

MySQL 存储过程编程

原著: Steven Feuerstein, Guy Harrison

译者: drekey (at) gmail dot com

出版: O'Reilly

出版日期: 2006 年五月

ISBN-10: 0-596-10089-2

ISBN-13: 978-0-59-610089-6

页数: 636 页

概览

MySQL5.0 中存储过程的实现作为早已非常著名 mysql 数据库的一部分,在使得 mysql 成为广泛的企业级应用的领头羊的过程中有着里程碑的意义。如果你打算严肃的来面对建立基于 web 的数据库应用程序的将来,你必须快速的掌握怎样用存储过程进行工作(并且用正确的方法建立他们),这本书的目的就是打算成为一本存储过程编程的圣经,作为一种资源,来共同面对这一 MySQL 程序员所无法逃避的现状。

在 MySQL 突如其来降生的靠十年中,它已然是现今具有统治地位的开源数据库,无论是性能和表现都能和那些诸如 Oracle 和 SQL Server 这样的商用 RDBMS(关系型数据库)相竞争。并且 MySQL 能够和 Linux 和 php 结合在一起,成为数百万应用程序的心脏,

现在,MySQL5.0 结合了对于存储过程,函数和触发器的支持,提供了作为真正企业级应用的编程能力。

MySQL 新的过程语言有着直白化的语法,我们可以用它很容易的写出简单的程序,但是要写出安全,容易维护,高效并且容易调试的程序却并非易事。作为一个新生事务,还很少有人能在 MySQL 领域拥有对存储过程实质性的编程经验,但是 Guy Harrison 和 Steven Feuerstein 正是这极少数的一员,他们在这专业领域拥有靠十年的经验

在 MySQL 存储过程编程这本书中,作者们很好的利用了自身积累的难得的经验,他们将充满代码的示例和各种语言级的基础知识融入进应用程序的构建所需要的各种协调和实践中,使之成为一本高可阅读性的 MySQL 开发一站式指南,这本书由以下四个方面组成:

- MySQL 存储过程编程基础 -- 指南,基本语句,存储过程中的 SQL 和错误处理
- 创建 MySQL 存储过程程序 -- 事务处理,内建函数,存储过程函数和触发器
- 在应用程序中使用 MySQL 存储过程程序 -- 在 PHP, Java, Perl, Python 和 .NET(C# 和 VB.NET)中使用存储过程编程
- 优化 MySQL 存储过程程序 -- 安全性,基本和高级 SQL 调试,优化存储过程代码,并且进行最有效的编程实践

这本书的主题网站拥有数千行的代码,方便你随时运行

Guy Harrison 是 Quest 软件公司数据库解决方案的总构架师,并且有大量关于 MySQL 的主题演说和文章,Steven Feuerstein 是 Oracle PL/SQL 编程(Oracle PL/SQL Programming)一书的作者,在 Oracle 存储过程编程工作超过十年,他们俩都在数据库开发领域拥有十年以上的经验,并且出版了大量的书籍

MySQL 存储过程编程所受到的赞誉

“我不太相信以这个话题为中心的书会写的不得干瘪，但 Guy 和 Steven 在这方面显示出的深度和提供给读者的资料，让我相信这是一本非常棒的书”

--Brian Aker, MySQL AB 总构架师

“很高兴和 Guy 及其他 O'Reilly 的编辑在一起做了许多有关这本书的技术文章。作者在这个方面拥有非常扎实的功底。我发觉这本书有很多带有代码的示例，很容易读懂，MySQL 的用户将发觉这本书是一个极好的学习资源。”

--Arjen Lentz, MySQL AB 社区版经理

“基于 MySQL 在现金企业级应用中的快速增长，开发者和数据库管理员都热切的期盼能够在创建高性能存储过程和其他 MySQL 特性代码方面得到专家级的知道，我相信没有人能找到比 Guy Harrison 和 Steven Feuerstein 写出的顶级的 MySQL 代码更有指导意义的书”

--Robin Schumacher, MySQL AB 项目经理

“这是第一本全神贯注与 MySQL 存储过程的书，我发现他的提点是我曾经都发觉过的”

--Peter Gulutzan, MySQL 软件构架师

“MySQL5.0 为用户打开了一个新的世界，而这本书是探索这个世界最好的指南”

--Andy Dustman, MySQL API for python 的作者

“Guy 和 Steven 给 MySQL 开发者带来了非常珍贵的东西，他们不仅覆盖了存储过程编写的细节方面，而且还提供了在实际开发中如何设计数据库应用程序的非常宝贵的建议，他们独特的幽默感增加了这本书的阅读乐趣”

--James Cooper 西雅图, WA 技术顾问

前言

在过去的五年时间里，我们亲眼见证了开源软件在商业领域应用的大爆发。Linux 几乎全面取代了各种 Unix--这种具有统治地位的非 Windows 操作系统；Apache 成为目前为止最有价值的 web 服务器；Perl 和 PHP 组件构成了成百万的商业网站；JBoss，Hibernate，Spring 和 Eclipse 强有力的入侵 Java，J2EE 开发以及应用服务器市场，虽然当今的关系型数据库仍然被一些商业机构（Oracle, IBM 和微软）所垄断着，但是，开源数据库在商业用户中正成指数增长。MySQL 是一个占有统治地位的开源数据库管理系统：他在基于 LAMP（Linux-Apache-MySQL-PHP/Perl/Python）和 LAMJ（Linux-Apache-MySQL-JBoss）这样的开源组合应用程序中起到了日益重要的作用，并且越来越多的被部署到需要高性能，稳定的关系型数据库的地方。

在 landmark 一书中改革的先驱者[*]Clayton Christensen 给出了为何开源和其他“颠覆性”技术会取代传统的可持续技术的第一个被广泛认可原因。

[*] The Innovator's Dilemma, Clayton Christensen (New York, 2000), HarperBusiness Essentials.

在好比 Linux 这样的颠覆性的技术刚刚出现的时候，它的能力和表现比主流及高端市场可以接受的尺度低得多，但是，新的技术对于那些有限制条件或者要在预算中去除现有商业构建花销的族群来说却有极高的吸引力。要知道那些低端市场总是伴随着低利润率和低产出，因此既有的供应商宁愿放弃这块市场来给这些颠覆性的技术以容身之处。无论是可持续/传统的或是颠覆性/创新的技术都在不断改进它们的能力，但当固有的技术显现出盛于平均用户需求甚至高端用户需求时，颠覆性技术在主流市场的部分人群中就开始变得有吸引力。

对于现有的供应商而言，低端市场总是伴随着低收益，于是供应商做出了一系列持续退出现有市场的决定，把市场留给了这些颠覆性的技术，同时，颠覆性的技术开始成为他们真正的威胁，现有的供应商无法在使用他们的既有产品所吞噬来的收入与之竞争，当然还有其他很多原因使他们最终失去了对市场的统治地位。

对于开源而言，MySQL 很典型的显示了颠覆性技术模型的这些特点。五年前，MySQL 的能力利主流商业需求还很遥远，以至于 MySQL 在商业环境下的使用几乎都没有听说过，虽然 MySQL 就像其他开源技术一样对于那些无法支付商用数据库的用户来说免费或者说成本极低[*]，但 MySQL 在历经了快速的技术变更中增加了事务，子查询和其他只有昂贵的商用数据库才能提供的功能，MySQL4.0 的发布使得这一数据库在很多旗帜鲜明的公司的关键性任务中得到了大量的应用，包括 Yahoo，Google 和 Sabre

[*] MySQL has a dual licensing model that allows for free use in many circumstances but does require a commercial license in some circumstances

同时，商业数据库公司增加了新的功能，虽然这些功能是相对于高端市场，但目前已经可以证实这些新特性已经远远超出大多数用户的需求，相对于混合对象数据类型，内建模拟 Java 虚拟机，或者是复杂的分区及聚合性能这些新特性，用户更关心性能，易于管理性和稳定性。

随着 5.0 的发布，MySQL 被证实跨越了成为可信赖的企业级应用的最后一个障碍，创建存储过程，函数，触发器和可更新视图等能力扫清了使得 MySQL 成为易维护性主流商用数据库的障碍，个别来说，在介绍存储过程之前首先要说 MySQLJava J2EE 的认证，因为这个认证需要产品包含存储例程，而在我们在 MySQL 中没有找到这些在商用数据库中所拥有的特性，当然，这些功能已经远远超出了主流数据库所应有的需求

我们相信 MySQL 将作为开源关系型数据库的领军人物继续前进，并且存储过程，函数触发器将在 MySQL 的成功故事中扮演重要的角色



首先，一点关于本书名称和属于的解释

在 IT 业界，媒体上或者 MySQL AB 自己的称呼都趋向于用存储过程一词带指存储过程和存储函数，虽然严格的说在技术上这是一种表述错误（函数并非过程），但是我们觉得相对于 MySQL

存储过程，函数和触发器编程这样一个沉长而不顺口的称呼，MySQL 存储过程编程这一书名更能准确的概述本书的用意。
--

为了避免引起歧义，我们是用存储过程程序这一术语来带指包括过程，函数和触发器在内的数据库例程，如果要特制其中的某一个，书中会特别指出。

这本书的目标

存储过程，函数和触发器所提供的新的能力（我们可以大致称他们为存储过程程序）给 MySQL 的开发者制定了新的游戏规则，只有在别的关系型数据库中已经拥有过经验，才能在 MySQL 应用程序中做的更好，更可靠及更有效。当然，不恰当的使用存储过程程序，或者差劲的存储过程程序构架，可能导致应用程序性能表现低劣，难以维护和不稳定。

基于这些原因，我们预见到了写一本书来帮追 MySQL 从业人员认识 MySQL 存储过程程序潜在能力的重要性。我们希望这本书能帮助你恰当的使用 MySQL 存储过程，并且写出可靠正确，有效且易于维护的过程，函数和触发器。

编写出可靠存储过程应用程序的实践依赖与以下四点：

恰当的使用

使用恰当的存储过程程序能够帮助你改善 MySQL 应用程序的性能，可靠性和可维护性，当然存储过程程序并非万能，他们只能在适当的场合被使用，在书中我们描述了存储过程程序可能被用来改善性能的场合，并且勾画了一些大致的模式（及不使用模式的）存储过程程序

可靠性

包括 MySQL 存储过程程序语言在能的所有编程语言都允许你写出在任何场合都有可预见行为的代码，但是这些语言也允许你写出受制于故障和各种不可预料场合的错误代码。我们勾画了怎样才能稳妥及可预见的面对错误，来轻松的面对各种程序错误

易维护性

我们总是对修改自己同事或自己所写的代码感到情绪低落，并且发现这些代码的意图，逻辑和机制几乎不能理解。所谓的“意大利面式的”代码可以用任何语言来写，这方面，MySQL 存储程序也不例外，我们将介绍如何通过管理，程序结构，注释和其他机制的实践来写出易于维护的代码

性能

任何非凡的应用程序都要面对潜在或显然的既定性能要求。数据库的 SQL 代码和存储程序代码的性能往往是影响应用程序全局性能的重要方面。此外，在要处理的数据和事务的体积增加时，落后的数据库代码经常不能彻底甚至完全不能达到预期的目的。在本书中，我们将向你展示什么时候该使用存储程序来改善性能以及如何用存储过程代码交付最高性能的应用程序。当然，结合 SQL 在内的存储程序经常是高性能应用程序的一个重要组成部分，所以我们也将在一定深度上来阐述如何写高性能的 SQL 代码

这本书的结构

MySQL 存储过程编程主要分为四个部分

第一部分，存储编程基础

第一部分主要介绍存储过程编程语言和详细的描述，语言结构及用法。

第 1 章，介绍 MySQL 存储过程程序，回答几个基本的问题：这种语言是怎么来的？它的好处有哪些？语言的主要功能有哪些？

第 2 章，MySQL 存储编程指南，作为一个指南来让你最快速度的开始使用语言，它向你展示了如何创建各种基本类型的存储程序，并提供了有关这种语言功能的交互式例子

第 3 章，语言基础，展示了如何使用变量，字面量，操作符和表达式

第 4 章，语句块，条件语句和迭代编程，并主要阐述了如何实现条件命令（IF 和 CASE）以及循环结构

第 5 章，在存储程序中使用 SQL，讨论怎样才能把 SQL 和这种语言结合起来

第 6 章，错误处理，提供了错误是怎样被处理的

第二部分，存储程序结构解析

这一部分将描述如何使用第一部分中的各个元素来创建功能强大而实用的程序

第 7 章，创建和维护存储程序，勾画用可用的语句创建和修改存储程序并提供了如何管理你的代码的一些建议

第 8 章，事务管理，阐述了在存储程序中使用事务的一些基础知识

第 9 章，MySQL 内建函数，详细介绍了可以用于存储程序的内建函数

第 10 章，存储函数，向你解释如何使用存储函数 -- 这一特别的存储程序

第 11 章，触发器，描述了如何使用另一种特别的存储程序：触发器 -- 在数据库表中被激活用来响应 DML（数据库操纵语言）

第三部分，在应用程序中的 MySQL 存储程序

存储程序可以被用来做各种不同的事情，包括提供给 MySQL 管理员和开发者的存储例程，当然，大多数重要的使用范畴都是像本章中所描述的和应用程序一起是使用的，存储程序允许我们将一些原本属于应用程序逻辑的代码移到数据库服务器内部；如果能够这部分，将能给我们的应用程序的安全性，有效性和易维护性带来很大的好处

第 12 章，在应用程序中使用 MySQL 存储程序，思考并在实践中体会在当今基于 web 的标准应用程序中使用存储程序的重大意义。其他的章节将向你展示如何在其他开发语言中和 MySQL 的存储过程和函数协同工作

第 13 章，在 PHP 中使用 MySQL 存储程序，描述如何在 PHP 中调用存储程序，我们将讨论 mysqlqi 及最近被绑定与 PHP 的 MySQL 连接器 PDO 和他们对于存储程序的支持

第 14 章，在 Java 中使用 MySQL 存储程序，介绍如何在 Java 的 JDBC, Servlets, 企业级 JavaBeans, Hibernate 和 Spring 中调用 MySQL 存储程序

第 15 章，在 Perl 中使用 MySQL 存储程序，介绍如何在 Perl 中使用 MySQL 存储程序

第 16 章，在 Python 中使用 MySQL 存储程序，介绍如何在 Python 中如何使用 MySQL 存储程序

第 17 章，在 .NET 中使用 MySQL 存储程序，介绍在 C# 和 VB.NET 中使用 MySQL 存储程序

第四部分，优化存储程序

本书的最后一个部分希望将“好”变得“更好”，能够使程序正确的运行是一件了不起的事情：任何正在运行的程

序都是一个好程序，而一个杰出的程序则需要性能优良，安全易维护且能应对一切

第 18 章，存储程序安全问题，独立的讨论安全性问题及由存储过程和函数引发的问题

第 19 章，调试存储程序和 SQL 代码，这一章节和接下来的 20 章，21，22 章将介绍存储程序的优化，这章将首先介绍性能优化的工具和一些技巧

第 20 章，基本 SQL 调试，你的存储程序的性能绝大部分取决于内部 SQL 代码，所以这一章将对 SQL 代码的调试基础给出指导

第 21 章，高级 SQL 调试，这一章是基于第 20 章，介绍了更多 SQL 高级调试的途径

第 22 章，优化存储程序代码，包含存储程序自身的性能优化

第 23 章，最好的存储程序开发实践，合上书本来看一下最好的存储程序开发实践，这些指导将让你写出快速安全，以维护，易调试的程序

你会发现本书在内容分配上比较均衡，这一点不仅体现在存储程序开发的章节中，同样也存在于例如 PHP 或者 Java 这些别的开发语言中，个别来说，我们假设你在不经 SQL 调试的情况下无法写出高性能的程序，所以我们在 SQL 调试上投入了大量的篇幅，再则，这样做无论 SQL 代码是否被内嵌与你的程序中都会有好处，同样的，讨论事务设计和安全问题在其他语言中也是可以接受的

这本书没有覆盖的内容

这本书并非覆盖了所有与 MySQL 相关的内容，它只关注于存储程序语言，下面的主题超出了本书的范围并且没有被覆盖，除了在偶然的场合才会被提到。

SQL 语言

我们假设你已经具备了使用 SQL 语言的知识，并且你应该知道怎样使用 `SELECT`, `UPDATE`, `INSERT` 和 `DELETE` 语句。

MySQL 数据库管理

虽然数据库管理员可以使用本书进行数据库创建和维护的知识，但是本书并未对 MySQL 的 DDL（数据定义语言）进行细致的探究。

本书中使用的约定

下面的约定将在本书中应用

斜体

将被用于 URL 和首次使用的术语的强调

等宽

将被用于代码示例中的 SQL 关键字

等宽加粗

在代码示例中，高亮当前被讨论的语句

等宽斜体

在代码示例中，指示应该由你提供的元素（比如：filename）

大写

在代码片段中指示 MySQL 关键字

小写

在代码片断中指示用户定义的变量或参数等

符号

在代码片段中为了严密输入

缩进

在代码片段中为了清楚的显示代码结构，当然这不是必须的

//

在代码片段中，单行注释将影响至这一行的结束

/*和*/

在代码片段中，多行注释定界符可以作用于多行

.

在代码片段和相关的讨论中，起到将对象名和成员名相分割的作用

[]

在语法描述中，表示可选参数

{ }

在语法描述中，表示你必须从中选取一个的参数列表

|

在语法描述中，分割在大括号中的元素，例如{TRUE|FALSE}。

...

在语法描述中，指示重复的项，也做和讨论无关的内容



指示提示，建议和注意事项，例如：我们将告诉你某个设定是不是与版本相关



指示警告或需要引起警惕，例如：我们将告诉你某些设定是否会与操作系统冲突

版本选择

这本书介绍了 MySQL5.0 的存储过程语言，虽然我们的源代码可以直接被使用在 5.1.7 这个 MySQL 版本上，但是 MySQL5.0.18 是最近被广泛使用的二进制社区版本

本书网站上的可用资源

我们在 O'Reilly 的站点上提供了所有本书的相关代码，进入

<http://www.oreilly.com/catalog/mysqlspp>

并且点击 Examples 来到本书的网站

要找到特定的代码片段，可以查找相应的文件或者指出代码出现的位置，比方说，要找到 Example 3-1，你可以进入 example0301.sql.

在网站上你也可以下载到在本书中使用的样例数据库的数据文件，源代码中包含了我们在开发过程中用到的样例文件，勘误表和附录

特别要指出，我们将用网站交流有关使用 MySQL 存储程序和其他工具的信息，因为 MySQL 存储程序相对来说是一个新生事务，MySQL 公司将在不同的 MySQL 数据库服务器版本中不断的修改，精炼这种语言的功能和行为，同样的在本书出版之时，在别的语言工具（PHP,Perl,Python,Hibernate）中对于存储程序的支持也是不完整的，所以我们将的网站中对这些语言的改进保持更新

使用样例代码

这本书是用来帮助你完成工作的，大体来说，你可以在你的书中，程序及文档中引用这些代码片段，除非你对这些代码做出了非常重大的修改，你不必联系我们已取得许可，举例来说，用书中的代码写了一个程序不需要请求许可，销售 O'Reilly 书中的代码的 CD-ROM 分法版不需要许可，用书中的代码回答技术问题不需要许可，在你的文档中使用本书的代码不需要许可

我们赞同在引用中申明标题，作者，出版商和 ISBN，举例来说，“MySQL Stored Procedure Programming by Guy Harrison with Steven Feuerstein. Copyright 2006 O'Reilly Media, Inc., 0-596-10089-2.”，但这并非必须

如果你觉得你在使用本书的时候超出了正常的使用范围，可以随时给我们来信 permissions@oreilly.com

Safari (R) 标志



当你在你钟爱的技术书本的封面上看到 Safari(R)标志的时候,这意味着这本书可以在 O'Reilly Network Safari 书架上在线得到

Safari 提供了一种比 ebook 更好的解决方案,它是一个虚拟的图书馆,可以让你轻松的搜索上千本顶级技术书籍,剪切,粘贴代码片段,下载章节并快速而精确的找到你要的信息,你可以进入这个网址 <http://safari.oreilly.com>.

怎样联系我们

我们在出版之前已经尽我们的所能对书中的信息和源代码进行了校验，但是由于巨大的数目以及不断发展的技术，你也许会发现许多功能都在发生变化，因而我们的书本出现了错误，如果那样的话，请联系我们

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

你也可以给我们发电子邮件，或者发到邮件列表或目录中

info@oreilly.com

有关技术问题的答疑和对本书的评论，请发送至

bookquestions@oreilly.com

同样，在前些章节已经提到过我们有一个和本书配到的网站，你在上面可以找到本书的源代码，勘误表（以前的错误报告和现在的错误公开）和其他一些本书的信息，你可以进入网址

<http://www.oreilly.com/catalog/mysqlspp>

其他有关本书的信息和别的事项，可以进入 O'Reilly 的官方网站

<http://www.oreilly.com>

翻译作者

[drekey \(at\) gmail dot com](mailto:drekey(at)gmail(dot)com)

对本书的相关工作人员表示感谢

我们首先要感谢我们在 O'Reilly 的编辑，感谢他不遗余力的工作和在幕后对各个项目的关注，还有很多 O'Reilly 的同仁在书本的撰写过程中也扮演了重要的角色，包括责任编辑 Adam Witwer 和 Rob Romano，美工和其他发行事宜都有 Argosy Publishing 赞助

在这本书的编撰过程中技术评论家的建议是相当苛刻的，这本书的范围不仅包括了 MySQL 存储程序语言本身，还有五种不同的其他开发语言及 MySQL5.0 自身的新特性，此外，在我们写书的时候存储程序本身也在不断的进化，如果没有我们的这些技术评论家宝贵的意见我们的书就无法在技术的精准度和技术范围的适应性上取得任何成绩，感谢技术评论者 Tim Allwine, Brian Aker, James Cooper, Greg Cottman, Paul DuBois, Andy Dustman, Peter Gultzan, Mike Hillyer, Arjen Lentz, 和 Mark Matthews。

对于整个开源社区特别是 MySQL 社区版的开发团队，我们也要说一声感谢，拥有如此高质量和革新性的开源软件的能力不断的满足着我们的各种需求并给我们以惊喜，很多 MySQL 及其他开源社区通过不同的途径贡献着他们的力量

我们使用了很多开源的第三方维护接口程序来确保能够支持 MySQL5.0 的新特性，特别感谢 Wez Furlong, Patrick Galbraith, 和 Andy Dustman 及他们在 PHP PDO, Perl DBI 和 Python MySQLdb 接口中所做的贡献。

Guy（本书的原作者）：（引用他自己的话来说）我得特别感谢我的妻子 Jenni 和孩子 Christopher, Katherine, Michael 和 William 对我在书籍的撰写过程中对我的宽容和支持，我很爱他们，也同样感谢 Steven 和我合写了这本书

Steven：我在最近的 10 年里花了大量的时间于 Oracle PL/SQL 的学习和工作上，这些经验清楚的告诉我存储程序的重要性，当 Guy 邀请我和他同写这一书时我也很高兴，并且我认为借助这次契机给 MySQL 程序员的帮助充分的利用了开源关系型数据库

第一章 MySQL 存储程序介绍

当 MySQL 于上个世纪 90 年代中期在 IT 界刚刚成型的时候，它只具有少量的商用关系型数据库所具备的特性。出现了比如事务处理，子查询，视图和存储过程这样的功能的明显缺失，后继版本提供了大量缺失的功能，现在介绍的 MySQL5.0 的存储过程，函数和触发器（还有可更新视图以及数据目录）等功能大大的缩短了 MySQL 和其他关系型数据库系统的差距。

现在介绍的存储程序（我们通常所说的存储过程，函数和触发器）在和其他竞争对手的功能战上赢得了简单的胜利。如果没有存储程序，MySQL 就无法以一种完整的姿态和别的竞争对手相比拼，因为在 ANSI、ISO 的标准当中也要求数据库管理系统应具备执行存储程序的功能。此外，正确的使用存储程序也有助于加强数据库的安全性和完整性及改善你的应用程序的性能和易维护性。我们将在本章的稍后部分介绍这些优势的具体内容。

简短的说，存储程序，过程，函数和触发器是一种 MySQL 的强大能力，而用好这些编程工具是 MySQL 专业上所必须具备的

本章将介绍 MySQL 存储程序语言，她的起源和她的能力，同样我们为 MySQL 存储程序开发者提供了一些附属资源的导引和综合开发的建议

1.1 什么是存储程序

数据库存储程序有时也被称为存储模块或者存储例程 -- 一种被数据库服务器所存储和执行的计算机程序（有一系列不同的称呼），存储程序的源代码（有时）可能是二进制编译版本几乎总是占据着数据库服务器系统的表空间，程序总是位于其数据库服务器的进程或线程的内存地址中被执行。

主要有三种类型的数据库存储程序

存储过程

存储过程是最常见的存储程序，存储过程是能够接受数个输入和输出参数并且能够在请求时被执行的程序单元。

存储函数

存储函数和存储过程很相像，但是它的执行结果会返回一个值。最重要的是存储函数可以被用来充当标准的 SQL 语句，允许程序员有效的扩展 SQL 语言的能力

触发器

触发器是用来响应激活或者数据库行为，事件的存储程序，通常，触发器用来作为 DML（数据库操纵语言）的响应而被调用，触发器可以被用来作为数据校验和自动反向格式化。



其他的数据库提供了别的存储程序，包括包和类 -- 允许你定义和组织一堆上下文过程和函数，MySQL 现在还不提供这种结构，每个程序都是一个单独的实体

在这本书中，我们将使用术语：存储程序来带指存储过程，函数和触发器，术语：存储程序语言带指用来写这些程序的语言，大多数程序的基本构建都可以在存储过程，函数和触发器中被使用；但是，存储函数和触发器在使用这些构建时是严格受限的。所以我们将另辟章节来专门介绍这些程序类型的限制

1.1.1 为什么使用存储程序

开发者总是有很多种编程语言可以选择，这意味着大多数的语言并非位于数据库内，并受其管理，存储程序相对于这些多用途语言而言拥有很多优势：

- 存储程序的是使用可以使你的数据库更安全
- 存储程序提供了一种数据访问的抽象机制，它能够极大的改善你的代码在底层数据结构演化过程中的易维护性
- 存储程序可以降低网络拥阻，因为属于数据库服务器的内部数据，这相比在网上传输数据要快的多
- 存储程序可以替多种使用不同构架的外围应用实现共享的访问例程，无论这些构架是基于数据库服务器外部还是内部。
- 以数据为中心的逻辑可以被独立的放置于存储程序中，这样可以为程序员带来更高，更为独特的数据库编程体验
- 在某些情况下，使用存储程序可以改善应用程序的可移植性

作为对这些优势的最初映像（大多数将在本书的稍后深入的探讨），我们不建议你立即将所有的程序逻辑移至存储程序中，在今天丰富而复杂的软件技术背景下，你必须清楚的认识你的软件中每个技术细节的优势和弱势，并且将这些优势最大化，我们将在第 12 章详细的讨论在什么地方，怎样使用 MySQL 存储程序

使用存储程序的底线是：正确的使用存储程序，过程，函数和触发器能够帮助你改善应用程序的性能，安全性，易维护性和稳定性

后续的章节将带你探索怎样来构建 MySQL 存储程序和他们的最大好处，在详细介绍之前，让我们来看一下有关这项技术的发展和语言特性的快速浏览

1.1.2 MySQL 的简史

MySQL 植根于由瑞典的 Tcx 公司于 1980 年开发的一个叫做 Unireg 的非 SQL 数据库系统，作为数据仓库的优化，Unireg 的作者 Michael "Monty" Widenius 于 1995 年为其加入了 SQL 接口，这就是 MySQL 的第一个版本。来自 Detron HB 的 David Axmark 为了让 MySQL 取得成功为其提供了双许可证，这使得 MySQL 能够广泛的免费得到，同时它也具备了商业使用上的优势，于是，Allan Larsson, David 和 Monty 成为 MySQL 公司的创建者

第一个被广泛使用的 MySQL 版本是于 1996 年发布的 3.11，随着 MySQL 及相关开源技术的快速发展，到 2005 年，MySQL 宣称其数据库已经拥有 6 百万的安装数

MySQL3 适合的应用层面很广，（当然适应于 web 应用程序是非常可以理解的），但是缺乏一些常规数据库做应具备的功能，举例来说：事务，视图和子查询都没有在这个最初的版本中被支持

不管怎样，MySQL 系统在诞生之初就被设计为具有可扩展数据访问结构，SQL 层用能够为底层的数据和文将访问层解耦。这允许使用各种自定义数据引擎来替换原有的本地 ISAM（索引顺序存取方法）数据库引擎，2001 年，BDB（Berkeley-DB）数据引擎（由 Sleepycat 负责开发维护）被作为 3.23.34 的可选组件被集成进 MySQL，BDB 提供了 MySQL 最初的事务处理能力，同时，开源的 InnoDB 很快成为 MySQL 用户的可选本地数据引擎。

2002 年早期发布的 4.0 版完整的整合了 InnoDB 这一选项，这让 MySQL 用户非常容易的获得了事务处理的支持，以及改进的数据同步能力，在 2004 年发布的 4.1 版本提供了对于子查询和 Unicode 的支持

2005 年晚期发布的 MySQL 5.0 在向商用关系型数据库系统的功能靠近的方面做出了改进，它开始支持存储过程，函数和触发器，及数据目录（INFORMATION_SCHEMA 的 SQL 标准），并支持了可更新视图

5.1 版本预计将于 2006 年中晚期发布，将提供例如内部作业调度，表分区，基于记录的同步功能和其他一些有意义的改进

1.1.3 MySQL 存储过程，函数和触发器

MySQL 选择将其存储程序作为 ANSI SQL：2003 SQL/PSM(数据持久模块)的一个子集来实现，本质上 MySQL 的存储程序过程，函数和触发器只是遵循了 ANSI 对于这些程序类型的开放标准。

很多 MySQL 和其他开源爱好者则希望将存储程序语言实现为基于其他开源语言，例如 PHP 或者 Python 及 Java 的版本，虽然最后选择了 ANSI 的规范 -- 同样的规范也被 IBM 的 DB2 数据库所采用，但这是 MySQL 在 ANSI 委员会中做了大量长期的工作的结果，这个标准在大量的商用关系型数据库公司所采纳的标准中具有相当的典型性

MySQL 存储程序语言是一种类似于 Pascal 的块语句结构的语言，包含了大量人们熟知的命令，包括变量操纵，条件语句的实现方式，迭代编程和错误处理等，其他现有的数据库存储程序用户（例如 Oracle 的 PL/SQL 或者 SQL Server 的 Transact-SQL）将发觉他们大体上看起来非常相似，与 PHP 或者 Java 相比，这种语言可能显的过于单薄，但是很快你将发现它很好的适应了数据库编程的常规需求。

1.2 快速浏览

让我们看一些包含 MySQL 存储程序结构和功能的基本要点的简单示例，完整的内容详见第二章。

1.2.1 和 SQL 集成

MySQL 存储过程语言一个非常重要的方面就是和 SQL 的紧密结合，你不需要借助于像 ODBC 或者 JDBC 这样的软件粘合剂来为你的存储程序创建独立的 SQL 语句，只要简单的将 UPDATE,INSERT 和 SELECT 这样的语句直接写进你的存储程序代码中，就像 Example 1-1 所显示的那样

Example 1-1. 内嵌 SQL 的存储程序

```
1 CREATE PROCEDURE example1( )
2 BEGIN
3     DECLARE
4         l_book_count INTEGER;
5     SELECT COUNT(*)
6         INTO l_book_count
7         FROM books
8         WHERE author LIKE '%HARRISON,GUY%';
9
10    SELECT CONCAT('Guy has written (or co-written) ',
11                l_book_count ,
12                ' books.');
```

13

```
14    -- Oh, and I changed my name, so...
15    UPDATE books
16        SET author = REPLACE (author, 'GUY', 'GUILLERMO')
17        WHERE author LIKE '%HARRISON,GUY%';
18
19 END
```

下表阐述了更为详细的代码信息

行号	解释
1	这个区块，是程序的头部，定义了程序的名称（example1）以及类型（PROCEDURE）
2	BEGIN 关键字指示了程序体的开始，其中包含了存储过程的变量申明和可执行代码，如果程序体包含了超过一个语句（就像这个程序中所看到的一样），那么要将多个语句包含在 BEGIN-END 块中
3	这里我们申明了一个整型的变量来保存我们将要执行的数据库查询代码
5-8	我们执行了一个数据库查询来获得 Guy 所编写和执笔的属的总数，特别关注一些第 6 行：INTO 子句和 SELECT 连用表示将数据库查询结果传递给存储程序的本地变量
10-12	我们是用了一个简单的 SQL 语句（例如：没有带 FROM 字句）来显示书的个数。如果我们使用了没有带 INTO

	字句的 SELECT 语句,那么结果将返回给调用程序,这是一个能够简单的得到结果集的非 ANSI 扩展(SQL Server 和其他关系型数据库所采用的方式)
14	单行注释解释了 UPDATE 的用意
15-17	Guy 大概想和他的 fans 讨论有关 Oracle, 并想改变他姓的拼写方法, 所以我们对 books 表使用了 UPDATE, 得益于内建的 REPLACE 函数我们能将表中所有包含“GUY”的实例替换为“GUILLERMO”。

1.2.2 控制和条件逻辑

当然，现实世界中的代码具有特定的用途并且相当复杂，你不可能仅仅在其中使用一系列的 SQL 语句，存储过程语言提供了我们非常丰富的条件和控制语句，这使我们能够编写出适应给定情景的程序，它们包括：

IF 和 CASE 语句

这些语句都使用不同的逻辑来实现条件逻辑，这允许我们表达像“如果书本的页数大于 1000，然后...”这样的逻辑

完整的循环和迭代控制

它包含简单循环，WHILE 循环和 REPEAT UNTIL 循环

Example 1-2 是一个用来显示帐户金额收支平衡的存储过程，给出了 MySQL 的控制语句示例

Example 1-2 包含控制和条件逻辑的存储过程

```
1 CREATE PROCEDURE pay_out_balance
2     (account_id_in INT)
3
4 BEGIN
5
6 DECLARE l_balance_remaining NUMERIC(10,2);
7
8 payout_loop:LOOP
9     SET l_balance_remaining = account_balance(account_id_in);
10
11     IF l_balance_remaining < 1000 THEN
12         LEAVE payout_loop;
13
14     ELSE
15         CALL apply_balance(account_id_in, l_balance_remaining);
16     END IF;
17
18 END LOOP;
19
20 END
```

