5.0.x'后就可以确认存储过程能够在这个客户端上正常工作。

The Sample "Database" 示例数据库

现在要做的第一件事是创建一个新的数据库然后设定为默认数据库实现这个步骤的SQL语句如下：

```
CREATE DATABASE db5;
USE db5;
```

例如：

```
mysql> CREATE DATABASE db5;
Query OK, 1 row affected (0.00 sec)

mysql> USE db5;
Database changed
```

在这里要避免使用有重要数据的实际的数据库然后我们创建一个简单的工作表。
实现这个步骤的SQL语句如下：

```
mysql> CREATE DATABASE db5;
Query OK, 1 row affected (0.01 sec)

mysql> USE db5;
Database changed

mysql> CREATE TABLE t (s1 INT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO t VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

你会发现我只在表中插入了一列。这样做的原因是我要保持表的简单，因为在这里并不需要展示查询数据的技巧，而是教授存储过程，不需要使用大的数据表，因为它本身已经够复杂了。

这就是示例数据库，我们将从这个名字为t的只包含一列的表开始

Pick a Delimiter 选择分隔符

现在我们需要一个分隔符，实现这个步骤的SQL语句如下：
DELIMITER //

例如：

mysql> DELIMITER //

分隔符是你通知mysql客户端你已经完成输入一个SQL语句的字符或字符串符号。一直以来我们都使用分号“；”，但在存储过程中，这会产生不少问题，因为存储过程中有许多语句，所以每一个都需要一个分号因此你需要选择一个不太可能出现在你的语句或程序中的字符串作为分隔符。我曾用过双斜杠“//”，也有人用竖线“|”。我曾见过在DB2程序中使用“@”符号的，但我不喜欢这样。你可以根据自己的喜好来选择，但是在这个课程中为了更容易理解，你最好选择跟我一样。如果以后要恢复使用“;”（分号）作为分隔符，输入下面语句就可以了：

"DELIMITER ;//".

CREATE PROCEDURE Example 创建程序实例

CREATE PROCEDURE p1 () SELECT * FROM t; //

也许这是你使用Mysql创建的第一个存储过程。假如是这样的话，最好在你的日记中记下这个重要的里程碑。
CREATE PROCEDURE p1 () SELECT * FROM t; // <--

SQL语句存储过程的第一部分是 “CREATE PROCEDURE”：

CREATE PROCEDURE p1 () SELECT * FROM t; // <--

第二部分是过程名，上面新存储过程的名字是p1。

Digression: Legal Identifiers 题外话：合法标识符的问题

存储过程名对大小写不敏感，因此‘P1’和‘p1’是同一个名字，在同一个数据库中你将不能给两个存储过程取相同的名字，因为这样将会导致重载。某些DBMS允许重载（Oracle支持），但是MySQL不支持（译者话：希望以后会支持吧。）。 你可以采取“数据库名.存储过程名”这样的折中方法，如“db5.p1”。存储过程名可以分开，它可以包括空格符，其长度限制为64个字符，但注意不要使用MySQL内建函数的名字，如果这样做了，在调用时将会出现下面的情况：

mysql> CALL pi();
Error 1064 (42000): You have a syntax error.
mysql> CALL pi ();
Error 1305 (42000): PROCEDURE does not exist.

在上面的第一个例子里，我调用的是一个名字叫pi的函数，但你必须在调用的函数名后加上空格，就像第二个例子那样。

CREATE PROCEDURE p1 () SELECT * FROM t; //　　<--

其中“（）”是“参数列表”。

CREATE PROCEDURE 语句的第三部分是参数列表。通常需要在括号内添加参数。例子中的存储过程没有参数，因此参数列表是空的——所以我只需要键入空括号，然而这是必须的。

CREATE PROCEDURE p1 () SELECT * FROM t; //　　<--

"SELECT * FROM t;"是存储过程的主体。

然后到了语句的最后一个部分了，它是存储过程的主体，是一般的SQL语句。过程体中语句"SELECT * FROM t;"包含一个分号，如果后面有语句结束符号（//）时可以不写这个分号。如果你还记得我把这部分叫做程序的主体将会是件好事，因为（body）这个词是大家使用的技术上的术语。通常我们不会将SELECT语句用在存储过程中，这里只是为了演示。所以使用这样的语句，能在调用时更好的看出程序是否正常工作。

Why MySQL Statements are Legal in a Procedure Body

什么MySQL语句在存储过程体中是合法的？

什么样的SQL语句在Mysql存储过程中才是合法的呢？你可以创建一个包含INSERT, UPDATE, DELETE, SELECT, DROP, CREATE, REPLACE等等的语句。你唯一需要记住的是如果代码中包含MySQL扩充功能，那么代码将不能移植。在标准SQL语句中：任何数据库定义语言都是合法的，如：

CREATE PROCEDURE p () DELETE FROM t; //

SET、COMMIT以及ROLLBACK也是合法的，如：

CREATE PROCEDURE p () SET @x = 5; //

MySQL的附加功能：任何数据操作语言的语句都将合法。

CREATE PROCEDURE p () DROP TABLE t; //

MySQL扩充功能：直接的SELECT也是合法的：

CREATE PROCEDURE p () SELECT 'a'; //

顺便提一下，我将存储过程中包括DDL语句的功能称为MySQL附加功能的原因是在SQL标准中把这个定义为非核心的，即可选组件。

在过程体中有一个约束，就是不能有对例程或表操作的数据库操作语句。例如下面的例子就是非法的：

```
CREATE PROCEDURE p1 ()
 CREATE PROCEDURE p2 () DELETE FROM t; //
```

下面这些对MySQL 5.0来说全新的语句，在过程体中是非法的：
CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE, CREATE FUNCTION, DROP FUNCTION, CREATE TRIGGER, DROP TRIGGER. 不过你可以使用"CREATE PROCEDURE db5.p1 () DROP DATABASE db5//"，但是类似"USE database"语句也是非法的，因为MySQL假定默认数据库就是过程的工作场所。

Call the Procedure　调用存储过程

1.　现在我们就可以调用一个存储过程了，你所需要输入的全部就是CALL和你过程名以及一个括号再一次强调，括号是必须的当你调用例子里面的p1过程时，结果是屏幕返回了t表的内容

```
mysql> CALL p1() //
+------+
| s1   |
+------+
|    5 |
+------+
1 row in set (0.03 sec)
Query OK, 0 rows affected (0.03 sec)
```

因为过程中的语句是"SELECT * FROM t;"

2.　其他实现方式

```
mysql> CALL p1() //
```

和下面语句的执行效果一样：

```
mysql> SELECT * FROM t; //
```

所以，你调用p1过程就相当于你执行了下面语句：

"SELECT * FROM t;".

好了，主要的知识点“创建和调用过程方法”已经清楚了。我希望你能对自己说这相当简单。但是很快我们就有一系列的练习，每次都加一条子句，或者改变已经存在的子句。那样在写复杂部件前我们将会有很多可用的子句。

Characteristics Clauses 特征子句

1.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL                                    <--
NOT DETERMINISTIC                               <--
SQL SECURITY DEFINER                            <--
COMMENT 'A Procedure'                           <--
SELECT CURRENT_DATE, RAND() FROM t //
```

这里我给出的是一些能反映存储过程特性的子句。子句内容在括号之后，主体之前。这些子句都是可选的，他们有什么作用呢？

2.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL                                    <--
NOT DETERMINISTIC
SQL SECURITY DEFINER
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

很好，这个LANGUAGE SQL子句是没有作用的。仅仅是为了说明下面过程的主体使用SQL语言编写。这条是系统默认的，但你在这里声明是有用的，因为某些DBMS（IBM的DB2）需要它，如果你关注DB2的兼容问题最好还是用上。此外，今后可能会出现除SQL外的其他语言支持的存储过程 。

3.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC                               <--
SQL SECURITY DEFINER
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

下一个子句，NOT DETERMINISTIC，是传递给系统的信息。这里一个确定过程的定义就是那些每次输入一样输出也一样的程序。在这个案例中，既然主体中含有SELECT语句，那返回肯定是未知的因此我们称其NOT DETERMINISTIC。但是MySQL内置的优化程序不会注意这个，至少在现在不注意。

4.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC
SQL SECURITY DEFINER                                    <--
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t //
```

下一个子句是SQL SECURITY，可以定义为SQL SECURITY DEFINER或SQL SECURITY INVOKER。 这就进入了权限控制的领域了，当然我们在后面将会有测试权限的例子。

SQL SECURITY DEFINER意味着在调用时检查创建过程用户的权限（另一个选项是SQL SECURITY INVOKER）。

现在而言，使用SQL SECURITY DEFINER指令告诉MySQL服务器检查创建过程的用户就可以了，当过程已经被调用，就不检查执行调用过程的用户了。而另一个选项（INVOKER）则是告诉服务器在这一步仍然要检查调用者的权限。

5.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC
SQL SECURITY DEFINER
COMMENT 'A Procedure'                                   <--
SELECT CURRENT_DATE, RAND() FROM t //
```

COMMENT 'A procedure' 是一个可选的注释说明。

最后，注释子句会跟过程定义存储在一起。这个没有固定的标准，我在文中会指出没有固定标准的语句，不过幸运的是这些在我们标准的SQL中很少。

6.

```
CREATE PROCEDURE p2 ()
LANGUAGE SQL
NOT DETERMINISTIC
SQL SECURITY DEFINER
COMMENT ''
SELECT CURRENT_DATE, RAND() FROM t //
```

上面过程跟下面语句是等效的：

```
CREATE PROCEDURE p2 ()
SELECT CURRENT_DATE, RAND() FROM t //
```

特征子句也有默认值，如果省略了就相当于：
LANGUAGE SQL NOT DETERMINISTIC SQL SECURITY DEFINER COMMENT ''.

