目　录

第 1 部分　基础篇

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

第1章　MySQL 的安装与配置

近几年，开源数据库逐渐流行起来。由于具有免费使用、配置简单、稳定性好、性能优良等优点，开源数据库在中低端应用上占据了很大的市场份额，而 MySQL 正是开源数据库中的杰出代表。

MySQL 数据库隶属于 MySQL AB 公司，总部位于瑞典。公司名中的“AB”是瑞典语“aktiebolag”或“股份公司”的首字母缩写。MySQL 支持几乎所有的操作系统，并且支持很大的表（MyISAM 存储引擎支持的最大表尺寸为 65536TB），这些特性使得 MySQL 的发展非常迅猛，目前已经广泛应用在各个行业中。

1.1 MySQL 的下载

用户通常可以到官方网站 www.mysql.com 下载最新版本的 MySQL 数据库。按照用户群分类，MySQL 数据库目前分为社区版（Community Server）和企业版（Enterprise），它们最重要的区别在于：社区版是自由下载而且完全免费的，但是官方不提供任何技术支持，适用于大多数普通用户；而企业版则是收费的，不能在线下载，相应地，它提供了更多的功能和更完备的技术支持，更适合于对数据库的功能和可靠性要求较高的企业客户。

MySQL 的版本更新很快，目前可以下载的版本包括 4.1、5.0、5.1 和 6.0。其中 4.1 和 5.0 是发行版，5.1 和 6.0 都还是测试版，这些不同版本之间的主要区别如表 1-1 所示。

表 1-1　　MySQL 不同版本之间的重要改进

版本	重要改进
4.1	增加了子查询的支持；字符集中增加了对 UTF8 的支持
5.0	增加了视图、过程、触发器的支持，增加了 INFORMATION_SCHEMA 系统数据库
5.1（Beta）	增加了表分区的支持
6.0（Alpha）	FALCON 存储引擎的支持

在每个版本里面，还分别有 3 种类型。

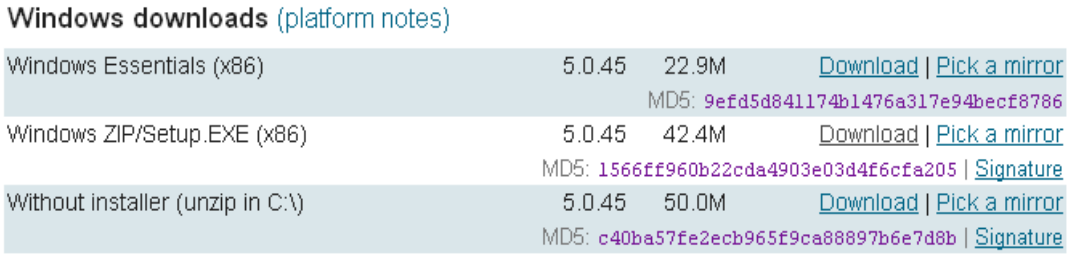
- Standard：推荐大多数用户下载。
- Max：除 Standard 的所有内容外，还有一些附加的新特性，这些特性还没有通过正式的测试发布，主要用于提升用户的认识和体验。
- Debug：和 Standard 类似，但是包括了一些调试信息，会影响系统性能，所以不推荐用户下载。

对于不同的操作系统平台，MySQL 提供了相应的版本，本章将以 Windows 平台下的 nointall 包和图形化安装包以及 Linux 平台下的 RPM 包为例，来说明 MySQL 的下载、安装、配置、启动和关闭过程。本章的测试环境分别是 32 位的 Windows XP 和 x86 平台上的 RedHat Linux AS3。

1.1.1 1.1.1　在 Windows 平台下下载 MySQL

打开浏览器，在地址栏中输入 http://dev.mysql.com/downloads/mysql/5.0.html#linux，打开 MySQL 下载页面，单击“Windows downloads”下“Without installer（unzip in C:\）”后面

的“Download”或者“Pick a mirror”链接来选择一个镜像站点进行下载，如图 1-1 所示。



Windows downloads (platform notes)

Windows Essentials (x86)	5.0.45	22.9M	Download | Pick a mirror
		MD5: 9efd5d8411174b1476a317e94becf8786	
Windows ZIP/Setup.EXE (x86)	5.0.45	42.4M	Download | Pick a mirror
		MD5: 1566ff960b22cda4903e03d4f6cfa205 | Signature	
Without installer (unzip in C:\)	5.0.45	50.0M	Download | Pick a mirror
		MD5: c40ba57fe2ecb965f9ca88897b6e7d8b | Signature	

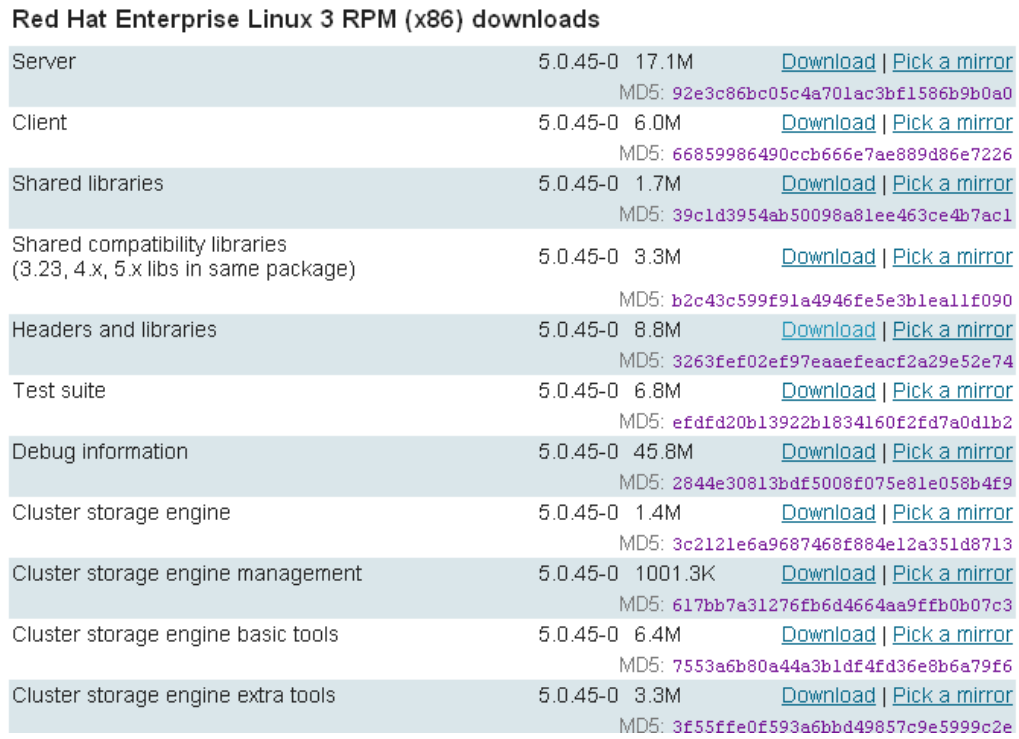
图 1-1　下载 Without installer (unzip in C:\)

1.1.2 1.1.2　在 Linux 平台下下载 MySQL

在 Linux 平台下，要下载 MySQL 可以采用以下两种方法。

- 通过网页直接下载

(1) 打开浏览器，在地址栏中输入 http://dev.mysql.com/downloads/mysql/5.0.html#linux，打开 MySQL 下载页面，单击“Red Hat Enterprise Linux 3 RPM (x86) downloads”下的“Server”和“Client”后面的“Download”或者“Pick a mirror”链接来选择一个镜像站点进行下载，如图 1-2 所示。



Red Hat Enterprise Linux 3 RPM (x86) downloads

Server	5.0.45-0	17.1M	Download | Pick a mirror
		MD5: 92e3c86bc05c4a701ac3bf1586b9b0a0	
Client	5.0.45-0	6.0M	Download | Pick a mirror
		MD5: 66859986490ccb666e7ae889d86e7226	
Shared libraries	5.0.45-0	1.7M	Download | Pick a mirror
		MD5: 39c1d3954ab50098a81ee463ce4b7ac1	
Shared compatibility libraries (3.23, 4.x, 5.x libs in same package)	5.0.45-0	3.3M	Download | Pick a mirror
		MD5: b2c43c599f91a4946fe5e3bleallf090	
Headers and libraries	5.0.45-0	8.8M	Download | Pick a mirror
		MD5: 3263fef02ef97eaaefeacf2a29e52e74	
Test suite	5.0.45-0	6.8M	Download | Pick a mirror
		MD5: efdfd20b13922b1834160f2fd7a0d1b2	
Debug information	5.0.45-0	45.8M	Download | Pick a mirror
		MD5: 2844e30813bdf5008f075e81e058b4f9	
Cluster storage engine	5.0.45-0	1.4M	Download | Pick a mirror
		MD5: 3c2121e6a9687468f884e12a351d8713	
Cluster storage engine management	5.0.45-0	1001.3K	Download | Pick a mirror
		MD5: 617bb7a31276fb6d4664aa9ffb0b07c3	
Cluster storage engine basic tools	5.0.45-0	6.4M	Download | Pick a mirror
		MD5: 7553a6b80a44a3b1df4fd36e8b6a79f6	
Cluster storage engine extra tools	5.0.45-0	3.3M	Download | Pick a mirror
		MD5: 3f55ffe0f593a6bbd49857c9e5999c2e	

图 1-2　下载 Server 和 Client

(2) 将下载后的文件用 FTP 等工具传送到 Linux 服务器上即可。

- 通过命令行方式下载

(1) 首先得到下载地址的 URL（用鼠标右键单击“Download”或者镜像地址的链接，“属性”对话框中显示的“地址”信息即是 URL），如图 1-3 所示。

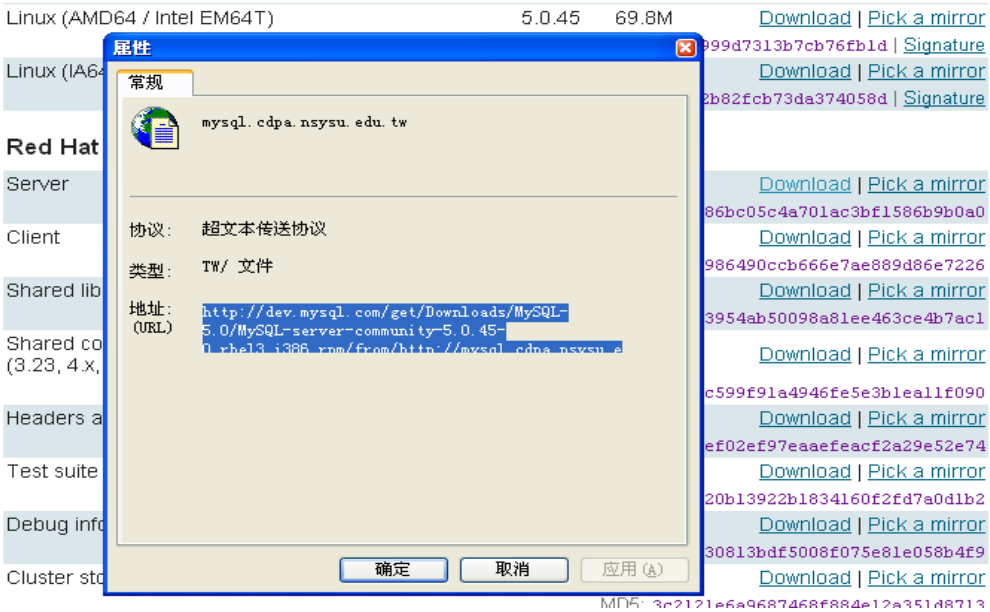


图 1-3　下载地址的 URL

在图 1-3 中，本例显示的 URL 是：

```
http://dev.mysql.com/get/Downloads/MySQL-5.0/mysql-noinstall-5.0.45-win32.zip/from/http://mysql.cdpa.nsysu.edu.tw/
```

（2）然后用 wget 命令在 Linux 服务器上直接下载 Server 和 Client 软件包。

在本例中，下载 Server 软件包的具体命令如下：

```
[zzx@localhost ~]$ wget http://dev.mysql.com/get/Downloads/MySQL-5.0/MySQL-server-community-5.0.45-0.rhel3.i386.rpm/from/http://mysyl.cdpa.nsysu.edu.tw/w/
--10:42:38--  http://dev.mysql.com/get/Downloads/MySQL-5.0/MySQL-server-community-5.0.45-0.rhel3.i386.rpm/from/http://mysql.cdpa.nsysu.edu.tw/
           => `index.html'
Resolving dev.mysql.com... 213.136.52.29
Connecting to dev.mysql.com|213.136.52.29|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: http://mysql.cdpa.nsysu.edu.tw/Downloads/MySQL-5.0/MySQL-server-community-5.0.45-0.rhel3.i386.rpm [following]
--10:42:39--  http://mysql.cdpa.nsysu.edu.tw/Downloads/MySQL-5.0/MySQL-server-community-5.0.45-0.rhel3.i386.rpm
           => `MySQL-server-community-5.0.45-0.rhel3.i386.rpm.1'
Resolving mysql.cdpa.nsysu.edu.tw... 140.110.123.9
Connecting to mysql.cdpa.nsysu.edu.tw|140.110.123.9|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 18,092,605 (17M) [text/plain]

100%[==========================================================================]
```

```
======>] 18,092,605      2.00M/s    ETA 00:00

    10:42:57   (1.11   MB/s)   -   `MySQL-server-community-5.0.45-0.rhel3.i386.rpm.1'   saved
[18092605/18092605]wget
http://dev.mysql.com/get/Downloads/MySQL-5.0/MySQL-client-community-5.0.45-0.rhel3.i386.rpm/
from/http://mysql.cdpa.nsysu.edu.tw/
```

在本例中，下载 Client 软件包的具体命令如下：

```
    [zzx@localhost                                    ~]$                                wget
http://dev.mysql.com/get/Downloads/MySQL-5.0/MySQL-client-community-5.0.45-0.rhel3.i386.rpm/
from/http://mysql.cdpa.nsysu.edu.tw/
    --10:47:55--
http://dev.mysql.com/get/Downloads/MySQL-5.0/MySQL-client-community-5.0.45-0.rhel3.i386.rpm/
from/http://mysql.cdpa.nsysu.edu.tw/
               => `index.html'
    Resolving dev.mysql.com... 213.136.52.29
    Connecting to dev.mysql.com|213.136.52.29|:80... connected.
    HTTP request sent, awaiting response... 302 Found
    Location:
http://mysql.cdpa.nsysu.edu.tw/Downloads/MySQL-5.0/MySQL-client-community-5.0.45-0.rhel3.i38
6.rpm [following]
    --10:47:56--
http://mysql.cdpa.nsysu.edu.tw/Downloads/MySQL-5.0/MySQL-client-community-5.0.45-0.rhel3.i38
6.rpm
               => `MySQL-client-community-5.0.45-0.rhel3.i386.rpm.1'
    Resolving mysql.cdpa.nsysu.edu.tw... 140.110.123.9
    Connecting to mysql.cdpa.nsysu.edu.tw|140.110.123.9|:80... connected.
    HTTP request sent, awaiting response... 200 OK
    Length: 6,257,771 (6.0M) [text/plain]

    100%[================================================================================
======>] 6,257,771      1.14M/s    ETA 00:00

    10:48:05   (838.78   KB/s)   -   `MySQL-client-community-5.0.45-0.rhel3.i386.rpm.1'   saved
[6257771/6257771]
```

1.2 MySQL 的安装

MySQL 的安装分很多种不同情况。下面将以 Windows 平台和 Linux 平台为例，介绍 MySQL 在不同操作系统平台上的安装方法。

1.1.3 1.2.1 在 Windows 平台下安装 MySQL

Window 平台下的安装包主要有两种，一种是 noinstall 包，顾名思义，不需要安装就可以直接使用；另一种是.zip 安装包，可以通过图形化界面进行安装。下面分别就两种安装方式进

行介绍。

1. noinstall 安装

在 Windows 平台下安装 MySQL，其安装步骤非常简单：将下载的文件 mysql-noinstall-5.0.45-win32.zip 放到自定义安装目录下，再用 WinRAR 等压缩工具解压即可。本例中解压到 C:\目录下，如图 1-4 所示。

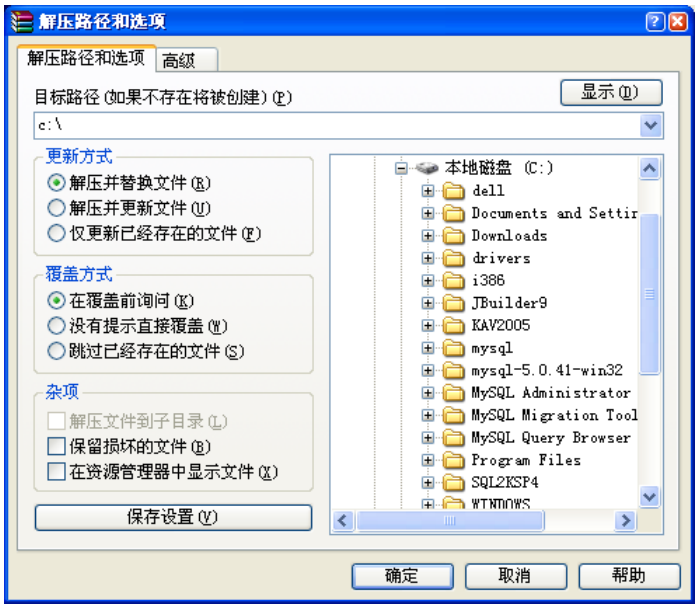


图 1-4　用 WinRAR 解压 noinstall 包到 c:\下

2. 图形化方式安装

在 Windows 平台下，采用图形化方式安装的操作步骤如下。

（1）将压缩文件 mysql-5.0.45-win32.zip 解压到自定义的一个目录下，在本例中解压到 c:\。

（2）双击位于 c:\下的 setup.exe 文件，进入 MySQL 欢迎安装界面，如图 1-5 所示。

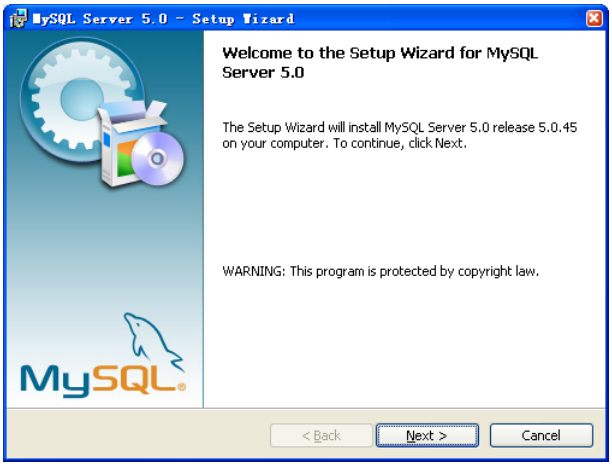


图 1-5　MySQL 安装欢迎界面

（3）单击“Next”按钮，进入“Setup Type”界面，选择 MySQL 安装类型，如图 1-6 所示。

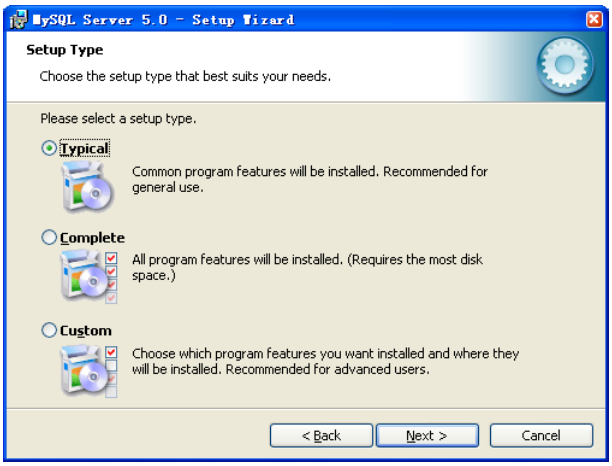


图 1-6　选择 MySQL 安装类型

这 3 种安装类型分别对应着不同的安装组件，其含义如下。

- Typical 表示一般常用的组件都会被安装，默认情况下安装到 c:\Program Files\MySQL\MySQL Server5.0 下，建议大多数情况下选择此安装套件。
- Complete 表示会安装所有的组件，此套件会占用较大的磁盘空间，一般情况下不要选。
- Custom 表示根据用户可以选择要安装的组件、可以更改默认的安装路径，这种安装类型最灵活，适用于高级用户。

在这里选择 Typical 类型。

（4）单击“Next”按钮，进入“Ready to Install Program”界面，如图 1-7 所示。

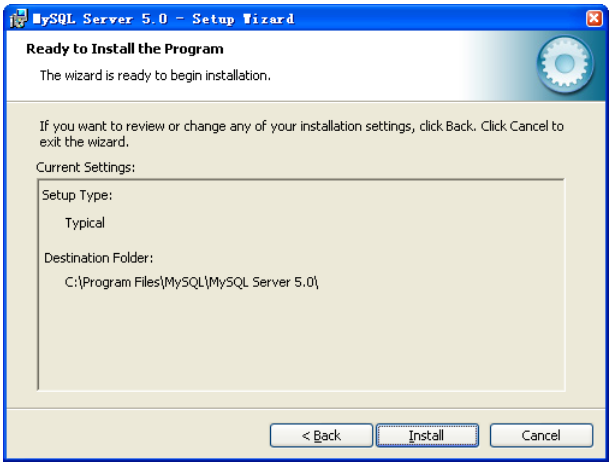


图 1-7　MySQL 安装前的确认界面

此界面进行了安装前的提示，确认安装类型和安装路径。如果想修改，可以单击“Back”按钮返回修改。

（5）单击“Install”按钮，开始安装过程，如图 1-8 所示。安装完毕前，系统会显示 MySQL Enterprise 版（企业版）的一些功能介绍界面，如图 1-9 所示，可以单击“Next”按钮继续看完，也可以单击右上角关闭按钮跳过。

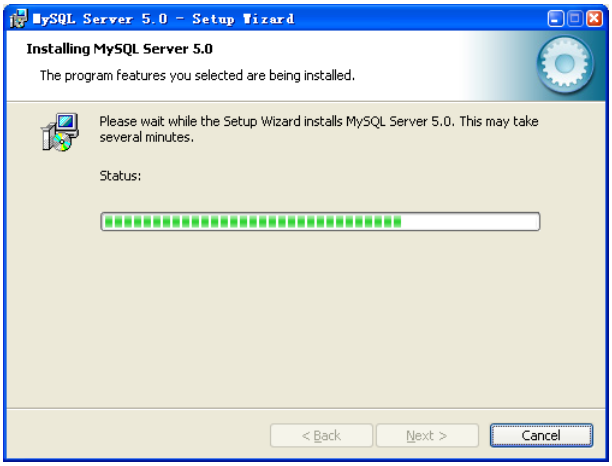


图 1-8　MySQL 安装进度

图 1-9　MySQL Enterprise 版的功能介绍

（6）最后，系统显示安装完毕，如图 1-10 所示。

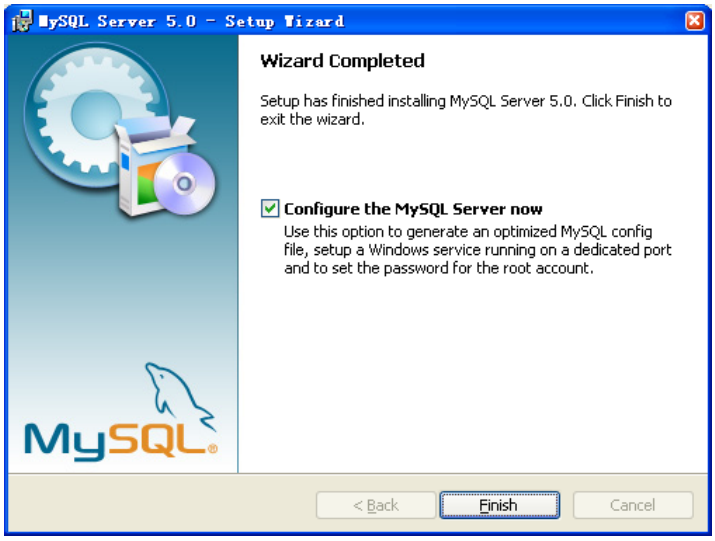


图 1-10　MySQL 安装完成

（7）单击“Finish”按钮完成安装过程，如果想马上配置数据库连接，则选中“Configure the MySQL Server now”复选框。如果想以后再配置，则取消复选框的选中状态。这里

取消选中状态，后面再进行配置的介绍。至此，MySQL 安装完毕，Windows 的“所有程序”菜单中已经多了“MySQL”一项，如图 1-11 所示。

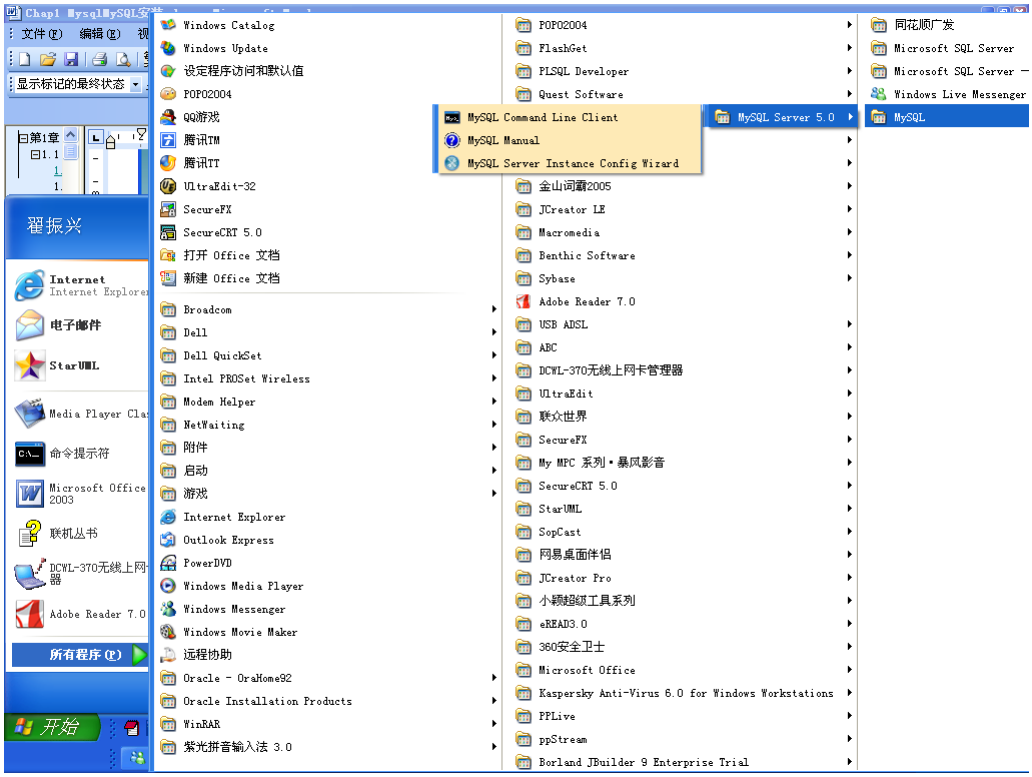


图 1-11　MySQL 安装后的程序项

1.2.2　在 Linux 平台下安装 MySQL

在 Linux 平台下安装和 Windows 平台有所不同，不能用图形化的方式来安装，并且在 Linux 下支持 3 种安装方式：RPM 包、二进制包、源码包。下面以 RPM 包为例来介绍如何在 Linux 平台下进行 MySQL 的安装，其他安装方式还会在本书第 4 篇的第 24 章中进行详细介绍。

RPM 是 Redhat Package Manage 的缩写，透过 RPM 的管理，使用者可以把 Source Code 包装成一种 Source 和 Binary 的档案形式，更加便于安装。MySQL 的 RPM 包包括很多套件，一般只安装 Server 和 Client 就可以了。其中 Server 包是 MySQL 服务端套件，为用户提供核心的 MySQL 服务；Client 包是连接 MySQL 服务的客户端工具，方便管理员和开发人员在服务器上进行各种管理工作。

安装 RPM 包的具体操作步骤如下。

（1）切换到 root 下（只有 root 才可以执行 RPM 包）：

```
[zzx@bj52 zzx]$ su
Password:
[root@bj52 zzx]#
```

（2）安装 MySQL Server 包：

```
[root@localhost zzx]# rpm -ivh MySQL-server-community-5.0.45-0.rhel3.i386.rpm
warning: MySQL-server-community-5.0.45-0.rhel3.i386.rpm: V3 DSA signature: NOKEY, key ID
5072e1f5
Preparing...                ########################################### [100%]
   1:MySQL-server-community ########################################### [100%]
PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
```

```
To do so, start the server, then issue the following commands:
/usr/bin/mysqladmin -u root password 'new-password'
/usr/bin/mysqladmin -u root -h localhost.localdomain password 'new-password'
See the manual for more instructions.
Please report any problems with the /usr/bin/mysqlbug script!

The latest information about MySQL is available on the web at
http://www.mysql.com
Support MySQL by buying support/licenses at http://shop.mysql.com
Starting MySQL[  OK  ]
```

（3）安装 MySQL client 包：

```
[root@localhost zzx]# rpm -ivh MySQL-client-community-5.0.45-0.rhel3.i386.rpm
warning: MySQL-client-community-5.0.45-0.rhel3.i386.rpm: V3 DSA signature: NOKEY, key ID
5072e1f5
Preparing...                ########################################### [100%]
   1:MySQL-client-community ########################################### [100%]
```

（4） 最后运行 MySQL：

```
[root@localhost zzx]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.45-community MySQL Community Edition (GPL)

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

mysql>
```

至此，MySQL 安装完毕。

注意：在 Server 安装过程中有时候会提示缺少 perl-DBI-1.40-8.i386.rpm，这时就需要先下载一个进行安装包，下载地址为 ftp://ftp.chg.ru/pub/Linux/scientific/43/i386/SL/RPMS/perl-DBI-1.40-8.i386.rpm。

1.3 MySQL 的配置

MySQL 安装完毕后，大多数情况下都可以直接启动 MySQL 服务，而不需要设置参数。因为系统对所有的参数都有一个默认值。如果要修改默认值，则必须要配置参数文件。下面就 Windows 和 Linux 两种平台下的配置方法进行介绍。

1.2.3 Windows 平台下配置 MySQL

对于 noinstall 方式安装的 MySQL，系统的参数配置、服务的启动关闭都需要手工在命令窗口中进行设置。参数文件可以在多个位置进行设置，这里用一个 c:\my.cnf 来进行操作，其他更详细的参数位置可以参考第 24 章中的参数设置方法。

对于初学者来说，my.cnf 并不知道该怎样配置。MySQL 为用户提供了几个样例文件，位于解压后的目录下，文件名类似于 my-***.ini，其中“***”分别代表了不同的环境特点，例如 my-small.ini、my-large.ini 分别代表了此文件适合于小型数据库和大型数据库，下面是一个

my-small.ini 的部分内容：

```
# Example MySQL config file for small systems.
#
# This is for a system with little memory (<= 64M) where MySQL is only used
# from time to time and it's important that the mysqld daemon
# doesn't use much resources.
……

# The following options will be passed to all MySQL clients
[client]
#password= your_password
port      = 3306
socket        = /tmp/mysql.sock

# Here follows entries for some specific programs

# The MySQL server
[mysqld]
port      = 3306
socket        = /tmp/mysql.sock
skip-locking
key_buffer = 16K
max_allowed_packet = 1M
table_cache = 4
sort_buffer_size = 64K
read_buffer_size = 256K
read_rnd_buffer_size = 256K
net_buffer_length = 2K
thread_stack = 64K

[mysqldump]
quick
max_allowed_packet = 16M

[mysql]
no-auto-rehash

[isamchk]
key_buffer = 8M
sort_buffer_size = 8M

[myisamchk]
key_buffer = 8M
sort_buffer_size = 8M
```

```
[mysqlhotcopy]
interactive-timeout
```

上面样例中的粗体代表了不同方面的参数，通常我们配置最多的是[mysqld]，也就是 mysql 服务器参数。如果将选择的 my-***.ini 文件另存为 c:\my.cnf 文件，则 MySQL 实例启动的时候就会读取此配置文件。

对于图形化的安装方式，MySQL 提供了一个图形化的实例配置向导，可以引导用户逐步进行实例参数的设置，具体操作步骤如下。

（1）单击“开始”➔“所有程序”➔“MySQL”➔“MySQL Server 5.0”➔“MySQL Server Instance Configuration Wizard”菜单，进入欢迎界面，如图 1-12 所示。

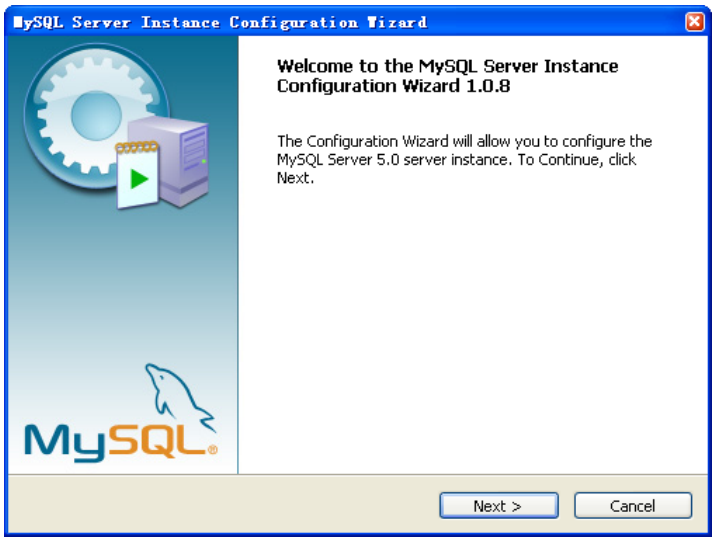


图 1-12　MySQL 实例配置欢迎界面

（2）单击“Next”按钮，进入选择配置类型界面，如图 1-13 所示。

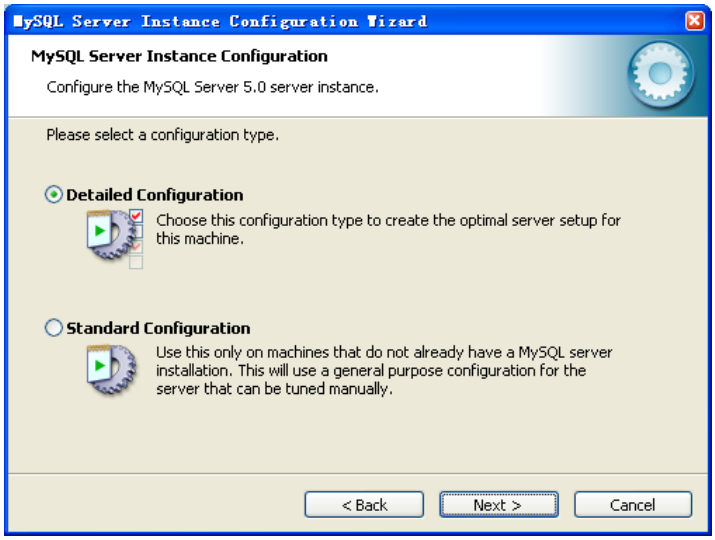


图 1-13　MySQL 实例配置类型选择界面

界面中显示了 MySQL 提供的两种配置类型，详细配置（Detailed Configuration）和标准配置（Standard Configuration），它们的区别在于详细配置列出了更详细的个性化配置向导，配置过程相对复杂而且较慢；而标准配置则是一个通用的配置，配置过程简单快速。这里用详细配置为例来进行介绍。

（3）这里选择“Detailed Configuration”单选按钮，单击“Next”按钮，进入如图 1-14 所示

的界面。

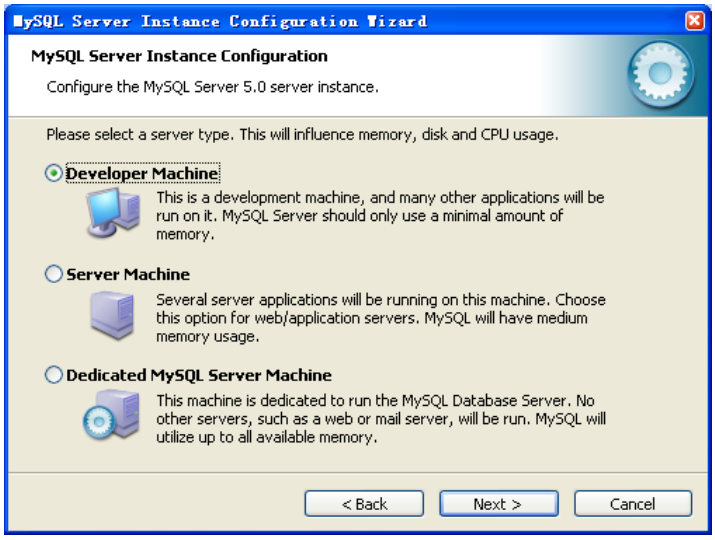


图 1-14　MySQL 应用类型选择界面

此界面中列出了 MySQL 的 3 种应用方式：

- Developer Machine（开发机），使用最小数量的内存；
- Server Machine（服务器），使用中等大小的内存；
- Dedicated MySQL Server Machine（专用服务器），使用当前可用的最大内存。

（4）这里选择“Developer Machine”单选按钮，单击“Next”按钮，进入数据库用途选择界面，如图 1-15 所示。

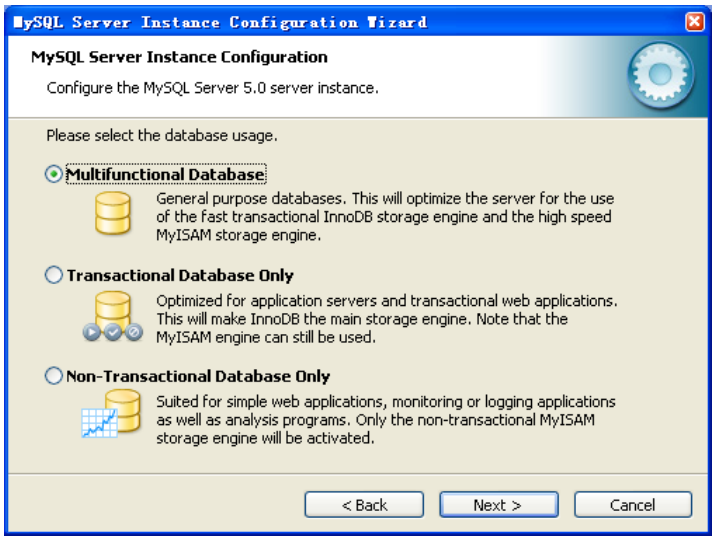


图 1-15　MySQL 数据库用途选择界面

该界面中列出了 3 种数据库用途选项。

- Multifunctional Database（多功能数据库），此选项对事务性（InnoDB）和非事务性（MyISAM）存储引擎的存取速度都很快。
- Transactional Database Only（事务性数据库），此选项主要优化了事务性（InnoDB）存储引擎，但是非事务性存储引擎（MyISAM）也可以使用。
- Non-Transactional Database Only（非事务型数据库），此选项主要优化了非事务性（MyISAM）存储引擎，注意事务性存储引擎（InnoDB）不能使用。

关于存储引擎的介绍，在后面的章节中将会专门进行介绍，读者可以理解为不同的表类型。

（5）这里选择“Multifunctional Database”单选按钮，单击“Next”按钮，进入 InnoDB 数

据文件目录配置界面，如图 1-16 所示。

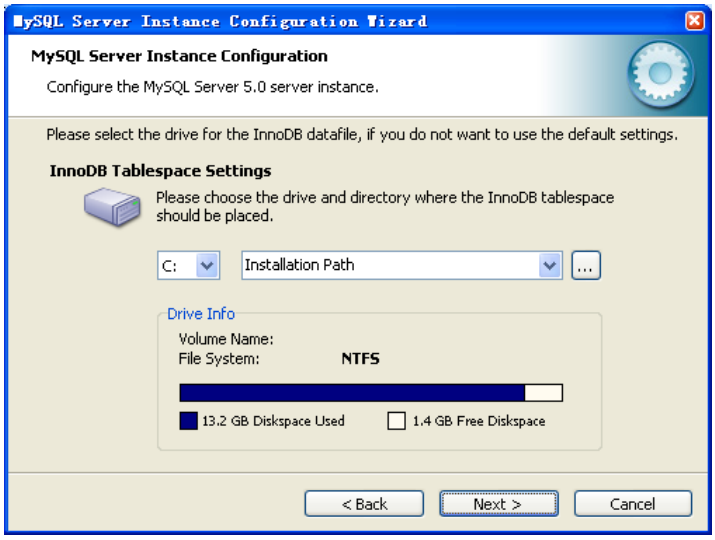


图 1-16　MySQL InnoDB 数据文件路径选择界面

InnoDB 的数据文件会在数据库第一次启动的时候进行创建，默认会创建在 MySQL 的安装目录下。用户可以根据实际的空间状况进行路径的选择，这里保留默认值。

（6）单击“Next”按钮，进入并发连接设置界面，如图 1-17 所示。

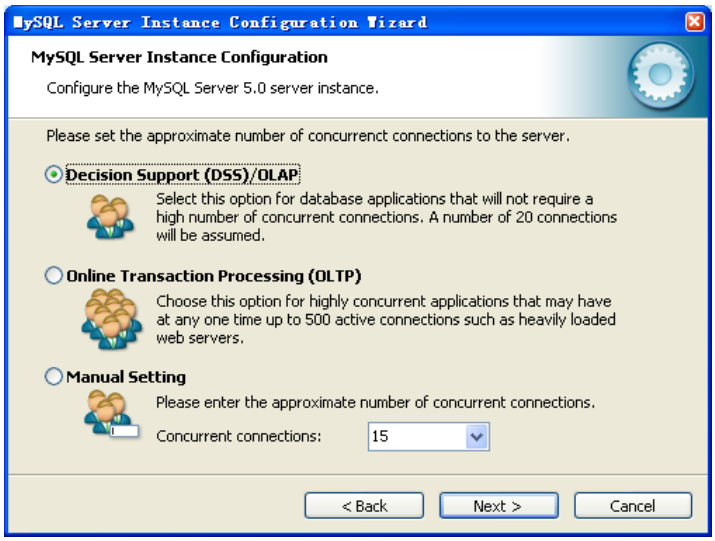


图 1-17　MySQL 并发连接设置界面

其中有 3 个选项，其含义分别如下：

- Decision Support（DSS）/OLAP（决策支持系统），设置连接数为 20。
- Online Transaction Processing（OLTP）（在线事务系统），设置连接数为 500。
- Manual Setting（手工设置），手工输入并发连接数。

（7）选择“Decision Support（DSS）/OLAP”单选按钮后，单击“Next”按钮，进入网络选项设置，如果 1-18 所示。

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

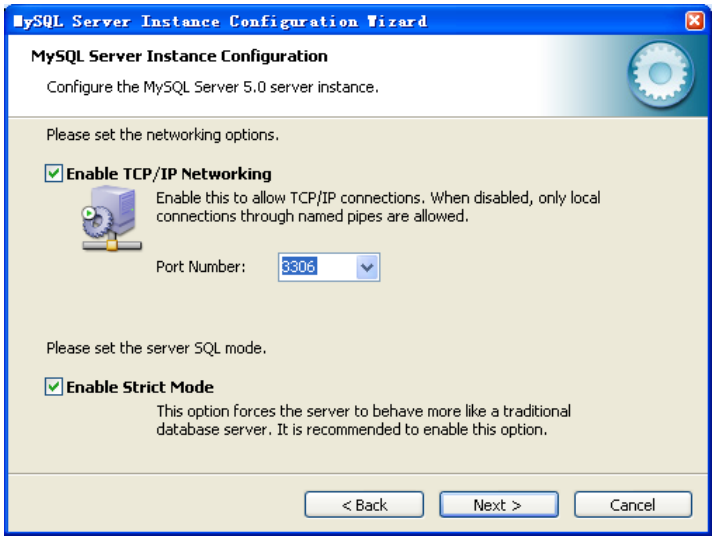


图 1-18　MySQL 并发连接设置界面

本界面中主要设置 MySQL 服务的端口号，选项“Enable TCP/IP Networking”表示是否运行 TCP/IP 连接，而选项“Enable Strict Mode”表示是否采用严格模式来启动服务，至于什么是 MySQL 的模式，将在本书的第 16 章中详细介绍。

（8）选择“Enable TCP/IP Networking”和“Enable Strict Mode”复选框后，单击“Next”按钮，进入默认字符集选择界面，如图 1-19 所示。

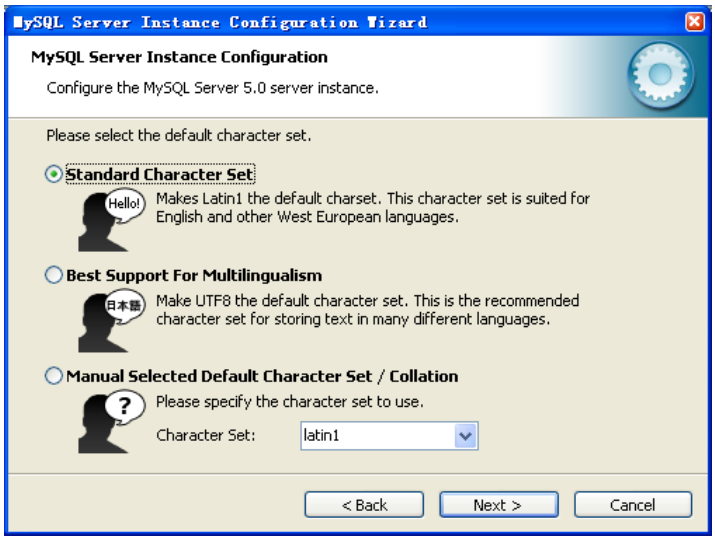


图 1-19　MySQL 实例默认字符集选择界面

该界面上的 3 种选项分别表示如下。

- Standard Character Set（标准字符集），默认是 Latin1。
- Best Support For Multilingualism（对多语言支持最好的字符集），是指 UTF8。
- Manual Selected Default Character Set/Collation（手工选择字符集）。

关于字符集，将会在第 9 章中详细介绍。

（9）选择“Standard Character Set”单选按钮后，单击“Next”按钮，进入 Windows 选项设置界面，如图 1-20 所示。

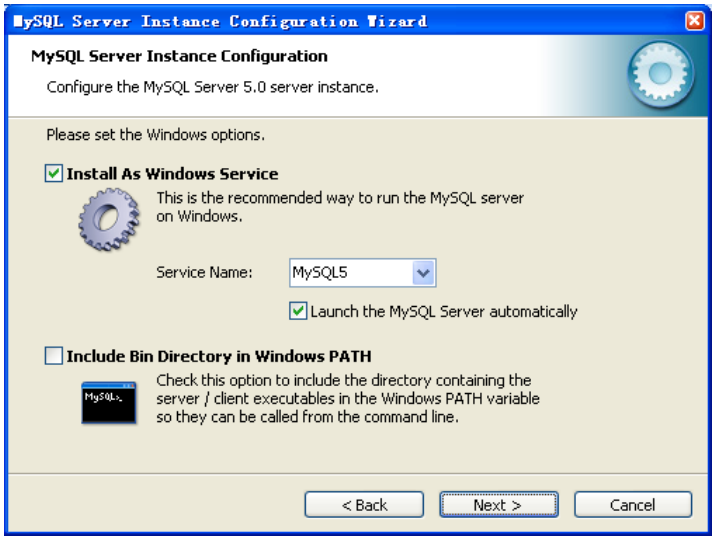


图 1-20　MySQL 相关的 Windows 选项设置

这个界面上部是设置 MySQL 是否作为 Windows 的一个服务，如果是，设置一个服务名称并设置是否 Windows 重启的时候自动装载。这里保留默认选项，将服务名称改为“MySQL5”。下面的“Include Bin Directory in Windows PATH”复选框表示 MySQL 的 Bin 目录是否写入 Windows 的 PATH 中，这里选中该复选框。

（10）单击“Next”按钮，进入 MySQL 的安全选项配置界面，如图 1-21 所示。

图 1-21　MySQL 的安全设置

图 1-21 中显示了 MySQL 的两个安全设置复选框，“Modify Security Settings”复选框确定是否修改默认 root 密码，因为默认的 root 密码是空，因此建议用户一定要修改；“Create An Anonymous Account”复选框确定是否创建一个匿名用户，建议用户不要创建，因为这样会给系统带来安全漏洞。这里为了简便起见，将 root 口令改为 123（正式的生产环境中一定要采用更为复杂的密码）。