下表阐述了更为详细的代码信息

行号	解释
1-3	这是存储过程的头部；第二行包含了过程的参数列表，接受一个数据参数（帐户的 id）
6	申明了一个保存帐余额的变量

8-18	一个简单循环(这样称呼是因为使用 LOOP 关键字来和 WHILE 及 REPEAT 进行区别)直到帐户余额少于 1000, 在 MySQL 中我们可以命名一个循环(第 8 行, payout_loop), 这使得我们可以在随后的代码中使用 LEAVE 语句(见第 12 行)来结束这个特定的循环, 结束循环后, MySQL 引擎将执行 END LOOP (见第 18 行)之后的代码
9	调用了函数 account_balance (当然这个函数必须已经在前面的代码中被定义)来获得帐户的收支状况。 MySQL 允许我们在存储程序中调用别的存储程序, 这可以有效的实现代码复用, 如果这是一个函数, 那么它将返回一个值并且能被其他存储程序和 MySQL 作业调度所调用
11-16	IF 语句引发了当帐户余额少于 1000 美元时的循环终结条件, 此外(ELSE 语句)能够对收支平衡进行进一步处理, 你可以用 ELSEIF 构建更为负责的布尔表达式
15	调用了存储过程 apply_balance , 这是一个代码复用的例子, 与其重复 apply_balance 的逻辑, 我们还是调用一个共享例程比较方便

1.2.3 存储函数

存储函数是能够返回一个值的存储程序, 它也可以当作内建函数一样对待(调用)。**Example 1-3** 将在存在出生年月的前提下返回这个人的年龄

Example 1-3. 用出生年月计算年龄的存储函数

```

1 CREATE FUNCTION f_age (in_dob datetime) returns int
2   NO SQL
3 BEGIN
4   DECLARE l_age INT;
5   IF DATE_FORMAT(NOW( ), '00-%m-%d') >= DATE_FORMAT(in_dob, '00-%m-%d') THEN
6     -- This person has had a birthday this year
7     SET l_age=DATE_FORMAT(NOW( ), '%Y')-DATE_FORMAT(in_dob, '%Y');
8   ELSE
9     -- Yet to have a birthday this year
10    SET l_age=DATE_FORMAT(NOW( ), '%Y')-DATE_FORMAT(in_dob, '%Y')-1;
11  END IF;
12  RETURN(l_age);

END;
```

接下来让我们看一下具体解释

行号	解释
1	定义函数: 名称, 参数(日期)和返回值(整型数)。
2	这个函数没有使用 SQL 语句, 这将在第 3 章和第 10 章进行详细的讨论
4	申明一个用来保存我们计算出的年龄的本地变量
5-11	在这个 IF-ELSE-END 块中, IF 块用来检测出生年月是否存在
7	如果出生年月存在, 那么我们就可以用现在的年份减去出生的年份得到年龄
10	此外(如果出生年月不存在), 我们必须在岁数计算中简单的减去当前的年份
12	返回年份计算的函数调用

我们可以在任何其他的存储程序,SET 语句或者在 Example 1-4 所显示的那样,在 SQL 语句中调用我们的存储函数

Example 1-4.在 SQL 语句中使用存储函数（延续上一部分）

```
mysql> SELECT firstname,surname, date_of_birth, f_age(date_of_birth) AS age
-> FROM employees LIMIT 5;
```

firstname	surname	date_of_birth	age
LUCAS	FERRIS	1984-04-17 07:04:27	21
STAFFORD	KIPP	1953-04-22 06:04:50	52
GUTHREY	HOLMES	1974-09-12 08:09:22	31
TALIA	KNOX	1966-08-14 11:08:14	39
JOHN	MORALES	1956-06-22 07:06:14	49

1.2.4.当发生错误时

即使我们的程序被反复检查并没有 bug,我们的输入错误仍然可能发生,MySQL 存储程序语言提供了一种错误处理的强大机制,在 Example 1-5 中,我们创建一个产品代号,如果产品已存在,我们就用新的名称更新它,存储过程的错误处理机制检测到我们试图使用一个重复的值,如果尝试插入失败,错误将被捕获并且使用 UPDATE 替换 INSERT,如果没有错误处理,存储程序将被终止执行,并且错误将被返回给它的调用程序

Example 1-5 错误处理

```
1 CREATE PROCEDURE sp_product_code
2     (in_product_code VARCHAR(2),
3     in_product_name VARCHAR(30))
4
5 BEGIN
6
7     DECLARE l_dupkey_indicator INT DEFAULT 0;
8     DECLARE duplicate_key CONDITION FOR 1062;
9     DECLARE CONTINUE HANDLER FOR duplicate_key SET l_dupkey_indicator =1;
10
11     INSERT INTO product_codes (product_code, product_name)
12     VALUES (in_product_code, in_product_name);
13
14     IF l_dupkey_indicator THEN
15         UPDATE product_codes
16             SET product_name=in_product_name
17             WHERE product_code=in_product_code;
18     END IF;
19
20 END
```


让我们看一下有关错误处理的详细内容

行号	解释
1-4	这是程序的头部，包含了两个输入参数产品代号和产品名称
7	申明了一个用来检测重复值出现的标志变量，这个变量被初始化为 0（false）；后续的代码将保证在重复值被替换时将把这个变量设置为 1（true）。
8	命名一个条件 <code>duplicate_key</code> 和 MySQL 系统错误 1062 相匹配，虽然这一步并非必需，但是我们建议你定义一个条件来改善你代码的可靠性（现在你可以使用错误名称而不是代码来引用这个错误）
9	定义一个错误处理器，它将在后续的代码中重复值出现的时候将 <code>l_dupkey_indicator</code> 变量的值设置为 1(true)
11-12	插入用户提供的代号和名称
14	检测变量 <code>l_dupkey_indicator</code> 的值，如果仍为 0，那么说明我们的插入成功了，如果值被修改成了 1（true），我们就知道出现了重复值，我们可以在地 15-17 行代码使用 <code>UPDATE</code> 语句将原来的产品名称和代号进行更新

1.2.5 触发器

触发器是一种用来相应数据库事件是自动回调的存储程序，在 MySQL5 的实现中，触发器将在特定表的 DML（数据库操纵语言）激活时被回调，触发器可以用来自动计算引用值或者格式化值。Example 1-6 展示了用于维护引用值的触发器；当员工 `salary` 的值被改变是，`contrib_401K` 列将被自动修改为特定值。

Example 1-6. 维护引用列的触发器

```

1 CREATE TRIGGER employees_trg_bu
2     BEFORE UPDATE ON employees
3     FOR EACH ROW
4 BEGIN
5     IF NEW.salary < 50000 THEN
6         SET NEW.contrib_401K=500;
7     ELSE
8         SET NEW.contrib_401K=500+(NEW.salary-50000)*.01;
9     END IF;
10 END

```

下表用来解释这个短小的触发器的代码

行号	解释
1	一个触发器有它独立的名称，通常，我们用名称来描述它的作用，举例来说，名称中的“bu”表示 BEFORE UPDATE（在更新前）使用的触发器
2	定义一个触发器激活条件，在这个例子中，触发器代码将在 UPDATE 语句起作用前被触发
3	FOR EACH ROW 关键字指示了触发器将在所有的记录被 DML 语句作用前被触发。这个字句在 MySQL5 的触发器实现中是强制执行的
4-10	BEGIN-END 定义了将被激活的触发器代码
5-9	自动修改 employees 表中 contrib_401K 列。如果 salary 列的值小于 50000，contrib_401K 列将被设置为 500，否则，这个值就参与执行第 8 行的计算

当然，还有有关 MySQL 存储过程值得说的很多故事，所以你还有上百页的书要看，这些基础的代码主要是让你培养一种对存储程序语言良好的感觉，其中的一些是很重要的语法点，请轻松的面对你所要写和阅读的语言代码。

1.3 为开发者准备的存储过程参考资料

已经介绍过 MySQL5 的存储程序在整个 MySQL 语言的演进过程中具有里程碑的意义。为了能够全神贯注于任何新技术的需要，技术用户需要大量的技术资料的支持，我们的目标是介绍尽可能全面资料来覆盖 MySQL 存储程序语言。

我们确信，无论如何你需要很多不同的帮助，所以我们在下面的章节中介绍了完整的书籍（作为其他 MySQL 技术的参考信息）或者社区支持和只需的新闻内容。在这些章节中我们将提供很多相关信息的概要。这些有极大作用的资源大多数都可以免费获得或以低廉的价格得到，你将在高 MySQL 开发体验上获得巨大的帮助

1.3.1 书

长期以来，O'Reilly 的 MySQL 系列积累了大量的书籍。在此我们列出了一些我们认为 MySQL 存储程序开发者应该关心的书籍。当然也包括了其他出版社的书籍，请在 O'Reilly OnLAMP 网站的 MySQL 区查看相关完整列表（<http://www.onlamp.com/onlamp/general/mysql.csp>）

MySQL Stored Procedure Programming, by Guy Harrison with Steven Feuerstein

This is the book you are holding now (or maybe even viewing online). This book was designed to be a complete and comprehensive guide to the MySQL stored program language. However, this book does not attempt complete coverage of the MySQL server, the SQL language, or other programming languages that you might use with MySQL. Therefore, you might want to complement this book with one or more other topics from the O'Reilly catalog or even heaven forbid from another publisher!

MySQL in a Nutshell, by Russell Dyer

This compact quick-reference manual covers the MySQL SQL language, utility programs, and APIs for Perl, PHP, and C. This book is the ideal companion for any MySQL user (O'Reilly).

Web Database Applications with PHP and MySQL, by Hugh Williams and David Lane

This is a comprehensive guide to creating web-based applications using PHP and MySQL. It covers PEAR (PHP Extension and Application Repository) and provides a variety of complete case studies (O'Reilly).

MySQL, by Paul DuBois

This classic reference now in its third edition is a comprehensive reference to MySQL development and administration. The third edition includes prerelease coverage of MySQL 5.0, including some information about stored procedures, functions, and triggers (SAMS).

High Performance MySQL, by Jeremy Zawodny and Derek Balling

This book covers the construction of high-performance MySQL server environments, along with how you can tune applications to take advantage of these environments. The book focuses on optimization, benchmarking, backups, replication, indexing, and load balancing (O'Reilly).

MySQL Cookbook, by Paul DuBois

This cookbook provides quick and easily applied recipes for common MySQL problems ranging from program setup to table manipulation and transaction management to data import/export and web interaction (O'Reilly).

Pro MySQL, by Michael Krukenberg and Jay Pipes

This book covers many advanced MySQL topics, including index structure, internal architecture, replication, clustering, and new features in MySQL 5.0. Some coverage of stored procedures, functions, and triggers is included, although much of the discussion is based on early MySQL 5 beta versions (APress).

MySQL Design and Tuning, by Robert D. Schneider

This is a good source of information on advanced development and administration topics, with a focus on performance (MySQL Press).

SQL in a Nutshell, by Kevin Kline, et al.

MySQL stored procedures, functions, and triggers rely on the SQL language to interact with database tables. This is a reference to the SQL language as implemented in Oracle, SQL Server, DB2, and MySQL (O'Reilly).

Learning SQL, by Alan Beaulieu

This book provides an excellent entry point for those unfamiliar with SQL. It covers queries, grouping, sets, filtering, subqueries, joins, indexes, and constraints, along with exercises (O'Reilly).

1.3.2 网络资源

网络上同样有大量关于 MySQL 程序员的极好的网站，包括一些关心存储过程的领域
当然我们于书籍相配套的网站上面也有更新和勘误表以及 MySQL 的信息，你也应该关心一下

MySQL

MySQL AB offers the most comprehensive collection of white papers, documentation, and forums on MySQL in general and MySQL stored programming in particular. Start at <http://www.mysql.com>. We outline some specific areas later.

MySQL Developer Zone

<http://dev.mysql.com/> is the main entry point for MySQL programmers. From here you can easily access software downloads, online forums, white papers, documentation, and the bug-tracking system.

MySQL online documentation

The MySQL reference manual including sections on stored procedures, functions, and triggers is available online at <http://dev.mysql.com/doc/>. You can also download the manual in various formats from here, or you can order various

selections in printed book format at <http://dev.mysql.com/books/mysqlpress/index.html>.

MySQL forums

MySQL forums are great places to discuss MySQL features with others in the MySQL community. The MySQL developers are also frequent participants in these forums. The general forum index can be found at <http://forums.mysql.com/>. The stored procedure forum includes discussions of both procedures and functions, and there is a separate forum for triggers.

MySQL blogs

There are many people blogging about MySQL nowadays, and MySQL has consolidated many of the most significant feeds on the Planet MySQL web site at <http://www.planetmysql.org/>.

MySQL stored routines library

Giuseppe Maxia initiated this routine library, which collects general-purpose MySQL 5 stored procedures and functions. The library is still young, but already there are some extremely useful routines available. For example, you will find routines that emulate arrays, automate repetitive tasks, and perform crosstab manipulations. Check it out at <http://savannah.nongnu.org/projects/mysql-sr-lib/>.

O'Reilly's OnLAMP MySQL section

O'Reilly hosts the OnLAMP site, which is dedicated to the LAMP stack (Linux, Apache, MySQL, PHP/Perl/Python) of which MySQL is such an important part. OnLAMP includes numerous MySQL articles, which you can find at <http://www.onlamp.com/onlamp/general/mysql.csp>.

1.4 给开发者的建议

事实上，每个人对于 MySQL 存储程序开发都会感到陌生，因为存储程序本身对于 MySQL 就是新事物，但是，Guy 和 Steven 在其他关系型数据库的存储程序编程上拥有大量的经验，特别是 Steven，他在 Oracle PL/SQL（Oracle 的存储程序语言）的开发商拥有超过十年的经验。我们希望这些建议能帮助你更有效的理解 MySQL 编程语言的强大魅力

1.4.1 万事不能操之过急

我们总是为限定的工期拼命的工作，追逐一个又一个的新事物，新潮流，我们没有时间可以浪费，我们有大量的代码要写，要怎样才能让我们恢复正常？

如果你想一下子接触深度的代码结构，奴隶般的将需求转化为数百行，千行甚至数万行，那么你将被巨大的混乱所摧毁，你的程序将变得难以调试和维护，不要被紧张的开发期限所压垮，我们更希望你能在紧张的期限中做好周密的计划。

我们强烈建议你顶住时间的压力，在你开始新的应用之前做好以下准备：

在你写代码之前建立良好的测试机制和测试脚本

你必须在动手写第一行代码之前给怎样才算一个成功的实现下一个定义。你更像是在为你的程序的该做什么建立一个接口，并彻底搞清楚这些功能的区别

为开发人员在应用程序中所写的 SQL 语句建立清晰的规则

总体来说，我们建议你认清这样一个事实：开发者并不需要写一大堆的 SQL 代码，相反的，各种对数据的查询，插入和更新操作都必须隐藏在我们预先建立并通过大量测试的存储过程函数中（这被称为数据封装），这样做你的程序就能比使用大量离散的 SQL 语句写出的程序更易于被优化，测试和维护。

为开发人员在错误处理上建立清晰的规则

如果你不树立标准，那么每个人都会有他自己的错误处理方法或者根本就不处理，这会造成软件混乱。最好的方法是将错误的处理逻辑集中在一个存储过程集合中，这个集合中的过程是专注于错误消息保存，错误的引发和传播方式的内部代码块（言下之意就是将错误处理的复杂度封装在这个过程集合中），并且保证你的开发者不需要为为了错误处理而建立非常复杂的代码。

必须分配充足的时间，使用抽丝剥茧的方法（逐层封装复杂度，也就是 a.k.a 的从上之下的设计模式）来消除你需求中的复杂度

我们时常要面对非常复杂的需求实现，如果你把所有的东西都放在一个“万能”的程序中，那么很快你就会发现这些意大利面式的代码将是你随后的日子里对代码的理解造成极大的困难，把你的巨大挑战分解为一个更小的问题，并把这些容易解决的小问题写成大小可以接受的程序，这样做，你将发现程序的可执行段明显的缩小，可读性也提高了，你的代码将变得易于维护又节省了时间

当你开始写代码的时候其实只有很少事情是你要放在心上的，请记住：在软件开发也盲目求速只能造成更大的浪费和更多 bug。

1.4.2 不要害怕请教问题

事实是这样，如果你是个专业的软件从业者，那么你一定很聪明，受过良好的教育，你学习认真，你经验丰富，而且你写出了如此生动的代码，你能解决大多数的问题，这是你骄傲。

不幸的是，你的成功却是你自大，傲慢，不再听取别人的意见（我想大家都知道答案）软件开发也是动态发展的，而这也使之成为最有危机的行业。

软件是有人写的：因此认识人的心理成为软件开发的重要环节。这里有一个例子：

Joe，一个由 6 人组成的高级软件开发团队的头目，在他的软件里出现了一个问题，他在这个问题上花了大量的时间，也经受了大量的挫败，但他始终未能指出 bug 的源头。他不想去问他的助手，因为他们的经验都没有自己丰富，最后他穷尽脑汁也未能奏效，他只得放弃，在叹息声中他打电话求援："Sandra,你能不能过来看看我程序中的问题，我不知道错在哪里？"Sandra 停下手中的活儿很快的浏览了以下代码，并很快指出了长期以来他没有注意到的问题，程序就这样被修复了，Joe 表示感谢，但事实上他内心非常尴尬。

就像“为什么我没看到”还有“如果我在自己这儿多做五分钟的调试就能发现它”始终在 Joe 的心中环绕，他无法理解，但事实上他被误导了，原因就在于我们已经太熟悉自己写的代码了，有些时候我们需要的仅仅是一个新的视角，某人和善的一个建议可能就能打开新的视野，这与资历，能力和经验无关

此外，Sandra 并不认为 Joe 很差劲，相反的，通过相互的帮助，Joe 是他自己更具人情味儿，这对团队开发有很大的好处

我们强烈的建议您在团队的管理中贯彻以下方针

原谅无知

在应用程序开发过程中隐藏你的无知是件及其危险的事情，培养一种能够把“I don't know”说出口的氛围并且鼓励问问题

请求帮助

如果你在 30 分钟内不能指出代码中的 bug，那么请立即请教别人，这样也就建立了一种“责任机制”，使得每一个被你问起的人都有一种责任感。不要让你一个人孤立的寻找问题的答案

建立一种代码互查机制

不要让你的代码敲上“金牌质量”的标签或者经不起你团队中任何人的批评（建设性的意见）

1.4.3 打破条条框框

我们都会落入俗套，在这一点上每个人你的方方面面都是相同的，人有创造的天性：你只用学过的一种方法编写代码；你的产品的功能限制有不自觉的假定；你不加思索就抛弃了可能的解决方案，开发者对他们的程序都有自己的偏见，他们总是不恰当的说出这种话：

- “它就像猪一样不可能让它再快了”
- “我不能按照用户的预想来实现，这得等到下一个版本”
- “我用过很多产品，但是他们都是小儿科，放在你眼前的这个不同，它的任何方面都已经做得相当出色”

但事实是你的程序总能运行的更快一些，别人总能把功能做的符合用户的要求，产品虽然都有它的限制，能力范围和弱点，但你永远都不要指望在下一个版本中去完善它，这种不需要借口，没有等待的解决问题的风格不是更能让人满意吗？

你究竟在干什么？打破你自以为是的那成百上千的阅历（或是你自己的小世界），估量你平日里养成的编程习惯。抛弃那些旧方法和你对产品固有性能的一切偏见已经成为现在商业社会的一股强大力量。

尝试各种新事物：用违背常理的方法去实验，你将发现作为一个程序员或者问题解决大师你将能学到多么不可思议的东西。长期以来，在一次次追问自己什么才是终极目标时总是惊奇的发现，当我们谦虚的点点头，轻声询问自己：“如果这样做会发生什么”总会比傲慢的说“用不可能做到”获得的更多。

1.5 结语

在这一章中，我们带大家快速浏览了 MySQL 关系型数据库及 MySQL 存储过程语言的大致情况。我们同样提供了大量的可用资源和建议，希望能给你带来帮助

在下一掌中，我们将正式开始进行 MySQL 存储过程，函数和触发器的学习，并会提供更多的相关内容

第二章 MySQL 存储过程编程指南

MySQL 存储过程编程是一个复杂的主题，我们将在本章中为你提供完成基本任务的内容，其中包括：

- 怎样创建存储程序
- 存储程序怎样进行输入输出
- 怎样和数据库交互
- 怎样用 MySQL 存储编程语言创建过程，函数和触发器

我们不会在本章中对主题进行深入，这些内容的主要目的仅仅是让你对存储过程程序有一个大体的映像，在随后的章节中，我们将对本章的内容进行升华

2.1 你所需要的工具

下面的工具是你运行本书中的示例代码所需要的工具：

- MySQL 5 server
- 一个文本编辑器（vi, emacs 或者 notepad）
- MySQL Query Browser（MySQL 官方 GUI TOOLS 内的查询工具）

您能够在 <http://dev.mysql.com> 上面得到 MySQL Server 及 MySQL Query Browser

2.2 第一个存储过程

开始，我们将创建一些非常简单的存储过程，你将需要一个编辑环境来写存储过程，还有一些工具把你的存储过程请求发送给 MySQL 服务器

你可以使用任何文本编辑器，下面提供了一些 MySQL 的代码提交工具：

- MySQL 命令行客户端
- MySQL Query Browser
- 第三方工具，例如 Toad for MySQL

着这篇文章中，我们假定你没有安装任何工具，所以我们是用了古老却又经典的 MySQL 命令行客户端

让我们用 root 帐户登录 localhost 的 3306 端口，我们将在 Example 2-1 使用 MySQL 预安装的“test”数据库

Example 2-1 连接 MySQL 命令行客户端

```
[gharriso@guyh-rh4-vm2 ~]$mysql -uroot -psecret -hlocalhost
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.16-nightly-20051017-log

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

mysql>
```

2.2.1 创建存储过程

你可以用 CREATE PROCEDURE , CREATE FUNCTION ,或者 CREATE TRIGGER 语句来创建存储程序。可以直接把这些语句直接输入 MySQL 命令行，但是对于一般的存储程序大小而言，这有些不太实际，所以我们建议你使用文本编辑器创建一个文本文件来容纳我们的存储程序，然后我们就可以使用命令行客户端和其他工具来递交这个文件

我们将使用 MySQL Query Browser 作为文本编辑器，如果你没有这个工具，你可以从 <http://dev.mysql.com/downloads/> 得到，你也可以使用任何操作系统上的编辑器例如 vi, emacs 或者 notepad，当然我们喜欢 MySQL Query Browser 的原因是它具备内建的帮助系统，语法高亮，执行 SQL 语句的能力以及其他一些功能

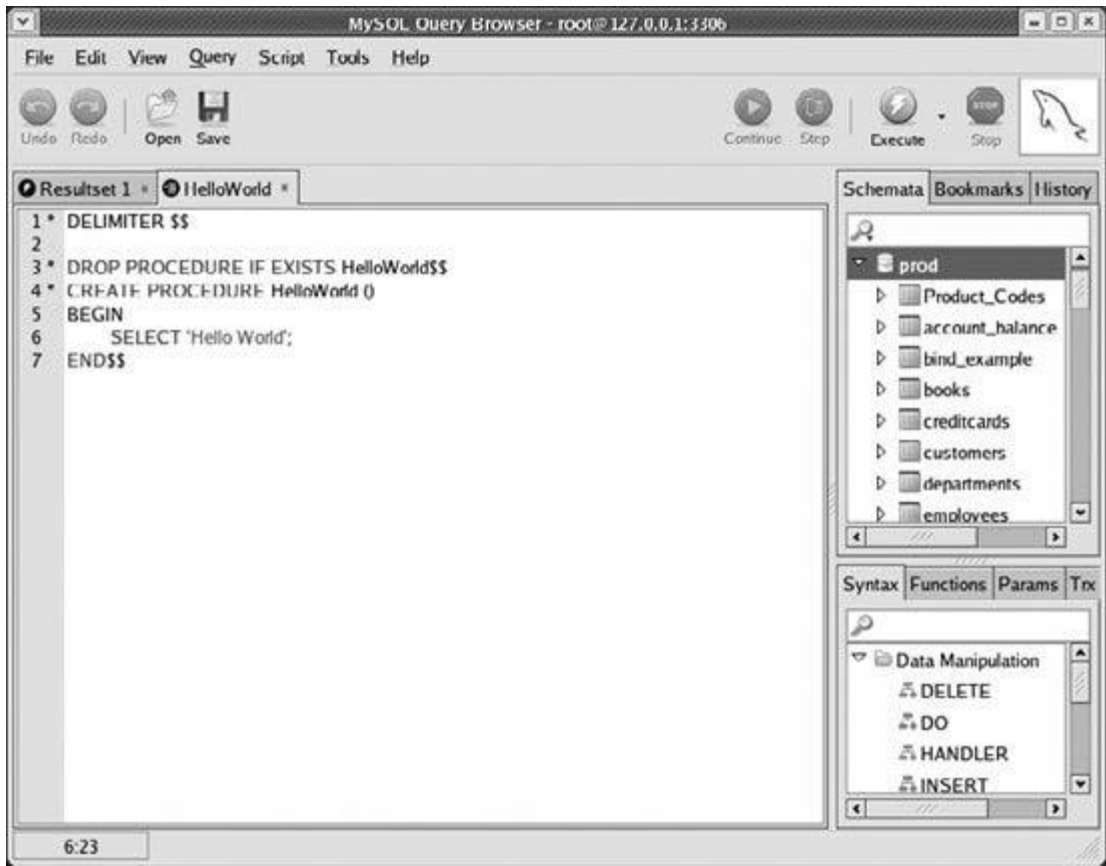
遵照下面的步骤

运行 MySQL Query Browser,在 Windows 上，从开始菜单中选择程序->MySQL->MySQL Query Browser。在 Linux 上从终端中输入 mysql-query-browser

从菜单中选择 File->New Script tab 来创建一个空白的脚本窗口

输入你的存储程序代码

Figure 2-1 显示了我们的第一个存储过程



我们可以使用 File->Sava As 菜单来把我们的文件保存，这样就可以用 mysql 客户端来执行它

第一个存储过程非常的简单，但是还是让我们一行行的来解释确保你能够完整的理解他们

行号	解释
1	DELIMITER 命令确保把 ‘\$\$’ 作为语句的终结条件，通常，MySQL 会把 “;” 作为语句的终结，但是因为存储过程在其过程体中
3	DROP PROCEDURE IF EXISTS 语句用来确保在同名存储过程已经存在的情况下将其移除，如果我们不这样做，那么在同名存储过程已存在的情况下将收到一个 MySQL 的修改重复执行的错误
4	CREATE PROCEDURE 语句指示一个存储过程定义的开始，注意,存储过程名 “HelloWorld” 的后面跟这一对内容为空的圆括号 “()”。如果存储过程有任何参数，那么我们就可以把参数放在里面。但是如果没有参数，我们同样要把圆括号放上，否则，我们将会收到一个语法错误
5	BEGIN 语句指示了存储程序的开始，所有超过一个语句的存储程序必须用至少一个 BEGIN-END 块来定义存储程序的开始和结束
6	这是存储过程中的一个单个语句：一个 SELECT 语句将 “Hello World” 返回给它的调用程序，马上将像我们看到的一样，存储程序中的 SELECT 能够向控制台和调用程序返回数据，就像我们直接把 SELECT 语句输入 MySQL 命令行一样
7	END 结束存储过程的定义。注意用\$\$来结束对存储过程的定义，这样 MySQL 就知道我们完整的结束了 CREATE PROCEDURE 语句

随着对存储过程的定义结束，我们可以用 mysql 客户端创建并执行我们的 HelloWorld 存储过程，就像 Example 2-2 所展示的那样

Example 2-2. 创建我们第一个存储过程

```
$ mysql -uroot -psecret -Dprod
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 16 to server version: 5.0.18-nightly-20051208-log

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

mysql> SOURCEHelloWorld.sql
Query OK, 0 rows affected, 1 warning (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALLHelloWorld( ) $$
+-----+
| Hello World |
+-----+
| Hello World |
+-----+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)

mysql>
```

下面是我们用于完成这一切 MySQL 命令的解释

命令	解释
SOURCE HelloWorld.sql	从指定的文件中读取命令，在这个例子中，我们用 MySQL Query Browser 制定了我们所保存的文件，没有错误返回，这说明我们已经成功的创建了存储过程
CALL HelloWorld() \$\$	执行存储过程，我们成功的执行了存储过程并且返回了“Hello World”作为结果集。我们用 ‘\$\$’ 来作为 CALL 命令的终结，这是因为 DELIMITER 的设置仍然在起作用

2.2.2 用 MySQL Query Browser 创建存储过程

在这个指南以及整本书中，我们要用一些过时的工具--MySQL 命令行终端创建大多数的存储程序代码示例。而你要做的仅仅是复制这些代码片段。因此，你可以是用一些图形化的工具来创建存储程序：网上有大量使用的第三方 MySQL 图形化工具，其中一个很好的选择是 MySQL Query Browser,你可以在此得到 <http://dev.mysql.com/downloads/>

在这一章节中我们将给出如何是用 MySQL Query Browser 创建存储过程，MySQL Query Browser 对于创建存储过程更为友好，虽然目前为止并非所有的操作系统平台都支持这一工具，所以你可以使用 MySQL 命令行或者其他第三方工具来代替

在 Windows 上，从开始菜单中选择程序->MySQL->MySQL Query Browser。在 Linux 上从终端中输入 mysql-query-browser

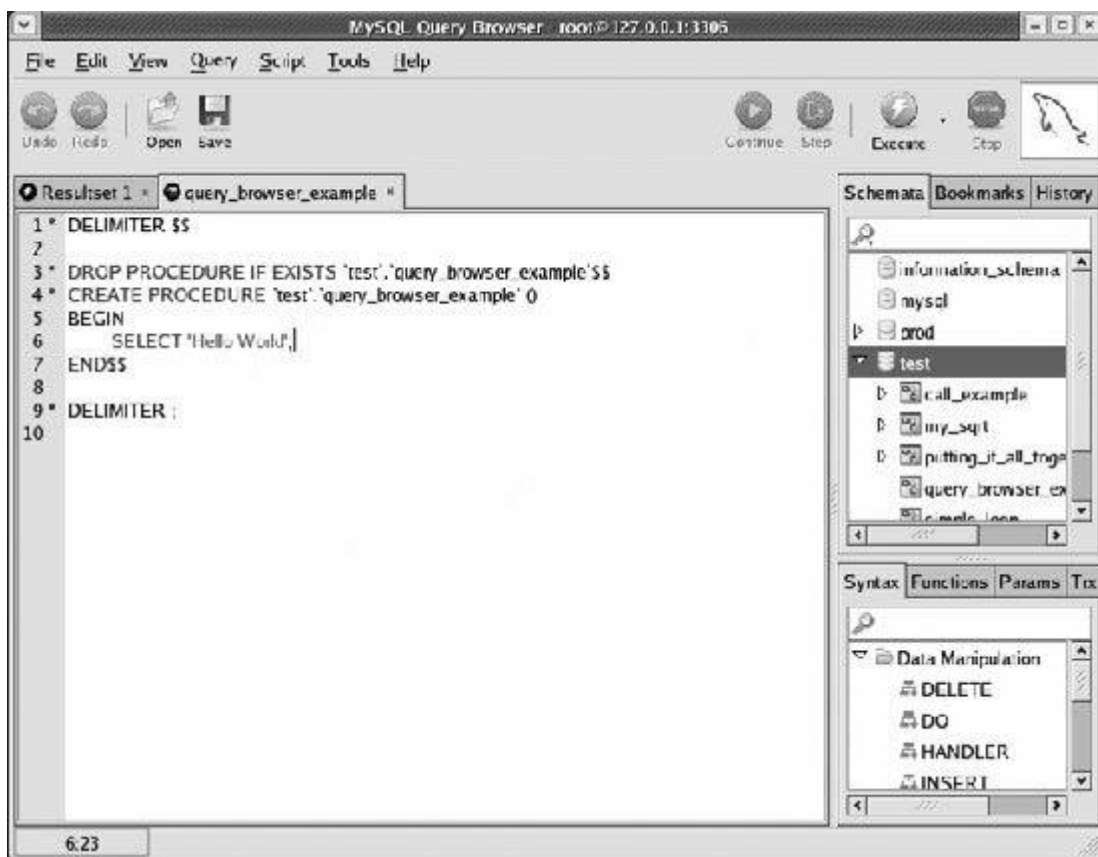
当查询工具被打开，它会提示你输入 MySQL 服务器的连接信息，然后将显示一个空白的图形化窗口。你可以使用菜单项 Script->Create Stored Procedure/Function 菜单创建存储程序，它会提示你按照名称来创建存储程序，然后会显示一个空白的存储程序模板 Figure 2-2 显示了这样一个模板的例子

Figure 2-2 用 MySQL Query Browser 创建存储过程



你可以在适当的位置存储过程代码（在 BEGIN 和 END 语句之间，光标将被自动的置于合适的位置方便你的输入）。当你输入完成，你可以简单的按下 Execute 按钮来执行存储过程，如果你的代码发生了错误，Query Browser 将在底部显示错误并用高亮标识发生错误的行，否则，你将在左侧的 Schemata 选项卡中发现你的存储过程已被成功的创建