一些题外话

：调用p2()//的结果

```
mysql> call p2() //
+--------------+-----------------+
| CURRENT_DATE | RAND()          |
+--------------+-----------------+
| 2004-11-09   | 0.7822275075896 |
+--------------+-----------------+
1 row in set (0.26 sec)
Query OK, 0 rows affected (0.26 sec)
```

当调用过程p2时，一个SELECT语句被执行返回我们期望获得的随机数。

：不会改变的sql_mode

```
mysql> set sql_mode='ansi' //
mysql> create procedure p3()select'a'||'b'//
mysql> set sql_mode=''//
mysql> call p3()//
+------------+
| 'a' || 'b' |
+------------+
| ab         |
+------------+
```

MySQL在过程创建时会自动保持运行环境。例如：我们需要使用两条竖线来连接字符串但是这只有在sql mode为ansi的时候才合法。如果我们将sql mode改为non-ansi，不用担心，它仍然能工作，只要它第一次使用时能正常工作。

Exercise 练习

Question 问题

如果你不介意练习一下的话，试试能否不看后面的答案就能处理这些请求。

创建一个过程，显示`Hello world`。用大约5秒时间去思考这个问题，既然你已经学到了这里，这个应该很简单。当你思考问题的时候，我们再随机选择一些刚才讲过的东西复习：DETERMINISTIC（确定性）子句是反映输出和输入依赖特性的子句……调用过程使用 CALL 过程名（参数列表）方式。好了，我猜时间也到了。

Answer 答案

好的，答案就是在过程体中包含"SELECT 'Hello, world'"语句

```
mysql> CREATE PROCEDURE p4 () SELECT 'Hello, world' //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p4()//
+--------------+
| Hello, world |
+--------------+
| Hello, world |
+--------------+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

Parameters 参数

让我们更进一步的研究怎么在存储过程中定义参数

1. CREATE PROCEDURE p5
   () ...

2. CREATE PROCEDURE p5
   ([IN] name data-type) ...

3. CREATE PROCEDURE p5
   (OUT name data-type) ...

4. CREATE PROCEDURE p5
   (INOUT name data-type) ...

回忆一下前面讲过的参数列表必须在存储过程名后的括号中。上面的第一个例子中的参数列表是空的，第二个例子中有一个输入参数。这里的词IN可选，因为默认参数为IN（input）。第三个例子中有一个输出参数，第四个例子中有一个参数，既能作为输入也可以作为输出。

IN example 输入的例子

```
mysql> CREATE PROCEDURE p5(p INT) SET @x = p //
Query OK, 0 rows affected (0.00 sec)
mysql> CALL p5(12345)//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x//
+-------+
| @x    |
+-------+
| 12345 |
+-------+
1 row in set (0.00 sec)
```

这个IN的例子演示的是有输入参数的过程。在过程体中我将会话变量x设定为参数p的值。然后调用过程，将12345传入参数p。选择显示会话变量@x，证明我们已经将参数值12345传入。

OUT example 输出的例子

```
mysql> CREATE PROCEDURE p6 (OUT p INT)
    -> SET p = -5 //
mysql> CALL p6(@y)//
mysql> SELECT @y//
+------+
| @y   |
+------+
| -5   |
+------+
```

这是另一个例子。这次的p是输出参数，然后在过程调用中将p的值传入会话变量@y中。
在过程体中，我们给参数赋值-5，在调用后我们可以看出，OUT是告诉DBMS值是从过程中传出的。
同样我们可以用语句"SET @y = -5;".来达到同样的效果

Compound Statements 复合语句

现在我们展开的详细分析一下过程体：

```
CREATE PROCEDURE p7 ()
BEGIN
  SET @a = 5;
  SET @b = 5;
  INSERT INTO t VALUES (@a);
  SELECT s1 * @a FROM t WHERE s1 >= @b;
END; //          /* I won't CALL this. 这个语句将不会被调用*/
```

完成过程体的构造就是BEGIN/END块。这个BEGIN/END语句块和Pascal语言中的BEGIN/END是基本相同的，和C语言的框架是很相似的。我们可以使用块去封装多条语句。在这个例子中，我们使用了多条设定会话变量的语句，然后完成了一些insert和select语句。如果你的过程体中有多条语句，那么你就需要BEGIN/END块了。 BEGIN/END块也被称为复合语句，在这里你可以进行变量定义和流程控制。

The New SQL Statements 新SQL语句

Variables 变量

在复合语句中声明变量的指令是DECLARE。

(1) Example with two DECLARE statements 两个DECLARE语句的例子

```
 CREATE PROCEDURE p8 ()
BEGIN
  DECLARE a INT;
  DECLARE b INT;
  SET a = 5;
  SET b = 5;
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //          /* I won't CALL this */
```

在过程中定义的变量并不是真正的定义，你只是在BEGIN/END块内定义了而已（译注：也就是形参）。 注意这些变量和会话变量不一样，不能使用修饰符@ 你必须清楚的在BEGIN/END块中声明变量和它们的类型。 变量一旦声明，你就能在任何能使用会话变量、文字、列名的地方使用。

(2) Example with no DEFAULT clause and SET statement 没有默认子句和设定语句的例子

```
 CREATE PROCEDURE p9 ()
BEGIN
  DECLARE a INT /* there is no DEFAULT clause */;
  DECLARE b INT /* there is no DEFAULT clause */;
  SET a = 5;      /* there is a SET statement */
  SET b = 5;      /* there is a SET statement */
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //          /* I won't CALL this */
```

有很多初始化变量的方法。如果没有默认的子句，那么变量的初始值为NULL。你可以在任何时候使用SET语句给变量赋值。

(3) Example with DEFAULT clause 含有DEFAULT子句的例子

```
 CREATE PROCEDURE p10 ()
BEGIN
  DECLARE a, b INT DEFAULT 5;
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //
```

我们在这里做了一些改变，但是结果还是一样的。在这里使用了DEFAULT子句来设定初始值，这就不需要把DECLARE和SET语句的实现分开了。

(4) Example of CALL 调用的例子

```
 mysql> CALL p10() //
+--------+
| s1 * a |
+--------+
|     25 |
|     25 |
+--------+
2 rows in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

结果显示了过程能正常工作

(5) Scope 作用域

```
CREATE PROCEDURE p11 ()
BEGIN
  DECLARE x1 CHAR(5) DEFAULT 'outer';
  BEGIN
    DECLARE x1 CHAR(5) DEFAULT 'inner';
    SELECT x1;
    END;
  SELECT x1;
  END; //
```

现在我们来讨论一下作用域的问题。例子中有嵌套的BEGIN/END块，当然这是合法的。同时包含两个变量，名字都是x1，这样也是合法的。内部的变量在其作用域内享有更高的优先权。当执行到END语句时，内部变量消失，此时已经在其作用域外，变量不再可见了，因此在存储过程外再也不能找到这个声明了的变量，但是你可以通过OUT参数或者将其值指派给会话变量来保存其值

调用作用域例子的过程：

```
mysql> CALL p11()//
+-------+
| x1    |
+-------+
| inner |
+-------+
+-------+
| x1    |
+-------+
| outer |
+-------+
```

我们看到的结果时第一个SELECT语句检索到最内层的变量，第二个检索到第二层的变量

Conditions and IF-THEN-ELSE　条件式和IF-THEN-ELSE

1.　现在我们可以写一些包含条件式的例子：

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

这里是一个包含IF语句的过程。里面有两个IF语句，一个是IF 语句 END IF，另一个是IF 语句 ELSE 语句 END IF。我们可以在这里使用复杂的过程，但我会尽量使其简单让你能更容易弄清楚。

2.

```
CALL p12 (0) //
```

我们调用这个过程，传入值为0，这样parameter1的值将为0。

3.

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;                    <--
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

这里变量variable1被赋值为parameter1加1的值，所以执行后变量variable1为1。

4.

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN                              <--
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

因为变量variable1值为1，因此条件"if variable1 = 0"为假，IF……END IF被跳过，没有被执行。

5.

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN                             <--
    UPDATE t SET s1 = s1 + 1;
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

到第二个IF条件，判断结果为真，于是中间语句被执行了

6.

```
CREATE PROCEDURE p12 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  IF variable1 = 0 THEN
    INSERT INTO t VALUES (17);
  END IF;
  IF parameter1 = 0 THEN
    UPDATE t SET s1 = s1 + 1;                    <--
  ELSE
    UPDATE t SET s1 = s1 + 2;
  END IF;
END; //
```

因为参数parameter1值等于0，UPDATE语句被执行。如果parameter1值为NULL，则下一条UPDATE语句将被执行 现在表t中有两行，他们都包含值5，所以如果我们调用p12，两行的值会变成6。

7.

```
mysql> CALL p12(0)//
Query OK, 2 rows affected (0.28 sec)

mysql> SELECT * FROM t//
+------+
| s1   |
+------+
|    6 |
|    6 |
+------+
2 rows in set (0.01 sec)
```

结果也是我们所期望的那样。

CASE 指令

1.

```
CREATE PROCEDURE p13 (IN parameter1 INT)
BEGIN
  DECLARE variable1 INT;
  SET variable1 = parameter1 + 1;
  CASE variable1
    WHEN 0 THEN INSERT INTO t VALUES (17);
    WHEN 1 THEN INSERT INTO t VALUES (18);
    ELSE INSERT INTO t VALUES (19);
  END CASE;
END; //
```

如果需要进行更多条件真假的判断我们可以使用CASE语句。CASE语句使用和IF一样简单。
我们可以参考上面的例子：

2.

```
mysql> CALL p13(1)//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t//
+------+
| s1   |
+------+
|    6 |
|    6 |
|   19 |
+------+
3 rows in set (0.00 sec)
```

执行过程后，传入值1，如上面例子，值19被插入到表t中。

Question 问题

问题: CALL p13(NULL) //的作用是什么？
另一个：这个CALL语句做了那些动作？
你可以通过执行后观察SELECT做了什么，也可以根据代码判断，在5秒内做出。

Answer 答案

```
 mysql> CALL p13(NULL)//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t//
+------+
| s1   |
+------+
|    6 |
|    6 |
|   19 |
|   19 |
+------+
4 rows in set (0.00 sec)
```

答案是当你调用p13时，MySQL插入了另一条包含数值19的记录。原因是变量variable1的值为NULL，CASE语句的ELSE部分就被执行了。希望这对大家有意义。如果你回答不出来，没有问题，我们可以继续向下走。

Loops 循环语句

WHILE ... END WHILE
LOOP ... END LOOP
REPEAT ... END REPEAT
GOTO

下面我们将会创建一些循环。我们有三种标准的循环方式：WHILE循环，LOOP循环以及 REPEAT循环。还有一种非标准的循环方式：GO TO（译者语：最好不要用吧，用了就使流程混乱）。

WHILE ... END WHILE

```
CREATE PROCEDURE p14 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  WHILE v < 5 DO
    INSERT INTO t VALUES (v);
    SET v = v + 1;
  END WHILE;
END; //
```

这是WHILE循环的方式。我很喜欢这种方式，它跟IF语句相似，因此不需要掌握很多新的语法。这里的INSERT和SET语句在WHILE和END WHILE之间，当变量v大于5的时候循环将会退出。使用"SET v = 0;"语句使为了防止一个常见的错误，如果没有初始化，默认变量值为NULL，而NULL和任何值操作结果都为NULL。