（11）单击“Next”按钮，进入准备执行界面，如图 1-22 所示。

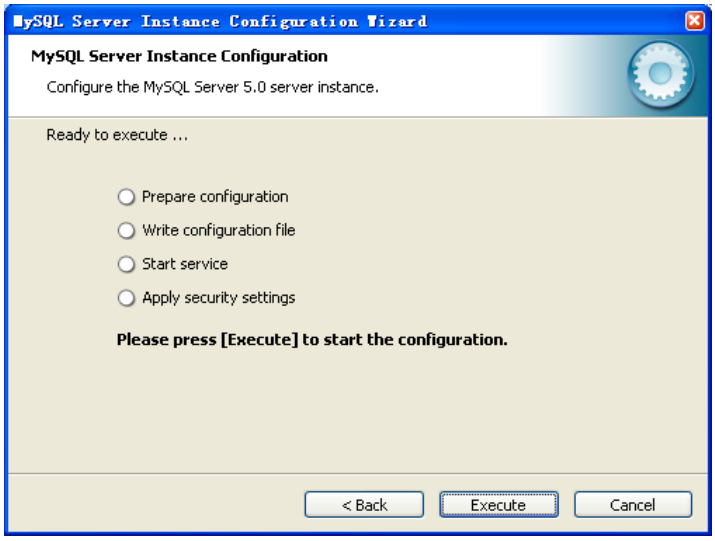


图 1-22　准备执行配置界面

（12）确认设置没有问题后，单击“Execute”按钮，开始执行。执行成功后的界面如图 1-23 所示。

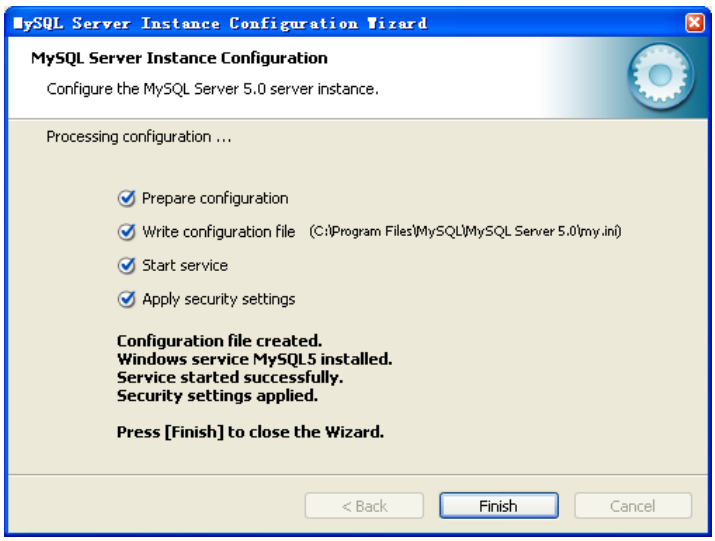


图 1-23　MySQL 配置执行成功界面

单击“Finish”按钮后，安装过程全部完成。这个时候可以发现，Windows 的服务列表中已经增加了“MySQL5”这一项，如图 1-24 所示。我们可以通过启动停止这个服务来启动和关闭 MySQL。

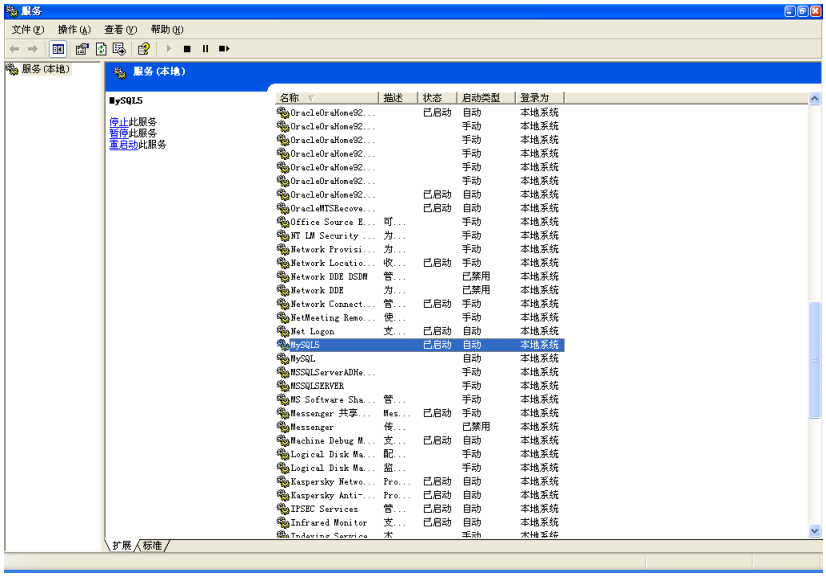


图 1-24　Window 服务列表中的“MySQL5”

1.2.4　Linux 平台下配置 MySQL

在 Linux 下配置 MySQL 和 Windows noinstall 方式配置非常类似，区别在于参数文件的位置和文件名不同。Linux 下也可以在多个位置部署配置文件，我们大多数情况下都放在/etc 下，文件名称只能是 my.cnf（在 Windows 下文件名称可以是 my.ini）。

对于初学者来说，和 Windows 下类似，还是建议用 MySQL 自带的多个样例参数文件来代替实际的参数文件。在 Linux 下，如果安装方式是 RPM 包，则自带的参数文件会放到/usr/share/mysql 下，如下所示：

```
[root@localhost mysql]# pwd
/usr/share/mysql
[root@localhost mysql]# ls *.cnf
my-huge.cnf  my-innodb-heavy-4G.cnf  my-large.cnf  my-medium.cnf  my-small.cnf
```

用户可以根据实际需求选择不同的配置文件 cp 到/etc 下，改名为 my.cnf，并根据实际需要做一些配置的改动。MySQL 启动的时候会读取此文件中的配置选项。

1.4 启动和关闭 MySQL 服务

安装配置完毕 MySQL 后，接下来就该启动 MySQL 服务了。这里强调一下，MySQL 服务和 MySQL 数据库不同，MySQL 服务是一系列后台进程，而 MySQL 数据库则是一系列的数据目录和数据文件；MySQL 数据库必须在 MySQL 服务启动之后才可以进行访问。下面就针对 Windows 和 Linux 两种平台，介绍一下 MySQL 服务的启动和关闭方法。

1.1.4 1.3.1　在 Windows 平台下启动和关闭 MySQL 服务

对于 noinstall 安装的 MySQL，可以在 DOS 窗口下通过命令行方式启动和关闭 MySQL 服务。

（1）启动服务：

```
cd C:\mysql-5.0.45-win32\bin
```

```
C:\mysql-5.0.45-win32\bin>mysqld --console
070703 17:19:10  InnoDB: Started; log sequence number 0 43655
070703 17:19:10 [Note] mysqld: ready for connections.
Version: '5.0.45-community-nt'  socket: ''  port: 3306  MySQL Community Edition
(GPL)
```

（2）关闭服务：

```
C:\mysql-5.0.45-win32\bin>mysqladmin -uroot shutdown
此时，控制台输出：
070703 17:21:13 [Note] mysqld: Normal shutdown


070703 17:21:13  InnoDB: Starting shutdown...
070703 17:21:16  InnoDB: Shutdown completed; log sequence number 0 43655
070703 17:21:16 [Note] mysqld: Shutdown complete


Error in my_thread_global_end(): 1 threads didn't exit
```

对于采用图形化方式安装的 MySQL，可以直接通过 Windows 的“开始”菜单（单击“开始”➔“控制面板”➔“管理工具”➔“服务”菜单）启动和关闭 MySQL，如图 1-25 所示。

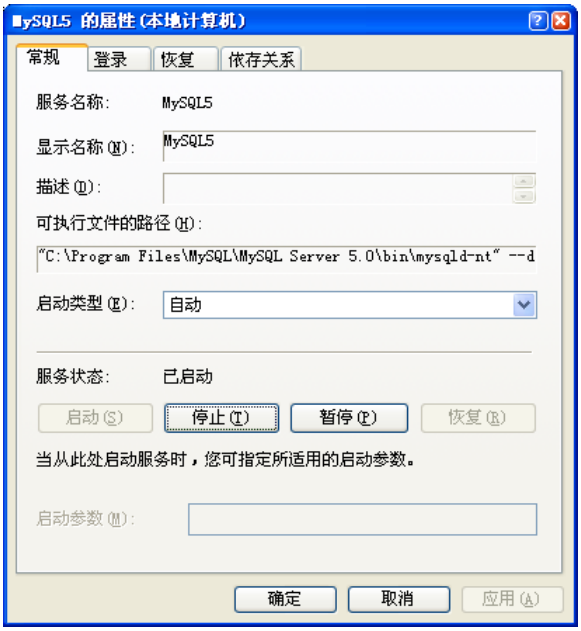


图 1-25　服务列表中启动和关闭 MySQL5

用户也可以在命令行中手工启动和关闭 MySQL 服务，如下所示。

（1）启动服务：

```
C:\Program Files\MySQL\MySQL Server 5.0\bin>net start mysql5
MySQL5 服务正在启动 .
MySQL5 服务已经启动成功。
```

（2）关闭服务：

```
C:\Program Files\MySQL\MySQL Server 5.0\bin>net stop mysql5
MySQL5 服务正在停止.
MySQL5 服务已成功停止。
```

1.1.5 1.3.2 在 Linux 平台下启动和关闭 MySQL 服务

在 Linux 平台下，可以采用如下命令查看 MySQL 服务的状态：

```
[root@localhost bin]# netstat -nlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State
PID/Program name
tcp        0      0 0.0.0.0:3306                0.0.0.0:*                   LISTEN
3168/mysqld
tcp        0      0 :::9922                     :::*                        LISTEN
1864/sshd
Active UNIX domain sockets (only servers)
Proto RefCnt Flags       Type       State         I-Node PID/Program name    Path
unix  2      [ ACC ]     STREAM     LISTENING     16537243 3168/mysqld
/var/lib/mysql/mysql.sock
unix  2      [ ACC ]     STREAM     LISTENING     4875   1915/xfs
/tmp/.font-unix/fs7100
```

其中 3306 端口就是 MySQL 服务器监听端口。

与在 Windows 平台上类似，在 Linux 平台上启动和关闭 MySQL 也有两种方法，一种是通过命令行方式启动和关闭，另外一种是通过服务的方式启动和关闭（适用于 RPM 包安装方式）。下面将分别对这两种方法进行介绍。

在命令行方式下，启动和关闭 MySQL 服务命令如下。

（1）启动服务：

```
[root@localhost bin]# cd /usr/bin
[root@localhost bin]# ./mysqld_safe &
[1] 23013
[root@localhost bin]# Starting mysqld daemon with databases from /var/lib/mysql
```

（2）关闭服务：

```
[root@localhost bin]# mysqladmin -uroot shutdown
STOPPING server from pid file /var/lib/mysql/localhost.localdomain.pid
070820 04:36:30  mysqld ended

[1]+  Done                    ./mysqld_safe
```

如果 MySQL 是用 RPM 包安装的，则启动和关闭 MySQL 服务过程如下。

（1）启动服务：

```
[root@localhost zzx]# service mysql start
Starting MySQL[  OK  ]
```

如果在启动状态，需要重启服务，可以用以下命令直接重启，而不需要先关闭再启动：

```
[root@localhost mysql]# service mysql restart
Shutting down MySQL..[  OK  ]
Starting MySQL[  O K  ]
```

（2）关闭服务：

```
[root@localhost bin]# service mysql stop
```

```
Shutting down MySQL. STOPPING server from pid file /var/lib/mysql/localhost.localdomain.pid
070727 06:30:31  mysqld ended

[  OK  ]
[1]+  Done                    mysqld_safe
```

注意：在命令行启动 MySQL 时候，如果不加“--console”，启动关闭信息将不会在界面中显示，而是记录在安装目录下的 data 目录里面，文件名字一般是 hostname.err，可以通过此文件查看 MySQL 的控制台信息

1.5 小结

本章以 Windows 平台和 Linux 平台为例讲述了 MySQL 在不同操作系统平台上的下载、安装、配置、启动关闭的过程。其中在 Windows 平台下介绍了主要的两种安装包：noinstall 包和图形化安装包；而在 Linux 平台下只介绍了 RPM 包，而没有介绍二进制包和源码包。之所以选择这几种包进行安装，主要是因为它们比较简单，适合初学者快速入门。在第 4 篇的第 24 章中，将会对 Linux 下的二进制包和源码包进行详细的介绍。

第2章　SQL 基础

本章将通过丰富的实例对 SQL 语言的基础进行详细介绍，MySQL，使得读者不但能够学习到标准 SQL 的使用，又能够学习到 MySQL 中一些扩展 SQL 的使用方法。

2.1 SQL 简介

当面对一个陌生的数据库时，通常需要一种方式与它进行交互，以完成用户所需要的各种工作，这个时候，就要用到 SQL 语言了。

SQL 是 Structure Query Language（结构化查询语言）的缩写，它是使用关系模型的数据库应用语言，由 IBM 在 20 世纪 70 年代开发出来，作为 IBM 关系数据库原型 System R 的原型关系语言，实现了关系数据库中的信息检索。

20 世纪 80 年代初，美国国家标准局（ANSI）开始着手制定 SQL 标准，最早的 ANSI 标准于 1986 年完成，就被叫作 SQL-86。标准的出台使 SQL 作为标准关系数据库语言的地位得到了加强。SQL 标准目前已几经修改更趋完善。

正是由于 SQL 语言的标准化，所以大多数关系型数据库系统都支持 SQL 语言，它已经发展成为多种平台进行交互操作的底层会话语言。

2.2 (My)SQL 使用入门

这里用了(My)SQL 这样的标题，目的是在介绍标准 SQL 的同时，也将一些 MySQL 在标准 SQL 上的扩展一同介绍给大家。希望读者看完本节后，能够对标准 SQL 的基本语法和 MySQL 的部分扩展语法有所了解。

2.2.1 SQL 分类

SQL 语句主要可以划分为以下 3 个类别。

- DDL（Data Definition Languages）语句：数据定义语言，这些语句定义了不同的数据段、数据库、表、列、索引等数据库对象的定义。常用的语句关键字主要包括 create、drop、alter 等。
- DML（Data Manipulation Language）语句：数据操纵语句，用于添加、删除、更新和查询数据库记录，并检查数据完整性，常用的语句关键字主要包括 insert、delete、udpate 和 select 等。
- DCL（Data Control Language）语句：数据控制语句，用于控制不同数据段直接的许可和访问级别的语句。这些语句定义了数据库、表、字段、用户的访问权限和安全级别。主要的语句关键字包括 grant、revoke 等。

2.2.2 DDL 语句

DDL 是数据定义语言的缩写，简单来说，就是对数据库内部的对象进行创建、删除、修改的操作语言。它和 DML 语言的最大区别是 DML 只是对表内部数据的操作，而不涉及到表的定义、结构的修改，更不会涉及到其他对象。DDL 语句更多的被数据库管理员（DBA）所使用，一般的开发人员很少使用。

下面通过一些例子来介绍 MySQL 中常用 DDL 语句的使用方法。

1．创建数据库

启动 MySQL 服务之后，输入以下命令连接到 MySQL 服务器：

```
[mysql@db3 ~]$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7344941 to server version: 5.1.9-beta-log

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

mysql>
```

在以上命令行中，mysql 代表客户端命令，-u 后面跟连接的数据库用户，-p 表示需要输入密码。

如果数据库设置正常，并输入正确的密码，将看到上面一段欢迎界面和一个 mysql>提示符。在欢迎界面中介绍了以下几部分内容。

- 命令的结束符，用;或者\g 结束。

- 客户端的连接 ID，这个数字记录了 MySQL 服务到目前为止的连接次数，每个新连接都会自动加 1，本例中是 7344941。
- MySQL 服务器的版本，本例中是“5.1.9-beta-log”，说明是 5.1.9 的测试版，如果是标准版，则会用 Standard 代替 Beta。
- 通过“help;”或者“\h”命令来显示帮助内容，通过“\c”命令来清除命令行 buffer。

在 mysql>提示符后面输入所要执行的的 SQL 语句，每个 SQL 语句以分号或者\g 结束，按回车键执行。

因为所有的数据都存储在数据库中，因此需要学习的第一个命令是创建数据库，语法如下所示：

```
CREATE DATABASE dbname
```

例如，创建数据库 test1，命令执行如下：

```
mysql> create database test1;
Query OK, 1 row affected (0.00 sec)
```

可以发现，执行完创建命令后，下面有一行提示“Query OK, 1 row affected (0.00 sec)”，这段提示可以分为 3 部分，“Query OK”表示上面的命令执行成功，读者可能奇怪，又不是执行查询操作，为什么显示查询成功？其实这是 MySQL 的一个特点，所有的 DDL 和 DML（不包括 SELECT）操作执行成功后都显示“Query OK”，这里理解为执行成功就可以了；“1 row affected”表示操作只影响了数据库中一行的记录，“0.00 sec”则记录了操作执行的时间。

如果已经存在这个数据库，系统会提示：

```
mysql> create database test1;
ERROR 1007 (HY000): Can't create database 'test1'; database exists
```

这个时候，如果需要知道系统中都存在哪些数据库，可以用以下命令来查看：

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| cluster            |
| mysql              |
| test               |
| test1              |
+--------------------+
5 rows in set (0.00 sec)
```

可以发现，在上面的列表中除了刚刚创建的 test1 外，还有另外 4 个数据库，它们都是安装 MySQL 时系统自动创建的，其各自功能如下。

- information_schema：主要存储了系统中的一些数据库对象信息。比如用户表信息、列信息、权限信息、字符集信息、分区信息等。
- cluster：存储了系统的集群信息。
- mysql：存储了系统的用户权限信息。
- test：系统自动创建的测试数据库，任何用户都可以使用。

在查看了系统中已有的数据库后，可以用如下命令选择要操作的数据库：

```
USE dbname
```

例如，选择数据库 test1：

```
mysql> use test1
Database changed
```

然后再用以下命令来查看 test1 数据库中创建的所有数据表：

```
mysql> show tables;
Empty set (0.00 sec)
```

由于 test1 是刚创建的数据库，还没有表，所以显示为空。命令行下面的“Empty set”表示操作的结果集为空。如果查看一下 mysql 数据库里面的表，则可以得到以下信息：

```
mysql> use mysql
Database changed
mysql> show tables;
+---------------------------+
| Tables_in_mysql           |
+---------------------------+
| columns_priv              |
| db                        |
| event                     |
| func                      |
| general_log               |
| help_category             |
| help_keyword              |
| help_relation             |
| help_topic                |
| host                      |
| plugin                    |
| proc                      |
| procs_priv                |
| slow_log                  |
| tables_priv               |
| time_zone                 |
| time_zone_leap_second     |
| time_zone_name            |
| time_zone_transition      |
| time_zone_transition_type |
| user                      |
+---------------------------+
21 rows in set (0.00 sec)
```

2. 删除数据库

删除数据库的语法很简单，如下所示：

```
drop database dbname;
```

例如，要删除 test1 数据库可以使用以下语句：

```
mysql> drop database test1;
```

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

可以发现，提示操作成功后，后面却显示了“0 rows affected”，这个提示可以不用管它，在 MySQL 里面，drop 语句操作的结果显示都是“0 rows affected”。

> 注意：数据库删除后，下面的所有表数据都会全部删除，所以删除前一定要仔细检查并做好相应备份.

3. 创建表

在数据库中创建一张表的基本语法如下：

```
CREATE TABLE  tablename (column_name_1 column_type_1 constraints，
column_name_2  column_type_2  constraints  ，  ……column_name_n  column_type_n
    constraints）
```

因为 MySQL 的表名是以目录的形式存在于磁盘上，所以表名的字符可以用任何目录名允许的字符。column_name 是列的名字，column_type 是列的数据类型，contraints 是这个列的约束条件，在后面的章节中会详细介绍。

例如，创建一个名称为 emp 的表。表中包括 3 个字段，ename（姓名），hiredate（雇用日期）、sal（薪水），字段类型分别为 varchar（10）、date、int（2）（关于字段类型将会在下一章中介绍）：

```
mysql> create table emp(ename varchar(10),hiredate date,sal decimal(10,2),deptno int(2));
Query OK, 0 rows affected (0.02 sec)
```

表创建完毕后，如果需要查看一下表的定义，可以使用如下命令：

```
DESC tablename
```

例如，查看 emp 表，将输出以下信息：

```
mysql> desc emp;
+----------+---------------+------+-----+---------+-------+
| Field    | Type          | Null | Key | Default | Extra |
+----------+---------------+------+-----+---------+-------+
| ename    | varchar(10)   | YES  |     |         |       |
| hiredate | date          | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)        | YES  |     |         |       |
+----------+---------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

虽然 desc 命令可以查看表定义，但是其输出的信息还是不够全面，为了查看更全面的表定义信息，有时就需要通过查看创建表的 SQL 语句来得到，可以使用如下命令实现：

```
mysql> show create table emp \G;
*************************** 1. row ***************************
       Table: emp
Create Table: CREATE TABLE `emp` (
  `ename` varchar(20) DEFAULT NULL,
  `hiredate` date DEFAULT NULL,
  `sal` decimal(10,2) DEFAULT NULL,
  `deptno` int(2) DEFAULT NULL,
```

```
  KEY `idx_emp_ename` (`ename`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk
1 row in set (0.02 sec)

ERROR:
No query specified

mysql>
```

从上面表的创建 SQL 语句中，除了可以看到表定义以外，还可以看到表的 engine（存储引擎）和 charset（字符集）等信息。“\G”选项的含义是使得记录能够按照字段竖着排列，对于内容比较长的记录更易于显示。

4. 删除表

表的删除命令如下：

```
DROP TABLE tablename
```

例如，要删除数据库 emp 可以使用以下命令：

```
mysql> drop table emp;
Query OK, 0 rows affected (0.00 sec)
```

5. 修改表

对于已经创建好的表，尤其是已经有大量数据的表，如果需要对表做一些结构上的改变，我们可以先将表删除（drop），然后再按照新的表定义重建表。这样做没有问题，但是必然要做一些额外的工作，比如数据的重新加载。而且，如果有服务在访问表，也会对服务产生影响。

因此，在大多数情况下，表结构的更改一般都使用 alter table 语句，以下是一些常用的命令。

（1） 修改表类型，语法如下：

```
ALTER TABLE tablename MODIFY [COLUMN] column_definition [FIRST | AFTER col_name]
```

例如，修改表 emp 的 ename 字段定义，将 varchar(10)改为 varchar(20)：

```
mysql> desc emp;
+----------+---------------+------+-----+---------+-------+
| Field    | Type          | Null | Key | Default | Extra |
+----------+---------------+------+-----+---------+-------+
| ename    | varchar(10)   | YES  |     |         |       |
| hiredate | date          | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)        | YES  |     |         |       |
+----------+---------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
mysql> alter table emp modify ename varchar(20);
Query OK, 0 rows affected (0.03 sec)
```

```
Records: 0  Duplicates: 0  Warnings: 0
mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

（2） 增加表字段，语法如下：

```
ALTER TABLE tablename ADD [COLUMN] column_definition [FIRST | AFTER col_name]
```

例如，表 emp 上新增加字段 age，类型为 int(3)：

```
mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> alter table emp add column age int(3);
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
| age      | int(3)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

（3）删除表字段，语法如下：

```
ALTER TABLE tablename DROP [COLUMN] col_name
```

例如，将字段 age 删除掉：

```
mysql> desc emp;
```

```
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
| age      | int(3)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
mysql> alter table emp drop column age;
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

（4）字段改名，语法如下：

```
ALTER TABLE tablename CHANGE [COLUMN] old_col_name column_definition
[FIRST|AFTER col_name]
```

例如，将 age 改名为 age1，同时修改字段类型为 int(4)：

```
mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2) | YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
| age      | int(3)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
mysql> alter table emp change age age1 int(4) ;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc emp
    -> ;
```

```
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2)| YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
| age1     | int(4)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

注意：change 和 modify 都可以修改表的定义，不同的是 change 后面需要写两次列名，不方便。但是 change 的优点是可以修改列名称，modify 则不能。

（5）修改字段排列顺序。

前面介绍的的字段增加和修改语法（ADD/CNAHGE/MODIFY）中，都有一个可选项 first|after column_name，这个选项可以用来修改字段在表中的位置，默认 ADD 增加的新字段是加在表的最后位置，而 CHANGE/MODIFY 默认都不会改变字段的位置。

例如，将新增的字段 birth date 加在 ename 之后：

```
mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2)| YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
| age      | int(3)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

mysql> alter table emp add birth date after ename;
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc emp;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ename    | varchar(20)  | YES  |     |         |       |
| birth    | date         | YES  |     |         |       |
| hiredate | date         | YES  |     |         |       |
| sal      | decimal(10,2)| YES  |     |         |       |
| deptno   | int(2)       | YES  |     |         |       |
| age      | int(3)       | YES  |     |         |       |
+----------+--------------+------+-----+---------+-------+
```

```
6 rows in set (0.00 sec)
```

修改字段 age，将它放在最前面：

```
mysql> alter table emp modify age int(3) first;
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc emp;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| age     | int(3)       | YES  |     |         |       |
| ename   | varchar(20)  | YES  |     |         |       |
| birth   | date         | YES  |     |         |       |
| hiredate| date         | YES  |     |         |       |
| sal     | decimal(10,2)| YES  |     |         |       |
| deptno  | int(2)       | YES  |     |         |       |
+---------+--------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

> 注意：CHANGE/FIRST|AFTER COLUMN 这些关键字都属于 MySQL 在标准 SQL 上的扩展，在其他数据库上不一定适用。

（6）表改名，语法如下：

```
ALTER TABLE    tablename RENAME [TO] new_tablename
```

例如，将表 emp 改名为 emp1，命令如下：

```
mysql> alter table emp rename emp1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc emp;
ERROR 1146 (42S02): Table 'sakila.emp' doesn't exist
mysql> desc emp1;
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| age     | int(3)       | YES  |     |         |       |
| ename   | varchar(20)  | YES  |     |         |       |
| birth   | date         | YES  |     |         |       |
| hiredate| date         | YES  |     |         |       |
| sal     | decimal(10,2)| YES  |     |         |       |
| deptno  | int(2)       | YES  |     |         |       |
+---------+--------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

2.2.3 DML 语句

DML 操作是指对数据库中表记录的操作，主要包括表记录的插入（insert）、更新（update）、

删除（delete）和查询（select），是开发人员日常使用最频繁的操作。下面将依次对它们进行介绍。

1．插入记录

表创建好后，就可以往里插入记录了，插入记录的基本语法如下：

```
INSERT INTO   tablename (field1,field2,……fieldn) VALUES(value1,value2,……valuesn);
```

例如，向表 emp 中插入以下记录：ename 为 zzx1，hiredate 为 2000-01-01，sal 为 2000，deptno 为 1，命令执行如下：

```
mysql> insert into emp (ename,hiredate,sal,deptno) values('zzx1','2000-01-01','2000',1);
Query OK, 1 row affected (0.00 sec)
```

也可以不用指定字段名称，但是 values 后面的顺序应该和字段的排列顺序一致：

```
mysql> insert into emp  values('lisa','2003-02-01','3000',2);
Query OK, 1 row affected (0.00 sec)
```

对于含可空字段、非空但是含有默认值的字段、自增字段，可以不用在 insert 后的字段列表里面出现，values 后面只写对应字段名称的 value，这些没写的字段可以自动设置为 NULL、默认值、自增的下一个数字，这样在某些情况下可以大大缩短 SQL 语句的复杂性。

例如，只对表中的 ename 和 sal 字段显式插入值：

```
mysql> insert into emp  (ename,sal) values('dony',1000);
Query OK, 1 row affected (0.00 sec)
```

来查看一下实际插入值：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 100.00  |      1 |
| lisa   | 2003-02-01 | 400.00  |      2 |
| bjguan | 2004-04-02 | 100.00  |      1 |
| dony   | NULL       | 1000.00 |   NULL |
+--------+------------+---------+--------+
```

果然，设置为可空的两个字段都显示为 NULL。

在 MySQL 中，insert 语句还有一个很好的特性，可以一次性插入多条记录，语法如下：

```
INSERT INTO   tablename   (field1, field2,……fieldn)
VALUES
(record1_value1, record1_value2,……record1_valuesn),
(record2_value1, record2_value2,……record2_valuesn),
……
(recordn_value1, recordn_value2,……recordn_valuesn)
;
```

可以看出，每条记录之间都用逗号进行了分隔。

下面的例子中，对表 dept 一次插入两条记录：

```
mysql> insert into dept values(5,'dept5'),(6,'dept6');
Query OK, 2 rows affected (0.04 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 5      | fin      |
| 5      | dept5    |
| 6      | dept6    |
+--------+----------+
5 rows in set (0.00 sec)
```

这个特性可以使得 MySQL 在插入大量记录时，节省很多的网络开销，大大提高插入效率。

2. 更新记录

对于表里的记录值，可以通过 update 命令进行更改，语法如下：

```
UPDATE tablename SET field1=value1，field2.=value2，……fieldn=valuen [WHERE CONDITION]
```

例如，将表 emp 中 ename 为“lisa”的薪水（sal）从 3000 更改为 4000：

```
mysql> update emp set sal=4000 where ename='lisa';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

在 MySQL 中，update 命令可以同时更新多个表中数据，语法如下：

```
UPDATE t1,t2…tn set t1.field1=expr1,tn.fieldn=exprn   [WHERE CONDITION]
```

在下例中，同时更新表 emp 中的字段 sal 和表 dept 中的字段 deptname：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 100.00  | 1      |
| lisa   | 2003-02-01 | 200.00  | 2      |
| bjguan | 2004-04-02 | 100.00  | 1      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)

mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 5      | fin      |
+--------+----------+
```

```
3 rows in set (0.00 sec)

mysql> update emp a,dept b set a.sal=a.sal*b.deptno,b.deptname=a.ename where a.deptno=b.deptno;
Query OK, 3 rows affected (0.04 sec)
Rows matched: 5  Changed: 3  Warnings: 0

mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 100.00  | 1      |
| lisa   | 2003-02-01 | 400.00  | 2      |
| bjguan | 2004-04-02 | 100.00  | 1      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+------------+---------+--------+
4 rows in set (0.01 sec)

mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | zzx      |
| 2      | lisa     |
| 5      | fin      |
+--------+----------+
3 rows in set (0.00 sec)
```

自此，两个表的数据同时进行了更新。

> 注意：多表更新的语法更多地用在了根据一个表的字段，来动态的更新另外一个表的字段

3. 删除记录

如果记录不再需要，可以用 delete 命令进行删除，语法如下：

```
DELETE FROM tablename [WHERE CONDITION]
```

例如，在 emp 中将 ename 为‘dony’的记录全部删除，命令如下：

```
mysql> delete from emp where ename='dony';
Query OK, 1 row affected (0.00 sec)
```

在 MySQL 中可以一次删除多个表的数据，语法如下：

```
DELETE t1,t2…tn FROM t1,t2…tn [WHERE CONDITION]
```

如果 from 后面的表名用别名，则 delete 后面的也要用相应的别名，否则会提示语法错误。在下例中，将表 emp 和 dept 中 deptno 为 3 的记录同时都删除：

```
mysql> select * from emp;
```

```
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 100.00  | 1      |
| lisa   | 2003-02-01 | 200.00  | 2      |
| bjguan | 2004-04-02 | 100.00  | 1      |
| bzshen | 2005-04-01 | 300.00  | 3      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+-----------+---------+--------+
5 rows in set (0.00 sec)

mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 3      | hr       |
| 5      | fin      |
+--------+----------+
4 rows in set (0.00 sec)
mysql> delete a,b from emp a,dept b where a.deptno=b.deptno and a.deptno=3;
Query OK, 2 rows affected (0.04 sec)

mysql>
mysql>
mysql> select * from emp;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 100.00  | 1      |
| lisa   | 2003-02-01 | 200.00  | 2      |
| bjguan | 2004-04-02 | 100.00  | 1      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+-----------+---------+--------+
4 rows in set (0.00 sec)

mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 5      | fin      |
```

```
+--------+----------+
3 rows in set (0.00 sec)
```

注意：不管是单表还是多表，不加 where 条件将会把表的所有记录删除，所以操作时一定要小心。

4. 查询记录

数据插入到数据库中后，就可以用 SELECT 命令进行各种各样的查询，使得输出的结果符合我们的要求。由于 SELECT 的语法很复杂，所有这里只介绍最基本的语法：

```
SELECT * FROM tablename [WHERE CONDITION]
```

查询最简单的方式是将记录全部选出，在下面的例子中，将表 emp 中的记录全部查询出来：

```
mysql> select * from emp;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 3      |
+--------+-----------+---------+--------+
3 rows in set (0.00 sec)
```

其中“*”表示要将所有的记录都选出来，也可以用逗号分割的所有字段来代替，例如，以下两个查询是等价的：

```
mysql> select * from emp;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 3      |
+--------+-----------+---------+--------+
3 rows in set (0.00 sec)
```

```
mysql> select ename,hiredate,sal,deptno from emp;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 3      |
+--------+-----------+---------+--------+
3 rows in set (0.00 sec)
```

“*”的好处是当需要查询所有字段信息时候，查询语句很简单，但是要只查询部分字段的时候，必须要将字段一个一个列出来。

上例中已经介绍了查询全部记录的语法，但是在实际应用中，用户还会遇到各种各样的查询要求，下面将分别介绍。

（1）查询不重复的记录。

有时需要将表中的记录去掉重复后显示出来，可以用 distinct 关键字来实现：

```
mysql> select ename,hiredate,sal,deptno from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
+--------+------------+---------+--------+
3 rows in set (0.00 sec)

mysql> select distinct deptno from emp;
+--------+
| deptno |
+--------+
| 1      |
| 2      |
+--------+
2 rows in set (0.00 sec)
```

（2）条件查询。

在很多情况下，用户并不需要查询所有的记录，而只是需要根据限定条件来查询一部分数据，用 where 关键字可以来实现这样的操作。

例如，需要查询所有 deptno 为 1 的记录：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
+--------+------------+---------+--------+
3 rows in set (0.00 sec)

mysql> select * from emp where deptno=1;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
+--------+------------+---------+--------+
2 rows in set (0.00 sec)
```

结果集中将符合条件的记录列出来。上面的例子中，where 后面的条件是一个字段的‘=’比较，除了‘=’外，还可以使用>、<、>=、<=、!=等比较运算符；多个条件之间还可以使用 or、and 等逻辑运算符进行多条件联合查询，运算符会在以后章节中详细讲解。
以下是一个使用多字段条件查询的例子：

```
mysql> select * from emp where deptno=1 and sal<3000;
+-------+-----------+---------+--------+
| ename | hiredate  | sal     | deptno |
+-------+-----------+---------+--------+
| zzx   | 2000-01-01 | 2000.00 | 1      |
+-------+-----------+---------+--------+
1 row in set (0.00 sec)
```

（3）排序和限制。
我们经常会有这样的需求，取出按照某个字段进行排序后的记录结果集，这就用到了数据库的排序操作，用关键字 ORDER BY 来实现，语法如下：

```
SELECT * FROM tablename [WHERE CONDITION] [ORDER BY field1 [DESC|ASC], field2 [DESC|ASC], ……fieldn [DESC|ASC]]
```

其中，DESC 和 ASC 是排序顺序关键字，DESC 表示按照字段进行降序排列，ASC 则表示升序排列，如果不写此关键字默认是升序排列。ORDER BY 后面可以跟多个不同的排序字段，并且每个排序字段可以有不同的排序顺序。
例如，把 emp 表中的记录按照工资高低进行显示：

```
mysql> select * from emp order by sal;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| bzshen | 2005-04-01 | 3000.00 | 3      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
+--------+-----------+---------+--------+
4 rows in set (0.00 sec)
```

如果排序字段的值一样，则值相同的字段按照第二个排序字段进行排序，以此类推。如果只有一个排序字段，则这些字段相同的记录将会无序排列。
例如，把 emp 表中的记录按照部门编号 deptno 字段排序：

```
mysql> select * from emp order by deptno;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+-----------+---------+--------+
4 rows in set (0.00 sec)
```

对于 deptno 相同的前两条记录，如果要按照工资由高到低排序，可以使用以下命令：

```
mysql> select * from emp order by deptno,sal desc;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| bjguan | 2004-04-02 | 5000.00 | 1      |
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+-----------+---------+--------+
4 rows in set (0.00 sec)
```

对于排序后的记录，如果希望只显示一部分，而不是全部，这时，就可以使用 LIMIT 关键字来实现，LIMIT 的语法如下：

```
SELECT ……[LIMIT offset_start,row_count]
```

其中 offset_start 表示记录的起始偏移量，row_count 表示显示的行数。

在默认情况下，起始偏移量为 0，只需要写记录行数就可以，这时候，显示的实际就是前 n 条记录，看下面例子：

例如，显示 emp 表中按照 sal 排序后的前 3 条记录：

```
mysql> select * from emp order by sal limit 3;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+-----------+---------+--------+
3 rows in set (0.00 sec)
```

如果要显示 emp 表中按照 sal 排序后从第二条记录开始，显示 3 条记录：

```
mysql> select * from emp order by sal limit 1,3;
+--------+-----------+---------+--------+
| ename  | hiredate  | sal     | deptno |
+--------+-----------+---------+--------+
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
+--------+-----------+---------+--------+
3 rows in set (0.00 sec)
```

limit 经常和 order by 一起配合使用来进行记录的分页显示。

> 注意：limit 属于 MySQL 扩展 SQL92 后的语法，在其他数据库上并不能通用。

（4）聚合。

很多情况下，我们需要进行一些汇总操作，比如统计整个公司的人数或者统计每个部门的人

数，这个时就要用到 SQL 的聚合操作。

聚合操作的语法如下：

```
SELECT [field1,field2,……fieldn] fun_name
FROM tablename
[WHERE where_contition]
[GROUP BY field1,field2,……fieldn
[WITH ROLLUP]]
[HAVING where_contition]
```

对其参数进行以下说明。

- fun_name 表示要做的聚合操作，也就是聚合函数，常用的有 sum（求和）、count(*)（记录数）、max（最大值）、min（最小值）。
- GROUP BY 关键字表示要进行分类聚合的字段，比如要按照部门分类统计员工数量，部门就应该写在 group by 后面。
- WITH ROLLUP 是可选语法，表明是否对分类聚合后的结果进行再汇总。
- HAVING 关键字表示对分类后的结果再进行条件的过滤。

> 注意：having 和 where 的区别在于 having 是对聚合后的结果进行条件的过滤，而 where 是在聚合前就对记录进行过滤，如果逻辑允许，我们尽可能用 where 先过滤记录，这样因为结果集减小，将对聚合的效率大大提高，最后再根据逻辑看是否用 having 进行再过滤。

例如，要 emp 表中统计公司的总人数：

```
mysql> select count(1) from emp;
+----------+
| count(1) |
+----------+
|        4 |
+----------+
1 row in set (0.00 sec)
```

在此基础上，要统计各个部门的人数：

```
mysql> select deptno,count(1) from emp group by deptno;
+--------+----------+
| deptno | count(1) |
+--------+----------+
|      1 |        2 |
|      2 |        1 |
|      4 |        1 |
+--------+----------+
3 rows in set (0.00 sec)
```

更细一些，既要统计各部门人数，又要统计总人数：

```
mysql> select deptno,count(1) from emp group by deptno with rollup;
+--------+----------+
| deptno | count(1) |
+--------+----------+
|      1 |        2 |
|      2 |        1 |
```

```
|      4 |        1 |
|   NULL |        4 |
+--------+----------+
4 rows in set (0.00 sec)
```

统计人数大于 1 人的部门：

```
mysql> select deptno,count(1) from emp group by deptno having count(1)>1;
+--------+----------+
| deptno | count(1) |
+--------+----------+
|      1 |        2 |
+--------+----------+
1 row in set (0.00 sec)
```

最后统计公司所有员工的薪水总额、最高和最低薪水：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 100.00  |      1 |
| lisa   | 2003-02-01 | 400.00  |      2 |
| bjguan | 2004-04-02 | 100.00  |      1 |
| dony   | 2005-02-05 | 2000.00 |      4 |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)

mysql> select sum(sal),max(sal),min(sal) from emp;
+----------+----------+----------+
| sum(sal) | max(sal) | min(sal) |
+----------+----------+----------+
| 2600.00  | 2000.00  | 100.00   |
+----------+----------+----------+
1 row in set (0.00 sec)
```

（5）表连接。

当需要同时显示多个表中的字段时，就可以用表连接来实现这样的功能。

从大类上分，表连接分为内连接和外连接，它们之间的最主要区别是内连接仅选出两张表中互相匹配的记录，而外连接会选出其他不匹配的记录。我们最常用的是内连接。

例如，查询出所有雇员的名字和所在部门名称，因为雇员名称和部门分别存放在表 emp 和 dept 中，因此，需要使用表连接来进行查询：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
```

```
| bjguan | 2004-04-02 | 5000.00 | 1      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)

mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 3      | hr       |
+--------+----------+
3 rows in set (0.00 sec)

mysql> select ename,deptname from emp,dept where emp.deptno=dept.deptno;
+--------+----------+
| ename  | deptname |
+--------+----------+
| zzx    | tech     |
| lisa   | sale     |
| bjguan | tech     |
| bzshen | hr       |
+--------+----------+
4 rows in set (0.00 sec)
```

外连接有分为左连接和右连接，具体定义如下。

- 左连接：包含所有的左边表中的记录甚至是右边表中没有和它匹配的记录
- 右连接：包含所有的右边表中的记录甚至是左边表中没有和它匹配的记录

例如，查询 emp 中所有用户名和所在部门名称：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+------------+---------+--------+
5 rows in set (0.00 sec)
mysql> select * from dept;
+--------+----------+
| deptno | deptname |
```

```
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 3      | hr       |
+--------+----------+
3 rows in set (0.00 sec)
mysql> select ename,deptname from emp left join dept  on  emp.deptno=dept.deptno;
+--------+----------+
| ename  | deptname |
+--------+----------+
| zzx    | tech     |
| lisa   | sale     |
| bjguan | tech     |
| bzshen | hr       |
| dony   |          |
+--------+----------+
5 rows in set (0.00 sec)
```

比较这个查询和上例中的查询，都是查询用户名和部门名，两者的区别在于本例中列出了所有的用户名，即使有的用户名（dony）并不存在合法的部门名称（部门号为 4，在 dept 中没有这个部门）；而上例中仅仅列出了存在合法部门的用户名和部门名称。

右连接和左连接类似，两者之间可以互相转化，例如，上面的例子可以改写为如下的右连接：

```
mysql> select ename,deptname from dept right join emp on dept.deptno=emp.deptno;
+--------+----------+
| ename  | deptname |
+--------+----------+
| zzx    | tech     |
| lisa   | sale     |
| bjguan | tech     |
| bzshen | hr       |
| dony   |          |
+--------+----------+
5 rows in set (0.00 sec)
```

（6）子查询。

某些情况下，当我们查询的时候，需要的条件是另外一个 select 语句的结果，这个时候，就要用到子查询。用于子查询的关键字主要包括 in、not in、=、!=、exists、not exists 等。

例如，从 emp 表中查询出所有部门在 dept 表中的所有记录：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
```

```
| bzshen | 2005-04-01 | 4000.00 | 3      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+------------+---------+--------+
5 rows in set (0.00 sec)

mysql> select * from dept;
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 3      | hr       |
| 5      | fin      |
+--------+----------+
4 rows in set (0.00 sec)

mysql> select * from emp where deptno in(select deptno from dept);
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)
```

如果子查询记录数唯一，还可以用=代替 in：

```
mysql> select * from emp where deptno = (select deptno from dept);
ERROR 1242 (21000): Subquery returns more than 1 row
mysql> select * from emp where deptno = (select deptno from dept limit 1);
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
+--------+------------+---------+--------+
2 rows in set (0.00 sec)
```

某些情况下，子查询可以转化为表连接，例如：

```
mysql> select * from emp where deptno in(select deptno from dept);
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
```

```
| bjguan | 2004-04-02 | 5000.00 | 1      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)
转换为表连接后：
mysql> select emp.* from emp ,dept where emp.deptno=dept.deptno;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 2000.00 | 1      |
| lisa   | 2003-02-01 | 4000.00 | 2      |
| bjguan | 2004-04-02 | 5000.00 | 1      |
| bzshen | 2005-04-01 | 4000.00 | 3      |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)
```

注意：子查询和表连接之间的转换主要应用在两个方面：

- MySQL 4.1 以前的版本不支持子查询，需要用表连接来实现子查询的功能
- 表连接在很多情况下用于优化子查询

（7）记录联合。

我们经常会碰到这样的应用，将两个表的数据按照一定的查询条件查询出来后，将结果合并到一起显示出来，这个时候，就需要用 union 和 union all 关键字来实现这样的功能，具体语法如下：

```
SELECT * FROM t1
UNION|UNION ALL
SELECT * FROM t2
……
UNION|UNION ALL
SELECT * FROM tn;
```

UNION 和 UNION ALL 的主要区别是 UNION ALL 是把结果集直接合并在一起，而 UNION 是将 UNION ALL 后的结果进行一次 DISTINCT，去除重复记录后的结果。

来看下面例子，将 emp 和 dept 表中的部门编号的集合显示出来：

```
mysql> select * from emp;
+--------+------------+---------+--------+
| ename  | hiredate   | sal     | deptno |
+--------+------------+---------+--------+
| zzx    | 2000-01-01 | 100.00  | 1      |
| lisa   | 2003-02-01 | 400.00  | 2      |
| bjguan | 2004-04-02 | 100.00  | 1      |
| dony   | 2005-02-05 | 2000.00 | 4      |
+--------+------------+---------+--------+
4 rows in set (0.00 sec)
mysql> select * from dept;
```

```
+--------+----------+
| deptno | deptname |
+--------+----------+
| 1      | tech     |
| 2      | sale     |
| 5      | fin      |
+--------+----------+
3 rows in set (0.00 sec)
mysql> select deptno from emp
    -> union all
    -> select deptno from dept;
+--------+
| deptno |
+--------+
| 1      |
| 2      |
| 1      |
| 4      |
| 1      |
| 2      |
| 5      |
+--------+
7 rows in set (0.00 sec)
```

如果希望将结果去掉重复记录后显示：

```
mysql> select deptno from emp
    -> union
    -> select deptno from dept;
+--------+
| deptno |
+--------+
| 1      |
| 2      |
| 4      |
| 5      |
+--------+
4 rows in set (0.00 sec)
```

2.2.4 DCL 语句

DCL 语句主要是 DBA 用来管理系统中的对象权限时所使用，一般的开发人员很少使用。下面通过一个例子来简单说明一下。

创建一个数据库用户 z1，具有对 sakila 数据库中所有表的 SELECT/INSERT 权限：

```
mysql> grant select,insert on sakila.* to 'z1'@'localhost' identified by '123';
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> exit
Bye
[mysql@db3 ~]$ mysql -uz1 -p123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21671 to server version: 5.1.9-beta-log

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

mysql> use sakila
Database changed
mysql> insert into emp values('bzshen','2005-04-01',3000,'3');
Query OK, 1 row affected (0.04 sec)
```

由于权限变更，需要将 z1 的权限变更，收回 INSERT，只能对数据进行 SELECT 操作：

```
 [mysql@db3 ~]$ mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21757 to server version: 5.1.9-beta-log

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

mysql> revoke insert on sakila.* from 'z1'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
```

用户 z1 重新登录后执行前面语句：

```
[mysql@db3 ~]$ mysql -uz1 -p123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21763 to server version: 5.1.9-beta-log

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

mysql> insert into emp values('bzshen','2005-04-01',3000,'3');
ERROR 1046 (3D000): No database selected
mysql> use sakila
Database changed
mysql> insert into emp values('bzshen','2005-04-01',3000,'3');
ERROR 1142 (42000): INSERT command denied to user 'z1'@'localhost' for table 'emp'
```

以上例子中的 grant 和 revoke 分别授出和收回了用户 z1 的部分权限，达到了我们的目的。关于权限的更多内容，将会在第 4 篇中详细介绍。

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

2.3 帮助的使用

在 MySQL 使用过程中，可能经常会遇到以下问题：

- 某个操作语法忘记了，需要快速查找。
- 当前版本上，某个字段类型我们想快速知道它的取值范围？
- 当前版本上，都支持哪些函数？希望有例子能快速入门。
- 当前版本上，是否支持某个功能？

对于上面列出的各种问题，我们可能想到的办法是查找 MySQL 的文档。不错，这些问题在 MySQL 官方文档中都可以很清楚地查到，但是却要耗费大量的时间和精力。
所以对于以上问题，最好的解决办法是使用 MySQL 安装后自带的帮助文档，这样在遇到问题时就可以方便快捷地进行查询。

2.3.1 按照层次看帮助

如果不知道帮助能够提供些什么，可以用“？contents”命令来显示所有可供查询的的分类，如下例所示：

```
mysql> ? contents
You asked for help about help category: "Contents"
For more information, type 'help <item>', where <item> is one of the following
categories:
   Account Management
   Administration
   Data Definition
   Data Manipulation
   Data Types
   Functions
   Functions and Modifiers for Use with GROUP BY
   Geographic Features
   Language Structure
   Plugins
   Storage Engines
   Stored Routines
   Table Maintenance
   Transactions
   Triggers
```

对于列出的分类，可以使用“？类别名称”的方式针对用户感兴趣的内容做进一步的查看。例如，想看看 MySQL 中都支持哪些数据类型，可以执行“? data types”命令：

```
mysql> ? data types
You asked for help about help category: "Data Types"
For more information, type 'help <item>', where <item> is one of the following
topics:
   AUTO_INCREMENT
   BIGINT
```

```
   BINARY
   BIT
   BLOB
   BLOB DATA TYPE
   BOOLEAN
   ……
```

上面列出了此版本支持的所有数据类型，如果想知道 int 类型的具体介绍，也可以利用上面的方法，做进一步的查看：

```
mysql> ? int
Name: 'INT'
Description:
INT[(M)] [UNSIGNED] [ZEROFILL]

A normal-size integer. The signed range is -2147483648 to 2147483647.
The unsigned range is 0 to 4294967295.
```

帮助文档中显示了 int 类型的详细描述。通过这种“？ 类别名称”的方式，就可以一层层地往下查找用户所关心的主题内容。

2.3.2 快速查阅帮助

在实际应用当中，如果需要快速查阅某项语法时，可以使用关键字进行快速查询。例如，想知道 show 命令都能看些什么东西，可以用如下命令：

```
mysql> ? show
Name: 'SHOW'
Description:
SHOW has many forms that provide information about databases, tables,
columns, or status information about the server. This section describes
those following:

SHOW AUTHORS
SHOW CHARACTER SET [LIKE 'pattern']
SHOW COLLATION [LIKE 'pattern']
SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE 'pattern']
SHOW CONTRIBUTORS
SHOW CREATE DATABASE db_name
SHOW CREATE EVENT event_name
SHOW CREATE FUNCTION funcname
……
```