Figure 2-3 用 Execute 按钮执行存储程序

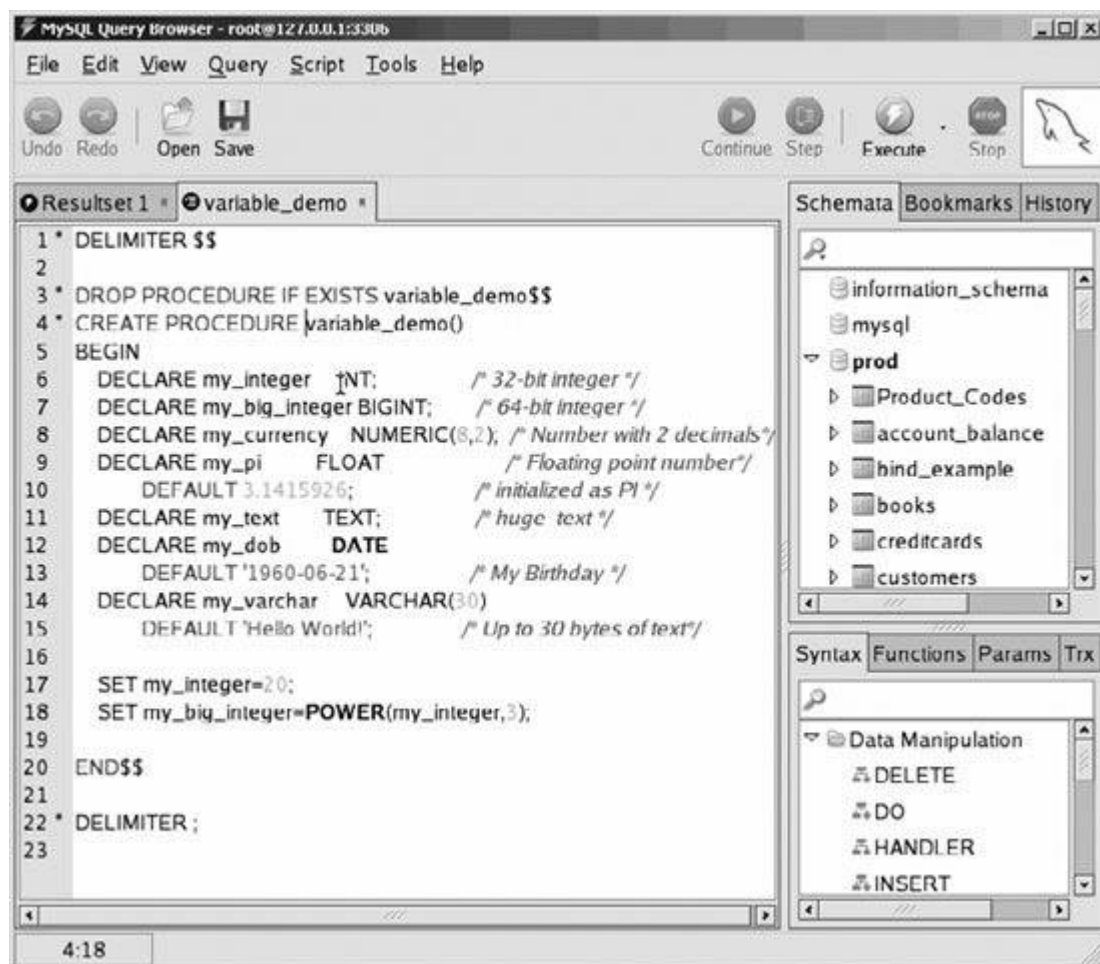


我们希望这个清晰的示例对于你用 MySQL Query Browser 创建和执行存储程序起到帮助，Query Browser 提供了一个简便的存储程序开发环境，但这一切都取决于你如何使用 Query Browser，第三方工具和你喜欢的编辑器及 MySQL 命令行终端

2.3 变量

本地变量可以用 DECLARE 语句进行声明。变量名称必须遵循 MySQL 的列名规则，并且可以使 MySQL 内建的任何数据类型。你可以用 DEFAULT 语句给变量一个初始值，并且可以用 SET 语句给变量赋一个新值，就像 Figure 2-5 所展示的那样。

Figure 2-5.在存储过程中使用变量



2.4 参数

我们大多数所写的存储程序都会包括一两个参数。参数可以使我们的存储程序更为灵活，更为实用，下面，让我们创建一个包含参数的存储过程

Figure 2-4 在 Query Browser 中执行存储过程

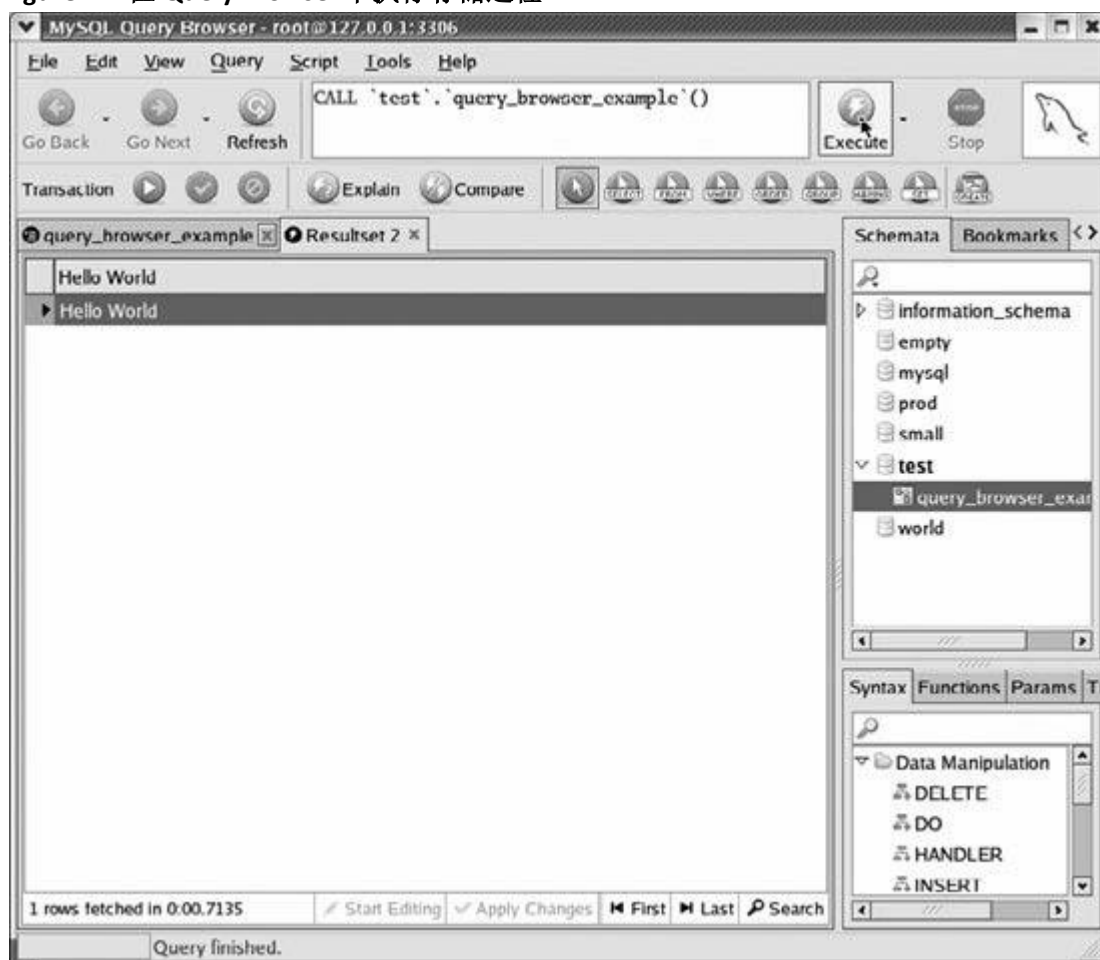


Figure 2-6 的存储过程接受一个整型数 `input-number` 作为参数,并且计算出了这个数的平方根，计算出的结果作为返回的结果集

把参数放置在紧随过程名的圆括号内，每一个参数都有自己的名称，数据类型还有可选的输入输出模式，有效的模式包括 `IN`（只读模式），`INOUT`（可读写模式）和 `OUT`（只写模式）。因为 `IN` 模式作为缺省的参数模式，所有没有出现在 Figure 2-6 当中

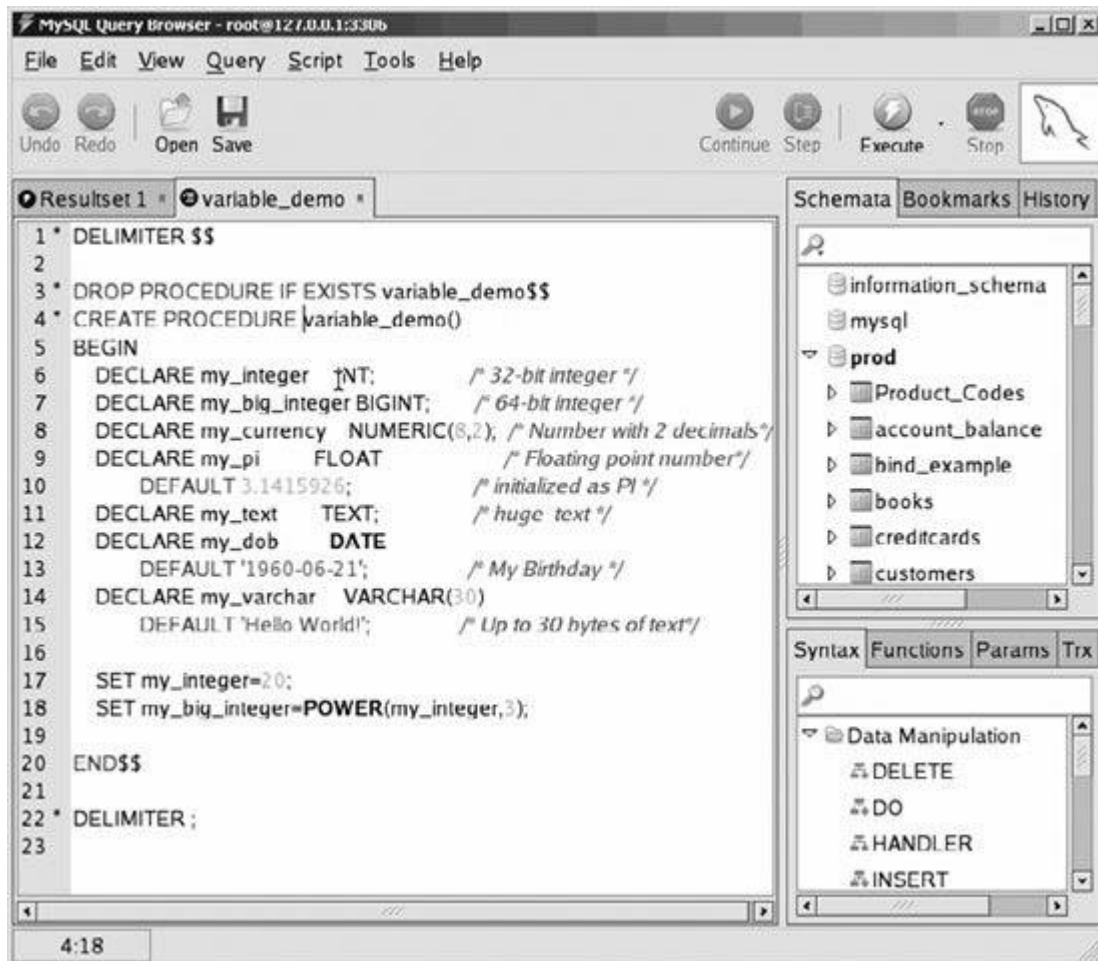
我们将通过示例对参数模式进行细致的观察

此外，MySQL 存储程序引入了两种有关参数的不同的特性：

`DECLARE`

一个用于创建存储程序内部使用的本地变量，在这个示例中，我们创建了一个名为 `l_sqrt` 的浮点数。

Figure 2-5 在存储过程中使用变量



SET

一个用来给变量赋值的语句，在这个示例中我们将参数的平方根（使用内置的 `SQRT` 函数）赋予那个用 `DECLARE` 命令声明的变量

我们可以在 MySQL 客户端中执行并测试存储过程的运行结果，就像 Example 2-3 所做的那样

Example 2-3 创建并执行使用参数的存储过程

```

mysql> SOURCEmy_sqrt.sql
Query OK, 0 rows affected (0.00 sec)

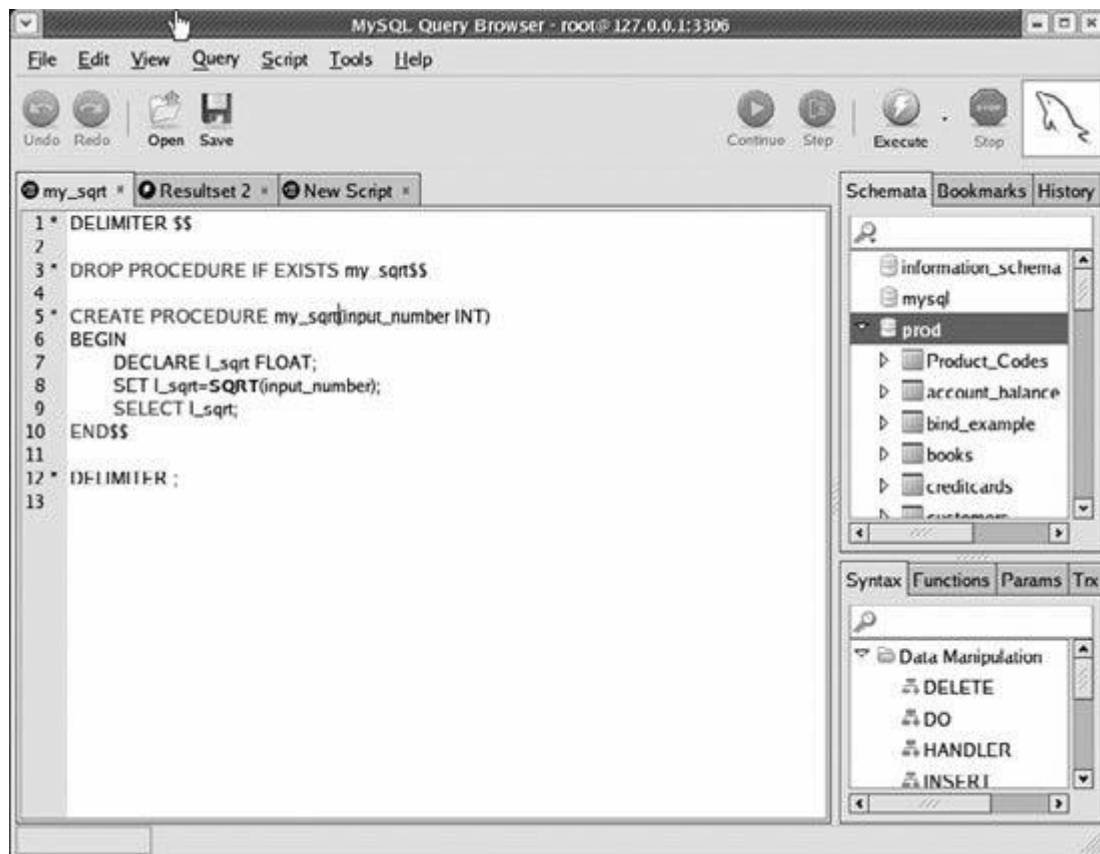
Query OK, 0 rows affected (0.00 sec)

mysql> CALLmy_sqrt(12)$$
+-----+
| l_sqrt |
+-----+
| 3.4641016151378 |
+-----+
1 row in set (0.12 sec)

Query OK, 0 rows affected (0.12 sec)

```

Figure 2-6 一个使用参数的存储过程



2.4.1 参数模式

MySQL 的参数模式可以被定义为 IN, OUT 和 INOUT。

IN

这是缺省的模式，它说明参数可以被传入存储程序内部，但是任何对于该参数的修改都不会被返回给调用它的程序

OUT

这个模式意味着存储程序可以对参数赋值（修改参数的值），并且这个被修改的值会被返回给它的调用程序

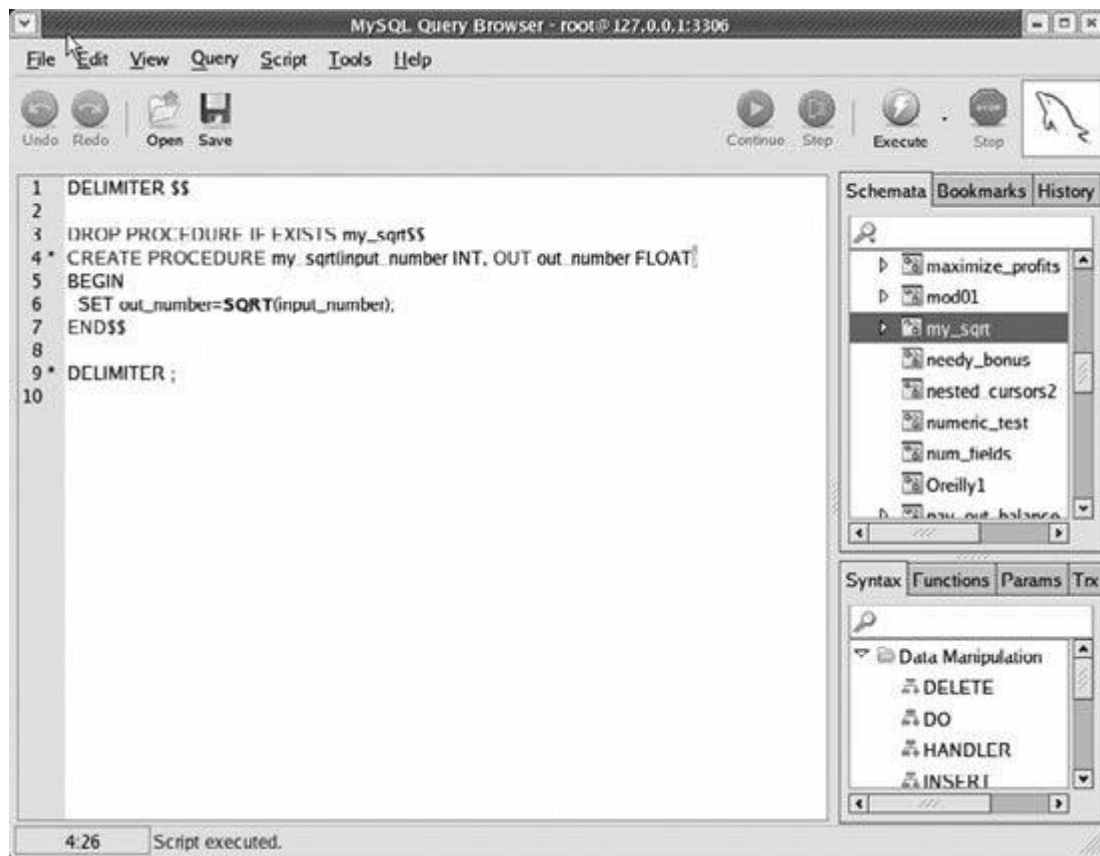
INOUT

这个模式意味着存储程序既可以读取传入的参数，而且任何对于该参数的修改对于它的调用程序而言都是可见的

你可以在存储过程的参数中使用上述所有的模式，但是对于存储函数而言，你只能使用 IN 模式（参考随后的“存储函数”章节）

让我们改变这个平方根程序，使它将计算结果放到 OUT 值中去，就像 Figure 2-7 所做的

Figure 2-7 在存储过程中使用 OUT 参数



在 MySQL 客户端中，我们提供了一个用来保存值的 OUT 参数，当存储过程执行完毕，我们可以回头检验这个变量的输出情况，就像 Example 2-4.

Example 2-4 创建和执行使用 OUT 参数的存储过程

```

mysql> SOURCEmy_sqrt2.sql
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.02 sec)

mysql> CALLmy_sqrt(12,@out_value) $$
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT@out_value $$
+-----+
| @out_value |
+-----+
| 3.4641016151378 |
+-----+
1 row in set (0.00 sec)

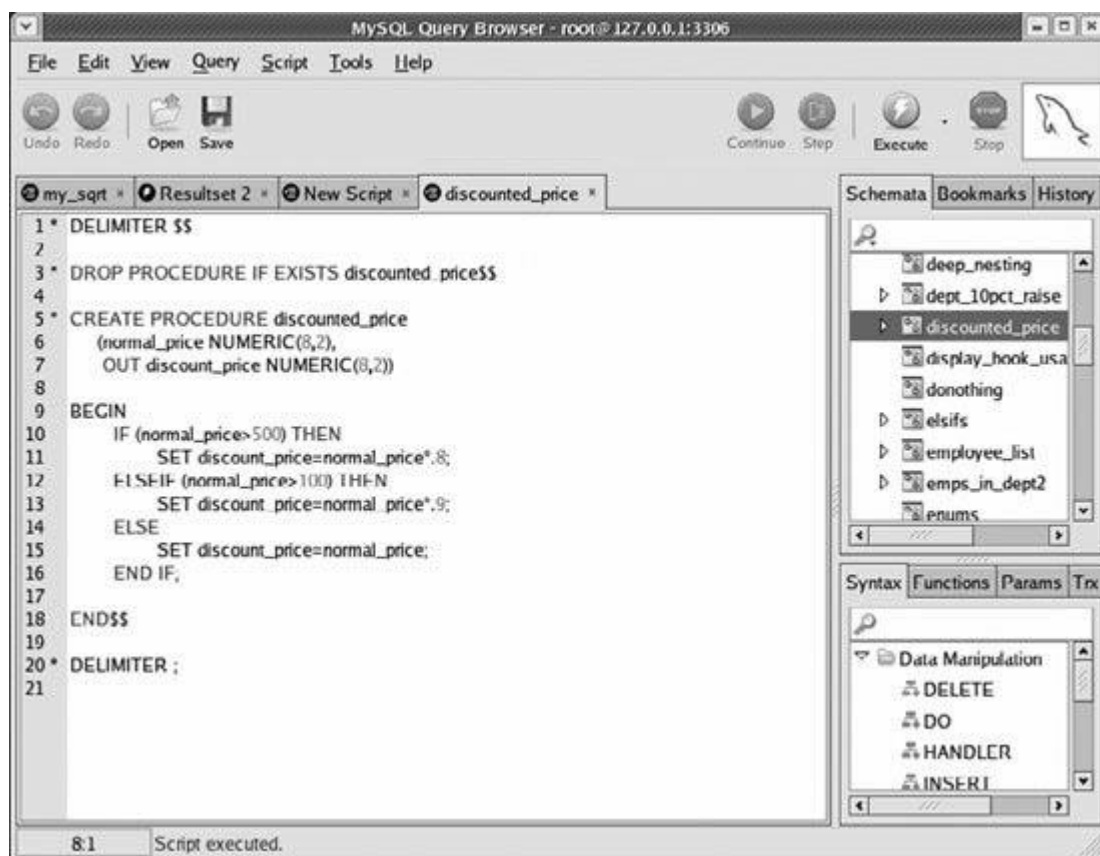
```

2.5 条件执行

你可以用 IF 或者 CASE 语句来控制存储程序的执行流程。它们具有相同的功能，因此，我们只用示例演示了 IF（CASE 的功能是相同的）

Figure2-8 演示了通过购买量的多少来计算出贴现率的存储程序，Example2-5 演示了它的执行结果，购买量超过\$500 可以返还 20%，购买量超过\$100 可以返还 10%。

Figure 2-8 使用 IF 语句的条件执行



Example 2-5 创建和执行包含 IF 语句的存储过程

```

mysql> SOURCEdiscounted_price.sql
Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> CALLdiscounted_price(300,@new_price) $$
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT@new_price$$
+-----+
| @new_price |
+-----+
| 270.0      |
+-----+

```

```
1 row in set (0.00 sec)
```

IF 语句允许你测试表达式的真实性（就像 `normal_price > 500`），并且基于表达式的结果执行一定的行为，作为一种编程语言，ELSEIF 可以被用来作为 IF 起始循环的条件转移，ELSE 语句将在 IF 和 ELSEIF 语句的布尔表达式为假时执行

CASE 具有相同的功能，并且当你对单个表达式进行对比是可以获得更清晰的值，这两个条件语句将在第 4 章做更为细致的探究和比对。

2.6 循环

循环允许在你的存储程序中重复性的执行某些行为，MySQL 存储程序语言提供了三种类型的循环

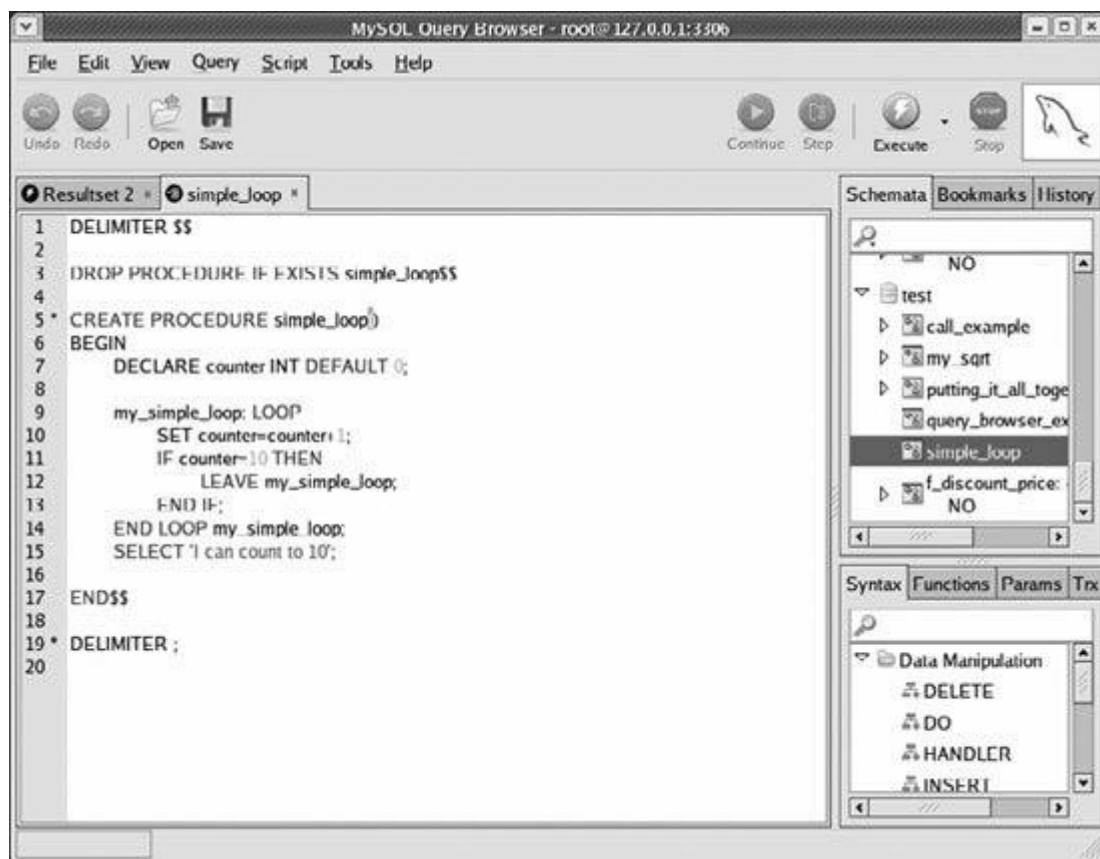
- 使用 LOOP 和 END LOOP 字句的简单循环
- 当条件为真时继续执行的循环，使用 WHILE 和 END WHILE 字句
- 循环直至条件为真，使用 REPEAT 和 UNTIL 字句

在这三种循环中，你都可以使用 LEAVE 子句来终止循环

在三种循环都将在第 4 章详细解释；我们只会在这个指南中给出 LOOP-LEAVE-END LOOP（简单循环）的例子

Figure 2-9 简单循环演示

Figure 2-9.存储过程中的简单循环



下面是对代码的详细解释

行号	解释
7	声明了一个名为 counter，初始值为 0 的简单数字变量
9-14	简单循环，所有在 LOOP 和 END LOOP 之间的部分都将在 LEAVE 子句被执行后终止
9	LOOP 语句带有前缀为 my_simple_loop 的标签，LEAVE 子句要求循环被标识，这样才能知道要退出哪个循环
10	给 counter 变量的值增加 1
11-13	测试 counter 的值，如果值为 10，则退出循环，否则，我们继续下一个迭代
15	我们骄傲的宣称我们可以数到 10

2.7 错误处理

当存储程序发生错误时，MySQL 默认的行为是终止程序的执行并把错误返回给它的调用程序。如果你需要以不同的方式来相应错误，你可以定义一个或多个可以被存储程序所响应的错误情况

如下两个相关联的情景被称为错误处理的定义：

- 如果你认为内嵌的 SQL 语句会返回空记录，或者你想用游标捕获所有 **SELECT** 语句所返回的记录，那么一个 **NOT FOUND** 错误处理可以防止存储程序过早的被终止
- 如果你认为 SQL 语句可能返回错误（比如：违背约束条件），你可以创建一个错误处理来阻止程序终止。这个处理将代替你的默认错误处理并继续程序的执行。

第 6 章将详细解释错误处理，在下一节中我们将演示一个使用 **NOT FOUND** 错误处理并结合游标显示的例子

2.8 和数据库交互

大多数存储过程包含了各种和数据库表的交互，它们包括四种主要的交互：

- 将一个 SQL 语句所返回的单个记录放入本地变量中
- 创建一个“游标”来迭代 SQL 语句所返回的结果集
- 执行一个 SQL 语句，将执行后的结果集返回给它的调用程序
- 内嵌一个不返回结果集的 SQL 语句，如 INSERT, UPDATE, DELETE 等

我们暂时来大致的看一下这几种和数据库交互的情况



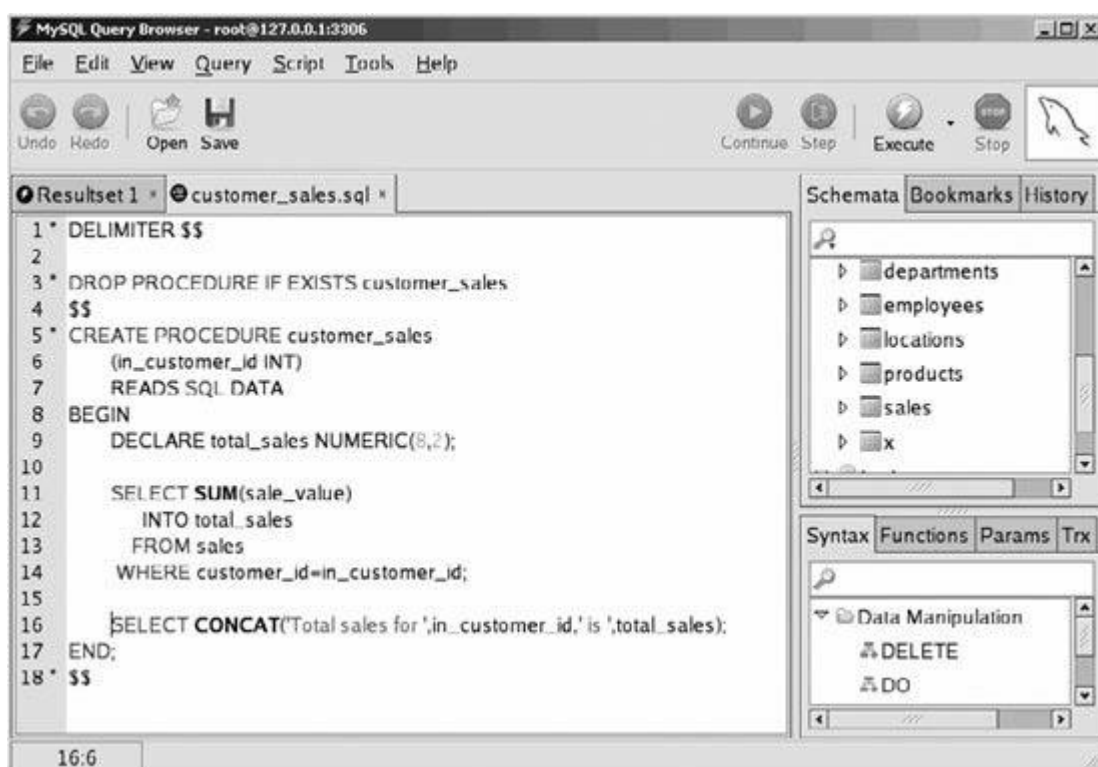
为了能运行本节的示例，你必须安装和本书配套的 sample 数据库，这个可以在本书的网站找到（详见前言）

2.8.1 对本地变量使用 SELECT INTO

当需要在单个记录数据中获取查询信息，你就可以使用 SELECT INTO 语法（无论是使用单个记录，多个记录的混合数据，还是多个表连接）。在这种情况下，你可以在 SELECT 语句中跟随一个 INTO 子句，告诉 MySQL 得到的查询数据返回给谁

Figure 2-10 演示了更具 customer ID 的不同来获取和显示销售量的存储过程。 Figure 2-6 是执行结果

Figure 2-10 在存储过程中使用 SELECT INTO 语句



Example 2-6 执行包含 SELECT INTO 语句的存储过程

```
mysql> CALL customer_sales(2) $$
+-----+
| CONCAT('Total sales for ',in_customer_id,' is ',total_sales) |
+-----+
| Total sales for 2 is 7632237                                |
+-----+
1 row in set (18.29 sec)

Query OK, 0 rows affected (18.29 sec)
```

2.8.2 使用游标

SELECT INTO 定义了单记录查询，但是很多应用程序要求查询多记录数据，你可以使用 MySQL 中的游标来实现这一切，游标允许你将一个或更多的 SQL 结果集放进存储程序变量中，通常用来执行结果集中各个但记录的处理。

在 Figure 2-11 中，存储程序使用游标来捕获所有 employees 表的记录

下面是对于代码的详细解释

Figure 2-11. 在存储过程中使用游标



行号	解释
8-12	声明本地变量，前面的三个是用来存放 SELECT 语句的结果。第四个（done）能让我们确认所有的记录行都被读取
14-16	定义我们的游标，这是基于一个简单 SELECT 语句从 employees 表中所返回的结果集
18	声明一个“handler”，它定义了当我们无法从 SELECT 语句得到更多记录时的行为。handler 可以用来捕获所有的错误类型，但是像示例中所演示的 handler 只是在我们需要的时候警告我们已经没有更多的记录可以被读取而已
20	打开游标
21-26	用一个简单循环来从游标中获取所有的记录
22	用 FETCH 子句将从游标中获取单个记录，然后放进我们的本地变量中。
23-25	检测变量 done 的值，如果它被设置成 1，那么就说明我们已经获取了最后一个数据，那么我们就用 LEAVE 语句来终止循环。