WHILE ... END WHILE example

```
 mysql> CALL p14()//
Query OK, 1 row affected (0.00 sec)
```

以上就是调用过程p14的结果

不用关注系统返回是"one row affected" 还是 "five rows affected"，因为这里的计数只对最后一个INSERT动作进行计数。

WHILE ... END WHILE example: CALL

```
mysql> select * from t; //
+------+
| s1   |
+------+
....
|    0 |
|    1 |
|    2 |
|    3 |
|    4 |
+------+
9 rows in set (0.00 sec)
```

调用后可以看到程序向数据库中插入了5行。

REPEAT ... END REPEAT

```
 CREATE PROCEDURE p15 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  REPEAT
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    UNTIL v >= 5
  END REPEAT;
END; //
```

这是一个REPEAT循环的例子，功能和前面WHILE循环一样。区别在于它在执行后检查结果，而WHILE则是执行前检查。（译者语：可能等同于DO WHILE吧）

REPEAT ... END REPEAT: look at the UNTIL： UNTIL的作用

```
 CREATE PROCEDURE p15 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  REPEAT
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    UNTIL v >= 5                                    <--
  END REPEAT;
END; //
```

注意到UNTIL语句后面没有分号，在这里可以不写分号，当然你加上额外的分号更好。

REPEAT ... END REPEAT: calling ： 调用

```
mysql> CALL p15()//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT COUNT(*) FROM t//
+----------+
| COUNT(*) |
+----------+
|          14 |
+----------+
1 row in set (0.00 sec)
```

我们可以看到调用p15过程后又插入了5行记录

LOOP ... END LOOP

```
 CREATE PROCEDURE p16 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
    END IF;
  END LOOP;
END; //
```

以上是LOOP循环的例子。LOOP循环不需要初始条件，这点和WHILE循环相似，同时它又和REPEAT循环一样也不需要结束条件。

LOOP ... END LOOP: with IF and LEAVE 包含IF和LEAVE的LOOP循环

```
CREATE PROCEDURE p16 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN                                    <--
      LEAVE loop_label;
    END IF;
  END LOOP;
END; //
```

在循环内部加入IF语句，在IF语句中包含LEAVE语句。这里LEAVE语句的意义是离开循环。LEAVE的语法是LEAVE加循环语句标号，关于循环语句的标号问题我会在后面进一步讲解。

LOOP ... END LOOP: calling ： 调用

```
 mysql> CALL p16()//
Query OK, 1 row affected (0.00 sec)

mysql> SELECT COUNT(*) FROM t//
+----------+
| COUNT(*) |
+----------+
|          19 |
+----------+
1 row in set (0.00 sec)
```

调用过程p16后，结果是另5行被插入表t中。

Labels 标号

```
CREATE PROCEDURE p17 ()
label_1: BEGIN
  label_2: WHILE 0 = 1 DO LEAVE label_2; END
WHILE;
  label_3: REPEAT LEAVE label_3; UNTIL 0 =0
END REPEAT;
  label_4: LOOP LEAVE label_4; END LOOP;
END; //
```

最后一个循环例子中我使用了语句标号。现在这里有一个包含4个语句标号的过程的例子。我们可以在BEGIN、WHILE、REPEAT或者LOOP语句前使用语句标号，语句标号只能在合法的语句前面使用。因此"LEAVE label_3"意味着离开语句标号名定义为label_3的语句或复合语句。

End Labels 标号结束符

```
 CREATE PROCEDURE p18 ()
label_1: BEGIN
  label_2: WHILE 0 = 1 DO LEAVE label_2; END
WHILE label_2;
  label_3: REPEAT LEAVE label_3; UNTIL 0 =0
END REPEAT label_3 ;
  label_4: LOOP LEAVE label_4; END LOOP
label_4 ;
END label_1 ; //
```

你也可以在语句结束时使用语句标号，和在开头时使用一样。这些标号结束符并不是十分有用。它们是可选的。如果你需要，他们必须和开始定义的标号名字一样当然为了有良好的编程习惯，方便他人阅读，最好还是使用标号结束符。

LEAVE and Labels 跳出和标号

```
 CREATE PROCEDURE p19 (parameter1 CHAR)
label_1: BEGIN
  label_2: BEGIN
    label_3: BEGIN
      IF parameter1 IS NOT NULL THEN
        IF parameter1 = 'a' THEN
          LEAVE label_1;
        ELSE BEGIN
          IF parameter1 = 'b' THEN
            LEAVE label_2;
          ELSE
            LEAVE label_3;
            END IF;
          END;
        END IF;
    END;
  END;
END;//
```

LEAVE语句使程序跳出复杂的复合语句。

ITERATE 迭代

如果目标是ITERATE（迭代）语句的话，就必须用到LEAVE语句

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

ITERATE（迭代）语句和LEAVE语句一样也是在循环内部的循环引用， 它有点像C语言中的“Continue”，同样它可以出现在复合语句中，引用复合语句标号，ITERATE（迭代）意思是重新开始复合语句。
那我们启动并观察下面这个循环，这是个需要迭代过程的循环：

ITERATE: Walking through the loop深入循环

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP                                    <--
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

让这个已经定义了标号的循环运行起来。

ITERATE: Walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN                                     <--
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

v的值变成3，然后我们把它增加到4。

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;                             <--
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
```

```
        LEAVE loop_label;
        END IF;
      END LOOP;
END; //
```

然后开始ITERATE（迭代）过程。

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP                                    <--
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //
```

这里的ITERATE（迭代）让循环又回到了循环的头部。

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;                               <--
      END IF;
    END LOOP;
END; //
```

当v的值变为5时，程序将执行LEAVE语句

ITERATE: walking through the loop

```
CREATE PROCEDURE p20 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  loop_label: LOOP
    IF v = 3 THEN
      SET v = v + 1;
```

```
      ITERATE loop_label;
      END IF;
    INSERT INTO t VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE loop_label;
      END IF;
    END LOOP;
END; //                                                    <--
```

LEAVE的结果就是跳出循环，使运行指令到达复合语句的最后一步。

GOTO

```
CREATE PROCEDURE p...
BEGIN
...
LABEL label_name;
...
GOTO label_name;
...
END;
```

MySQL的存储过程中可以使用GOTO语句。虽然这不是标准SQL语句，而且在这里建立标号的方法也和惯例中的不一样。由于为了和其他DBMS兼容，这个语句会慢慢被淘汰，所以我们在MySQL参考手册中没有提及。

Grand combination 大组合

```
CREATE PROCEDURE p21
(IN parameter_1 INT, OUT parameter_2 INT)
LANGUAGE SQL DETERMINISTIC SQL SECURITY INVOKER
BEGIN
  DECLARE v INT;
  label goto_label; start_label: LOOP
    IF v = v THEN LEAVE start_label;
    ELSE ITERATE start_label;
    END IF;
  END LOOP start_label;
  REPEAT
    WHILE 1 = 0 DO BEGIN END;
    END WHILE;
    UNTIL v = v END REPEAT;
  GOTO goto_label;
END;//
```

上面例子中的语句包含了我们之前讲的所有语法，包括参数列表，特性参数，BEGIN/END块复合语句，变量声明，IF，WHILE，LOOP，REPEAT，LEAVE，ITERATE，GOTO。这是一个荒谬的存储过程，我不会运行它，因为里面有无尽的循环。但是里面的语法却十分合法。

这些是新的流程控制和变量声明语句。下面我们将要接触更多新的东西。

Error Handling 异常处理

后面几页的信息摘要
Sample Problem 问题样例
Handlers 异常处理器
Conditions 条件

好了，我们现在要讲的是异常处理

1. Sample Problem: Log Of Failures 问题样例：故障记录

当INSERT失败时，我希望能将其记录在日志文件中我们用来展示出错处理的问题样例是很普通的。我希望得到错误的记录。当INSERT失败时，我想在另一个文件中记下这些错误的信息，例如出错时间，出错原因等。我对插入特别感兴趣的原因是它将违反外键关联的约束

2. Sample Problem: Log Of Failures (2)

```
mysql> CREATE TABLE t2
        s1 INT, PRIMARY KEY (s1))
        engine=innodb;//
mysql> CREATE TABLE t3 (s1 INT, KEY (s1),
        FOREIGN KEY (s1) REFERENCES t2 (s1))
        engine=innodb;//
mysql> INSERT INTO t3 VALUES (5);//
...
ERROR 1216 (23000): Cannot add or update a child row: a foreign key
constraint fails （这里显示的是系统的出错信息）
```

我开始要创建一个主键表，以及一个外键表。我们使用的是InnoDB，因此外键关联检查是打开的。然后当我向外键表中插入非主键表中的值时，动作将会失败。当然这种条件下可以很快找到错误号1216。

3. Sample Problem: Log Of Failures

```
 CREATE TABLE error_log (error_message
CHAR(80))//
```

下一步就是建立一个在做插入动作出错时存储错误的表。

4. Sample Problem: Log Of Errors

```
 CREATE PROCEDURE p22 (parameter1 INT)
BEGIN
  DECLARE EXIT HANDLER FOR 1216
    INSERT INTO error_log VALUES
    (CONCAT('Time: ',current_date,
     '. Foreign Key Reference Failure For
Value = ',parameter1));
  INSERT INTO t3 VALUES (parameter1);
END;//
```

上面就是我们的程序。这里的第一个语句DECLARE EXIT HANDLER是用来处理异常的。意思是如果错误1215发生了，这个程序将会在错误记录表中插入一行。EXIT意思是当动作成功提交后退出这个复合语句。

5. Sample Problem: Log Of Errors

CALL p22 (5) //

调用这个存储过程会失败，这很正常，因为5值并没有在主键表中出现。但是没有错误信息返回因为出错处理已经包含在过程中了。t3表中没有增加任何东西，但是error_log表中记录下了一些信息，这就告诉我们 INSERT into table t3 动作失败。

DECLARE HANDLER syntax 声明异常处理的语法

DECLARE
{ EXIT | CONTINUE }
HANDLER FOR
{ error-number | { SQLSTATE error-string } | condition }
SQL statement

上面就是错误处理的用法，也就是一段当程序出错后自动触发的代码。MySQL允许两种处理器，一种是EXIT处理，我们刚才所用的就是这种。另一种就是我们将要演示的，CONTINUE处理，它跟EXIT处理类似，不同在于它执行后，原主程序仍然继续运行，那么这个复合语句就没有出口了。

1. DECLARE CONTINUE HANDLER example CONTINUE处理例子

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

这是MySQL参考手册上的CONTINUE处理的例子，这个例子十分好，所以我把它拷贝到这里。通过这个例子我们可以看出CONTINUE处理是如何工作的。

2. DECLARE CONTINUE HANDLER 声明CONTINUE异常处理