例如，如果想参看 CREATE TABLE 的语法，可以使用以下命令：

```
mysql> ? create table
Name: 'CREATE TABLE'
Description:
Syntax:
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
```

```
    [table_option ...]
    [partition_options]

Or:

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_option ...]
    [partition_options]
    select_statement
、  。。。。。。
```

2.3.3 常用的网络资源

http://dev.mysql.com/downloads/是 MySQL 的官方网站，可以下载到各个版本的 MySQL 以及相关客户端开发工具等。
http://dev.mysql.com/doc/提供了目前最权威的 MySQL 数据库及工具的在线手册。
http://bugs.mysql.com/这里可以查看到 MySQL 已经发布的 bug 列表，或者向 MySQL 提交 bug 报告。
http://www.mysql.com/news-and-events/newsletter/通常会发布各种关于 MySQL 的最新消息。

2.4 小结

本章简单地介绍了 SQL 语句的基本分类 DML/DDL/DCL，并对每一种分类下的常用 SQL 的用法进行了举例说明。MySQL 在标准 SQL 的基础上进行了很多扩展，本章对常用的一些语法做了简单介绍，更详细的说明，读者可以参考 MySQL 的帮助或者官方文档。在本章的最后，还介绍了用户应如何使用 MySQL 中的帮助文档，以便快速查找各种语法定义。

第3章　MySQL 支持的数据类型

每一个常量，变量和参数都有数据类型，它用来指定一定的存储格式、约束和有效范围。MySQL 提供了多种数据类型，主要包括数值型、字符串类型、日期和时间类型。不同的 MySQL 版本支持的数据类型可能会稍有不同，用户可以通过查询相应版本的帮助文件来获得具体信息。本章将以 MySQL 5.0 为例，详细介绍 MySQL 中的各种数据类型。

3.1 数值类型

MySQL 支持所有标准 SQL 中的数值类型，其中包括严格数值类型（INTEGER、SMALLINT、DECIMAL 和 NUMERIC），以及近似数值数据类型（FLOAT、REAL 和 DOUBLE PRECISION），并

在此基础上做了扩展。扩展后增加了 TINYINT、MEDIUMINT 和 BIGINT 这 3 种长度不同的整型，并增加了 BIT 类型，用来存放位数据。表 3-1 中列出了 MySQL 5.0 中支持的所有数值类型，其中 INT 是 INTEGER 的同名词，DEC 是 DECIMAL 的同名词。

表 3-1　　MySQL 中的数值类型

整数类型	字节	最小值	最大值
TINYINT	1	有符号-128 无符号 0	有符号 127 无符号 255
SMALLINT	2	有符号-32768 无符号 0	有符号 32767 无符号 65535
MEDIUMINT	3	有符号-8388608 无符号 0	有符号 8388607 无符号 1677215
INT、INTEGER	4	有符号-2147483648 无符号 0	有符号 2147483647 无符号 4294967295
BIGINT	8	有符号-9223372036854775808 无符号 0	有符号 9223372036854775807 无符号 18446744073709551615
浮点数类型	**字节**	**最小值**	**最大值**
FLOAT	4	±1.175494351E-38	±3.402823466E+38
DOUBLE	8	±2.2250738585072014E-308	±1.7976931348623157E+308
定点数类型	**字节**	**描述**	
DEC(M,D)， DECIMAL(M,D)	M+2	最大取值范围与 DOUBLE 相同，给定 DECIMAL 的有效取值范围由 M 和 D 决定	
位类型	**字节**	**最小值**	**最大值**
BIT(M)	1～8	BIT(1)	BIT(64)

在整数类型中，按照取值范围和存储方式不同，分为 tinyint、smallint、mediumint、int、bigint 这 5 个类型。如果超出类型范围的操作，会发生“Out of range”错误提示。为了避免此类问题发生，在选择数据类型时要根据应用的实际情况确定其取值范围，最后根据确定的结果慎重选择数据类型。

对于整型数据，MySQL 还支持在类型名称后面的小括号内指定显示宽度，例如 int(5)表示当数值宽度小于 5 位的时候在数字前面填满宽度，如果不显示指定宽度则默认为 int(11)。一般配合 zerofill 使用，顾名思义，zerofill 就是用“0”填充的意思，也就是在数字位数不够的空间用字符“0”填满。以下几个例子分别描述了填充前后的区别。

（1）创建表 t1，有 id1 和 id2 两个字段，指定其数值宽度分别为 int 和 int(5)。

```
mysql> create table t1 (id1 int,id2 int(5));
Query OK, 0 rows affected (0.03 sec)
mysql> desc t1;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| id1   | int(11) | YES  |     | NULL    |       |
| id2   | int(5)  | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

（2）在 id1 和 id2 中都插入数值 1，可以发现格式没有异常。

```
mysql> insert into t1 values(1,1);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t1;
+------+------+
| id1  | id2  |
+------+------+
|    1 |    1 |
+------+------+
1 row in set (0.00 sec)
```

（3）分别修改 id1 和 id2 的字段类型，加入 zerofill 参数：

```
mysql> alter table t1 modify id1 int zerofill;
Query OK, 1 row affected (0.04 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> alter table t1 modify id2 int(5) zerofill;
Query OK, 1 row affected (0.03 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> select * from t1;
+------------+-------+
| id1        | id2   |
+------------+-------+
| 0000000001 | 00001 |
+------------+-------+
1 row in set (0.00 sec)
```

可以发现，在数值前面用字符“0”填充了剩余的宽度。大家可能会有所疑问，设置了宽度限制后，如果插入大于宽度限制的值，会不会截断或者插不进去报错？答案是肯定的：不会对插入的数据有任何影响，还是按照类型的实际精度进行保存，这是，宽度格式实际已经没有意义，左边不会再填充任何的“0”字符。下面在表 t1 的字段 id1 中插入数值 1，id2 中插入数值 1111111，位数为 7，大于 id2 的显示位数 5，再观察一下测试结果：

```
mysql> insert into t1 values(1,1111111);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t1;
+------------+---------+
| id1        | id2     |
+------------+---------+
| 0000000001 |   00001 |
| 0000000001 | 1111111 |
+------------+---------+
2 rows in set (0.00 sec)
```

很显然，如上面所说，id2 中显示了正确的数值，并没有受宽度限制影响。

所有的整数类型都有一个可选属性 UNSIGNED（无符号），如果需要在字段里面保存非

负数或者需要较大的上限值时，可以用此选项，它的取值范围是正常值的下限取 0，上限取原值的 2 倍，例如，tinyint 有符号范围是-128～+127，而无符号范围是 0～255。如果一个列指定为 zerofill，则 MySQL 自动为该列添加 UNSIGNED 属性。

另外，整数类型还有一个属性：AUTO_INCREMENT。在需要产生唯一标识符或顺序值时，可利用此属性，这个属性只用于整数类型。AUTO_INCREMENT 值一般从 1 开始，每行增加 1。在插入 NULL 到一个 AUTO_INCREMENT 列时，MySQL 插入一个比该列中当前最大值大 1 的值。一个表中最多只能有一个 AUTO_INCREMENT 列。对于任何想要使用 AUTO_INCREMENT 的列，应该定义为 NOT NULL，并定义为 PRIMARY KEY 或定义为 UNIQUE 键。例如，可按下列任何一种方式定义 AUTO_INCREMENT 列：

```
CREATE TABLE  AI (ID INT AUTO_INCREMENT NOT NULL PRIMARY KEY);
CREATE  TABLE  AI(ID INT AUTO_INCREMENT NOT NULL ,PRIMARY KEY(ID));
CREATE  TABLE  AI (ID INT AUTO_INCREMENT NOT NULL ,UNIQUE(ID));
```

对于小数的表示，MySQL 分为两种方式：浮点数和定点数。浮点数包括 float（单精度）和 double（双精度），而定点数则只有 decimal 一种表示。定点数在 MySQL 内部以字符串形式存放，比浮点数更精确，适合用来表示货币等精度高的数据。

浮点数和定点数都可以用类型名称后加“(*M*,*D*)”的方式来进行表示，“(*M*,*D*)”表示该值一共显示 *M* 位数字（整数位+小数位），其中 *D* 位位于小数点后面，*M* 和 *D* 又称为精度和标度。例如，定义为 float(7,4)的一个列可以显示为-999.9999。MySQL 保存值时进行四舍五入，因此如果在 float(7,4)列内插入 999.00009，近似结果是 999.0001。值得注意的是，浮点数后面跟“(*M*,*D*)”的用法是非标准用法，如果要用于数据库的迁移，则最好不要这么使用。float 和 double 在不指定精度时，默认会按照实际的精度（由实际的硬件和操作系统决定）来显示，而 decimal 在不指定精度时，默认的整数位为 10，默认的小数位为 0。

下面通过一个例子来比较 float、double 和 decimal 三者之间的不同。

（1）创建测试表，分别将 id1、id2、id3 字段设置为 float(5,2)、double(5,2)、decimal(5,2)。

```
CREATE TABLE `t1` (
  `id1` float(5,2) default NULL,
  `id2` double(5,2) default NULL,
  `id3` decimal(5,2) default NULL
)
```

（2）往 id1、id2 和 id3 这 3 个字段中插入数据 1.23。

```
mysql> insert into t1 values(1.23,1.23,1.23);
Query OK, 1 row affected (0.00 sec)

mysql>
mysql> select * from t1;
+------+------+------+
| id1  | id2  | id3  |
+------+------+------+
| 1.23 | 1.23 | 1.23 |
+------+------+------+
1 row in set (0.00 sec)
```

可以发现，数据都正确地插入了表 t1。

（3）再向 id1 和 id2 字段中插入数据 1.234，而 id3 字段中仍然插入 1.23。

```
mysql> insert into t1 values(1.234,1.234,1.23);
```

```
Query OK, 1 row affected (0.00 sec)

mysql> select * from t1;
+------+------+------+
| id1  | id2  | id3  |
+------+------+------+
| 1.23 | 1.23 | 1.23 |
| 1.23 | 1.23 | 1.23 |
+------+------+------+
2 rows in set (0.00 sec)
```

可以发现，id1、id2、id3 都插入了表 t1，但是 id1 和 id2 由于标度的限制，舍去了最后一位，数据变为了 1.23。

（4）同时向 id1、id2、id3 字段中都插入数据 1.234。

```
mysql> insert into t1 values(1.234,1.234,1.234);
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> show warnings;
+-------+------+----------------------------------------+
| Level | Code | Message                                |
+-------+------+----------------------------------------+
| Note  | 1265 | Data truncated for column 'id3' at row 1 |
+-------+------+----------------------------------------+
1 row in set (0.00 sec)

mysql> select * from t1;
+------+------+------+
| id1  | id2  | id3  |
+------+------+------+
| 1.23 | 1.23 | 1.23 |
| 1.23 | 1.23 | 1.23 |
| 1.23 | 1.23 | 1.23 |
+------+------+------+
3 rows in set (0.00 sec)
```

此时发现，虽然数据都插入进去，但是系统出现了一个 warning，报告 id3 被截断。如果是在传统的 SQLMode（第 16 章将会详细介绍 SQLMode）下，这条记录是无法插入的。

（5）将 id1、id2、id3 字段的精度和标度全部去掉，再次插入数据 1.23。

```
mysql> alter table t1 modify id1 float;
Query OK, 3 rows affected (0.03 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> alter table t1 modify id2 double;
Query OK, 3 rows affected (0.04 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
mysql> alter table t1 modify id3 decimal;
Query OK, 3 rows affected, 3 warnings (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
mysql> desc t1;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id1   | float        | YES  |     | NULL    |       |
| id2   | double       | YES  |     | NULL    |       |
| id3   | decimal(10,0) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)

mysql> insert into t1 values(1.234,1.234,1.234);
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> show warnings;
+-------+------+-----------------------------------------+
| Level | Code | Message                                 |
+-------+------+-----------------------------------------+
| Note  | 1265 | Data truncated for column 'id3' at row 1 |
+-------+------+-----------------------------------------+
1 row in set (0.00 sec)

mysql> select * from t1;
+-------+-------+------+
| id1   | id2   | id3  |
+-------+-------+------+
| 1.234 | 1.234 |    1 |
+-------+-------+------+
1 row in set (0.00 sec)
```

这个时候，可以发现 id1、id2 字段中可以正常插入数据，而 id3 字段的小数位被截断。

上面这个例子验证了上面提到的浮点数如果不写精度和标度，则会按照实际精度值显示，如果有精度和标度，则会自动将四舍五入后的结果插入，系统不会报错；定点数如果不写精度和标度，则按照默认值 decimal(10,0)来进行操作，并且如果数据超越了精度和标度值，系统则会报错。

对于 BIT（位）类型，用于存放位字段值，BIT(M)可以用来存放多位二进制数，M 范围从 1～64，如果不写则默认为 1 位。对于位字段，直接使用 SELECT 命令将不会看到结果，可以用 bin()（显示为二进制格式）或者 hex()（显示为十六进制格式）函数进行读取。

下面的例子中，对测试表 t2 中的 bit 类型字段 id 做 insert 和 select 操作，这里重点观察一下 select 的结果：

```
mysql> desc t2;
+-------+--------+------+-----+---------+-------+
```

```
| Field | Type   | Null | Key | Default | Extra |
+-------+--------+------+-----+---------+-------+
| id    | bit(1) | YES  |     | NULL    |       |
+-------+--------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> insert into t2 values(1);
Query OK, 1 row affected (0.00 sec)
mysql> select * from t2;
+------+
| id   |
+------+
|      |
+------+
1 row in set (0.00 sec)
```

可以发现，直接 select * 结果为 NULL。改用 bin()和 hex()函数再试试：

```
mysql> select bin(id),hex(id) from t2;
+---------+---------+
| bin(id) | hex(id) |
+---------+---------+
| 1       | 1       |
+---------+---------+
1 row in set (0.00 sec)
```

结果可以正常显示为二进制数字和十六进制数字。

数据插入 bit 类型字段时，首先转换为二进制，如果位数允许，将成功插入；如果位数小于实际定义的位数，则插入失败，下面的例子中，在 t2 表插入数字 2，因为它的二进制码是“10”，而 id 的定义是 bit(1)，将无法进行插入：

```
mysql> insert into t2 values(2);
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> show warnings;
+---------+------+-----------------------------------------------------+
| Level   | Code | Message                                             |
+---------+------+-----------------------------------------------------+
| Warning | 1264 | Out of range value adjusted for column 'id' at row 1 |
+---------+------+-----------------------------------------------------+
1 row in set (0.01 sec)
```

将 ID 定义修改为 bit(2)后，重新插入，插入成功：

```
mysql> alter table t2 modify id bit(2);
Query OK, 1 row affected (0.02 sec)
Records: 1  Duplicates: 0  Warnings: 0

mysql> insert into t2 values(2);
```

```
Query OK, 1 row affected (0.00 sec)

mysql> select bin(id),hex(id) from t2;
+---------+---------+
| bin(id) | hex(id) |
+---------+---------+
| 1       | 1       |
| 10      | 2       |
+---------+---------+
2 rows in set (0.00 sec)
```

3.2 日期时间类型

MySQL 中有多种数据类型可以用于日期和时间的表示，不同的版本可能有所差异，表 3-2 中列出了 MySQL 5.0 中所支持的日期和时间类型。

表 3-2　　MySQL 中的日期和时间类型

日期和时间类型	字节	最小值	最大值
DATE	4	1000-01-01	9999-12-31
DATETIME	8	1000-01-01 00:00:00	9999-12-31 23:59:59
TIMESTAMP	4	19700101080001	2038 年的某个时刻
TIME	3	-838:59:59	838:59:59
YEAR	1	1901	2155

这些数据类型的主要区别如下：

- 如果要用来表示年月日，通常用 DATE 来表示。
- 如果要用来表示年月日时分秒，通常用 DATETIME 表示。
- 如果只用来表示时分秒，通常用 TIME 来表示。
- 如果需要经常插入或者更新日期为当前系统时间，则通常使用 TIMESTAMP 来表示。TIMESTAMP 值返回后显示为“YYYY-MM-DD HH:MM:SS”格式的字符串，显示宽度固定为 19 个字符。如果想要获得数字值，应在 TIMESTAMP 列添加+0。
- 如果只是表示年份，可以用 YEAR 来表示，它比 DATE 占用更少的空间。YEAR 有 2 位或 4 位格式的年。默认是 4 位格式。在 4 位格式中，允许的值是 1901～2155 和 0000。在 2 位格式中，允许的值是 70～69，表示从 1970～2069 年。MySQL 以 YYYY 格式显示 YEAR 值。

从表 3-2 中可以看出，每种日期时间类型都有一个有效值范围，如果超出这个范围，在默认的 SQLMode 下，系统会进行错误提示，并将以零值来进行存储。不同日期类型零值的表示如表 3-3 所示。

表 3-3　　MySQL 中日期和时间类型的零值表示

数据类型	零值表示
DATETIME	0000-00-00 00:00:00
DATE	0000-00-00
TIMESTAMP	00000000000000

TIME	00:00:00
YEAR	0000

DATE、TIME 和 DATETIME 是最常使用的 3 种日期类型，以下例子在 3 种类型字段插入了相同的日期值，来看看它们的显示结果：

首先创建表 t，字段分别为 date、time、datetime 三种日期类型：

```
mysql> create table t (d date,t time,dt datetime);
Query OK, 0 rows affected (0.01 sec)

mysql> desc t;
+-------+----------+------+-----+---------+-------+
| Field | Type     | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| d     | date     | YES  |     | NULL    |       |
| t     | time     | YES  |     | NULL    |       |
| dt    | datetime | YES  |     | NULL    |       |
+-------+----------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

用 now()函数插入当前日期：

```
mysql> insert into t values(now(),now(),now());
Query OK, 1 row affected (0.00 sec)
```

查看显示结果：

```
mysql> select * from t;
+------------+----------+---------------------+
| d          | t        | dt                  |
+------------+----------+---------------------+
| 2007-07-19 | 17:41:13 | 2007-07-19 17:41:13 |
+------------+----------+---------------------+
1 row in set (0.00 sec)
```

显而易见，DATETIME是DATE和TIME的组合，用户可以根据不同的需要，来选择不同的日期或时间类型以满足不同的应用。

TIMESTAMP也用来表示日期，但是和DATETIME有所不同，后面的章节中会专门介绍。下例对TIMESTAMP类型的特性进行一些测试。

创建测试表t，字段id1为TIMESTAMP类型：

```
mysql> create table t (id1 timestamp);
Query OK, 0 rows affected (0.03 sec)
mysql> desc t;
+-------+-----------+------+-----+-------------------+-------+
| Field | Type      | Null | Key | Default           | Extra |
+-------+-----------+------+-----+-------------------+-------+
| id2   | timestamp | YES  |     | CURRENT_TIMESTAMP |       |
+-------+-----------+------+-----+-------------------+-------+
1 row in set (0.00 sec)
```

可以发现，系统给 tm 自动创建了默认值 CURRENT_TIMESTAMP（系统日期）。插入一个 NULL 值试试：

```
mysql> insert into t values(null);
Query OK, 1 row affected (0.00 sec)
mysql> select * from t;
+---------------------+
| t                   |
+---------------------+
| 2007-07-04 16:37:24 |
+---------------------+
1 row in set (0.00 sec)
```

果然，t中正确插入了系统日期。注意，MySQL只给表中的第一个TIMESTAMP字段设置默认值为系统日期，如果有第二个TIMESTAMP类型，则默认值设置为0值，测试如下：

```
mysql> alter table t add id2 timestamp;
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> show create table t \G;
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `id1` timestamp NOT NULL default CURRENT_TIMESTAMP,
  `id2` timestamp NOT NULL default '0000-00-00 00:00:00'
) ENGINE=MyISAM DEFAULT CHARSET=gbk
1 row in set (0.00 sec))
```

当然，可以修改id2的默认值为其他常量日期，但是不能再修改为current_timestmap，因为MySQL规定TIMESTAMP类型字段只能有一列的默认值为current_timestmap，如果强制修改，系统会报如下错误提示：

```
mysql> alter table t modify id2 timestamp default current_timestamp;
ERROR 1293 (HY000): Incorrect table definition; there can be only one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT or ON UPDATE clause
```

TIMESTAMP还有一个重要特点，就是和时区相关。当插入日期时，会先转换为本地时区后存放；而从数据库里面取出时，也同样需要将日期转换为本地时区后显示。这样，两个不同时区的用户看到的同一个日期可能是不一样的，下面的例子演示了这个差别。

（1）创建表t8，包含字段id1（TIMESTAMP）和id2（DATETIME），设置id2的目的是为了和id1做对比：

```
CREATE TABLE `t8` (
  `id1` timestamp NOT NULL default CURRENT_TIMESTAMP,
  `id2` datetime default NULL
)
Query OK, 0 rows affected (0.03 sec)
```

（2）查看当前时区：

```
mysql> show variables like 'time_zone';
+---------------+--------+
```

```
| Variable_name | Value  |
+---------------+--------+
| time_zone     | SYSTEM |
+---------------+--------+
1 row in set (0.00 sec)
```

可以发现，时区的值为“SYSTEM”，这个值默认是和主机的时区值一致的，因为我们在中国，这里的“SYSTEM”实际是东八区（+8:00）。

（3）用 now()函数插入当前日期：

```
mysql> select * from t8;
+---------------------+---------------------+
| id1                 | id2                 |
+---------------------+---------------------+
| 2007-09-25 17:26:50 | 2007-09-25 17:26:50 |
+---------------------+---------------------+
1 row in set (0.01 sec)
```

结果显示 id1 和 id2 的值完全相同。

（4）修改时区为东九区，再次查看表中日期：

```
mysql> set time_zone='+9:00';
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t8;
+---------------------+---------------------+
| id1                 | id2                 |
+---------------------+---------------------+
| 2007-09-25 18:26:50 | 2007-09-25 17:26:50 |
+---------------------+---------------------+
1 row in set (0.00 sec)
```

结果中可以发现，id1 的值比 id2 的值快了 1 个小时，也就是说，东九区的人看到的“2007-09-25 18:26:50”是当地时区的实际日期，也就是东八区的“2007-09-25 17:26:50”，如果还是以“2007-09-25 17:26:50”理解时间必然导致误差。

TIMESTAMP的取值范围为19700101080001到2038年的某一天，因此它不适合存放比较久远的日期，下面简单测试一些这个范围：

```
mysql> insert into t values (19700101080001);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+---------------------+
| t                   |
+---------------------+
| 1970-01-01 08:00:01 |
+---------------------+
1 row in set (0.00 sec)

mysql> insert into t values (19700101080000);
```

```
Query OK, 1 row affected, 1 warning (0.00 sec)
```

其中 19700101080000 超出了 tm 的下限，系统出现警告提示。查询一下，发现插入值变成了 0 值。

```
mysql> select * from t;
+---------------------+
| t                   |
+---------------------+
| 1970-01-01 08:00:01 |
| 0000-00-00 00:00:00 |
+---------------------+
2 rows in set (0.00 sec)
```

再来测试一下 TIMESTAMP 的上限值：

```
mysql> insert into t values('2038-01-19 11:14:07');
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+---------------------+
| t                   |
+---------------------+
| 2038-01-19 11:14:07 |
+---------------------+
1 row in set (0.00 sec)

mysql> insert into t values('2038-01-19 11:14:08');
Query OK, 1 row affected, 1 warning (0.00 sec)

mysql> select * from t;
+---------------------+
| t                   |
+---------------------+
| 2038-01-19 11:14:07 |
| 0000-00-00 00:00:00 |
+---------------------+
2 rows in set (0.00 sec)
```

从上面例子可以看出，TIMESTAMP和DATETIME的表示方法非常类似，区别主要有以下几点。

- TIMESTAMP支持的时间范围较小，其取值范围从19700101080001到2038年的某个时间，而DATETIME是从1000-01-01 00:00:00到9999-12-31 23:59:59，范围更大。
- 表中的第一个TIMESTAMP列自动设置为系统时间。如果在一个TIMESTAMP列中插入NULL，则该列值将自动设置为当前的日期和时间。在插入或更新一行但不明确给TIMESTAMP列赋值时也会自动设置该列的值为当前的日期和时间，当插入的值超出取值范围时，MySQL认为该值溢出，使用“0000-00-00 00:00:00”进行填补。

- TIMESTAMP的插入和查询都受当地时区的影响，更能反应出实际的日期。而DATETIME则只能反应出插入时当地的时区，其他时区的人查看数据必然会有误差的。
- TIMESTAMP的属性受MySQL版本和服务器SQLMode的影响很大，本章都是以MySQL 5.0为例进行介绍，在不同的版本下可以参考相应的MySQL帮助文档。

YEAR 类型主要用来表示年份，当应用只需要记录年份时，用 YEAR 比 DATE 将更节省空间。下面的例子在表 t 中定义了一个 YEAR 类型字段，并插入一条记录：

```
mysql> create table t(y year);
Query OK, 0 rows affected (0.01 sec)

mysql> desc t;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| y     | year(4) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> insert into t values(2100);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+------+
| y    |
+------+
| 2100 |
+------+
1 row in set (0.00 sec)
```

MySQL 以 YYYY 格式检索和显示 YEAR 值，范围是 1901～2155。当使用两位字符串表示年份时，其范围为“00”到“99”。

- “00”到“69”范围的值被转换为 2000～2069 范围的 YEAR 值
- “70”到“99”范围的值被转换为 1970～1999 范围的 YEAR 值。

细心的读者可能发现，在上面的例子中，日期类型的插入格式有很多，包括整数（如 2100）、字符串（如 2038-01-19 11:14:08）、函数（如 NOW()）等，大家可能会感到疑惑，到底什么样的格式才能够正确地插入到对应的日期字段中呢？下面以 DATETIME 为例进行介绍。

- YYYY-MM-DD HH:MM:SS 或 YY-MM-DD HH:MM:SS 格式的字符串。允许“不严格”语法：任何标点符都可以用做日期部分或时间部分之间的间割符。例如，“98-12-31 11:30:45”、“98.12.31 11+30+45”、“98/12/31 11*30*45”和“98@12@31 11^30^45”是等价的。对于包括日期部分间割符的字符串值，如果日和月的值小于 10，不需要指定两位数。“1979-6-9”与“1979-06-09”是相同的。同样，对于包括时间部分

间割符的字符串值，如果时、分和秒的值小于 10，不需要指定两位数。“1979-10-30 1:2:3”与“1979-10-30 01:02:03”相同。

- YYYYMMDDHHMMSS 或 YYMMDDHHMMSS 格式的没有间割符的字符串，假定字符串对于日期类型是有意义的。例如，“19970523091528”和“970523091528”被解释为“1997-05-23 09:15:28”，但“971122129015”是不合法的（它有一个没有意义的分钟部分），将变为“0000-00-00 00:00:00”。
- YYYYMMDDHHMMSS 或 YYMMDDHHMMSS 格式的数字，假定数字对于日期类型是有意义的。例如，19830905132800 和 830905132800 被解释为“1983-09-05 13:28:00”。数字值应为 6、8、12 或者 14 位长。如果一个数值是 8 或 14 位长，则假定为 YYYYMMDD 或 YYYYMMDDHHMMSS 格式，前 4 位数表示年。如果数字 是 6 或 12 位长，则假定为 YYMMDD 或 YYMMDDHHMMSS 格式，前 2 位数表示年。其他数字被解释为仿佛用零填充到了最近的长度。
- 函数返回的结果，其值适合 DATETIME、DATE 或者 TIMESTAMP 上下文，例如 NOW() 或 CURRENT_DATE。

对于其他数据类型，其使用原则与上面的内容类似，限于篇幅，这里就不再赘述。

最后通过一个例子，说明如何采用不同的格式将日期“2007-9-3 12:10:10”插入到 DATETIME 列中。

```
mysql> create table t6(dt datetime);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t6 values('2007-9-3 12:10:10');
Query OK, 1 row affected (0.00 sec)

mysql> insert into t6 values('2007/9/3 12+10+10');
Query OK, 1 row affected (0.00 sec)

mysql> insert into t6 values('20070903121010');
Query OK, 1 row affected (0.01 sec)

mysql> insert into t6 values(20070903121010);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t6;
+---------------------+
| dt                  |
+---------------------+
| 2007-09-03 12:10:10 |
| 2007-09-03 12:10:10 |
| 2007-09-03 12:10:10 |
| 2007-09-03 12:10:10 |
+---------------------+
4 rows in set (0.00 sec)
```

3.3 字符串类型

MySQL 中提供了多种对字符数据的存储类型，不同的版本可能有所差异。以 5.0 版本为例，MySQL 包括了 CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM 和 SET 等多种字符串类型。表 3-4 中详细列出了这些字符类型的比较。

表 3-4　　MySQL 中的字符类型

字符串类型	字节	描述及存储需求
CHAR（M）	M	M 为 0～255 之间的整数
VARCHAR（M）		M 为 0～65535 之间的整数，值的长度+1 个字节
TINYBLOB		允许长度 0～255 字节，值的长度+1 个字节
BLOB		允许长度 0～65535 字节，值的长度+2 个字节
MEDIUMBLOB		允许长度 0～167772150 字节，值的长度+3 个字节
LONGBLOB		允许长度 0～4294967295 字节，值的长度+4 个字节
TINYTEXT		允许长度 0～255 字节，值的长度+2 个字节
TEXT		允许长度 0～65535 字节，值的长度+2 个字节
MEDIUMTEXT		允许长度 0～167772150 字节，值的长度+3 个字节
LONGTEXT		允许长度 0～4294967295 字节，值的长度+4 个字节
VARBINARY（M）		允许长度 0～M 个字节的变长字节字符串，值的长度+1 个字节
BINARY（M）	M	允许长度 0～M 个字节的定长字节字符串

下面将分别对这些字符串类型做详细的介绍。

3.3.1 CHAR 和 VARCHAR 类型

CHAR 和 VARCHAR 很类似，都用来保存 MySQL 中较短的字符串。二者的主要区别在于存储方式的不同：CHAR 列的长度固定为创建表时声明的长度，长度可以为从 0～255 的任何值；而 VARCHAR 列中的值为可变长字符串，长度可以指定为 0～255（5.0.3 以前）或者 65535（5.0.3 以后）之间的值。在检索的时候，CHAR 列删除了尾部的空格，而 VARCHAR 则保留这些空格。下面的例子中通过给表 vc 中的 VARCHAR(4)和 char(4)字段插入相同的字符串来描述这个区别。

（1）创建测试表 vc，并定义两个字段 v VARCHAR(4)和 c CHAR(4)：

```
mysql> CREATE TABLE vc (v VARCHAR(4), c CHAR(4));
Query OK, 0 rows affected (0.16 sec)
```

（2）v 和 c 列中同时插入字符串“ab　”：

```
mysql> INSERT INTO vc VALUES ('ab  ', 'ab  ');
Query OK, 1 row affected (0.05 sec)
```

（3）显示查询结果：

```
mysql> select length(v),length(c) from vc;
+-----------+-----------+
```

```
| length(v) | length(c) |
+-----------+-----------+
|         4 |         2 |
+-----------+-----------+
1 row in set (0.01 sec)
```

可以发现，c 字段的 length 只有 2。给两个字段分别追加一个“+”字符看得更清楚：

```
mysql> SELECT CONCAT(v, '+'), CONCAT(c, '+') FROM vc;
+---------------+---------------+
| CONCAT(v, '+') | CONCAT(c, '+') |
+---------------+---------------+
| ab  +          | ab+           |
+---------------+---------------+
1 row in set (0.00 sec)
```

显然，CHAR 列最后的空格在做操作时都已经被删除，而 VARCHAR 依然保留空格。

3.3.2 BINARY 和 VARBINARY 类型

BINARY 和 VARBINARY 类似于 CHAR 和 VARCHAR，不同的是它们包含二进制字符串而不包含非二进制字符串。在下面的例子中，对表 t 中的 binary 字段 c 插入一个字符，研究一下这个字符到底是怎么样存储的。

（1）创建测试表 t，字段为 c BINARY(3)：

```
mysql> CREATE TABLE t (c BINARY(3));
Query OK, 0 rows affected (0.14 sec)
```

（2）往 c 字段中插入字符“a”：

```
mysql> INSERT INTO t SET   c='a';
Query OK, 1 row affected (0.06 sec)
```

（3）分别用以下几种模式来查看 c 列的内容：

```
mysql> select *,hex(c),c='a',c='a\0',c='a\0\0'  from t10;
+------+--------+-------+--------+-----------+
| c    | hex(c) | c='a' | c='a\0' | c='a\0\0' |
+------+--------+-------+--------+-----------+
| a    | 610000 |     0 |       0 |         1 |
+------+--------+-------+--------+-----------+
1 row in set (0.00 sec)
```

可以发现，当保存 BINARY 值时，在值的最后通过填充“0x00”（零字节）以达到指定的字段定义长度。从上例中看出，对于一个 BINARY(3)列，当插入时'a'变为'a\0\0'。

3.3.3 ENUM 类型

ENUM 中文名称叫枚举类型，它的值范围需要在创建表时通过枚举方式显式指定，对 1～255 个成员的枚举需要 1 个字节存储；对于 255～65535 个成员，需要 2 个字节存储。最多允许有 65535 个成员。下面往测试表 t 中插入几条记录来看看 ENUM 的使用方法。

（1）创建测试表 t，定义 gender 字段为枚举类型，成员为'M'和'F'：

```
mysql> create table t (gender enum('M','F'));
Query OK, 0 rows affected (0.14 sec)
```

（2）插入 4 条不同的记录：

```
mysql> INSERT INTO t  VALUES('M'),('1'),('f'),(NULL);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select * from t;
+--------+
| gender |
+--------+
| M      |
| M      |
| F      |
| NULL   |
+--------+
4 rows in set (0.01 sec)
```

从上面的例子中，可以看出 ENUM 类型是忽略大小写的，对'M'、'f'在存储的时候将它们都转成了大写，还可以看出对于插入不在 ENUM 指定范围内的值时，并没有返回警告，而是插入了 enum('M','F')的第一值'M'，这点用户在使用时要特别注意。

另外，ENUM 类型只允许从值集合中选取单个值，而不能一次取多个值。

3.3.4 SET 类型

Set 和 ENUM 类型非常类似，也是一个字符串对象，里面可以包含 0～64 个成员。根据成员的不同，存储上也有所不同。

- 1～8 成员的集合，占 1 个字节。
- 9～16 成员的集合，占 2 个字节。
- 17～24 成员的集合，占 3 个字节。
- 25～32 成员的集合，占 4 个字节。
- 33～64 成员的集合，占 8 个字节。

Set 和 ENUM 除了存储之外，最主要的区别在于 Set 类型一次可以选取多个成员，而 ENUM 则只能选一个。下面的例子在表 t 中插入了多组不同的成员：

```
Create table  t (col   set  ('a','b','c','d') ;
insert into t  values('a,b'),('a,d,a'),('a,b'),('a,c'),('a');
mysql> select * from t;
+------+
| col  |
+------+
| a,b  |
| a,d  |
| a,b  |
| a,c  |
| a    |
```

```
+------+
5 rows in set (0.00 sec)
```

SET 类型可以从允许值集合中选择任意 1 个或多个元素进行组合，所以对于输入的值只要是在允许值的组合范围内，都可以正确地注入到 SET 类型的列中。对于超出允许值范围的值例如（'a,d,f'）将不允许注入到上面例子中设置的 SET 类型列中，而对于（'a,d,a'）这样包含重复成员的集合将只取一次，写入后的结果为“a,d”，这一点请注意。

3.4 小结

本章主要介绍了 MySQL 支持的各种数据类型，并通过多个实例对它们的使用方法做了详细的说明。学完本章后，读者可以对每种数据类型的用途、物理存储、表示范围等有一个概要的了解。这样在面对具体应用时，就可以根据相应的特点来选择合适的数据类型，使得我们能够争取在满足应用的基础上，用较小的存储代价换来较高的数据库性能。

第4章　MySQL 中的运算符

MySQL 支持多种类型的运算符，来连接表达式的项。这些类型主要包括算术运算符、比较运算符、逻辑运算符和位运算符。本章将通过实例对 MySQL 5.0 支持的这几种运算符进行详细的介绍。

4.1 算术运算符

MySQL 支持的算术运算符包括加、减、乘、除和模运算。它们是最常使用、最简单的一类运算符。表 4-1 列出了这些运算符及其作用。

表 4-1　　MySQL 支持的算术运算符

运算符	作用
+	加法
-	减法
*	乘法
/，DIV	除法，返回商
%，MOD	除法，返回余数

下例中简单地描述了这几种运算符的使用方法：

```
mysql> select 0.1+ 0.3333 ,0.1-0.3333, 0.1*0.3333, 1/2,1%2;
+------------+-----------+-----------+--------+------+
| 0.1+ 0.3333 | 0.1-0.3333 | 0.1*0.3333 | 1/2    | 1%2  |
+------------+-----------+-----------+--------+------+
|      0.4333 |    -0.2333 |    0.03333 | 0.5000 |    1 |
+------------+-----------+-----------+--------+------+
1 row in set (0.00 sec)
```

- +运算符用于获得一个或多个值的和。

- -运算符用于从一个值中减去另一个值。
- *运算符使数字相乘,得到两个或多个值的乘积。
- /运算符用一个值除以另一个值得到商。
  - %运算符用一个值除以另外一个值得到余数。

除法运算和模运算中，如果除数为 0，将是非法除数，返回结果为 NULL，如下例所示：

```
mysql> select 1/0, 100%0 ;
+------+-------+
| 1/0  | 100%0 |
+------+-------+
| NULL |  NULL |
+------+-------+
1 row in set (0.02 sec)
```

对于模运算，还有另外一种表达方式，使用 MOD(a,b)函数与 a%b 效果一样：

```
mysql> select 3%2,mod(3,2);
+------+----------+
| 3%2  | mod(3,2) |
+------+----------+
|    1 |        1 |
+------+----------+
1 row in set (0.01 sec)
```

4.2 比较运算符

熟悉了最简单的算术运算符，再来看一下比较运算符。当使用 SELECT 语句进行查询时，MySQL 允许用户对表达式的左边操作数和右边操作数进行比较，比较结果为真，则返回 1，为假则返回 0，比较结果不确定则返回 NULL。表 4-2 列出了 MySQL 5.0 支持的各种比较运算符。

表 4-2　　MySQL 支持的比较运算符

运算符	作用
=	等于
<>或!=	不等于
<=>	NULL 安全的等于(NULL-safe)
<	小于
<=	小于等于
>	大于
>=	大于等于
BETWEEN	存在与指定范围
IN	存在于指定集合
IS NULL	为 NULL
IS NOT NULL	不为 NULL
LIKE	通配符匹配
REGEXP 或 RLIKE	正则表达式匹配

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

比较运算符可以用于比较数字、字符串和表达式。数字作为浮点数比较，而字符串以不区分大小写的方式进行比较。下面通过实例来学习各种比较运算符的使用。

- “=”运算符，用于比较运算符两侧的操作数是否相等，如果两侧操作数相等返回值为 1，否则为 0。注意 NULL 不能用于“=”比较。

```
mysql> select 1=0,1=1,NULL=NULL;
+-----+-----+-----------+
| 1=0 | 1=1 | NULL=NULL |
+-----+-----+-----------+
|   0 |   1 |      NULL |
+-----+-----+-----------+
1 row in set (0.00 sec)
```

- “<>”运算符，和“=”相反，如果两侧操作数不等，则值为 1，否则为 0。NULL 不能用于“<>”比较。

```
mysql> select 1<>0,1<>1,null<>null;
+------+------+------------+
| 1<>0 | 1<>1 | null<>null |
+------+------+------------+
|    1 |    0 |       NULL |
+------+------+------------+
1 row in set (0.00 sec)
```

- “<=>”安全的等于运算符，和“=”类似，在操作数相等时值为 1，不同之处在于即使操作的值为 NULL 也可以正确比较。

```
mysql> select 1<=>1,2<=>0 ,0<=>0, NULL<=>NULL;
+-------+-------+-------+-------------+
| 1<=>1 | 2<=>0 | 0<=>0 | NULL<=>NULL |
+-------+-------+-------+-------------+
|     1 |     0 |     1 |           1 |
+-------+-------+-------+-------------+
1 row in set (0.17 sec)
```

- “<”运算符，当左侧操作数小于右侧操作数时，其返回值为 1，否则其值为 0。

```
mysql> select 'a'<'b' ,'a'<'a' ,'a'<'c',1<2;
+---------+---------+---------+-----+
| 'a'<'b' | 'a'<'a' | 'a'<'c' | 1<2 |
+---------+---------+---------+-----+
|       1 |       0 |       1 |   1 |
+---------+---------+---------+-----+
1 row in set (0.03 sec)
```

- “<=”运算符，当左侧操作数小于等于右侧操作数时，其返回值为 1，否则返回值为 0。

```
mysql> select 'bdf'<='b','b'<='b' ,0<1;
+------------+----------+-----+
| 'bdf'<='b' | 'b'<='b' | 0<1 |
+------------+----------+-----+
```

```
|          0 |         1 |   1 |
+------------+-----------+-----+
1 row in set (0.00 sec)
```

- “>”运算符，当左侧操作数大于右侧操作数时，其返回值为 1，否则返回值为 0。

```
mysql> select 'a'>'b','abc'>'a' ,1>0;
+---------+-----------+-----+
| 'a'>'b' | 'abc'>'a' | 1>0 |
+---------+-----------+-----+
|       0 |         1 |   1 |
+---------+-----------+-----+
1 row in set (0.03 sec)
```

- “>=”运算符，当左侧操作数大于等于右侧操作数时，其返回值为 1，否则返回值为 0。

```
mysql> select 'a'>='b','abc'>='a' ,1>=0 ,1>=1;
+----------+------------+------+------+
| 'a'>='b' | 'abc'>='a' | 1>=0 | 1>=1 |
+----------+------------+------+------+
|        0 |          1 |    1 |    1 |
+----------+------------+------+------+
1 row in set (0.00 sec)
```

- “BETWEEN”运算符的使用格式为“a BETWEEN min AND max”，当 a 大于等于 min 并且小于等于 max，则返回值为 1，否则返回 0；当操作数 a、min、max 类型相同时，此表达式等价于（a>=min and a<=max），当操作数类型不同时，比较时会遵循类型转换原则进行转换后，再进行比较运算。下例中描述了 BETWEEN 的用法：

```
mysql> select 10 between 10 and 20, 9 between 10 and 20;
+---------------------+--------------------+
| 10 between 10 and 20 | 9 between 10 and 20 |
+---------------------+--------------------+
|                   1 |                  0 |
+---------------------+--------------------+
1 row in set (0.00 sec)
```

- “IN”运算符的使用格式为“a IN (value1,value2,...)”，当 a 的值存在于列表中时，则整个比较表达式返回的值为 1，否则返回 0。

```
 mysql> select 1 in (1,2,3) , 't' in ('t','a','b','l','e'),0 in (1,2);
+-------------+-----------------------------+------------+
| 1 in (1,2,3) | 't' in ('t','a','b','l','e') | 0 in (1,2) |
+-------------+-----------------------------+------------+
|           1 |                           1 |          0 |
+-------------+-----------------------------+------------+
1 row in set (0.00 sec)
```

- “IS NULL”运算符的使用格式为“a IS NULL”，当 a 的值为 NULL，则返回值为 1，否则返回 0。

```
mysql> select 0 is null, null is null;
+-----------+-------------+
```

```
| 0 is null | null is null |
+-----------+--------------+
|         0 |            1 |
+-----------+--------------+
1 row in set (0.02 sec)
```

- “IS NOT NULL”运算符的使用格式为“a IS NOT NULL”。和“IS NULL”相反，当 a 的值不为 NULL，则返回值为 1，否则返回 0。

```
mysql> select 0 is not  null, null is not  null;
+---------------+------------------+
| 0 is not  null | null is not  null |
+---------------+------------------+
|              1 |                0 |
+---------------+------------------+
1 row in set (0.00 sec)
```

- “LIKE”运算符的使用格式为“a LIKE %123%”，当 a 中含有字符串“123”时，则返回值为 1，否则返回 0。

```
mysql> select 123456 like '123%',123456 like '%123%',123456 like '%321%';
+--------------------+--------------------+--------------------+
| 123456 like '123%' | 123456 like '%123%' | 123456 like '%321%' |
+--------------------+--------------------+--------------------+
|                  1 |                  1 |                  0 |
+--------------------+--------------------+--------------------+
1 row in set (0.00 sec)
```

- “REGEXP”运算符的使用格式为“str REGEXP str_pat”，当 str 字符串中含有 str_pat 相匹配的字符串时，则返回值为 1，否则返回 0。REGEXP 运算符的使用方法将会在第 17 章中详细介绍。

```
mysql> select 'abcdef' regexp 'ab' ,'abcdefg' regexp 'k';
+---------------------+---------------------+
| 'abcdef' regexp 'ab' | 'abcdefg' regexp 'k' |
+---------------------+---------------------+
|                   1 |                   0 |
+---------------------+---------------------+
1 row in set (0.00 sec)
```

4.3 逻辑运算符

逻辑运算符又称为布尔运算符，用来确认表达式的真和假。MySQL 支持 4 种逻辑运算符，如表 4-3 所示。

表 4-3　　MySQL 中的逻辑运算符

运算符	作用
NOT 或!	逻辑非
AND 或&&	逻辑与
OR 或 ||	逻辑或

XOR	逻辑异或

- ““NOT”或“!”表示逻辑非。返回和操作数相反的结果：当操作数为 0（假），则返回值为 1，否则值为 0。但是有一点除外，那就是 NOT NULL 的返回值为 NULL，这一点请大家注意。如下例所示：

```
mysql> select not 0, not 1, not null ;
+-------+-------+----------+
| not 0 | not 1 | not null |
+-------+-------+----------+
|     1 |     0 |     NULL |
+-------+-------+----------+
1 row in set (0.00 sec)
```

- ““AND”或“&&”表示逻辑与运算。当所有操作数均为非零值并且不为 NULL 时，计算所得结果为 1，当一个或多个操作数为 0 时，所得结果为 0，操作数中有任何一个为 NULL 则返回值为 NULL。如下例所示：

```
mysql> select (1 and 1),(0 and 1) ,(3 and 1 ) ,(1 and null);
+-----------+-----------+-----------+--------------+
| (1 and 1) | (0 and 1) | (3 and 1 ) | (1 and null) |
+-----------+-----------+-----------+--------------+
|         1 |         0 |          1 |         NULL |
+-----------+-----------+-----------+--------------+
1 row in set (0.00 sec)
```

- “OR”或“||”表示逻辑或运算。当两个操作数均为非 NULL 值时，如有任意一个操作数为非零值，则结果为 1，否则结果为 0。当有一个操作数为 NULL 时，如另一个操作数为非零值，则结果为 1，否则结果为 NULL。假如两个操作数均为 NULL，则所得结果为 NULL。如下例所示：

```
mysql> select (1 or 0) ,(0 or 0),(1 or null) ,(1 or 1),(null or null);
+----------+----------+-------------+----------+----------------+
| (1 or 0) | (0 or 0) | (1 or null) | (1 or 1) | (null or null) |
+----------+----------+-------------+----------+----------------+
|        1 |        0 |           1 |        1 |           NULL |
+----------+----------+-------------+----------+----------------+
1 row in set (0.00 sec)
```

- “XOR”表示逻辑异或。当任意一个操作数为 NULL 时，返回值为 NULL。对于非 NULL 的操作数，如果两个的逻辑真假值相异，则返回结果 1；否则返回 0。如下例所示：

```
mysql> select 1 xor 1 ,0 xor 0,1 xor 0,0 xor 1,null xor 1;
+---------+---------+---------+---------+------------+
| 1 xor 1 | 0 xor 0 | 1 xor 0 | 0 xor 1 | null xor 1 |
+---------+---------+---------+---------+------------+
|       0 |       0 |       1 |       1 |       NULL |
+---------+---------+---------+---------+------------+
1 row in set (0.00 sec)
```

4.4 位运算符

位运算是将给定的操作数转化为二进制后，对各个操作数每一位都进行指定的逻辑运算，得到的二进制结果转换为十进制数后就是位运算的结果。MySQL 5.0 支持 6 种位运算符，如表 4-4 所示。

表 4-4　　MySQL 支持的位运算符

运算符	作用
&	位与（位 AND）
|	位或 （位 OR ）
^	位异或（位 XOR）
~	位取反
>>	位右移
<<	位左移

可以发现，位运算符中的位与“&”和位或“|”和前面介绍的逻辑与和逻辑或非常类似。其他操作符和逻辑操作有所不同，下面将分别举例介绍。

- “位与”对多个操作数的二进制位作逻辑与操作，例如 2&3，因为 2 的二进制数是 10，3 是 11，所有 10&11 的结果是 10，十进制数字还是 2，来看实际结果：

```
mysql> select 2&3;
+-----+
| 2&3 |
+-----+
|   2 |
+-----+
1 row in set (0.00 sec)
```

可以对 2 个以上操作数做或操作，测试一下 2&3&4，因为 4 的二进制是 100，和上面的 10 做与操作 100&010 后，结果应该是 000，可以看实际结果为：

```
mysql> select 2&3&4;
+-------+
| 2&3&4 |
+-------+
|     0 |
+-------+
1   row in set (0.00 sec)
```

- “位或”对多个操作数的二进制位作逻辑或操作，还是上面的例子，2|3 的结果应该是 10|11，结果还是 11，应该是 3，实际结果如下：

```
mysql> select 2|3;
+-----+
| 2|3 |
+-----+
|   3 |
+-----+
1 row in set (0.00 sec)
```

- “位异或”对操作数的二进制位做异或操作，10^11 的结果是 01，结果应该是 1，可以看实际结果为：

```
mysql> select 2^3;
+-----+
| 2^3 |
+-----+
|   1 |
+-----+
1 row in set (0.00 sec)
```