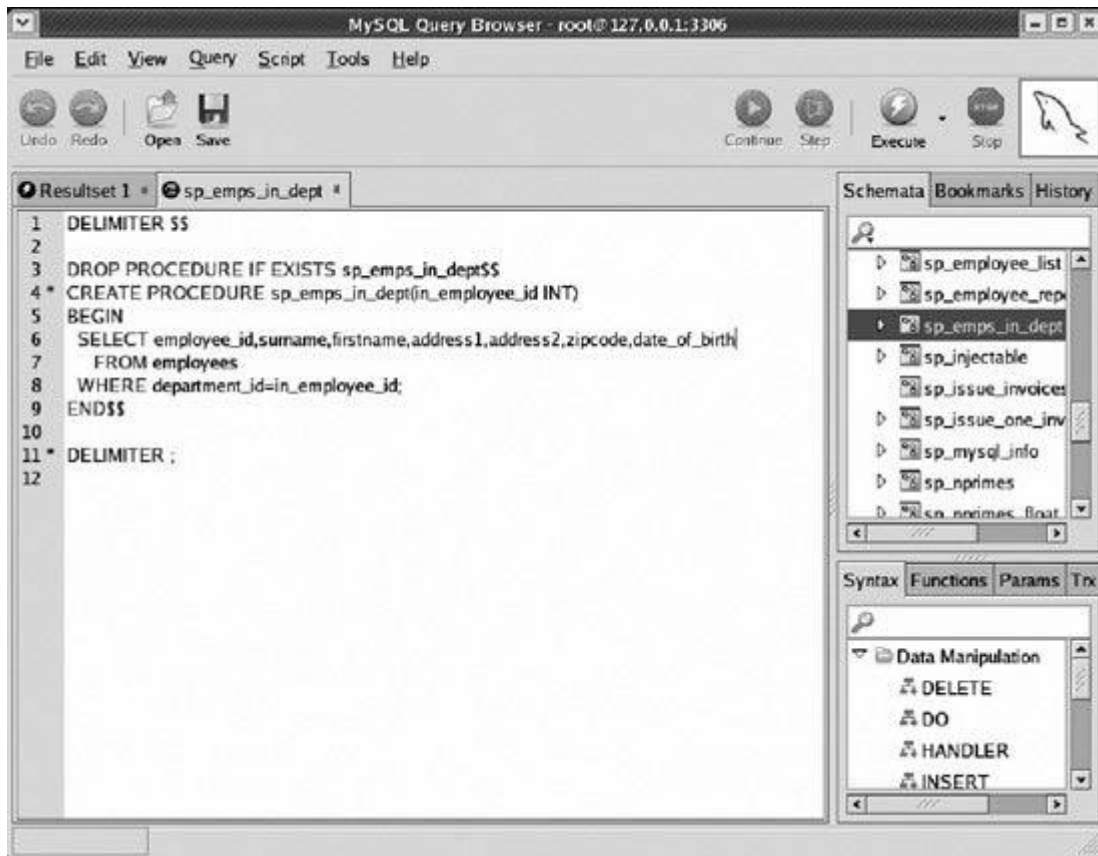
2.8.3 返回结果集的存储过程

在这本书的前些部分，我们已经在和存储过程和数据库的交互中使用过一些并不包含 INTO 子句和游标的沉长的 SELECT 语句，它们被用于在存储过程中返回一些状态数据和结果集，迄今为止，我们只使用过单记录结果集，但是你也可以在存储过程中包含一些复杂的 SQL 语句来返回多个结果。

如果我们在 MySQL 命令行中执行这样的存储过程，结果集将像我们执行 SELECT 和 SHOW 语句一个被返回。Figure 2-12

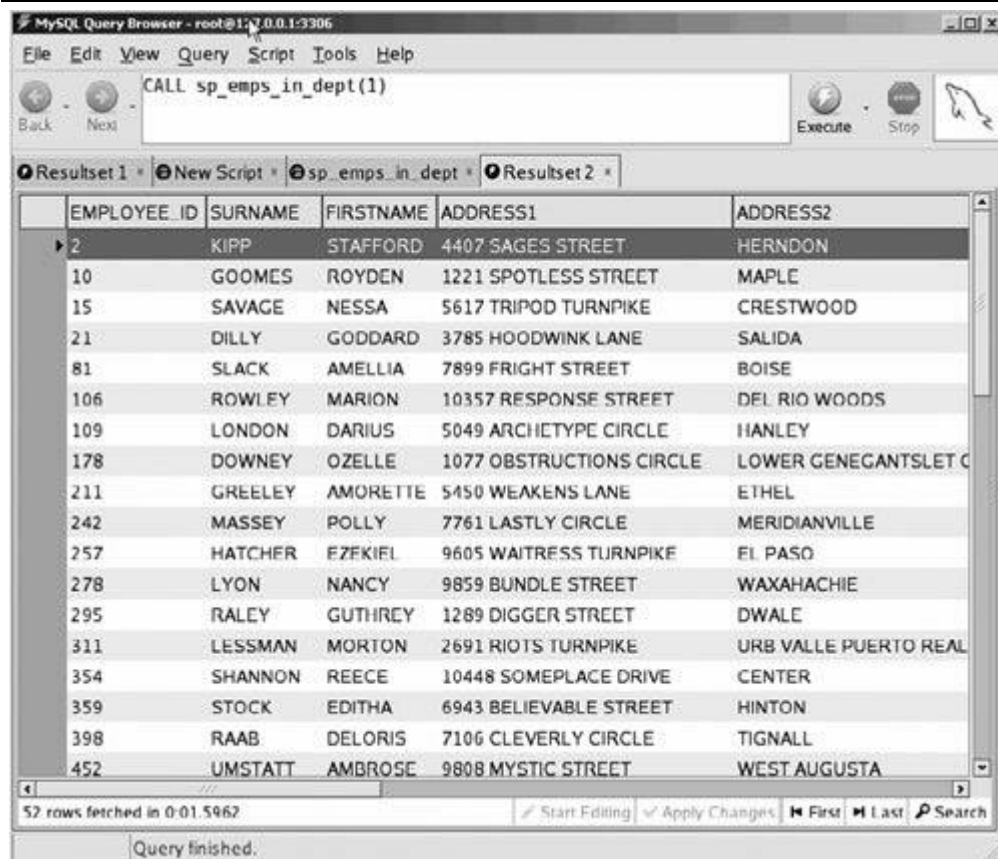
向我们展示了包含了沉长的 SELECT 语句的存储过程

Figure 2-12 包含沉长 SELECT 语句的存储过程



如果我们在执行这个存储过程时为其参数提供适当的值，那么包含 SELECT 的存储过程将被返回，在 Figure 2-13 中我们将看到用 MySQL Query Browser 执行的包含 SELECT 语句的存储过程将结果返回的状况。

Figure 2-13 包含沉长 SELECT 语句的存储程序的结果返回状况



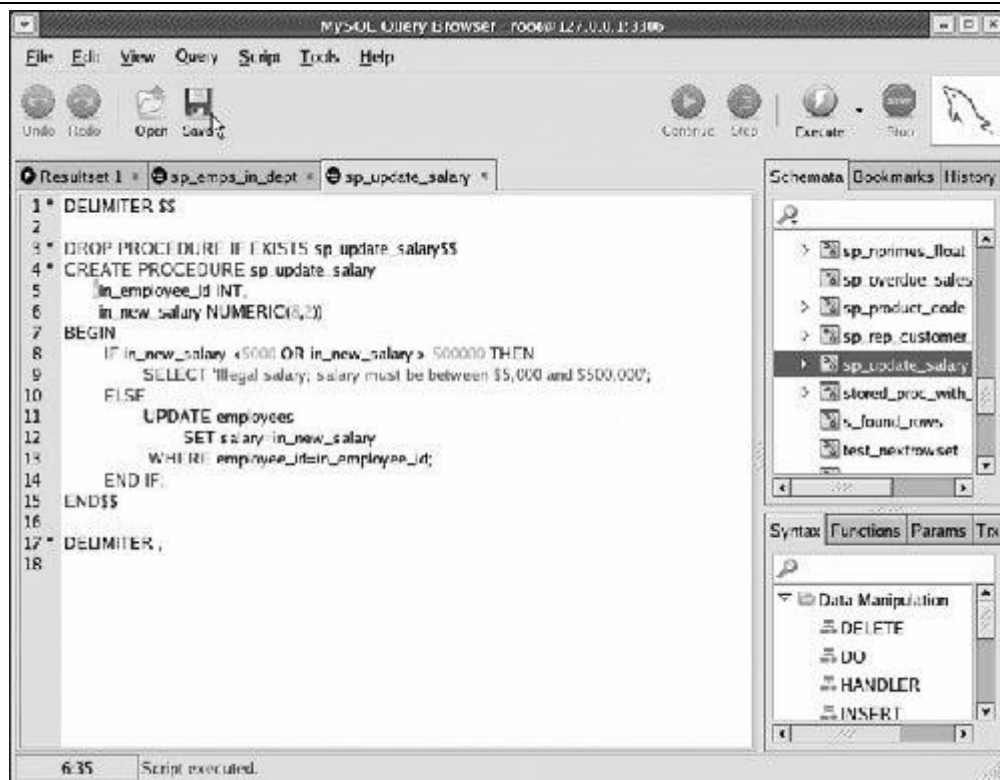
注意：存储过程调用可能返回多个结果集，这给程序调用带来了挑战，我们将在第 13 章到第 17 章中的每个部分中分别来介绍相关的情况

2.8.4 内建不返回结果的 SQL 语句

不返回结果集的“简单”的 SQL 语句也可以被嵌入存储程序中，包含 DML（数据操纵语言）如 UPDATE、INSERT 和 DELETE 以及 DDL（数据定义语言）如 CREATE TABLE 等都可以被包括在内，当然，某些特定的用来创建和控制存储程序的语句并不允许被使用，这将在第 5 章阐述

Figure 2-14 演示了包含更新操作的存储过程，其中的 UPDATE 语句被某些验证逻辑所封装来达到阻止非有效数据输入的目的

Figure 2-14 内嵌 UPDATE 的存储过程

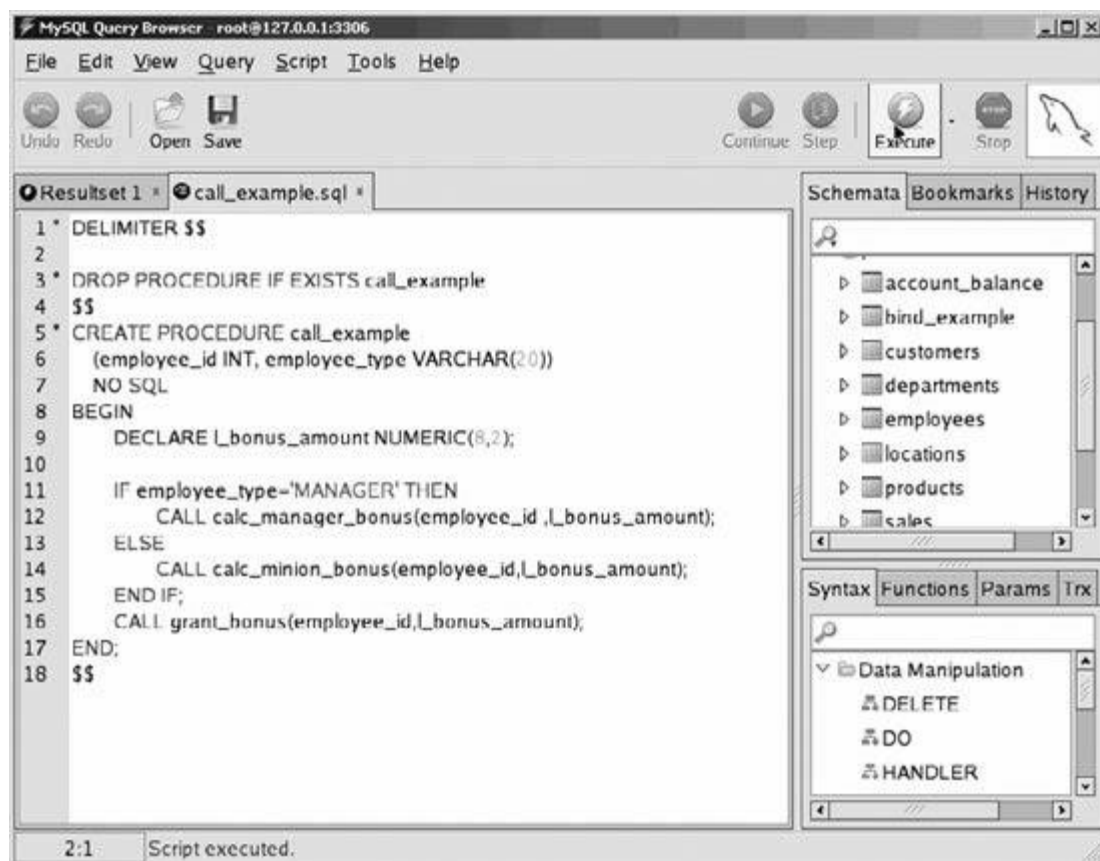


2.9 在其他存储程序中调用存储程序

在其他存储程序中调用存储程序是很简单的。就像你在 MySQL 命令行客户端中所做的一样，仅仅需要使用 CALL 语句

Figure 2-15 演示了一个通过输入参数来选择不同存储过程的简单的例子。存储过程的输出（l_bonus_amount 是一个外界传入的 OUT 类型的参数）将被传递给第三个过程

Figure 2-15 从另一个存储过程中调用存储过程



下面是对代码段的解释

行号	解释
11	判断 employee 是否是 manager，如果他是 manager，我们就调用存储过程 calc_manager_bonus；如果他不是 manager，那么我们就调用 calc_minion_bonus
12 和 14	这两个存储过程都被传递入 employee_id，并且提供了变量 l_bonus_amount 来接受存储过程的输出
16	用 employee_id 和 bonus amount（用我们在 12 或 14 行计算出的值）作为参数调用存储过程 grant_bonus

2.10 把所有的东西组装起来

Example 2-7 向我们演示了所有已经提到过的存储程序语言特性的例子

Example 2-7. 更为复杂的存储过程

```

1 CREATE PROCEDURE putting_it_all_together(in_department_id INT)
2     MODIFIES SQL DATA
3 BEGIN
4     DECLARE l_employee_id INT;
5     DECLARE l_salary      NUMERIC(8,2);
6     DECLARE l_department_id INT;
7     DECLARE l_new_salary  NUMERIC(8,2);
8     DECLARE done          INT DEFAULT 0;
9
10    DECLARE cur1 CURSOR FOR
11        SELECT employee_id, salary, department_id
12        FROM employees
13        WHERE department_id=in_department_id;
14
15
16    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;
17
18    CREATE TEMPORARY TABLE IF NOT EXISTS emp_raises
19        (employee_id INT, department_id INT, new_salary NUMERIC(8,2));
20
21    OPEN cur1;
22    emp_loop: LOOP
23
24        FETCH cur1 INTO l_employee_id, l_salary, l_department_id;
25
26        IF done=1 THEN      /* No more rows*/
27            LEAVE emp_loop;
28        END IF;
29
30        CALL new_salary(l_employee_id,l_new_salary); /*get new salary*/
31
32        IF (l_new_salary<>l_salary) THEN              /*Salary changed*/
33
34            UPDATE employees
35                SET salary=l_new_salary
36                WHERE employee_id=l_employee_id;
37            /* Keep track of changed salaries*/
38            INSERT INTO emp_raises (employee_id,department_id,new_salary)
39                VALUES (l_employee_id,l_department_id,l_new_salary);
40        END IF;

```

```

41
42     END LOOP emp_loop;
43     CLOSE curl;
44     /* Print out the changed salaries*/
45     SELECT employee_id,department_id,new_salary from emp_raises
46     ORDER BY employee_id;
47 END;

```

这是我们迄今写过的最为复杂的存储过程，所以让我们一行行的解释

行号	解释
1	创建一个过程。它包括一个参数 <code>in_department_id</code> 。因为我们没有制定 <code>OUT</code> 或 <code>INOUT</code> 模式，所以输入参数是 <code>IN</code> 模式的（这意味着，调用程序不能读取任何参数在存储过程内的改动状况）。
4-8	声明在存储过程内部的本地变量。最后一个参数 <code>done</code> 给出了一个初始值 0。
10-13	创建一个用于获取 <code>employees</code> 表中数据行的游标。存储程序只获取那些被参数 <code>department_id</code> 传入的员工
16	创建一个用于处理“not found”情况的错误处理，这样当最后一个数据行被游标捕获，程序就不会被错误所终止。
18	创建一个临时表来保存被存储过程所作用的数据列表。这个表就像所有的临时表一样被创建于会话中，也就是说会随着会话的结束所终结。
21	为返回的记录结果打开游标
22	创建一个循环来为每个由存储过程返回的结果执行行为，这个循环终止与 42 行
24	将游标中的结果集装入我们在存储过程内部预定义的本地变量。
26-28	声明一个 IF 条件以确保当变量 <code>done</code> 的值为 1 时终止循环（这个工作由“not found”处理来完成，意味着所有的行都已被捕获）。
30	调用存储过程 <code>new_salary</code> 来计算出员工的新薪水。它接受一个 <code>employee_id</code> 作为输入参数和一个接受新薪水（ <code>l_new_salary</code> ）的输出参数。
32	将 30 行中由调用过程计算出的员工新薪水和在第 10 行由游标返回的薪水值相比对。如果不同，则执行第 32 到 40 行的代码块
34-36	用过程 <code>new_salary</code> 计算出的新薪水值代替现有的员工薪水。
38 和 39	在临时表中插入数据行来记录薪水的调整状况（定义域 21 行）
43	在所有的数据行都被处理完毕后，关闭游标
45	用一个沉长的 <code>SELECT</code> 语句（比方说没有 <code>WHERE</code> 子句的）来作用于临时表，获取哪个员工的薪水被更新。因为这个 <code>SELECT</code> 语句没有使用游标或 <code>INTO</code> 子句，所以数据行将被作为结果集返回给它的调用程序
47	终止存储过程

当这个存储过程被传入一个参数 18，并在 MySQL 命令行客户端中执行时，一个更新的薪水列表将被打印，见 Example 2-8

Example 2-8 样例“putting it all together”的输出情况

```

mysql> CALL cursor_example2(18) //
+-----+-----+-----+
| employee_id | department_id | new_salary |
+-----+-----+-----+
|          396 |           18 |  75560.00 |
|          990 |           18 | 118347.00 |

```



```
+-----+-----+-----+
2 rows in set (0.23 sec)

Query OK, 0 rows affected (0.23 sec)
```

2.11 存储函数

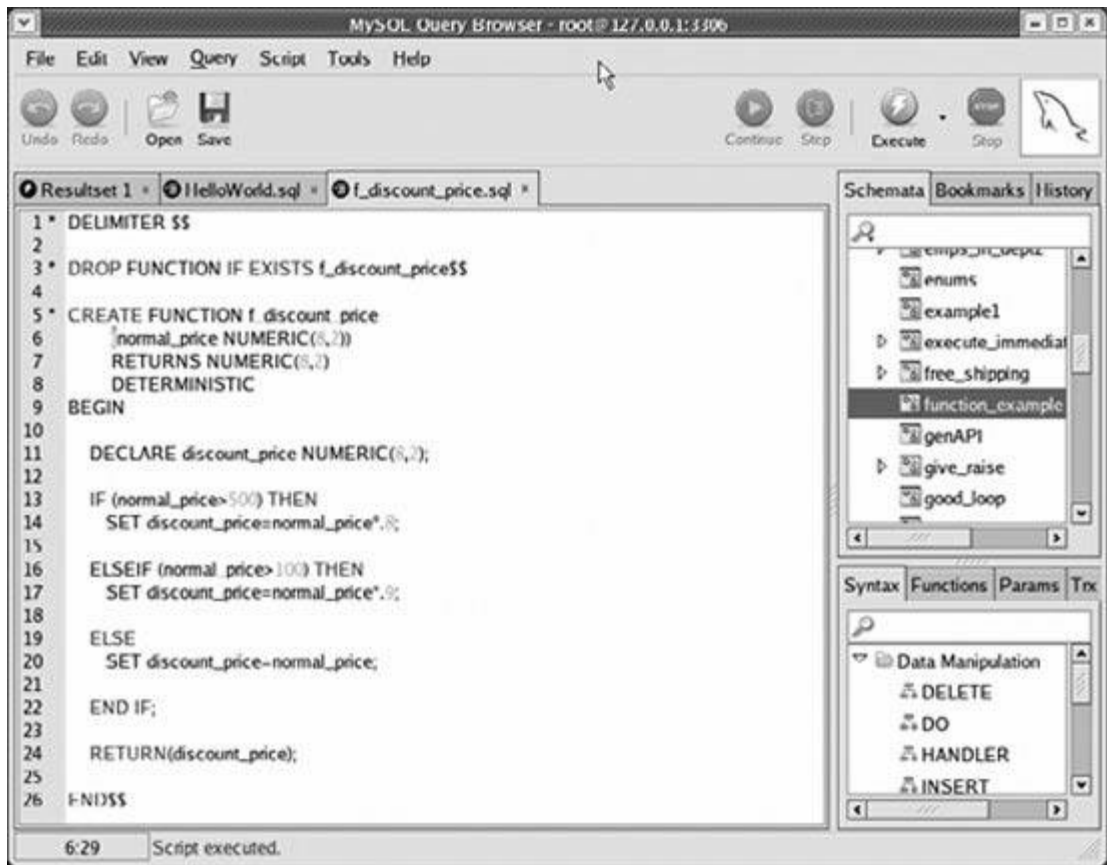
存储函数和存储过程很相像：它们都是包含一个或多个 MySQL 语句的被命名程序单元。和存储过程不同的地方有以下几点：

- 函数的参数列表中模式只能为 IN。OUT 和 INOUT 参数不被允许。制定 IN 关键字是被允许也是缺省的
- 函数必须返回一个值，它的类型被定义于函数的头部
- 函数能被 SQL 语句所调用
- 函数可能不返回任何结果集

大体来说，在程序的真正目的是比对值和需要返回值时或者你希望在 SQL 语句中创建用户自定义函数的时候更多的考虑使用存储函数，而不是存储过程

Figure 2-16 演示了用一个存储函数来实现我们在早些章节中提到过的 discount_price 存储过程所有的一些功能

Figure 2-16. 存储函数



下表揭示了一些在这个存储函数中和存储过程相等价的事物：

行号	解释
7	作为函数的定义，制定 RETURNS 子句。它制定了函数的返回类型
8	MySQL 相较于存储过程，对于存储函数有更严格的规则。函数必须声明不修改 SQL（使用 NO SQL 或者 READS SQL DATA 子句）或者声明为 DETERMINISTIC（如果服务器被允许开启二进制日志）。这个限制是为了防止当函数返回不确定值时，数据库同步复制的不一致性（详见第 10 章），我们的样例例程使用了“deterministic”，这样我们就能确保在提供了相同的参数的情况下返回相同的值

Example 2-9 展示了在 SQL 语句中调用这个函数

Example 2-9 在 SQL 语句中调用函数

```
mysql> SELECT f_discount_price(300) $$
+-----+
| f_discount_price(300) |
+-----+
|           270.0      |
+-----+
```

我们可以在其他存储程序中（过程，函数和触发器）或者任何我们可以使用 MySQL 内建函数的地方。

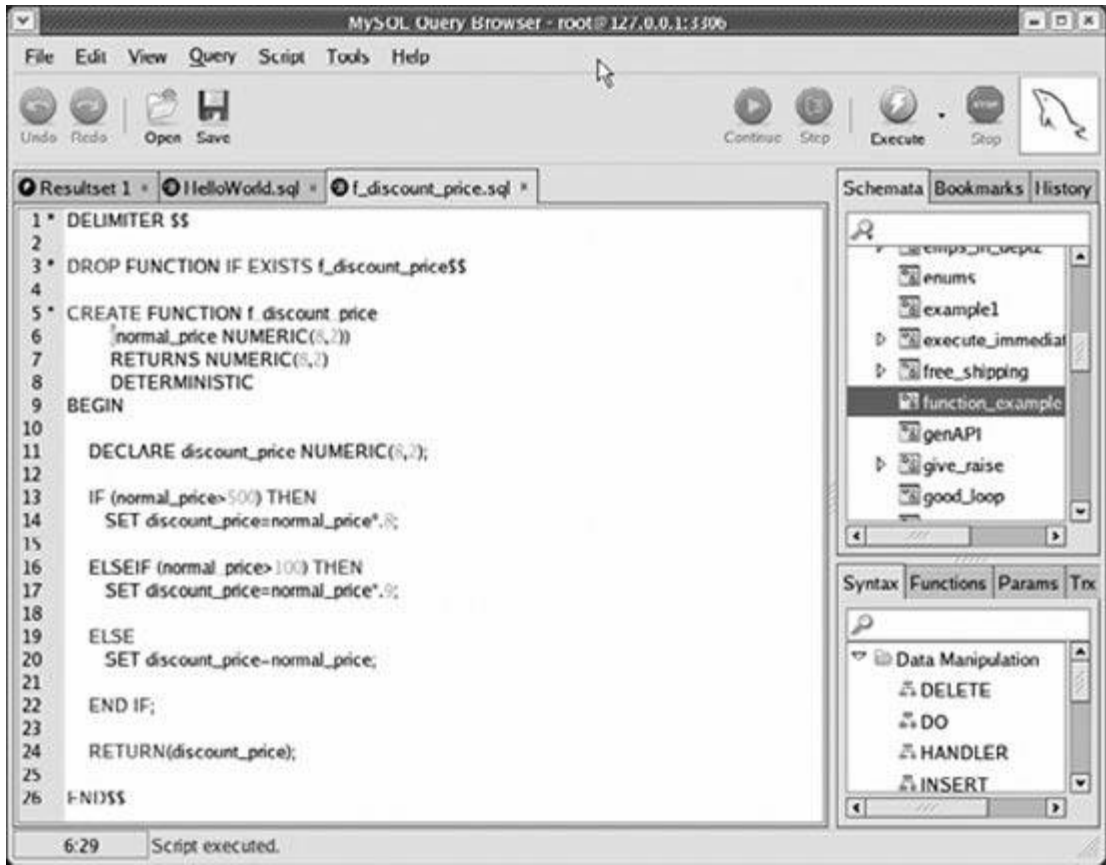
2.12 触发器

触发器是一种在数据库表被 INSERT，UPDATE 或者 DELETE 等(DML)语句所作用时所激活的特殊存储程序。只要表发生改变就会激活触发器的功能，因为触发器直接依附于表，所以程序代码无法绕过数据库触发器（没有办法让触发器失效）。

通常，触发器被用来实现严格的商业逻辑，高效的反向格式化数据和审核表的更改状况。触发器可以被定义触发于特定的 DML 语句执行之前或之后。

在 Figure 2-17，我们创建了一个在对 sales 表进行 INSERT 操作完成前的触发器。它将自动对指定值进行零售和折扣率的计算

Figure 2-17 一个数据库触发器



下面是对触发器定义的解释

行号	解释
5	定制触发器名称
6	指定触发器将于 sales 表的 insert 操作之前被激活
7	在当前的代码段中 FOR EACH ROW 子句说明触发器的语句将对每个插入到 sales 表中的行发生作用
8	使用 BEGIN 来开始包含由触发器执行的代码
9-13	如果 sale_value 的值大于\$500,则设置 free_shipping 列的值为 ‘Y’，否则，设置为 ‘N’。
15-19	如果 sale_value 的值大于\$1000，则对该值进行 15%的折扣率计算并插入到 discount 列中。否则设置折扣率为 0.

触发器对于 `free_shipping` 和 `discount` 列值的作用会自动进行。考虑一下 Example 2-10 所展示的 `INSERT` 语句

Example 2-10 对 `sales` 表的数据插入

```
INSERT INTO sales
    (customer_id, product_id, sale_date, quantity, sale_value,
     department_id, sales_rep_id)
VALUES (20,10,now( ),20,10034,4,12)
```

销售价为\$10034,满足了 15%零售和折扣率的条件。Example 2-11 示范了触发器正确的设置了这些值

Example 2-11 触发器自动设置了 `free_shipping` 和 `discount` 列的值

```
mysql> SELECT sale_value,free_shipping,discount
-> FROM sales
-> WHERE sales_id=2500003;
+-----+-----+-----+
| sale_value | free_shipping | discount |
+-----+-----+-----+
|      10034 | Y             |      1505 |
+-----+-----+-----+
1 row in set (0.22 sec)
```

使用触发器来维护 `free_shipping` 和 `discount` 列确保了列值的正确设置，无论这个 SQL 语句是用 PHP，C#或者 Java 还是 MySQL 命令行客户端执行的。

2.13 在 PHP 中调用存储过程

我们已经向大家展示了如何用 MySQL 命令行客户端，MySQL Query Browser 和其他存储程序来调用存储程序。在实际的开发过程中，你更有可能要从别的语言环境中来调用我们的存储过程，例如 PHP，Java，Perl 或者 .NET。我们将用专门的章节来讨论如何在这些环境中调用存储程序（第 12 章到第 17 章）。

现在，让我们看看如何用 PHP，这种与 MySQL 关系最亲密的开发环境来调用存储过程。

当使用 PHP 和 MySQL 交互时，我们可以使用与数据库无关的 PEAR::DB 扩展工具，mysqli（MySQL "improved"）或者是最近的 PDO（PHP 数据对象）工具。在这个例子中，我们是用了 mysqli。第 13 章将详细讲述这些扩展工具。

Figure 2-19 展示了在 PHP 中连接 MySQL 服务器和调用存储过程的代码，我们不会对代码进行深入，只是希望它能给你如何在 WEB 应用程序和别的应用中使用存储程序留下映像。

Figure 2-18. 在 PHP 中调用存储过程



PHP 程序提示用户指定部门 ID；然后调用存储过程 employee_list 来获取该部门的员工名单 Figure 2-20 展示了 PHP + 存储过程例子的输出状况

Figure 2-19 PHP 程序调用存储过程的示例

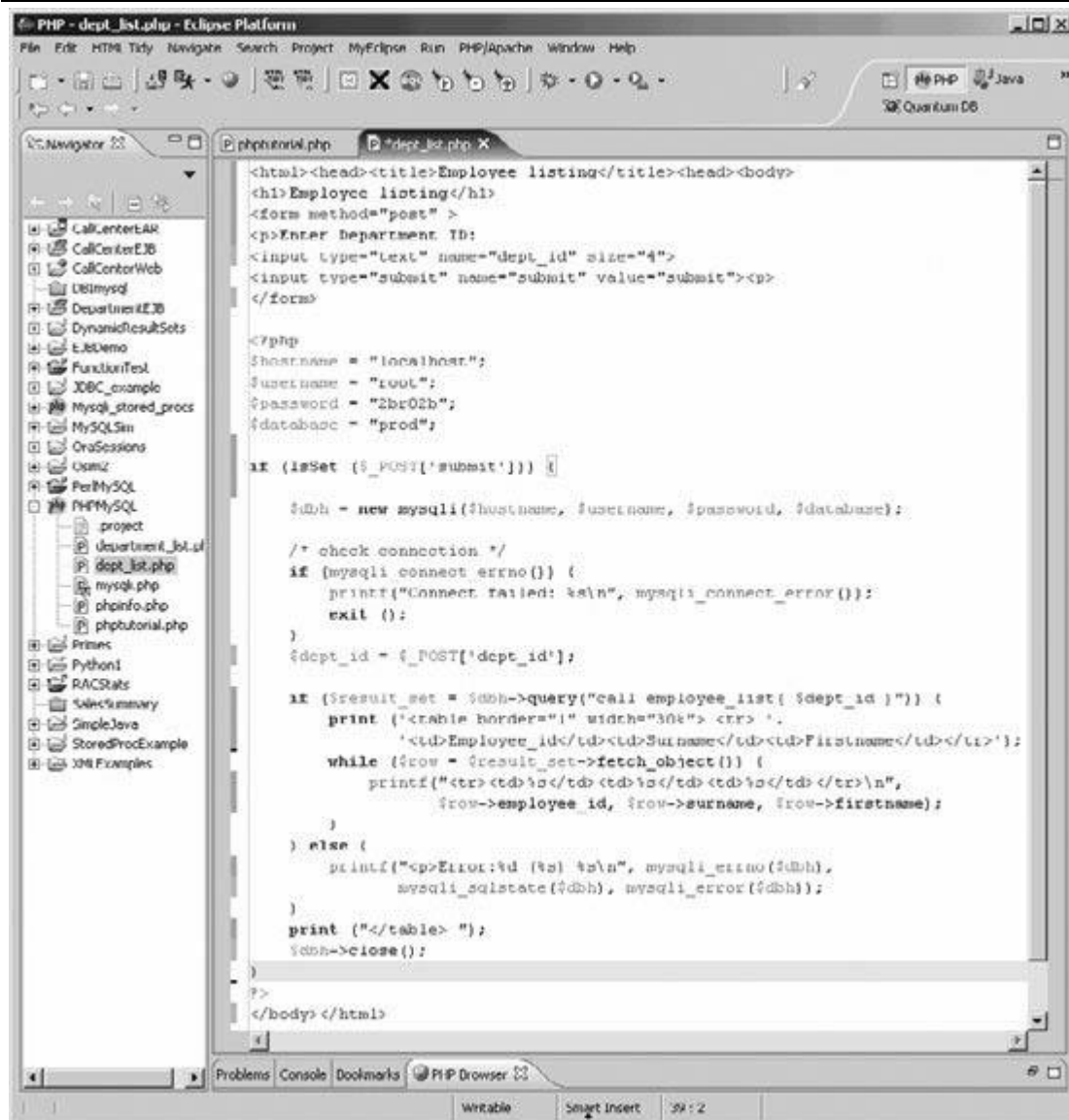
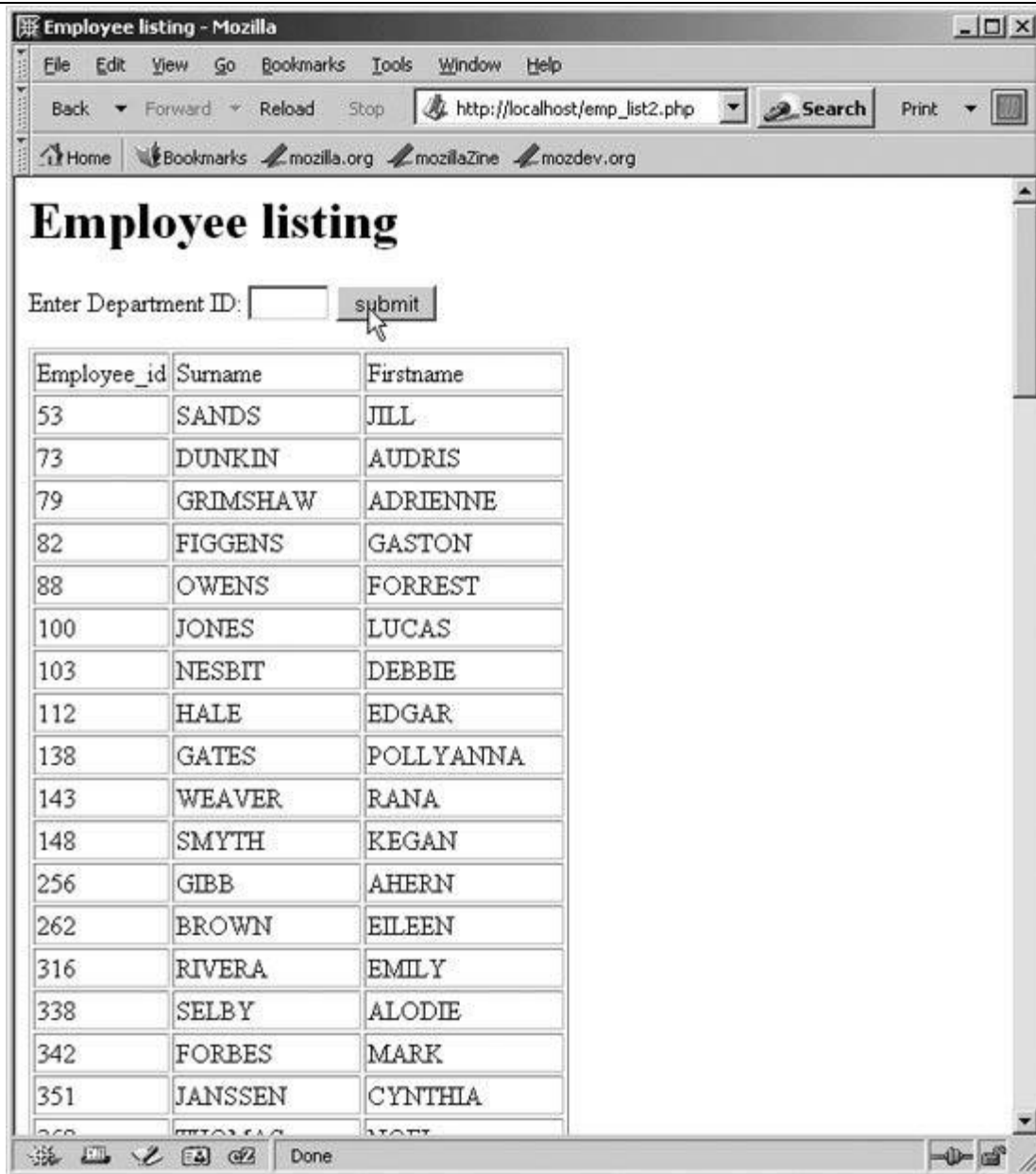