```
 CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;                    <--
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

这次我将为SQLSTATE值定义一个处理程序。还记得前面我们使用的MySQL错误代码1216吗？事实上这里的23000SQLSTATE是更常用的，当外键约束出错或主键约束出错就被调用了。

3. DECLARE CONTINUE HANDLER

```
 CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;                                          <--
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

这个存储过程的第一个执行的语句是"SET @x = 1"。

4. DECLARE CONTINUE HANDLER example

```
 CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);                          <--
 SET @x = 3;
 END;//
```

运行后值1被插入到主键表中。

5. DECLARE CONTINUE HANDLER

```
CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;                                         <--
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

然后@x的值变为2。

6. DECLARE CONTINUE HANDLER example

```
 CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
```

```
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);                    <--
 SET @x = 3;
 END;//
```

然后程序尝试再次往主键表中插入数值，但失败了，因为主键有唯一性限制。

7. DECLARE CONTINUE HANDLER example

```
 CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;           <--
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;
 END;//
```

由于插入失败，错误处理程序被触发，开始进行错误处理。下一个执行的语句是错误处理的语句，@x2被设为2。

8. DECLARE CONTINUE HANDLER example

```
 CREATE TABLE t4 (s1 int,primary key(s1));//
CREATE PROCEDURE p23 ()
BEGIN
 DECLARE CONTINUE HANDLER
 FOR   SQLSTATE '23000' SET @x2 = 1;
 SET @x = 1;
 INSERT INTO t4 VALUES (1);
 SET @x = 2;
 INSERT INTO t4 VALUES (1);
 SET @x = 3;                                   <--
 END;//
```

到这里并没有结束，因为这是CONTINUE异常处理。所以执行返回到失败的插入语句之后，继续执行将@x设定为3动作。

9. DECLARE CONTINUE HANDLER example

```
 mysql> CALL p23()//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x, @x2//
+------+------+
| @x   | @x2  |
+------+------+
| 3    | 1    |
+------+------+
1 row in set (0.00 sec)
```

运行过程后我们观察@x的值，很确定的可以知道是3， 观察@x2的值，为1。 从这里可以判断程序运行无误，完全按照我们的思路进行。大家可以花点时间去调整错误处理器，让检查放在语句段的首部，而不是放在可能出现 错误的地方，虽然那样看起来程序很紊乱，跳来跳去的感觉。但是这样的代码很安全也很清楚。

1. DECLARE CONDITION

```
 CREATE PROCEDURE p24 ()
BEGIN
  DECLARE `Constraint Violation`
   CONDITION FOR SQLSTATE '23000';
  DECLARE EXIT HANDLER FOR
   `Constraint Violation` ROLLBACK;
  START TRANSACTION;
  INSERT INTO t2 VALUES (1);
  INSERT INTO t2 VALUES (1);
  COMMIT;
  END; //
```

这是另外一个错误处理的例子，在前面的基础上修改的。事实上你可给SQLSTATE或者错误代码其他的名字，你就可以在处理中使用自己定义的名字了。下面看看它是怎么实现的：我把表t2定义为InnoDB表，所以对这个表的插入操作都会ROLLBACK（回滚），ROLLBACK（回滚事务）也是恰好会发生的。因为对主键插入两个同样的值会导致SQLSTATE 23000 错误发生，这里SQLSTATE 23000是约束错误。

2. DECLARE CONDITION 声明条件

```
CREATE PROCEDURE p24 ()
BEGIN
  DECLARE `Constraint Violation`
   CONDITION FOR SQLSTATE '23000';
  DECLARE EXIT HANDLER FOR
   `Constraint Violation` ROLLBACK;
  START TRANSACTION;
  INSERT INTO t2 VALUES (1);
  INSERT INTO t2 VALUES (1);
  COMMIT;
  END; //
```

这个约束错误会导致ROLLBACK（回滚事务）和SQLSTATE 23000错误发生。

3. DECLARE CONDITION

```
 mysql> CALL p24()//
Query OK, 0 rows affected (0.28 sec)

mysql> SELECT * FROM t2//
Empty set (0.00 sec)
```

我们调用这个存储过程看看结果是什么，从上面结果我们看到表t2没有插入任何记录。全部事务都回滚了。这正是我们想要的。

4. DECLARE CONDITION

```
 mysql> CREATE PROCEDURE p9 ()
    -> BEGIN
    ->     DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;
    ->     DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN END;
    ->     DECLARE EXIT HANDLER FOR SQLWARNING BEGIN END;
    -> END;//
Query OK, 0 rows affected (0.00 sec)
```

这里是三个预声明的条件： NOT FOUND (找不到行), SQLEXCEPTION (错误),

SQLWARNING (警告或注释)。因为它们是预声明的，因此不需要声明条件就可以使用。不过如果你去做这样的声明："DECLARE SQLEXCEPTION CONDITION ..."，你将会得到错误信息提示。

Cursors 游标

游标实现功能摘要：

DECLARE cursor-name CURSOR FOR SELECT ...;
OPEN cursor-name;
FETCH cursor-name INTO variable [, variable];
CLOSE cursor-name;

现在我们开始着眼游标了。虽然我们的存储过程中的游标语法还并没有完整的实现，但是已经可以完成基本的事务如声明游标，打开游标，从游标里读取，关闭游标。

1. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

我们看一下包含游标的存储过程的新例子。

2. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;                                    <--
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

这个过程开始声明了三个变量。附带说一下，顺序是十分重要的。首先要进行变量声明，然后声明条件，随后声明游标，再后面才是声明错误处理器。如果你没有按顺序声明，系统会提示错误信息。

3. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;                <--
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

程序第二步声明了游标cur_1，如果你使用过嵌入式SQL的话，就知道这和嵌入式SQL差不多。

4. Cursor Example

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND                    <--
    SET b = 1;                                              <--
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

最后进行的是错误处理器的声明。这个CONTINUE处理没有引用SQL错误代码和SQLSTATE值。它使用的是NOT FOUND系统返回值，这和SQLSTATE 02000是一样的。

5. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;                                               <--
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

过程第一个可执行的语句是OPEN　cur_1，它与SELECT s1 FROM t语句是关联的，　过程将执行SELECT s1 FROM t，返回一个结果集。

6. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;                                    <--
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

这里第一个FETCH语句会获得一行从SELECT产生的结果集中检索出来的值，然而表t中有多行，因此这个语句会被执行多次，当然这是因为语句在循环块内。

7. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;                                             <--
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//
```

最后当MySQL的FETCH没有获得行时，CONTINUE处理被触发，将变量b赋值为1。

8. Cursor Example

```
CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;                                             <--
  SET return_val = a;
END;//
```

到了这一步 UNTIL b=1 条件就为真，循环结束。在这里我们可以自己编写代码关闭游标，也可以由系统执行，系统会在复合语句结束时自动关闭游标，但是最好不要太依赖系统的自动关闭行为（译注：这可能跟Java的Gc一样，不可信）。

9. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;                                <--
END;//
```

这个例程中我们为输出参数指派了一个局部变量，这样在过程结束后的结果仍能使用。

10. Cursor Example

```
 CREATE PROCEDURE p25 (OUT return_val INT)
BEGIN
  DECLARE a,b INT;
  DECLARE cur_1 CURSOR FOR SELECT s1 FROM t;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET b = 1;
  OPEN cur_1;
  REPEAT
    FETCH cur_1 INTO a;
    UNTIL b = 1
  END REPEAT;
  CLOSE cur_1;
  SET return_val = a;
END;//

mysql> CALL p25(@return_val)//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @return_val//
+-------------+
| @return_val |
+-------------+
| 5           |
+-------------+
1 row in set (0.00 sec)
```

上面是过程调用后的结果。可以看到return_val参数获得了数值5，因为这是表t的最后一行。由此可以知道游标工作正常，出错处理也工作正常。

Cursor Characteristics 游标的特性

摘要：
READ ONLY 只读属性
NOT SCROLLABLE 顺序读取
ASENSITIVE 敏感

在5.0版的MySQL中，你只可以从游标中取值，不能对其进行更新。因为游标是（READ ONLY）只读的。你可以这样做：
FETCH cursor1 INTO variable1;
UPDATE t1 SET column1 = 'value1' WHERE CURRENT OF cursor1;

游标也是不可以滚动的，只允许逐一读取下一行，不能在结果集中前进或后退。下面代码就是错误的：
FETCH PRIOR cursor1 INTO variable1;
FETCH ABSOLUTE 55 cursor1 INTO variable1;

同时也不允许在已打开游标进行操作的表上执行updates事务，因为游标是（ASENSITIVE）敏感的。因为如果你不阻止update事务，那就不知道结果会变成什么。如果你使用的是InnoDB而不是MyISAM存储引擎的话，结果也会不一样。

Security 安全措施

摘要
Privileges (1) CREATE ROUTINE
Privileges (2) EXECUTE
Privileges (3) GRANT SHOW ROUTINE?
Privileges (4) INVOKERS AND DEFINERS

这里我们要讨论一些关于特权和安全相关的问题。但因为在MySQL安全措施的功能并没有完全，所以我们不会对其进行过多讨论。

2. Privileges CREATE ROUTINE

GRANT CREATE ROUTINE
ON database-name . *
TO user(s)
[WITH GRANT OPTION];
现在用root就可以了

在这里要介绍的特权是CREATE ROUTINE，它不仅同其他特权一样可以创建存储过程和函数，还可以创建视图和表。Root用户拥有这种特权，同时还有ALTER ROUTINE特权。

2. Privileges EXECUTE

GRANT EXECUTE ON p TO peter
[WITH GRANT OPTION];

上面的特权是决定你是否可以使用或执行存储过程的特权，过程创建者默认拥有这个特权。

3. Privileges SHOW ROUTINE?

GRANT SHOW ROUTINE ON db6.* TO joey
[WITH GRANT OPTION];

因为我们已经有控制视图的特权了：GRANT SHOW VIEW。所以在这个基础上，为了保证兼容，日后可能会添加GRANT SHOW ROUTINE特权。这样做是不太符合标准的，在写本书的时候，