- “位取反”对操作数的二进制位作 NOT 操作，这里的操作数只能是一位，下面看一个经典的取反例子：对 1 做位取反，具体如下所示：

```
mysql> select ~1 ,~ 18446744073709551614;
+---------------------+-----------------------+
| ~1                  | ~ 18446744073709551614 |
+---------------------+-----------------------+
| 18446744073709551614 |                     1 |
+---------------------+-----------------------+
1 row in set (0.00 sec)
```

结果让大家可能有些疑惑，1 的位取反怎么会是这么大的数字？来研究一下，在 MySQL 中，常量数字默认会以 8 个字节来表示，8 个字节就是 64 位，常量 1 的二进制表示为 63 个“0”加 1 个“1”,位取反后就是 63 个“1”加一个“0”,转换为二进制后就是 18446744073709551614，实际结果如下：

```
mysql> select bin(18446744073709551614);
+-----------------------------------------------------------------+
| bin(18446744073709551614)                                       |
+-----------------------------------------------------------------+
| 1111111111111111111111111111111111111111111111111111111111111110 |
+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

- “位右移”对左操作数向右移动右操作数指定的位数。例如 100>>3，就是对 100 的二进制数 0001100100 右移 3 位，左边补 0，结果是 0000001100，转换为二进制数是 12，实际结果如下：

```
mysql> select 100>>3;
+--------+
| 100>>3 |
+--------+
|     12 |
+--------+
1 row in set (0.00 sec)
```

- “位左移”对左操作数向左移动右操作数指定的位数。例如 100<<3，就是对 100 的二进制数 0001100100 左移 3 位，右边补 0，结果是 1100100000，转换为二进制数是 800，实际结果如下：

```
mysql> select 100<<3;
+--------+
```

```
| 100<<3 |
+--------+
|    800 |
+--------+
1 row in set (0.00 sec)
```

4.5 运算符的优先级

前面介绍了 MySQL 支持的各种运算符的使用方法。在实际应用中，很可能将这些运算符进行混合运算，那么应该先进行哪些运算符的操作呢？表 4-5 中列出了所有的运算符，优先级由低到高排列，同一行中的运算符具有相同的优先级。

表 4-5　　MySQL 中的运算符优先级

优先级顺序	运算符
1	:=
2	||, OR, XOR
3	&&, AND
4	NOT
5	BETWEEN, CASE, WHEN, THEN, ELSE
6	=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
7	|
8	&
9	<<, >>
10	-, +
11	*, /, DIV, %, MOD
12	^
13	- (一元减号), ~ (一元比特反转)
14	!

在实际运行的时候，可以参考表 4-5 中的优先级。实际上，很少有人能将这些优先级熟练记忆，很多情况下我们都是用“（）”来将需要优先的操作括起来，这样既起到了优先的作用，又使得其他用户看起来更加易于理解。

4.6 小结

本章主要介绍了 MySQL 中支持的各种运算符。这些运算符可以帮助用户完成算术、比较、逻辑和位逻辑操作，大家在使用时要注意运算符的优先级。另外，在使用比较运算符时要保证比较的操作数类型是一致的，这样可以避免由于操作数类型的不一致而得出错误的数据。

第5章　常用函数

经常编写程序的朋友一定体会得到函数的重要性，丰富的函数往往能使用户的工作事半功倍。函数能帮助用户做很多事情，比如说字符串的处理、数值的运算、日期的运算等，在这方面MySQL 提供了多种内建函数帮助开发人员编写简单快捷的 SQL 语句，其中常用的函数有字符串函数、日期函数和数值函数。

在 MySQL 数据库中，函数可以用在 SELECT 语句及其子句（例如 where、order by、having 等）中，也可以用在 UPDATE、DELETE 语句及其子句中。本章将配合一些实例对这些常用函数进行详细的介绍。

5.1 字符串函数

字符串函数是最常用的一种函数了，如果大家编写过程序的话，不妨回过头去看看自己使用过的函数，可能会惊讶地发现字符串处理的相关函数占已使用过的函数很大一部分。MySQL 中字符串函数也是最丰富的一类函数，表 5-1 中列出了这些函数以供参考。

表 5-1　　MySQL 中的常用字符串函数

函数	功能
CANCAT(S1,S2,…Sn)	连接 S1,S2,…Sn 为一个字符串
INSERT(str,x,y,instr)	将字符串 str 从第 x 位置开始，y 个字符长的子串替换为字符串 instr
LOWER(str)	将字符串 str 中所有字符变为小写
UPPER(str)	将字符串 str 中所有字符变为大写
LEFT(str ,x)	返回字符串 str 最左边的 x 个字符
RIGHT(str,x)	返回字符串 str 最右边的 x 个字符
LPAD(str,n ,pad)	用字符串 pad 对 str 最左边进行填充，直到长度为 n 个字符长度
RPAD(str,n,pad)	用字符串 pad 对 str 最右边进行填充，直到长度为 n 个字符长度
LTRIM(str)	去掉字符串 str 左侧的空格
RTRIM(str)	去掉字符串 str 行尾的空格
REPEAT(str,x)	返回 str 重复 x 次的结果
REPLACE(str,a,b)	用字符串 b 替换字符串 str 中所有出现的字符串 a
STRCMP(s1,s2)	比较字符串 s1 和 s2
TRIM(str)	去掉字符串行尾和行头的空格
SUBSTRING(str,x,y)	返回从字符串 str x 位置起 y 个字符长度的字串

下面通过具体的实例来逐个地研究每个函数的用法，需要注意的是这里的例子仅仅在于说明各个函数的使用方法，所以函数都是单个出现的，但是在一个具体的应用中通常可能需要综合几个甚至几类函数才能实现相应的应用。

- CANCAT(S1,S2,…Sn)函数：把传入的参数连接成为一个字符串。

下面的例子把“aaa”、“bbb”、“ccc”3 个字符串连接成了一个字符串“aaabbbccc”。另外，任何字符串与 NULL 进行连接的结果都将是 NULL。

```
mysql> select concat('aaa','bbb','ccc') ,concat('aaa',null);
+-------------------------+------------------+
```

```
| concat('aaa','bbb','ccc') | concat('aaa',null) |
+---------------------------+--------------------+
| aaabbbccc                 | NULL               |
+---------------------------+--------------------+
1 row in set (0.05 sec)
```

- INSERT(str ,x,y,instr)函数：将字符串 str 从第 x 位置开始，y 个字符长的子串替换为字符串 instr。

下面的例子把字符串“beijing2008you”中的从第 12 个字符开始以后的 3 个字符替换成“me”。

```
mysql> select INSERT('beijing2008you',12,3, 'me') ;
+--------------------------------------+
| INSERT('beijing2008you',12,3, 'me') |
+--------------------------------------+
| beijing2008me                        |
+--------------------------------------+
1 row in set (0.00 sec)
```

- LOWER(str)和 UPPER(str)函数：把字符串转换成小写或大写。

在字符串比较中，通常要将比较的字符串全部转换为大写或者小写，如下例所示：

```
mysql>  select LOWER('BEIJING2008'), UPPER('beijing2008');
+---------------------+---------------------+
| LOWER('BEIJING2008') | UPPER('beijing2008') |
+---------------------+---------------------+
| beijing2008          | BEIJING2008          |
+---------------------+---------------------+
1 row in set (0.00 sec)
```

- LEFT(str,x)和 RIGHT(str,x)函数：分别返回字符串最左边的 x 个字符和最右边的 x 个字符。如果第二个参数是 NULL，那么将不返回任何字符串。

下例中显示了对字符串“beijing2008”应用函数后的结果。

```
mysql> SELECT LEFT('beijing2008',7),LEFT('beijing',null),RIGHT('beijing2008',4);

+----------------------+---------------------+-----------------------+
| LEFT('beijing2008',7) | LEFT('beijing',null) | RIGHT('beijing2008',4) |
+----------------------+---------------------+-----------------------+
| beijing               |                      | 2008                  |
+----------------------+---------------------+-----------------------+
1 row in set (0.00 sec)
```

- LPAD(str,n ,pad)和 RPAD(str,n ,pad)函数：用字符串 pad 对 str 最左边和最右边进行填充，直到长度为 n 个字符长度。

下例中显示了对字符串“2008”和“beijing”分别填充后的结果。

```
mysql> select lpad('2008',20,'beijing'),rpad('beijing',20,'2008');
+--------------------------+--------------------------+
| lpad('2008',20,'beijing') | rpad('beijing',20,'2008') |
+--------------------------+--------------------------+
| beijingbeijingbe2008      | beijing2008200820082      |
+--------------------------+--------------------------+
```

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

- LTRIM(str)和 RTRIM(str)函数：去掉字符串 str 左侧和右侧空格。

下例中显示了字符串“beijing”加空格进行过滤后的结果。

```
mysql> select ltrim('  |beijing'),rtrim('beijing|     ');
+--------------------+-----------------------+
| ltrim('  |beijing') | rtrim('beijing|     ') |
+--------------------+-----------------------+
| |beijing           | beijing|              |
+--------------------+-----------------------+
1 row in set (0.00 sec)
```

- REPEAT(str,x)函数：返回 str 重复 x 次的结果。

下例中对字符串“mysql”重复显示了 3 次。

```
mysql> select  repeat('mysql ',3);
+--------------------+
| repeat('mysql ',3) |
+--------------------+
| mysql mysql mysql  |
+--------------------+
1 row in set (0.00 sec)
```

- REPLACE(str,a,b)函数：用字符串 b 替换字符串 str 中所有出现的字符串 a。

下例中用字符串“2008”代替了字符串“beijing_2010”中的“_2010”。

```
mysql> select replace('beijing_2010','_2010','2008');
+-------------------------------------+
| replace('beijing_2010','_2010','2008') |
+-------------------------------------+
| beijing2008                         |
+-------------------------------------+
1 row in set (0.00 sec)
```

STRCMP(s1,s2)函数：比较字符串 s1 和 s2 的 ASCII 码值的大小。如果 s1 比 s2 小，那么返回-1；如果 s1 与 s2 相等，那么返回 0；如果 s1 比 s2 大，那么返回 1。如下例：

```
mysql> select  strcmp('a','b'),strcmp('b','b'),strcmp('c','b');
+-----------------+-----------------+-----------------+
| strcmp('a','b') | strcmp('b','b') | strcmp('c','b') |
+-----------------+-----------------+-----------------+
|              -1 |               0 |               1 |
+-----------------+-----------------+-----------------+
1 row in set (0.00 sec)
```

- TRIM(str)函数：去掉目标字符串的开头和结尾的空格。

下例中对字符串“$ beijing2008 $ ”进行了前后空格的过滤。

```
mysql> select trim(' $ beijing2008 $  ');
+----------------------------+
| trim(' $ beijing2008 $  ') |
+----------------------------+
| $ beijing2008 $            |
```

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

- SUBSTRING(str,x,y)函数：返回从字符串 str 中的第 x 位置起 y 个字符长度的字串。

此函数经常用来对给定字符串进行字串的提取，如下例所示。

```
mysql> select substring('beijing2008',8,4),substring('beijing2008',1,7);
+----------------------------+----------------------------+
| substring('beijing2008',8,4) | substring('beijing2008',1,7) |
+----------------------------+----------------------------+
| 2008                       | beijing                    |
+----------------------------+----------------------------+
```

5.2 数值函数

MySQL 中另外一类很重要的函数就是数值函数，这些函数能处理很多数值方面的运算。可以想象，如果没有这些函数的支持，用户在编写有关数值运算方面的代码时将会困难重重，举个例子，如果没有 ABS 函数的话，如果要取一个数值的绝对值，就需要进行好多次判断才能返回这个值，而数字函数能够大大提高用户的工作效率。表 5-2 中列出了在 MySQL 中会经常使用的数值函数。

表 5-2　　MySQL 中的常用数值函数

函数	功能
ABS(x)	返回 x 的绝对值
CEIL(x)	返回大于 x 的最大整数值
FLOOR(x)	返回小于 x 的最大整数值
MOD(x，y)	返回 x/y 的模
RAND()	返回 0 到 1 内的随机值
ROUND(x,y)	返回参数 x 的四舍五入的有 y 位小数的值
TRUNCATE(x,y)	返回数字 x 截断为 y 位小数的结果

下面将结合实例对这些函数进行介绍。

- ABS(x)函数：返回 x 的绝对值。

下例中显示了对正数和负数分别取绝对值之后的结果。

```
mysql> select ABS(-0.8) ,ABS(0.8);
+-----------+----------+
| ABS(-0.8) | ABS(0.8) |
+-----------+----------+
|       0.8 |      0.8 |
+-----------+----------+
1 row in set (0.09 sec)
```

- CEIL(x)函数：返回大于 x 的最大整数。

下例中显示了对 0.8 和－0.8 分别 CEIL 后的结果。

```
mysql> select CEIL(-0.8),CEIL(0.8);
+------------+-----------+
| CEIL(-0.8) | CEIL(0.8) |
```

```
+------------+-----------+
|          0 |         1 |
+------------+-----------+
1 row in set (0.03 sec)
```

- FLOOR(x)函数：返回小于 x 的最大整数，和 CEIL 的用法刚好相反。

下例中显示了对 0.8 和－0.8 分别 FLOOR 后的结果。

```
mysql> select FLOOR(-0.8), FLOOR(0.8);
+-------------+------------+
| FLOOR(-0.8) | FLOOR(0.8) |
+-------------+------------+
|          -1 |          0 |
+-------------+------------+
1 row in set (0.00 sec)
```

- MOD(x，y)函数：返回 x/y 的模。

和 x%y 的结果相同，模数和被模数任何一个为 NULL 结果都为 NULL。如下例所示：

```
mysql> select MOD(15,10),MOD(1,11),MOD(NULL,10);
+------------+-----------+--------------+
| MOD(15,10) | MOD(1,11) | MOD(NULL,10) |
+------------+-----------+--------------+
|          5 |         1 |         NULL |
+------------+-----------+--------------+
1 row in set (0.00 sec)
```

- RAND()函数：返回 0 到 1 内的随机值。

每次执行结果都不一样，如下例所示：

```
mysql> select RAND(),RAND();
+------------------+------------------+
| RAND()           | RAND()           |
+------------------+------------------+
| 0.12090325459922 | 0.83369727882901 |
+------------------+------------------+
1 row in set (0.00 sec)
```

利用此函数可以取任意指定范围内的随机数，比如需要产生 0～100 内的任意随机整数，可以操作如下：

```
mysql> select ceil(100*rand()),ceil(100*rand());
+-----------------+-----------------+
| ceil(100*rand()) | ceil(100*rand()) |
+-----------------+-----------------+
|              91 |              15 |
+-----------------+-----------------+
1 row in set (0.00 sec)
```

- ROUND(x,y)函数：返回参数 x 的四舍五入的有 y 位小数的值。

如果是整数，将会保留 y 位数量的 0；如果不写 y，则默认 y 为 0，即将 x 四舍五入后取整。适合于将所有数字保留同样小数位的情况。如下例所示。

```
mysql> select ROUND(1.1),ROUND(1.1,2),ROUND(1,2);
+------------+--------------+------------+
| ROUND(1.1) | ROUND(1.1,2) | ROUND(1,2) |
+------------+--------------+------------+
|          1 |         1.10 |       1.00 |
+------------+--------------+------------+
1 row in set (0.00 sec)
```

- TRUNCATE(x,y)函数：返回数字 x 截断为 y 位小数的结果。

注意 TRUNCATE 和 ROUND 的区别在于 TRUNCATE 仅仅是截断，而不进行四舍五入。下例中描述了二者的区别：

```
mysql> select ROUND(1.235,2),TRUNCATE(1.235,2);
+----------------+-----------------+
| ROUND(1.235,2) | TRUNCATE(1.235,2) |
+----------------+-----------------+
|           1.24 |              1.23 |
+----------------+-----------------+
1 row in set (0.00 sec)
```

5.3 日期和时间函数

有时我们可能会遇到这样的需求：当前时间是多少、下个月的今天是星期几、统计截止到当前日期前 3 天的收入总和等。这些需求就需要日期和时间函数来实现，表 5-3 列出了 MySQL 中支持的一些常用日期和时间函数。

表 5-3　　MySQL 中的常用日期时间函数

函数	功能
CURDATE()	返回当前日期
CURTIME()	返回当前时间
NOW()	返回当前的日期和时间
UNIX_TIMESTAMP(date)	返回日期 date 的 UNIX 时间戳
FROM_UNIXTIME	返回 UNIX 时间戳的日期值
WEEK(date)	返回日期 date 为一年中的第几周
YEAR(date)	返回日期 date 的年份
HOUR(time)	返回 time 的小时值
MINUTE(time)	返回 time 的分钟值
MONTHNAME(date)	返回 date 的月份名
DATE_FORMAT(date,fmt)	返回按字符串 fmt 格式化日期 date 值
DATE_ADD(date,INTERVAL expr type)	返回一个日期或时间值加上一个时间间隔的时间值
DATEDIFF(expr,expr2)	返回起始时间 expr 和结束时间 expr2 之间的天数

下面结合一些实例来逐个讲解每个函数的使用方法。

- CURDATE()函数：返回当前日期，只包含年月日。

```
mysql> select CURDATE();
```

```
+------------+
| CURDATE()  |
+------------+
| 2007-07-11 |
+------------+
1 row in set (0.03 sec)
```

- CURTIME()函数：返回当前时间，只包含时分秒。

```
mysql> select CURTIME();
+-----------+
| CURTIME() |
+-----------+
| 14:13:46  |
+-----------+
1 row in set (0.00 sec)
```

- NOW()函数：返回当前的日期和时间，年月日时分秒全都包含。

```
mysql> select NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2007-07-11 14:14:06 |
+---------------------+
1 row in set (0.00 sec)
```

- UNIX_TIMESTAMP(date)函数：返回日期 date 的 UNIX 时间戳。

```
mysql> select UNIX_TIMESTAMP(now());
+------------------------+
| UNIX_TIMESTAMP(now()) |
+------------------------+
|            1184134516 |
+------------------------+
1 row in set (0.02 sec)
```

- FROM_UNIXTIME（unixtime）函数：返回 UNIXTIME 时间戳的日期值，和 UNIX_TIMESTAMP(date)互为逆操作。

```
mysql> select FROM_UNIXTIME(1184134516) ;
+---------------------------+
| FROM_UNIXTIME(1184134516) |
+---------------------------+
| 2007-07-11 14:15:16       |
+---------------------------+
1 row in set (0.00 sec)
```

- WEEK(DATE)和 YEAR(DATE)函数：前者返回所给的日期是一年中的第几周，后者返回所给的日期是哪一年。

```
mysql> select WEEK(now()),YEAR(now());
+-------------+-------------+
| WEEK(now()) | YEAR(now()) |
```

```
+-------------+-------------+
|          27 |        2007 |
+-------------+-------------+
1 row in set (0.02 sec)
```

- HOUR(time)和 MINUTE(time)函数：前者返回所给时间的小时，后者返回所给时间的分钟。

```
mysql> select HOUR(CURTIME()),MINUTE(CURTIME());
+-----------------+-------------------+
| HOUR(CURTIME()) | MINUTE(CURTIME()) |
+-----------------+-------------------+
|              14 |                18 |
+-----------------+-------------------+
1 row in set (0.00 sec)
```

- MONTHNAME(date)函数：返回 date 的英文月份名称。

```
mysql> select MONTHNAME(now());
+------------------+
| MONTHNAME(now()) |
+------------------+
| July             |
+------------------+
1 row in set (0.00 sec)
```

- DATE_FORMAT(date,fmt)函数：按字符串 fmt 格式化日期 date 值，此函数能够按指定的格式显示日期，可以用到的格式符如表 5-4 所示。

表 5-4　　MySQL 中的日期时间格式

格式符	格式说明
%S,%s	两位数字形式的秒（00,01,...,59）
%i	两位数字形式的分（00,01,...,59）
%H	两位数字形式的小时，24 小时（00,01,...,23）
%h,%I	两位数字形式的小时，12 小时（01,02,...,12）
%k	数字形式的小时，24 小时（0,1,...,23）
%l	数字形式的小时，12 小时（1,2,...,12）
%T	24 小时的时间形式（hh:mm:ss）
%r	12 小时的时间形式（hh:mm:ssAM 或 hh:mm:ssPM）
%p	AM 或 PM
%W	一周中每一天的名称（Sunday,Monday,...,Saturday）
%a	一周中每一天名称的缩写（Sun,Mon,...,Sat）
%d	两位数字表示月中的天数（00,01,...,31）
%e	数字形式表示月中的天数（1,2，...,31）
%D	英文后缀表示月中的天数（1st,2nd,3rd,...）
%w	以数字形式表示周中的天数（0=Sunday,1=Monday,...,6=Saturday）
%j	以 3 位数字表示年中的天数（001,002,...,366）
%U	周（0,1,52），其中 Sunday 为周中的第一天
%u	周（0,1,52），其中 Monday 为周中的第一天
%M	月名（January,February,...,December）

%b	缩写的月名（January,February,...,December）
%m	两位数字表示的月份（01,02,...,12）
%c	数字表示的月份（1,2,...,12）
%Y	4 位数字表示的年份
%y	两位数字表示的年份
%%	直接值“%”

下面的例子将当前时间显示为“月，日，年”格式：

```
mysql> select DATE_FORMAT(now(),'%M,%D,%Y');
+------------------------------+
| DATE_FORMAT(now(),'%M,%D,%Y') |
+------------------------------+
| July,11th,2007               |
+------------------------------+
1 row in set (0.00 sec)
```

- DATE_ADD(date,INTERVAL expr type)函数：返回与所给日期 date 相差 INTERVAL 时间段的日期。

其中 INTERVAL 是间隔类型关键字，expr 是一个表达式，这个表达式对应后面的类型，type 是间隔类型，MySQL 提供了 13 种间隔类型，如表 5-5 所示。

表 5-5　　MySQL 中的日期间隔类型

表达式类型	描述	格式
HOUR	小时	hh
MINUTE	分	mm
SECOND	秒	ss
YEAR	年	YY
MONTH	月	MM
DAY	日	DD
YEAR_MONTH	年和月	YY-MM
DAY_HOUR	日和小时	DD hh
DAY_MINUTE	日和分钟	DD hh:mm
DAY_ SECOND	日和秒	DD hh:mm:ss
HOUR_MINUTE	小时和分	hh:mm
HOUR_SECOND	小时和秒	hh:ss
MINUTE_SECOND	分钟和秒	mm:ss

来看一个具体的例子，在这个例子中第 1 列返回了当前日期时间，第 2 列返回距离当前日期 31 天后的日期时间，第 3 列返回距离当前日期一年两个月后的日期时间。

```
mysql> select now() current,date_add(now(),INTERVAL  31 day) after31days,
date_add(now(),INTERVAL  '1_2' year_month) after_oneyear_twomonth;
+---------------------+---------------------+-----------------------+
| current             | after31days         | after_oneyear_twomonth |
+---------------------+---------------------+-----------------------+
| 2007-09-03 11:30:48 | 2007-10-04 11:30:48 | 2008-11-03 11:30:48    |
+---------------------+---------------------+-----------------------+
1 row in set (0.01 sec)
```

同样也可以用负数让它返回之前的某个日期时间，如下第 1 列返回了当前日期时间，第 2 列返回距离当前日期 31 天前的日期时间，第 3 列返回距离当前日期一年两个月前的日期时间。

```
mysql> select now() current,date_add(now(),INTERVAL  -31 day) after31days,date_a
dd(now(),INTERVAL  '-1_-2' year_month) after_oneyear_twomonth;
+---------------------+---------------------+-----------------------+
| current             | after31days         | after_oneyear_twomonth |
+---------------------+---------------------+-----------------------+
| 2007-09-03 11:36:35 | 2007-08-03 11:36:35 | 2006-07-03 11:36:35    |
+---------------------+---------------------+-----------------------+
1 row in set (0.00 sec)
```

- DATEDIFF（date1，date2）函数：用来计算两个日期之间相差的天数。
  下面的例子计算出当前距离 2008 年 8 月 8 日的奥运会开幕式还有多少天：

```
mysql> select DATEDIFF('2008-08-08',now());
+-------------------------------+
| DATEDIFF('2008-08-08',now()) |
+-------------------------------+
|                           328 |
+-------------------------------+
1 row in set (0.01 sec)
```

5.4 流程函数

流程函数也是很常用的一类函数，用户可以使用这类函数在一个 SQL 语句中实现条件选择，这样做能够提高语句的效率。表 5-6 列出了 MySQL 中跟条件选择有关的流程函数，下面将通过具体的实例来讲解每个函数的用法。

表 5-6　　MySQL 中的流程函数

函数	功能
IF(value,t f)	如果 value 是真，返回 t；否则返回 f
IFNULL(value1,value2)	如果 value1 不为空返回 value1，否则返回 value2
CASE WHEN [value1] THEN[result1]…ELSE[default]END	如果 value1 是真，返回 result1，否则返回 default
CASE [expr] WHEN [value1] THEN[result1]…ELSE[default]END	如果 expr 等于 value1，返回 result1，否则返回 default

下面的例子中模拟了对职员薪水进行分类，这里首先创建并初始化一个职员薪水表：

```
mysql> create table salary (userid int,salary decimal(9,2));
Query OK, 0 rows affected (0.06 sec)
```

插入一些测试数据：

```
mysql> insert into salary values(1,1000),(2,2000), (3,3000),(4,4000),(5,5000), (1,null);
Query OK, 6 rows affected (0.00 sec)
```

```
mysql> select * from salary;
+--------+---------+
| userid | salary  |
+--------+---------+
| 1      | 1000.00 |
| 2      | 2000.00 |
| 3      | 3000.00 |
| 4      | 4000.00 |
| 5      | 5000.00 |
| 1      | NULL    |
+--------+---------+
6 rows in set (0.00 sec)
```

接下来，通过这个表来介绍各个函数的应用。

- IF(value,t,f)函数：我们认为月薪在 2000 元以上的职员属于高薪，用“high”表示；而 2000 元以下的职员属于低薪，用“low”表示。

```
mysql> select if(salary>2000,'high','low') from salary;
+------------------------------+
| if(salary>2000,'high','low') |
+------------------------------+
| low                          |
| low                          |
| high                         |
| high                         |
| high                         |
+------------------------------+
5 rows in set (0.01 sec)
```

- IFNULL(value1,value2)函数：这个函数一般用来替换 NULL 值的，我们知道 NULL 值是不能参与数值运算的，下面这个语句就是把 NULL 值用 0 来替换。

```
mysql> select ifnull(salary,0) from salary;
+------------------+
| ifnull(salary,0) |
+------------------+
| 1000.00          |
| 2000.00          |
| 3000.00          |
| 4000.00          |
| 5000.00          |
| 0.00             |
+------------------+
6 rows in set (0.00 sec)
```

- CASE WHEN [value1] THEN[result1]…ELSE[default]END 函数：我们也可以用 case when…then 函数实现上面例子中高薪低薪的问题。

```
mysql> select case when salary<=2000 then 'low' else 'high' end from salary;
```

```
+-------------------------------------------------+
| case when salary<=2000 then 'low' else 'high' end |
+-------------------------------------------------+
| low                                             |
| low                                             |
| high                                            |
| high                                            |
| high                                            |
| high                                            |
+-------------------------------------------------+
6 rows in set (0.00 sec)
```

CASE [expr] WHEN [value1] THEN[result1]…ELSE[default]END 函数：这里还可以分多种情况把职员的薪水分多个档次，比如下面的例子分成高、中、低 3 种情况。同样还可以分成更多种情况，这里就不再举例了，有兴趣的读者可以自己测试一下。

```
mysql> select case salary when 1000 then 'low' when 2000 then 'mid' else 'high' end from salary;
+-----------------------------------------------------------------------+
| case salary when 1000 then 'low' when 2000 then 'mid' else 'high' end |
+-----------------------------------------------------------------------+
| low                                                                   |
| mid                                                                   |
| high                                                                  |
| high                                                                  |
| high                                                                  |
| high                                                                  |
+-----------------------------------------------------------------------+
6 rows in set (0.00 sec)
```

5.5 其他常用函数

MySQL 提供的函数很丰富，除了前面介绍的字符串函数、数字函数、日期函数、流程函数以外还有很多其他函数，在此不再一一列举，有兴趣的读者可以参考 MySQL 官方手册。表 5-7 列举了一些其他常用的函数。

表 5-7　　MySQL 中的其他常用函数

函数	功能
DATABASE()	返回当前数据库名
VERSION()	返回当前数据库版本
USER()	返回当前登录用户名
INET_ATON(IP)	返回 IP 地址的数字表示
INET_NTOA(num)	返回数字代表的 IP 地址
PASSWORD(str)	返回字符串 str 的加密版本
MD5()	返回字符串 str 的 MD5 值

下面结合实例简单介绍一下这些函数的用法。

- DATABASE()函数：返回当前数据库名。

```
mysql> select DATABASE();
+------------+
| DATABASE() |
+------------+
| test       |
+------------+
1 row in set (0.00 sec)
```

- VERSION()函数：返回当前数据库版本。

```
mysql> select VERSION();
+-----------+
| VERSION() |
+-----------+
| 5.0.18-nt |
+-----------+
1 row in set (0.00 sec)
```

- USER()函数：返回当前登录用户名。

```
mysql> select USER();
+----------------+
| USER()         |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.03 sec)
```

- INET_ATON(IP)函数：返回 IP 地址的网络字节序表示。

```
mysql> select INET_ATON('192.168.1.1');
+--------------------------+
| INET_ATON('192.168.1.1') |
+--------------------------+
|               3232235777 |
+--------------------------+
1 row in set (0.00 sec)
```

- INET_NTOA(num)函数：返回网络字节序代表的 IP 地址。

```
mysql> select INET_NTOA(3232235777);
+-----------------------+
| INET_NTOA(3232235777) |
+-----------------------+
| 192.168.1.1           |
+-----------------------+
1 row in set (0.00 sec)
```

INET_ATON(IP)和 INET_NTOA(num)函数主要的用途是将字符串的 IP 地址转换为数字表示的网络字节序，这样可以更方便地进行 IP 或者网段的比较。比如在下面的表 t 中，要想知道在

“192.168.1.3”和“192.168.1.20”之间一共有多少 IP 地址：

```
mysql> select * from t;
+--------------+
| ip           |
+--------------+
| 192.168.1.1  |
| 192.168.1.3  |
| 192.168.1.6  |
| 192.168.1.10 |
| 192.168.1.20 |
| 192.168.1.30 |
+--------------+
6 rows in set (0.00 sec)
```

按照正常的思维，应该用字符串来进行比较，下面是字符串的比较结果：

```
mysql> select * from t where ip>='192.168.1.3' and ip<='192.168.1.20';
Empty set (0.01 sec)
```

结果没有如我们所愿，竟然是个空集。其实原因就在于字符串的比较是一个字符一个字符的比较，当对应字符相同时候，就比较下一个，直到遇到能区分出大小的字符，才停止比较，后面的字符也将忽略。显然，在此例中，“192.168.1.3”其实比“192.168.1.20”要“大”，因为“3”比“2”大，而不能用我们日常的思维 3<20，所以“ip>='192.168.1.3' and ip<='192.168.1.20'”必然是个空集。

在这里，如果要想实现上面的功能，就可用函数 INET_ATON 来实现，将 IP 转换为字节序后再比较，如下所示：

```
mysql> select * from t where inet_aton(ip)>=inet_aton('192.168.1.3') and inet_aton(ip)<=inet_aton('192.168.1.20');
+--------------+
| ip           |
+--------------+
| 192.168.1.3  |
| 192.168.1.6  |
| 192.168.1.10 |
| 192.168.1.20 |
+--------------+
4 rows in set (0.00 sec)
```

结果完全符合我们的要求。

- PASSWORD(str)函数：返回字符串 str 的加密版本，一个 41 位长的字符串。

此函数只用来设置系统用户的密码，但是不能用来对应用的数据加密。如果应用方面有加密的需求，可以使用 MD5 等加密函数来实现。

下例中显示了字符串“123456”的 PASSWORD 加密后的值：

```
mysql> select PASSWORD('123456');
+-------------------------------------------+
| PASSWORD('123456')                        |
+-------------------------------------------+
| *6BB4837EB74329105EE4568DDA7DC67ED2CA2AD9 |
```

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

```
+------------------------------------------+
1 row in set (0.08 sec)
```

- MD5(str)函数：返回字符串 str 的 MD5 值，常用来对应用中的数据进行加密。

下例中显示了字符串“123456”的 MD5 值：