Figure 2-20 PHP 示例的输出状况



2.14 第二章结语

在这一章中，我们呈现了一个明朗的“基础指南”并介绍了一些基本的 MySQL 存储程序。我们想你展示了一下特点

- 创建“Hello World”存储过程
- 定义本地变量和过程参数
- 用 IF 语句进行条件执行
- 用简单循环进行迭代处理
- 包含 SQL 语句的存储过程，包括如何使用游标对各个数据行进行处理
- 从其他存储程序中调用存储程序
- 创建存储函数（以及存储函数和存储过程的区别）
- 为表的反响格式化创建触发器
- 从 PHP 中调用存储过程

现在你也许已经可以放下书本开始写一些 MySQL 存储程序了，我们对于你急切的心情表示赞同。但是我们可不可以提一个小小的建议：你可不可以先花更多的时间来阅读接下来各个章节中有关功能的具体内容，这样，你可以写出更少错误，更高质量的代码

第三章 语言基础

这一章着重介绍 MySQL 存储过程语言，一种基于 **NSI SQL:2003 SQL/PSM**（存储固定组建）标准的简单，易读的编程语言。

MySQL 存储程序语言是一种块结构语言（就像 **Pascal**）包含了用来控制变量，实现条件执行，进行迭代处理和错误处理的为人熟知的语句。其他存储程序语言的用户如 **Oracle PL/SQL** 或 **Microsoft SQL Server Transact-SQL**，将会发觉他们的功能很相似，而事实上使用 **IBM DB2 SQL** 过程语言的用户将会发觉它和 MySQL 存储程序语言几乎同样是基于 **SQL/PSM** 规范的。作为其他程序语言的用户，通常在 **PHP**，**Java** 或者 **Perl** 中使用 MySQL 会感到这种存储程序语言并非那么简洁，但是基本上和所学的所有语言相比并没有什么困难。

在这一章中，我们将会关注程序构建的基本要素：存储程序语言的块，字面常量，参数，注释，操作符，表达式和数据类型。我们也将讨论 MySQL 5 “**strict**” 模式以及它的内在意义。在下一章，我们将以此为基础，描述块结构，条件语句（**IF** 和 **CASE**）和语言的循环能力。

3.1 变量，字面量，参数和注释

让我们对变量类型定义，字面量和参数的回顾开始我们存储程序的学习，在这一章中，我们还将学到如何在代码中加入注释和文档。

3.1.1 变量

首先，让我们看一下 MySQL 存储过程是如何处理变量和字面量的，因为不具备对这些概念的了解，我们就不能创建一些更有意义的主题。

变量是一个值可以在程序执行过程中被改变的命名数据项。字面量（将在下一章进行探讨）是一个可以被赋值给变量的未命名数据项。通常，字面量是你程序中的硬代码，并且通常拿来赋值给变量，传递给参数，或者作为 SELECT 语句的参数。

DECLARE 语句允许我们创建变量。这一点我们接下来将看到，它将出现在代码块中任何游标和处理及任何过程语句声明之前，DECLARE 语句的语法如下：

```
DECLARE variable_name [,variable_name...] datatype [DEFAULT value];
```

多个变量可以在一个 DECLARE 语句中被声明，而且变量可以给出一个默认值（初始值）。如果你不给出 DEFAULT 子句，那么这个变量将会被赋予空值。

使用 DEFAULT 是一项很好的实践，因为除非你给出一个初始值，否则任何依赖于这个变量的后续操作都将在赋值之前返回 NULL 值。我们将在本章稍后部分给出这种错误类型的示例代码

datatype 可以是任何你可以在 CREATE TABLE 语句中使用的有效 MySQL 数据类型。我们将在本章稍后部分给出详细的描述；Table 3-1 给出了最常用的类型

Table 3-1. 常用的 MySQL 数据类型

数据类型	解释	相应值的示例
INT, INTEGER	32 位整数。取值范围为-21 亿到+21 亿，如果是非符号数，值可以达到 42 亿，但这样做就不能包括负数	123, 345 -2,000,000,000
BIGINT	64 位整数。取值范围为-9 万亿到+9 万亿或者非负的 0 到 18 万亿	9,000,000,000,000,000,000 -9,000,000,000,000,000,000
FLOAT	32 位浮点数。取值范围为 1.7e38 to 1.7e38 或者非负的 0 到 3.4e38	0.000000000000002 17897.890790 -345.8908770 1.7e21
DOUBLE	64 位浮点数。取值范围接近无限（1.7e308）	1.765e203 -1.765e100
DECIMAL (precision,scale) NUMERIC (precision,scale)	定点数。存储情况取决于 precision，能保存可能出现的数字范围。 NUMERIC 通常用来保存重要的十进制数，例如现金	78979.00 -87.50 9.95

DATE	日期类型，没有详述时间	'1999-12-31'
DATETIME	日期和时间，时间精确到秒	'1999-12-31 23:59:59'
CHAR(length)	定长字符串。值会被空白填充至制定长度，最大长度为 255 字节	'hello world '
VARCHAR(length)	最大长度为 64K 的可变字符串	'Hello world'
BLOB, TEXT	最大 64K 长度，BLOB 用来保存 2 进制数据，TEXT 用来保存文本数据	任何能想象的内容
LOBLOB, LONGTEXT	BLOB 和 TEXT 的加长版本，存储能力达 4GB	任何能想象的内容，但比 BLOB 和 TEXT 能存放更大的长度

Example 3-1 向我们演示了各种数据类型的声明

Example 3-1 数据类型声明演示

```

DECLARE l_int1      int default -2000000;
DECLARE l_int2      INT unsigned default 4000000;
DECLARE l_bigint1   BIGINT DEFAULT 4000000000000000;
DECLARE l_float     FLOAT DEFAULT 1.8e8;
DECLARE l_double    DOUBLE DEFAULT 2e45;
DECLARE l_numeric   NUMERIC(8,2) DEFAULT 9.95;

DECLARE l_date      DATE DEFAULT '1999-12-31';
DECLARE l_datetime  DATETIME DEFAULT '1999-12-31 23:59:59';

DECLARE l_char      CHAR(255) DEFAULT 'This will be padded to 255 chars';
DECLARE l_varchar   VARCHAR(255) DEFAULT 'This will not be padded';

DECLARE l_text      TEXT DEFAULT 'This is a really long string. In stored programs
                                we can use text columns fairly freely, but in tables there are some
                                limitations regarding indexing and use in various expressions.';

```

3.1.2 字面常量

字面常量是你程序中的硬代码。通常你可以将字面常量用于赋值语句和条件比对（比如 IF），存储过程，函数的参数或者 SQL 语句中。

下面是三大基本字面量类型

数字字面常量

数字字面量适用于表达数字并可以被定义为十进制数字（300, 30.45 等），十六进制数或者科学计数法表示的数。科学计数法用于表达非常大或者精度要求极高的值。字母 ‘e’ 在这里表示该数值为将 ‘e’ 的左侧的数乘以 10 的 ‘e’ 右侧数值次幂，因此 2.4e 就等于 2.4 x 104 或者 24,000。你不能在数字字面常量中使用逗号。

十六进制值使用的是传统的表示方法，在它的前面加上 ‘0x’。因此 0xA 表示十六进制中的 ‘A’，也就是十进制中的 10。

日期字面常量

一个日期字面常量是一个类似于 ‘YYYY-MM-DD’ 这样的字符串，或者在 `DATETIME` 中以 ‘YYYY-MM-DD HH24:MI:SS’ 这样的格式表示。所以 ‘1999-12-31 23:59:59’ 表示上个世纪的最后几秒钟（除非你不相信有公元 0 年这种说法，上个世纪实际上终止与 2000-12-32）。

字符字面常量

字符常量是任何被简单的包含在单引号中的值。如果你要在单引号中表示另一对单引号，那么你可以用两个双引号或者加上前缀的反斜杠(\)进行转意来表示他们。你也可以用双引号来闭合一个字符串，并且你可以使用转意序列字符（\t 表示 tab，\n 表示换行，\\表示反斜杠，等等）。

3.1.3.变量命名规则

MySQL 在变量命名方面具有惊人的灵活性。并不像其他大多数编程语言，MySQL 允许非常长的变量名（大于 255 个字符）；他们可以包含特殊的字符并且以数字字符开始。尽管如此，我们还是建议您不要把 MySQL 这种灵活性的优势取代明智的命名习惯，并且避免使用过长的变量名（见第 23 章获得更多相关信息和最佳实践）

3.1.4 变量赋值

你可以使用 SET 语句操纵变量赋值，请使用如下语法：

```
SET variable_name = expression [,variable_name = expression ...]
```

就像你所看到的，你完全可以通过使用一个 SET 语句来完成多次赋值。

大多数语言并不在对变量赋值时使用 SET 语句，于是结果就很容易造成了在不使用具体的 SET 来对变量赋值时造成的错误，就像 Example 3-2。

Example 3-2. 尝试不使用 SET 语句对变量赋值

```
mysql> Create procedure no_set_stmt( )
BEGIN
    DECLARE i INTEGER;
    i=1;
END;
$$

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds
to your MySQL server version for the right syntax to use near
'procedure no_set_stmt( )
BEGIN
    DECLARE i INT;
```

```
i=1;
END' at line 1
```

就像经常会犯的存储程序编译错误，错误信息并不会直接指明当前缺失的 **SET** 语句，所以当检查你程序的编译错误时，应该加倍检查所有的变量赋值，确保他们包含 **SET**。

3.1.5 参数

参数是可以被主叫程序传入或传出与存储程序的变量。参数被函数或过程创建时定义于 **CREATE** 语句内，就像下面所展示的一样：

```
Create procedure|function(
    [[IN
|OUT
|INOUT
] parameter_name data_type...])
```

参数的命名和变量的命名具有相同的规则，其中的 **data_type** 可以是任何本地变量类型。参数可以附加上 **IN**，**OUT** 或者 **INOUT** 属性

IN

除非被具体定义，否则参数都假定 **IN** 属性。这意味着他们的值必须被主叫程序所指定，并且任何在存储程序内部对该参数的修改都不能在主叫程序中起作用

OUT

一个 **OUT** 参数可以被存储程序所修改，并且这个被修改的值可以在主叫程序中生效，主叫程序必须提供一个变量来接受由 **OUT** 参数输出的内容，但是存储程序本身并没有对这个可能已经初始化的变量的操作权限，当存储程序开始时，任何 **OUT** 变量的值都被赋值为 **NULL**，不管这个值在主叫程序中是否被赋予其他值。

INOUT

INOUT 参数同时扮演着 **IN** 和 **OUT** 参数的角色。那意味着，主叫程序可以提供一个值，而被叫程序自身可以修改这个参数的值，并且当存储程序结束时主叫程序对该修改后的值具有访问权限

IN，**OUT** 和 **INOUT** 关键字只能被应用于存储过程而不适用于存储函数，在存储函数中所有的参数都被视为 **IN** 参数（虽然你不能指定 **IN** 关键字）

下面的三个例子举例说明了上面这些原理。

首先，虽然 MySQL 允许我们修改 **In** 参数，但这种修改在主叫程序中并不可见。**Example 3-3** 的存储程序打印并修改了参数的值。当存储程序内部对于输入参数的修改被允许时，原本的变量(**@p_in**)并没有改变。

Example 3-3. IN 参数的例子

```
mysql> CREATE PROCEDURE sp_demo_in_parameter(IN p_in INT)
BEGIN
    /* We can see the value of the IN parameter */
    SELECT p_in;
    /* We can modify it*/
    SET p_in=2;
    /* show that the modification took effect */
    select p_in;
END;

/* This output shows that the changes made within the stored program cannot be accessed
from
    the calling program (in this case, the mysql client):*/

mysql> set @p_in=1

Query OK, 0 rows affected (0.00 sec)

mysql> call sp_demo_in_parameter(@p_in)

+-----+-----+
| p_in | We can see the value of the IN parameter |
+-----+-----+
| 1 | We can see the value of the IN parameter |
+-----+-----+
1 row in set (0.00 sec)

+-----+-----+
| p_in | IN parameter value has been changed |
+-----+-----+
| 2 | IN parameter value has been changed |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> select @p_in, 'We can't see the changed value from the calling program'

+-----+-----+
| @p_in | We can't see the changed value from the calling program |
+-----+-----+
| 1 | We can't see the changed value from the calling program |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

接下来，在 Example 3-4 中，我们将验证 OUT 参数的行为。虽然主叫程序已经初始化了 OUT 参数的值，但是被叫程序无法看到这个值。无论如何，主叫程序只有在被叫过程执行完成后才能看到参数的改变

Example 3-4. OUT 参数的例子

```
mysql> CREATE PROCEDURE sp_demo_out_parameter(OUT p_out INT)

BEGIN
    /* We can't see the value of the OUT parameter */
    SELECT p_out, 'We can't see the value of the OUT parameter';
    /* We can modify it*/
    SET p_out=2;
    SELECT p_out, 'OUT parameter value has been changed';

END;
```

```
mysql> SET @p_out=1

Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL sp_demo_out_parameter(@p_out)
```

p_out	We can't see the value of the OUT parameter in the stored program
NULL	We can't see the value of the OUT parameter in the stored program

```
1 row in set (0.00 sec)
```

p_out	OUT parameter value has been changed
2	OUT parameter value has been changed

```
1 row in set (0.00 sec)
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT @p_out, "Calling program can see the value of the changed OUT parameter"
```

@p_out	Calling program can see the value of the changed OUT parameter
2	

```
1 row in set (0.00 sec)
```


最后，Example 3-5 将向我们演示 INOUT 参数的值，可以为我们的被叫程序所见，所修改并返回给它的主叫程序

Example 3-5.INOUT 参数的例子

```
mysql> CREATE PROCEDURE sp_demo_inout_parameter(INOUT p_inout INT)

BEGIN

    SELECT p_inout,'We can see the value of the INOUT parameter in the stored program';

    SET p_inout=2;
    SELECT p_inout,'INOUT parameter value has been changed';

END;
//
Query OK, 0 rows affected (0.00 sec)

set @p_inout=1
//
Query OK, 0 rows affected (0.00 sec)

call sp_demo_inout_parameter(@p_inout) //

+-----+-----+
| p_inout | We can see the value of the INOUT parameter in the stored program |
+-----+-----+
|      1 | We can see the value of the INOUT parameter in the stored program |
+-----+-----+
1 row in set (0.00 sec)

+-----+-----+
| p_inout | INOUT parameter value has been changed |
+-----+-----+
|      2 | INOUT parameter value has been changed |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

select @p_inout ,"Calling program can see the value of the changed INOUT parameter"
//

+-----+-----+
| @p_inout | Calling program can see the value of the changed INOUT parameter |
+-----+-----+
| 2        | Calling program can see the value of the changed INOUT parameter |
+-----+-----+
```

```
1 row in set (0.00 sec)
```

3.1.6 用户变量

用户变量是在 MySQL 中被定义并且可以在存储程序中或存储程序之外被操作的变量。他们在 MySQL3 中就可以被使用，并不像存储程序，我们可以通过以下两种方法来使用用户变量：

因为用户变量具有独立于存储程序个体的作用，他们可以用来描述那些能够被任何存储程序所读写的会话。这有些接近于在其他编程语言中全局变量的原理

用户变量可以给方法传递参数以第二种选择，存储程序对用户变量具有读写权限，这样可以避免使用参数传值的必要（见早些章节的“参数”或许更多有关参数的信息）

用户变量可以被 MySQL 命令行客户端从任何其他程序创建并操纵。来确保 MySQL 语句使用 SET 语句。Example 3-6 展示了一些在 MySQL 客户端使用 SET 的例子

Example 3-6. 在 MySQL 客户端操作用户变量

```
mysql> SELECT 'Hello World' into @x;
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @x;
+-----+
| @x      |
+-----+
| Hello World |
+-----+
1 row in set (0.03 sec)

mysql> SET @y='Goodbye Cruel World';
Query OK, 0 rows affected (0.00 sec)

mysql> select @y;
+-----+
| @y      |
+-----+
| Goodbye Cruel World |
+-----+
1 row in set (0.00 sec)

mysql> SET @z=1+2+3;
Query OK, 0 rows affected (0.00 sec)

mysql> select @z;
+-----+
```

```
| @z |
+-----+
| 6 |
+-----+
1 row in set (0.00 sec)
```

你可以在当前会话（比如 `connection`）里从存储程序中使用任何用户变量。举例来说，Example 3-7 展示了如何不使用过程参数向存储过程传递信息

Example 3-7. 使用用户变量在主叫程序和被叫程序之间传递信息

```
mysql> CREATE PROCEDURE GreetWorld( )
    -> SELECT CONCAT(@greeting, ' World');
Query OK, 0 rows affected (0.00 sec)

mysql> SET @greeting='Hello';
Query OK, 0 rows affected (0.00 sec)

mysql> CALL GreetWorld( );
+-----+
| CONCAT(@greeting, ' World') |
+-----+
| Hello World                  |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

我们也可以用一个存储程序创建用户变量。这会使该变量在其他存储程序中都可使用，正如同 PHP 中全局变量(global)所扮演的角色。举例来说，在 Example 3-8 中过程 `p1()` 创建了一个用户变量，这对过程 `p2()` 也可见。

Example 3-8. 把用户变量当成全局变量交叉使用

```
mysql> CREATE PROCEDURE p1( )
    -> SET @last_procedure='p1';
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE PROCEDURE p2( )
    -> SELECT CONCAT('Last procedure was ', @last_procedure);
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p1( );
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p2( );
```

```

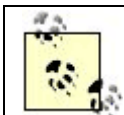
+-----+
| CONCAT('Last procedure was ',@last_procedure) |
+-----+
| Last procedure was p1 |
+-----+
1 row in set (0.00 sec)

```

用户变量是一种可变数据类型，能够保存字符串，日期或者数字值。日期类型的转换是自动的。用户变量存在于一个持续的 MySQL 会话中，在此会话中的任何程序和语句都可以访问该用户变量，当然，别的会话则无法访问它。

在某些编程语言中（如 PHP），独立于单个函数的变量的作用域必须用 **global** 关键字标示。在其他语言中对这些变量的定义语法可能是不同的，但是它们总是与“全局”变量相关联。在 MySQL 中，SET 语句的 **global** 子句允许你设定服务器域的系统变量，这个概念和 PHP 的全局变量不等价。因此，如果说用户变量是“全局”作用域的那会引起冲突并且应该予以避免。注意：你不能在 SET 语句的 **global** 子句中创建你自己的变量。

使用用户变量去实现跨越多个存储程序的变量在某些场合会非常有用。但是，你必须明确这样做的意图，并谨慎的使用之。如同所有编程语言一样，过度的使用作用域超越单个程序的全局变量会让你的代码不易读并且难于维护。使用这些变量的例程会变得高耦合并难以维护，测试和理解



在你的存储程序中适时使用“用户”变量，过度使用超出程序作用域的变量会导致你的程序非模块化并难于维护

3.1.7. 注释

MySQL 存储程序支持两种不同风格的注释：

两个连字符跟上一个空格创建了一个到当前行末的注释

C 语言风格的注释，用 **/*** 开始，以 ***/** 结束。我们称呼它为多行注释

单行注释对于变量声明和简单的语句很有效，多行注释对于创建大的注释块很有效，比如位于存储程序头部的注释块一样。

下面这块代码 **Example 3-9** 展示了这两种不同的注释风格

Example 3-9. 存储程序注释的例子

```

create procedure comment_demo
    (in p_input_parameter INT -- Dummy parameter to illustrate styles
    )
/*
|   Program: comment_demo
|   Purpose: demonstrate comment styles
|   Author:  Guy Harrison
*/

```

```
|   Change History:
|       2005-09-21 - Initial
|
*/
```

3.2. 操作符

MySQL 包括大家在大多数语言中早已熟识的操作符，但是 C 风格的操作符（`++`，`+=`，etc）并不被支持。

操作符经常是和 `SET` 语句一起来改变变量的值，和比较语句如 `IF` 或者 `CASE`，和循环控制表达式。Example 3-10 展示了一些在存储程序中使用操作符的简单例子。

Example 3-10 存储程序中操作符的例子

```
create procedure operators( )
begin
    DECLARE a int default 2;
    declare b int default 3;
    declare c FLOAT;

    set c=a+b; select 'a+b=',c;
    SET c=a/b; select 'a/b=',c;
    SET c=a*b; Select 'a*b=',c;

    IF (a<b) THEN
        select 'a is less than b';
    END IF;
    IF NOT (a=b) THEN
        SELECT 'a is not equal to b';
    END IF;
end;
```

有关不同种类的操作符（数学，比较，逻辑和位操作）将在下面的字章节中做出阐述

3.2.1.数学操作符

MySQL 支持你在小学（付费补习班）中学到的基本数学操作符：加（`+`），减（`-`），乘（`*`）和除（`/`）。

此外，MySQL 支持两种与除法操作相关的操作符：整除（`DIV`）操作符返回除法操作的整数部分，而模操作符（`%`）返回整除后的余数部分。Table 3-2 给出了一个有关 MySQL 数学操作符的完整列表并给出了描述和用例。

Table 3-2.MySQL 数学操作符

操作符	描述	用例
<code>+</code>	加	<code>SET var1=2+2; → 4</code>
<code>-</code>	减	<code>SET var2=3-2; → 1</code>
<code>*</code>	乘	<code>SET var3=3*2; → 6</code>

/	除	SET var4=10/3; → 3.3333
DIV	整除	SET var5=10 DIV →3; 3
%	模	SET var6=10%3; → 1

3.2.2.比较操作符

比较操作符比较两个值并返回 TRUE，FALSE，UNKNOWN（通常如果一个值被比较后返回 NULL 或者 UNKNOWN）。他们通常被使用在 IF，CASE，和循环控制表达式中。

Table 3-3 总结了 MySQL 比较操作符

Table 3-3. 比较操作符

操作符	描述	示例	示例结果
>	是否大于	1>2	False
<	是否小于	2<1	False
<=	是否小于等于	2<=2	True
>=	是否大于等于	3>=2	True
BETWEEN	是否位于两个值之间	5 BETWEEN 1 AND 10	True
NOT BETWEEN	是否不位于两个值之间	5 NOT BETWEEN 1 AND 10	False
IN	值位于列表中	5 IN (1,2,3,4)	False
NOT IN	值不位于列表中	5 NOT IN (1,2,3,4)	True
=	等于	2=3	False
<>, !=	不等于	2<>3	False
<=>	Null 安全等于（如果两个值均为 Null 返回 TRUE）	NULL<=>NULL	True
LIKE	匹配简单模式	"Guy Harrison" LIKE "Guy%"	True
REGEXP	匹配扩展正则表达式	"Guy Harrison" REGEXP "[Gg]reg"	False
IS NULL	值为空	0 IS NULL	False
IS NOT NULL	值不为空	0 IS NOT NULL	True

3.2.3 逻辑操作符

逻辑操作符操作三个逻辑值 TRUE，FALSE 和 NULL 并返回类似的这三个值。那些操作符通常被用来和比较操作符相结合创建除更为复杂的表达式

对于大多数逻辑操作符而言，如果其中任何值被比较得出为 NULL，那么最终的结果就为 NULL。在创建逻辑表达式时记住这样一个事实很重要，否则，你的代码就可能隐含一些微小的错误

AND 操作符比较两个逻辑表达式，并且只在两个表达式都为真是才返回 TRUE。Table 3-4 展示了 AND 操作符所生成的可能的值

Table 3-4. AND 操作符的一些事实

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	NULL
NULL	NULL	NULL	NULL

OR 操作符比对两个逻辑表达式，并且只要其中的一个表达式为真即返回 TRUE（Table 3-5）

Table 3-5. OR 操作符的一些事实

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

XOR 操作符只有在两个值不完全为真时才返回 TRUE。Table 3-6 展示了 XOR 表达式的可能值

Table 3-6. XOR 操作符的一些事实

XOR	TRUE	FALSE	NULL
TRUE	FALSE	TRUE	NULL
FALSE	TRUE	FALSE	NULL
NULL	NULL	NULL	NULL

Example 3-11 展示了如何使用 AND 操作符连接多个比较语句

Example 3-11.逻辑语句实践的例子

```
CREATE FUNCTION f_title(in_gender CHAR(1),
                        in_age INT, in_marital_status VARCHAR(7))
  RETURNS VARCHAR(6)
BEGIN
  DECLARE title VARCHAR(6);
  IF in_gender='F' AND in_age<16 THEN
    SET title='Miss';
  ELSEIF in_gender='F' AND in_age>=16 AND in_marital_status='Married' THEN
    SET title='Mrs';
  ELSEIF in_gender='F' AND in_age>=16 AND in_marital_status='Single' THEN
    SET title='Ms';
  ELSEIF in_gender='M' AND in_age<16 THEN
    SET title='Master';
  ELSEIF in_gender='M' AND in_age>=16 THEN
    SET title='Mr';
  END IF;
  RETURN(title);
END;
```


3.2.4 位操作符

位操作符主要是对二进制值进行操作。Table 3-7 列出了位操作符

Table 3-7. 位操作符

操作符	用途
	OR
&	AND
<<	Shift bits to left
>>	Shift bits to right
~	NOT or invert bits

位操作符类似于逻辑操作符，除了他们的操作对象是二进制的位并得出一个结果

举例来说，考虑整数（二进制的 101）和 4（二进制的 010）。或操作符操作的是二进制中的每一位；所以 $5|2=7$ ，因为 $101|010=111$ ，这就是十进制中的 7

位操作符 AND 操作二进制位中的每一位，只有在二者都为 true 时才设置位数。所以 $5\&6=4$ ，因为 $101\&110=100$ ，那就等于 4

3.3. 表达式

表达式是字面量，变量和操作符的集合，用以计算出某些需要的值。条件执行和流程控制表达式经常会依赖于这些值来决定循环条件和代码分支。

Example 3-12 演示了几种表达式

Example 3-12.表达式用例

```
Myvariable_name  
Myvariable_name+1  
ABS(Myvariable_name)  
3.14159  
IF(Myvariable='M','Male','Female')  
(2+4)/12
```

3.4 内建函数

你可以在 MySQL 语句中使用大多数 MySQL 产品允许的函数，你能在 MySQL 手册中找到他们的详尽文档，并且我们在第 9 章提供了大多数函数的明细和样例，我们也将第十章介绍如何在你的 MySQL 存储程序中使用自己创建的“存储”函数。

在 SQL 中可以被使用的函数未必都能在存储程序中被编组操作符(多行)调用。那些函数包括 SUM, COUNT, MIN, MAX 和 AVG。MySQL 允许表达式中包括函数，但是就像 Example 3-13 所演示的那样，他们会返回 NULL 值

Example 3-13 在存储过程中的函数调用返回 NULL

```
mysql> create procedure functions( )
begin
    DECLARE a int default 2;
    declare b int default 3;
    declare c FLOAT;

    SET c=SUM(a); select c;

end;

Query OK, 0 rows affected (0.00 sec)

mysql> call functions( );

+-----+
| c      |
+-----+
| NULL   |
+-----+
1 row in set (0.00 sec)
```

MySQL 函数被归类为以下几个类型

字符串函数

这些函数主要对字符串变量执行操作，比方说：你可以连接字符串，在字符串中查找字符，得到子串和其他常规操作

数学函数

这些函数主要对数字进行操作，比方说：你可以进行乘方（平方），三角函数（sin，cos），随机数函数和对数函数等。

日期和时间函数

折现函数主要的操作对象是日期和时间，比方说：你可以得到当前时间，从一个日期上加上或减去一个时间间

隔，找出两个日期间的差异，获取某个确定的时间点（比如：得到一天中某时间的小时数）

其它函数

这些函数包括了所有不容易被分入上面类别中函数。他们包括类型转换函数，流程控制函数（比如：CASE），信息反馈函数（比如服务器版本）和加密函数

Table 3-8 总结了大多数常用的函数；相见第九章来获取函数语法和样例的完整覆盖

Table 3-8. 经常被使用的 MySQL 函数

函数	描述
<code>ABS(number)</code>	返回提供数字的绝对值。比方说， <code>ABS(-2.3)=2.3</code> 。
<code>CEILING(number)</code>	返回下一个最大整数，比方说， <code>CEILING(2.3)=3</code> 。
<code>CONCAT(string1[,string2,string3,...])</code>	返回所有提供字符串的连接形式的值
<code>CURDATE</code>	返回当前时间（不带时间）
<code>DATE_ADD(date,INTERVAL amount_type)</code>	给提供的时间值加上一个时间间隔并返回一个新时间。正确的形式有 <code>SECOND</code> , <code>MINUTE</code> , <code>HOURL</code> , <code>DAY</code> , <code>MONTH</code> 和 <code>YEAR</code>
<code>DATE_SUB(date,INTERVAL interval_type)</code>	从提供的时间值上减去一个时间间隔并返回一个新的时间。正确的形式有 <code>SECOND</code> , <code>MINUTE</code> , <code>HOURL</code> , <code>DAY</code> , <code>MONTH</code> 和 <code>YEAR</code>
<code>FORMAT(number,decimals)</code>	返回一个指定精确度的数值，并给与以 1000 为单位的分割（通常使用 “,”）
<code>GREATEST(num1,num2[,num3, ...])</code>	返回所有提供参数中的最大数
<code>IF(test, value1,value2)</code>	测试一个逻辑条件，如果为真则返回 <code>value1</code> ，如果为假返回 <code>value2</code>
<code>IFNULL(value,value2)</code>	返回第一个值，除非第一个值为空；这样的话返回第二个值
<code>INSERT(string,position,length,new)</code>	把一个字符串插入到另一个字符串中
<code>INSTR(string,substring)</code>	返回一个字符串中子串的位置
<code>ISNULL(expression)</code>	如果参数为空则返回 1，否则返回 0
<code>LEAST(num1,num2[,num3, ...])</code>	返回参数列表中的最小值
<code>LEFT(string,length)</code>	返回字符串最左边的部分
<code>LENGTH(string)</code>	返回字符串中的字节数。 <code>CHAR_LENGTH</code> 可以被用来返回字符数（这会在你使用多字节字符集是产生差异）
<code>LOCATE(substring,string[,number])</code>	返回字符串中子串的位置，可选的第三个参数为在父字符串中开始搜索的位置
<code>LOWER(string)</code>	返回给定字符串的小写形式
<code>LPAD(string,length,padding)</code>	返回字符串 <code>str</code> ，其左边由字符串 <code>padding</code> 填补到 <code>length</code> 字符长度，第三个参数为填充字符
<code>LTRIM(string)</code>	删除所有字符串中的前缀空格
<code>MOD(num1,num2)</code>	返回第一个数除于第二个数后的模（余数部分）
<code>NOW</code>	返回当前日期和时间
<code>POWER(num1,num2)</code>	返回 <code>num1</code> 的 <code>num2</code> 次方
<code>RAND([seed])</code>	返回一个随机数。 <code>seed</code> 可被用于随机数生成器的种子数
<code>REPEAT(string,number)</code>	返回一个重复 <code>number</code> 次 <code>string</code> 的字符串
<code>REPLACE(string,old,new)</code>	用 <code>new</code> 替换所有出现 <code>old</code> 的地方
<code>ROUND(number[,decimal])</code>	舍去给定数值的指定精度的位数
<code>RPAD(string,length,padding)</code>	返回字符串 <code>str</code> ，其右边由字符串 <code>padding</code> 填补到 <code>length</code> 字符