MySQL还没实现这个功能。

4. Privileges Invokers and Definers 特权调用者和定义者

```
CREATE PROCEDURE p26 ()
  SQL SECURITY INVOKER
  SELECT COUNT(*) FROM t //
CREATE PROCEDURE p27 ()
  SQL SECURITY DEFINER
  SELECT COUNT(*) FROM t //
GRANT INSERT ON db5.* TO peter; //
```

现在我们测试一下SQL SECURITY子句吧。Security是我们前面提到的程序特性的一部分。你是root用户，将插入权赋给了peter。然后使用peter登陆进行新的工作，我们看看peter可以怎么使用 存储过程，注意：peter没有对表t的select权力，只有root用户有。

5. Privileges Invokers and Definers

```
/* Logged on with current_user = peter */ 使用帐户peter登陆

mysql> CALL p26();
ERROR 1142 (42000): select command denied to user
 'peter'@'localhost' for table 't'

mysql> CALL p27();
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

当peter尝试调用含有调用保密措施的过程p26时会失败。那是因为peter没有对表的select的权力。但是当petre调用含有定义保密措施的过程时就能成功。原因是root有select权力，Peter有root的权力，因此过程可以执行。

Functions 函数

Summary: 摘要
CREATE FUNCTION
Limitations of functions 函数的限制

我们已经很清楚可以在存储过程中使用的元素了。下面我要讲的是前面没有提到的函数。

CREATE FUNCTION 创建函数

```
CREATE FUNCTION factorial (n DECIMAL(3,0))
    RETURNS DECIMAL(20,0)
    DETERMINISTIC
  BEGIN
    DECLARE factorial DECIMAL(20,0) DEFAULT 1;
    DECLARE counter     DECIMAL(3,0);
    SET counter = n;
    factorial_loop: REPEAT
      SET factorial = factorial * counter;
      SET counter = counter - 1;
    UNTIL counter = 1
    END REPEAT;
    RETURN factorial;
  END //
```

（代码来源："Understanding SQL's stored procedures"，这里只是作为例子引用）函数跟过程很相似，唯一需要指出的语法上的不同就是创建函数后必须有RETURN语句返回函数指定的类型值。

这个例子来自Jim Melton的大作，他是SQL standard committee的成员，"Understanding SQL's stored procedures"的作者。原例在书的223页。我决定使用这个例子是因为它的规范性。

2. Examples

```
INSERT INTO t VALUES (factorial(pi)) //
SELECT s1, factorial (s1) FROM t //
UPDATE t SET s1 = factorial(s1)
WHERE factorial(s1) < 5 //
```

上面就是我们需要的函数，把它放到SQL语句中跟其他函数看起来是一样的。如果能很好的处理，函数将是美妙的，就像乐曲中的小调一样。不过，它们也有缺陷，那就是不能在函数中访问表，这使它们不如存储过程强大。

3. Limitations 限制

Illegal: 非法声明：
ALTER 'CACHE INDEX' CALL COMMIT CREATE DELETE
DROP 'FLUSH PRIVILEGES' GRANT INSERT KILL
LOCK OPTIMIZE REPAIR REPLACE REVOKE
ROLLBACK SAVEPOINT 'SELECT FROM table'
'SET system variable' 'SET TRANSACTION'
SHOW 'START TRANSACTION' TRUNCATE UPDATE

不能访问表的限制削弱了函数的功能，因此你不能够进行数据操作、数据描述、特权转化或是事务控制。但我们的工作主要是靠这些，也许以后会支持这些特性吧。

4. Limitations

合法声明：

'BEGIN END' DECLARE IF ITERATE LOOP
REPEAT RETURN 'SET declared variable'
WHILE

利用函数你能做的全部就是设置变量，然后在控制流语句中使用它们。实际上这个功能很强大，但是离人们想要的却还很远。

Metadata元数据

摘要：

SHOW CREATE PROCEDURE / SHOW CREATE FUNCTION
SHOW PROCEDURE STATUS / SHOW FUNCTION STATUS
SELECT from mysql.proc
SELECT from information_schema

到这里我们已经创建了很多过程了，它们也都保存在MySQL数据库中。我们如果要查看MySQL实际上保存了什么信息，有四种实现方法，两种使用SHOW语句，两种使用SELECT语句。

1. Show

```
mysql> show create procedure p6//
+-----------+----------+------------------+
| Procedure | sql_mode | Create Procedure   |
+-----------+----------+------------------+
| p6          |             | CREATE PROCEDURE   |
|               |             | `db5`.`p6`(out p   |
|               |             | int) set p = -5    |
+-----------+----------+------------------+
1 row in set (0.00 sec)
```

第一种获得元数据信息的方法是执行SHOW CREATE PROCEDURE。这同SHOW CREATE TABLE以及其他类似MySQL语句一样。它并不返回你创建过程时设定的返回值，但在大部分情况下已经够用了。

2. Show

```
mysql> SHOW PROCEDURE STATUS LIKE 'p6'//
+------+------+-----------+----------------+
| Db     | Name | Type         | Definer            | ...
+------+------+-----------+----------------+
| db5    | p6     | PROCEDURE | root@localhost | ...
+------+------+-----------+----------------+
1 row in set (0.01 sec)
```

第二种获得metadata信息的方法是执行SHOW PROCEDURE STATUS。这种方法可以返回更多信息的细节。

3. SELECT from mysql.proc

```
SELECT * FROM mysql.proc WHERE name = 'p6'//
+------+------+-----------+---------------+
| db     | name | type          | specific_name | ...
+------+------+-----------+---------------+
| db5    | p6     | PROCEDURE | p6                  | ...
+------+------+-----------+---------------+
1 row in set (0.00 sec)
```

第三种方法是执行SELECT语句，它能提供最多的信息。

4. SELECT from information_schema：我最喜欢的方式。

第四种方法是"SELECT ... FROM information_schema. ..."。我特别倾向使用"ANSI/ISO标准"的方式去完成工作。我相信这是种好实现方式，因为其他方式可能会出现错误。当然有不同MySQL用户会坚持不同的观点，也有不同理由，认真的持怀疑态度的看看这些理由。

1.其他DBMS，例如SQL Server 2000，使用information_schema。只有MySQL有SHOW方式。
2.我们访问mysql.proc的特权是没有保障的，因为我们有访问information_schema视图的特权，每个用户都有内隐的对information_schema数据库的SELECT特权。
3.SELECT功能很多，可以计算表达式，分组，排序，产生可以获取信息的结果集。而这些功能SHOW没有。

现在了解我喜欢它的原因了吧，那下面我们举几个简单的例子来演示一下。
首先我会使用SELECT information_schema来显示information_schema例程中有哪些列。

```
mysql> SELECT TABLE_NAME, COLUMN_NAME, COLUMN_TYPE FROM
INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_NAME = 'ROUTINES';//
+------------+--------------------+--------------+
| TABLE_NAME | COLUMN_NAME            | COLUMN_TYPE    |
+------------+--------------------+--------------+
| ROUTINES    | SPECIFIC_NAME        | varchar(64)    |
| ROUTINES    | ROUTINE_CATALOG      | longtext         |
| ROUTINES    | ROUTINE_SCHEMA       | varchar(64)    |
| ROUTINES    | ROUTINE_NAME         | varchar(64)    |
| ROUTINES    | ROUTINE_TYPE         | varchar(9)     |
| ROUTINES    | DTD_IDENTIFIER       | varchar(64)    |
| ROUTINES    | ROUTINE_BODY         | varchar(8)     |
| ROUTINES    | ROUTINE_DEFINITION | longtext         |
| ROUTINES    | EXTERNAL_NAME        | varchar(64)    |
| ROUTINES    | EXTERNAL_LANGUAGE   | varchar(64)    |
| ROUTINES    | PARAMETER_STYLE      | varchar(8)     |
| ROUTINES    | IS_DETERMINISTIC     | varchar(3)     |
| ROUTINES    | SQL_DATA_ACCESS      | varchar(64)    |
| ROUTINES    | SQL_PATH               | varchar(64)    |
| ROUTINES    | SECURITY_TYPE        | varchar(7)     |
| ROUTINES    | CREATED                | varbinary(19) |
| ROUTINES    | LAST_ALTERED         | varbinary(19) |
| ROUTINES    | SQL_MODE               | longtext         |
| ROUTINES    | ROUTINE_COMMENT      | varchar(64)    |
| ROUTINES    | DEFINER                | varchar(77)    |
+------------+--------------------+--------------+
20 rows in set (0.01 sec)
```

漂亮吧？当我们想要查看information_schema视图时，我们从information_schema中select信息，

就跟从TABLES和COLUMNS获取一样。获取的是元数据的数据元素。这里我们看到的是我在数据库db6中定义的存储过程。

```
mysql> SELECT COUNT(*) FROM INFORMATION_SCHEMA.ROUTINES
    -> WHERE ROUTINE_SCHEMA = 'db6';//
+----------+
| COUNT(*) |
+----------+
|          28 |
+----------+
1 row in set (0.02 sec)
```

现在进一步看看我们第一个创建的过程p1，我们重新格式mysql客户端输出窗口。

```
| SPECIFIC_NAME | ROUTINE_CATALOG | ROUTINE_SCHEMA
+---------------+-----------------+---------------
    p19               | NULL                | p19
| ROUTINE_NAME   | ROUTINE_TYPE      | DTD_IDENTIFIER
+---------------+-----------------+---------------
| p19               | PROCEDURE          | NULL
| ROUTINE_BODY | ROUTINE_DEFINITION | EXTERNAL_NAME
+--------------+--------------------+--------------
| SQL               | select * from t      | NULL
| EXTERNAL_LANGUAGE | PARAMETER_STYLE | IS_DETERMINISTIC
|-------------------+-----------------+-----------------
| NULL                    | SQL                   | NO
| SQL_DATA_ACCESS | SQL_PATH | SECURITY_TYPE | CREATED
+-----------------+----------+---------------+--------
| CONTAINS SQL      | NULL       | DEFINER          | 2004-12-19
| CREATED                 | LAST_ALTERED           | SQL_MODE
+---------------------+---------------------+---------
| 2004-12-19 15:00:26 | 2004-12-19 15:00:26 |
| ROUTINE_COMMENT | DEFINER
+-----------------+---------------
|                         | root@localhost
```

Access control for the ROUTINE_DEFINITION column

ROUTINE_DEFINITION列的访问控制

在INFORMATION_SCHEMA中的ROUTINE_DEFINITION列是由过程或函数组成过程体获得的。这里可能会有敏感信息，因此只对过程创建者可见。

CURRENT_USER <> INFORMATION_SCHEMA.ROUTINES.DEFINER ： 如果对它使用SELECT的用户不是创建它的用户，那么mysql将返回NULL值，而不是ROUTINE_DEFINITION列。这个检查功能在作此书时还没实现。

Additional clause in SHOW PROCEDURE STATUS显示过程状态子句

SHOW PROCEDURE STATUS中的辅助子句

既然我已经列出INFORMATION_SCHEMA.ROUTINES中的列，就可以回去解释SHOW PROCEDURE STATUS的新细节，语法是：