```
mysql> select MD5('123456');
+----------------------------------+
| MD5('123456')                    |
+----------------------------------+
| e10adc3949ba59abbe56e057f20f883e |
+----------------------------------+
1 row in set (0.06 sec)
```

5.6 小结

本章主要对 MySQL 常用的各类常用函数通过实例做了介绍。MySQL 有很多内建函数，这些内建函数实现了很多应用需要的功能并且拥有很好的性能，如果用户在工作中需要实现某种功能，最好先查一下 MySQL 官方文档或者帮助，看是否已经有相应的函数实现了我们需要的功能，可以大大提高工作效率。由于篇幅所限，本章并没有介绍所有的函数，读者可以去进一步查询相关文档。

第6章　图形化工具的使用

在日常各种工作中，图形化工具为用户提供了很大的便利。针对 MySQL 数据库，很多公司也都开发了自己的图形化工具。本章将主要介绍 MySQL 公司开发的 Administrator 管理工具和 Query Brower 查询工具，以及 phpMyAdmin 开发组开发的 Web 管理工具 phpMyAdmin。前者是非常流行的 C/S 客户端管理工具，用户可以从 MySQL 的官方网站（http://dev.mysql.com/downloads/gui-tools/5.0.html）下载最新的版本；后者是非常流行的 Web 管理工具，用户可以从 phpMyAdmin 开发组的网站（http://www.phpmyadmin.net/home_page/ index.php）下载到它。下载后它们的安装都非常简单，本章不再为此详细介绍，而重点介绍这几个工具的常用功能。

6.1 MySQL Administrator

MySQL Administrator 是 MySQL 为 4.0 以上版本数据库提供的可视化界面的 MySQL 数据库管理控制台，可以方便地管理和操作 MySQL 数据库。提供的功能包括启动关闭数据库、用户管理、参数配置、数据库对象管理、备份恢复管理等。

管理控制台提供的功能也可以通过 Mysqladmin 或者 MySQL 实现，但是 Administrator 以其图形化操作的直观性和简便性受到 MySQL 数据库使用者的广泛好评。

通过 MySQL GUI Tools 5.0\MySQLAdministrator.exe 进入登录界面，如图 6-1 所示，正确录入

服务器的 IP 地址、端口号、用户名和口令，即可以登录数据库。

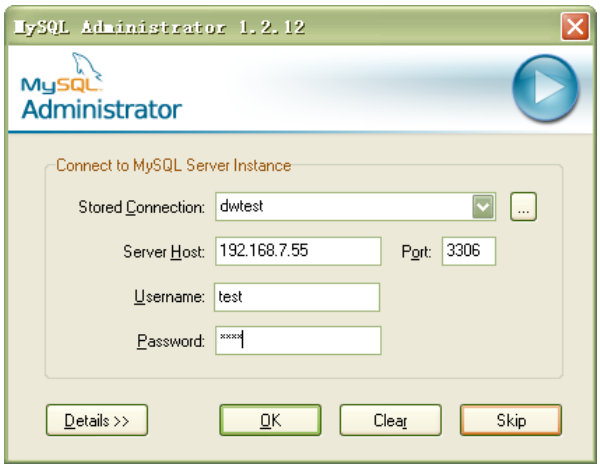


图 6-1　MySQL Administrator 登录界面

管理控制台上的部分功能只能用于管理本地数据库，对于远程数据库这些选项将不可用。例如，服务控制、参数配置和日志查询等。

下面的章节会着重介绍管理控制台的几个常用功能，DBA 日常的管理和维护工作都可以使用这些功能轻松完成。

6.1.1 连接管理

连接管理可以用来查看当前活跃的数据库连接，这与 SHOW PROCESSLIST 命令的执行结果相同。由于 SHOW PROCESSLIST 命令不能指定查看某个 PID 或者某个用户的信息，当数据库连接非常多的时候，执行 SHOW PROCESSLIST 命令就会返回很多条结果，不便查找问题。但管理控制台可以方便地拖动滚动条找到用户关心的 PID 的信息，或者通过对用户的分组快速定位到数据库连接的信息，如图 6-2 所示。

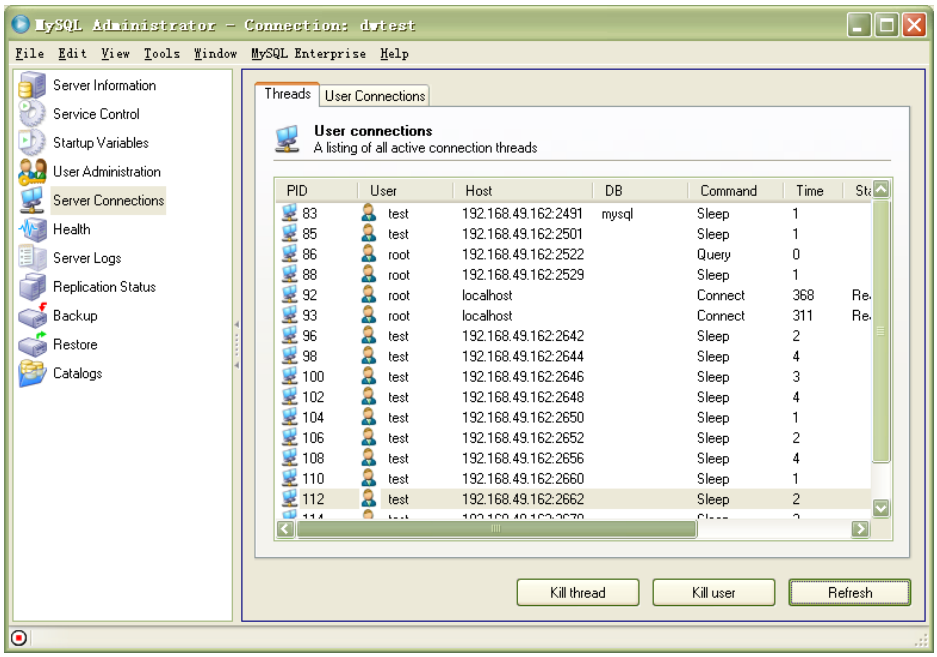


图 6-2　MySQLAdministrator 连接管理

如果当前连接的用户有 PROCESS 权限，那么可以查看全部的线程；如果当前连接的用户有

SUPER 权限，那么该用户还可以通过单击窗口下面的“Kill thread”按钮杀掉指定的线程，或者通过单击“Kill user”按钮杀掉指定用户的全部线程。

> 注意：通常，数据库管理员可以使用连接管理中的连接时间、当前的状态和 INFO 等信息来查找当前数据库可能存在的问题，并通过 Kill thread 或者 Kill user 将可疑的线程杀掉以解决错误操作带来的相关问题。

6.1.2 健康检查

健康检查是 DBA 日常监控数据库使用情况的图形化工具，可以随时了解数据库连接的变化情况、SQL 查询执行的数量、缓冲区的命中率等。另外，DBA 还可以通过 Status Variables 和 System Variables 查看系统的变量以了解应用的相关情况。例如，通过查看 Commands Executed 可以得到 SELECT、INSERT、UPDATE、DELETE 操作执行的比例，由此可以大致判断应用的类型是以查询操作为主，还是以更新操作为主。

如图 6-3 所示是健康检查中与连接相关的信息的展示。

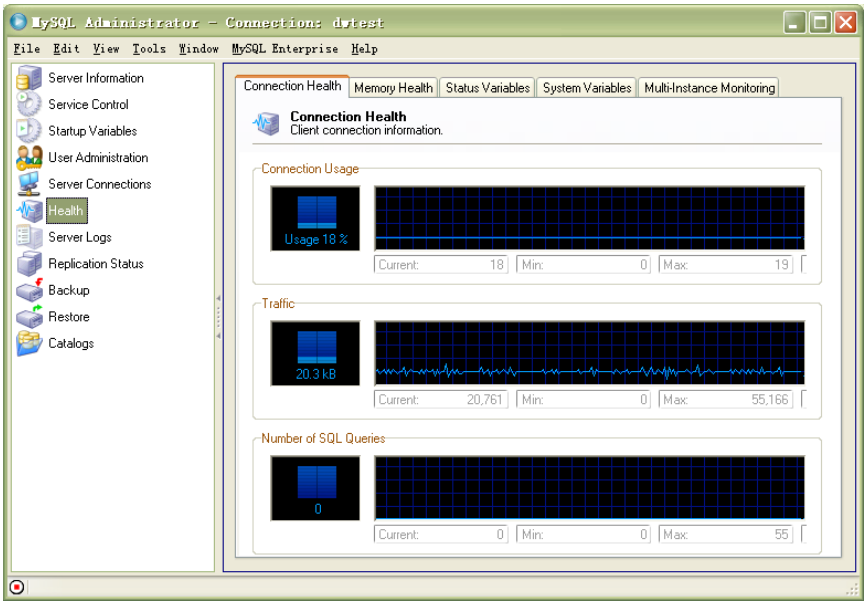


图 6-3　MySQL Administrator 健康检查-连接信息

健康检查还提供一个非常灵活的功能，允许用户自己定义监控的变量，当用户觉得工具本身提供的这些监控参数不能满足自己的要求时，可以自己增加一个 Page，并在其上面添加自己的 Group 和 Graph。

例如，可以增加一个 Test Page，在增加 Graph 的窗口里，只需要定义需要监控的 VARIABLE 是什么（可以是单个变量的值，也可以是几个变量的计算结果），确定后就可以在 Test Page 中看到参数实时变化的情况。在如图 6-4 所示的例子中，就定义查看 InnoDB 表 INSERT、UPDATE、DELETE、SELECT 这 4 个变量的变化情况。

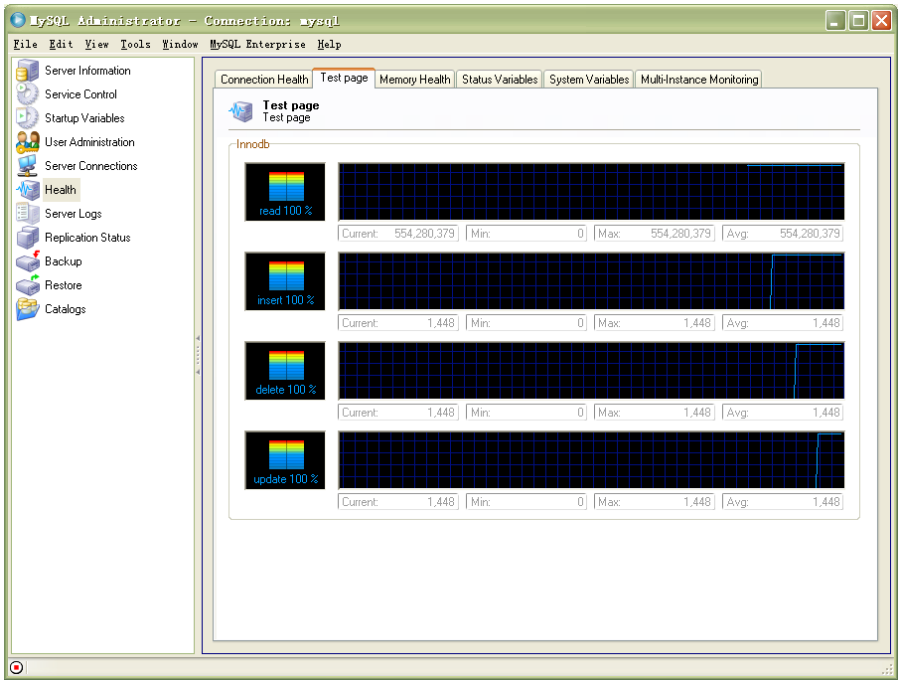


图 6-4　MySQL Administrator 健康检查-自定义监控内容

> 注意：DBA 需要对数据库正常情况下的连接情况和负载使用情况比较熟悉，这样一旦发现异常的连接数过高或者命中率明显下降，就可以马上做出响应并着手查找问题的原因。

6.1.3 备份管理

管理控制台提供的备份管理比较灵活，操作也很简单。这个工具提供了与 Mysqldump 工具相似的功能，在本书的第 30 章中将会对 MySQL 数据库的备份进行详细的介绍，但是那里提到的备份方法都需要以命令行方式执行，这就需要数据库管理员对命令行有一定的了解。图 6-5 显示的是备份管理的高级选项界面，在创建了备份项目，指定了需要备份的数据库和表之后，可以在这个界面上进行更详细的配置，包括备份执行的方法和输出文件的选项，例如，是否需要进行全备份、备份的类型等。

下面分别对备份执行方法的选项进行说明。

- InnoDB Online Backup：为了确保 InnoDB 表备份结果的数据的一致性，会在备份开始的时候启动一个事务，推荐只在备份 InnoDB 类型的表时使用。
- Lock all tables：为了确保 MyISAM 表备份结果的数据的一致性，会在备份开始时对本次要备份的表执行 Lock 操作，防止新的数据写入。这样在备份期间会阻塞表的更新，对于更新密集型的应用要谨慎选择使用这个选项。
- Online with binlog pos：除了实现和 InnoDB Online Backup 相同的功能外，还记录了当前 Binlog 的位置，便于恢复时知道需要恢复的 Binlog 的起点。
- Normal backup：只在备份每个表的时候才锁定该表，这种情况下对应用的影响是最小的，但是相应的这种备份方法生成的备份结果中，表间的数据一致性是最没有保障的，选择这种备份方式前要考虑到这个问题，通过选择合适的时间执行备份操作来减小备份数据不一致的风险。
- Complete backup：选定数据库的全备份，而忽视前面选择的表的列表。这种方式对于数据库中的表定期增加的情况非常有用，省去每次修改备份列表的工作。

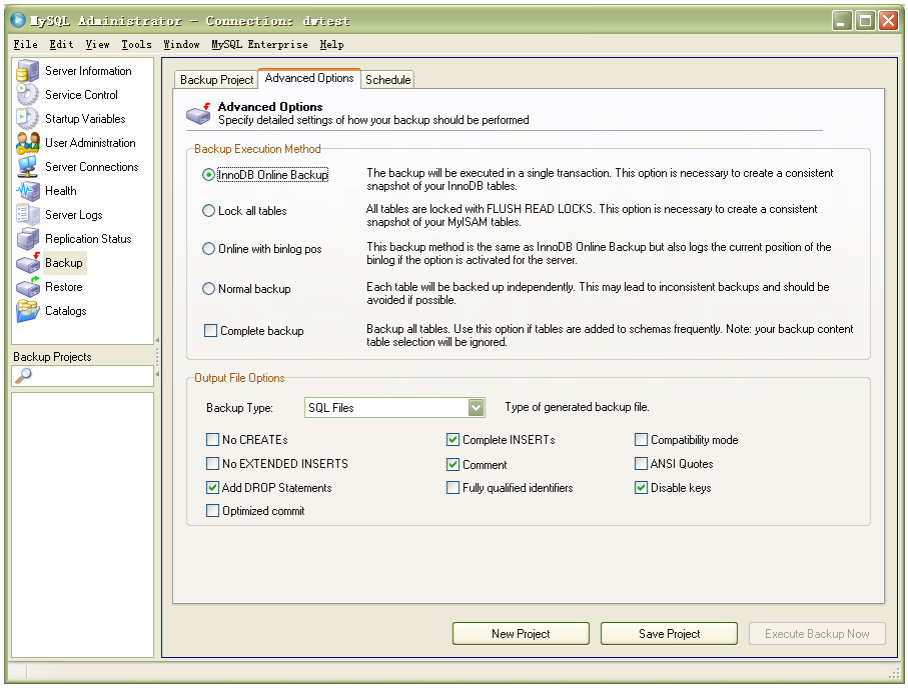


图 6-5　MySQL Administrator 备份管理

定义的备份项目可以马上执行，也可以定义成任务定期执行。

> 注意：控制台的备份管理提供的选项比命令行少很多，例如不能指定导出字符集，不能指定导出记录的 WHERE 条件，也不能按照指定的分隔符和换行符导出数据成文本文件。所以如果用户需要更高级的数据备份功能，还是要熟练掌握 mysqldump 的各个命令行参数。

6.1.4 Catalogs

控制台提供的管理表、索引、视图和过程的工具。可以用来查询和修改已有的数据库对象或者创建新的数据库对象，取决于连接的用户所拥有的权限。

图 6-6 显示的是指定数据库的全部表的基本信息和指定表的一些更详细的情况，表的基本信息除了包括表名、存储引擎、行数以外，还比较形象地显示了数据的大小和索引的大小，比使用 SHOW TABLE STATUS 命令行显示的结果更加直观。

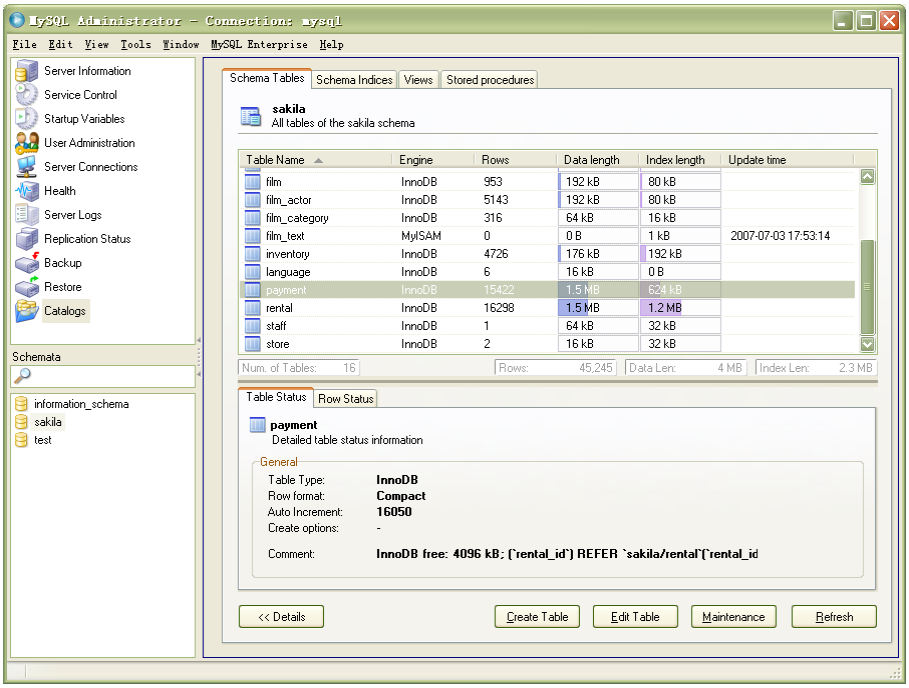


图 6-6　MySQL Administrator Catalogs

通过使用窗口下方的“Create Table”按钮和“Edit Table”按钮可以创建新表或者修改指定表的结构，通过“Maintenance”按钮可以对指定表进行检查和修复的操作。在指定表上单击右键，在弹出的菜单中还可以选择“Drop Table”或者“Edit Table Data”命令，其中“Edit Table Data”命令会自动激活 MySQL Query Brower 程序进行数据的查询和编辑，关于 MySQL Query Brower 的内容将会在后面的章节中介绍。

> 注意：目录管理提供的表的相关功能中，对分区的支持还不是很好，不论是新创建一个分区表，还是修改已有的分区表，都没有可以指定分区特性的地方，所以如果你需要创建和维护分区表，那么暂时还只能通过命令行来创建，相信很快就会有支持分区功能的版本发布了。

6.2 MySQL Query Brower

MySQL Query Brower 也是 MySQL 为 4.0 以上版本数据库提供的可视化界面的 MySQL 数据库管理工具，和 MySQL Administrator 不同，Query Brower 主要用于管理数据库中保存的数据，而 Administrator 主要用于管理数据库的对象。

MySQL Query Brower 提供的工具包括数据的查询、更新和 SQL 的优化分析，虽然对于 MySQL Query Brower 提供的大多数功能，用字符界面的 MySQL 客户端都可以实现，但是相比之下，Query Brower 返回的结果更直观，在查询的列非常多的时候不会因为内容换行而使结果看起来很让人困惑。此外，对于查询的返回结果，可以选择导出成 XML、HTML、EXCEL、CVS 等多种格式，相比命令行的方式更灵活和方便。

MySQL Query Brower 的登录界面和 MySQL Administrator 基本相同，这里不再复述，下面简单介绍 MySQL Query Brower 的一些简单功能。

MySQL Query Brower 的主要查询窗口，如图 6-7 所示。

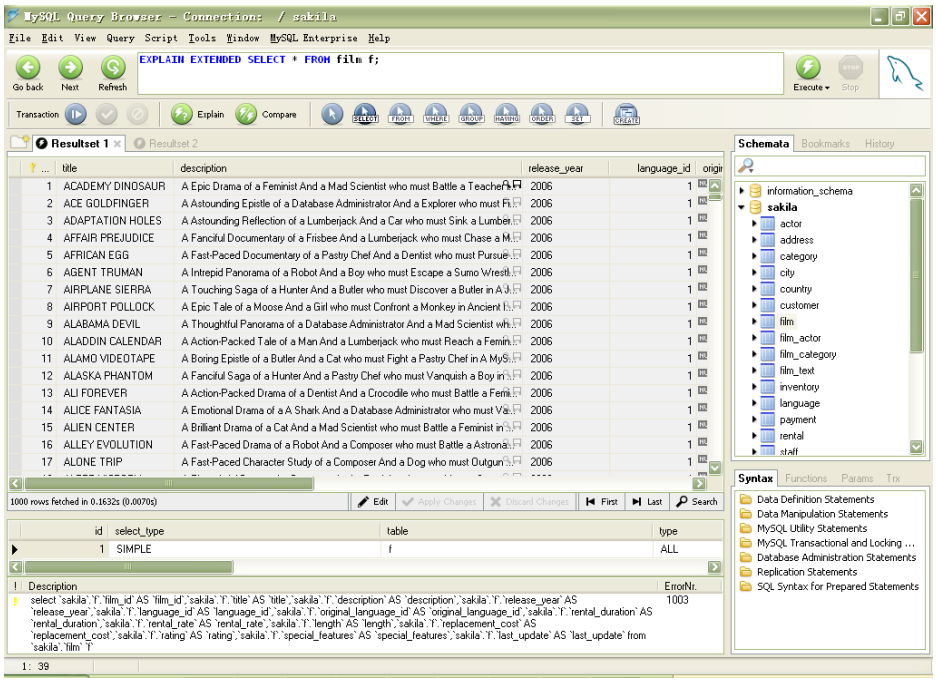


图 6-7　MySQL Query Brower

主查询窗口可以划分为以下几个区域。

- 查询工具栏：窗口的最上方，用来创建和执行查询，可以通过“Go back”、“Next”或“Refresh”按钮切换不同的 SQL 语句，“Execute”按钮用来执行查询，“Stop”按钮用来中止正在执行的查询。
- 高级工具栏：查询工具栏的下方是高级工具栏，可以用来进行事务的控制，生成查询的执行计划或者进行比较，以及一些查询生成的工具。
- 查询结果区：窗口的中间区域是查询结果区，用来显示查询返回的结果，可以显示多个 Tab 页，也可以同时执行多个查询。
- 对象浏览区：在窗口的右边是对象浏览区，可以在这里查看数据库和数据库对象的信息，单击右键还可以创建新的数据库对象或者对选中的数据库对象进行修改，操作和 MySQL Administrator 相似。
- 语法查询区：在窗口的右下方，可以查询到非常详细的语法信息。

6.3 phpMyAdmin

phpMyAdmin（简称 PMA），是一个用 PHP 编写的、可以通过 Web 控制和操作 MySQL 数据库的工具。其最突出的特点是可以直接从 Web 上去管理 MySQL，不需要直接在 MySQL 数据库服务器上进行维护。因其功能全面、使用方便成为众多 MySQL 数据库管理员维护数据库的首选工具。

phpMyAdmin 的功能非常全面，包括数据库管理、数据对象管理、用户管理、数据导入导出、数据库管理、数据管理等，下面将对几个重要的管理功能进行详细介绍。

6.3.1 数据库管理

登录后进入 phpMyAdmin 的主页面，在主页面中 phpMyAdmin 列出了当前数据库的一些基本信息，包括数据库版本、连接方式、连接用户、server 的字符集等，可以在主页面选择创

建一个新的数据库，或者在窗口的右边下拉框中选择一个已经存在的数据库。在窗口的下方，列出一些常用管理功能，其中包括进程管理、用户管理、数据导入导出、存储引擎管理等。图 6-8 显示的是在主页面中创建一个新数据库：首先输入要创建的数据库名字，选择数据库的字符集，然后，单击“创建”按钮即可成功创建一个新的数据库。

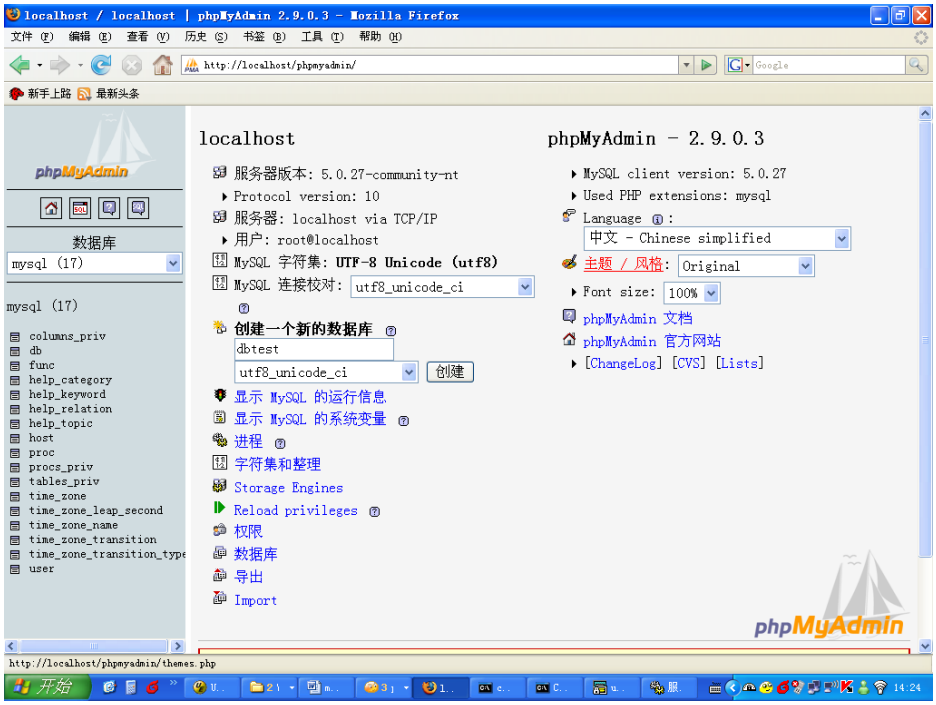


图 6-8　phpMyAdmin 主页面

从图 6-8 可以看到，在窗口的左边因为选择了 MySQL 数据库，可以看到下拉框下方列出了 MySQL 数据库中的数据库对象，(17)表示 MySQL 数据库中包含的数据库对象的数量。如果需要查看数据库对象的详细情况或者数据库对象保存的数据，则可以直接选择该数据库对象，打开对应的数据库对象的窗口。

数据库的创建和选择都可以在主页面上完成，如果需要删除数据库，则需要进入数据库对象管理的页面中，在菜单的最后一项选择 DROP 命令可以删除当前选择的数据库。注意，删除数据库操作会删除该数据库包含的所有数据库对象，删除之前最好确保已有备份。

6.3.2 数据库对象管理

在主页面完成创建数据库的操作或者在主页面选择已经存在的数据库后，可以进入表的管理、维护界面，进行表的创建、更新、删除等管理维护工作。

图 6-9 显示了一个标准的数据库对象管理页面，在窗口的左边选择了表 test，那么在窗口的右边就可以看到该表的表结构、索引情况、空间使用的情况等详细信息，同时可以对表的结构、索引进行修改。

这些操作都和实际执行命令行操作所完成的功能是相同的，不同的是使用图形界面可以大大方便修改的过程，屏蔽因为语法错误带来的相关问题。

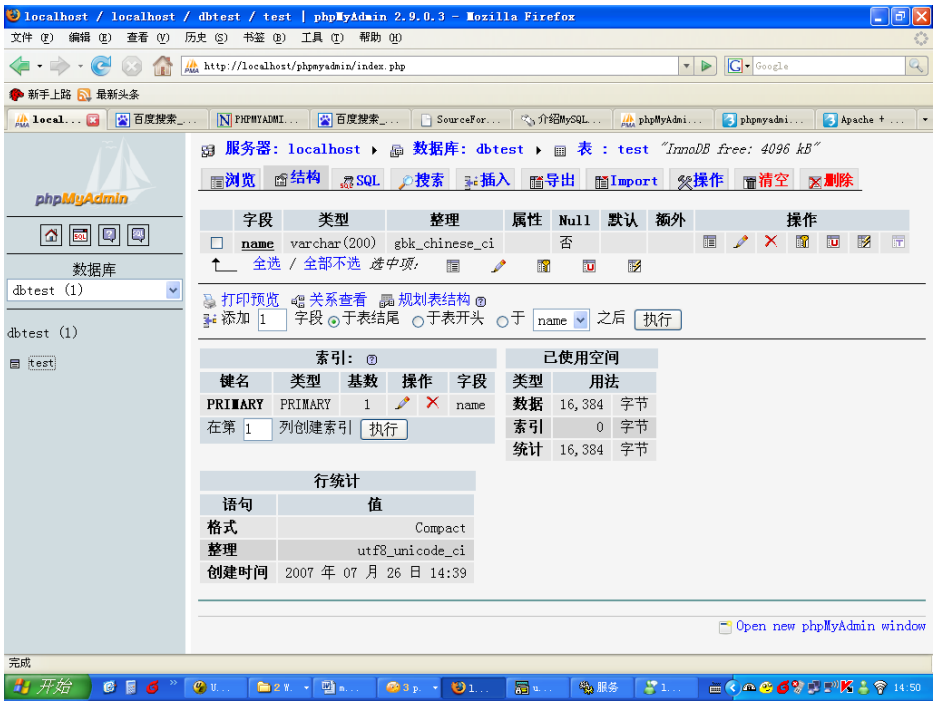


图 6-9　phpMyAdmin 表管理维护页面

在页面的最上方，可以看到所有数据库对象管理可以完成的操作，包括直接执行 SQL 语句、插入记录、导出导入数据、表分析表检查、清空表、删除表等，这里不再对这些页面进行逐一介绍，建议读者对每个功能都进行简单的测试，便于在以后的开发维护中能够更加熟练地使用这些功能。

6.3.3 权限管理

在主页面中单击“权限”链接即可以进入权限管理界面。在权限管理功能中，phpMyAdmin 实现了添加用户、删除用户、对新老用户权限进行修改和设置等功能。

图 6-10 显示的是 phpMyAdmin 添加新用户时的操作界面，使用者可以在添加用户时设置密码，分配各项数据库权限，指定用户可以访问的数据库等信息。另外在窗口的最下方，还可以设置每个用户每个小时最多的查询、更新、连接的次数等，便于数据库管理员在多用户的数据库中合理地分配数据库资源。

图 6-10　phpMyAdmin 权限管理

6.3.4 导入导出数据

数据的导入导出是数据库管理工具一项非常重要的功能，phpMyAdmin 提供的导入导出功能也比较完善，支持导出成 CSV、Excel 2000、Text、PDF、SQL 等多种格式，SQL 兼容性允许导出其他数据库语法的 SQL 语句，支持的数据库包括 DB2、Oracle、SQL Server、MaxDB、PostgreSQL 等，为数据在异构数据库间的迁移提供了便利。

进入主页后单击“导出”按钮后，进入数据导出页面，导出页面如图 6-11 所示，使用者在导出数据时，需要先选择导出的数据库，再选择导出的数据格式，然后根据导出数据格式设置相应的选项，最后单击“执行”按钮完成数据导出。

图 6-11　phpMyAdmin 数据库的转存

导入数据的操作也非常简单。从主页进入导入数据管理界面后，在图 6-12 中单击“浏览”按钮，选择用户要导入的文件，然后单击“执行”按钮，即可完成导入操作。

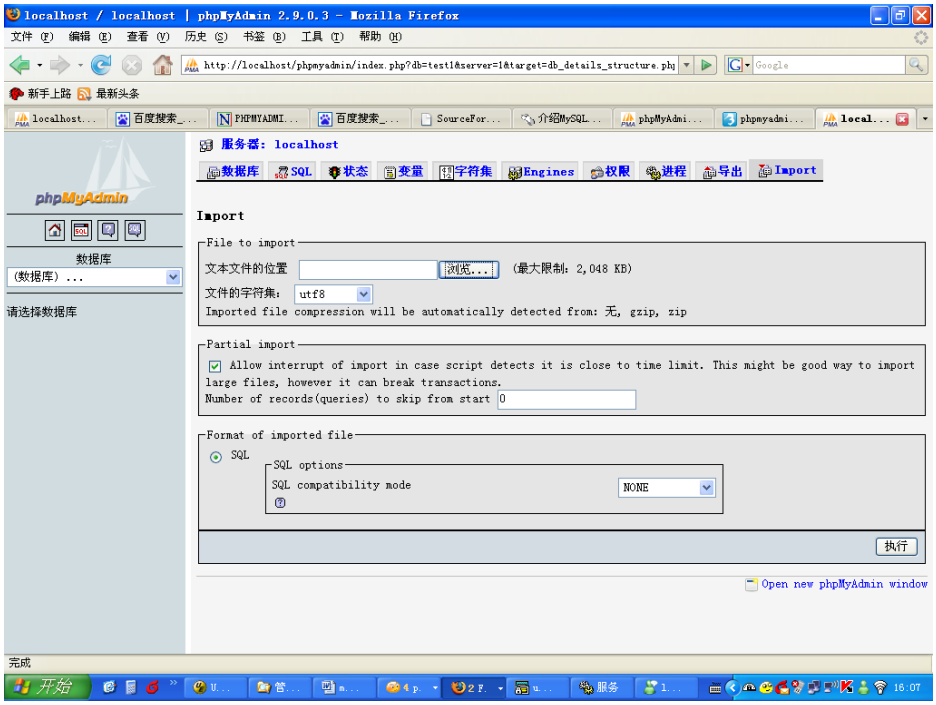


图 6-12　phpMyAdmin 导入数据

导入成功后会出现导入是否成功的详细信息，导入结束后的提示界面如图 6-13 所示。

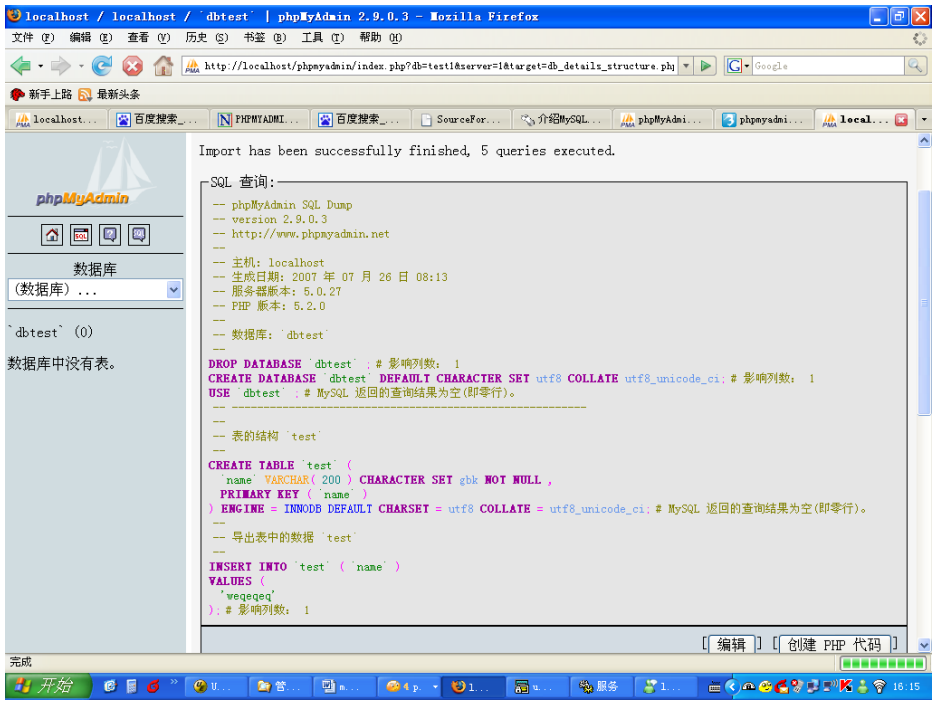


图 6-13　phpMyAdmin 导入成功显示

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

6.4 小结

本章主要介绍了 MySQL 常用的几种图形化工具。

MySQL Administrator 是 MySQL 公司提供了图形化管理工具，主要用于 MySQL 数据库的连接管理、健康检查、备份管理、Catalogs 管理等，使用此工具可以实现远程图形化管理服务器，应用比较方便。

MySQL Query Brower 是 MySQL 公司提供的客户端查询工具，开发和管理人员可以使用它进行数据库直接访问和统计，同时可以将查询结果在客户端导出保存。

phpMyAdmin 不是 MySQL 公司的产品，它的特点是可以通过 Web 方式对 MySQL 数据库进行管理，并且它的功能也比较强大，许多网站管理员都会选择使用它管理自己的网站数据库。

这几种工具各有侧重，可以根据需要使用选择最适合自己的工具，提高数据库开发和维护的效率。图形工具通常只能完成相对简单的工作，对于一些复杂数据库的操作，还是需要通过命令行来执行，在后面章节中所介绍的操作大部分都是通过命令行来完成了。大家在享受图形工具的便利时，也要尽量熟悉命令行的相关操作，减少对图形工具的依赖。

第2篇　开发篇

第7章　表类型（存储引擎）的选择

和大多数数据库不同，MySQL 中有一个存储引擎的概念，针对不同的存储需求可以选择最优的存储引擎。本章将详细介绍存储引擎的概念、分类以及实际应用中的选择原则。

7.1 MySQL 存储引擎概述

插件式存储引擎是 MySQL 数据库最重要的特性之一，用户可以根据应用的需要选择如何存储和索引数据、是否使用事务等。MySQL 默认支持多种存储引擎，以适用于不同领域的数据库应用需要，用户可以通过选择使用不同的存储引擎提高应用的效率，提供灵活的存储，用户甚至可以按照自己的需要定制和使用自己的存储引擎，以实现最大程度的可定制性。

MySQL 5.0 支持的存储引擎包括 MyISAM、InnoDB、BDB、MEMORY、MERGE、EXAMPLE、NDB Cluster、ARCHIVE、CSV、BLACKHOLE、FEDERATED 等，其中 InnoDB 和 BDB 提供事务安全表，其他存储引擎都是非事务安全表。

默认情况下，创建新表不指定表的存储引擎，则新表是默认存储引擎的，如果需要修改默认的存储引擎，则可以在参数文件中设置 default-table-type。查看当前的默认存储引擎，可以使用以下命令：

```
mysql> show variables like 'table_type';
+---------------+--------+
| Variable_name | Value  |
+---------------+--------+
| table_type    | MyISAM |
```

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

可以通过下面两种方法查询当前数据库支持的存储引擎，第一种方法为：

```
mysql> SHOW ENGINES \G
*************************** 1. row ***************************
      Engine: MyISAM
     Support: DEFAULT
     Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: InnoDB
     Support: YES
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 5. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
5 rows in set (0.00 sec)
```

或者采用第二种方法：

```
mysql> SHOW VARIABLES LIKE 'have%';
+---------------------------+-------+
| Variable_name             | Value |
```

```
+---------------------------+-------+
| have_archive              | NO    |
| have_bdb                  | NO    |
| have_blackhole_engine     | NO    |
| have_compress             | YES   |
| have_crypt                | YES   |
| have_csv                  | YES   |
| have_dlopen               | YES   |
| have_example_engine       | NO    |
| have_federated_engine     | NO    |
| have_geometry             | YES   |
| have_innodb               | YES   |
| have_ndbcluster           | NO    |
| have_openssl              | NO    |
| have_partitioning         | YES   |
| have_query_cache          | YES   |
| have_row_based_replication | YES   |
| have_rtree_keys           | YES   |
| have_symlink              | YES   |
+---------------------------+-------+
18 rows in set (0.00 sec)
```

以上两种方法都可以用来查看当前支持哪些存储引擎，其中 Value 显示为“DISABLED”的记录表示支持该存储引擎，但是数据库启动的时候被禁用。

在创建新表的时候，可以通过增加 ENGINE 关键字设置新建表的存储引擎，例如，在下面的例子中，表 ai 就是 MyISAM 存储引擎的，而 country 表就是 InnoDB 存储引擎的：

```
CREATE TABLE ai (
  i bigint(20) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (i)
) ENGINE=MyISAM DEFAULT CHARSET=gbk;

CREATE TABLE country (
  country_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  country VARCHAR(50) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (country_id)
)ENGINE=InnoDB DEFAULT CHARSET=gbk;
```

也可以使用 ALTER TABLE 语句，将一个已经存在的表修改成其他的存储引擎。下面的例子介绍了如何将表 ai 从 MyISAM 存储引擎修改成 InnoDB 存储引擎：

```
mysql> alter table ai engine = innodb;
Query OK, 0 rows affected (0.13 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> show create table ai \G
```

```
*************************** 1. row ***************************
       Table: ai
Create Table: CREATE TABLE `ai` (
  `i` bigint(20) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`i`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk
1 row in set (0.00 sec)
```

这样修改后，ai 表成为 InnoDB 存储引擎，可以使用 InnoDB 存储引擎的相关特性。

7.2 各种存储引擎的特性

下面重点介绍几种常用的存储引擎,并对比各个存储引擎之间的区别，以帮助读者理解不同存储引擎的使用方式。

表 7-1　　常用存储引擎的对比

特点	MyISAM	InnoDB	MEMORY	MERGE	NDB
存储限制	有	64TB	有	没有	有
事务安全		支持			
锁机制	表锁	行锁	表锁	表锁	行锁
B 树索引	支持	支持	支持	支持	支持
哈希索引			支持		支持
全文索引	支持				
集群索引		支持			
数据缓存		支持	支持		支持
索引缓存	支持	支持	支持	支持	支持
数据可压缩	支持				
空间使用	低	高	N/A	低	低
内存使用	低	高	中等	低	高
批量插入的速度	高	低	高	高	高
支持外键		支持			

下面将重点介绍最常使用的 4 种存储引擎：MyISAM、InnoDB、MEMORY 和 MERGE。NDB 存储引擎会在第 33 章 MySQL CLUSTER 中详细介绍，这里不再赘述。

7.2.1 MyISAM

MyISAM 是 MySQL 的默认存储引擎。MyISAM 不支持事务、也不支持外键，其优势是访问的速度快，对事务完整性没有要求或者以 SELECT、INSERT 为主的应用基本上都可以使用这个引擎来创建表。

每个 MyISAM 在磁盘上存储成 3 个文件，其文件名都和表名相同，但扩展名分别是：

- .frm（存储表定义）；
- .MYD（MYData，存储数据）；
- .MYI （MYIndex，存储索引）。

数据文件和索引文件可以放置在不同的目录，平均分布 IO，获得更快的速度。

要指定索引文件和数据文件的路径，需要在创建表的时候通过 DATA DIRECTORY 和 INDEX DIRECTORY 语句指定，也就是说不同 MyISAM 表的索引文件和数据文件可以放置到不同的路径下。文件路径需要是绝对路径，并且具有访问权限。

MyISAM 类型的表可能会损坏，原因可能是多种多样的，损坏后的表可能不能访问，会提示需要修复或者访问后返回错误的结果。MyISAM 类型的表提供修复的工具，可以用 CHECK TABLE 语句来检查 MyISAM 表的健康，并用 REPAIR TABLE 语句修复一个损坏的 MyISAM 表。表损坏可能导致数据库异常重新启动，需要尽快修复并尽可能地确认损坏的原因。具体的操作步骤可以参见第 35 章应急处理。

MyISAM 的表又支持 3 种不同的存储格式，分别是：

- 静态（固定长度）表；
- 动态表；
- 压缩表。

其中，静态表是默认的存储格式。静态表中的字段都是非变长字段，这样每个记录都是固定长度的，这种存储方式的优点是存储非常迅速，容易缓存，出现故障容易恢复；缺点是占用的空间通常比动态表多。静态表的数据在存储的时候会按照列的宽度定义补足空格，但是在应用访问的时候并不会得到这些空格，这些空格在返回给应用之前已经去掉。

但是也有些需要特别注意的问题，如果需要保存的内容后面本来就带有空格，那么在返回结果的时候也会被去掉，开发人员在编写程序的时候需要特别注意，因为静态表是默认的存储格式，开发人员可能并没有意识到这一点，从而丢失了尾部的空格。以下例子演示了插入的记录包含空格时处理的情况：

```
mysql> create table Myisam_char (name char(10)) engine=myisam;
Query OK, 0 rows affected (0.04 sec)

mysql> insert into Myisam_char values('abcde'),('abcde  '),('  abcde'),('  abcde  ');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> select name,length(name) from Myisam_char;
+---------+--------------+
| name    | length(name) |
+---------+--------------+
| abcde   | 5            |
| abcde   | 5            |
|   abcde | 7            |
|   abcde | 7            |
+---------+--------------+
4 rows in set (0.00 sec)
```

从上面的例子可以看出，插入记录后面的空格都被去掉了，前面的空格保留。

动态表中包含变长字段，记录不是固定长度的，这样存储的优点是占用的空间相对较少，但是频繁地更新删除记录会产生碎片，需要定期执行 OPTIMIZE TABLE 语句或 myisamchk -r 命令来改善性能，并且出现故障的时候恢复相对比较困难。

压缩表由 myisampack 工具创建，占据非常小的磁盘空间。因为每个记录是被单独压缩的，

所以只有非常小的访问开支。

7.2.2 InnoDB

InnoDB 存储引擎提供了具有提交、回滚和崩溃恢复能力的事务安全。但是对比 MyISAM 的存储引擎，InnoDB 写的处理效率差一些并且会占用更多的磁盘空间以保留数据和索引。

下面将重点介绍 InnoDB 存储引擎的表使用过程中不同于其他存储引擎的特点。

1、自动增长列

InnoDB 表的自动增长列可以手工插入，但是插入的值如果是空或者 0，则实际插入的将是自动增长后的值。下面定义新表 autoincre_demo，其中列 i 使用自动增长列，对该表插入记录，然后查看自动增长列的处理情况，可以发现插入 0 或者空实际插入的都将是自动增长后的值：

```
mysql> create table autoincre_demo
    -> (i smallint not null auto_increment,
    -> name varchar(10),primary key(i)
    -> )engine=innodb;
Query OK, 0 rows affected (0.13 sec)

mysql> insert into autoincre_demo values(1,'1'),(0,'2'),(null,'3');
Query OK, 3 rows affected (0.04 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from autoincre_demo;
+---+------+
| i | name |
+---+------+
| 1 | 1    |
| 2 | 2    |
| 3 | 3    |
+---+------+
3 rows in set (0.00 sec)
```

可以通过“ALTER TABLE *** AUTO_INCREMENT = n;”语句强制设置自动增长列的初识值，默认从 1 开始，但是该强制的默认值是保留在内存中的，如果该值在使用之前数据库重新启动，那么这个强制的默认值就会丢失，就需要在数据库启动以后重新设置。

可以使用 LAST_INSERT_ID()查询当前线程最后插入记录使用的值。如果一次插入了多条记录，那么返回的是第一条记录使用的自动增长值。下面的例子演示了使用 LAST_INSERT_ID()的情况：

```
mysql> insert into autoincre_demo values(4,'4');
Query OK, 1 row affected (0.04 sec)
```

```
mysql> select LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
| 2                |
+------------------+
1 row in set (0.00 sec)

mysql> insert into autoincre_demo(name) values('5'),('6'),('7');
Query OK, 3 rows affected (0.05 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
| 5                |
+------------------+
1 row in set (0.00 sec)
```

对于 InnoDB 表，自动增长列必须是索引。如果是组合索引，也必须是组合索引的第一列，但是对于 MyISAM 表，自动增长列可以是组合索引的其他列，这样插入记录后，自动增长列是按照组合索引的前面几列进行排序后递增的。例如，创建一个新的 MyISAM 类型的表 autoincre_demo，自动增长列 d1 作为组合索引的第二列，对该表插入一些记录后，可以发现自动增长列是按照组合索引的第一列 d2 进行排序后递增的：

```
mysql> create table autoincre_demo
    -> (d1 smallint not null auto_increment,
    -> d2 smallint not null,
    -> name varchar(10),
    -> index(d2,d1)
    -> )engine=myisam;
Query OK, 0 rows affected (0.03 sec)

mysql> insert into autoincre_demo(d2,name) values(2,'2'),(3,'3'),(4,'4'),(2,'2'),(3,'3'),
(4,'4');
Query OK, 6 rows affected (0.00 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> select * from autoincre_demo;
+----+----+------+
| d1 | d2 | name |
+----+----+------+
| 1  | 2  | 2    |
| 1  | 3  | 3    |
| 1  | 4  | 4    |
```

```
| 2  | 2  | 2    |
| 2  | 3  | 3    |
| 2  | 4  | 4    |
+----+----+------+
6 rows in set (0.00 sec)
```

2、外键约束

MySQL 支持外键的存储引擎只有 InnoDB，在创建外键的时候，要求父表必须有对应的索引，子表在创建外键的时候也会自动创建对应的索引。

下面是样例数据库中的两个表，country 表是父表，country_id 为主键索引，city 表是子表，country_id 字段对 country 表的 country_id 有外键。

```
CREATE TABLE country (
  country_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  country VARCHAR(50) NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (country_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE city (
  city_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  city VARCHAR(50) NOT NULL,
  country_id SMALLINT UNSIGNED NOT NULL,
  last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY  (city_id),
  KEY idx_fk_country_id (country_id),
  CONSTRAINT `fk_city_country` FOREIGN KEY (country_id) REFERENCES country (country_id) ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

在创建索引的时候，可以指定在删除、更新父表时，对子表进行的相应操作，包 RESTRICT、CASCADE、SET NULL 和 NO ACTION。其中 RESTRICT 和 NO ACTION 相同，是指限制在子表有关联记录的情况下父表不能更新；CASCADE 表示父表在更新或者删除时，更新或者删除子表对应记录；SET NULL 则表示父表在更新或者删除的时候，子表的对应字段被 SET NULL。选择后两种方式的时候要谨慎，可能会因为错误的操作导致数据的丢失。

例如对上面创建的两个表，子表的外键指定是 ON DELETE RESTRICT ON UPDATE CASCADE 方式的，那么在主表删除记录的时候，如果子表有对应记录，则不允许删除，主表在更新记录的时候，如果子表有对应记录，则子表对应更新：

```
mysql> select * from country where country_id = 1;
+------------+-------------+---------------------+
| country_id | country     | last_update         |
+------------+-------------+---------------------+
| 1          | Afghanistan | 2006-02-15 04:44:00 |
+------------+-------------+---------------------+
1 row in set (0.00 sec)
```

```
mysql> select * from city where country_id = 1;
+---------+-------+------------+---------------------+
| city_id | city  | country_id | last_update         |
+---------+-------+------------+---------------------+
| 251     | Kabul | 1          | 2006-02-15 04:45:25 |
+---------+-------+------------+---------------------+
1 row in set (0.00 sec)

mysql> delete from country where country_id=1;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`sakila/city`, CONSTRAINT `fk_city_country` FOREIGN KEY (`country_id`) REFERENCES `country` (`country_id`) ON UPDATE CASCADE)

mysql> update country set country_id = 10000 where country_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from country where country = 'Afghanistan';
+------------+-------------+---------------------+
| country_id | country     | last_update         |
+------------+-------------+---------------------+
| 10000      | Afghanistan | 2007-07-17 09:45:23 |
+------------+-------------+---------------------+
1 row in set (0.00 sec)

mysql> select * from city where city_id = 251;
+---------+-------+------------+---------------------+
| city_id | city  | country_id | last_update         |
+---------+-------+------------+---------------------+
| 251     | Kabul | 10000      | 2006-02-15 04:45:25 |
+---------+-------+------------+---------------------+
1 row in set (0.00 sec)
```

当某个表被其他表创建了外键参照，那么该表的对应索引或者主键禁止被删除。

在导入多个表的数据时，如果需要忽略表之前的导入顺序，可以暂时关闭外键的检查；同样，在执行 LOAD DATA 和 ALTER TABLE 操作的时候，可以通过暂时关闭外键约束来加快处理的速度，关闭的命令是“SET FOREIGN_KEY_CHECKS = 0;”，执行完成之后，通过执行“SET FOREIGN_KEY_CHECKS = 1;”语句改回原状态。

对于 InnoDB 类型的表，外键的信息通过使用 show create table 或者 show table status 命令都可以显示。

```
mysql> show table status like 'city' \G
*************************** 1. row ***************************
           Name: city
```

```
             Engine: InnoDB
            Version: 10
         Row_format: Compact
               Rows: 427
     Avg_row_length: 115
        Data_length: 49152
    Max_data_length: 0
       Index_length: 16384
          Data_free: 0
     Auto_increment: 601
        Create_time: 2007-07-17 09:45:33
        Update_time: NULL
         Check_time: NULL
          Collation: utf8_general_ci
           Checksum: NULL
     Create_options:
            Comment: InnoDB free: 0 kB; (`country_id`) REFER `sakila/country`(`country_id`) ON
UPDATE
    1 row in set (0.00 sec)
```

3、存储方式

InnoDB 存储表和索引有以下两种方式。

- 使用共享表空间存储，这种方式创建的表的表结构保存在.frm 文件中，数据和索引保存在 innodb_data_home_dir 和 innodb_data_file_path 定义的表空间中，可以是多个文件。
- 使用多表空间存储，这种方式创建的表的表结构仍然保存在.frm 文件中，但是每个表的数据和索引单独保存在.ibd 中。如果是个分区表，则每个分区对应单独的.ibd 文件，文件名是“表名+分区名”，可以在创建分区的时候指定每个分区的数据文件的位置，以此来将表的 IO 均匀分布在多个磁盘上。

要使用多表空间的存储方式，需要设置参数 innodb_file_per_table，并重新启动服务后才可以生效，对于新建的表按照多表空间的方式创建，已有的表仍然使用共享表空间存储。如果将已有的多表空间方式修改回共享表空间的方式，则新建表会在共享表空间中创建，但已有的多表空间的表仍然保存原来的访问方式。所以多表空间的参数生效后，只对新建的表生效。

多表空间的数据文件没有大小限制，不需要设置初始大小，也不需要设置文件的最大限制、扩展大小等参数。

对于使用多表空间特性的表，可以比较方便地进行单表备份和恢复操作，但是直接复制.ibd 文件是不行的，因为没有共享表空间的数据字典信息，直接复制的.ibd 文件和.frm 文件恢复时是不能被正确识别的，但可以通过以下命令：

```
ALTER TABLE tbl_name DISCARD TABLESPACE;
ALTER TABLE tbl_name IMPORT TABLESPACE;
```

将备份恢复到数据库中，但是这样的单表备份，只能恢复到表原来在的数据库中，而不

能恢复到其他的数据库中。如果要将单表恢复到目标数据库，则需要通过 mysqldump 和 mysqlimport 来实现。

> 注意：即便在多表空间的存储方式下，共享表空间仍然是必须的，InnoDB 把内部数据词典和未作日志放在这个文件中。

7.2.3 MEMORY

MEMORY 存储引擎使用存在内存中的内容来创建表。每个 MEMORY 表只实际对应一个磁盘文件，格式是.frm。MEMORY 类型的表访问非常得快，因为它的数据是放在内存中的，并且默认使用 HASH 索引，但是一旦服务关闭，表中的数据就会丢失掉。

下面例子创建了一个 MEMORY 的表，并从 city 表获得记录：

```
mysql> CREATE TABLE tab_memory ENGINE=MEMORY
    ->     SELECT city_id,city,country_id
    ->     FROM city GROUP BY city_id;
Query OK, 600 rows affected (0.06 sec)
Records: 600  Duplicates: 0  Warnings: 0

mysql> select count(*) from tab_memory;
+----------+
| count(*) |
+----------+
| 600      |
+----------+
1 row in set (0.00 sec)

mysql> show table status like 'tab_memory' \G
*************************** 1. row ***************************
           Name: tab_memory
         Engine: MEMORY
        Version: 10
     Row_format: Fixed
           Rows: 600
 Avg_row_length: 155
    Data_length: 127040
Max_data_length: 16252835
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: NULL
    Update_time: NULL
     Check_time: NULL
      Collation: gbk_chinese_ci
       Checksum: NULL
 Create_options:
```

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

给 MEMORY 表创建索引的时候，可以指定使用 HASH 索引还是 BTREE 索引：

```
mysql> create index mem_hash USING HASH on tab_memory (city_id) ;
Query OK, 600 rows affected (0.04 sec)
Records: 600  Duplicates: 0  Warnings: 0

mysql> SHOW INDEX FROM tab_memory \G
*************************** 1. row ***************************
       Table: tab_memory
  Non_unique: 1
    Key_name: mem_hash
Seq_in_index: 1
 Column_name: city_id
   Collation: NULL
 Cardinality: 300
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: HASH
     Comment:
1 row in set (0.01 sec)

mysql> drop index mem_hash on tab_memory;
Query OK, 600 rows affected (0.04 sec)
Records: 600  Duplicates: 0  Warnings: 0

mysql> create index mem_hash USING BTREE on tab_memory (city_id) ;
Query OK, 600 rows affected (0.03 sec)
Records: 600  Duplicates: 0  Warnings: 0

mysql> SHOW INDEX FROM tab_memory \G
*************************** 1. row ***************************
       Table: tab_memory
  Non_unique: 1
    Key_name: mem_hash
Seq_in_index: 1
 Column_name: city_id
   Collation: A
 Cardinality: NULL
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
```

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

在启动 MySQL 服务的时候使用--init-file 选项，把 INSERT INTO ... SELECT 或 LOAD DATA INFILE 这样的语句放入这个文件中，就可以在服务启动时从持久稳固的数据源装载表。

服务器需要足够内存来维持所有在同一时间使用的 MEMORY 表，当不再需要 MEMORY 表的内容之时，要释放被 MEMORY 表使用的内存，应该执行 DELETE FROM 或 TRUNCATE TABLE，或者整个地删除表（使用 DROP TABLE 操作）。

每个 MEMORY 表中可以放置的数据量的大小，受到 max_heap_table_size 系统变量的约束，这个系统变量的初始值是 16MB，可以按照需要加大。此外，在定义 MEMORY 表的时候，可以通过 MAX_ROWS 子句指定表的最大行数。

MEMORY 类型的存储引擎主要用在那些内容变化不频繁的代码表，或者作为统计操作的中间结果表，便于高效地对中间结果进行分析并得到最终的统计结果。对 MEMORY 存储引擎的表进行更新操作要谨慎，因为数据并没有实际写入到磁盘中，所以一定要对下次重新启动服务后如何获得这些修改后的数据有所考虑。

7.2.4 MERGE

MERGE 存储引擎是一组 MyISAM 表的组合，这些 MyISAM 表必须结构完全相同，MERGE 表本身并没有数据，对 MERGE 类型的表可以进行查询、更新、删除的操作，这些操作实际上是对内部的实际的 MyISAM 表进行的。对于 MERGE 类型表的插入操作，是通过 INSERT_METHOD 子句定义插入的表，可以有 3 个不同的值，使用 FIRST 或 LAST 值使得插入操作被相应地作用在第一或最后一个表上，不定义这个子句或者定义为 NO，表示不能对这个 MERGE 表执行插入操作。

可以对 MERGE 表进行 DROP 操作，这个操作只是删除 MERGE 的定义，对内部的表没有任何的影响。

MERGE 表在磁盘上保留两个文件，文件名以表的名字开始，一个.frm 文件存储表定义，另一个.MRG 文件包含组合表的信息，包括 MERGE 表由哪些表组成、插入新的数据时的依据。可以通过修改.MRG 文件来修改 MERGE 表，但是修改后要通过 FLUSH TABLES 刷新。

下面是一个创建和使用 MERGE 表的例子。

（1）创建 3 个测试表 payment_2006、payment_2007 和 payment_all，其中 payment_all 是前两个表的 MERGE 表：

```
mysql> create table payment_2006(
    -> country_id smallint,
    -> payment_date datetime,
    -> amount DECIMAL(15,2),
    -> KEY idx_fk_country_id (country_id)
    -> )engine=myisam;
Query OK, 0 rows affected (0.03 sec)

mysql> create table payment_2007(
    -> country_id smallint,
    -> payment_date datetime,
    -> amount DECIMAL(15,2),
    -> KEY idx_fk_country_id (country_id)
    -> )engine=myisam;
```

```
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE payment_all(
    -> country_id smallint,
    -> payment_date datetime,
    -> amount DECIMAL(15,2),
    -> INDEX(country_id)
    -> )engine=merge union=(payment_2006,payment_2007) INSERT_METHOD=LAST;
Query OK, 0 rows affected (0.04 sec)
```

（2）分别向 payment_2006 和 payment_2007 表中插入测试数据：

```
mysql> insert into payment_2006 values(1,'2006-05-01',100000),(2,'2006-08-15',150000);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> insert into payment_2007 values(1,'2007-02-20',35000),(2,'2007-07-15',220000);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

（3）分别查看这 3 个表中的记录：

```
mysql> select * from payment_2006;
+------------+---------------------+-----------+
| country_id | payment_date        | amount    |
+------------+---------------------+-----------+
| 1          | 2006-05-01 00:00:00 | 100000.00 |
| 2          | 2006-08-15 00:00:00 | 150000.00 |
+------------+---------------------+-----------+
2 rows in set (0.00 sec)

mysql> select * from payment_2007;
+------------+---------------------+-----------+
| country_id | payment_date        | amount    |
+------------+---------------------+-----------+
| 1          | 2007-02-20 00:00:00 | 35000.00  |
| 2          | 2007-07-15 00:00:00 | 220000.00 |
+------------+---------------------+-----------+
2 rows in set (0.00 sec)

mysql> select * from payment_all;
+------------+---------------------+-----------+
| country_id | payment_date        | amount    |
+------------+---------------------+-----------+
| 1          | 2006-05-01 00:00:00 | 100000.00 |
| 2          | 2006-08-15 00:00:00 | 150000.00 |
```

```
| 1          | 2007-02-20 00:00:00 | 35000.00  |
| 2          | 2007-07-15 00:00:00 | 220000.00 |
+------------+---------------------+-----------+
4 rows in set (0.00 sec)
```

可以发现，payment_all 表中的数据是 payment_2006 和 payment_2007 表的记录合并后的结果集。

下面向 MERGE 表插入一条记录，由于 MERGE 表的定义是 INSERT_METHOD=LAST，就会向最后一个表中插入记录，所以虽然这里插入的记录是 2006 年的，但仍然会写到 payment_2007 表中。

```
mysql> insert into payment_all values(3,'2006-03-31',112200);
Query OK, 1 row affected (0.00 sec)

mysql> select * from payment_all;
+------------+---------------------+-----------+
| country_id | payment_date        | amount    |
+------------+---------------------+-----------+
| 1          | 2006-05-01 00:00:00 | 100000.00 |
| 2          | 2006-08-15 00:00:00 | 150000.00 |
| 1          | 2007-02-20 00:00:00 | 35000.00  |
| 2          | 2007-07-15 00:00:00 | 220000.00 |
| 3          | 2006-03-31 00:00:00 | 112200.00 |
+------------+---------------------+-----------+
5 rows in set (0.00 sec)

mysql> select * from payment_2007;
+------------+---------------------+-----------+
| country_id | payment_date        | amount    |
+------------+---------------------+-----------+
| 1          | 2007-02-20 00:00:00 | 35000.00  |
| 2          | 2007-07-15 00:00:00 | 220000.00 |
| 3          | 2006-03-31 00:00:00 | 112200.00 |
+------------+---------------------+-----------+
3 rows in set (0.00 sec)
```

这也是 MERGE 表和分区表的区别，MERGE 表并不能智能地将记录写到对应的表中，而分区表是可以的（分区功能在 5.1 版中正式推出）。通常我们使用 MERGE 表来透明地对多个表进行查询和更新操作，而对这种按照时间记录的操作日志表则可以透明地进行插入操作。

7.3 如何选择合适的存储引擎

在选择存储引擎时，应根据应用特点选择合适的存储引擎，对于复杂的应用系统可以根据实际情况选择多种存储引擎进行组合。

下面是常用存储引擎的适用环境。

- MyISAM：默认的 MySQL 插件式存储引擎。如果应用是以读操作和插入操作为主，

只有很少的更新和删除操作，并且对事务的完整性、并发性要求不是很高，那么选择这个存储引擎是非常适合的。MyISAM 是在 Web、数据仓储和其他应用环境下最常使用的存储引擎之一。

- InnoDB：用于事务处理应用程序，支持外键。如果应用对事务的完整性有比较高的要求，在并发条件下要求数据的一致性，数据操作除了插入和查询以外，还包括很多的更新、删除操作，那么 InnoDB 存储引擎应该是比较合适的选择。InnoDB 存储引擎除了有效地降低由于删除和更新导致的锁定，还可以确保事务的完整提交（Commit）和回滚（Rollback），对于类似计费系统或者财务系统等对数据准确性要求比较高的系统，InnoDB 都是合适的选择。
- MEMORY：将所有数据保存在 RAM 中，在需要快速定位记录和其他类似数据的环境下，可提供极快的访问。MEMORY 的缺陷是对表的大小有限制，太大的表无法 CACHE 在内存中，其次是要确保表的数据可以恢复，数据库异常终止后表中的数据是可以恢复的。MEMORY 表通常用于更新不太频繁的小表，用以快速得到访问结果。
- MERGE：用于将一系列等同的 MyISAM 表以逻辑方式组合在一起，并作为一个对象引用它们。MERGE 表的优点在于可以突破对单个 MyISAM 表大小的限制，并且通过将不同的表分布在多个磁盘上，可以有效地改善 MERGE 表的访问效率。这对于诸如数据仓储等 VLDB 环境十分适合。

> 注意：以上只是我们按照实施经验提出的关于存储引擎选择的一些建议，但是不同应用的特点是千差万别的，选择使用哪种存储引擎才是最佳方案也不是绝对的，这需要根据用户各自的应用进行测试，从而得到最适合自己的结果。

7.4 小结

本章重点介绍了 MySQL 提供的几种主要的存储引擎及其使用、特性，以及如何根据应用的需要选择合适的存储引擎。这些提供的存储引擎有各自的优势和适用的场合，正确地选择存储引擎对改善应用的效率可以起到事半功倍的效果。

正确地选择了存储引擎之后，还需要正确选择表中的数据类型，下一章我们将详细介绍如何选择合适的数据类型。

第8章　选择合适的数据类型

在使用 MySQL 创建数据表时都会遇到一个问题，如何为字段选择合适的数据类型。例如，创建一张员工表用来记录员工的信息，这时对员工的各种属性如何来进行定义？也许大家会想，这个问题很简单，每个字段可以使用很多种数据类型来定义，比如 int、float、double、decimal 等。其实正因为可选择的数据类型太多，才需要依据一些原则来“挑选”最适合的数据类型。本章将详细介绍字符、数值、日期数据类型的一些选择原则。

8.1 CHAR 与 VARCHAR

CHAR 和 VARCHAR 类型类似，都用来存储字符串，但它们保存和检索的方式不同。CHAR 属于固定长度的字符类型，而 VARCHAR 属于可变长度的字符类型。

表 8-1 显示了将各种字符串值保存到 CHAR(4)和 VARCHAR(4)列后的结果，说明了 CHAR 和 VARCHAR 之间的差别。

表 8-1　　CHAR 和 VARCHAR 的对比

值	CHAR(4)	存储需求	VARCHAR(4)	存储需求
''	' '	4 个字节	''	1 个字节
'ab'	'ab '	4 个字节	'ab '	3 个字节
'abcd'	'abcd'	4 个字节	'abcd'	5 个字节
'abcdefgh'	'abcd'	4 个字节	'abcd'	5 个字节

请注意表 8-1 中最后一行的值只适用非“严格模式”时，如果 MySQL 运行在严格模式，超过列长度的值将不会保存，并且会出现错误提示，关于“严格模式””将在第 16 章的 SQL MODE 及其相关问题的章节中详细介绍。

从 CHAR(4)和 VARCHAR(4)列检索的值并不总是相同，因为检索时从 CHAR 列删除了尾部的空格。下面通过一个例子说明该差别：