	长度，第三个参数为填充字符
RTRIM(<i>string</i>)	删除字符串尾部的空格
SIGN(<i>number</i>)	如果 <i>number</i> 小于 0 则返回 -1，如果大于 0 则返回 1，如果为 0 则返回 0
SQRT(<i>number</i>)	返回 <i>number</i> 的平方根
STRCMP(<i>string1</i> , <i>string2</i>)	如果两个值相同则返回 0，若根据当前分类次序，第一个参数小于第二个，则返回 -1，其它情况返回 1。
SUBSTRING(<i>string</i> , <i>position</i> , <i>length</i>)	从字符串指定位置开始返回 <i>length</i> 个字符
UPPER(<i>string</i>)	将指定字符串转换为大写
VERSION	返回 MySQL 服务器当前版本号的字符串

函数可以被用在任何接受表达式的场合，在 SET 语句，条件语句中（IF，CASE），和循环控制子句。Example 3-14 展示了在 SET 和 IF 子句中使用函数的例子

Example 3-14. 在 SET 和 IF 子句中使用函数的例子

```
CREATE PROCEDURE function_example( )
BEGIN

    DECLARE TwentyYearsAgoToday DATE;
    DECLARE mystring VARCHAR(250);

    SET TwentyYearsAgoToday=date_sub(curdate( ), interval 20 year);

    SET mystring=concat('It was ',TwentyYearsAgoToday,
        ' Sgt Pepper taught the band to play...');

    SELECT mystring;

    IF (CAST(SUBSTR(version( ),1,3) AS DECIMAL(2,1)) <5.0) THEN
        SELECT 'MySQL versions earlier than 5.0 cannot run stored programs - you
            must be hallucinating';
    ELSE
        SELECT 'Thank goodness you are running 5.0 or higher!';
    END IF;

END$$

CALL function_example( )$$

+-----+
| mystring                                     |
+-----+
| It was 1985-11-22 Sgt Pepper taught the band to play... |
+-----+
1 row in set (0.03 sec)

+-----+
```

```
| Thank goodness you are running 5.0 or higher! |  
+-----+  
| Thank goodness you are running 5.0 or higher! |  
+-----+  
1 row in set (0.03 sec)
```

3.5.数据类型

MySQL 存储程序允许你将任何 MySQL 表列中的数据类型赋给变量。我们早在 Table 3-1 就提供了大多数数据类型。

在 MySQL 存储程序中的所有变量都是单纯的标量，也就是变量存储的只是单纯的个体，存储程序中没有与数组，记录或结构体等你可以在别的语言中找到的对应的事物。

3.5.1 字符串数据类型

MySQL 提供两种基本的字符串数据类型：CHAR 和 VARCHAR，CHAR 存储定长字符串，而 VARCHAR 存储可变长度的字符串。如果 CHAR 变量被赋予小于其声明长度的值，那么他将空白填充至声明长度。这样的事在 VARCHAR 变量中不会发生。

如果在 MySQL 表中，对于 CHAR 和 VARCHAR 的选择就非常重要，因为这直接关系到磁盘存储空间需求。虽然，在存储程序中，额外的内存需求将会最小化，并且 CHAR 和 VARCHAR 的是用在所有的表达式中都是可以相互替换的，但是两者的不同应用都有其各自小小的又是。我们通常使用 VARCHAR，因为他们能存放长一些的字符串。

CHAR 数据类型最大可以存放 255 个自己的数据，而 VARCHAR 最大可以存放 65,532 字节的数据

3.5.1.1 枚举数据类型

枚举数据类型用来存放一系列允许的值。这些值可以用他们的字符串值或者他们在这列数据中的索引值进行访问。如果你试图将一个没有出现在列表中的值插入到 ENUM 中，MySQL 将回应你一个警告并且在 sql_mode 中包含“strict”值时向这个枚举数据插入一个 NULL 值

Example 3-15 在程序中使用枚举 ENUM

```
CREATE PROCEDURE sp_enums(in_option ENUM('Yes','No','Maybe'))
BEGIN
    DECLARE position INTEGER;
    SET position=in_option;
    SELECT in_option,position;
END
```

Query OK, 0 rows affected (0.01 sec)

```
CALL sp_enums('Maybe')
```

+-----+-----+
in_option position
+-----+-----+

```

| Maybe      |      3 |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

-----

CALL sp_enums(2)
-----

+-----+-----+
| in_option | position |
+-----+-----+
| No        |      2   |
+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

-----

CALL sp_enums('What?')
-----

ERROR 1265 (01000): Data truncated for column 'in_option' at row 1

```

3.5.1.2.SET 数据类型

SET 很想枚举 SET 类型，但是在 SET 中可以插入多个列表中的值（见 Example 3-16）。和 ENUM 一样，尝试在 SET 中插入列表中不存在的值时，如果打开“strict”模式，则会受到一个错误提示或是警告。

Example 3-16. 存储程序中 SET 变量的行为

3.5.2. 数字数据类型

MySQL 支持两族系的数字类型：

精确数据类型比如 INT 和 DECIMAL 类型

近似数字类型比如 FLOAT

精确数据类型存放一个要求精度的数字。不同的 INT 数据类型（INT, BIGINT, TINYINT）在他们对于存储空间上的要求是不同的，这也限制了他们所能存放数据的大小。其中的任何一种类型都能带符号（可以被存放正数或负数）或无符号数据，这也限制了数据类型所能存放数据的最大值（无符号双精确度类型可能是能存放最大值的整型）。Table 3-9 展示了各种整形的存储范围

Table 3-9. 各种整型数的存储范围

数据类型	空间要求 (bit)	有符号最大值	无符号最大值
TINYINT	8	127	255
SMALLINT	16	32767	65535
MEDIUMINT	24	8388607	16777215
INT	32	2147483647	4294967295
BIGINT	64	9223372036854775807	9223372036854775807

浮点数数据类型 (FLOAT, DOUBLE, REAL) 存储变量的大小和精度。在 MySQL 表中, FLOAT 类型使用 32bit 的存储空间, 而 DOUBLE 使用 64bit 的存储空间

需要注意的是, 虽然, 浮点数数据类型存放数字的近似值。大多数情况下这并不重要, 但是在某些清苦按下你可能会用到近似数据类型, 例如 DECIMAL 或 NUMERIC, 为了避免对浮点数数学操作过程中产生舍入错误。

3.5.3. 日期和时间数据类型

MySQL 以秒为精度存放日期-时间数据, 在 MySQL 表中, 日期列只能存放日期-时间数据的日期部分, 而 DATETIME 可以用来存放日期和时间的所有部分

3.5.4. TEXT 和 BLOB 数据类型

在 MySQL 表中, TEXT 数据类型可以存放 64K 的数据, 而 LONGTEXT 可以存放 4,294,967,295 字符, BLOB 和 LONGBLOB 可以存放类似的数据范围, 但是它可以用来存放二进制数据。

3.6.MySQL 5 "Strict" 模式

当 `STRICT_TRANS_TABLES` 或者 `STRICT_ALL` 被包含在 `sql_mode` 设置变量的可选参数中时，MySQL 5 “strict” 模式将会被启用，`STRICT_ALL_TABLES` 将导致任何在列中尝试插入无效值的行为返回错误信息。`STRICT_TRANS_TABLES` 和它具有相同的作用，但是只有在开启事务功能的表中才会起作用

如果两个设定都没有起作用，MySQL 将会接受更新指令或将无效值“最佳匹配”为合法列值，举例来说，如果你尝试在一个整型列中插入一个字符串值，MySQL 会将该列值设置为 0。同时将会在这类“截断”发生时报告一个警告信息。

严格模式在你的 `INSERT` 语句中丢失列时同样会给出一个错误提示，除非该列被设置有 `DEFAULT` 子句。

当你在对没有事务支持的表执行多行更新或插入操作时，`STRICT_ALL_TABLES` 可以产生一些可疑的副作用。因为，对于没有事务支持的表而言不具备回滚功能，错误可能将在一定数量的有效行更新后发生。这意味着在这个没有事务支持的表中产生了严格模式的错误，这个 SQL 语句或许部分执行成功。这是一个被渴望已久的行为，所以因为这个原因，MySQL 5.0 中 `STRICT_TRANS_TABLES` 是缺省设置

你可以在任何时候用 `SET` 语句修改严格模式的设置：

```
SET sql_mode='STRICT_ALL_TABLES'
```

严格模式同样决定着存储程序如何对待尝试将不正确的值赋予变量。如果任何一个严格模式生效，那么当尝试给变量赋予不正确的值时，将会收到错误信息。如果严格模式未被开启，那么只会产生警告

注意这个行为受到 `sql_mode` 设定的控制并且将在程序创建时生效，而并不是运行时。所以当严格的存储程序被创建时，它仍然是严格的，除非 `sql_mode` 设定随后被放宽。同样的道理，如果程序在没有设定严格模式的前提下被创建那么它在无效赋值的时候将产生警告而不是错误。而在程序运行时它并不关注 `sql_mode` 是否生效。

3.6.1. 存储程序行为和严格模式

在 MySQL 存储程序中除了用户变量之外，所有的其他变量都必须在使用前被声明，你可以使用 `@` 符号将变量定义在存储程序之外。此外，MySQL 存储程序中的变量必须被赋予某个确定的数据类型，并且在程序执行过程中数据类型不得被更改。在这方面，MySQL 存储程序语言有点像诸如 C，Java 和 C# 等“强类型”语言，而却别与如 Perl 和 PHP 这样的动态语言

当在严格模式下创建程序是，就像在早些章节所解释的那样，存储程序将在尝试将无效数据赋值给变量是使用了不恰当的变量赋值是，都会弹出一个错误。这种被拒绝的赋值包含尝试将字符串赋予数字数据或者尝试赋值时使用了超出声明存储空间的变量。

虽然，当存储过程程序在非严格模式下被创建时，MySQL 将使用最佳匹配来适应不正确数据并且只是给出警告而不是错误。这允许你可以把个别的字符串赋给整型数，如果你不能谨慎的确认你在赋值时总是能你的变量选择合适的类型，那么这种非严格的行为可能导致不预期的结果和微妙的错误，因为通常最好在严格模式下常见存储程序，这样可以产生一个错无而不至于让你在程序的测试过程中被忽略掉

3.6.2. 程序样例

我们将举例说明并比较 MySQL 存储程序在非严格模式下与其他几种编程语言的差别

Example 3-17 展示了意图用 Java 程序打印出字符串"99 bottles of beer on the wall"时进行烦复的整型数和字符串值之间的转换的操作。不幸的是为了烦复的约束，程序员偶尔会将变量 c 声明为 int，而不是 String。Java 编译器将在检测到尝试把字符串表达式转换成整型数时在编译期报告一个错误，而程序也不得不在失败中结束。

Example 3-17. Java 程序的类型检测

```
$cat simplejava.java
package simplejava;

public class SimpleJava {

    public static void main(String[] args) {
        String b;
        int a;
        int c;
        a=99;
        b="Bottles of beer on the wall";
        c=a+ " "+c;

        System.out.println(c);
    }
}

$javac simplejava.java
simplejava.java:11: incompatible types
found   : java.lang.String
required: int
        c=a+ " "+c;
          ^
1 error
```

现在，让我们看一个与之类似的例子（在动态类型语言中，这个例子中是 PHP），在 PHP 和 Perl 中，变量的数据类型在运行时可以随着需求而改变。在 Example 3-18，变量 c 刚开始是一个数字值，但当它被一个字符串变量赋值时，这个数据类型动态的改变为字符串型，而程序本身任然能按照需求正常的工作。

Example 3-18. PHP 中的动态变量类型

```
$cat simplephp.php
<?php

    $a=99;
    $b="Bottles of beer on the wall";
    $c=0;          #c is a number
    $c=$a." ".$b;  #c is now a string
```

```

        print $c."\n";
?>

$php simplephp.php
99 Bottles of beer on the wall

```

现在，让我们看一下在没有开启严格模式的 MySQL 存储程序版本中的逻辑。就像 Example 3-19 所展示的那样。这个存储过程和前一个例子中程序具有相同的数据类型错误，变量 c 应该被定义为 VARCHAR，但是在这个例子中他被声明为 INT

Example 3-19. MySQL 存储程序非严格模式下的类型检测

```

CREATE PROCEDURE strict_test( )
BEGIN

    DECLARE a INT;
    DECLARE b VARCHAR(20);
    DECLARE c INT;

    SET a=99;
    SET b="Bottles of beer on the wall";
    SET c=CONCAT(a," ",b);
    SELECT c;
END
-----

Query OK, 0 rows affected (0.01 sec)

mysql> call strict_test( );
+-----+
| C      |
+-----+
|  99   |
+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message                                     |
+-----+-----+-----+
| Warning | 1265 | Data truncated for column 'b' at row 1 |
| Warning | 1265 | Data truncated for column 'c' at row 1 |
+-----+-----+-----+
2 rows in set (0.01 sec)

```

在没有开启严格模式的情况下，MySQL 并不会在尝试对一个整型变量赋予字符串值时报告一个错误，并且也不是动态的将数据类型转换为整型。取而代之，它只会不预期的，不正确的把字符串表达式的数字部分赋予整型变量。但是，如果你是在严格模式下创建的存储过程，我们就能收到一个运行时的错误，就像 Example 3-20 所展示的那样。

Example 3-20. 存储程序严格模式下的类型检测

```
mysql> CALL strict_test( );  
ERROR 1406 (22001): Data too long for column 'b' at row 1
```

在绝大多数时候你总是希望把程序在严格模式下运行，虽然一个非严格的程序可以在某些严格的程序产生错误是任然保持运行，但是非严格程序所产生的不可预期的错误和不适合的行为总是会带给你的程序过高的风险。始终记住：存储程序的行为依赖于程序创建时 `sql_mode` 变量的设置，而并非运行时。



存储程序应该始终运行在严格模式下来避免不正规赋值过程中的不可预期的行为。
存储程序严格模式决定与程序创建是 `sql_mode` 变量的设定，而与运行时无关

通常，程序员的责任就是保证数据类型的正确适用，如 Bruce Eckel 在他的文章中指出的那样“强类型 vs 强大的测试”(<http://www.mindview.net/WebLog/log-0025>)，强类型只不过是计算机语言提供的一种看似安全检测手段的幻觉，而正确的行为只有通过强大的测试才能获得，你不能假设在声明变量时给定的某个确定的类型就表明你已经隐含的完成了适当的变量类型校验。

3.7.第三章结语

在这一章中我们提供了用 MySQL 存储程序语言来创建程序片段的概览。MySQL 存储程序语言是一种基于 ANSI SQL:2003 PSM 标准的块结构语言，他支持所以你期望在过程语言中应该具备的要素。对于存储程序语言而言，你需要熟悉主要有以下几个方面：

- DECLARE 语句，允许你定义并初始化程序变量。
- 存储程序参数，允许你向存储过程传入或传出信息。
- SET 语句，允许你改变程序变量的值。
- MySQL 函数，操作符和数据类型，MySQL 存储程序语言有效的利用了 MySQL SQL 语言中的大多数等价物。

存储程序类型检查非常强烈的依赖于 `sql_mode` 设置变量的设定情况，如果一个程序在 `sql_mode` 变量包含任何一个严格设定（`STRICT_TRANS_TABLES` 或 `STRICT_ALL_TABLES`），那么程序在进行无效赋值是将产生一个错误。如果其中的任何一个都没有生效，那么存储程序将在无效赋值发生时产生一个错误（事实上是警告），但是它任然能继续执行。非严格存储程序模式的行为可能导致不预期的，微妙的错误，因此我们强烈建议你在通常情况下请在严格模式下创建你的存储程序。

第四章 程序块，条件表达式和迭代编程

这一章描述了在 MySQL 语言程序创建过程中作用域的控制和流程控制

在 MySQL 中，和所有的块结构语言一样，成组的语句可以编辑为块。块可以出现在任何正规的单个语句出现的地方，并且块可以包含他自己的变量，游标和处理结构声明。

MySQL 存储程序编程语言支持两种不同的流程控制语句：条件控制语句和迭代（循环）语句。你所写的几乎任何一个代码片段都需要条件控制，这是一种基于条件的指导你程序执行流程的能力。一般我们使用 **IF-THEN-ELSE** 和 **CASE** 语句来完成这一切。

迭代控制结构或者说是循环体，允许你将同一段代码重复执行。MySQL 提供了三种不同的循环控制：

简单循环

一直执行直到你使用 **LEAVE** 语句明确的声明终止循环为止

REPEAT UNTIL 循环

当表达式返回真是继续执行

WHILE 循环

一直循环直到表达式为真

4.1 存储程序的块结构

大多数 MySQL 存储程序由一个或多个程序块构成（唯一的例外就是只有单个语句构成的存储程序）。每个程序块都有 BEGIN 语句开始，由 END 语句结束，所以最简单的存储程序就是由程序定义声明（CREATE PROCEDURE, CREATE FUNCTION 或 CREATE TRIGGER）后面跟随一个包含要执行代码的块构成的

```
CREATE { PROCEDURE|FUNCTION|TRIGGER} program_name
BEGIN
    program_statements
END;
```


使用程序块有两个原因：

将逻辑相关的代码部分放在一起

举例来说，一个错误处理声明（见第六章对于错误处理的解释）可以包含一个能够执行多行命令的块。在块中的所有所有语句都将在错误处理被调用是被执行

控制变量和其他对戏那个的作用域

你可以在一个块中定义一个变量，这样在块的外部就无法看到这个变量。其次，你可以在一个块的内部定义一个覆盖块外部同名变量的变量



一个混合语句由一个 BEGIN-END 块构成，这个块其的作用就是闭合一个或多个存储程序指令

4.1.1 块的结构

一个块由多种不同的声明（比如说：变量，游标，错误处理单元）和程序代码（比如：赋值，条件语句，循环）。它们之间的顺序如下：

1. 变量和条件声明。变量早在第三章就讨论过，而条件声明将在第六章讨论。
2. 游标声明，将在第五章讨论
3. 处理单元声明，将在第六章讨论
4. 程序代码

如果你违反这个顺序，比如说当一个 DECLARE 语句位于 SET 语句之后，MySQL 将会在创建你的程序代码时产生一个错误信息。错误信息并不总会指出你没有按照顺序使用语句，所以注意培养自己声明事物的正确顺序很重要。



在块中语句的顺序必须是变量（Variables）和条件，接下来是游标（Cursors），然后是异常处理（Exception handlers），而最后是其语句（Other）。我们可以使用下面的藏头诗来助记：“Very Carefully Establish Order”（千万小心创建顺序）。

你也可以给你的块命名一个标签（label）。标签必须同时出现在 **BEGIN** 语句之前和 **END** 语句之后。命名一个标签具有以下优势：

- 这有助于改善代码的可读性，比如说者可以让你快速的找到与 **BEGIN** 语句匹配的 **END** 语句
- 这允许你使用 **LEAVE** 语句终止程序块的执行（见本章后面部分关于这种语句的描述）。

所以这种块结构的最简单形式如下：

```
[label:] BEGIN
    variable and condition declarations]
    cursor declarations
    handler declarations

    program code

END[label];
```

4.1.2. 嵌套块

如果每个存储程序都只包含一个块，那么块结构就会相当难以维护，虽然很多程序只包括一个最起码的块，并且吧所有的程序代码都闭合在这个主块中。正如早已提醒过的，在块中声明的变量在块的外部不可用，但是对于此块中定义的嵌套块却是可见的。你可以在块中覆盖定义“外部”变量，并且你可以在不影响外部变量的情况下操作这个内部变量

让我们举些例子来说明这些原理

在 **Example 4-1** 中，我们在一个块中创建了一个变量，这个变量在块的外部是不可用的，所以这个例子产生了一个错误

Example 4-1. 在块中定义的变量在块外部不可见

```
mysql> CREATE PROCEDURE nested_blocks1( )
BEGIN
    DECLARE outer_variable VARCHAR(20);
    BEGIN
        DECLARE inner_variable VARCHAR(20);
        SET inner_variable='This is my private data';
    END;
    SELECT inner_variable,' This statement causes an error ';
END;
$$
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL nested_blocks1( )
```

```
-----
```

```
ERROR 1054 (42S22): Unknown column 'inner_variable' in 'field list'
```

在 Example 4-2 中，我们在“内部”块中修改了一个由“外部”块声明的变量。这个修改对于内部块之外的作用域是可见的

Example 4-2. 内部块的变量可以覆盖外部块定义的变量

```
mysql> CREATE PROCEDURE nested_blocks2( )
```

```
BEGIN
```

```
    DECLARE my_variable varchar(20);
```

```
    SET my_variable='This value was set in the outer block';
```

```
    BEGIN
```

```
        SET my_variable='This value was set in the inner block';
```

```
    END;
```

```
    SELECT my_variable, 'Changes in the inner block are visible in the outer block';
```

```
END;
```

```
$$
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL nested_blocks2( )
```

```
//
```

```
+-----+-----+
| my_variable          | Changes in the inner block are visible in the outer block |
+-----+-----+
| This value was set   |                               |
| in the inner block | Changes in the inner block are visible in the outer block |
+-----+-----+
1 row in set (0.00 sec)
```

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

在 Example 4-3，我们在内部块中创建了一个与外部块同名的变量，当我们修改这个内部块中的变量时，这种变化并

没有反映到外部块中，这是因为虽然两个变量拥有相同的名字，但是他们其实是两个相对独立的变量。用这种方法在内部块中覆盖外部的变量会产生一定的困惑，这可能降低代码的可读性，并对错误的产生起了积极的作用，因此，一般来说不要使用这种覆盖变量的定义，除非你对此抱有相当强烈的目的

Example 4-3. 在内部块中重载的变量的修改对于块的外部并不可见

```
mysql> CREATE PROCEDURE nested_blocks3( )
BEGIN
    DECLARE my_variable varchar(20);
    SET my_variable='This value was set in the outer block';
    BEGIN
        DECLARE my_variable VARCHAR(20);
        SET my_variable='This value was set in the inner block';
    END;
    SELECT my_variable, 'Can't see changes made in the inner block';
END;
//
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> CALL nested_blocks3( )
$$
```

```
+-----+-----+
| my_variable          | Can't see changes made in the inner block |
+-----+-----+
| This value was set in the |          |
| outer block          | Can't see changes made in the inner block |
+-----+-----+
1 row in set (0.00 sec)
```

Query OK, 0 rows affected (0.00 sec)



请避免在内部块中覆盖定义外部块中已存在的变量

在我们最后一个嵌套块的例子中（Example 4-4）我们使用了块标签并使用 **LEAVE** 语句来终止块的执行。我们将会在本章的随后部分讨论 **LEAVE** 语句的使用，但现在，我们已经可以向你指明当你是用块标签的场合下，你可以随时使用 **LEAVE** 语句来终止一个块的执行。

Example 4-4. 使用 **LEAVE** 语句来离开（终止）一个标签块

```
mysql> CREATE PROCEDURE nested_blocks5( )
outer_block: BEGIN
```

```
DECLARE l_status int;
SET l_status=1;
inner_block: BEGIN
    IF (l_status=1) THEN
        LEAVE inner_block;
    END IF;
    SELECT 'This statement will never be executed';
END inner_block;
SELECT 'End of program';
END outer_block$$
```

Query OK, 0 rows affected (0.00 sec)

mysql> CALL nested_blocks5()\$\$

```
+-----+
| End of program |
+-----+
| End of program |
+-----+
1 row in set (0.00 sec)
```

Query OK, 0 rows affected (0.00 sec)

4.2 条件控制

条件控制（流程控制）语句允许你基于变量或表达式来执行代码。正如我们早就说过的，一个表达式可以由任何 MySQL 字面常量，变量，操作符和函数等能够返回值的元素构成。条件控制表达式允许你根据这些值或表达式返回值的不同而做出不同的行为，而这些值可以引用存储程序的参数，或许是数据库中的数据，或许是变量（如一周中的某一天或者一天中的某个时间）。

MySQL 存储程序语言支持两种不同的条件控制表达式：IF 和 CASE。它们都表现出十分相似的行为，并且通常我们总可以用 IF 语句来改写 CASE 语句的代码或者正好相反，通常来说 IF 和 CASE 之间的选择总是牵涉到个人的习惯和编程标准。但是，他们在不同的状况下总会表现出他们各自区别与对方的优势

在下面的小小章节中描述了他们各自的语法，提供了用例，最后对这两个语句的正反两方面进行了对比

4.2.1. IF 语句

所有的语言总是倾向于提供 IF 语句的各种变种，MySQL 对于 IF 语句的实现也不意外。存储程序中 IF 语句的语法是：

```
IF expression THEN commands
    [ELSEIF expression THEN commands ....]
    [ELSE commands]
END IF;
```

4.2.1.1. TRUE 或 FALSE（或者其他）？

在 IF 或者 ELSEIF 语句中的命令只有在与之配合的表达式测试为 TRUE 时才会执行。例如 `1=1` 或 `2>1` 的表达式会返回 TRUE。表达式 `1>3` 将被返回 FALSE。

虽然，假如你同时操作一个或多个变量，并且其中的一个变量值为 NULL，那么这个表达式的结果就是 NULL 而不是 TRUE 或 FALSE，这样的话如果你认为代码中的测试条件不是 TRUE 就是 FALSE（或正好相反），就会产生一些错误的结果，所以，举例来说，在 Example 4-5 中，如果我们在版本字符串中不能找到 ‘alpha’ 或 ‘beta’，我们就会以为这个发行版是正式的产品。虽然，如果 `l_version` 是 NULL，那么 ELSE 条件将总是被执行，但事实上我们任何依据来证明这种断言。

Example 4-5. 假设不是 TRUE 就是 FALSE 是错误的

```
IF (INSTR(l_version_string,'alpha')>0) THEN
    SELECT 'Alpha release of MySQL';
ELSEIF (INSTR(l_version_string,'beta')>0) THEN
    SELECT 'Beta release of MySQL';
ELSE
    SELECT 'Production release of MySQL';
END IF;
```



不要假设测试表达式的结果不是 **TRUE** 就是 **FALSE**。它在表达式中的任何一个变量为 **NULL** 时也可以返回 **NULL** (**UNKNOWN**)。

同样要注意任何返回数字值或字符串的表达式看上去是数字，但测试结果可能是 **TRUE**，**FALSE** 或 **NULL**。规则如下：

- 任何绝对值大于 1 的数字表达式，在 **IF** 或 **ELSEIF** 语句的测试结果为 **TRUE**。注意“绝对值”的称呼意味无论是 1 还是 -1 测试结果都为 **TRUE**
- 如果数字值表达式为 0，则测试结果为 **FALSE**

4.2.1.2 简单的 IF-THEN 组合

在最简单的形式中，**IF** 可以被指向一个语句集合，只要其中的一个条件测试为 **TRUE** 就执行，这种 **IF** 表达式的语法如下：

```
IF expression THEN
    statements
END IF;
```

三值逻辑

布尔表达式可以返回三个可能的结果，当布尔表达式中的值都是已知时，结果可能是 **TRUE** 或 **FALSE**。举例来说，如同下面的表达式无疑会返回真或假

```
(2 < 3) AND (5 < 10)
```

有时，你可能并不知晓所有表达式的值，这是因为数据库允许值为 **NULL** 或者被遗漏。那么随后会发生什么，从表达式返回的结果可以包含 **NULL** 吗？请看例子：

```
2 < NULL
```

因为你不知道遗漏的值是哪一个，你所能给的唯一答案就是“我不知道。”这就是最为基本的，所谓的“三值逻辑”，所以你能得到的可能答案不仅是 **TRUE** 和 **FALSE**，还有 **NULL**。

关于三值逻辑的深层次知识，我们推荐 C.J.Date 的书《Database In Depth》(O'Reilly) 为从业者准备的关系理论书籍

Example 4-6 展示了简单的 **IF** 语句

Example 4-6. IF 语句的简单例子

```
IF sale_value > 200 THEN

    CALL apply_free_shipping(sale_id);
```

```
END IF;

;
```

我们可以 THEN 和 END IF 子句中包含多个语句，就像 Example 4-7

Example 4-7. IF 语句中的多个语句

```
IF sale_value > 200 THEN
    CALL apply_free_shipping(sale_id);
    CALL apply_discount(sale_id,10);
END IF;
```

正如 Example 4-8 所展示的那样，我们同样可以在 IF 语句中包含任何可执行语句，比如循环结果，SET 语句和别的 IF 语句（当然，正如你即将看到的一样，我们竟可能在这种形式中避免使用嵌套 IF 语句）。

Example 4-8. 嵌套 IF 语句

```
IF sale_value > 200 THEN
    CALL apply_free_shipping(sale_id);
    CALL apply_discount(sale_id,10);
END IF;
```

正如 Example 4-9 所示，没有必要为 IF 语句断行，在这个例子中的所有 IF 语句都被 MySQL 一视同仁。

Example 4-9. IF 语句的候选格式

```
IF sale_value > 200 THEN CALL apply_free_shipping(sale_id); END IF;

IF sale_value > 200
THEN
    CALL apply_free_shipping(sale_id);
END IF;

IF sale_value > 200 THEN
    CALL apply_free_shipping(sale_id);
END IF;
```

大体上把非常简单的 IF 语句放在同一行上是没有问题的，但是对于复杂的或者嵌套的 IF 结构而言这却不是一种很好的编程实践（风格）。举例来说，以下这种格式更易读，易理解，易维护：

```
IF sale_value > 200 THEN
    CALL apply_free_shipping(sale_id);
    IF sale_value > 500 THEN
        CALL apply_discount(sale_id,20);
    END IF;
END IF;
```

```
END IF;
```

或者是：

```
IF sale_value > 200 THEN CALL apply_free_shipping(sale_id); IF sale_value >
500 THEN CALL apply_discount(sale_id,20);END IF;END IF;
```

某些程序员喜欢将 **THEN** 子句放在独立的一行上，如下所示：

```
IF sale_value > 200
THEN
    CALL apply_free_shipping(sale_id);
END IF;
```

但这更像是个人的喜好或者编程标准



对于一个非凡的 IF 语句而言，使用缩排和格式化来确保你的 IF 语句的逻辑更容易理解

4.2.1.3. IF-THEN-ELSE 语句

为你的 IF 语句增加 ELSE 条件允许你在假设 IF 条件非真时执行你所需的代码，我们还是要再次强调一下非真并不总代表假。如果 IF 语句条件测试为假，那么 ELSE 语句任然会被执行；这当你无法保证 IF 条件中出现 NULL 值时可能会带来微妙的错误。

IF-THEN-ELSE 块有如下语法：

```
IF expression THEN
    statements that execute if the expression is TRUE
ELSE
    statements that execute if the expression is FALSE or NULL
END IF;
```

所以我们在 Example 4-10 中，我们在销售额低于\$200 时对某个订单进行托运，否则我们就对其进行折扣处理（并且不处理托运）

```
IF sale_value <200 THEN
    CALL apply_shipping(sale_id);
ELSE
    CALL apply_discount(sale_id);
END IF;
```


4.2.1.4. IF-THEN-ELSEIF-ELSE 语句

IF 语句的完整语法允许你定义多个条件。第一个测试为 TRUE 的条件将被执行。如果没有任何表达式测试为 TRUE，那么 ELSE 子句（如果存在）将被执行。IF-THEN-ELSEIF-ELSE 语句的语法是这样的

```
IF expression THEN
    statements that execute if the expression is TRUE
ELSEIF expression THEN
    statements that execute if expression1 is TRUE
ELSE
    statements that execute if all the preceding expressions are FALSE or NULL
END IF;
```

你可以在其中放置任意多的 ELSEIF 条件

条件之间并非互相排斥。这意味着，不止一个条件可能被测试为真。而第一个被测试为真的条件将会得到执行。创建叠加的条件看起来会很有用，但你排放条件时必须非常小心。举例来说，考虑一下 Example 4-11 的 IF-ELSEIF 语句。

Example 4-11. 使用叠加条件的 IF-ELSEIF 块的例子

```
IF (sale_value>200) THEN
    CALL free_shipping(sale_id);
ELSEIF (sale_value >200 and customer_status='PREFERRED') THEN
    CALL free_shipping(sale_id);
    CALL apply_discount(sale_id,20);
END IF;
```

这段代码的意图非常明确：对于订购超过\$200 的客户提供免费托运，并且给与优先客户以 20%的折扣，虽然因为所有超过\$200 的订购将使第一个条件测试为真，但是对于 ELSEIF 中订购超过\$200 的测试将永远得不到执行，这意味着优先的客户将不能得到他们的折扣，客户没有得到折扣意味着我们的存储过程程序员就得不到年终的奖金！

有一大堆的方法可以让这个语句变得更严谨：其中的一种是，我们可以将 ELSEIF 条件分支移入 IF 子句中来确保它首先得到测试，或者，我们可以将一个包含 sale_value>200 条件的 IF 语句嵌套与 IF 子句中来测试客户的状态，就像 Example 4-12 所展示的那样

Example 4-12. 两种纠正前面例子中逻辑错误的方面

```
/* Reordering the IF conditions */
IF (sale_value >200 and customer_status='PREFERRED') THEN
    CALL free_shipping(sale_id);
    CALL apply_discount(sale_id,20);
ELSEIF (sale_value>200) THEN
    CALL free_shipping(sale_id);
END IF;
```

```

/* Nesting the IF conditions */

IF (sale_value >200) THEN
    CALL free_shipping(sale_id);
    IF (customer_satus='PREFERRED') THEN
        CALL apply_discount(sale_id,20);
    END IF;
END IF;

```

在 Example 4-12 中的两种替换方案几乎都是完美有效的。一般来说我们总是希望尽可能避免嵌套 IF 语句，但是当存在大量的 sale_value 大于 200 的条件分支需要我们去处理，那么对 sale_value 进行一次测试，然后对其他个别的条件进行测试就变得有意义。所以我们的业务规则就是给订购超过\$200 的客户以免费托运，并且让我们的客户忠实度程序基于客户的状态来给出不同的折扣率。这种在 IF-ELSEIF 块中的单个逻辑就像 Example 4-13 所演示的一样。