```
SHOW PROCEDURE STATUS [WHERE condition];
```

语句中的条件判断和SELECT语句的一样：如果为真，则在输出中返回行。但这里有个需要

特别注意的部分：在WHERE子句中你必须使用INFORMATION_SCHEMA列的名字，结果显示的是SHOW PROCEDURE STATUS字段的名字。例如：

```
mysql> SHOW PROCEDURE STATUS WHERE Db = 'p';
ERROR 1054 (42S22): Unknown column 'Db' in 'where clause'

mysql> SHOW PROCEDURE STATUS WHERE ROUTINE_NAME = 'p';
+------+------+-----------+----------------+
| Db   | Name | Type      | Definer        | ...
+------+------+-----------+----------------+
| db11 | p    | PROCEDURE | root@localhost | ...
+------+------+-----------+----------------+
1 row in set (0.00 sec)
```

知道了这点，我们也许就能在MySQL认证考试中获得高分，但是实际中从来不会用到。

Details 细节

后面几页的摘要：

ALTER and DROP
Oracle / SQL Server / DB2 / ANSI comparison
Style
Bugs
Feature Requests
Resources

最后的环节中我们介绍了很多细节，但不会根据它们应该受到注意的程度去学习，这没关系，因为在试错中你会发现这些。

ALTER and DROP

```
ALTER PROCEDURE p6 COMMENT 'Unfinished' //
DROP PROCEDURE p6 //
```

Oracle Comparison 与Oracle的比较

Summary: 摘要

- Oracle允许在打开后再声明。
  MySQL必须在开始的时候声明。

Oracle允许"CURSOR cursorname IS"这样的声明。
MySQL必须使用"DECLARE cursorname CURSOR"声明。

Oracle不强制需要'()'。
MySQL必须有'()'。

Oracle允许在函数中访问表元素。
MySQL不允许在函数中访问表元素。

Oracle支持"packages"。
MySQL不支持"packages"。

如果你使用过Oracle的PL/SQL存储过程的话，你会发现Oracle和MySQL的存储过程有很多不同（译者语：PL/SQL的功能比MySQL的存储过程强大，也许这就是区别，哈哈）。我刚才指出的只是一些常见的区别。
你也可以去http://www.ispirer.com网站阅读更多的资料，这是一家提供将Oracle存储过程转换成MySQL存储过程的产品的厂家。我并没有说这家公司的产品很好的意思， 因为我没有用过。我只是感兴趣已经对有人做出将其他DBMS的存储过程迁移到MySQL上来的程序。

Tips for migration..数据迁移的技巧

把"a:=b"类似的赋值语句改成"SET a=b"。
将过程中的RETURN语句改为"LEAVE label_at_start"，这里的label_at_start是你最初为存储过程设定的标记。如：

[在Oracle存储过程中]
CREATE PROCEDURE ... RETURN; ...
[在MySQL存储过程中]
CREATE PROCEDURE () label_at_start: BEGIN ... LEAVE label_at_start; END

这一步骤仅仅在过程中需要，因为MySQL函数支持RETURN。

Side-By-Side 平行比较

Oracle	MySQL
CREATE PROCEDURE	CREATE PROCEDURE
sp_name	sp_name
AS	BEGIN
variable1 INTEGER	DECLARE variable1 INTEGER;
variable1 := 55	SET variable1 = 55;
END	END

与SQL Server的对比

摘要：

SQL Server参数名字必须以'@'开头。
MySQL参数名是常规标识符。

SQL Server可以同时进行多个声明，如："DECLARE v1 [data type], v2 [data type]"。
MySQL只允许每次声明一个，如："DECLARE v1 [data type]; DECLARE v2 [data type]"。

SQL Server存储过程体中没有BEGIN ... END。
MySQL必须有BEGIN ... END语句。

SQL Server不需要以';'号结束语句。
MySQL必须使用';'号作为语句结束标志，除了最后一条语句外。

SQL Server可以进行"SET NOCOUNT"设置和"IF @@ROWCOUNT"判断，
MySQL没有这些，但可以使用FOUND_ROWS()进行判断。

SQL Server中使用"WHILE ... BEGIN"语句。
MySQL使用"WHILE ... DO"语句。

SQL Server允许使用"SELECT"进行指派。
MySQL只允许SET进行指派。

SQL Server允许在函数中访问表。
MySQL不允许在函数中访问表

Microsoft SQL Server的区别特别多，所以讲Microsoft或Sybase的程序转换成MySQL程序将会是个冗长的过程，而且区别都是在语法定义上的，所以转换需要更多特别的技巧。

Some migration tips ... 一些迁移技巧

如果SQL Server中有名为@xxx的变量，你必须将其转换，因为@在MySQL中并不代表过程变量，

而是全局变量，不要仅仅改成xxx，那样会使其含义不明确。因为在数据库的某个表中可能有一列的列名叫xxx，所以最好把@前缀改成你的自定义字符，如将@xxx改成var_xxx。

3. Side by Side 平行对比

SQL Server	**MySQL**
CREATE PROCEDURE	CREATE PROCEDURE
sp_procedure1	sp_procedure1
AS	()
DECLARE @x VARCHAR(100)	BEGIN
EXECUTE sp_procedure2 @x	DECLARE v__x VARCHAR(100);
DECLARE c CURSOR FOR	CALL sp_procedure2(v__x);
DECLARE c CURSOR FOR	SELECT * FROM t;
SELECT * FROM t	END
END	

与DB2的对比

DB2允许PATH（路径）语句。
MySQL不允许PATH（路径）语句。

DB2允许SIGNAL（信令）语句。
MySQL不允许SIGNAL（信令）语句。

DB2允许例程名的重载。
MySQL不允许对例程名的重载。

DB2有"label_x: ... GOTO label_x"语法。
MySQL有非正式的"label label_x; ... GOTO label_x"语法。

DB2允许函数访问表。
MySQL不允许函数访问表。

DB2存储过程基本和MySQL一致，唯一的不同在于MySQL还没有引进DB2的一些语句，还有就是DB2允许重载，因此DB2可以有两个名字一样的例程，通过例程的参数或返回类型来决定执行哪个，所以DB2存储过程可以向下与MySQL的兼容。

Some migration tips ... 一些迁移技巧
这里的迁移基本不需要任何技巧。MySQL缺少SIGNAL语句，我们会在其他地方讨论临时工作区的问题。而对DB2的GOTO语句，我们直接用MySQL的GOTO代替就可以了。PATH（确定DBMS寻找例程的数据库目录）问题只需要在例程名前加上前缀就可以避免了。关于函数访问表的问题，我建议大家用OUT参数的存储过程来代替就行了。

Side by Side 平行对比

DB2	**MySQL**
CREATE PROCEDURE	CREATE PROCEDURE
sp_name	sp_name
(parameter1 INTEGER)	(parameter1 INTEGER)
LANGUAGE SQL	LANGUAGE SQL
BEGIN	BEGIN
DECLARE v INTEGER;	DECLARE v INTEGER;
IF parameter1 >=5 THEN	IF parameter1 >=5 THEN
CALL p26();	CALL p26();
SET v = 2;	SET v = 2;
END IF;	END IF;
INSERT INTO t VALUES (v);	INSERT INTO t VALUES (v);
END @	END //

Standard Comparison 与SQL标准的比较

摘要：

Standard SQL requires: 标准SQL的要求
[跟DB2中的一样]

MySQL的目标是支持以下两个标准SQL特性：
Feature P001 "Stored Modules"（ 特性 P001 “存储模式”）
Feature P002 "Computational completeness"（ 特性 P002 “计算完整性”）

DB2和MySQL相似的原因是两者都支持标准SQL中的存储过程。因此MySQL和DB2的区别就像我们背离ANSI/ISO标准语法那样。但是我们比Oracle或SQL Server更标准。

Style 编程风格

```
CREATE PROCEDURE p ()
BEGIN
  /* Use comments! */
  UPDATE t SET s1 = 5;
  CREATE TRIGGER t2_ai ...
END;//
```

在这个例子中，我们使用了一种编程风格来写这个存储过程。关键字要大写。在命名约定中，我认为在手册中表名最好为“t”什么的，列名为“s”什么的。 实际上作为DBA，我们可以参考文章"SQL Naming Conventions"（来自http://dbazine.com/gulutzan5.shtml）。

注释和C语言中的一样：

在BEGIN后缩进（一般是一个TAB字符）。
在END前回缩（使END语句与BEGIN前的语句齐平），事实上我不喜欢这个细节。我更喜欢下面这样：

```
CREATE PROCEDURE p ()
BEGIN
  /* Use comments! */
  UPDATE t SET s1 = 5;
  CREATE TRIGGER t2_ai ...
  END;//                                    <--
```

就是当在END后回缩，而不是之前。在这里我并不是要大家都跟随我的风格，而是希望大家都有自己的风格，然后能在编写存储过程中坚持下去。这会使你的代码更好阅读和维护。

下面是一些函数和过程的例子，大家可以作为参考学习：

Stored Procedure Example: tables_concat() 字符串连接的函数

这是把所有表名连接到一个单一字符串的函数，可以和MySQL内建的group_concat()函数对比一下。

```
CREATE PROCEDURE tables_concat
(OUT parameter1 VARCHAR(1000))
BEGIN
  DECLARE variable2 CHAR(100);
  DECLARE c CURSOR FOR
  SELECT table_name FROM information_schema.tables;
  DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;      /* 1 */
  SET sql_mode='ansi';                               /* 2 */
  SET parameter1 = '';
  OPEN c;
  LOOP
    FETCH c INTO variable2;                          /* 3 */
    SET parameter1 = parameter1 || variable2 || '.';
    END LOOP;
  CLOSE c;
  END;
```

/* 1 */: 这里的"BEGIN END"语句没有任何作用，就像其他DBMS中的NULL语句。
/* 2 */: 将sql_mode设置为'ansi'以便"||"能正常连接，在退出存储过程后sql_mode仍为'ansi'。
/* 3 */: 另一种跳出循环LOOP的方法：声明EXIT出错处理，当FETCH没有返回行时。

以下就是我们调用这个过程的结果：