```
mysql> CREATE TABLE vc (v VARCHAR(4), c CHAR(4));
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO vc VALUES ('ab  ', 'ab  ');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT CONCAT(v, '+'), CONCAT(c, '+') FROM vc;
+---------------+---------------+
| CONCAT(v, '+') | CONCAT(c, '+') |
+---------------+---------------+
| ab  +          | ab+           |
+---------------+---------------+
1 row in set (0.00 sec)
```

由于 CHAR 是固定长度的，所以它的处理速度比 VARCHAR 快得多，但是其缺点是浪费存储空间，程序需要对行尾空格进行处理，所以对于那些长度变化不大并且对查询速度有较高要求的数据可以考虑使用 CHAR 类型来存储。

另外，随着 MySQL 版本的不断升级，VARCHAR 数据类型的性能也在不断改进并提高，所以在许多的应用中，VARCHAR 类型被更多地使用。

在 MySQL 中，不同的存储引擎对 CHAR 和 VARCHAR 的使用原则有所不同，这里简单概括如下。

- MyISAM 存储引擎：建议使用固定长度的数据列代替可变长度的数据列。
- MEMORY 存储引擎：目前都使用固定长度的数据行存储，因此无论使用 CHAR 或 VARCHAR 列都没有关系。两者都是作为 CHAR 类型处理。
- InnoDB 存储引擎：建议使用 VARCHAR 类型。对于 InnoDB 数据表，内部的行存储

格式没有区分固定长度和可变长度列（所有数据行都使用指向数据列值的头指针），因此在本质上，使用固定长度的 CHAR 列不一定比使用可变长度 VARCHAR 列性能要好。因而，主要的性能因素是数据行使用的存储总量。由于 CHAR 平均占用的空间多于 VARCHAR，因此使用 VARCHAR 来最小化需要处理的数据行的存储总量和磁盘 I/O 是比较好的。

8.2 TEXT 与 BLOB

一般在保存少量字符串的时候，我们会选择 CHAR 或者 VARCHAR；而在保存较大文本时，通常会选择使用 TEXT 或者 BLOB，二者之间的主要差别是 BLOB 能用来保存二进制数据，比如照片；而 TEXT 只能保存字符数据，比如一篇文章或者日记。TEXT 和 BLOB 中有分别包括 TEXT、MEDIUMTEXT、LONGTEXT 和 BLOB、MEDIUMBLOB、LONGBLOB3 种不同的类型，它们之间的主要区别是存储文本长度不同和存储字节不同，用户应该根据实际情况选择能够满足需求的最小存储类型。本节主要对 BLOB 和 TEXT 存在的一些常见问题进行介绍。

- BLOB 和 TEXT 值会引起一些性能问题，特别是在执行了大量的删除操作时。

删除操作会在数据表中留下很大的“空洞”，以后填入这些“空洞”的记录在插入的性能上会有影响。为了提高性能，建议定期使用 OPTIMIZE TABLE 功能对这类表进行碎片整理，避免因为“空洞”导致性能问题。

下面的例子描述了 OPTIMIZE TABLE 的碎片整理功能。

（1）创建测试表 t，字段 id 和 context 的类型分别为 varchar(100)和 text：

```
mysql> create table t (id varchar(100),context  text);
Query OK, 0 rows affected (0.01 sec)
```

（2）往 t 中插入大量记录，这里使用 repeat 函数插入大字符串：

```
mysql> insert into t values(1,repeat('haha',100));
Query OK, 1 row affected (0.00 sec)

mysql> insert into t values(2,repeat('haha',100));
Query OK, 1 row affected (0.00 sec)

mysql> insert into t values(3,repeat('haha',100));
Query OK, 1 row affected (0.00 sec)

mysql> insert into t select * from t;
 …
mysql> insert into t select * from t;
Query OK, 196608 rows affected (4.86 sec)
Records: 196608  Duplicates: 0  Warnings: 0

mysql> exit
Bye
```

（3）退出到操作系统下，查看表 t 的物理文件大小：

```
[bjguan@zzx test]$ du -sh t.*
```

```
16K     t.frm
155M    t.MYD
8.0K    t.MYI
```

这里数据文件显示为 155MB。

（4）从表 t 中删除 id 为“1”的数据，这些数据占总数据量的 1/3：

```
[bjguan@localhost test]$ mysql -u root -p1234
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 24 to server version: 5.0.45-log

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

mysql> use test
Database changed
mysql> delete from t where id=1;
Query OK, 131072 rows affected (4.33 sec)

mysql> exit
Bye
```

（5）再次退出到操作系统下，查看表 t 的物理文件大小：

```
[bjguan@zzx test]$ du -sh t.*
16K     t.frm
155M    t.MYD
8.0K    t.MYI
```

可以发现，表 t 的数据文件仍然为 155MB，并没有因为数据删除而减少。

（6）接下来对表进行 OPTIMIZE（优化）操作：

```
[bjguan@localhost test]$ mysql -u root -p1234
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 24 to server version: 5.0.45-log

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

mysql> use test;
Database changed
mysql> OPTIMIZE TABLE  t;
+--------+----------+----------+----------+
| Table  | Op       | Msg_type | Msg_text |
+--------+----------+----------+----------+
| test.t | optimize | status   | OK       |
+--------+----------+----------+----------+
1 row in set (2.88 sec)

mysql> exit
```

```
Bye
```

（7）再次查看表 t 的物理文件大小：

```
[bjguan@localhost test]$ du -sh t.*
16K     t.frm
104M    t.MYD
8.0K    t.MYI
```

可以发现，表的数据文件大大缩小，“空洞”空间已经被回收。

■ 可以使用合成的（Synthetic）索引来提高大文本字段（BLOB 或 TEXT）的查询性能。简单来说，合成索引就是根据大文本字段的内容建立一个散列值，并把这个值存储在单独的数据列中，接下来就可以通过检索散列值找到数据行了。但是，要注意这种技术只能用于精确匹配的查询（散列值对于类似<或>=等范围搜索操作符是没有用处的）。可以使用 MD5()函数生成散列值，也可以使用 SHA1()或 CRC32()，或者使用自己的应用程序逻辑来计算散列值。请记住数值型散列值可以很高效率地存储。同样，如果散列算法生成的字符串带有尾部空格，就不要把它们存储在 CHAR 或 VARCHAR 列中，它们会受到尾部空格去除的影响。合成的散列索引对于那些 BLOB 或 TEXT 数据列特别有用。用散列标识符值查找的速度比搜索 BLOB 列本身的速度快很多。

下面通过实例介绍一下合成索引的使用方法。

（1）创建测试表 t，字段 id、context、hash_value 字段类型分别为 varchar(100)、blob、varchar(40)：

```
mysql> create table t (id varchar(100),context  blob,hash_value varchar(40));
Query OK, 0 rows affected (0.03 sec)
```

（2）t 中插入测试数据，其中 hash_value 用来存放 context 列的 MD5 散列值：

```
mysql> insert into t values(1,repeat('beijing',2),md5(context));
Query OK, 1 row affected (0.00 sec)

mysql> insert into t values(2,repeat('beijing',2),md5(context));
Query OK, 1 row affected (0.00 sec)

mysql> insert into t values(3,repeat('beijing 2008',2),md5(context));
Query OK, 1 row affected (0.00 sec)
mysql> select * from t;
+------+-------------------------+----------------------------------+
| id   | context                 | hash_value                       |
+------+-------------------------+----------------------------------+
| 1    | beijingbeijing          | 09746eef633dbbccb7997dfd795cff17 |
| 2    | beijingbeijing          | 09746eef633dbbccb7997dfd795cff17 |
| 3    | beijing 2008beijing 2008 | 1c0ddb82cca9ed63e1cacbddd3f74082 |
+------+-------------------------+----------------------------------+
3 rows in set (0.00 sec)
```

（3）如果要查询 context 值为“beijing 2008beijing 2008”的记录，可以通过相应的散列值来查询：

```
mysql> select * from t where hash_value=md5(repeat('beijing 2008',2));
+------+--------------------------+----------------------------------+
| id   | context                  | hash_value                       |
+------+--------------------------+----------------------------------+
| 3    | beijing 2008beijing 2008 | 1c0ddb82cca9ed63e1cacbddd3f74082 |
+------+--------------------------+----------------------------------+
1 row in set (0.00 sec)
```

上面的例子展示了合成索引的用法，由于这种技术只能用于精确匹配，在一定程度上减少 I/O，从而提高查询效率。如果需要对 BLOB 或者 CLOB 字段进行模糊查询，MySQL 提供了前缀索引，也就是只为字段的前 n 列创建索引，举例如下：

```
mysql> create index idx_blob on t(context(100));
Query OK, 3 rows affected (0.04 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> desc select * from t where context like 'beijing%' \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: range
possible_keys: idx_blob
          key: idx_blob
      key_len: 103
          ref: NULL
         rows: 2
        Extra: Using where
1 row in set (0.00 sec)
```

可以发现，对 context 前 100 个字符进行模糊查询，就可以用到前缀索引。请注意，这里的查询条件中，“%”不能放在最前面，否则索引将不会被使用。

- 在不必要的时候避免检索大型的 BLOB 或 TEXT 值。

例如，SELECT * 查询就不是很好的想法，除非能够确定作为约束条件的 WHERE 子句只会找到所需要的数据行。否则，很可能毫无目的地在网络上传输大量的值。这也是 BLOB 或 TEXT 标识符信息存储在合成的索引列中对用户有所帮助的例子。用户可以搜索索引列，决定需要的哪些数据行，然后从符合条件的数据行中检索 BLOB 或 TEXT 值。

- 把 BLOB 或 TEXT 列分离到单独的表中。

在某些环境中，如果把这些数据列移动到第二张数据表中，可以把原数据表中的数据列转换为固定长度的数据行格式，那么它就是有意义的。这会减少主表中的碎片，可以得到固定长度数据行的性能优势。它还可以使主数据表在运行 SELECT * 查询的时候不会通过网络传输大量的 BLOB 或 TEXT 值。

8.3 浮点数与定点数

浮点数一般用于表示含有小数部分的数值。当一个字段被定义为浮点类型后，如果插入

数据的精度超过该列定义的实际精度，则插入值会被四舍五入到实际定义的精度值，然后插入，四舍五入的过程不会报错。在 MySQL 中 float、double（或 real）用来表示浮点数。

定点数不同于浮点数，定点数实际上是以字符串形式存放的，所以定点数可以更加精确的保存数据。如果实际插入的数值精度大于实际定义的精度，则 MySQL 会进行警告（默认的 SQLMode 下），但是数据按照实际精度四舍五入后插入；如果 SQLMode 是在 TRADITIONAL（传统模式）下，则系统会直接报错，导致数据无法插入。在 MySQL 中，decimal（或 numberic）用来表示定点数。

在简单了解了浮点数和定点数的区别之后，来看一个例子，回顾一下前面讲到的浮点数精确性问题。

```
mysql> create table t (f float( 8,1));
Query OK, 0 rows affected (0.03 sec)

mysql> desc t;
+-------+-----------+------+-----+---------+-------+
| Field | Type      | Null | Key | Default | Extra |
+-------+-----------+------+-----+---------+-------+
| f     | float(8,1) | YES  |     | NULL    |       |
+-------+-----------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> insert into t values (1.23456);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+------+
| f    |
+------+
|  1.2 |
+------+
1 row in set (0.00 sec)

mysql> insert into t values (1.25456);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+------+
| f    |
+------+
|  1.2 |
|  1.3 |
+------+
2 rows in set (0.00 sec)
```

从上面的例子中，可以发现对于第一次插入值 1.23456 到 float（8,1）时，该值被截断，并保存为 1.2，而第二次插入值 1.25456 到 float（8,1）时，该值进行了四舍五入然后被截断，

并保存为 1.3，所以在选择浮点型数据保存小数时，要注意四舍五入的问题，并尽量保留足够的小数位，避免存储的数据不准确。

为了能够让大家了解浮点数与定点数的区别，再来看一个例子：

```
mysql> CREATE TABLE test (c1 float(10,2),c2 decimal(10,2));
Query OK, 0 rows affected (0.29 sec)
mysql> insert into test values(131072.32,131072.32);
Query OK, 1 row affected (0.07 sec)
mysql> select * from test;
+-----------+-----------+
| c1        | c2        |
+-----------+-----------+
| 131072.31 | 131072.32 |
+-----------+-----------+
1 row in set (0.00 sec)
```

从上面的例子中可以看到，c1 列的值由 131072.32 变成了 131072.31，这是上面的数值在使用单精度浮点数表示时，产生了误差。这是浮点数特有的问题。因此在精度要求比较高的应用中（比如货币）要使用定点数而不是浮点数来保存数据。

另外，浮点数的比较也是一个普遍存在的问题，下面的程序片断中对两个浮点数做减法运算：

```
public class Test {
               public static void main(String[] args) throws Exception {
                  System.out.print("7.22-7.0=" + (7.22f-7.0f));
               }
               }
```

对上面 Java 程序的输出结果可能会想当然的认为是 0.22，但是，实际结果却是 7.22-7.0=0.21999979，因此，在编程中应尽量避免浮点数的比较，如果非要使用浮点数比较则最好使用范围比较而不要使用“==”比较。

再看一下使用定点数来实现上面的例子：

```
import java.math.BigDecimal;
/*
 * 提供精确的减法运算。
 * @param v1
 * @param v2
 */

public class Test {

    public static void main(String[] args) throws Exception {

       System.out.print("7.22-7.0=" + subtract(7.22,7.0));
    }

    public static double subtract(double v1, double v2)    {
```

```
            BigDecimal b1 = new BigDecimal(Double.toString(v1));

            BigDecimal b2 = new BigDecimal(Double.toString(v2));

            return b1.subtract(b2).doubleValue();
        }
    }
```

上面的实例使用 Java 的 BigDecimal 类实现了定点数的精确计算，所以 7.22 减 7.0 的结果和预想的相同，为 7.22-7.0=0.22。

> 注意：在今后关于浮点数和定点数的应用中，用户要考虑到以下几个原则：
>
> - 浮点数存在误差问题；
> - 对货币等对精度敏感的数据，应该用定点数表示或存储；
> - 在编程中，如果用到浮点数，要特别注意误差问题，并尽量避免做浮点数比较；
> - 要注意浮点数中一些特殊值的处理。

8.4 日期类型选择

MySQL 提供的常用日期类型有 DATE、TIME 、DATETIME、TIMESTAMP，它们之间的区别在第 3 章中已经进行过详细论述，这里就不再赘述。下面主要总结一下选择日期类型的原则。

- 根据实际需要选择能够满足应用的最小存储的日期类型。如果应用只需要记录“年份”，那么用 1 个字节来存储的 YEAR 类型完全可以满足，而不需要用 4 个字节来存储的 DATE 类型。这样不仅仅能节约存储，更能够提高表的操作效率。
- 如果要记录年月日时分秒，并且记录的年份比较久远，那么最好使用 DATETIME，而不要使用 TIMESTAMP。因为 TIMESTAMP 表示的日期范围比 DATETIME 要短得多。
- 如果记录的日期需要让不同时区的用户使用，那么最好使用 TIMESTAMP，因为日期类型中只有它能够和实际时区相对应。

8.5 小结

本章中主要介绍了常见数据类型的选择原则，简单归纳如下。

- 对于字符类型，要根据存储引擎来进行相应的选择。
- 对精度要求较高的应用中，建议使用定点数来存储数值，以保证结果的准确性。
- 对含有 TEXT 和 BLOB 字段的表，如果经常做删除和修改记录的操作要定时执行 OPTIMIZE TABLE 功能对表进行碎片整理。
- 日期类型要根据实际需要选择能够满足应用的最小存储的日期类型。

第9章　字符集

从本质上来说，计算机只能识别二进制代码，因此，不论是计算机程序还是其处理的数据，最终都必须转换成二进制码，计算机才能认识。为了使计算机不仅能做科学计算，也能处理文字信息，人们想出了给每个文字符号编码以便于计算机识别处理的办法，这就是计算机字符集的由来。本章将详细介绍字符集的发展历程以及 MySQL 中字符集的使用。

9.1 字符集概述

简单地说字符集就是一套文字符号及其编码、比较规则的集合。1960 年代初期，美国标准化组织 ANSI 发布了第一个计算机字符集——ASCII（American Standard Code for Information Interchange），后来进一步变成了国际标准 ISO-646。这个字符集采用 7 位编码，定义了包括大小写英文字母、阿拉伯数字和标点符号，以及 33 个控制符号等。虽然现在看来，这个美式的字符集很简单，包括的符号也很少，但直到今天它依然是计算机世界里奠基性的标准，其后制定的各种字符集基本都兼容 ASCII 字符集。

自 ASCII 之后，为了处理不同的文字，各大计算机公司、各国政府、标准化组织等先后发明了几百种字符集，如大家熟悉的 ISO-8859 系列、GB2312-80、GBK、BIG5 等。这些五花八门的字符集，从收录的字符到编码规则各不相同，给计算机软件开发和移植带来了很大困难。一个软件要在使用不同文字的国家或地区发布，必须进行本地化开发！基于这个原因，统一字符编码，成了 1980 年代计算机业的迫切需要和普遍共识。

9.2 Unicode 简述

为了统一字符编码，国际标准化组织ISO（International Organization for Standardization）的一些成员国于1984年发起制定新的国际字符集标准，以容纳全世界各种语言文字和符号。这个标准最后叫做“Universal Multiple-Octet Coded Character Set”，简称UCS，标准编号则定为ISO-10646。ISO-10646标准采用4字节（32bit）编码，因此简称UCS-4。具体编码规则是：将代码空间划分为组（group）、面（plane）、行（row）和格（ceil）；第1个字节代表组（group），第2个字节代表面（plane），第3个字节代表行（row），第4个字节代表格（ceil），并规定字符编码的第32位必须为0，且每个面（plane）的最后两个码位FFFEh和FFFFh保留不用；因此，ISO-1064共有128个群组（0～0x7F），每个群组有256个字面（00～0xFF），每个字面有256行（00～0xFF），每行包括256格（0～0xFF），共有256 * 128 = 32,768 个字面，每个字面有256×256－2＝65,534个码位，合计65534×32768＝2,147,418,112 个码位。

ISO-10646发布以后，遭到了部分美国计算机公司的反对。1988年Xerox公司提议制定新的以16位编码的统一字符集Unicode，并联合Apple、IBM、DEC、Sun、Microsoft、Novell等公司成立Unicode协会（The Unicode Consortium），并成立Unicode技术委员会（Unicode Technical Committee），专门负责Unicode文字的搜集、整理和编码，并于1991年推出了Unicode 1.0。

都是为了解决字符编码统一问题，ISO和Unicode协会却推出了两个不同的编码标准，这显然是不利的。后来，大家都认识到了这一点，经过双方谈判，1991年10月达成协议，ISO将Unicode

编码并入ISO-10646的0组0字面，叫作基本多语言文字面（Basic Multi-lingual Plane, BMP），共有65,534个码位，并根据不同用途分为若干区域。除BMP外的32,767个字面又分为辅助字面（supplementary planes）和专用字面（private use planes）两部分，辅助字面用以收录ISO-10646后续搜集的各国文字，专用字面供使用者自定义收录ISO-10646未收录的文字符号。其实，大部分用户只使用BMP字面就足够了，早期的ISO-10646-1标准也只要求实现BMP字面，这样只需要2字节来编码就足够了，Unicode也正是这么做的，这叫作ISO-10646编码的基本面形式，简称为UCS-2编码，UCS-2编码转换成UCS-4编码也很容易，只要在前面加两个取值为0的字节即可。

ISO-10646的编码空间足以容纳人类从古至今使用过的所有文字和符号，但其实许多文字符号都已经很少使用了，超过99%的在用文字符号都编入了BMP，因此，绝大部分情况下，Unicode的双字节编码方式都能满足需求，而这种双字节编码方式比起ISO-10646的4字节原始编码来说，在节省内存和处理时间上都具有优势，这也是Unicode编码方式更流行的原因。但如果万一要使用ISO-10646 BMP字面以外的文字怎么办呢？Unicode提出了名为UTF-16或代理法（surrogates）的解决方案，UTF是UCS/Unicode Transformation Format 的缩写。UTF-16的解决办法是：对BMP字面的编码保持二字节不变，对其他字面的文字按一定规则将其32位编码转换为两个16位的Unicode编码，其两个字节的取值范围分别限定为0xD800～0xDBFF和0xDC00～0xDFFF，因此，UTF-16共有（4×256）×（4×256）＝1048576个码位。

虽然UTF-16解决了ISO-10646除BMP外第1至第15字面的编码问题，但当时的计算机和网络世界还是ASCII的天下，只能处理单字节数据流，UTF-16在离开Unicode环境后，在传输和处理中都存在问题。于是Unicode又提出了名为UTF-8的解决方案，UTF-8按一定规则将一个ISO-10646或Unicode字元码转换成1至4个字节的编码，其中将ASCII码（0～0x7F）转换成单字节编码，也就是严格兼容ASCII字符集；UTF-8的2字节编码，用以转换ISO-10646标准0x0080～0x07FF的UCS-4原始码；UTF-8的3字节编码，用以转换ISO-10646标准 0x0800～0xFFFF的UCS-4原始码；UTF-8的4字节编码，用以转换ISO-10646标准 0x00010000～0001FFFF的UCS-4原始码。

上述各种编码方式，看起来有点让人迷惑。其实，ISO-10646只是给每一个文字符号分配了一个4字节无符号整数编号（UCS-4），并未规定在计算机中如何去表示这个无符号整数编号。UTF-16和UTF-8就是其两种变通表示方式。

ISO-10646与Unicode统一以后，两个组织虽然都继续发布各自的标准，但二者之间是一致的。由于Unicode最早投入应用，其编码方式更加普及，因此，许多人都知道Unicode，但对ISO-10646却了解不多。但由于二者是一致的，因此，区分ISO-10646和Unicode的意义也就不大了。现在，大家说Unicode和ISO-10646，一般指的是同一个东西，只是Unicode更直接、更普及罢了。二者不同版本的对应关系如下。

- Unicode 2.0等同于ISO/IEC 10646-1:1993。
- Unicode 3.0等同于ISO/IEC 10646-1:2000。
- Unicode 4.0等同于ISO/IEC 10646:2003。

最后要说的是，UTF-16和UTF-32因字节序的不同，又有了UTF-16BE（Big Endian）、UTF-16LE（Little Endian）和UTF-32BE（Big Endian）、UTF-32LE（Little Endian）等，在此不做进一步介绍。

9.3 汉字及一些常见字符集

在计算机发展的不同阶段，我国也参照当时的国际标准和实际需要，制定了一些汉字字符集编码标准，主要包括：

- GB2312-80：全称《信息交换用汉字编码字符集 基本集》，于 1980 年发布。根据 ISO/IEC 2022 提供的字符编码扩充规范，形成双字节编码的字符集。收录了 6763 个常用汉字和 682 个非汉字图形符号。
- GB13000：全称《信息技术 通用多八位编码字符集(UCS) 第一部分：体系结构与基本多文种平面》，于 1993 年发布。根据 ISO/IEC 10646-1:1993，在 CJK（中、日、韩简称）统一汉字区和 CJK 统一汉字扩充区 A，除收录 GB2312-80 外，还收录了第 1、3、5、7 辅助集的全部汉字，共 27，484 个，以及一些偏旁部首等。但 GB13000 推出后，几乎没有得到业界的支持，也就成了一个形式上的标准。
- GBK：全称《汉字内码扩展规范》1.0 版，发布于 1995 年。GBK 在 GB2312 内码系统的基础上进行了扩充，收录了 GB13000.1-1993 的全部 20902 个 CJK 统一汉字，包括 GB2312 的全部 6763 个汉字。此外，它增补编码了 52 个汉字，13 个汉字结构符（在 ISO/IEC 10646.1: 2000 中称为表意文字描述符）和一些常用部首与汉字部件。在 GBK 的内码系统中，GB2312 汉字所在码位保持不便，这样，保证了 GBK 对 GB2312 的完全兼容。同时，GBK 内码与 GB13000.1 代码一一对应，为 GBK 向 GB13000.1 的转换提供了解决办法。有意思的是 GBK 并不是一个强制性的国家标准，只是一个行业指导规范，并没有强制力，但由于得到了 Microsoft Windows 95 的支持而大为流行。
- GB18030：全称《信息技术信息交换用汉字编码字符集、基本集的扩充》，发布于 2000 年。根据 ISO/IEC 10646-1:2000，收录了 ISO/IEC 10646.1: 2000 全部 27,484 个 CJK 统一汉字，13 个表意文字描述符、部分汉字部首和部件、欧元符号等。GB18030 采用 2 字节或 4 字节编码，其二字节编码部分与 GBK 保持一致，因此，GB18030 是 GBK 的超集，也完全与 GB13000 向上兼容，制定 GB18030 也是为了解决 GBK 强制力不够的问题。

以上简要介绍了几种汉字字符集，下面将一些常用字符集的特点归纳如表 9-1 所示。

表 9-1　　常用字符集比较

字符集	是否定长	编码方式	其他说明
ACSII	是	单字节 7 位编码	最早的奠基性字符集
ISO-8859-1/latin1	是	单字节 8 位编码	西欧字符集，经常被一些程序员用来转码
GB2312-80	是	双字节编码	早期标准，不推荐再使用
GBK	是	双字节编码	虽然不是国标，但支持的系统不少
GB18030	否	2 字节或 4 字节编码	开始有一些支持，但数据库支持的还少见
UTF-32	是	4 字节编码	UCS-4 原始编码，目前很少采用
UCS-2	是	2 字节编码	Windows 2000 内部用 UCS-2
UTF-16	否	2 字节或 4 字节编码	Java 和 Windows XP/NT 等内部使用 UTF-16
UTF-8	否	1 至 4 字节编码	互联网和 UNIX/Linux 广泛支持的 Unicode 字符集；MySQLServer 也使用 UTF-8

9.4 怎样选择合适的字符集

对数据库来说，字符集更加重要，因为数据库存储的数据大部分都是各种文字，字符集

对数据库的存储、处理性能，以及日后系统的移植、推广都会有影响。

MySQL 5.0 目前支持几十种字符集，UTF-8 是 MySQL 5.0 支持的唯一 Unicode 字符集，但版本是 3.0，不支持 4 字节的扩展部分。面对众多的字符集，我们该如何选择呢？

虽然没有一定之规，但在选择数据库字符集时，可以根据应用的需求，结合上面介绍的一些字符集的特点来权衡，主要考虑因素包括：

（1）满足应用支持语言的需求，如果应用要处理各种各样的文字，或者将发布到使用不同语言的国家或地区，就应该选择 Unicode 字符集。对 MySQL 来说，目前就是 UTF-8。

（2）如果应用中涉及已有数据的导入，就要充分考虑数据库字符集对已有数据的兼容性。假如已有数据是 GBK 文字，如果选择 GB2312-80 为数据库字符集，就很可能出现某些文字无法正确导入的问题。

（3）如果数据库只需要支持一般中文，数据量很大，性能要求也很高，那就应该选择双字节定长编码的中文字符集，比如 GBK。因为，相对于 UTF-8 而言，GBK 比较“小”，每个汉字只占 2 个字节，而 UTF-8 汉字编码需要 3 个字节，这样可以减少磁盘 I/O、数据库 cache，以及网络传输的时间，从而提高性能。相反，如果应用主要处理英文字符，仅有少量汉字数据，那么选择 UTF-8 更好，因为 GBK、UCS-2、UTF-16 的西文字符编码都是 2 个字节，会造成很大不必要的开销。

（4）如果数据库需要做大量的字符运算，如比较、排序等，选择定长字符集可能更好，因为定长字符集的处理速度要比变长字符集的处理速度快。

（5）如果所有客户端程序都支持相同的字符集，应该优先选择该字符集作为数据库字符集。这样可以避免因字符集转换带来的性能开销和数据损失。

9.5 MySQL 支持的字符集简介

MySQL 服务器可以支持多种字符集，在同一台服务器、同一个数据库、甚至同一个表的不同字段都可以指定使用不同的字符集，相比 Oracle 等其他数据库管理系统，在同一个数据库只能使用相同的字符集，MySQL 明显存在更大的灵活性。

查看所有可用的字符集的命令是 show character set：

```
mysql> show character set;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| dec8     | DEC West European           | dec8_swedish_ci     | 1      |
| cp850    | DOS West European           | cp850_general_ci    | 1      |
| hp8      | HP West European            | hp8_english_ci      | 1      |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    | 1      |
……
```

或者查看 information_schema.character_set，可以显示所有的字符集和该字符集默认的校对规则。

```
mysql> desc information_schema.character_sets;
+----------------------+-------------+------+-----+---------+-------+
| Field                | Type        | Null | Key | Default | Extra |
+----------------------+-------------+------+-----+---------+-------+
| CHARACTER_SET_NAME   | varchar(64) | NO   |     |         |       |
```

```
| DEFAULT_COLLATE_NAME | varchar(64) | NO   |     |         |       |
| DESCRIPTION          | varchar(60) | NO   |     |         |       |
| MAXLEN               | bigint(3)   | NO   |     | 0       |       |
+----------------------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

MySQL 的字符集包括字符集（CHARACTER）和校对规则（COLLATION）两个概念。字符集是用来定义 MySQL 存储字符串的方式，校对规则则是定义了比较字符串的方式。字符集和校对规则是一对多的关系，MySQL 支持 30 多种字符集的 70 多种校对规则。

每个字符集至少对应一个校对规则。可以用“SHOW COLLATION LIKE '***';”命令或者查看 information_schema.COLLATIONS。查看相关字符集的校对规则。

```
mysql> SHOW COLLATION LIKE 'gbk%';
+---------------+---------+----+---------+----------+---------+
| Collation     | Charset | Id | Default | Compiled | Sortlen |
+---------------+---------+----+---------+----------+---------+
| gbk_chinese_ci | gbk     | 28 | Yes     | Yes      | 1       |
| gbk_bin        | gbk     | 87 |         | Yes      | 1       |
+---------------+---------+----+---------+----------+---------+
2 rows in set (0.00 sec)
```

校对规则命名约定：它们以其相关的字符集名开始，通常包括一个语言名，并且以_ci（大小写不敏感）、_cs（大小写敏感）或_bin（二元，即比较是基于字符编码的值而与language无关）结束。

例如，上面例子中 GBK 的校对规则，其中 gbk_chinese_ci 是默认的校对规则，大小写不敏感的，gbk_bin 按照编码的值进行比较，是大小写敏感的。

下面的这个例子中，如果指定'A'和'a'按照 gbk_chinese_ci 校对规则进行比较，则认为两个字符是相同的，如果按照 gbk_bin 校对规则进行比较，则认为两个字符是不同的。我们事先需要确认应用的需求，是需要按照什么样的排序方式，是否需要区分大小写，以确定校对规则的选择。

```
mysql> select case when 'A' COLLATE gbk_chinese_ci = 'a' collate gbk_chinese_ci  then 1 else
0 end;
+-----------------------------------------------------------------------------------------
+
| case when 'A' COLLATE gbk_chinese_ci = 'a' collate gbk_chinese_ci  then 1 else 0 end |
+-----------------------------------------------------------------------------------------
+
| 1                                                                                       |
+-----------------------------------------------------------------------------------------
+
1 row in set (0.00 sec)

mysql>  select case when 'A' COLLATE gbk_bin = 'a' collate gbk_bin  then 1 else 0 end;
+-------------------------------------------------------------------+
| case when 'A' COLLATE gbk_bin = 'a' collate gbk_bin  then 1 else 0 end |
+-------------------------------------------------------------------+
```

```
| 0                                                                          |
+----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

9.6 MySQL 字符集的设置

MySQL 的字符集和校对规则有 4 个级别的默认设置：服务器级、数据库级、表级和字段级。它们分别在不同的地方设置，作用也不相同。

9.6.1 服务器字符集和校对规则

服务器字符集和校对，在 MySQL 服务启动的时候确定。

```
可以在 my.cnf 中设置：
[mysqld]
default-character-set=gbk
或者在启动选项中指定：
mysqld --default-character-set=gbk
或者在编译的时候指定：
./configure --with-charset=gbk
```

如果没有特别的指定服务器字符集，默认使用 latin1 作为服务器字符集。上面 3 种设置的方式都只指定了字符集，没有指定校对规则，这样是使用该字符集默认的校对规则，如果要使用该字符集的非默认校对规则，则需要在指定字符集的同时指定校对规则。

可以用“show variables like 'character_set_server';”命令查询当前服务器的字符集和校对规则。

```
mysql> show variables like 'character_set_server';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| character_set_server | gbk   |
+----------------------+-------+
1 row in set (0.00 sec)

mysql> show variables like 'collation_server';
+------------------+----------------+
| Variable_name    | Value          |
+------------------+----------------+
| collation_server | gbk_chinese_ci |
+------------------+----------------+
1 row in set (0.00 sec)
```

9.6.2 数据库字符集和校对规则

数据库的字符集和校对规则在创建数据库的时候指定，也可以在创建完数据库后通过“alter database”命令进行修改。需要注意的是，如果数据库里已经存在数据，因为修改字符集并

不能将已有的数据按照新的字符集进行存放，所以不能通过修改数据库的字符集直接修改数据的内容，在 9.7 小节中通过一个具体的例子介绍了字符集的修改方法。

设置数据库字符集的规则是：

- 如果指定了字符集和校对规则，则使用指定的字符集和校对规则；
- 如果指定了字符集没有指定校对规则，则使用指定字符集的默认校对规则；
- 如果没有指定字符集和校对规则，则使用服务器字符集和校对规则作为数据库的字符集和校对规则。

推荐在创建数据库的时候明确指定字符集和校对规则，避免受到默认值的影响。

要显示当前数据库的字符集和校对规则，可以使用“show variables like 'character_set_database'”和“show variables like 'collation_database'”命令查看：

```
mysql> show variables like 'character_set_database';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| character_set_database | utf8  |
+--------------------------+-------+
1 row in set (0.00 sec)

mysql> show variables like 'collation_database';
+--------------------+-----------------+
| Variable_name      | Value           |
+--------------------+-----------------+
| collation_database | utf8_general_ci |
+--------------------+-----------------+
1 row in set (0.00 sec)
```

9.6.3 表字符集和校对规则

表的字符集和校对规则在创建表的时候指定，可以通过 alter table 命令进行修改，同样，如果表中已有记录，修改字符集对原有的记录并没有影响，不会按照新的字符集进行存放。表的字段仍然使用原来的字符集。

设置表的字符集的规则和上面基本类似：

- 如果指定了字符集和校对规则，使用指定的字符集和校对规则；
- 如果指定了字符集没有指定校对规则，使用指定字符集的默认校对规则；
- 如果没有指定字符集和校对规则，使用数据库字符集和校对规则作为表的字符集和校对规则。

推荐在创建表的时候明确指定字符集和校对规则，避免受到默认值的影响。要显示表的字符集和校对规则，可以使用 show create table 命令查看：

```
mysql>  show create table z1 \G;
*************************** 1. row ***************************
       Table: z1
Create Table: CREATE TABLE `z1` (
  `id` int(11) default NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_bin
1 row in set (0.00 sec)
```

9.6.4 列字符集和校对规则

MySQL 可以定义列级别的字符集和校对规则，主要是针对相同的表不同字段需要使用不同的字符集的情况，应该说一般遇到这种情况的几率比较小，这只是 MySQL 提供给我们一个灵活设置的手段。
列字符集和校对规则的定义可以在创建表时指定，或者在修改表时调整，如果在创建表的时候没有特别指定字符集和校对规则，则默认使用表的字符集和校对规则。

9.6.5 连接字符集和校对规则

上面 4 种设置方式，确定的是数据保存的字符集和校对规则，对于实际的应用访问来说，还存在客户端和服务器之间交互的字符集和校对规则的设置。
对于客户端和服务器的交互操作，MyISQL 提供了 3 个不同的参数：character_set_client、character_set_connection 和 character_set_results，分别代表客户端、连接和返回结果的字符集，通常情况下，这 3 个字符集应该是相同的，才可以确保用户写入的数据可以正确地读出，特别是对于中文字符，不同的写入字符集和返回结果字符集将导致写入的记录不能正确读出。
通常情况下，不会单个地设置这 3 个参数，可以通过以下命令：

```
SET NAMES ***;
```

来设置连接的字符集和校对规则，这个命令可以同时修改这 3 个参数的值。使用这个方法修改连接的字符集和校对规则，需要应用每次连接数据库后都执行这个命令。
另外一个更简便的办法，是在 my.cnf 中设置以下语句：

```
[mysql]
default-character-set=gbk
```

这样服务器启动后，所有连接默认就是使用 GBK 字符集进行连接的，而不需要在程序中再执行 set names 命令。
另外，字符串常量的字符集也是由 character_set_connection 参数来指定的。
可以通过“[_charset_name]'string' [COLLATE collation_name]”命令强制字符串的字符集和校对规则。例如：

```
select _gbk '字符集';
select _latin1 '字符集';
```

通常情况下，基本不需要用户强制指定字符串字符集。

9.7 字符集的修改步骤

如果在应用开始阶段没有正确的设置字符集，在运行一段时间以后才发现存在不能满足要求需要调整，又不想丢弃这段时间的数据，那么就需要进行字符集的修改。字符集的修改不能直接通过“alter database character set ***”或者“alter table tablename character set ***”命令进行，这两个命令都没有更新已有记录的字符集，而只是对新创建的表或者记录生效。已有记录的字符集调整，需要先将数据导出，经过适当的调整重新导入后才可完成。
以下模拟的是将 latin1 字符集的数据库修改成 GBK 字符集的数据库的过程。
（1）导出表结构：

```
mysqldump -uroot -p --default-character-set=gbk -d databasename> createtab.sql
```

其中--default-character-set=gbk 表示设置以什么字符集连接，-d 表示只导出表结构，不导出数据。

（2）手工修改 createtab.sql 中表结构定义中的字符集为新的字符集。

（3）确保记录不再更新，导出所有记录。

```
mysqldump -uroot -p --quick --no-create-info --extended-insert
--default-character-set=latin1  databasename> data.sql
```

- --quick：该选项用于转储大的表。它强制 mysqldump 从服务器一次一行地检索表中的行而不是检索所有行，并在输出前将它缓存到内存中。
- --extended-insert：使用包括几个 VALUES 列表的多行 INSERT 语法。这样使转储文件更小，重载文件时可以加速插入。
- --no-create-info：不写重新创建每个转储表的 CREATE TABLE 语句。
- --default-character-set=latin1：按照原有的字符集导出所有数据，这样导出的文件中，所有中文都是可见的，不会保存成乱码。

（4）打开 data.sql，将 SET NAMES latin1 修改成 SET NAMES gbk。

（5）使用新的字符集创建新的数据库。

```
create database databasename default charset gbk;
```

（6）创建表，执行 createtab.sql。

```
mysql -uroot -p databasename < createtab.sql
```

（7）导入数据，执行 data.sql。

```
mysql -uroot -p databasename < data.sql
```

> 注意：选择目标字符集的时候，要注意最好是源字符集的超级，或者确定比源字符集的字库更大，否则如果目标字符集的字库小于源字符集的字库，那么目标字符集中不支持的字符倒入后会变成乱码，丢失一部分数据。例如，GBK 字符集的字库大于 GB2312 字符集，那么 GBK 字符集的数据，如果导入 GB2312 数据库中，就会丢失 GB2312 中不支持的那部分汉字的数据。

9.8 小结

这一章主要介绍了 MySQL 中字符集和校对规则的概念、设置方法，以及推荐读者使用的字符集。最后，举例介绍了字符集修改的步骤和修改过程中遇到过的问题，希望会对读者有所帮助。

第10章　索引的设计和使用

索引是数据库中用来提高性能的最常用工具。本章主要介绍了 MySQL 5.0 支持的索引类型，并简单介绍了索引的设计原则。在后面的优化篇中，将会对索引做更多的介绍。

10.1 索引概述

所有 MySQL 列类型都可以被索引，对相关列使用索引是提高 SELECT 操作性能的最佳途径。根据存储引擎可以定义每个表的最大索引数和最大索引长度，每种存储引擎（如 MyISAM、InnoDB、BDB、MEMORY 等）对每个表至少支持 16 个索引，总索引长度至少为 256 字节。大多数存储引擎有更高的限制。

MyISAM 和 InnoDB 存储引擎的表默认创建的都是 BTREE 索引。MySQL 目前还不支持函数索引，但是支持前缀索引，即对索引字段的前 N 个字符创建索引。前缀索引的长度跟存储引擎相关，对于 MyISAM 存储引擎的表，索引的前缀长度可以达到 1000 字节长，而对于 InnoDB 存储引擎的表，索引的前缀长度最长是 767 字节。请注意前缀的限制应以字节为单位进行测量，而 CREATE TABLE 语句中的前缀长度解释为字符数。在为使用多字节字符集的列指定前缀长度时一定要加以考虑。

MySQL 中还支持全文本（FULLTEXT）索引，该索引可以用于全文搜索。但是当前最新版本中（5.0）只有 MyISAM 存储引擎支持 FULLTEXT 索引，并且只限于 CHAR、VARCHAR 和 TEXT 列。索引总是对整个列进行的，不支持局部（前缀）索引。

也可以为空间列类型创建索引，但是只有 MyISAM 存储引擎支持空间类型索引，且索引的字段必须是非空的。

默认情况下，MEMORY 存储引擎使用 HASH 索引，但也支持 BTREE 索引。

索引在创建表的时候可以同时创建，也可以随时增加新的索引。创建新索引的语法为：

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
   [USING index_type]
   ON tbl_name (index_col_name,...)

index_col_name:
   col_name [(length)] [ASC | DESC]
```

也可以使用 ALTER TABLE 的语法来增加索引，语法可 CREATE INDEX 类似，可以查询帮助获得详细的语法，这里不再复述。

例如，要为 city 表创建了 10 个字节的前缀索引，语法是：

```
mysql> create index cityname on city (city(10));
Query OK, 600 rows affected (0.26 sec)
Records: 600  Duplicates: 0  Warnings: 0
```

如果以 city 为条件进行查询，可以发现索引 cityname 被使用：

```
mysql> explain select * from city where city = 'Fuzhou' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: city
         type: ref
possible_keys: cityname
          key: cityname
      key_len: 32
          ref: const
         rows: 1
        Extra: Using where
```

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

索引的删除语法为：

```
    DROP INDEX index_name ON tbl_name
```

例如，想要删除 city 表上的索引 cityname，可以操作如下：

```
    mysql> drop index cityname on city;
    Query OK, 600 rows affected (0.23 sec)
    Records: 600  Duplicates: 0  Warnings: 0
```

10.2 设计索引的原则

索引的设计可以遵循一些已有的原则，创建索引的时候请尽量考虑符合这些原则，便于提升索引的使用效率，更高效地使用索引。

- 搜索的索引列，不一定是所要选择的列。换句话说，最适合索引的列是出现在 WHERE 子句中的列，或连接子句中指定的列，而不是出现在 SELECT 关键字后的选择列表中的列。
- 使用惟一索引。考虑某列中值的分布。索引的列的基数越大，索引的效果越好。例如，存放出生日期的列具有不同值，很容易区分各行。而用来记录性别的列，只含有“ M”和“F”，则对此列进行索引没有多大用处，因为不管搜索哪个值，都会得出大约一半的行。
- 使用短索引。如果对字符串列进行索引，应该指定一个前缀长度，只要有可能就应该这样做。例如，如果有一个 CHAR(200)列，如果在前 10 个或 20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。对前 10 个或 20 个字符进行索引能够节省大量索引空间，也可能会使查询更快。较小的索引涉及的磁盘 IO 较少，较短的值比较起来更快。更为重要的是，对于较短的键值，索引高速缓存中的块能容纳更多的键值，因此，MySQL 也可以在内存中容纳更多的值。这样就增加了找到行而不用读取索引中较多块的可能性。
- 利用最左前缀。在创建一个 n 列的索引时，实际是创建了 MySQL 可利用的 n 个索引。多列索引可起几个索引的作用，因为可利用索引中最左边的列集来匹配行。这样的列集称为最左前缀。
- 不要过度索引。不要以为索引“越多越好”，什么东西都用索引是错误的。每个额外的索引都要占用额外的磁盘空间，并降低写操作的性能。在修改表的内容时，索引必须进行更新，有时可能需要重构，因此，索引越多，所花的时间越长。如果有一个索引很少利用或从不使用，那么会不必要地减缓表的修改速度。此外，MySQL 在生成一个执行计划时，要考虑各个索引，这也要花费时间。创建多余的索引给查询优化带来了更多的工作。索引太多，也可能会使 MySQL 选择不到所要使用的最好索引。只保持所需的索引有利于查询优化。
- 对于 InnoDB 存储引擎的表，记录默认会按照一定的顺序保存，如果有明确定义的主键，则按照主键顺序保存。如果没有主键，但是有唯一索引，那么就是按照唯一索引的顺序保存。如果既没有主键又没有唯一索引，那么表中会自动生成一个内部列，按照这个列的顺序保存。按照主键或者内部列进行的访问是最快的，所以 InnoDB 表尽量自己指定主键，当表中同时有几个列都是唯一的，都可以作为主键的时候，要选择最常作为访问条件的列作为主键，提高查询的效率。另外，还需要注意，InnoDB 表的普通索引都会保存主键的键值，所以主键要尽可能选择较短的数据类型，可以有效地减少索引的磁盘占用，提高索引的缓存效果。

10.3 BTREE 索引与 HASH 索引

MEMORY 存储引擎的表可以选择使用 BTREE 索引或者 HASH 索引，两种不同类型的索引各有其不同的适用范围。HASH 索引有一些重要的特征需要在使用的时候特别注意，如下所示。

- 只用于使用=或<=>操作符的等式比较。
- 优化器不能使用 HASH 索引来加速 ORDER BY 操作。
- MySQL 不能确定在两个值之间大约有多少行。如果将一个 MyISAM 表改为 HASH 索引的 MEMORY 表，会影响一些查询的执行效率。
- 只能使用整个关键字来搜索一行。

而对于 BTREE 索引，当使用>、<、>=、<=、BETWEEN、!=或者<>，或者 LIKE 'pattern'（其中'pattern'不以通配符开始）操作符时，都可以使用相关列上的索引。

下列范围查询适用于 BTREE 索引和 HASH 索引：

```
SELECT * FROM t1 WHERE key_col = 1 OR key_col IN (15,18,20);
```

下列范围查询只适用于 BTREE 索引：

```
SELECT * FROM t1 WHERE key_col > 1 AND key_col < 10;
SELECT * FROM t1 WHERE key_col LIKE 'ab%' OR key_col BETWEEN 'lisa' AND 'simon';
```

例如，创建一个和 city 表完全相同的 MEMORY 存储引擎的表 city_memory：

```
mysql> CREATE TABLE city_memory (
    ->   city_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    ->   city VARCHAR(50) NOT NULL,
    ->   country_id SMALLINT UNSIGNED NOT NULL,
    ->      last_update  TIMESTAMP  NOT  NULL  DEFAULT  CURRENT_TIMESTAMP  ON  UPDATE
CURRENT_TIMESTAMP,
    ->   PRIMARY KEY  (city_id),
    ->   KEY idx_fk_country_id (country_id)
    -> )ENGINE=Memory DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.03 sec)

mysql> insert into city_memory select * from city;
Query OK, 600 rows affected (0.00 sec)
Records: 600  Duplicates: 0  Warnings: 0
```

当对索引字段进行范围查询的时候，只有 BTREE 索引可以通过索引访问：

```
mysql> explain SELECT * FROM city WHERE country_id > 1 and country_id < 10 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: city
         type: range
possible_keys: idx_fk_country_id
          key: idx_fk_country_id
```

```
        key_len: 2
            ref: NULL
           rows: 24
          Extra: Using where
1 row in set (0.00 sec)
```

而 HASH 索引实际上是全表扫描的：

```
mysql> explain SELECT * FROM city_memory WHERE country_id > 1 and country_id < 10 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: city_memory
         type: ALL
possible_keys: idx_fk_country_id
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 600
        Extra: Using where
1 row in set (0.00 sec)
```

了解了 BTREE 索引和 HASH 索引不同后，当使用 MEMORY 表的时候，如果是默认创建的 HASH 索引，就要注意 SQL 语句的编写，确保可以使用上索引，如果一定要使用范围查询，那么在创建索引的时候，就应该选择创建成 BTREE 索引。