Example 4-13. IF 块伴随着大量的冗长的条件

```

IF (sale_value >200 and customer_status='PLATINUM') THEN
    CALL free_shipping(sale_id);      /* Free shipping*/
    CALL apply_discount(sale_id,20); /* 20% discount */

ELSEIF (sale_value >200 and customer_status='GOLD') THEN
    CALL free_shipping(sale_id);      /* Free shipping*/
    CALL apply_discount(sale_id,15); /* 15% discount */

ELSEIF (sale_value >200 and customer_status='SILVER') THEN
    CALL free_shipping(sale_id);      /* Free shipping*/
    CALL apply_discount(sale_id,10); /* 10% discount */

ELSEIF (sale_value >200 and customer_status='BRONZE') THEN
    CALL free_shipping(sale_id);      /* Free shipping*/
    CALL apply_discount(sale_id,5); /* 5% discount*/

ELSEIF (sale_value>200) THEN
    CALL free_shipping(sale_id);      /* Free shipping*/

END IF;

```

在这个例子中不变的重复着的 sale_value 条件和 free_shipping 调用，实际上渐渐的破坏了代码的可读性，并且是效率降低（见第 22 章）。如果使用嵌套的 IF 结构来清晰的给每个超过\$200 的客户以免费托运，而后，根据仅客户忠诚度给与相应的折扣会显得更好。Example 4-14 展示了嵌套 IF 的实现版本。

Example 4-14. 使用嵌套 IF 来避免呈长的测试条件

```

IF (sale_value > 200) THEN

```

```

CALL free_shipping(sale_id);    /*Free shipping*/

IF (customer_status='PLATINUM') THEN
    CALL apply_discount(sale_id,20); /* 20% discount */

ELSEIF (customer_status='GOLD') THEN
    CALL apply_discount(sale_id,15); /* 15% discount */

ELSEIF (customer_status='SILVER') THEN
    CALL apply_discount(sale_id,10); /* 10% discount */

ELSEIF (customer_status='BRONZE') THEN
    CALL apply_discount(sale_id,5); /* 5% discount*/
END IF;

END IF;

```

4.2.2. CASE 语句

CASE 语句是一种条件执行或者流程控制的可替换形式。任何 CASE 语句能做的事都可以用 IF 语句完成（或者正好相反），但是 CASE 语句通常更可读并且在处理多个测试条件时更有效，特别是当所有的条件都输出比对同一个表达式时

4.2.2.1. 简单 CASE 语句

CASE 语句有两种形式。第一种形式通常被称为简单 CASE 语句，用来作为多个表达式的输出相比对

```

CASE expression
  WHEN value THEN
    statements
  [WHEN value THEN
    statements ...]
  [ELSE
    statements]
END CASE;

```

这种语法在检测若干清晰的表达式输出集合时会很有用。举例来说，我们可以使用简单 CASE 语句来检查前一个样例中的客户忠诚度状态，就像 Example 4-15 所示

Example 4-15. 简单 CASE 表达式的例子

```
CASE customer_status
  WHEN 'PLATINUM' THEN
    CALL apply_discount(sale_id,20); /* 20% discount */

  WHEN 'GOLD' THEN
    CALL apply_discount(sale_id,15); /* 15% discount */

  WHEN 'SILVER' THEN
    CALL apply_discount(sale_id,10); /* 10% discount */

  WHEN 'BRONZE' THEN
    CALL apply_discount(sale_id,5); /* 5% discount */
END CASE;
```

就像 IF 命令一样，你可以指定多个 WHEN 语句并且你可以指定一个 ELSE 子句来执行其他条件未得到满足时的情况

当然，你必须清醒的认识到 CASE 语句将会在没有任何条件匹配的情况下产生异常，这意味着在 Example 4-15 中如果 customer_status 不是 'PLATINUM', 'GOLD', 'SILVER' 或 'BRONZE' 其中的任何一个，那么接下来就会发生运行时异常：

ERROR 1339 (20000): Case not found for CASE statement

我们可以为此创建一个异常处理单元来忽略这种错误（就像第六章中所描述的一样），但是也许使用一个 ELSE 子句来确保所有可能的条件都得到处理是一种更好的编程实践。因此，我们或许应该适应前面的例子，给其增加一个 ELSE 子句来给那些没有遇到过前面这种情况的客户以 0 折扣率



如果没有任何一个 CASE 语句与输入的条件相匹配，CASE 将会生成一个 MySQL 1339 错误。你可以为其创建一个错误处理单元来忽略错误，或者在 CASE 语句中使用 ELSE 子句确保异常永远不再发生

在简单 CASE 语句中比对一些列表表达式的值显得很有用。虽然，简单的 CASE 语句并不能轻易的，完全的匹配范围，或者处理更为复杂的包含多表达式的条件，因此，对于那些更为复杂的“情况”我们可以使用“查询”CASE 语句，将在下一节中被描述。

4.2.2.2. "查询" CASE 语句

查询 CASE 语句和 IF-ELSEIF-ELSE-END 块所具备的功能基本等同，查询 CASE 语句具有如下语法：

```
CASE
```

```
WHEN condition THEN
    statements
[WHEN condition THEN
    statements...]
[ELSE
    statements]
END CASE;
```

使用查询 CASE 结构，我们可以实现早些段落中是用 IF 来实现的免费托运和折扣率的逻辑。在 Example 4-16 中我们展示了是用查询 CASE 语句来是先的销售折扣率和免费托运的逻辑。

Example 4-16. 查询 CASE 表达式的例子

```
CASE
WHEN (sale_value >200 AND customer_status='PLATINUM') THEN
    CALL free_shipping(sale_id);    /* Free shipping*/
    CALL apply_discount(sale_id,20); /* 20% discount */

WHEN (sale_value >200 AND customer_status='GOLD') THEN
    CALL free_shipping(sale_id);    /* Free shipping*/
    CALL apply_discount(sale_id,15); /* 15% discount */

WHEN (sale_value >200 AND customer_status='SILVER') THEN
    CALL free_shipping(sale_id);    /* Free shipping*/
    CALL apply_discount(sale_id,10); /* 10% discount */

WHEN (sale_value >200 AND customer_status='BRONZE') THEN
    CALL free_shipping(sale_id);    /* Free shipping*/
    CALL apply_discount(sale_id,5); /* 5% discount*/

WHEN (sale_value>200) THEN
    CALL free_shipping(sale_id);    /* Free shipping*/

END CASE;
```

虽然请始终记住如果没有任何 WHERE 子句被匹配时，将会产生一个 1339 错误，这些代码将在订单少于\$200 或者客户并不在我们的忠诚度程序时会导致致命的错误，因此我们必须插入 ELSE 子句来保护我们的代码和工作的安全，正如 Example 4-17 所做的那样。

Example 4-17 为我们的查询 CASE 例子增加一个“哑” ELSE 子句

```
CASE
WHEN (sale_value >200 AND customer_status='PLATINUM') THEN
    CALL free_shipping(sale_id);    /* Free shipping*/
```

```

        CALL apply_discount(sale_id,20); /* 20% discount */

    WHEN (sale_value >200 AND customer_status='GOLD') THEN
        CALL free_shipping(sale_id);      /* Free shipping*/
        CALL apply_discount(sale_id,15); /* 15% discount */

    WHEN (sale_value >200 AND customer_status='SILVER') THEN
        CALL free_shipping(sale_id);      /* Free shipping*/
        CALL apply_discount(sale_id,10); /* 10% discount */

    WHEN (sale_value >200 AND customer_status='BRONZE') THEN
        CALL free_shipping(sale_id);      /* Free shipping*/
        CALL apply_discount(sale_id,5); /* 5% discount*/

    WHEN (sale_value>200)      THEN
        CALL free_shipping(sale_id);      /* Free shipping*/

END CASE;

```

注意因为 MySQL 在存储程序语言中缺乏 NULL 语句（不干任何事的），所以我们必须加入一个哑语句，但是这个语句开销几乎为 0

就像我们用 IF 所实现的逻辑一样，我们可以使用嵌套 CASE 语句来是先出一个逻辑更为清晰的版本。在 Example 4-18 中我们结合了简单 CASE 和查询 CASE 语句，并且包含了一个“not found”的错误处理单元来避免使用 ELSE 语句。我们使用块来闭合其中的内容，因此我们的处理单元并不会感应到此存储程序中的其他语句

Example 4-18. 嵌套 CASE 语句和一个块作用域的“not found”处理单元

```

BEGIN
    DECLARE not_found INT DEFAULT 0;
    DECLARE CONTINUE HANDLER FOR 1339 SET not_found=1;

    CASE
        WHEN (sale_value>200) THEN
            CALL free_shipping(sale_id);
            CASE customer_status
                WHEN 'PLATINUM' THEN
                    CALL apply_discount(sale_id,20);
                WHEN 'GOLD' THEN
                    CALL apply_discount(sale_id,15);
                WHEN 'SILVER' THEN
                    CALL apply_discount(sale_id,10);

```

```
        WHEN 'BRONZE' THEN  
            CALL apply_discount(sale_id,5);  
        END CASE;  
    END CASE;  
  
END;
```

4.2.3 IF 和 CASE 的比较

我们已经看到无论是 IF 还是 CASE 语句都能实现相同的流程控制功能。那么哪一个是最好的？在很大程度上，对两者的选择更关乎与个人风格和编程标准，而并非取决于两者所暗示的优势。尽管如此，我们在对 CASE 和 IF 进行选择时，可以遵照如下的原则：

- 一致的风格往往比两者在其各自的特殊场合所显现的微弱优势更重要。我们更希望你能够一致的选择你的 CASE 或者 IF，而不是根据你的心情，天气或者掷色子来随机的决定使用哪一个
- 当你使用一个表达式来和一系列清晰的值相对比时 CASE 会略显清晰（使用“简单”CASE 语句）。
- 当你基于多个变量对一系列复杂表达式进行测试时，IF 结构可能会显得更为有好并容易理解
- 如果你选择 CASE，你必须保证至少一个 CASE 条件得到匹配，或者定义一个错误处理单元来捕获错误，如果没有任何 CASE 条件得到满足，那么使用 IF 就没有这种烦恼。

无论你选择哪一个结构，请记住一下要点

- 如果 CASE 或 IF 结构中的条件得到满足，则别的条件将不再得到测试，这意味这你的条件排放顺序需要相当的严格
- MySQL 存储程序语言使用三值逻辑；因此若是一个语句是非真并不意味着他必定为 FALSE，也可能是 NULL。
- 你必须仔细的思考你代码的可读性，优势使用 IF 或 CASE 嵌套集可以使代码更可读，更有效。虽然在通常状况下应该避免嵌套的发生，特别是那些嵌套层次很深的代码（这里指的是三层或三层以上）。

4.3. 循环中的迭代处理

在这一节中我们将具体讨论 MySQL 存储程序语言所提供的迭代（重复）处理命令。基于许多原因促使程序需要迭代：

- 程序可以提供一个可以进行主循环的接口进行等待，然后处理，接受用户输入（当然，这在存储程序中并不被支持）。
- 许多数学算法只有使用计算机程序中的循环来实现
- 当处理一个文件时，程序应该在文件中的每一条记录间循环并进行处理
- 一个数据库程序应该在记录间循环并使用 `SELECT` 语句进行返回。

有一点非常显而易见，上述的最后一个原因使用 `SELECT` 语句返回记录可能是 MySQL 存储程序中需要是用循环的最大原因，并且我们将对此在第五章进行着重讨论，而在这一章中，我们只考察循环命令的一般形式

4.3.1. 循环语句

这可能是循环语句中最简单的结构，语法如下：

```
[label:] LOOP
    statements
END LOOP
[label];
```

在 `LOOP` 和 `END LOOP` 之间的语句将会无限循环，知道 `LOOP` 终止。你可以使用 `LEAVE` 语句来终止 `LOOP`，接下来将会简短的对其进行描述。

你可以给循环命名一个标签，这与我们给 `BEGIN-END` 块加上标签具有相同的语法。标签可以帮助我们区别特定的 `LOOP` 语句。同样重要的是，标签还能够用于控制执行流程，就像接下来的章节中我们会看到的那样。

Example 4-19 展示了一个非常简单的循环（但非常危险），他会永远运行下去，或者知道你设法终止它位置。因为存储程序实在数据库服务器内部运行的，所以使用 `Ctrl-C` 或者其他例如使用 `KILL` 命令来终止 MySQL 会话进程的方法将其结束，或者关闭数据库服务器，同时，循环会占用大量的 CPU 资源，所以，我们不推荐你在关键任务系统中运行以下示例

Example 4-19. 无限循环（不要轻易自我尝试）

```
Infinite_loop: LOOP
    SELECT 'Welcome to my infinite
loop from hell!!';
END LOOP infinite_loop;
```

显而易见，我们几乎都不希望程序掉进死循环，因此我们需要一些方法来终止循环。我们可以用 `LEAVE` 语句来做到这一切，所以让我们不要质疑，马上就开始...

4.3.2. LEAVE 语句

LEAVE 语句允许你终止循环，LEAVE 的大致语法是这样的：

```
LEAVE label;
```

LEAVE 会使当前循环终止，标签匹配了要终止的循环，所以当循环被另一个循环所闭合，我们可以使用一个语句忠实所有的循环。

在 Example 4-20 这个最简单的例子中我们执行一个 LEAVE 当我们准备离开循环

Example 4-20. 使用 LEAVE 来终止循环

```
SET i=1;
myloop: LOOP
    SET i=i+1;
    IF i=10 then
        LEAVE myloop;
    END IF;
END LOOP myloop;
SELECT 'I can count to 10';
```

LEAVE 可以用来终止任何可选的循环结构，就像我们接下来所要讲的那样，事实上，你也可以使用 LEAVE 来终止一个已命名 BEGIN-END 块（早在这一章早些部分已被介绍过）

4.3.3. ITERATE 语句

ITERATE 语句用来重新从循环头部开始执行，而不执行任何在循环中遗留下来的语句，ITERATE 的语法如下：

```
ITERATE label;
```

当 MySQL 遇到 ITERATE 语句是，它会从新跳转到指定循环的头部开始执行，在 Example 4-21 中，我们打印了 10 以下的偶数。ITERATE 将在我们得到的数是非偶数是重复循环。LEAVE 是用来在我们达到 10 时终止循环的。

Example 4-21. 使用 ITERATE 返回到循环头部

```
SET i=0;
loop1: LOOP
    SET i=i+1;
    IF i>=10 THEN          /*Last number - exit loop*/
        LEAVE loop1;
    ELSEIF MOD(i,2)=0 THEN /*Even number - try again*/
        ITERATE loop1;
    END IF;
```

```
SELECT CONCAT(i," is an odd number");

END LOOP loop1;
```

当这个循环清晰的指出 **LEAVE** 和 **ITERATE** 对于循环的控制作用的时候，其本省的算法结构却是失败的，我们可以简单的在循环中每次对循环计数变量增加 2 而不是 1 来实现它。

ITERATE 是的执行能够重新从循环的头部开始，。如果你使用 **REPEAT** 循环（将在下一节中介绍哦），这意味那个循环将无条件的重新执行，直到船体给 **UNTIL** 的条件得到满足才终止循环。这有可能导致不预期的行为。在 **WHILE** 循环中，**ITERATE** 将在下一次迭代循环之前对 **WHILE** 条件重新测试。

我们可以使用 **LOOP**，**LEAVE** 和 **ITERATE** 语句构造任何可以想象的循环形式。当然在实践中使用这些“手工”构造的循环比起我们应该考虑的其他备选法案来，显的有些笨拙。将要在下一节描述的 **WHILE** 循环和 **REPEAT** 循环允许你创建更可写，可读，可维护的循环。

4.3.4.REPEAT ... UNTIL 循环

REPEAT 和 **UNTIL** 语句可以用来创建一直重复直到遇到某些逻辑条件才终止的循环，**REPEAT...UNTIL** 循环的语法如下：

```
[label:] REPEAT
    statements
UNTIL expression
END REPEAT [label]
```

REPEAT 循环将会一直执行，知道定义在 **UNTIL** 子句中的测试条件为 **TRUE** 为止。基本上，**REPEAT** 循环逻辑上等同与 **LOOP-LEAVE-END LOOP** 块

```
some_label:LOOP
    statements
    IF expression THEN LEAVE some_label; END IF;
END LOOP;
```

REPEAT 循环从某种程度上看起来更易维护，因为它清晰的给出了在什么条件下终止循环。**LEAVE** 在简单循环中可以出现在任何地方，而 **UNTIL** 语句总是伴随这 **END REPEAT** 子句出现在循环的最底端。此外，我们不必为 **REPEAT** 循环给出一个标签，因为 **UNTIL** 条件总是指向当前循环，当然我们还是建议你对 **REPEAT** 循环加上标签来改善可读性，特别是循环被嵌套时。

Example 4-22 展示了使用 **REPEAT** 来打印小于 10 的奇数。请对比我们前面使用 **LOOP** 和 **LEAVE** 语句的语法

Example 4-22.REPEAT 循环的例子

```
SET i=0;
loop1: REPEAT
    SET i=i+1;
    IF MOD(i,2)<>0 THEN /*Even number - try again*/
```

```

    Select concat(i," is an odd number");
END IF;
UNTIL i >= 10
END REPEAT;

```

同样，REPEAT 还是具有一些没有价值的东西

REPEAT 循环的循环体总是能确保至少运行一次，这意味着，UNTIL 将爱第一次循环体被执行后才得到测试，对于那些不需要确保至少运行一次而只有条件得到满足才运行的场合下，请使用 WHILE（看下一节）。

在 REPEAT 循环中使用 ITERATE 可能会导致不预期的结果，因为这样做的话，可能会在传递给 UNTIL 的条件尚未得到满足的情况下结束本次循环，所以，你也许不希望在 REPEAT 循环中使用 ITERATE。

4.3.5. WHILE 循环

WHILE 循环只有在条件为真是才执行循环。如果条件不为真，那么循环体将永远的得不到执行，这一点并不像 REPEAT 循环，它的循环体至少能被执行一次

WHILE 循环的语法如下：

```

[label:] WHILE expression DO
    statements
END WHILE [label]

```

WHILE 循环的功能大致等同于简单的 LOOP-LEAVE-END LOOP 结构，它将 LEAVE 子句作为第一个语句，就像“LEAVE 语句”一节中所描述的那样。Example 4-23 证明了这种 LOOP-LEAVE-END-LOOP 结构

Example 4-23. 使用 LOOP-END LOOP 来实现 WHILE 循环的功能

```

myloop: LOOP
    IF expression THEN LEAVE myloop; END IF;
    other statements;
END LOOP myloop;

```

Example 4-24 展示了我们使用 WHILE 循环实现的求得小于 10 的奇数的版本

Example 4-24. 使用 WHILE 实现求得小于 10 的奇数

```

SET i=1;
loop1: WHILE i<=10 DO
    IF MOD(i,2)<>0 THEN /*Even number - try again*/
        SELECT CONCAT(i," is an odd number");
    END IF;
    SET i=i+1;
END WHILE loop1;

```

4.3.6. 嵌套循环

我们经常要使用嵌套循环，在 Example 4-25 这段简单的代码中，我们使用嵌套 LOOP-LEAVE-END LOOP 结构打印了基本的“时间表”

Example 4-25. 嵌套循环的例子

```
DECLARE i,j INT DEFAULT 1;
outer_loop: LOOP
    SET j=1;
    inner_loop: LOOP
        SELECT concat(i," times ", j," is ",i*j);
        SET j=j+1;
        IF j>12 THEN
            LEAVE inner_loop;
        END IF;
    END LOOP inner_loop;
    SET i=i+1;
    IF i>12 THEN
        LEAVE outer_loop;
    END IF;
END LOOP outer_loop;
```

当嵌套一个循环式，对于循环的开始和结束的进行标签命名可以使其看起来更清晰，同样对于我们分清循环的各个层次也有很大的帮助。当然，如果你需要使用 LEAVE，你就必须使用标签对循环命名。

4.3.7. 对循环的部分注释

现在，我们已经认识了三种简单循环，以及使用三种 MySQL 存储程序语言所提供的不同结构的循环实现的循环方法。三种循环结构事实上已经覆盖了所有你需要自己实现的循环逻辑

在这一章的例子中所给出的循环事实上都非常的简单，并且和现实时间的关联也很小，我们这样做的目的是让一切看起来足够清晰，但是在你实际的存储程序编程过程中都回在循环结构中使用 SELECT 语句对数据行进行迭代，而这正式下一章所要诉说的内容。

4.4. 第四章结语

在这一章中我们认识了 MySQL 存储程序语言的条件和迭代控制结构，几乎所有具有一定规模的程序都回使用到各种基于输入数据的决定，而这些决定通常都是通过 IF 或 CASE 语句进行表达的

循环是另一种存储程序编程时所要用到的常规操作，你可以用之与 SQL 语句配合对输出结果进行迭代。MySQL 提供了大量的循环备选方案，包括使用 LEAVE 语句终止的简单循环，REPEAT UNTIL 循环和 WHILE 循环。

第五章 在存储编程中使用 SQL

当你使用 MySQL 存储程序语言来完成传统的编程任务时会注意到，实际上所有的存储程序都会频繁的执行 SQL 语句来和数据库交互。这一章中我们主要关注怎样在存储程序中使用 SQL

在这一章中我们会认识到很多种在存储程序中使用 SQL 的方法：

- 简单 SQL 语句（非 SELECT）是一种可以被轻松嵌入存储过程和函数中的不返回结果集的语句
- 一个 SELECT 语句可以将返回的单个记录适用 INTO 传入本地变量
- 一个 SELECT 语句返回多个记录时，你可以使用基本的游标在各个记录之间循环，并且对那些记录使用任何你认为适合的操作。
- 任何 SELECT 语句都可以使用 INTO 子句和 CURSOR 语句“不受限的”被包含在存储过程中（但不是存储函数）。这些 SQL 语句将会把结果返回给它的调用程序（但是很可惜，不能直接返回给存储过程）。
- SQL 语句可以使用筹备语句在服务器端动态的被筹备（只有在存储过程中被使用）

5.1.在存储程序中使用非 SELECT SQL 语句

当我们在存储程序中使用不返回结果集的语句，例如：UPDATE, INSERT 或 SET 时，它会工作的非常正常，正如你在别的上下文环境中执行一样（比如你在 PHP 中调用或从 MySQL 命令行调用）

存储程序中的 SQL 语句与不使用存储过程调用的语法是相同的。SQL 语句具有对存储程序变量完全的访问权限，它通常可以使用字面常量或这表达式等返回的结果。

你可以在存储程序中使用大多数 SQL 语句，DML,DDL 和其他公用语句都可以不受限的使用

Example 5-1 结合使用了 DDL 和 DML 来创建和操纵表中的数据。

Example 5-1.在存储程序中内嵌非 SELECT 语句

```
CREATE PROCEDURE simple_sqls( )
BEGIN
    DECLARE i INT DEFAULT 1;

    /* Example of a utility statement */
    SET autocommit=0;

    /* Example of DDL statements */
    DROP TABLE IF EXISTS test_table ;
    CREATE TABLE test_table
        (id          INT PRIMARY KEY,
         some_data VARCHAR(30))
    ENGINE=innodb;

    /* Example of an INSERT using a procedure variable */
    WHILE (i<=10) DO
        INSERT INTO TEST_TABLE VALUES(i,CONCAT("record ",i));
        SET i=i+1;
    END WHILE;

    /* Example of an UPDATE using procedure variables*/
    SET i=5;
    UPDATE test_table
        SET some_data=CONCAT("I updated row ",i)
        WHERE id=i;

    /* DELETE with a procedure variable */
    DELETE FROM test_table
        WHERE id>i;

END;
```

5.2.在 SELECT 语句中使用 INTO 子句

如果你使用一个只返回一个记录的 SELECT 表达式,你可以在 SELECT 语句中使用 INTO 语句将返回传递给存储程序的变量,这种 SELECT 的格式为:

```
SELECT expression1 [, expression2 ....]
    INTO variable1 [, variable2 ...]
    other SELECT statement clauses
```

Example 5-2 展示了我们怎样从一个客户记录中得到明细。客户的 ID 被传递给参数

Example 5-2.使用 SELECT-INTO 语句

```
CREATE PROCEDURE get_customer_details(in_customer_id INT)
BEGIN
    DECLARE l_customer_name    VARCHAR(30);
    DECLARE l_contact_surname  VARCHAR(30);
    DECLARE l_contact_firstname VARCHAR(30);

    SELECT customer_name, contact_surname,contact_firstname
        INTO l_customer_name,l_contact_surname,l_contact_firstname
    FROM customers
    WHERE customer_id=in_customer_id;

    /* Do something with the customer record */

END;
```

如果 SQL 语句返回了超过一个记录,那么将会产生一个运行时错误。举例来说,如果我们在 Example 5-2 中忽略了 WHERE 子句,那么当试图运行这个存储过程时将产生如下错误:

```
mysql> CALL get_customer_details(2) ;
ERROR 1172 (42000): Result consisted of more than one row
```


5.3.创建和使用游标

当处理一个返回多个记录的 `SELECT` 语句时，我们必须为其创建和操纵一个游标。游标是一个可以访问你的 `SELECT` 语句返回结果集的可编程对象。你可以使用游标对你的结果集中的每条记录进行迭代并且赋予他们对各个结果的不同行为

目前为止，MySQL 只提供对于你 `SELECT` 语句返回结果集的自首至尾的依次迭代。你不能从最后一条记录开始访问，一直到第一条记录，也不可以直接跳转到你所指定的记录

5.3.1.定义游标

使用 `DECLARE` 语句来定义游标，它的语法如下：

```
DECLARE cursor_name CURSOR FOR SELECT_statement;
```

就像我们第三章所提到的一样，游标的声明必须在我们所有的变量声明之后。在我们的变量之前定义游标会产生一个 1337 错误，就像 Example 5-3 所演示的那样。

Example 5-3.在变量声明之前声明游标会产生一个 1337 错误

```
mysql> CREATE PROCEDURE bad_cursor( )
BEGIN
    DECLARE c CURSOR FOR SELECT * from departments;
    DECLARE i INT;
END;

ERROR 1337 (42000): Variable or condition declaration after cursor or handler declaration
```

游标总是和 `SELECT` 语句配合使用；Example 5-4 展示了一个声明为从一个客户表中获取特定列的游标

Example 5-4. 简单游标声明

```
DECLARE cursor1 CURSOR FOR
    SELECT customer_name, contact_surname,contact_firstname
    FROM customers;
```

一个游标可以使用 `WHERE` 子句来引用存储程序中的变量或表中的列列表（不过很少会发生）。在 Example 5-5 中，其中的游标在 `WHERE` 子句和 `SELECT` 列表中包含了对存储程序参数的引用，。当游标被打开时，它会使用参数中的变量值来决定具体返回哪一条记录

Example 5-5.在游标定义中包含一个存储程序变量

```
CREATE PROCEDURE cursor_demo (in_customer_id INT)
BEGIN
    DECLARE v_customer_id INT;
```

```
DECLARE v_customer_name VARCHAR(30);
DECLARE c1 CURSOR FOR
  SELECT in_customer_id, customer_name
  FROM customers
  WHERE customer_id=in_customer_id;
```

5.3.2. 游标语句

MySQL 存储程序员支持三种对游标不同操作的语句：

OPEN

初始化游标的结果集。我们必须在从游标中获取结果之前打开游标，OPEN 语句的语法相当简单

```
OPEN cursor_name;
```

FETCH

获取游标中的下一个记录并把游标在结果集中的“指针”下移。它的如法如下：

```
FETCH cursor_name INTO variable list;
```

在变量列表中，必须包含至少一个和在声明游标时包含的 SELECT 语句所返回的列相一致的数据类型的变量。我们将在本章的稍后部分来着重讨论 FETCH

CLOSE

解除游标并释放游标所占用的内存。这个语句的语法如下：

```
CLOSE cursor_name ;
```

我们必须在获取结果后关闭游标，或者在我们需要在改变游标结果集中的变量生效之后再次打开它。

在接下来的章节中，我们将看到很多有关这些语句的实际操作例子。

5.3.3. 从游标中获取单条记录

这是对于游标的最基本使用：我们打开一个游标，获取一条记录，然后关闭结果集，就像 Example 5-6 所示（打开的游标被定义与 Example 5-4）这个逻辑和简单的 SELET-INTO 语句相同。

Example 5-6. 从游标中获取单条记录

```
OPEN cursor1;
FETCH cursor1 INTO l_customer_name, l_contact_surname, l_contact_firstname;
```

```
CLOSE cursor1;
```

5.3.4. 获取所有记录集

游标最广泛的用途是用来处理由游标的 SELECT 语句返回的每个不同的记录，对于获得的数据进行一个或多个操作，然后在最后一条记录被获取后关闭游标。

Example 5-7 展示了如何声明和打开游标，然后自在循环中获取游标中的记录，并在最后关闭游标。

Example 5-7. 简单（有缺陷）游标循环

```
DECLARE c_dept CURSOR FOR
    SELECT department_id
    FROM departments;

OPEN c_dept;
dept_cursor: LOOP
    FETCH c_dept INTO l_dept_id;
END LOOP dept_cursor;
CLOSE c_dept;
```

虽然表面上这段代码看起来很明显很完整，但是它是有点问题的：如果你试图在游标中的最后一条记录之后获取数据，MySQL 将产生一个“no data to fetch”错误（MySQL 错误 1329；SQLSTATE 02000）。所以 Example 5-7 的代码将会异常退出，就像：

```
mysql> call simple_cursor_loop( );
ERROR 1329 (02000): No data to FETCH
```

为了避免错误，我们可以声明一个错误处理单元来捕获“no data to fetch”并设立一个标志位（被实现为一个本地变量）。然后我们询问那个标志变量来决定我们是不是已经获取了最后一行记录。使用这个技巧，我们可以主动的终止循环并关闭游标，同时，是的代码易于理解。

我们将在第六章中讨论错误处理，但现在我们可以向我们的代码加入如下语句：

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_row_fetched=1;
```

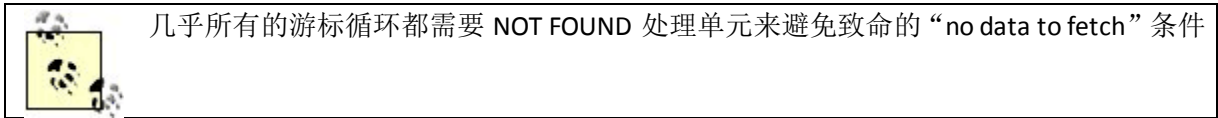
这个处理单元会促使 MySQL 遇到“no data to fetch”时做两件事情：

1. 设定变量“last row variable”（l_last_row_fetched）的值为 1；
2. 允许程序继续执行。

现在我们的程序可以检测变量 l_last_row_fetched 的值是否为 1，这样我们就知道最后一行记录是否已被获取，于是我们就可以终止循环并关闭游标

还有一件事情很重要，那就是当我们关闭游标后把“end of result set”指示器复位。否则的话，当我们下一次尝试从游标获取数据时，程序将立即终止循环，想到这点，你就知道我们这样做的用意了。

Example 5-8 展示了所有这些步骤：声明一个 **CONTINUE** 处理单元，循环结果集中的记录，当变量被设定事离开循环并释放资源。



Example 5-8. 简单游标循环

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_row_fetched=1;

SET l_last_row_fetched=0;
OPEN cursor1;
cursor_loop:LOOP
    FETCH cursor1 INTO l_customer_name,l_contact_surname,l_contact_firstname;
    IF l_last_row_fetched=1 THEN
        LEAVE cursor_loop;
    END IF;
    /*Do something with the row fetched*/
END LOOP cursor_loop;
CLOSE cursor1;
SET l_last_row_fetched=0;
```

注意我们并不需要处理结果集中的每条记录；我们随时可以使用 **LEAVE** 语句在我们需要的数据得到处理后终止循环。

5.3.5. 游标循环类型

我们可以使用三种循环结构中的任意一种来迭代游标返回的记录。并且其中的任何一种都需要我们在“last row variable”被 **NOT FOUND** 处理单元设置时终止循环。

考虑一下 Example 5-9 中的游标和 **NOT FOUND** 处理单元。

Example 5-9. 结合处理单元声明的游标

```
DECLARE dept_csr CURSOR FOR
    SELECT department_id,department_name, location
    FROM departments;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;
```

最简单的结构是 **LOOP-LEAVE-END LOOP** 序列。在这种情况下，我们的游标循环将像 Example 5-10 所示

Example 5-10. **LOOP-LEAVE-END LOOP** 游标循环

```

OPEN dept_csr;
dept_loop1:LOOP
    FETCH dept_csr INTO l_department_id,l_department_name,l_location;
    IF no_more_departments=1 THEN
        LEAVE dept_loop1;
    END IF;
    SET l_department_count=l_department_count+1;
END LOOP;
CLOSE dept_csr;
SET no_more_departments=0;

```

Example 5-10 的逻辑很简单：我们打开一个游标并迭代获取记录。如果我们尝试获取最后一条记录之后的记录，处理单元将设置 `no_more_departments` 为 1 并且我们可以调用 `LEAVE` 语句来终止循环。最后我们关闭游标并复位 `no_more_departments` 变量。

程序员大多总喜欢选择 `WHILE` 作为创建游标循环，并且这看起来也是很自然的选择。但事实上，你会发现 `REPEAT UNTIL` 更适合拿来创建游标循环，`REPEAT` 总能确保在条件表达式得到测试前它的循环体得到一次循环。在游标处理的上下文中，我们总是希望获取至少一条记录，而后在看是否应该对游标中的结果集进行处理。因此，使用 `REPEAT UNTIL` 循环可以产生更可读的代码，就像 **Example 5-11** 所示。

Example 5-11. 使用 REPEAT UNTIL 的游标循环

```

DECLARE dept_csr CURSOR FOR
    SELECT department_id,department_name, location
    FROM departments;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
REPEAT
    FETCH dept_csr INTO l_department_id,l_department_name,l_location;
UNTIL no_more_departments
END REPEAT;
CLOSE dept_csr;
SET no_more_departments=0;

```

当然，这个循环能够正常工作完全是因为我们对从游标获取的结果没有进行任何处理。只获取结果而不进行处理是很少见的，通常情况下我们会对返回的结果进行处理，我们需要一种方法来避免在最后一条记录之后进行处理的问题。所以事实上，即使我们使用 `REPEAT UNTIL` 循环，我们还是需要 `LEAVE` 语句来避免处理最后一次获取不存在的返回记录（或者是根本没有返回的记录）。那样的话，如果我们想对游标返回的结果技术（或者对这些结果进行别的处理）我们就必须为循环增加一个标签和 `LEAVE` 语句，就像我们 **Example 5-12** 所示的改良版本一样。