```
mysql> CALL tables_concat(@x);
Query OK, 0 rows affected (0.05 sec)

mysql> SELECT @x;
+-------------------------------------------------+
| SCHEMATA.TABLES.COLUMNS.CHARACTER_SETS.COLLATIONS.C
OLLATION_CHARACTER_SET_APPLICABILITY.ROUTINES.STATIST
ICS.VIEWS.USER_PRIVILEGES.SCHEMA_PRIVILEGES.TABLE_PRI
VILEGES.COLUMN_PRIVILEGES.TABLE_CONSTRAINTS.KEY_COLUM
N_USAGE.TABLE_NAMES.columns_priv.db.fn.func.help_cate
gory.help_keyword.help_relation.help_topic.host.proc.
tables_priv.time_zone.time_zone_leap_second.time_zone
1 row in set (0.00 sec)
```

下面过程的目的是获得整型包含行的数量的结果集，类似其他DBMS中的ROWNUM()。我们需要一个用户变量来保存在每次调用rno()后的结果，就命名为@rno吧。

```
CREATE FUNCTION rno ()
RETURNS INT
BEGIN
  SET @rno = @rno + 1;
  RETURN @rno;
  END;
```

通过rno()方法的SELECT我们获得了行数。下面是调用程序的结果：

```
mysql> SET @rno = 0;//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT rno(),s1,s2 FROM t;//
+-------+------+------+
| rno() | s1   | s2   |
+-------+------+------+
|     1 |    1 | a    |
|     2 |    2 | b    |
|     3 |    3 | c    |
|     4 |    4 | d    |
|     5 |    5 | e    |
+-------+------+------+
5 rows in set (0.00 sec)
```

在SELECT中将@rno置零的技巧是使用了WHERE的求值功能，而这个特性在今后的MySQL中可能丢失。

```
CREATE FUNCTION rno_reset ()
RETURNS INTEGER
BEGIN
  SET @rno = 0;
  RETURN 1;
  END;
SELECT rno(),s1,s2 FROM t WHERE rno_reset()=1;//
```

Function Example: running_total()

这个累加的函数建立在rno()基础上。不同在于我们要在每次调用时中传值到参数中。

```
CREATE FUNCTION running_total (IN adder INT)
RETURNS INT
 BEGIN
 SET @running_total = @running_total + adder;
 RETURN @running_total;
 END;
```

下面是调用函数后的结果：

```
mysql> SET @running_total = 0;//
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT s1,running_total(s1),s2 FROM t ORDER BY s1;//
+------+------------------+------+
| s1   | running_total(s1) | s2   |
+------+------------------+------+
|    1 |                1 | a    |
|    2 |                3 | b    |
|    3 |                6 | c    |
|    4 |               10 | d    |
|    5 |               15 | e    |
+------+------------------+------+
5 rows in set (0.01 sec)
```

running_total()函数在ORDER BY完成后调用，但这样写既不标准也不能被移植。

Procedure Example: MyISAM "Foreign Key" insertion（ MyISAM外键插入）

MyISAM存储引擎不支持外键，但是你可以将这个逻辑加入存储过程引擎进行检查。如下例：

```
CREATE PROCEDURE fk_insert (p_fk INT, p_animal VARCHAR(10))
BEGIN
  DECLARE v INT;
  BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, NOT FOUND
    SET v = 0;
    IF p_fk IS NOT NULL THEN
      SELECT 1 INTO v FROM tpk WHERE cpk = p_fk LIMIT 1;
      INSERT INTO tfk VALUES (p_fk, p_animal);
    ELSE
      SET v = 1;
    END IF;
  END;
  IF v <> 1 THEN
    DROP TABLE `The insertion failed`;
  END IF;
END;
```

注意：SQLEXCEPTION或NOT FOUND条件都会导致v变成0，如果这些条件为假，则v会变成1，因为SELECT会给v赋值1，而EXIT HANDLER没有运行。下面看看运行结果：

```
mysql> CREATE TABLE tpk (cpk INT PRIMARY KEY);//
Query OK, 0 rows affected (0.01 sec)
mysql> CREATE TABLE tfk (cfk INT, canimal VARCHAR(10));//
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tpk VALUES (1),(7),(10);//
Query OK, 3 rows affected (0.01 sec)
Records: 3   Duplicates: 0   Warnings: 0

mysql> CALL fk_insert(1,'wombat');//
Query OK, 1 row affected (0.02 sec)

mysql> CALL fk_insert(NULL,'wallaby');//
Query OK, 0 rows affected (0.00 sec)
mysql> CALL fk_insert(17,'wendigo');//
ERROR 1051 (42S02): Unknown table 'The insertion failed'
```

Procedure Example: Error Propagation 错误传递

设想如果过程1调用过程2，过程2调用了过程3，过程3中的错误就会传递到过程1。如果没有异常处理器捕获异常，那异常就会传递，导致过程2出错，最后导致过程1出错，最终异常传递到了调用者（MySQL客户端实例）。这种特性使得标准SQL中存在SIGNAL语句来使异常强制发生，其他DBMS中也有类似措施（RAISEERROR）。MySQL还不支持SIGNAL，直到支持此特性的新版本推出，大家就可以用SIGNAL来进行处理了。大家也可以用下面的异常处理方式。

```
 CREATE PROCEDURE procedure1 ()
BEGIN
  CALL procedure2();
  SET @x = 1;
END;
CREATE PROCEDURE procedure2 ()
BEGIN
  CALL procedure3();
  SET @x = 2;
END;
CREATE PROCEDURE procedure3 ()
  BEGIN
  DROP TABLE error.`error #7815`;
  SET @x = 3;
END;
```

调用过程1后结果如下：

```
mysql> CALL procedure1()//
ERROR 1051 (42S02): Unknown table 'error #7815'
```

@x并没有改变，因为没有一条"SET @x = ..."语句成功运行，而使用DROP可以产生一些可供诊断的错误信息。不过，在这里我们最好祈祷没有名字叫做`error`的表存在。

Procedure Example: Library 库

对库的应用有详细的规格说明。我们希望拥有权限的用户都能调用过程，所以我们这样设置：

GRANT ALL ON database-name.* TO user-name;

如果要其他用户只有访问使用过程的权限，没有问题，只需要定义SQL SECURITY DEFINER特性就可以了，而这个选项是默认的，但最好显式的声明出来。

下面是一个向数据库中添加书本的过程，这里必须测试书的id是否确定，书名是否为空。例子是对MySQL不支持的CHECK限制功能的替代。

```
CREATE PROCEDURE add_book
(p_book_id INT, p_book_title VARCHAR(100))
SQL SECURITY DEFINER
BEGIN
 IF p_book_id < 0 OR p_book_title='' THEN
    SELECT 'Warning: Bad parameters';
    END IF;
 INSERT INTO books VALUES (p_book_id, p_book_title);
 END;
```

我们需要一个添加买主的过程，过程必须检查是否有超过一个的买主，如有则给出警告。
这个功能可以在一个子查询中完成，如下：

```
IF (SELECT COUNT(*) FROM table-name) > 2) THEN ... END IF;
```

不过，在写作此书时子查询功能有漏洞，于是我们用"SELECT COUNT(*) INTO variable-name"代替。

```
CREATE PROCEDURE add_patron
(p_patron_id INT, p_patron_name VARCHAR(100))
SQL SECURITY DEFINER
BEGIN
  DECLARE v INT DEFAULT 0;
  SELECT COUNT(*) FROM patrons INTO v;
  IF v > 2 THEN
    SELECT 'warning: already there are ',v,'patrons!';
  END IF;
  INSERT INTO patrons VALUES (p_patron_id,p_patron_name);
END;
```

下面我们需要书本付帐的过程。在事务处理过程中我们希望显示已经拥有本书的买主，以及这些买主拥有的书，这些信息可以通过对游标CURSOR的Fetch来获得，我们会使用两种不同的方法来测试是否fetch数据已经完毕，第一种是检查变量在fetch动作后是否是NULL，第二种是通过NOT FOUND错误处理捕获fetch的失败动作。如果两个游标在不同的BEGIN/END块中程序看起来会显得整洁，但我们要在主BEGIN/END块 中声明这些变量，这样做才能使它们的作用域覆盖整个过程。

```
CREATE PROCEDURE checkout (p_patron_id INT, p_book_id INT)
SQL SECURITY DEFINER
BEGIN
  DECLARE v_patron_id, v_book_id INT;
  DECLARE no_more BOOLEAN default FALSE;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more=TRUE;
  BEGIN
    DECLARE c1 CURSOR FOR SELECT patron_id
    FROM transactions WHERE book_id = p_book_id;
```

```
  OPEN c1;
  SET v_patron_id=NULL;
  FETCH c1 INTO v_patron_id;
  IF v_patron_id IS NOT NULL THEN
    SELECT 'Book is already out to this patron:',
    v_patron_id;
    END IF;
  CLOSE c1;
  END;
BEGIN
  DECLARE c2 CURSOR FOR SELECT book_id
  FROM transactions WHERE patron_id = p_patron_id;
  OPEN c2;
  book_loop: LOOP
    FETCH c2 INTO v_book_id;
    IF no_more THEN
      LEAVE book_loop;
       END IF;
    SELECT 'Patron already has this book:', v_book_id;
     END LOOP;
   END;
 INSERT INTO transactions VALUES (p_patron_id, p_book_id);
 END;
```

Procedure Example: Hierarchy (I) 分层次

hierarchy()过程实现的是其他DBMS中CONNECT BY部分功能。我们拥有一个Persons表，表中的后代通过person_id列与祖先相连。我们通过参数start_with传递列表的第一个人。然后按顺序显示祖先和后代。这个过程（实际是两个）代码在后面两页，相信大家学了那么多，阅读得仔细点也可以自己跟上的，不过开始我还是会给一些说明性的注释。hierarchy()过程接受person的名字作为输入参数，作用有点像初始化动作，不过重要的是它建立了个临时表，用来存储查找到的结果行的数据。然后它调用了hierarchy2()过程，此时hierarchy2()过程开始进行循环，不停的调用自身。如果父亲只有1或0个儿子，这一步就不需要了，但事实上不会如此，所以要查询到每个分支，因此要对树进行不断的循环查询，直到最后节点才返回分支点进行另一个分支的查询。我们这里还在每条SQL语句执行后使用了错误处理set a flag (error)，如果语句失败，程序使用"SELECT 'string'"输出诊断信息，然后离开出错的过程。

同时在这里还使用了复合语句的嵌套（就是在BEGIN END块中放置BEGIN END块），所以我们才可以为关联特定语句的变量和出错处理进行声明。不过记住在外层复合语句中的声明在内层语句中仍有效，除非你使用了重载，还有就是在内层复合语句完毕后，内层的声明就失效。

下一页是hierarchy()过程的创建代码，后页则是hierarchy2()过程的，在此之后的是成功调用hierarchy()过程的结果。