10.4 小结

索引用于快速找出在某个列中有一特定值的行。如果不使用索引，MySQL 必须从第 1 条记录开始然后读完整个表直到找出相关的行。表越大，花费的时间越多。如果表中查询的列有一个索引，MySQL 能快速到达一个位置去搜寻数据文件的中间，没有必要看所有数据。如果一个表有 1000 行，这比顺序读取至少快 100 倍。注意如果需要访问大部分行，顺序读取要快得多，因为此时应避免磁盘搜索。

大多数 MySQL 索引（如 PRIMARY KEY、UNIQUE、INDEX 和 FULLTEXT 等）在 BTREE 中存储。只是空间列类型的索引使用 RTREE，并且 MEMORY 表还支持 HASH 索引。

关于什么情况下数据库会使用索引，以及什么情况下数据库不会使用索引的详细介绍，可参见优化篇的相关章节，这里不再赘述。

第11章　视图

MySQL 从 5.0.1 版本开始提供视图功能，本章将对 MySQL 中的视图进行介绍。

11.1 什么是视图

视图（View）是一种虚拟存在的表，对于使用视图的用户来说基本上是透明的。视图并不在数据库中实际存在，行和列数据来自定义视图的查询中使用的表，并且是在使用视图时动态生成的。

视图相对于普通的表的优势主要包括以下几项。

- 简单：使用视图的用户完全不需要关心后面对应的表的结构、关联条件和筛选条件，对用户来说已经是过滤好的复合条件的结果集。
- 安全：使用视图的用户只能访问他们被允许查询的结果集，对表的权限管理并不能限制到某个行某个列，但是通过视图就可以简单的实现。
- 数据独立：一旦视图的结构确定了，可以屏蔽表结构变化对用户的影响，源表增加列对视图没有影响；源表修改列名，则可以通过修改视图来解决，不会造成对访问者的影响。

11.2 视图操作

视图的操作包括创建或者修改视图、删除视图，以及查看视图定义。

11.2.1 创建或者修改视图

创建视图需要有 CREATE VIEW 的权限，并且对于查询涉及的列有 SELECT 权限。如果使用 CREATE OR REPLACE 或者 ALTER 修改视图，那么还需要该视图的 DROP 权限。

创建视图的语法为：

```
CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
        VIEW view_name [(column_list)]
        AS select_statement
        [WITH [CASCADED | LOCAL] CHECK OPTION]
```

修改视图的语法为：

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
        VIEW view_name [(column_list)]
        AS select_statement
        [WITH [CASCADED | LOCAL] CHECK OPTION]
```

例如，要创建了视图 staff_list_view，可以使用以下命令：

```
mysql> CREATE OR REPLACE VIEW staff_list_view AS
    -> SELECT s.staff_id,s.first_name,s.last_name,a.address
    -> FROM staff AS s,address AS a
    -> where s.address_id = a.address_id ;
Query OK, 0 rows affected (0.00 sec)
```

MySQL 视图的定义有一些限制，例如，在 FROM 关键字后面不能包含子查询，这和其他数据库是不同的，如果视图是从其他数据库迁移过来的，那么可能需要因此做一些改动，可以将子查询的内容先定义成一个视图，然后对该视图再创建视图就可以实现类似的功能了。

视图的可更新性和视图中查询的定义有关系，以下类型的视图是不可更新的。

- 包含以下关键字的 SQL 语句：聚合函数（SUM、MIN、MAX、COUNT 等）、DISTINCT、GROUP BY、HAVING、UNION 或者 UNION ALL。
- 常量视图。
- SELECT 中包含子查询。
- JION。
- FROM 一个不能更新的视图。
- WHERE 字句的子查询引用了 FROM 字句中的表。

例如，以下的视图都是不可更新的：

```
--包含聚合函数
mysql> create or replace view payment_sum as
    -> select staff_id,sum(amount) from payment group by staff_id;
Query OK, 0 rows affected (0.00 sec)

--常量视图
mysql> create or replace view pi as select 3.1415926 as pi;
Query OK, 0 rows affected (0.00 sec)

--select 中包含子查询
mysql> create view city_view as
    -> select (select city from city where city_id = 1) ;
Query OK, 0 rows affected (0.00 sec)
```

WITH [CASCADED | LOCAL] CHECK OPTION 决定了是否允许更新数据使记录不再满足视图的条件。这个选项与 Oracle 数据库中的选项是类似的，其中：

- LOCAL 是只要满足本视图的条件就可以更新；
- CASCADED 则是必须满足所有针对该视图的所有视图的条件才可以更新。

如果没有明确是 LOCAL 还是 CASCADED，则默认是 CASCADED。

例如，对 payment 表创建两层视图，并进行更新操作：

```
mysql> create or replace view payment_view as
    -> select payment_id,amount from payment
    -> where amount < 10 WITH CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> create or replace view payment_view1 as
    -> select payment_id,amount from payment_view
    -> where amount > 5 WITH LOCAL CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> create or replace view payment_view2 as
    -> select payment_id,amount from payment_view
```

```
    -> where amount > 5 WITH CASCADED CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from payment_view1 limit 1;
+------------+--------+
                                        ?
| payment_id | amount |
+------------+--------+
| 3          | 5.99   |
+------------+--------+
1 row in set (0.00 sec)

mysql> update payment_view1 set amount=10
    -> where payment_id = 3;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update payment_view2 set amount=10
    -> where payment_id = 3;
ERROR 1369 (HY000): CHECK OPTION failed 'sakila.payment_view2'
```

从测试结果可以看出，payment_view1 是 WITH LOCAL CHECK OPTION 的，所以只要满足本视图的条件就可以更新，但是 payment_view2 是 WITH CASCADED CHECK OPTION 的，必须满足针对该视图的所有视图才可以更新，因为更新后记录不再满足 payment_view 的条件，所以更新操作提示错误退出。

11.2.2 删除视图

用户可以一次删除一个或者多个视图，前提是必须有该视图的 DROP 权限。

```
DROP VIEW [IF EXISTS] view_name [, view_name] ...[RESTRICT | CASCADE]
```

例如，删除 staff_list 视图：

```
mysql> drop view staff_list;
Query OK, 0 rows affected (0.00 sec)
```

11.2.3 查看视图

从 MySQL 5.1 版本开始，使用 SHOW TABLES 命令的时候不仅显示表的名字，同时也会显示视图的名字，而不存在单独显示视图的 SHOW VIEWS 命令。

```
mysql> use sakila
Database changed
mysql> show tables;
+---------------------------+
| Tables_in_sakila          |
+---------------------------+
……
```

```
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
26 rows in set (0.00 sec)
```

同样，在使用 SHOW TABLE STATUS 命令的时候，不但可以显示表的信息，同时也可以显示视图的信息。所以，可以通过下面的命令显示视图的信息：

```
SHOW TABLE STATUS [FROM db_name] [LIKE 'pattern']
```

下面演示的是查看 staff_list 视图信息的操作：

```
mysql> show table status like 'staff_list' \G
*************************** 1. row ***************************
           Name: staff_list
         Engine: NULL
        Version: NULL
     Row_format: NULL
           Rows: NULL
 Avg_row_length: NULL
    Data_length: NULL
Max_data_length: NULL
   Index_length: NULL
      Data_free: NULL
 Auto_increment: NULL
    Create_time: NULL
    Update_time: NULL
     Check_time: NULL
      Collation: NULL
       Checksum: NULL
 Create_options: NULL
        Comment: VIEW
1 row in set (0.01 sec)
```

如果需要查询某个视图的定义，可以使用 SHOW CREATE VIEW 命令进行查看：

```
mysql> show create view staff_list \G
*************************** 1. row ***************************
       View: staff_list
Create View: CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL SECURITY DEFINER VIEW `staff_list` AS select `s`.`staff_id` AS `ID`,concat(`s`.`first_name`,_utf8' ',`s`.`last_name`) AS `name`,`a`.`address` AS `address`,`a`.`postal_code` AS `zip code`,`a`.`phone` AS `phone`,`city`.`city` AS `city`,`country`.`country` AS `country`,`s`.`store_id` AS `SID` from (((`staff` `s` join `address` `a` on((`s`.`address_id` = `a`.`address_id`))) join `city` on((`a`.`city_id` = `city`.`city_id`))) join `country` on((`city`.`country_id` = `country`.`country_id`))
1 row in set (0.00 sec)
```

最后，通过查看系统表 information_schema.views 也可以查看视图的相关信息：

```
    mysql> select * from views where table_name = 'staff_list' \G
    *************************** 1. row ***************************
      TABLE_CATALOG: NULL
       TABLE_SCHEMA: sakila
         TABLE_NAME: staff_list
    VIEW_DEFINITION: select `s`.`staff_id` AS `ID`,concat(`s`.`first_name`,_utf8'
',`s`.`last_name`) AS `name`,`a`.`address` AS `address`,`a`.`postal_code` AS `zip
code`,`a`.`phone` AS `phone`,`sakila`.`city`.`city` AS `city`,`sakila`.`country`.`country` AS
`country`,`s`.`store_id` AS `SID` from (((`sakila`.`staff` `s` join `sakila`.`address` `a`
on((`s`.`address_id` = `a`.`address_id`))) join `sakila`.`city` on((`a`.`city_id` =
`sakila`.`city`.`city_id`))) join `sakila`.`country` on((`sakila`.`city`.`country_id` =
`sakila`.`country`.`country_id`)))
       CHECK_OPTION: NONE
       IS_UPDATABLE: YES
            DEFINER: root@localhost
      SECURITY_TYPE: DEFINER
    1 row in set (0.00 sec)
```

11.3 小结

本章主要介绍了 MySQL 提供的视图创建、维护等相关语法。如果从不支持视图的旧版本升级到提供视图功能的新版本后，要想使用视图，则需要升级授权表，使之包含与视图有关的权限。相关的升级步骤，可以参见 27.4 节 MySQL 升级内容。

第12章　存储过程和函数

MySQL 从 5.0 版本开始支持存储过程和函数。

12.1 什么是存储过程和函数

存储过程和函数是事先经过编译并存储在数据库中的一段 SQL 语句的集合，调用存储过程和函数可以简化应用开发人员的很多工作，减少数据在数据库和应用服务器之间的传输，对于提高数据处理的效率是有好处的。

存储过程和函数的区别在于函数必须有返回值，而存储过程没有，存储过程的参数可以使用 IN、OUT、INOUT 类型，而函数的参数只能是 IN 类型的。如果有函数从其他类型的数据库迁移到 MySQL，那么就可能因此需要将函数改造成存储过程。

12.2 存储过程和函数的相关操作

在对存储过程或函数进行操作时，需要首先确认用户是否具有相应的权限。例如，创建存储过程或者函数需要 CREATE ROUTINE 权限，修改或者删除存储过程或者函数需要 ALTER ROUTINE 权限，执行存储过程或者函数需要 EXECUTE 权限。

12.2.1 创建、修改存储过程或者函数

创建、修改存储过程或者函数的语法：

```
CREATE PROCEDURE sp_name ([proc_parameter[,...]])
       [characteristic ...] routine_body

CREATE FUNCTION sp_name ([func_parameter[,...]])
       RETURNS type
       [characteristic ...] routine_body

        proc_parameter:
       [ IN | OUT | INOUT ] param_name type

        func_parameter:
       param_name type

type:
       Any valid MySQL data type

characteristic:
       LANGUAGE SQL
   | [NOT] DETERMINISTIC
   | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
   | SQL SECURITY { DEFINER | INVOKER }
   | COMMENT 'string'

routine_body:
       Valid SQL procedure statement or statements

ALTER {PROCEDURE | FUNCTION} sp_name [characteristic ...]

characteristic:
       { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
   | SQL SECURITY { DEFINER | INVOKER }
   | COMMENT 'string'
```

调用过程的语法如下：

```
CALL sp_name([parameter[,...]])
```

MySQL 的存储过程和函数中允许包含 DDL 语句，也允许在存储过程中执行提交（Commit，即确认之前的修改）或者回滚（Rollback，即放弃之前的修改），但是存储过程和函数中不允许执行 LOAD DATA INFILE 语句。此外，存储过程和函数中可以调用其他的过程或者函数。

下面创建了一个新的过程 film_in_stock：

```
mysql> DELIMITER $$
mysql>
mysql> CREATE PROCEDURE film_in_stock(IN p_film_id INT, IN p_store_id INT, OUT p_film_count INT)
    -> READS SQL DATA
    -> BEGIN
    ->     SELECT inventory_id
    ->     FROM inventory
    ->     WHERE film_id = p_film_id
    ->     AND store_id = p_store_id
    ->     AND inventory_in_stock(inventory_id);
    ->
    ->     SELECT FOUND_ROWS() INTO p_film_count;
    -> END $$
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> DELIMITER ;
```

上面是在使用的样例数据库中创建的一个过程，该过程用来检查 film_id 和 store_id 对应的 inventory 是否满足要求，并且返回满足要求的 inventory_id 以及满足要求的记录数。

通常我们在执行创建过程和函数之前，都会通过“DELIMITER $$”命令将语句的结束符从“;”修改成其他符号，这里使用的是“$$”，这样在过程和函数中的“;”就不会被 MySQL 解释成语句的结束而提示错误。在存储过程或者函数创建完毕，通过“DELIMITER ;”命令再将结束符改回成“;”。

可以看到在这个过程中调用了函数 inventory_in_stock()，并且这个过程有两个输入参数和一个输出参数。下面可以通过调用这个过程来看看返回的结果。

如果需要检查 film_id=2 store_id=2 对应的 inventory 的情况，则首先手工执行过程中的 SQL 语句，以查看执行的效果：

```
mysql> SELECT inventory_id
    -> FROM inventory
    -> WHERE film_id = 2
    -> AND store_id = 2
    -> AND inventory_in_stock(inventory_id);
+--------------+
| inventory_id |
+--------------+
| 10           |
| 11           |
+--------------+
```

```
2 rows in set (0.00 sec)
```

满足条件的记录应该是两条，inventory_id 分别是 10 和 11。如果将这个查询封装在存储过程中调用，那么调用过程的执行情况如下：

```
mysql> CALL film_in_stock(2,2,@a);
+--------------+
| inventory_id |
+--------------+
| 10           |
| 11           |
+--------------+
2 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> select @a;
+------+
| @a   |
+------+
| 2    |
+------+
1 row in set (0.00 sec)
```

可以看到调用存储过程与直接执行 SQL 的效果是相同的，但是存储过程的好处在于处理逻辑都封装在数据库端，调用者不需要了解中间的处理逻辑，一旦处理逻辑发生变化，只需要修改存储过程即可，而对调用者的程序完全没有影响。

另外，和视图的创建语法稍有不同，存储过程和函数的 CREATE 语法不支持使用 CREATE OR REPLACE 对存储过程和函数进行修改，如果需要对已有的存储过程或者函数进行修改，需要执行 ALTER 语法。

下面对 characteristic 特征值的部分进行简单的说明。

- LANGUAGE SQL：说明下面过程的 BODY 是使用 SQL 语言编写，这条是系统默认的，为今后 MySQL 会支持的除 SQL 外的其他语言支持的存储过程而准备。
- [NOT] DETERMINISTIC：DETERMINISTIC 确定的，即每次输入一样输出也一样的程序，NOT DETERMINISTIC 非确定的，默认是非确定的。当前，这个特征值还没有被优化程序使用。
- { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }：这些特征值提供子程序使用数据的内在信息，这些特征值目前只是提供给服务器，并没有根据这些特征值来约束过程实际使用数据的情况。CONTAINS SQL 表示子程序不包含读或写数据的语句。NO SQL 表示子程序不包含 SQL 语句。READS SQL DATA 表示子程序包含读数据的语句，但不包含写数据的语句。MODIFIES SQL DATA 表示子程序包含写数据的语句。如果这些特征没有明确给定，默认使用的值是 CONTAINS SQL。
- SQL SECURITY { DEFINER | INVOKER }：可以用来指定子程序该用创建子程序者的许可来执行，还是使用调用者的许可来执行。默认值是 DEFINER。
- COMMENT 'string'：存储过程或者函数的注释信息。

下面的例子对比了 SQL SECURITY 特征值的不同，使用 root 用户创建了两个相似的存储过程，

分别指定使用创建者的权限执行和调用者的权限执行，然后使用一个普通用户调用这两个存储过程，对比执行的效果：

首先用 root 用户创建以下两个存储过程 film_in_stock_definer 和 film_in_stock_invoker：

```
mysql> DELIMITER $$
mysql>
mysql> CREATE PROCEDURE film_in_stock_definer(IN p_film_id INT, IN p_store_id INT, OUT p_film_count INT)
    -> SQL SECURITY DEFINER
    -> BEGIN
    ->     SELECT inventory_id
    ->     FROM inventory
    ->     WHERE film_id = p_film_id
    ->     AND store_id = p_store_id
    ->     AND inventory_in_stock(inventory_id);
    ->
    ->     SELECT FOUND_ROWS() INTO p_film_count;
    -> END $$
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> CREATE PROCEDURE film_in_stock_invoker(IN p_film_id INT, IN p_store_id INT, OUT p_film_count INT)
    -> SQL SECURITY INVOKER
    -> BEGIN
    ->     SELECT inventory_id
    ->     FROM inventory
    ->     WHERE film_id = p_film_id
    ->     AND store_id = p_store_id
    ->     AND inventory_in_stock(inventory_id);
    ->
    ->     SELECT FOUND_ROWS() INTO p_film_count;
    -> END $$
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> DELIMITER ;
```

给普通用户 lisa 赋予可以执行存储过程的权限，但是不能查询 inventory 表：

```
mysql> GRANT EXECUTE ON sakila.* TO 'lisa'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

使用 lisa 登录后，直接查询 inventory 表会提示查询被拒绝：

```
mysql> select count(*) from inventory;
ERROR 1142 (42000): SELECT command denied to user 'lisa'@'localhost' for table 'inventory'
```

lisa 用户分别调用 film_in_stock_definer 和 film_in_stock_invoker：

```
mysql> CALL film_in_stock_definer(2,2,@a);
+--------------+
| inventory_id |
+--------------+
| 10           |
| 11           |
+--------------+
2 rows in set (0.03 sec)

Query OK, 0 rows affected (0.03 sec)

mysql> CALL film_in_stock_invoker(2,2,@a);
ERROR 1142 (42000): SELECT command denied to user 'lisa'@'localhost' for table 'inventory'
```

从上面的例子可以看出，film_in_stock_definer 是以创建者的权限执行的，因为是 root 用户创建的，所以可以访问 inventory 表的内容，film_in_stock_invoker 是以调用者的权限执行的，lisa 用户没有访问 inventory 表的权限，所以会提示权限不足。

12.2.2 删除存储过程或者函数

一次只能删除一个存储过程或者函数，删除存储过程或者函数需要有该过程或者函数的 ALTER ROUTINE 权限，具体语法如下：

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

例如，使用 DROP 语法删除 film_in_stock 过程：

```
mysql> DROP PROCEDURE film_in_stock;
Query OK, 0 rows affected (0.00 sec)
```

12.2.3 查看存储过程或者函数

存储过程或者函数创建后，用户可能需要查看存储过程或者函数的状态或者定义等信息，便于了解存储过程或者函数的基本情况。下面将介绍如何查看存储过程或函数相关信息。

1. 查看存储过程或者函数的状态

```
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

下面演示的是查看过程 film_in_stock 的信息：

```
mysql> show procedure status like 'film_in_stock'\G
*************************** 1. row ***************************
           Db: sakila
         Name: film_in_stock
         Type: PROCEDURE
      Definer: root@localhost
     Modified: 2007-07-06 09:29:00
```

```
      Created: 2007-07-06 09:29:00
Security_type: DEFINER
      Comment:
1 row in set (0.00 sec)
```

2. 查看存储过程或者函数的定义

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name
```

下面演示的是查看过程 film_in_stock 的定义：

```
mysql> show create procedure film_in_stock \G
*************************** 1. row ***************************
       Procedure: film_in_stock
        sql_mode:
Create Procedure: CREATE DEFINER=`root`@`localhost` PROCEDURE `film_in_stock`(IN p_film_id
INT, IN p_store_id INT, OUT p_film_count INT)
    READS SQL DATA
BEGIN
     SELECT inventory_id
     FROM inventory
     WHERE film_id = p_film_id
     AND store_id = p_store_id
     AND inventory_in_stock(inventory_id);
     SELECT FOUND_ROWS() INTO p_film_count;
END
1 row in set (0.00 sec)
```

3、通过查看 information_schema. Routines 了解存储过程和函数的信息

除了以上两种方法，我们还可以查看系统表来了解存储过程和函数的相关信息，通过查看 information_schema. Routines 就可以获得存储过程和函数的包括名称、类型、语法、创建人等信息。

例如，通过查看 information_schema. Routines 得到过程 film_in_stock 的定义：

```
mysql> select *  from routines where ROUTINE_NAME = 'film_in_stock' \G
*************************** 1. row ***************************
     SPECIFIC_NAME: film_in_stock
   ROUTINE_CATALOG: NULL
    ROUTINE_SCHEMA: sakila
      ROUTINE_NAME: film_in_stock
      ROUTINE_TYPE: PROCEDURE
    DTD_IDENTIFIER: NULL
```

```
          ROUTINE_BODY: SQL
    ROUTINE_DEFINITION: BEGIN
         SELECT inventory_id
         FROM inventory
         WHERE film_id = p_film_id
         AND store_id = p_store_id
         AND inventory_in_stock(inventory_id);
         SELECT FOUND_ROWS() INTO p_film_count;
    END
         EXTERNAL_NAME: NULL
     EXTERNAL_LANGUAGE: NULL
       PARAMETER_STYLE: SQL
      IS_DETERMINISTIC: NO
       SQL_DATA_ACCESS: READS SQL DATA
             SQL_PATH: NULL
         SECURITY_TYPE: DEFINER
               CREATED: 2007-07-06 09:29:00
          LAST_ALTERED: 2007-07-06 09:29:00
              SQL_MODE:
       ROUTINE_COMMENT:
               DEFINER: root@localhost
    1 row in set (0.00 sec)
```

12.2.4 变量的使用

存储过程和函数中可以使用变量，而且在 MySQL 5.1 版本中，变量是不区分大小写的。

1. 变量的定义

通过 DECLARE 可以定义一个局部变量，该变量的作用范围只能在 BEGIN…END 块中，可以用在嵌套的块中。变量的定义必须写在复合语句的开头，并且在任何其他语句的前面。可以一次声明多个相同类型的变量。如果需要，可以使用 DEFAULT 赋默认值。

定义一个变量的语法如下：

```
    DECLARE var_name[,...] type [DEFAULT value]
```

例如，定义一个 DATE 类型的变量，名称是 last_month_start：

```
    DECLARE last_month_start DATE;
```

2. 变量的赋值

变量可以直接赋值，或者通过查询赋值。

直接赋值使用 SET，可以赋常量或者赋表达式，具体语法如下：

```
    SET var_name = expr [, var_name = expr] ...
```

给刚才定义的变量 last_month_start 赋值，具体语法如下：

```
SET last_month_start = DATE_SUB(CURRENT_DATE(), INTERVAL 1 MONTH);
```

也可以通过查询将结果赋给变量，这要求查询返回的结果必须只有一行，具体语法如下：

```
SELECT col_name[,...] INTO var_name[,...] table_expr
```

通过查询将结果赋值给变量 v_payments：

```
CREATE FUNCTION get_customer_balance(p_customer_id INT,
p_effective_date DATETIME)
RETURNS DECIMAL(5,2)
DETERMINISTIC
READS SQL DATA
BEGIN
   …
   DECLARE v_payments DECIMAL(5,2); #SUM OF PAYMENTS MADE PREVIOUSLY
   …
   SELECT IFNULL(SUM(payment.amount),0) INTO v_payments
   FROM payment
   WHERE payment.payment_date <= p_effective_date
   AND payment.customer_id = p_customer_id;
   …
   RETURN v_rentfees + v_overfees - v_payments;
END $$
```

12.2.5 定义条件和处理

条件的定义和处理可以用来定义在处理过程中遇到问题时相应的处理步骤。

1. 条件的定义

```
DECLARE condition_name CONDITION FOR condition_value

condition_value:
       SQLSTATE [VALUE] sqlstate_value
   | mysql_error_code
```

2. 条件的处理

```
DECLARE handler_type HANDLER FOR condition_value[,...] sp_statement

handler_type:
       CONTINUE
   | EXIT
   | UNDO

condition_value:
```

```
        SQLSTATE [VALUE] sqlstate_value
    | condition_name
    | SQLWARNING
    | NOT FOUND
    | SQLEXCEPTION
    | mysql_error_code
```

下面将通过两个例子来说明：在向 actor 表中插入记录时，如果没有进行条件的处理，那么在主键重的时候会抛出异常并退出，如果对条件进行了处理，那么就不会再抛出异常。

（1）当没有进行条件处理时，执行结果如下：

```
mysql> select max(actor_id) from actor;
+---------------+
| max(actor_id) |
+---------------+
| 200           |
+---------------+
1 row in set (0.00 sec)

mysql> delimiter $$
mysql>
mysql> CREATE PROCEDURE actor_insert ()
    -> BEGIN
    ->   SET @x = 1;
    ->   INSERT INTO actor(actor_id,first_name,last_name) VALUES (201,'Test','201');
    ->   SET @x = 2;
    ->   INSERT INTO actor(actor_id,first_name,last_name) VALUES (1,'Test','1');
    ->   SET @x = 3;
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call actor_insert();
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
mysql> select @x;
+------+
| @x   |
+------+
| 2    |
+------+
1 row in set (0.00 sec)
```

从上面的例子可以看出，执行到插入 actor_id=1 的记录时，会主键重并退出，没有执行到下面其他的语句。

（2）当对主键重的异常进行处理时，执行结果如下：

```
mysql> delimiter $$
mysql>
mysql> CREATE PROCEDURE actor_insert ()
    -> BEGIN
    ->   DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
    ->   SET @x = 1;
    ->   INSERT INTO actor(actor_id,first_name,last_name) VALUES (201,'Test','201');
    ->   SET @x = 2;
    ->   INSERT INTO actor(actor_id,first_name,last_name) VALUES (1,'Test','1');
    ->   SET @x = 3;
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call actor_insert();
Query OK, 0 rows affected (0.06 sec)

mysql> select @x,@x2;
+------+------+
| @x   | @x2  |
+------+------+
| 3    | 1    |
+------+------+
1 row in set (0.00 sec)
```

调用条件处理的过程，再遇到主键重的错误时，会按照定义的处理方式进行处理，由于例子中定义的是 CONTINUE，所以会继续执行下面的语句。

handler_type 现在还只支持 CONTINUE 和 EXIT 两种，CONTINUE 表示继续执行下面的语句，EXIT 则表示执行终止，UNDO 现在还不支持。

condition_value 的值可以是通过 DECLARE 定义的 condition_name，可以是 SQLSTATE 的值或者 mysql-error-code 的值或者 SQLWARNING、NOT FOUND、SQLEXCEPTION，这 3 个值是 3 种定义好的错误类别，分别代表不同的含义。

·SQLWARNING 是对所有以 01 开头的 SQLSTATE 代码的速记。

·NOT FOUND 是对所有以 02 开头的 SQLSTATE 代码的速记。

·SQLEXCEPTION 是对所有没有被 SQLWARNING 或 NOT FOUND 捕获的 SQLSTATE 代码的速记。

因此，上面的例子还可以写成以下几种方式：

```
--捕获 mysql-error-code:
DECLARE CONTINUE HANDLER FOR 1062 SET @x2 = 1;
--事先定义 condition_name:
DECLARE DuplicateKey CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR DuplicateKey SET @x2 = 1;
--捕获 SQLEXCEPTION
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET @x2 = 1;
```

12.2.6 光标的使用

在存储过程和函数中可以使用光标对结果集进行循环的处理。光标的使用包括光标的声明、OPEN、FETCH 和 CLOSE，其语法分别如下。

- 声明光标：

```
DECLARE cursor_name CURSOR FOR select_statement
```

- OPEN 光标：

```
OPEN cursor_name
```

- FETCH 光标：

```
FETCH cursor_name INTO var_name [, var_name] ...
```

- CLOSE 光标：

```
CLOSE cursor_name
```

以下例子是一个简单的使用光标的过程，对 payment 表按照行进行循环的处理，按照 staff_id 值的不同累加 amount 的值，判断循环结束的条件是捕获 NOT FOUND 的条件，当 FETCH 光标找不到下一条记录的时候，就会关闭光标然后退出过程。

```
mysql> delimiter $$
mysql>
mysql> CREATE PROCEDURE payment_stat ()
    -> BEGIN
    ->   DECLARE i_staff_id int;
    ->   DECLARE d_amount decimal(5,2);
    ->   DECLARE cur_payment cursor for select staff_id,amount from payment;
    ->   DECLARE EXIT HANDLER FOR NOT FOUND CLOSE cur_payment;
    ->
    ->   set  @x1 = 0;
    ->   set  @x2 = 0;
    ->
    ->   OPEN cur_payment;
    ->
    ->   REPEAT
    ->     FETCH cur_payment INTO i_staff_id, d_amount;
    ->        if i_staff_id = 2 then
    ->           set @x1 = @x1 + d_amount;
    ->        else
    ->           set @x2 = @x2 + d_amount;
    ->        end if;
    ->   UNTIL 0 END REPEAT;
    ->
    ->   CLOSE cur_payment;
    ->
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

```
mysql>
mysql> call payment_stat();
Query OK, 0 rows affected (0.11 sec)

mysql> select @x1,@x2;
+----------+----------+
| @x1      | @x2      |
+----------+----------+
| 33927.04 | 33489.47 |
+----------+----------+
1 row in set (0.00 sec)
```

> 注意：变量、条件、处理程序、光标都是通过 DECLARE 定义的，它们之间是有先后顺序的要求的。变量和条件必须在最前面声明，然后才能是光标的声明，最后才可以是处理程序的声明。

12.2.7 流程控制

可以使用 IF、CASE、LOOP、LEAVE、ITERATE、REPEAT 及 WHILE 语句进行流程的控制，下面将逐一进行说明。

1. IF 语句

IF 实现条件判断，满足不同的条件执行不同的语句列表，具体语法如下：

```
IF search_condition THEN statement_list
       [ELSEIF search_condition THEN statement_list] ...
       [ELSE statement_list]
END IF
```

12.2.6 小节中使用光标的例子中已经涉及了 IF 语句的使用，这里不再举例说明。

2. CASE 语句

CASE 实现比 IF 更复杂一些的条件构造，具体语法如下：

```
CASE case_value
       WHEN when_value THEN statement_list
       [WHEN when_value THEN statement_list] ...
       [ELSE statement_list]
END CASE
Or:
CASE
       WHEN search_condition THEN statement_list
       [WHEN search_condition THEN statement_list] ...
```

```
        [ELSE statement_list]
    END CASE
```

在上文光标的使用例子中，IF 语句也可以使用 CASE 语句来完成：

```
        case
          when i_staff_id = 2 then
           set @x1 = @x1 + d_amount;
          else
           set @x2 = @x2 + d_amount;
        end case;
    或者：
        case i_staff_id
          when 2 then
           set @x1 = @x1 + d_amount;
          else
           set @x2 = @x2 + d_amount;
        end case;
```

3．LOOP 语句

LOOP 实现简单的循环，退出循环的条件需要使用其他的语句定义，通常可以使用 LEAVE 语句实现，具体语法如下：

```
    [begin_label:] LOOP
        statement_list
    END LOOP [end_label]
```

如果不在 statement_list 中增加退出循环的语句，那么 LOOP 语句可以用来实现简单的死循环。

4．LEAVE 语句

用来从标注的流程构造中退出，通常和 BEGIN ... END 或者循环一起使用。

下面是一个使用 LOOP 和 LEAVE 的简单例子，循环 100 次向 actor 表中插入记录，当插入 100 条记录后，退出循环：

```
    mysql> CREATE PROCEDURE actor_insert ()
        -> BEGIN
        ->   set @x = 0;
        ->   ins: LOOP
        ->      set @x = @x + 1;
        ->      IF @x = 100 then
        ->      leave ins;
        ->      END IF;
        ->      INSERT INTO actor(first_name,last_name) VALUES ('Test','201');
        ->   END LOOP ins;
        -> END;
```

```
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> call actor_insert();
Query OK, 0 rows affected (0.01 sec)

mysql> select count(*) from actor where first_name='Test';
+----------+
| count(*) |
+----------+
| 100      |
+----------+
1 row in set (0.00 sec)
```

5. ITERATE 语句

ITERATE 语句必须用在循环中，作用是跳过当前循环的剩下的语句，直接进入下一轮循环。下面的例子使用了 ITERATE 语句，当@x 变量是偶数的时候，不再执行循环中剩下的语句，而直接进行下一轮的循环：

```
mysql> CREATE PROCEDURE actor_insert ()
    -> BEGIN
    ->   set @x = 0;
    ->   ins: LOOP
    ->     set @x = @x + 1;
    ->     IF @x = 10 then
    ->     leave ins;
    ->     ELSEIF mod(@x,2) = 0 then
    ->     ITERATE ins;
    ->     END IF;
    ->     INSERT INTO actor(actor_id,first_name,last_name) VALUES (@x+200,'Test',@x);
    ->   END LOOP ins;
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> call actor_insert();
Query OK, 0 rows affected (0.00 sec)

mysql> select actor_id,first_name,last_name from actor where first_name='Test';
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
| 201      | Test       | 1         |
| 203      | Test       | 3         |
```

```
| 205      | Test       | 5         |
| 207      | Test       | 7         |
| 209      | Test       | 9         |
+----------+------------+-----------+
5 rows in set (0.00 sec)
```

6. REPEAT 语句

有条件的循环控制语句，当满足条件的时候退出循环，具体语法如下：

```
[begin_label:] REPEAT
       statement_list
UNTIL search_condition
END REPEAT [end_label]
```

在“12.2.6 光标的使用”小节中的例子就使用 REPEAT 语句实现光标的循环获得，下面节选的代码就是其中使用 REPEAT 语句的部分，详细的执行过程请参照 12.2.6 小节，这里不再赘述。

```
    ->   REPEAT
    ->     FETCH cur_payment INTO i_staff_id, d_amount;
    ->        if i_staff_id = 2 then
    ->           set @x1 = @x1 + d_amount;
    ->        else
    ->           set @x2 = @x2 + d_amount;
    ->        end if;
    ->   UNTIL 0 END REPEAT;
```

7. WHILE 语句

WHILE 语句实现的也是有条件的循环控制语句，即当满足条件时执行循环的内容，具体语法如下：

```
[begin_label:] WHILE search_condition DO
       statement_list
END WHILE [end_label]
```

WHILE 循环和 REPEAT 循环的区别在于：WHILE 是满足条件才执行循环，REPEAT 是满足条件退出循环；WHILE 在首次循环执行之前就判断条件，所以循环最少执行 0 次，而 REPEAT 是在首次执行循环之后才判断条件，所以循环最少执行 1 次。

以下例子用来对比 REPEAT 和 WHILE 语句的功能：

```
mysql> delimiter $$
mysql> CREATE PROCEDURE loop_demo ()
    -> BEGIN
    ->     set @x = 1 , @x1 = 1;
    ->     REPEAT
    ->        set @x = @x + 1;
    ->     until @x > 0 end repeat;
```

```
    ->
    ->     while @x1 < 0 do
    ->         set @x1 = @x1 + 1;
    ->     end while;
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
mysql> call loop_demo();
Query OK, 0 rows affected (0.00 sec)

mysql> select @x,@x1;
+------+------+
| @x   | @x1  |
+------+------+
| 2    | 1    |
+------+------+
1 row in set (0.00 sec)
```

从判断的条件上看，初始值都是满足退出循环的条件的，但是 REPEAT 循环仍然执行了一次以后才退出循环的，而 WHILE 循环则一次都没有执行。

12.3 小结

本章主要介绍了存储过程和函数的创建、修改的方式，存储过程、函数的适用场合，并介绍了存储过程和函数中的变量、条件和处理、光标、流程控制的定义和使用，这些对初学者编写简单的存储过程和函数会有所帮助。虽然使用变量、条件和处理、光标和流程控制可以编写功能强大的存储过程和函数，并进行复杂的逻辑处理，但是由于篇幅问题，本章并没有对这部分内容进行深入，读者如果有兴趣的话，可以查询在线的 MySQL 文档获得帮助。

最后要强调的是，存储过程和函数的优势是可以将数据的处理放在数据库服务器上进行，避免将大量的结果集传输给客户端，减少数据的传输，但是在数据库服务器上进行大量的复杂运算也会占用服务器的 CPU，造成数据库服务器的压力，所以不要在存储过程和函数中进行大量的复杂运算，应尽量将这些运算操作分摊到应用服务器上执行。

第13章　触发器

MySQL 从 5.0.2 版本开始支持触发器的功能。触发器是与表有关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据的完整性。本章将详细介绍 MySQL 中触发器的使用方法。

13.1 创建触发器

创建触发器的语法如下：

```
CREATE TRIGGER trigger_name trigger_time trigger_event
       ON tbl_name FOR EACH ROW trigger_stmt
```

> 注意：触发器只能创建在永久表（Permanent Table）上，不能对临时表（Temporary Table）创建触发器。

其中 trigger_time 是触发器的触发时间，可以是 BEFORE 或者 AFTER，BEFORE 的含义指在检查约束前触发，而 AFTER 是在检查约束后触发。

而 trigger_event 就是触发器的触发事件，可以是 INSERT、UPDATE 或者 DELETE。

对同一个表相同触发时间的相同触发事件，只能定义一个触发器。例如，对某个表的不同字段的 AFTER 更新触发器，在使用 Oracle 数据库的时候，可以定义成两个不同的 UPDATE 触发器，更新不同的字段时触发单独的触发器，但是在 MYSQL 数据库中，只能定义成一个触发器，在触发器中通过判断更新的字段进行对应的处理。

使用别名 OLD 和 NEW 来引用触发器中发生变化的记录内容，这与其他的数据库是相似的。现在触发器还只支持行级触发的，不支持语句级触发。

在样例数据库中，为 film 表创建了 AFTER INSERT 的触发器，具体如下：

```
DELIMITER $$
CREATE TRIGGER ins_film
AFTER INSERT ON film FOR EACH ROW BEGIN
    INSERT INTO film_text (film_id, title, description)
        VALUES (new.film_id, new.title, new.description);
END;
$$
delimiter ;
```

插入 film 表记录的时候，会向 film_text 表中也插入相应的记录。

```
mysql> INSERT INTO film VALUES
    -> (1001,'ACADEMY DINOSAUR',
    ->'A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The Canadian Rockies',
    ->       2006,1,NULL,6,'0.99',86,'20.99','PG','Deleted       Scenes,Behind       the Scenes','2006-02-15 05:03:42');
Query OK, 1 row affected (0.05 sec)

mysql> select * from film_text where film_id=1001 \G
*************************** 1. row ***************************
    film_id: 1001
      title: ACADEMY DINOSAUR
description: A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in The Canadian Rockies
1 row in set (0.00 sec)
```

对于 INSERT INOT…ON DUPLICATE KEY UPDATE…语句来说，触发触发器的顺序可能会造成疑惑。下面对 film 表分别创建了 BEFROE INSERT、AFTER INSERT、BEFORE UPDATE、AFTER UPDATE

触发器，然后插入记录，观察触发器的触发情况：

```
--创建 BEFROE INSERT、AFTER INSERT、BEFORE UPDATE、AFTER UPDATE 触发器：
mysql> create table tri_demo(id int AUTO_INCREMENT,note varchar(20),PRIMARY KEY (id));
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TRIGGER ins_film_bef
    -> BEFORE INSERT ON film FOR EACH ROW BEGIN
    ->     INSERT INTO tri_demo (note) VALUES ('before insert');
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TRIGGER ins_film_aft
    -> AFTER INSERT ON film FOR EACH ROW BEGIN
    ->     INSERT INTO tri_demo (note) VALUES ('after insert');
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TRIGGER upd_film_bef
    -> BEFORE update ON film FOR EACH ROW BEGIN
    ->     INSERT INTO tri_demo (note) VALUES ('before update');
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TRIGGER upd_film_aft
    -> AFTER update ON film FOR EACH ROW BEGIN
    ->     INSERT INTO tri_demo (note) VALUES ('after update');
    -> END;
    -> $$
Query OK, 0 rows affected (0.00 sec)

--插入记录已经存在的情况：
mysql> select film_id,title from film where film_id = 1001;
+---------+-----------------+
| film_id | title           |
+---------+-----------------+
| 1001    | ACADEMY DINOSAUR |
+---------+-----------------+
1 row in set (0.00 sec)

mysql> INSERT INTO film VALUES
    -> (1001,'Only test',
```

```
    -> 'Only test',2006,1,NULL,6,'0.99',86,'20.99','PG',
    -> 'Deleted Scenes,Behind the Scenes','2006-02-15 05:03:42')
    -> ON DUPLICATE KEY
    -> UPDATE title='update record';
Query OK, 2 rows affected (0.05 sec)

mysql> select * from tri_demo;
+----+---------------+
| id | note          |
+----+---------------+
| 1  | before insert |
| 2  | before update |
| 3  | after update  |
+----+---------------+
3 rows in set (0.00 sec)

--插入新记录的情况：
mysql> delete from tri_demo;
Query OK, 3 rows affected (0.00 sec)

mysql> select film_id,title from film where film_id = 1002;
Empty set (0.00 sec)

mysql> INSERT INTO film VALUES
    -> (1002,'Only test',
    -> 'Only test',2006,1,NULL,6,'0.99',86,'20.99','PG',
    -> 'Deleted Scenes,Behind the Scenes','2006-02-15 05:03:42')
    -> ON DUPLICATE KEY
    -> UPDATE title='update record';

Query OK, 1 row affected (0.05 sec)

mysql>
mysql> select * from tri_demo;
+----+---------------+
| id | note          |
+----+---------------+
| 4  | before insert |
| 5  | after insert  |
+----+---------------+
2 rows in set (0.00 sec)
```

从上面的例子可以知道，对于有重复记录，需要进行 UPDATE 操作的 INSERT，触发器触发的顺序是 BEFORE INSERT、BEFORE UPDATE、AFTER UPDATE；对于没有重复记录的 INSERT，就是简单的执行 INSERT 操作，触发器触发的顺序是 BEFORE INSERT、AFTER INSERT。对于那些

实际执行 UPDATE 操作的记录，仍然会执行 BEFORE INSERT 触发器的内容，在设计触发器的时候一定要考虑这种情况，避免错误地触发了触发器。

13.2 删除触发器

一次可以删除一个触发程序，如果没有指定 schema_name，默认为当前数据库,具体语法如下：

```
DROP TRIGGER [schema_name.]trigger_name
```

例如，要删除 film 表上的触发器 ins_film，可以使用以下命令：

```
mysql> drop trigger ins_film;
Query OK, 0 rows affected (0.00 sec)
```

13.3 查看触发器

可以通过执行 SHOW TRIGGERS 命令查看触发器的状态、语法等信息，但是因为不能查询指定的触发器，所以每次都返回所有的触发器的信息，使用起来不是很方便，具体语法如下：

```
mysql> show triggers \G
*************************** 1. row ***************************
  Trigger: customer_create_date
    Event: INSERT
    Table: customer
Statement: SET NEW.create_date = NOW()
   Timing: BEFORE
  Created: NULL
 sql_mode:
STRICT_TRANS_TABLES,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,TRADITIONAL,NO_AUTO_CREATE_USER
  Definer: root@localhost
*************************** 2. row ***************************
……
```

另外一个查看方式是查询系统表的 information_schema.triggers 表，这个方式可以查询指定触发器的指定信息，操作起来明显方便很多：

```
mysql> desc triggers;
+----------------------------+--------------+------+-----+---------+-------+
| Field                      | Type         | Null | Key | Default | Extra |
+----------------------------+--------------+------+-----+---------+-------+
| TRIGGER_CATALOG            | varchar(512) | YES  |     |         |       |
| TRIGGER_SCHEMA             | varchar(64)  | NO   |     |         |       |
| TRIGGER_NAME               | varchar(64)  | NO   |     |         |       |
| EVENT_MANIPULATION         | varchar(6)   | NO   |     |         |       |
| EVENT_OBJECT_CATALOG       | varchar(512) | YES  |     |         |       |
| EVENT_OBJECT_SCHEMA        | varchar(64)  | NO   |     |         |       |
```

```
| EVENT_OBJECT_TABLE          | varchar(64)  | NO   |     |         |       |
| ACTION_ORDER                | bigint(4)    | NO   |     | 0       |       |
| ACTION_CONDITION            | longtext     | YES  |     |         |       |
| ACTION_STATEMENT            | longtext     | NO   |     |         |       |
| ACTION_ORIENTATION          | varchar(9)   | NO   |     |         |       |
| ACTION_TIMING               | varchar(6)   | NO   |     |         |       |
| ACTION_REFERENCE_OLD_TABLE  | varchar(64)  | YES  |     |         |       |
| ACTION_REFERENCE_NEW_TABLE  | varchar(64)  | YES  |     |         |       |
| ACTION_REFERENCE_OLD_ROW    | varchar(3)   | NO   |     |         |       |
| ACTION_REFERENCE_NEW_ROW    | varchar(3)   | NO   |     |         |       |
| CREATED                     | datetime     | YES  |     |         |       |
| SQL_MODE                    | longtext     | NO   |     |         |       |
| DEFINER                     | longtext     | NO   |     |         |       |
+-----------------------------+--------------+------+-----+---------+-------+
19 rows in set (0.00 sec)

mysql> select * from triggers where trigger_name = 'ins_film_bef' \G
*************************** 1. row ***************************
           TRIGGER_CATALOG: NULL
            TRIGGER_SCHEMA: sakila
              TRIGGER_NAME: ins_film_bef
        EVENT_MANIPULATION: INSERT
      EVENT_OBJECT_CATALOG: NULL
       EVENT_OBJECT_SCHEMA: sakila
        EVENT_OBJECT_TABLE: film
              ACTION_ORDER: 0
          ACTION_CONDITION: NULL
          ACTION_STATEMENT: BEGIN
    INSERT INTO tri_demo (note) VALUES ('before insert');
END
        ACTION_ORIENTATION: ROW
             ACTION_TIMING: BEFORE
ACTION_REFERENCE_OLD_TABLE: NULL
ACTION_REFERENCE_NEW_TABLE: NULL
  ACTION_REFERENCE_OLD_ROW: OLD
  ACTION_REFERENCE_NEW_ROW: NEW
                   CREATED: NULL
                  SQL_MODE:
                   DEFINER: root@localhost
1 row in set (0.01 sec)
```

13.4 触发器的使用

触发器执行的语句有以下两个限制。

- 触发程序不能调用将数据返回客户端的存储程序，也不能使用采用 CALL 语句的动态 SQL 语句，但是允许存储程序通过参数将数据返回触发程序。也就是存储过程或者函数通过 OUT 或者 INOUT 类型的参数将数据返回触发器是可以的，但是不能调用直接返回数据的过程。
- 不能在触发器中使用以显式或隐式方式开始或结束事务的语句，如 START TRANSACTION、COMMIT 或 ROLLBACK。

MySQL 的触发器是按照 BEFORE 触发器、行操作、AFTER 触发器的顺序执行的，其中任何一步操作发生错误都不会继续执行剩下的操作。如果是对事务表进行的操作，那么会整个作为一个事务被回滚（Rollback），但是如果是对非事务表进行的操作，那么已经更新的记录将无法回滚，这也是设计触发器的时候需要注意的问题。

13.5 小结

本节主要介绍了触发器的定义、修改以及触发器使用的一些注意事项。需要注意的是触发器是行触发的，每次增加、修改或者删除记录都会触发进行处理，编写过于复杂的触发器或者增加过多的触发器对记录的插入、更新、删除操作肯定会有比较严重的影响，因此数据库设计的时候要有所考虑，不要将应用的处理逻辑过多的依赖于触发器来处理。

第14章　事务控制和锁定语句

MySQL 支持对 MyISAM 和 MEMORY 存储引擎的表进行表级锁定，对 BDB 存储引擎的表进行页级锁定，对 InnoDB 存储引擎的表进行行级锁定。默认情况下，表锁和行锁都是自动获得的，不需要额外的命令。但是在有的情况下，用户需要明确地进行锁表或者进行事务的控制，以便确保整个事务的完整性，这样就需要使用事务控制和锁定语句来完成。

有关锁机制、不同存储引擎对锁的处理、死锁等内容，将会在后面的优化篇中进行更详细的介绍，有兴趣的读者可以参见相关的章节。

14.1 LOCK TABLE 和 UNLOCK TABLE

LOCK TABLES 可以锁定用于当前线程的表。如果表被其他线程锁定，则当前线程会等待，直到可以获取所有锁定为止。

UNLOCK TABLES 可以释放当前线程获得的任何锁定。当前线程执行另一个 LOCK TABLES 时，或当与服务器的连接被关闭时，所有由当前线程锁定的表被隐含地解锁，具体语法如下：

```
LOCK TABLES
    tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
    [, tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...
```