Example 5-12.大多数 Most REPEAT 还是需要 LEAVE 语句

```

DECLARE dept_csr CURSOR FOR
    SELECT department_id,department_name, location
    FROM departments;

```

```

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
dept_loop:REPEAT
    FETCH dept_csr INTO l_department_id,l_department_name,l_location;
    IF no_more_departments THEN
        LEAVE dept_loop;
    END IF;
    SET l_department_count=l_department_count+1;
UNTIL no_more_departments
END REPEAT dept_loop;
CLOSE dept_csr;
SET no_more_departments=0;

```

这种在几乎每个 **REPEAT UNTIL** 循环中增加 **LEAVE** 语句的必要性使得 **UNTIL** 子句的出现变得沉长，但不可争议的是它改善了代码的可读性并确保你在 **LEAVE** 语句执行失败（或许你的 **IF** 子句编码错误）的情况下不陷入死循环。最后，有效的游标循环可以被创建成任何形式，并且其中没有任何必要去推荐所谓最好的形式。我们所要说的是，把你的代码作为一个整体，如果你能选择一直的游标代码压实，将会使其变得更可读。

一种替换 **LEAVE** 语句的选择是使用 **IF** 来执行由 **FETCH** 到达结果集最后一行时决定的处理。**Example 5-13** 展示了这种循环结构。在这种情况下，**IF** 语句仅仅在 **no_more_departments** 变量尚未被设置的情况下才对我们的记录进行处理。

Example 5-13. 使用 **IF** 块来替换 **LEAVE** 语句在 **REPEAT UNTIL** 游标循环中的作用

```

DECLARE dept_csr CURSOR FOR
    SELECT department_id, department_name, location
    FROM departments;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
dept_loop:REPEAT
    FETCH dept_csr INTO l_department_id,l_department_name,l_location;
    IF no_more_departments=0 THEN
        SET l_department_count=l_department_count+1;
    END IF;
UNTIL no_more_departments
END REPEAT dept_loop;
CLOSE dept_csr;
SET no_more_departments=0;

```

第三种游标循环的形式使用的是 **WHILE-END WHILE** 循环。**WHILE** 会在循环体第一次执行前测试条件，所以它是一种与 **REPEAT-UNTIL** 或 **LOOP-END LOOP** 相比更缺乏逻辑的选择，因为从理论上来说我们不可能在获取至少一行记录前知道游标是否抵达末尾。另一方面，**WHILE** 可能是所有语言中用于构建循环结构使用最平凡的循环，所以这也告诉

那些不熟悉 MySQL 存储程序语言的程序员必须明确自己的意图。

在任何情况下，如果需要在循环中处理游标的结果，**WHILE** 循环同样需要 **LEAVE** 语句，所以 Example 5-14 的代码看起来和我们的前一个例子看起来很相似

Example 5-14.WHILE 游标循环

```
DECLARE dept_csr CURSOR FOR
    SELECT department_id, department_name, location
    FROM departments;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_departments=1;

SET no_more_departments=0;
OPEN dept_csr;
dept_loop:WHILE(no_more_departments=0) DO
    FETCH dept_csr INTO l_department_id, l_department_name, l_location;
    IF no_more_departments=1 THEN
        LEAVE dept_loop;
    END IF;
    SET l_department_count=l_department_count+1;
END WHILE dept_loop;
CLOSE dept_csr;
SET no_more_departments=0;
```

5.3.6.嵌套游标循环

嵌套游标循环并非不常用。举例来说，一个循环可能会获取一个我们关注的用户列表，同时，内层循环用于获取客户的订单。对于这种循环，最重要的事莫过于无论什么时候我们都将在游标的任务完成时设置 **NOT FOUND** 处理单元变量。所以你必须非常小心确保 **NOT FOUND condition** 不会使两层循环的游标都关闭。

举例来说，请思考 Example 5-15 的嵌套游标循环。

Example 5-15.嵌套游标循环（有缺陷）

```
CREATE PROCEDURE bad_nested_cursors( )
    READS SQL DATA
BEGIN

    DECLARE l_department_id INT;
    DECLARE l_employee_id   INT;
    DECLARE l_emp_count     INT DEFAULT 0 ;
    DECLARE l_done          INT DEFAULT 0;

    DECLARE dept_csr cursor FOR
        SELECT department_id FROM departments;

    DECLARE emp_csr cursor FOR
```

```

SELECT employee_id FROM employees
WHERE department_id=l_department_id;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_done=1;

OPEN dept_csr;
dept_loop: LOOP    Loop through departments
  FETCH dept_csr into l_department_id;

  IF l_done=1 THEN
    LEAVE dept_loop;
  END IF;

  OPEN emp_csr;
  SET l_emp_count=0;
  emp_loop: LOOP    -- Loop through employee in dept.
    FETCH emp_csr INTO l_employee_id;

    IF l_done=1 THEN
      LEAVE emp_loop;
    END IF;
    SET l_emp_count=l_emp_count+1;
  END LOOP;
  CLOSE emp_csr;

  SELECT CONCAT('Department ',l_department_id,' has ',
    l_emp_count,' employees');

END LOOP dept_loop;
CLOSE dept_csr;

END;
```

这个存储过程隐含了一些微妙的错误。当第一个“内层”循环中的游标 `emp_csr` 完成它的使命的时候，变量 `l_done` 被设置为 1。接下来，“外层”循环中的游标 `dept_csr` 开始下一次迭代，而此时变量 `l_done` 的值任然是 1，并且这是外层循环会不注意的被终止。结果，我们甚至只对一个部门进行了处理。面对这个问题，有两种可能的解决方案：可以很简单的在两次循环结束是复位“not found”变量，就像 Example 5-16 所示。

Example 5-16. 一个正确的嵌套游标的例子

```

CREATE PROCEDURE good_nested_cursors1( )
  READS SQL DATA
BEGIN

  DECLARE l_department_id INT;
  DECLARE l_employee_id   INT;
  DECLARE l_emp_count     INT DEFAULT 0 ;
```



```

DECLARE l_done          INT DEFAULT 0;

DECLARE dept_csr cursor FOR
    SELECT department_id FROM departments;
DECLARE emp_csr cursor FOR
    SELECT employee_id FROM employees
    WHERE department_id=l_department_id;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_done=1;

OPEN dept_csr;
dept_loop: LOOP    -- Loop through departments
    FETCH dept_csr into l_department_id;

    IF l_done=1 THEN
        LEAVE dept_loop;
    END IF;

    OPEN emp_csr;
    SET l_emp_count=0;
emp_loop: LOOP    -- Loop through employee in dept.
    FETCH emp_csr INTO l_employee_id;

    IF l_done=1 THEN
        LEAVE emp_loop;
    END IF;
    SET l_emp_count=l_emp_count+1;
END LOOP;
CLOSE emp_csr;
SET l_done=0;

SELECT CONCAT('Department ',l_department_id,' has ',
    l_emp_count,' employees');

END LOOP dept_loop;
CLOSE dept_csr;

END;

```

对于每次“not found”变量使用完毕后对其进行复位以确保接下来的游标迭代不受影响。



在每次游标循环终止是复位"not found"变量。如果不这样做，可能导致接下来的嵌套游标循环过早的终止。

一个稍稍复杂，但无可争议更强大的解决方案是给每一个游标以自己的处理单元，因为你只能在当前块中拥有一个 NOT FOUND 处理单元，这样的话你可以把每一个游标闭合在属于自己的一个块中。举例来说，我们可以将“销售”

游标和它的 NOT FOUND 处理单元至于它们自己的块中，就像 Example 5-17 所做的一样。

Example 5-17.使用嵌套块的嵌套游标

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_customer=1;

SET l_last_customer=0;
OPEN customer_csr;
cust_loop:LOOP      /* Loop through overdue customers*/

    FETCH customer_csr INTO l_customer_id;
    IF l_last_customer=1 THEN LEAVE cust_loop; END IF; /*no more rows*/
    SET l_customer_count=l_customer_count+1;

    sales_block: BEGIN
        DECLARE l_last_sale INT DEFAULT 0;

        DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_sale=1;
        OPEN sales_csr;
        sales_loop:LOOP    /* Get all sales for the customer */

            FETCH sales_csr INTO l_sales_id;
            IF l_last_sale=1 THEN LEAVE sales_loop; END IF; /*no more rows*/

            CALL check_sale(l_sales_id); /* Check the sale status */
            SET l_sales_count=l_sales_count+1;

        END LOOP sales_loop;
        SET l_last_sale=0;
        CLOSE sales_csr;
    END sales_block;

END LOOP cust_loop;
SET l_last_customer=0;
CLOSE customer_csr;
```

注意我们现在拥有了两个属于其各自游标的独立的“not found”变量，并且我们去除了各种可能造成在一个游标关闭却会影响另一个的情况。即使那样，你还是注意到我们在每一个使用完毕后将“not found”变量复位。这当你在同一个块中重新打开当前游标时被极力推荐。

5.3.7.过早的离开游标循环

不要假设只有在最后一行记录被获取时才会离开游标循环，任何时候你都可以在处理完毕后使用 LEAVE 语句。这时候你可能发现你需要处理的仅是结果集中的一条或少数几条记录，或者你发现其他对于数据更深层次的处理已经不再必要。

5.3.8.游标错误条件

游标语句必须依循 OPEN-FETCH-CLOSE 的顺序，任何这个顺序的变种都有可能导致运行时错误的发生。

举例来说，如果你试图关闭或获取一个尚未打开的游标，你就会遇到一个游标未打开错误，就像 Example 5-18 所示的那样。

Example 5-18.游标未打开错误

```
mysql> CREATE PROCEDURE csr_error2( )
BEGIN
    DECLARE x INT DEFAULT 0;
    DECLARE c cursor for select 1 from departments;
    CLOSE c;

END;

Query OK, 0 rows affected (0.00 sec)
mysql> CALL csr_error2( );

ERROR 1326 (24000): Cursor is not open
```

试图打开一个已打开结果集的游标，会产生一个游标已打开错误，就像 Example 5-19 所展示的那样。

Example 5-19.游标已打开错误

```
mysql> CREATE PROCEDURE csr_error3( )
BEGIN
    DECLARE x INT DEFAULT 0;
    DECLARE c cursor for select 1 from departments;
    OPEN c;
    OPEN c;

END;
//

Query OK, 0 rows affected (0.00 sec)
mysql> CALL csr_error3( );

ERROR 1325 (24000): Cursor is already open
```

5.4.使用非受限 SELECT 语句

MySQL 存储程序(但不包括函数)可以将结果集返回给调用程序(但非常不幸的是,不能直接返回给一个存储过程)。在存储程序中返回结果集时,那些不配合着 INTO 子句或游标,直接返回结果集的 SQL 语句被我们称为非首先 SQL 语句。这些 SQL 语句通常包括 SELECT 语句,当然其他用于存储过程的返回结果集的语句如 SHOW, EXPLAIN, DESC 也可以被包括在内。

我们已经在很多例子中使用过非受限 SELECT 语句,他们都被用于在存储过程执行过程中返回信息。你或许最希望将其用于 DEBUG 或将有用的状态信息返回给你的调用程序。Example 5-20 展示了一个用于返回指定部门的员工表的存储过程例子。

Example 5-20.使用非受限 SELECT 将数据返回给调用程序

```
CREATE PROCEDURE emps_in_dept(in_department_id INT)
BEGIN
    SELECT department_name, location
    FROM departments
    WHERE department_id=in_department_id;

    SELECT employee_id,surname,firstname
    FROM employees
    WHERE department_id=in_department_id;
END;
```

当运行时, Example 5-20 的存储过程产生了如下输出:

```
mysql> CALL emps_in_dept(31) //
+-----+-----+
| department_name | location |
+-----+-----+
| ADVANCED RESEARCH | PAYSON   |
+-----+-----+
1 row in set (0.00 sec)

+-----+-----+-----+
| employee_id | surname | firstname |
+-----+-----+-----+
| 149 | EPPLING | LAUREL |
| 298 | CHARRON | NEWLIN |
| 447 | RAMBO | ROSWALD |
| 596 | GRESSETT | STANFORD |
| 745 | KANE | CARLIN |
| 894 | ABELL | JAMIE |
| 1043 | BROOKS | LYNN |
| 1192 | WENSEL | ZENAS |
| 1341 | ZANIS | ALDA |
| 1490 | PUGH | ALICE |
| 1639 | KUEHLER | SIZA |
```

```
|      1788 | RUST      | PAINE      |
|      1937 | BARRY     | LEO        |
+-----+-----+-----+
13 rows in set (0.00 sec)
```

在某些方面，使用存储过程来返回结果集的功能和给特定的查询创建视图有些相似。和视图一样，存储过程一个封装复杂的 SQL 操作，这样就使用户不需要懂得复杂的结构设计就可以简单的获得数据。将 SQL 封装进存储过程同样可以改善安全性，因为你可以对 SQL 进行复杂的有效性检查，甚至在返回结果集之前进行封装/解封

和视图不同，存储过程可以返回多个结果集，就像 Example 5-20 所展示的那样。返回多个结果集可以方面的将所有所需的程序逻辑封装进一个对于数据库的程序数据调用中。

5.4.1. 在调用程序中获取结果集

在存储程序中获取结果集相对来说是一件简单的事。如果只是从存储过程中翻译一个结果集，那么就可以按照普通的 SQL 调用来对待。Example 5-21 展示了在 PHP 中使用 mysqli 接口从存储过程调用中返回单个结果集的例子。

Example 5-21. 从 PHP 中获取存储过程的返回结果集

```
1 <h1>Department listing</h1>
2 <table border="1" width="90%">
3 <tr> <td><b>Department ID</b></td>
4      <td><b>Department Name</b></td>
5 <?php
6     $hostname="localhost";
7     $username="root";
8     $password="secret";
9     $database="sqlltune";
10
11     $p1="";
12     $p2="";
13
14
15     $dbh = new mysqli($hostname, $username, $password, $database);
16
17     /* check connection */
18     if (mysqli_connect_errno( )) {
19         printf("Connect failed: %s\n", mysqli_connect_error( ));
20         exit( );
21     }
22
23     if ($result_set = $dbh->query("call department_list( )"))
24     {
25         printf('');
26         while($row=$result_set->fetch_object( ))
27         {
28             printf("<tr><td>%s</td><td>%s</td></tr>\n",
```

```

29         $row->department_id, $row->department_name);
30     }
31 }
32 else // Query failed - show error
33 {
34     printf("<p>Error retrieving stored procedure result set:%d (%s) %s\n",
35         mysqli_errno($dbh),mysqli_sqlstate($dbh),mysqli_error($dbh));
36     $dbh->close( );
37     exit( );
38 }
39 /* free result set */
40 $result_set->close( );
41 $dbh->close( );
42
43 ?>
44 </table>
45 </body>
46 </html>

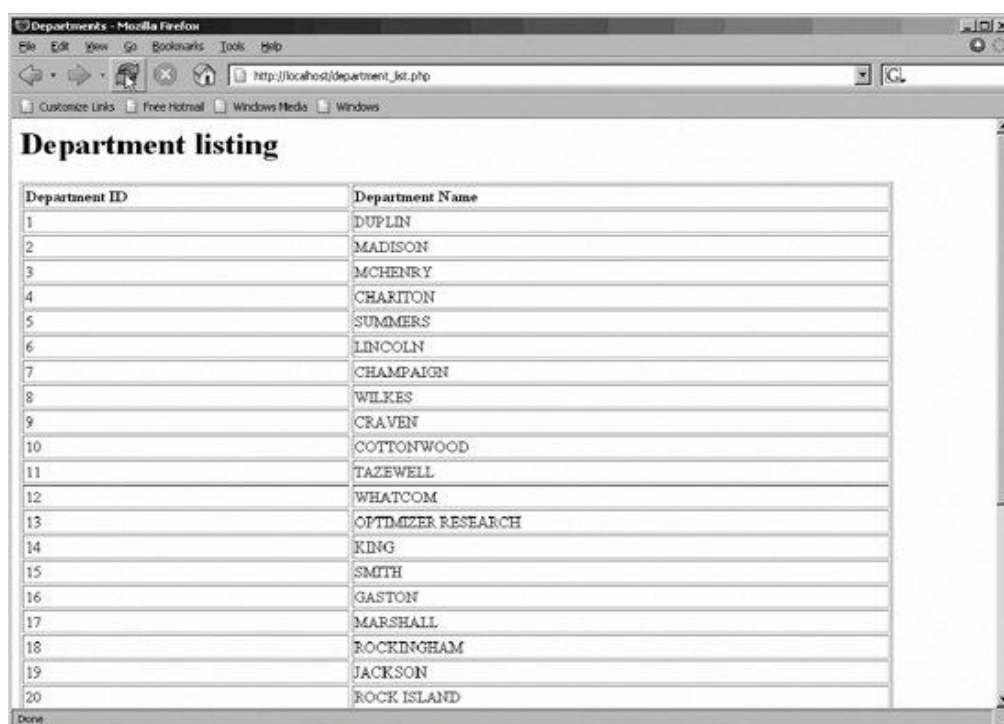
```

Example 5-21 中包含了需要注意的代码行

行号	解释
23	调用 department_list 存储过程，这将返回一个包含部门列表的结果集。对象\$result_set 作为结果集被返回
26	迭代调用 fetch_object 方法，返回一个表示单个记录的对象
28 和 29	使用 department_id 和 department_name 属性作为从\$row 对象中提取列信息的条件，其中包括了相应列中的值

Figure 5-1 中展示了 PHP 程序的输出状况。

Figure 5-1.从存储过程中获取结果集的 PHP 程序输出



Department ID	Department Name
1	DUPLIN
2	MADISON
3	MCHENRY
4	CHARITON
5	SUMMERS
6	LINCOLN
7	CHAMPAIGN
8	WILKES
9	CRAVEN
10	COTTONWOOD
11	TAZEWELL
12	WHATCOM
13	OPTIMIZER RESEARCH
14	KING
15	SMITH
16	GASTON
17	MARSHALL
18	ROCKINGHAM
19	JACKSON
20	ROCK ISLAND

存储程序返回多个结果集的能力是一种庇佑，也是一种诅咒，这完全取决于你看问题的角度。返回多结果集的功能允许你在一个操作中返回多个逻辑相关结果的集合。举例来说，你可以用一个数据库操作完成对所有记录集的多层详情报告填充。这可以极大的层次上分离表现逻辑（经常是 WEB）和数据访问（数据库）逻辑。

当然，对于多结果集的处理要求我们的客户端编程有不一般的要求。一些第三方报告生成工具或许并不支持在一个数据库调用中返回多个结果集。事实上，某些第三方工具甚至完全不能支持在存储过程调用中返回结果集。

幸运的是，我们所使用的主要的编程接口如 PHP, Java, Perl, Python, and .NET C# 和 VB.NET 都支持多结果集的处理。在第十三章到第十七章中，我们将探索如何在那些语言中处理 MySQL 存储程序的结果集和别的操作。为了给你一个处理的大体映像，Example 5-22 给出了如何在 Java 中处理 MySQL 存储过程获得的多结果集。

Example 5-22. 用 Java 获得存储过程的多结果集

```
1 private void empsInDept(Connection myConnect, int deptId) throws SQLException {
2
3     CallableStatement cStmt = myConnect
4         .prepareCall("{CALL sp_emps_in_dept(?)}");
5     cStmt.setInt(1, deptId);
6     cStmt.execute( );
7     ResultSet rs1 = cStmt.getResultSet( );
8     while (rs1.next( )) {
9         System.out.println(rs1.getString("department_name") + " "
10             + rs1.getString("location"));
11     }
12     rs1.close( );
13
14     /* process second result set */
15     if (cStmt.getMoreResults( )) {
16         ResultSet rs2 = cStmt.getResultSet( );
17         while (rs2.next( )) {
18             System.out.println(rs2.getInt(1) + " " + rs2.getString(2) + " "
19                 + rs2.getString(3));
20         }
21         rs2.close( );
22     }
23     cStmt.close( );
24 }
```

让我们看一下 Example 5-22 中比较重要的部分

行号	解释
3	创建一个 CallableStatement 对象，与 Example 5-20 中的存储过程相对应。
5	给存储过程提供一个参数（department_id）。
6	执行存储过程。
7	创建一个 ResultSet 对象与第一个结果集相对应。
8-11	在结果集的记录之间循环并且向控制台输出结果。

15	使用 <code>getMoreResults</code> 方法将指针移向下一个结果集。
16	为第二个结果集创建 <code>ResultSet</code> 对象
17-20	获取结果集中的记录并向控制台输出结果

5.4.2.向另一个存储过程返回结果集

我们知道可以将结果集返回给调用程序（比如 PHP）但是该怎样将结果返回给另一个存储过程呢？

不幸的是，要将结果集从一个存储过程传递给另一个的唯一方法就是将其传递给一个临时表。这是一种笨拙的方法，并且因为临时表的作用域贯穿了整个会话，所以出于很多相同的易维护性方面的考虑，它使用了全局变量。但是如果一个存储程序需要向另一个提供结果，那么临时表可能是最好的解决方案。

让我们看一下例子。在 **Example 5-23** 中，我们有一个专门负责船检包含过期销售的临时表的存储过程。虽然这段 SQL 代码足够简洁，以至于我们可以在任何一个需要处理过期订单的存储程序中重写它，这样做使得我们只需要在一堆组建的一个模块中将此表创建一次，这种做法极大的改善了程序的模块性和易维护性。

Example 5-23.用于创建临时表的存储过程

```
CREATE PROCEDURE sp_overdue_sales ( )

BEGIN

    DROP TEMPORARY TABLE IF EXISTS overdue_sales_tmp;
    CREATE TEMPORARY TABLE overdue_sales_tmp AS
    SELECT sales_id,customer_id,sale_date,quantity,sale_value
        FROM sales
        WHERE sale_status='O';

END;
```

在 **Example 5-24** 中我们看到一个存储过程调用了前面的那个存储过程并消费（使用）了位于临时表中的记录。在实践中，这与将结果集从一个存储过程传递到另一个几乎完美的等同

Example 5-24.消费临时表中数据的存储过程

```
CREATE PROCEDURE sp_issue_invoices( )

BEGIN

    DECLARE l_sale_id INT;
    DECLARE l_last_sale INT DEFAULT 0;

    DECLARE sale_csr CURSOR FOR
        SELECT sales_id
            FROM overdue_sales_tmp;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_sale=1;

    CALL sp_overdue_sales( );
```



```
OPEN sale_csr;
sale_loop:LOOP
    FETCH sale_csr INTO l_sale_id;
    IF l_last_sale THEN
        LEAVE sale_loop;
    END IF;
    CALL sp_issue_one_invoice(l_sale_id);
END LOOP sale_loop;
CLOSE sale_csr;

END;
```

注意，在 **MySQL** 中，临时表的作用域仅与创建该表的会话具有相同的作用域，并且它会在会话结束时自动被清楚，所以你完全不必担心清理临时表或者考虑这个表会被另一个会话所更新（修改）。

5.5.使用预处理语句执行动态 SQL

MySQL 支持一项功能名为服务器端预处理语句，被用于提供独立于 API 的，在反复执行过程中具备高效性和安全性的 SQL 语句预处理。从编程的角度看，预处理语句具备很大的优势，因为它允许你创建动态 SQL 调用。

我们使用 **PREPARE** 语句来创建预处理语句：

```
PREPARE statement_name FROM sql_text
```

当 SQL 执行前必须在 SQL 文本中包含数据值的占位符。占位符用字符 `?` 表示。

预处理语句使用 **EXECUTE** 语句进行执行：

```
EXECUTE statement_name [USING variable [,variable...]]
```

USING 子句可以为 **PREPARE** 语句中的占位符提供指定的值。这些值必须是用户变量（以`@`为前缀字符），这在第三章已经提到过。

最后，我们可以用 **DEALLOCATE** 语句撤销预处理语句

```
DEALLOCATE PREPARE statement_name
```

Example 5-25 展示了在命令行客户端下使用预处理语句的例子

Example 5-25.使用预处理语句

```
mysql> PREPARE prod_insert_stmt FROM "INSERT INTO product_codes VALUES (?,?) ";
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql>
mysql> SET @code='QB';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @name='MySQL Query Browser';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE prod_insert_stmt USING @code,@name;
Query OK, 1 row affected (0.00 sec)

mysql> SET @code='AD';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @name='MySQL Administrator';
Query OK, 0 rows affected (0.02 sec)

mysql> EXECUTE prod_insert_stmt USING @code,@name;
```

```
Query OK, 1 row affected (0.00 sec)

mysql> DEALLOCATE PREPARE prod_insert_stmt;
Query OK, 0 rows affected (0.00 sec)
```

现在我们注意到预处理语句减少了 SQL 语句中少数数据值改变时重新解析（预处理）的开销，并且因为允许使用 SQL 语句参数，从而防止了 SQL 注入（在第 18 章中有关于 SQL 注入更详尽的介绍）。存储过程并不需要预处理语句，因为在存储过程中 SQL 语句在执行前就已被“预处理”。此外，SQL 注入对于存储函数几乎没有威胁（而如果你使用预处理语句则有这种可能！）。

当然，预处理语句在存储过程中还是炙手可热，因为它允许你在过程中执行动态 SQL（但不支持触发器和函数）。如果你在运行时创建一个 SQL 语句，那么它就是动态的（而在编译时创建的程序单元则是静态 SQL 语句）。你通常会在编译时无法提供完整的信息时需要用到动态 SQL。这种事情在你需要从别的数据源获取用户输入时经常会发生。

Example 5-26 给出了一个使用预处理语句来运行动态 SQL 的例子；这段代码事实上能够执行任何从参数传入的 SQL。

Example 5-26. 使用动态 SQL 的存储过程

```
CREATE PROCEDURE execute_immediate(in_sql VARCHAR(4000))
BEGIN

    SET @tmp_sql=in_sql;
    PREPARE s1 FROM @tmp_sql;
    EXECUTE s1;
    DEALLOCATE PREPARE s1;

END;
```

存储过程中被执行的 SQL 语句事实上扮演者和内嵌于过程中的静态 SQL 语句一样的角色。当然，EXECUTE 语句并不支持 INTO 子句或者在预处理语句中定义游标的可能性。因此，任何返回给调用程序的预处理语句执行结果并不能被存储过程内部捕获。要捕获由动态 SQL 调用返回的结果，必须把结果先存储在一张临时表上，这一点已经在早些章节“Returning Result Sets to Another Stored Procedure”提到过。

你只有在需要时才去使用动态 SQL。它比静态 SQL 更复杂，能力却更弱，但是它允许你实现一些用其他方法不可能完成的任务，创造出更有效，更通用的公用例程。举例来说，Example 5-27 的存储过程可以接受一个表名，列名，WHERE 子句和值；过程使用这些参数构造出一个能更新任何表列值的 UPDATE 语句。

Example 5-27. 可以更新任何表中列值的存储过程

```
CREATE PROCEDURE set_col_value
    (in_table    VARCHAR(128),
    in_column    VARCHAR(128),
    in_new_value VARCHAR(1000),
    in_where     VARCHAR(4000))

BEGIN
    DECLARE l_sql VARCHAR(4000);
    SET l_sql=CONCAT_ws(' ',
        'UPDATE ',in_table,
```

```

        'SET',in_column,'=',in_new_value,
        ' WHERE',in_where);
SET @sql=l_sql;
PREPARE s1 FROM @sql;
EXECUTE s1;
DEALLOCATE PREPARE s1;
END;
```

我们可以称这个为首席雇员（employee ID 1）零支出程序（CEO！这回看你怎么办），它对上面的存储过程进行如下调用：

```
mysql> CALL set_col_value('employees','salary','0','employee_id=1')
```

另一个动态 SQL 的应用是创建 WHERE 子句。我们经常会创建一个可供用户选择搜索条件的用户界面，如果使用传统的 SQL 处理“遗漏”的条件会表的复杂而笨拙，者也会给 MySQL 的优化造成困难。Example 5-28 展示了一个可供用户任意指定客户姓名，联系人姓名或电话号码等搜索条件的搜索过程。

Example 5-28. 使用动态 SQL 的搜索过程

```

CREATE PROCEDURE sp_customer_search
    (in_customer_name VARCHAR(30),
    in_contact_surname VARCHAR(30),
    in_contact_firstname VARCHAR(30),
    in_phoneno VARCHAR(10))

BEGIN
    SELECT *
    FROM customers
    WHERE (customer_name LIKE in_customer_name
        OR in_customer_name IS NULL)
        AND (contact_surname LIKE in_contact_surname
        OR in_contact_surname IS NULL)
        AND (contact_firstname LIKE in_contact_firstname
        OR in_contact_firstname IS NULL)
        AND (phoneno LIKE in_phoneno
        OR in_phoneno IS NULL) ;

END;
```

Example 5-28 中的 SQL 并非十分复杂，但是当候选的搜索列增加时，语句的可维护性就会逐渐减小。甚至对于语句本身，我们也能合乎情理的考虑到也许提供给最终用户的可选搜索条件并未得到正确的优化。因此，我们或许希望建立一个自定义度更高的搜索查询。Example 5-29 为我们展示了一个使用 WHERE 动态匹配用户提供的搜索条件并且调用使用预处理语句的动态 SQL 的例子。

Example 5-29. 使用动态 SQL 的搜索过程

```

CREATE PROCEDURE sp_customer_search_dyn
    (in_customer_name VARCHAR(30),
    in_contact_surname VARCHAR(30),
```

```

    in_contact_firstname VARCHAR(30),
    in_phoneno VARCHAR(10))

BEGIN

    DECLARE l_where_clause VARCHAR(1000) DEFAULT 'WHERE';

    IF in_customer_name IS NOT NULL THEN
        SET l_where_clause=CONCAT(l_where_clause,
            ' customer_name='',in_customer_name,'');
    END IF;

    IF in_contact_surname IS NOT NULL THEN
        IF l_where_clause<>'WHERE' THEN
            SET l_where_clause=CONCAT(l_where_clause, ' AND ');
        END IF;
        SET l_where_clause=CONCAT(l_where_clause,
            ' contact_surname='',in_contact_surname,'');
    END IF;

    IF in_contact_firstname IS NOT NULL THEN
        IF l_where_clause<>'WHERE' THEN
            SET l_where_clause=CONCAT(l_where_clause, ' AND ');
        END IF;
        SET l_where_clause=CONCAT(l_where_clause,
            ' contact_firstname='',in_contact_firstname,'');
    END IF;

    IF in_phoneno IS NOT NULL THEN
        IF l_where_clause<>'WHERE' THEN
            SET l_where_clause=CONCAT(l_where_clause, ' AND ');
        END IF;
        SET l_where_clause=CONCAT(l_where_clause,
            ' phoneno='',in_phoneno,'');
    END IF;

    SET @sql=CONCAT('SELECT * FROM customers ',
        l_where_clause);

    PREPARE s1 FROM @sql;
    EXECUTE s1;
    DEALLOCATE PREPARE s1;

END;
```

虽然 Example 5-29 中的存储过程比 Example 5-28 中的静态例子更长，更复杂，但是它可以运行的更快，因为我们在最终被执行的 SQL 中去除了多余的 WHERE 子句。从这方面来说，我们给与了 MySQL 更好的数据来决定怎样去索引

以及其他的各种优化。

你或许不会经常用到动态 SQL 和预处理语句，但是当你创建基于用户输入或存储程序参数的 SQL 语句时，它确实可以为你节省大量的时间。当然，最后的忠告：当你创建基于用户输入的 SQL 时，你就增大了像 SQL 注入这种安全攻击的可能性，而由于存储过程使用独一无二的执行上下文，因此，注入式攻击事实上给存储过程的风险提出了一个大问题。我们暂且啊第十八章重点讨论 SQL 注入攻击。

5.6.SQL 错误处理：预览

MySQL 存储程序的错误处理是如此的重要，也是如此复杂，因此我们决定把该部分内容放到第六章中讨论。但是，我们在此可以提供快速的概览。

通常，如果一个存储程序中的 SQL 语句发生了错误，那么存储程序会停止运行并把错误返回给它的调用程序。如果你不希望发生这种事情，你必须用如下语法指定一个错误处理单元：

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
    {SQLSTATE sqlstate_code| MySQL error code| condition_name}
    stored_program_statement
```

处理单元使用 MySQL 错误代码，ANSI 标准的 SQLSTATE 或者一个命名条件作为错误条件，其中的命名条件描述了遭遇到的错误。处理单元可以做以下事情中的一件：

- 允许继续执行
- 是处理单元的当前块所在的存储程序终止

处理单元将在所指定的存储程序语句执行时被激活。这样的语句经常会设定一些状态变量，以方便程序主干检测到，但也可以被指定为包含若干条枝干的 BEGIN-END 块。

我们已经看过如何在游标返回结果集的最后一行是使用错误处理单元（相见前些章节中的“获取结果集”）。

我们将在下一章中深入讨论处理单元。

5.7.第五章结语

在这一章中我们回顾了 MySQL 提供的在存储程序中包含 SQL 的功能。下面的几种 SQL 语句可以出现在存储程序中：

- 简单内嵌非 SELECT 语句，包括 DML 语句（INSERT, DELETE, UPDATE）和 DDL 语句（CREATE, DROP, ALTER 等）可以没有任何限制的被包含在存储程序中。
- SELECT 语句的返回结果可以传递给存储程序的 INTO 子句中指定的变量
- SELECT 语句允许你使用游标来迭代由其返回的多个结果。游标具有更强的编程作用，包括循环结构和当所有记录获取完毕时使用条件处理来防止 “no data to fetch” 错误。不管怎样，游标可能是你在存储程序中处理复杂数据的有力武器
- 那些不包含 INTO 子句和游标的“非受限”SELECT 语句可以包含在你的存储过程中(但不是存储函数。)SELECT 语句的输出结果将返回给调用程序（但不能返回给调用它的存储过程。）你必须在你的主叫程序中使用一些特殊的代码来处理从其他存储过程中返回的结果集，特表示当你返回的记录不止一个时。

SQL 语句还可以使用 MySQL 服务器端预处理语句进行预处理。

如果你的 SQL 语句产生了一个错误，那么存储程序会终止运行并把控制权返回给主叫程序，除非你为错误顶一个了一个处理单元来“捕获”错误或类似的行为。我们在本章中介绍了如何在访问游标的最后一条记录是处理 NOT FOUND 错误，在下一章中我们将更详细的覆盖错误处理的具体内容。