```
CREATE PROCEDURE hierarchy (start_with CHAR(10))
proc:
BEGIN
  DECLARE temporary_table_exists BOOLEAN;
  BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN END;
    DROP TABLE IF EXISTS Temporary_Table;
  END;
  BEGIN
    DECLARE v_person_id, v_father_id INT;
    DECLARE v_person_name CHAR(20);
    DECLARE done, error BOOLEAN DEFAULT FALSE;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET error = TRUE;
    CREATE TEMPORARY TABLE Temporary_Table
    (person_id INT, person_name CHAR(20), father_id INT);
    IF error THEN
      SELECT 'CREATE TEMPORARY failed'; LEAVE proc; END IF;
    SET temporary_table_exists=TRUE;
    SELECT person_id, person_name
    INTO v_person_id, v_person_name FROM Persons
    WHERE person_name = start_with limit 1;
    IF error THEN
      SELECT 'First SELECT failed'; LEAVE proc; END IF;
    IF v_person_id IS NOT NULL THEN
      INSERT INTO Temporary_Table VALUES
      (v_person_id, v_person_name, v_father_id);
      IF error THEN
      SELECT 'First INSERT failed'; LEAVE proc; END IF;
      CALL hierarchy2(v_person_id);
      IF error THEN
      SELECT 'First CALL hierarchy2() failed'; END IF;
      END IF;
    SELECT person_id, person_name, father_id
    FROM Temporary_Table;
    IF error THEN
    SELECT 'Temporary SELECT failed'; LEAVE proc; END IF;
  END;
  IF temporary_table_exists THEN
    DROP TEMPORARY TABLE Temporary_Table;
    END IF;
END;
```

```
CREATE PROCEDURE hierarchy2 (start_with INT)
proc:
BEGIN
  DECLARE v_person_id INT, v_father_id INT;
  DECLARE v_person_name CHAR(20);
  DECLARE done, error BOOLEAN DEFAULT FALSE;
  DECLARE c CURSOR FOR
  SELECT person_id, person_name, father_id
  FROM Persons WHERE father_id = start_with;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  SET error = TRUE;
  OPEN c;
  IF error THEN
  SELECT 'OPEN failed'; LEAVE proc; END IF;
  REPEAT
    SET v_person_id=NULL;
    FETCH c INTO v_person_id, v_person_name, v_father_id;
    IF error THEN
    SELECT 'FETCH failed'; LEAVE proc; END IF;
    IF done=FALSE THEN
      INSERT INTO Temporary_Table VALUES
      (v_person_id, v_person_name, v_father_id);
      IF error THEN
      SELECT 'INSERT in hierarchy2() failed'; END IF;
    CALL hierarchy2(v_person_id);
    IF error THEN
    SELECT 'Recursive CALL hierarchy2() failed'; END IF;
    END IF;
    UNTIL done = TRUE
  END REPEAT;
  CLOSE c;
  IF error THEN
  SELECT 'CLOSE failed'; END IF;
END;
```

下面是调用hierarchy()后的结果：

```
mysql> CREATE TABLE Persons (person_id INT, person_name CHAR(20), father_id INT);//
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO Persons VALUES (1,'Grandpa',NULL);//
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO Persons VALUES (2,'Pa-1',1),(3,'Pa-2',1);//
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> INSERT INTO Persons VALUES (4,'Grandson-1',2),(5,'Grandson-2',2);//
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> call hierarchy('Grandpa')//
+-----------+-------------+-----------+
| person_id | person_name | father_id |
+-----------+-------------+-----------+
|         1 | Grandpa     |      NULL |
|         2 | Pa-1        |         1 |
|         4 | Grandson-1  |         2 |
|         5 | Grandson-2  |         2 |
|         3 | Pa-2        |         1 |
+-----------+-------------+-----------+
5 rows in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

查询从祖先一直查到孙子，然后又从儿子节点开始查询，使用的是“深度优先”算法。也许hierarchy()过程过于复杂，会有些我没考虑到的漏洞。循环太深的话也许会出错，试想如果有100代，则当有人是自己的爷爷时（这里是直译的，不过大家也能想到的，就是其中树的节点互相缠绕，形成一个环），就会进入无限循环。不过我在这里只是为了说明MySQL也可通过对存储过程的循环实现CONNECT BY功能。

Tips when writing long routines 书写长例程时的技巧

MySQL没有存储过程调试器，因此输出的错误信息大多可能没用。下面是我在创建一个20行的过程时进行异常处理的方法：

打开文本编辑器，将语句拷贝到编辑器窗口，使用copy和paste功能在文本编辑器和MySQL窗口间切换很方便。

如果是语法错误，可以每次去掉一句进行排错。这比你光用眼睛看程序代码效率高多了，因为看的时候似乎一切都没错。

如果是运行时错误，则在每条可执行的语句后加入"SELECT n;"（n可以是0、1、2来标示运行到了哪条语句），调用后你就可以追中控制流来进行诊断了。

如果是数据错误，则可以通过添加"DECLARE CONTINUE HANDLER FOR ... BEGIN END;"来跳过，因为往往测试时数据库的数据不存在，因此跳过这个使我们可以继续进行。

Bugs 漏洞或缺陷

MySQL仍然是alpha版（译者语：软件中alpha版意思即内测，beta则是外测，翻译时已出beta版），其存储过程中仍有许多严重漏洞，我计算的2004-10-27和2004-12-14的漏洞如下：

Bug Count On October 27 2004:

Category 种类	Count
Causes Operating System To Reboot （导致系统重启）	1
Crashes or Hangs （导致崩溃或挂起）	14
Returns 'Packets out of order' （返回'Packets out of order'）	5
Fails	21
	--
	41

Bug Count On December 14 2004:

Category	Count
Causes Operating System To Reboot	1
Crashes or Hangs	23
Returns 'Packets out of order'	6
Fails	31
	--
	61

你可以通过访问MySQL漏洞发布网站来查询存储过程和触发器的错误，下面是步骤：

1. 打开网址http://bugs.mysql.com
2. 点击"Advanced Search"
3. 在"Find bugs with any of the words"框中输入"procedure* trigger*".
4. 在下拉框中默认为返回10个错误，然活选择"All"
5. 点击"Boolean mode"
6. 最后是点击"Search"

Feature Requests 将会出现的特性

摘要：
SIGNAL
PATH
Accessing Tables in Functions or Triggers 在函数或触发器中访问表
BEGIN ATOMIC ... END
Encrypted storage 加密存储
Timeout 超时
Debugger 调试器
External Languages 外部语言特性
DROP ... CASCADE|RESTRICT

下面是我们急切需要添加的功能：
我们函数最需要的特性是拥有DB2和ANSI/ISO标准的强大功能，同时还需要超时功能使运行时间过长的循环或动作停止。

Resources 您可以获得的资源

(I) MySQL的网站
(II) MySQL可供下载的资源
(III) 教程和参考手册

在书的最后我将告诉读者获得更多信息的方法和地方。

MySQL网站

MySQL参考手册存储过程章节：
http://www.mysql.com/doc/en/Stored_Procedures.html

“MySQL存储过程”最新文章：
http://www.mysql.com/newsletter/2004-01/a0000000297.html

MySQL用户会议的幻灯片：
mysql.mirror.anlx.net/Downloads/Presentations/MySQL-User-Conference-2003/MySQL-Stored-Procedures.pdf

这些都是你可以从网站获取的资源。

MySQL下载

```
> cd ~/mysql-5.0/mysql-test/t
> dir sp*
   10025 2004-10-23 05:19 sp-error.test
    3974 2004-10-23 05:19 sp-security.test
     754 2004-08-07 08:51 sp-threads.test
   45344 2004-10-24 06:09 sp.test
> cd ~/mysql-5.0/Docs
> dir sp*
   41909 2004-08-05 09:19 sp-imp-spec.txt
    4948 2004-08-05 09:19 sp-implemented.txt
```

如果需要更多的信息，你可以下载MySQL 5.0源代码包。当然我不是让读者去读代码本身，而是去阅读mysql-test目录里的测试脚本，在Docs目录里有文档资料和新闻。（上面有操作过程）

Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM （书名，不翻译了）
作者Author: Jim Melton
发行Publisher: Morgan Kaufmann
Only available second-hand.

这是一本关于标准SQL存储过程的参考书，MySQL也尽量符合SQL标准，因此作为MySQL用户的您最好能有一本作为参考。

Conclusion结束语

到了最后我就不再去复述些什么概念了，相信大家可以记住全部的。如果您喜欢这本书，你可以在我们的网站找到更多"MySQL 5.0 New Features"系列书籍，下一本书将会是关于“触发器”和“视图”。

感谢您的关注，如果您对这本书有什么评论欢迎来MySQL论坛：http://forums.mysql.com/。

About MySQL （这部分就不翻译了，估计大家也不看，呵呵）

MySQL AB develops and supports a family of high performance, affordable database servers and tools. The company's flagship product is MySQL, the world's most popular open source database, with more than six million active installations. Many of the world's largest organizations, including Yahoo!, Sabre Holdings, The Associated Press, Suzuki and NASA are realizing significant cost savings by using MySQL to power high-volume Web sites, business-critical enterprise applications and packaged software.
With headquarters in Sweden and the United States ¨C and operations around the world ¨C MySQL AB supports both open source values and corporate customers' needs in a profitable, sustainable business. For more information about MySQL, please visit www.mysql.com.

译者的话：这是第一次做这样的尝试，所以挑选了这本比较入门的书来翻译，目的是对国内MySQL用户的支持，里面可能会有不少的错误，希望大家可以原谅里面的错误，但是我保证，错误只会是文法上的，绝对不会影响大家的操作。谢谢各位支持，转载时请保留作者信息和声明。这是Peter Gulutzan先生大作——《MySQL 5.0新特性》系列的第一册，以后我会抽空翻译此系列的其他资料，当然大家如果需要什么资料也可以给我写信，如果时间充足我会给您回复翻译，希望得到各位支持和鼓励，谢谢大家的阅读和宝贵意见。

作者：陈朋奕
毕业院校：西安电子科技大学
爱好：Java，JSP、Oracle PL/SQL，MySQL
邮箱：chenpengyi_007@163.com
地址：西安电子科技大学96＃113（可能很快就走了……）