```
UNLOCK TABLES
```

如表 14-1 所示是一个获得表锁和释放表锁的简单例子，演示的是 film_text 表获得 read 锁的情况，其他 session 更新该表记录会等待锁，film_text 表释放锁以后，其他 session 可以进行更新操作。其中 session1 和 session2 表示两个同时打开的 session，表格中的每一行表示同一时刻两个 session 的运行状况，后面的例子也都是同样格式，不再重复说明。

表 14-1　　一个获得表锁和释放表锁的简单例子

session_1	session_2
获得表 film_text 的 READ 锁定 mysql> lock table film_text read; Query OK, 0 rows affected (0.00 sec)	
当前 session 可以查询该表记录 mysql> select film_id,title from film_text where film_id = 1001; +---------+------------------+ | film_id | title | +---------+------------------+ | 1001 | ACADEMY DINOSAUR | +---------+------------------+ 1 row in set (0.00 sec)	其他 session 也可以查询该表的记录 mysql> select film_id,title from film_text where film_id = 1001; +---------+------------------+ | film_id | title | +---------+------------------+ | 1001 | ACADEMY DINOSAUR | +---------+------------------+ 1 row in set (0.00 sec)
	其他 session 更新锁定表会等待获得锁： mysql> update film_text set title = 'Test' where film_id = 1001; 等待
释放锁 mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待
	Session 获得锁，更新操作完成： mysql> update film_text set title = 'Test' where film_id = 1001; Query OK, 1 row affected (1 min 0.71 sec) Rows matched: 1 Changed: 1 Warnings: 0

有关表锁的使用，将在后面的章节中进行介绍，读者可以参见“20.2 Myisam 表锁”一节以获得更详细的信息。

14.2 事务控制

MySQL 通过 SET AUTOCOMMIT、START TRANSACTION、COMMIT 和 ROLLBACK 等语句支持本地事务，具体语法如下。

```
START TRANSACTION | BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET AUTOCOMMIT = {0 | 1}
```

默认情况下，MySQL 是自动提交（Autocommit）的，如果需要通过明确的 Commit 和 Rollback 来提交和回滚事务，那么需要通过明确的事务控制命令来开始事务，这是和 Oracle 的事务管理明显不同的地方。如果应用是从 Oracle 数据库迁移到 MySQL 数据库，则需要确保应用中是否对事务进行了明确的管理。

- START TRANSACTION 或 BEGIN 语句可以开始一项新的事务。
- COMMIT 和 ROLLBACK 用来提交或者回滚事务。
- CHAIN 和 RELEASE 子句分别用来定义在事务提交或者回滚之后的操作，CHAIN 会立即启动一个新事物，并且和刚才的事务具有相同的隔离级别，RELEASE 则会断开和客户端的连接。
- SET AUTOCOMMIT 可以修改当前连接的提交方式，如果设置了 SET AUTOCOMMIT=0，则设置之后的所有事务都需要通过明确的命令进行提交或者回滚。

如果只是对某些语句需要进行事务控制，则使用 START TRANSACTION 语句开始一个事务比较方便，这样事务结束之后可以自动回到自动提交的方式，如果希望所有的事务都不是自动提交的，那么通过修改 AUTOCOMMIT 来控制事务比较方便，这样不用在每个事务开始的时候再执行 START TRANSACTION 语句。

如表 14-2 所示的例子演示了使用 START TRANSACTION 开始的事务在提交后自动回到自动提交的方式；如果在提交的时候使用 COMMIT AND CHAIN，那么会在提交后立即开始一个新的事务。

表 14-2　　START TRANSACTION 和 COMMIT AND CHAIN 的使用例子

session_1	session_2
从表 actor 中查询 actor_id=201 的记录，结果为空： mysql> select * from actor where actor_id = 201; Empty set (0.00 sec)	从表 actor 中查询 actor_id=201 的记录，结果为空： mysql> select * from actor where actor_id = 201; Empty set (0.00 sec)
用 start transaction 命令启动一个事务，往表 actor 中插入一条记录，没有 commit： mysql> start transaction; Query OK, 0 rows affected (0.00 sec) mysql> insert into actor (actor_id,first_name,last_name) values(201,'Lisa','Tom'); Query OK, 1 row affected (0.00 sec)	
	查询表 actor，结果仍然为空： mysql> select * from actor where actor_id = 201; Empty set (0.00 sec)
执行提交： mysql> commit; Query OK, 0 rows affected (0.04 sec)	

	再次查询表 actor，可以查询到结果： mysql> select actor_id,last_name from actor where actor_id in (201,202); +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 201 | Tom | +----------+-----------+ 1 row in set (0.00 sec)
这个事务是按照自动提交执行的： mysql> insert into actor (actor_id,first_name,last_name) values(202,'Lisa','Lan'); Query OK, 1 row affected (0.04 sec)	
	可以从 actor 表中查询到 session1 刚刚插入的数据。 mysql> select actor_id,last_name from actor where actor_id in (201,202); +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 201 | Tom | | 202 | Lan | +----------+-----------+ 2 rows in set (0.00 sec)

重新用 start transaction 启动一个事务：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
 	
往表 actor 中插入一条记录：
mysql> insert into actor
(actor_id,first_name,last_name)
values(203,'Lisa','TT');
Query OK, 1 row affected (0.00 sec)
用 commit and chain 命令提交：
mysql> commit and chain;
Query OK, 0 rows affected (0.03 sec)
 	
此时自动开始一个新的事务：
mysql> insert into actor
(actor_id,first_name,last_name)
values(204,'Lisa','Mou');
Query OK, 1 row affected (0.00 sec)	

	session1 刚插入的记录无法看到： mysql> select actor_id,last_name from actor where first_name = 'Lisa'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 178 | MONROE T | | 201 | Tom | | 202 | Lan | | 203 | TT | +----------+-----------+ 4 rows in set (0.00 sec)
用 commit 命令提交： mysql> commit; Query OK, 0 rows affected (0.06 sec)	
	session1 插入的新记录可以看到： mysql> select actor_id,last_name from actor where first_name = 'Lisa'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 178 | MONROE T | | 201 | Tom | | 202 | Lan | | 203 | TT | | 204 | Mou | +----------+-----------+ 5 rows in set (0.00 sec)

如果在锁表期间，用 start transaction 命令开始一个新事务，会造成一个隐含的 unlock tables 被执行，如表 14-3 所示。

表 14-3　　start transaction 导致的 unlock tables

session_1	session_2
从表 actor 中查询 actor_id=201 的记录，结果为空： mysql> select * from actor where actor_id = 201; Empty set (0.00 sec)	从表 actor 中查询 actor_id=201 的记录，结果为空： mysql> select * from actor where actor_id = 201; Empty set (0.00 sec)
对表 actor 加写锁： mysql> lock table actor write; Query OK, 0 rows affected (0.00 sec)	

	对表 actor 的读操作被阻塞： mysql> select actor_id,last_name from actor where actor_id = 201; 等待
插入一条记录 mysql> insert into actor (actor_id,first_name,last_name) values(201,'Lisa','Tom'); Query OK, 1 row affected (0.04 sec)	等待
回滚刚才的记录： mysql> rollback; Query OK, 0 rows affected (0.00 sec)	等待
用 start transaction 命令重新开始一个事务： mysql> start transaction; Query OK, 0 rows affected (0.00 sec)	等待
	session1 开始一个事务时，表锁被释放，可以查询： mysql> select actor_id,last_name from actor where actor_id = 201; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 201 | Tom | +----------+-----------+ 1 row in set (17.78 sec) 对 lock 方式加的表锁，不能通过 rollback 进行回滚。

因此，在同一个事务中，最好不使用不同存储引擎的表，否则 ROLLBACK 时需要对非事务类型的表进行特别的处理，因为 COMMIT、ROLLBACK 只能对事务类型的表进行提交和回滚。

通常情况下，只对提交的事务记录到二进制的日志中，但是如果一个事务中包含非事务类型的表，那么回滚操作也会被记录到二进制日志中，以确保非事务类型表的更新可以被复制到从（Slave）数据库中。

和 Oracle 的事务管理相同，所有的 DDL 语句是不能回滚的，并且部分的 DDL 语句会造成隐式的提交。

在事务中可以通过定义 SAVEPOINT，指定回滚事务的一个部分，但是不能指定提交事务的一个部分。对于复杂的应用，可以定义多个不同的 SAVEPOINT，满足不同的条件时，回滚不同的 SAVEPOINT。需要注意的是，如果定义了相同名字的 SAVEPOINT，则后面定义的 SAVEPOINT 会覆盖之前的定义。对于不再需要使用的 SAVEPOINT，可以通过 RELEASE SAVEPOINT 命令删除 SAVEPOINT，删除后的 SAVEPOINT，不能再执行 ROLLBACK TO SAVEPOINT 命令。

如表 14-4 所示的例子就是模拟回滚事务的一个部分，通过定义 SAVEPOINT 来指定需要回滚的事务的位置。

表 14-4　　模拟回滚事务

session_1	session_2
从表 actor 中查询 first_name=’ Simon’ 的记录，结果为空： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)	从表 actor 中查询 first_name=’ Simon’ 的记录，结果为空： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)
启动一个事务，往表 actor 中插入一条记录： mysql> start transaction; Query OK, 0 rows affected (0.02 sec) mysql> insert into actor (actor_id,first_name,last_name) values(301,'Simon','Tom'); Query OK, 1 row affected (0.00 sec)	
可以查询到刚插入的记录： mysql> select actor_id,last_name from actor where first_name = 'Simon'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 301 | Tom | +----------+-----------+ 1 row in set (0.00 sec)	无法从 actor 表中查到 session1 刚插入的记录： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)
定义 savepoint，名称为 test： mysql> savepoint test; Query OK, 0 rows affected (0.00 sec) 继续插入一条记录： mysql> insert into actor (actor_id,first_name,last_name) values(302,'Simon','Cof'); Query OK, 1 row affected (0.00 sec)	
可以查询到两条记录： mysql> select actor_id,last_name from actor where first_name = 'Simon'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 301 | Tom | | 302 | Cof | +----------+-----------+ 2 rows in set (0.00 sec)	仍然无法查询到结果： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)

回滚到刚才定义的 savepoint： mysql> rollback to savepoint test; Query OK, 0 rows affected (0.00 sec)	
只能从表 actor 中查询到第一条记录，因为第二条已经被回滚： mysql> select actor_id,last_name from actor where first_name = 'Simon'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 301 | Tom | +----------+-----------+ 1 row in set (0.00 sec)	仍然无法查询到结果： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)
用 commit 命令提交： mysql> commit; Query OK, 0 rows affected (0.05 sec)	
只能从 actor 表中查询到第一条记录： mysql> select actor_id,last_name from actor where first_name = 'Simon'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 301 | Tom | +----------+-----------+ 1 row in set (0.00 sec)	只能从 actor 表中查询到 session1 插入的第一条记录： mysql> select actor_id,last_name from actor where first_name = 'Simon'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 301 | Tom | +----------+-----------+ 1 row in set (0.00 sec)

14.3 分布式事务的使用

MySQL 从 5.0.3 开始支持分布式事务，当前分布式事务只支持 InnoDB 存储引擎。一个分布式事务会涉及多个行动，这些行动本身是事务性的。所有行动都必须一起成功完成，或者一起被回滚。

14.3.1 分布式事务的原理

在 MySQL 中，使用分布式事务的应用程序涉及一个或多个资源管理器和一个事务管理器。

- 资源管理器（RM）用于提供通向事务资源的途径。数据库服务器是一种资源管理器。该管理器必须可以提交或回滚由 RM 管理的事务。例如，多台 MySQL 数据库作为多台资源管理器或者几台 Mysql 服务器和几台 Oracle 服务器作为资源管理器。
- 事务管理器（TM）用于协调作为一个分布式事务一部分的事务。TM 与管理每个事务的 RMs 进行通讯。一个分布式事务中各个单个事务均是分布式事务的“分支事务”。分布式事务和各分支通过一种命名方法进行标识。

MySQL 执行 XA MySQL 时，MySQL 服务器相当于一个用于管理分布式事务中的 XA 事务的资源管理器。与 MySQL 服务器连接的客户端相当于事务管理器。

要执行一个分布式事务，必须知道这个分布式事务涉及到了哪些资源管理器，并且把每个资源管理器的事务执行到事务可以被提交或回滚时。根据每个资源管理器报告的有关执行情况的内容，这些分支事务必须作为一个原子性操作全部提交或回滚。要管理一个分布式事务，必须要考虑任何组件或连接网络可能会故障。

用于执行分布式事务的过程使用两阶段提交，发生时间在由分布式事务的各个分支需要进行的行动已经被执行之后。

- 在第一阶段，所有的分支被预备好。即它们被 TM 告知要准备提交。通常，这意味着用于管理分支的每个 RM 会记录对于被稳定保存的分支的行动。分支指示是否它们可以这么做。这些结果被用于第二阶段。
- 在第二阶段，TM 告知 RMs 是否要提交或回滚。如果在预备分支时，所有的分支指示它们将能够提交，则所有的分支被告知要提交。如果在预备时，有任何分支指示它将不能提交，则所有分支被告知回滚。

在有些情况下，一个分布式事务可能会使用一阶段提交。例如，当一个事务管理器发现，一个分布式事务只由一个事务资源组成（即单一分支），则该资源可以被告知同时进行预备和提交。

14.3.2 分布式事务的语法

分布式事务（XA 事务）的 SQL 语法主要包括：

```
XA {START|BEGIN} xid [JOIN|RESUME]
```

XA START xid 用于启动一个带给定 xid 值的 XA 事务。每个 XA 事务必须有一个唯一的 xid 值，因此该值当前不能被其他的 XA 事务使用。

xid 是一个 XA 事务标识符，用来唯一标识一个分布式事务。xid 值由客户端提供，或由 MySQL 服务器生成。xid 值包含 1～3 个部分：

```
xid: gtrid [, bqual [, formatID ]]
```

- gtrid 是一个分布式事务标识符，相同的分布式事务应该使用相同的 gtrid，这样可以明确知道 xa 事务属于哪个分布式事务。
- bqual 是一个分支限定符，默认值是空串。对于一个分布式事务中的每个分支事务，bqual 值必须是唯一的。
- formatID 是一个数字，用于标识由 gtrid 和 bqual 值使用的格式，默认值是 1。

下面其他 XA 语法中用到的 xid 值，都必须和 START 操作使用的 xid 值相同，也就是表示对这个启动的 XA 事务进行操作。

```
XA END xid [SUSPEND [FOR MIGRATE]]


XA PREPARE xid
```

使事务进入 PREPARE 状态，也就是两阶段提交的第一个提交阶段。

```
XA COMMIT xid [ONE PHASE]
XA ROLLBACK xid
```

这两个命令用来提交或者回滚具体的分支事务。也就是两阶段提交的第二个提交阶段，分支事务被实际的提交或者回滚。

```
XA RECOVER
```

XA RECOVER 返回当前数据库中处于 PREPARE 状态的分支事务的详细信息。

分布式的关键在于如何确保分布式事务的完整性，以及在某个分支出现问题时的故障解决。XA 的相关命令就是提供给应用如何在多个独立的数据库之间进行分布式事务的管理，包括启动一个分支事务、使事务进入准备阶段以及事务的实际提交回滚操作等，如表 14-5 所示的例子演示了一个简单的分布式事务的执行，事务的内容是在 DB1 中插入一条记录，同时在 DB2 中更新一条记录，两个操作作为同一事务提交或者回滚。

表 14-5　　分布式事务例子

session_1 in DB1	session_2 in DB2
在数据库 DB1 中启动一个分布式事务的一个分支事务，xid 的 gtrid 为“test”，bqual 为“db1”： mysql> xa start 'test','db1'; Query OK, 0 rows affected (0.00 sec) 分支事务 1 在表 actor 中插入一条记录： mysql> insert into actor (actor_id,first_name,last_name) values(301,'Simon','Tom'); Query OK, 1 row affected (0.00 sec) 对分支事务 1 进行第一阶段提交，进入 prepare 状态： mysql> xa end 'test','db1'; Query OK, 0 rows affected (0.00 sec) mysql> xa prepare 'test','db1'; Query OK, 0 rows affected (0.02 sec)	在数据库 DB2 中启动分布式事务“test”的另外一个分支事务，xid 的 gtrid 为“test”，bqual 为“db2”： mysql> xa start 'test','db2'; Query OK, 0 rows affected (0.00 sec) 分支事务 2 在表 film_actor 中更新了 23 条记录： mysql> update film_actor set last_update=now() where actor_id = 178; Query OK, 23 rows affected (0.04 sec) Rows matched: 23 Changed: 23 Warnings: 0 对分支事务 2 进行第一阶段提交，进入 prepare 状态： mysql> xa end 'test','db2'; Query OK, 0 rows affected (0.00 sec) mysql> xa prepare 'test','db2'; Query OK, 0 rows affected (0.02 sec)
用 xa recover 命令查看当前分支事务状态： mysql> xa recover \G *************************** 1. row *************************** formatID: 1 gtrid_length: 4 bqual_length: 3 data: testdb1 1 row in set (0.00 sec)	用 xa recover 命令查看当前分支事务状态： mysql> xa recover \G *************************** 1. row *************************** formatID: 1 gtrid_length: 4 bqual_length: 3 data: testdb2 1 row in set (0.00 sec)
两个事务都进入准备提交阶段，如果之前遇到任何错误，都应该回滚所有的分支，以确保分布式事务的正确。	
提交分支事务 1： mysql> xa commit 'test','db1'; Query OK, 0 rows affected (0.03 sec)	提交分支事务 2： mysql> xa commit 'test','db2'; Query OK, 0 rows affected (0.03 sec)

两个事务都到达准备提交阶段后，一旦开始进行提交操作，就需要确保全部的分支都提交成功。	

14.3.3 存在的问题

虽然 MySQL 支持分布式事务，但是在测试过程中，还是发现存在一些问题。

如果分支事务在达到 prepare 状态时，数据库异常重新启动，服务器重新启动以后，可以继续对分支事务进行提交或者回滚得操作，但是提交的事务没有写 binlog，存在一定的隐患，可能导致使用 binlog 恢复丢失部分数据。如果存在复制的数据库，则有可能导致主从数据库的数据不一致。以下演示了这个过程：

（1）从表 actor 中查询 first_name = 'Simon'的记录，有一条。

```
mysql> select actor_id,last_name from actor where first_name = 'Simon';
+----------+-----------+
| actor_id | last_name |
+----------+-----------+
| 301      | Tom       |
+----------+-----------+
1 row in set (0.00 sec)
```

（2）启动分布式事务“test”，删除刚才查询的记录。

```
mysql> xa start 'test';
Query OK, 0 rows affected (0.00 sec)

mysql> delete from actor where actor_id = 301;
Query OK, 1 row affected (0.00 sec)

mysql> select actor_id,last_name from actor where first_name = 'Simon';
Empty set (0.00 sec)
```

（3）完成第一阶段提交，进入 prepare 状态。

```
mysql> xa end 'test';
Query OK, 0 rows affected (0.00 sec)

mysql>  xa prepare 'test';
Query OK, 0 rows affected (0.03 sec)
```

（4）此时，数据库异常终止，查询出错。

```
mysql> select actor_id,last_name from actor where first_name = 'Simon';
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/mnt/db/mysqld.sock'
(2)
ERROR:
```

```
Can't connect to the server
```

（5）启动数据库后，分支事务依然存在。

```
mysql> xa recover \G
*************************** 1. row ***************************
    formatID: 1
gtrid_length: 4
bqual_length: 0
        data: test
1 row in set (0.00 sec)
```

（6）表中记录并没有被删除。

```
mysql> select actor_id,last_name from actor where first_name = 'Simon';
+----------+-----------+
| actor_id | last_name |
+----------+-----------+
| 301      | Tom       |
+----------+-----------+
1 row in set (0.00 sec)
```

（7）可以进行提交或者回滚。

```
mysql>  xa commit 'test';
Query OK, 0 rows affected (0.02 sec)

mysql> select actor_id,last_name from actor where first_name = 'Simon';
Empty set (0.00 sec)
```

在上面测试中，如果重新启动MySQL数据库以后，可以在MySQL的数据库日志中看到分布式事务的处理情况，数据库启动的时候发现有一个prepare状态的事务，提示需要进行处理：

```
InnoDB: Transaction 0 117471044 was in the XA prepared state.
InnoDB: 1 transaction(s) which must be rolled back or cleaned up
InnoDB: in total 0 row operations to undo
InnoDB: Trx id counter is 0 117471488
070710 16:55:41  InnoDB: Started; log sequence number 29 2758352865
070710 16:55:41  InnoDB: Starting recovery for XA transactions...
070710 16:55:41  InnoDB: Transaction 0 117471044 in prepared state after recovery
070710 16:55:41  InnoDB: Transaction contains changes to 1 rows
070710 16:55:41  InnoDB: 1 transactions in prepared state after recovery
070710 16:55:41 [Note] Found 1 prepared transaction(s) in InnoDB
070710 16:55:41 [Warning] Found 1 prepared XA transactions
```

使用 mysqlbinlog 查看 binlog，可以确认最后提交的这个分支事务并没有记录到 binlog 中，因为复制和灾难恢复都是依赖于 binlog 的，所以 binlog 的缺失会导致复制环境的不同步，

以及使用 binlog 恢复丢失部分数据。

如果分支事务的客户端连接异常中止，那么数据库会自动回滚未完成的分支事务，如果此时分支事务已经执行到 prepare 状态，那么这个分布式事务的其他分支可能已经成功提交，如果这个分支回滚，可能导致分布式事务的不完整，丢失部分分支事务的内容。

表 14-6　　客户端连接中止导致分布式事务失败例子

session_1	session_2
从表 actor 中查询 first_name='Simon' 的记录，结果为空： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)	从表 actor 中查询 first_name='Simon' 的记录，结果为空： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)
启动分布式事务 test： mysql> xa start 'test'; Query OK, 0 rows affected (0.00 sec) 往 actor 表中插入一条记录： mysql> insert into actor (actor_id,first_name,last_name) values(301,'Simon','Tom'); Query OK, 1 row affected (0.00 sec) 事务结束： mysql> xa end 'test'; Query OK, 0 rows affected (0.00 sec)	
查询刚插入的记录，可以显示结果： mysql> select actor_id,last_name from actor where first_name = 'Simon'; +----------+-----------+ | actor_id | last_name | +----------+-----------+ | 301 | Tom | +----------+-----------+ 1 row in set (0.00 sec)	查询刚插入的记录，显示结果为空： mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)
完成第一阶段提交，进入 prepare 状态。 mysql> xa prepare 'test'; Query OK, 0 rows affected (0.02 sec)	
	查询分布式事务“test”的状态： mysql> xa recover \G *************************** 1. row *************************** formatID: 1 gtrid_length: 4 bqual_length: 3

	data: test 1 row in set (0.00 sec)
session_1 异常中止	
session_1 被回滚。	session1 异常中止后，分布式事务被回滚，session2 中无法查询到 session1 插入的记录，如果此时 session2 存在分支事务并且被成功提交，则会导致分布式事务的不完整。 mysql> select * from actor where first_name = 'Simon'; Empty set (0.00 sec)

如果分支事务在执行到 prepare 状态时，数据库异常，且不能再正常启动，需要使用备份和 binlog 来恢复数据，那么那些在 prepare 状态的分支事务因为并没有记录到 binlog，所以不能通过 binlog 进行恢复，在数据库恢复后，将丢失这部分的数据。

总之，MySQL 的分布式事务还存在比较严重的缺陷，在数据库或者应用异常的情况下，可能会导致分布式事务的不完整。如果应用对于数据的完整性要求不是很高，则可以考虑使用。如果应用对事务的完整性有比较高的要求，那么对于当前的版本，则不推荐使用分布式事务。

14.4 小结

事务控制和锁定是 MySQL 的重要特点之一，本章介绍了 MySQL 提供的事务控制和锁定语法，并对分布式事务进行了简单的介绍。MySQL 中锁的管理涉及的内容很广泛，在后面的优化篇中我们将会对锁机制、死锁和应用中需要注意的其他问题进行了更深入的讨论。

第15章 SQL 中的安全问题

在日常开发过程中，程序员一般只关心 SQL 是否能实现预期的功能，而对于 SQL 的安全问题一般都不太重视。实际上，如果 SQL 语句写作不当，将会给应用系统造成很大的安全隐患，其中最重要的隐患就是 SQL 注入。本章以 MySQL 为例，将会对 SQL 注入以及相应的防范措施进行详细的介绍。

15.1 SQL 注入简介

结构化查询语言（SQL）是一种用来和数据库交互的文本语言。SQL Injection 就是利用某些数据库的外部接口将用户数据插入到实际的数据库操作语言（SQL）当中，从而达到入侵数据库乃至操作系统的目的。它的产生主要是由于程序对用户输入的数据没有进行严格的过滤，导致非法数据库查询语句的执行。

SQL 注入（SQL Injection）攻击具有很大的危害，攻击者可以利用它读取、修改或者删除数据库内的数据，获取数据库中的用户名和密码等敏感信息，甚至可以获得数据库管理

员的权限，而且，SQL Injection 也很难防范。网站管理员无法通过安装系统补丁或者进行简单的安全配置进行自我保护，一般的防火墙也无法拦截 SQL Injection 攻击。

下面的用户登录验证程序就是 SQL 注入的一个例子（以 PHP 程序举例）。

（1）创建用户表 user：

```
CREATE TABLE user (
    userid int(11) NOT NULL auto_increment,
    username varchar(20) NOT NULL default '',
    password varchar(20) NOT NULL default '',
    PRIMARY KEY (userid)
    ) TYPE=MyISAM AUTO_INCREMENT=3 ;
```

（2）给用户表 user 添加一条用户记录：

```
INSERT INTO `user` VALUES (1, 'angel', 'mypass');
```

（3）验证用户 root 登录 localhost 服务器：

```
<?php
    $servername = "localhost";
    $dbusername = "root";
    $dbpassword = "";
    $dbname = "injection";
    mysql_connect($servername,$dbusername,$dbpassword) or die ("数据库连接失败");
    $sql = "SELECT * FROM user WHERE username='$username' AND password='$password'";
    $result = mysql_db_query($dbname, $sql);
    $userinfo = mysql_fetch_array($result);
    if (empty($userinfo))
    {
    echo "登录失败";
    } else {
    echo "登录成功";
    }
    echo "<p>SQL Query:$sql<p>";
    ?>
```

（4）然后提交如下 URL：

```
http://127.0.0.1/injection/user.php?username=angel' or '1=1
```

结果发现，这个 URL 可以成功登录系统，但是很显然这并不是我们预期的结果。同样也可以利用 SQL 的注释语句实现 SQL 注入，如下面的例子：

```
http://127.0.0.1/injection/user.php?username=angel'/*
http://127.0.0.1/injection/user.php?username=angel'#
```

因为在 SQL 语句中，“/*”或者“#”都可以将后面的语句注释掉。这样上述语句就可以通过这两个注释符中任意一个将后面的语句给注释掉了，结果导致只根据用户名而没有密码的 URL 都成功进行了登录。利用“or”和注释符的不同之处在于，前者是利用逻辑运算，而后者则是根据 MySQL 的特性，这个比逻辑运算简单得多了。虽然这两种情况实现的原理不同，但是达到了同样的 SQL 注入效果，都是我们应该关注的。

15.2 应用开发中可以采取的应对措施

对于上面提到的 SQL 注入隐患，后果可想而知是很严重的，轻则获得数据信息，重则可以将数据进行非法更改。那么对这种情况有没有防范措施呢？答案是肯定的，本节将介绍一些常用的防范方法。

15.2.1 PrepareStatement+Bind-variable

对 Java、JSP 开发的应用,可以使用 PrepareStatement+Bind-variable 来防止 SQL 注入，另外从 PHP 5 开始，也在扩展的 MySQLI 中支持 PrepareStatement，所以在使用这类语言作数据库开发时，强烈建议使用 PrepareStatement+Bind-variable 来实现，而尽量不要使用拼接的 SQL，下面以 Java 为例说明一下实现方法：

```
…
String sql = "select * from users u where u.id = ? and u.password = ?";
preparedstatement ps = connection.preparestatement(sql);
ps.setint(1,id);
ps.setstring(2,pwd);
```

```
resultset rs = ps.executequery();
…
```

15.2.2 使用应用程序提供的转换函数

很多应用程序接口都提供了对特殊字符进行转换的函数，恰当地使用这些函数，可以防止应用程序用户输入使应用程序生成不期望的语句。

- MySQL C API：使用 mysql_real_escape_string() API 调用。
- MySQL++：使用 escape 和 quote 修饰符。
- PHP：使用 mysql_real_escape_string()函数（适用于 PHP 4.3.0 版本）。从 PHP 5 开始,可以使用扩展的 MySQLI，这是对 MySQL 新特性的一个扩展支持，其中的一个优点就是支持 PrepareStatement。
- Perl DBI：使用 placeholders 或者 quote()方法。
- Ruby DBI：使用 placeholders 或者 quote()方法。

15.2.3 自己定义函数进行校验

如果现有的转换函数仍然不能满足要求，则需要自己编写函数进行输入校验。输入验证是一个很复杂的问题。输入验证的途径可以分为以下几种：

- 整理数据使之变得有效；
- 拒绝已知的非法输入；
- 只接受已知的合法输入。

所以如果想要获得最好的安全状态，目前最好的解决办法就是对用户提交或者可能改变的数据进行简单分类，分别应用正则表达式来对用户提供的输入数据进行严格的检测和验证。

下面采用正则表达式的方法提供一个验证函数，以供大家参考。

已知非法符号有：

“”、“;”、“=”、“(”、“)”、“/*”、“*/”、“%”、“+”、“”、“>”、“<”、“--”、“[”、“]”；

其实只需要过滤非法的符号组合就可以阻止已知形式的攻击，并且如果发现更新的攻击符号组合，也可以将这些符号组合增添进来，继续防范新的攻击。特别是空格符号和与其产生相同作用的分隔关键字的符号，例如“/**/”，如果能成功过滤这种符号，那么有很多注入攻击将不能发生，并且同时也要过滤它们的十六进制表示“%XX”。

由此可以构造如下正则表达式：

```
(|\'|(\%27)|\;|(\%3b)|\=|(\%3d)|\(|(\%28)|\)|(\%29)|(\/*)|(\%2f%2a)|(\*/)|(\%2a%2f)|\+|(\%2b)|\<|(\%3c)|\>|(\%3e)|\(--))|\[|\%5b|\]|\%5d)
```

根据上述的正则表达式，可以提供一个函数（以 PHP 举例），可以防范大多数的 SQL 注入，具体函数如下：

```
function SafeRequest ($ParaName, $ParaType)
{
    /* ---传入参数--- */
    /* ParaName:参数名称-字符型 */
    /* ParaType: 参数类型-数字型（1 表示参数是数字或字符，0 表示参数为其他）*/

    if ($ParaType == 1)
    {
        $re = "/[^\w+$]/";
    }
    else
    {
        $re = "/(|\'|(\%27)|\;|(\%3b)|\=|(\%3d)|\(|(\%28)|\)|(\%29)|(\/*)|(\%2f%2a)|(\*/)|(\%2a%2f)|\+|(\%2b)|\<|(\%3c)|\>|(\%3e)|\(--))|\[|\%5b|\]|\%5d)/";
    }

    if (preg_match($re, $ParaName) > 0)
    {
        echo("参数不符合要求，请重新输入!");
        return 0;
    }
    else
    {
        return 1;
    }
}
```

15.3 小结

本章主要从 SQL 注入的角度讨论了 SQL 的安全问题，阐述了 SQL 注入的原理以及防范措施，最后通过一个 PHP 函数例子给出了类似问题解决方法的参考。

本章的内容不仅仅适用于 MySQL 数据库，一些原理以及解决方案同样适用于其他数据库系

统，因为 SQL 注入问题是一个数据库应用普遍存在的安全问题。

第16章　SQL Mode 及相关问题

与其他数据库不同，MySQL 可以运行不同的 SQL Mode（SQL 模式）下。SQL Mode 定义了 MySQL 应支持的 SQL 语法、数据校验等，这样可以更容易地在不同的环境中使用 MySQL。本章将详细介绍常用的 SQL Mode 及其在实际中的应用。

16.1 MySQL SQL Mode 简介

在 MySQL 中，SQL Mode 常用来解决下面几类问题。

- 通过设置 SQL Mode，可以完成不同严格程度的数据校验，有效地保障数据准确性。
- 通过设置 SQL Mode 为 ANSI 模式，来保证大多数 SQL 符合标准的 SQL 语法,这样应用在不同数据库之间进行迁移时，则不需要对业务 SQL 进行较大的修改。
- 在不同数据库之间进行数据迁移之前，通过设置 SQL Mode 可以使 MySQL 上的数据更方便地迁移到目标数据库中。

下面通过一个简单的实例，让大家了解如何使用 SQL Mode 实现数据校验。
在 MySQL 5.0 上，查询默认的 SQL Mode（sql_mode 参数）为：REAL_AS_FLOAT、PIPES_AS_CONCAT、ANSI_QUOTES、GNORE_SPACE 和 ANSI。在这种模式下允许插入超过字段长度的值，只是在插入后，MySQL 会返回一个 warning。通过修改 sql_mode 为 STRICT_TRANS_TABLES（严格模式）实现了数据的严格校验，使错误数据不能插入表中，从而保证了数据的准确性，具体实现如下。
（1）查看默认 SQL Mode 的命令如下：

```
mysql> select @@sql_mode;
+-------------------------------------------------------------+
| @@sql_mode                                                  |
+-------------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI |
+-------------------------------------------------------------+
1 row in set (0.00 sec)
```

（2）查看测试表 t 的表结构的命令如下：

```
mysql> desc t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| name  | varchar(20) | YES  |     | NULL    |       |
| email | varchar(40) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

```
2 rows in set (0.00 sec)
```

（3）在表 t 中插入一条记录，其中 name 故意超出了实际的定义值 varchar（20）：

```
mysql> insert into t values('12340000000000000000999999','beijing@126.com');
Query OK, 1 row affected, 1 warning (0.00 sec)
```

（4）可以发现，记录可以插入，但是显示了一个 warning，查看 warning 内容：

```
mysql>  show warnings;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1265 | Data truncated for column 'name' at row 1 |
+---------+------+-----------------------------------------+
1 row in set (0.00 sec)
```

（5）warning 提示对插入的 name 值进行了截断，从表 t 中查看实际插入值：

```
mysql> select * from t;
+---------------------+-----------------+
| name                | email           |
+---------------------+-----------------+
| 12340000000000000000 | beijing@126.com |
+---------------------+-----------------+
1 row in set (0.00 sec)
```

果然，记录虽然插入进去，但是只截取了前 20 位字符。

（6）接下来设置 SQL Mode 为 STRICT_TRANS_TABLES（严格模式）：

```
mysql> set session sql_mode='STRICT_TRANS_TABLES';
Query OK, 0 rows affected (0.01 sec)

mysql> select @@sql_mode;
+--------------------+
| @@sql_mode         |
+--------------------+
| STRICT_TRANS_TABLES |
+--------------------+
1 row in set (0.01 sec)
```

（7）再次尝试插入上面的测试记录：

```
mysql> insert into t values('12340000000000000000999999','beijing@126.com');
ERROR 1406 (22001): Data too long for column 'name' at row 1
```

结果发现，这次记录没有插入成功，给出了一个 ERROR，而不仅仅是 warning。

上面的例子中，给出了 sql_mode 的一种修改方法，即 SET [SESSION|GLOBAL] sql_mode='modes'，其中 SESSION 选项表示只在本次连接中生效；而 GLOBAL 选项表示在本

次连接中并不生效，而对于新的连接则生效，这种方法在 MySQL 4.1 开始有效。另外，也可以通过使用“--sql-mode="modes"”选项，在 MySQL 启动时设置 sql_mode。

下面介绍一下 SQL Mode 的常见功能。

（1）效验日期数据合法性，这是 SQL Mode 的一项常见功能。

在下面的例子中，观察一下非法日期“2007-04-31”（因为 4 月份没有 31 日）在不同 SQL Mode 下能否正确插入。

```
mysql> set session sql_mode='ANSI';
Query OK, 0 rows affected (0.00 sec)

mysql> create table t (d datetime);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values('2007-04-31');
Query OK, 1 row affected, 1 warning (0.00 sec)
mysql> select * from t;
+---------------------+
| d                   |
+---------------------+
| 0000-00-00 00:00:00 |
+---------------------+
1 row in set (0.00 sec)

mysql> set session sql_mode='TRADITIONAL';
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t values('2007-04-31');
ERROR 1292 (22007): Incorrect datetime value: '2007-04-31' for column 'd' at row 1
```

很显然，在 ANSI 模式下，非法日期可以插入，但是插入值却变为“0000-00-00 00:00:00”，并且系统给出了 warning；而在 TRADITIONAL 模式下，在直接提示日期非法，拒绝插入。

（2）在 INSERT 或 UPDATE 过程中，如果 SQL MODE 处于 TRADITIONAL 模式，运行 MOD(X，0)会产生错误，因为 TRADITIONAL 也属于严格模式，在非严格模式下 MOD(X，0)返回的结果是 NULL，所以在含有 MOD 的运算中要根据实际情况设定好 sql_mode。

下面的实例展示了不同 sql_mode 下，MOD(X，0)返回的结果。

```
mysql> set sql_mode='ANSI' ;
Query OK, 0 rows affected (0.00 sec)

mysql> create table t (i int);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into t values(9%0);
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
```

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

mysql> set session  sql_mode='TRADITIONAL';
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t values(9%0);
ERROR 1365 (22012): Division by 0
```

（3）启用 NO_BACKSLASH_ESCAPES 模式，使反斜线成为普通字符。在导入数据时，如果数据中含有反斜线字符，启用 NO_BACKSLASH_ESCAPES 模式保证数据的正确性，是个不错的选择。

以下实例说明了启用 NO_BACKSLASH_ESCAPES 模式前后对反斜线“\”插入的变化。

```
mysql> set sql_mode='ansi';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@sql_mode;
+-------------------------------------------------------------+
| @@sql_mode                                                  |
+-------------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI |
+-------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> create table t (context  varchar(20));
Query OK, 0 rows affected (0.04 sec)

mysql> insert into t  values('\beijing');
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+---------+
| context |
+---------+
|eijing |
+---------+
1 row in set (0.00 sec)

mysql> insert into t  values('\\beijing');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> select * from t;
+----------+
| context  |
+----------+
|eijing  |
| \beijing |
+----------+
2 rows in set (0.00 sec)

mysql>set
sql_mode='REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI,NO_BACKSLASH_ESCAPES';
Query OK, 0 rows affected (0.00 sec)

mysql> select @@sql_mode;
+-------------------------------------------------------------------------------------------+
| @@sql_mode                                                                                |
+-------------------------------------------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI,NO_BACKSLASH_ESCAPES |
+-------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> insert into t  values('\\beijing');
Query OK, 1 row affected (0.00 sec)

mysql> select * from t;
+-----------+
| context   |
+-----------+
|eijing   |
| \beijing  |
| \\beijing |
+-----------+
3 rows in set (0.00 sec)
```

通过上面的实例可以看到，当在 ANSI 模式中增加了 NO_BACKSLASH_ESCAPES 模式后，反斜线变为了普通字符。如果导入的数据存在反斜线，可以设置此模式，保证导入数据的正确性。

（4）启用 PIPES_AS_CONCAT 模式。将“||”视为字符串连接操作符，在 Oracle 等数据库中，“||”被视为字符串的连接操作符，所以，在其他数据库中含有“||”操作符的 SQL 在 MySQL 中将无法执行，为了解决这个问题，MySQL 提供了 PIPES_AS_CONCAT 模式。

下面通过实例介绍一下 PIPES_AS_CONCAT 模式的作用。

```
mysql> set sql_mode='ansi';
Query OK, 0 rows affected (0.00 sec)

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

```
| @@sql_mode                                                  |
+-------------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI |
+-------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> select 'beijing'||'2008'  ;
+------------------+
| 'beijing'||'2008' |
+------------------+
| beijing2008      |
+------------------+
1 row in set (0.01 sec)
```

通过上面的实例可以看到，ANSI 模式中包含了 PIPES_AS_CONCAT 模式，所以默认情况下 MySQL 新版本支持将“||”视为字符串连接操作符。

16.2 常用的 SQL Mode

熟悉并了解经常使用的 SQL Mode 会帮助用户更好地使用它。表 16-1 总结出了常用的 SQL Mode 值及其说明。

表 16-1　　MySQL 中的 SQL Mode

sql_mode 值	描述
ANSI	等同于 REAL_AS_FLOAT、PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE 和 ANSI 组合模式，这种模式使语法和行为更符合标准的 SQL
STRICT_TRANS_TABLES	STRICT_TRANS_TABLES 适用于事务表和非事务表，它是严格模式，不允许非法日期，也不允许超过字段长度的值插入字段中，对于插入不正确的值给出错误而不是警告
TRADITIONAL	TRADITIONAL 模式等同于 STRICT_TRANS_TABLES、STRICT_ALL_TABLES、NO_ZERO_IN_DATE、NO_ZERO_DATE、ERROR_FOR_DIVISION_BY_ZERO、TRADITIONAL 和 NO_AUTO_CREATE_USER 组合模式，所以它也是严格模式，对于插入不正确的值是给出错误而不是警告。可以应用在事务表和非事务表，用在事务表时，只要出现错误就会立即回滚

可以发现，表格中第一列 SQL Mode 的值其实都是一些原子模式的组合，类似于角色和权限的关系。这样当实际应用时，只需要设置一个模式组合，就可以设置很多的原子模式，大大方便了用户的工作。

16.3 SQL Mode 在迁移中如何使用

如果 MySQL 与其他异构数据库之间有数据迁移的需求的话，那么 MySQL 中提供的的数据库组合模式则会对数据迁移过程会有所帮助。

从表 16-2 可以看出，MySQL 提供了很多数据库的组合模式名称，例如“ORACLE”、“DB2”等，这些模式组合是由很多小的 sql_mode 组合而成，在异构数据库之间迁移数据时可以尝试使用这些模式来导出适合于目标数据库格式的数据，这样就使得导出数据更容易导入目标

数据库。

表 16-2　　MySQL 中的常用数据库 Mode

组合后的模式名称	组合模式中的各个 sql_mode
DB2	PIPES_AS_CONCAT 、 ANSI_QUOTES 、 IGNORE_SPACE 、 NO_KEY_OPTIONS 、 NO_TABLE_OPTIONS、NO_FIELD_OPTIONS
MAXDB	PIPES_AS_CONCAT 、 ANSI_QUOTES 、 IGNORE_SPACE 、 NO_KEY_OPTIONS 、 NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、 NO_AUTO_CREATE_USER
MSSQL	PIPES_AS_CONCAT 、 ANSI_QUOTES 、 IGNORE_SPACE 、 NO_KEY_OPTIONS 、 NO_TABLE_OPTIONS、 NO_FIELD_OPTIONS
ORACLE	PIPES_AS_CONCAT 、 ANSI_QUOTES 、 IGNORE_SPACE 、 NO_KEY_OPTIONS 、 NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER
POSTGRESQL	PIPES_AS_CONCAT 、 ANSI_QUOTES 、 IGNORE_SPACE 、 NO_KEY_OPTIONS 、 NO_TABLE_OPTIONS、NO_FIELD_OPTIONS

在数据迁移过程中，可以设置 SQL Mode 为 NO_TABLE_OPTIONS 模式，这样将去掉 show create table 中的“engine”关键字，获得通用的建表脚本。

测试实例如下：

```
mysql> show create table emp \G;
*************************** 1. row ***************************
       Table: emp
Create Table: CREATE TABLE `emp` (
  `ename` varchar(20) DEFAULT NULL,
  `hiredate` date DEFAULT NULL,
  `sal` decimal(10,2) DEFAULT NULL,
  `deptno` int(2) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=gbk
1 row in set (0.00 sec)

mysql> set session sql_mode='NO_TABLE_OPTIONS';
Query OK, 0 rows affected (0.00 sec)

mysql>  show create table emp \G;
*************************** 1. row ***************************
       Table: emp
     Create Table: CREATE TABLE `emp` (
  `ename` varchar(20) DEFAULT NULL,
  `hiredate` date DEFAULT NULL,
  `sal` decimal(10,2) DEFAULT NULL,
  `deptno` int(2) DEFAULT NULL
)
1 row in set (0.00 sec)
```

16.4 小结

本节中介绍了 SQL Mode 的含义以及实际用途，重点讨论了以下内容。

- SQL Mode 的“严格模式”为 MySQL 提供了很好的数据校验功能，保证了数据的准确性，TRADITIONAL、STRICT_TRANS_TABLES 是常用的两种严格模式。
- SQL Mode 的多种模式可以灵活组合，组合后的模式可以更好地满足应用程序的需求。尤其在数据迁移中，SQL Mode 的使用更为重要。

第17章　常用 SQL 技巧和常见问题

在日常开发过程中，大家可能会遇到一些常规方法很难完成的工作，而往往这些工作可以通过一些很简单的 SQL 技巧来完成。因此本章将介绍一下在开发中经常遇到的问题以及针对这些问题的常用 SQL 技巧。

17.1 正则表达式的使用

正则表达式（Regular Expression），是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。在很多文本编辑器或其他工具裡，正则表达式通常被用来检索和/或替换那些符合某个模式的文本内容。许多程序设计语言都支持利用正则表达式进行字符串操作。例如，在 Perl 中就内建了一个功能强大的正则表达式引擎。正则表达式这个概念最初是由 UNIX 中的工具软件（例如 SED 和 GREP）普及开的，通常缩写成“REGEX”或者“REGEXP”。MySQL 利用 REGEXP 命令提供给用户扩展的正则表达式功能，REGEXP 实现的功能，类似 UNIX 上 GREP 和 SED 的功能，并且 REGEXP 在进行模式匹配时是区分大小写的。熟悉并掌握 REGEXP 的功能可以使模式匹配工作事半功倍。

MySQL 5.0 中可以使用的模式序列如表 17-1 所示。

表 17-1　　正则表达式中的模式

序列	序列说明
^	在字符串的开始处进行匹配
$	在字符串的末尾处进行匹配
.	匹配任意单个字符，包括换行符
[...]	匹配出括号内的任意字符
[^...]	匹配不出括号内的任意字符
a*	匹配零个或多个 a（包括空串）
a+	匹配 1 个或多个 a（不包括空串）
a?	匹配 1 个或零个 a
a1|a2	匹配 a1 或 a2
a(m)	匹配 m 个 a
a(m,)	匹配 m 个或更多个 a
a(m,n)	匹配 m 到 n 个 a
a(,n)	匹配 0 到 n 个 a

（…）	将模式元素组成单一元素

下面举一些例子来介绍常用正则表达式的使用方法。

- “^”在字符串的开始处进行匹配，返回结果为 1 表示匹配，返回结果为 0 表示不匹配。下例中尝试匹配字符串“abcdefg”是否以字符“a”开始：

```
mysql> select 'abcdefg' REGEXP '^a';
+-----------------------+
| 'abcdefg' REGEXP '^a' |
+-----------------------+
|                     1 |
+-----------------------+
1 row in set (0.39 sec)
```

- “$”在字符串的末尾处进行匹配，下例中尝试匹配字符串“abcdefg”是否以字符“g”结束：

```
mysql> select 'abcdefg' REGEXP 'g$';
+-----------------------+
| 'abcdefg' REGEXP 'g$' |
+-----------------------+
|                     1 |
+-----------------------+
1 row in set (0.01 sec)
```

- “.”匹配任意单个字符，包括换行符。下例中字符串“abcdefg”尝试匹配单字符“h”和“f”：

```
mysql> select 'abcdefg' REGEXP '.h', 'abcdefg' REGEXP '.f';
+-----------------------+-----------------------+
| 'abcdefg' REGEXP '.h' | 'abcdefg' REGEXP '.f' |
+-----------------------+-----------------------+
|                     0 |                     1 |
+-----------------------+-----------------------+
1 row in set (0.00 sec)
```

- “[…]”匹配出括号内的任意字符。下例中字符串“abcdefg”尝试匹配“fhk”中的任意一个字符，如果有一个字符能匹配上，则返回 1。

```
mysql> select 'abcdefg' REGEXP "[fhk]";
+--------------------------+
| 'abcdefg' REGEXP "[fhk]" |
+--------------------------+
|                        1 |
```

```
    +-------------------------+
    1 row in set (0.01 sec)
```

- “[^...]”匹配不出括号内的任意字符。和“[...]”刚好相反。下例中字符串“efg”和“X”中如果有任何一个字符匹配不上“[XYZ]”中的任意一个字符，则返回 0；如果全部都能匹配上，则返回 1。

```
mysql> select 'efg' REGEXP "[^XYZ]",'X' REGEXP "[^XYZ]";
+----------------------+--------------------+
| 'efg' REGEXP "[^XYZ]" | 'X' REGEXP "[^XYZ]" |
+----------------------+--------------------+
|                    1 |                  0 |
+----------------------+--------------------+
1 row in set (0.00 sec)
```

上面我们介绍了正则表达式的常见使用方法。但是在实际工作中正则表达式到底什么地方会用到呢？下面举一个实际的例子，使用正则表达式查询出使用 163.com 邮箱的用户和邮箱。

（1）创建测试数据：

```
    mysql> create table t(name varchar(20),email varchar(40));
    Query OK, 0 rows affected (0.00 sec)

    mysql> insert into t values('beijing','beijing@163.com');
    Query OK, 1 row affected (0.00 sec)

    mysql> insert into t values('beijing126','beijing@126.com');
    Query OK, 1 row affected (0.00 sec)

    mysql> insert into t values('beijing188','beijing@188.com');
    Query OK, 1 row affected (0.00 sec)

    mysql> select * from t;
    +------------+-----------------+
    | name       | email           |
    +------------+-----------------+
    | beijing    | beijing@163.com |
    | beijing126 | beijing@126.com |
    | beijing188 | beijing@188.com |
    +------------+-----------------+
    3    rows in set (0.00 sec)
```

（2）使用正则表达式“$”和“[...]”进行匹配：

```
    mysql> select name ,email  from t where  email REGEXP "@163[.,]com$";
    +---------+-----------------+
    | name    | email           |
    +---------+-----------------+
    | beijing | beijing@163.com |
```

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

从上例中可以看出，如果不使用正则表达式而使用普通的 LIKE 语句，则 WHERE 条件需要写成如下格式：

```
email like “@163%.com” or email like “@163%,com”
```

显然，采用正则表达式使得我们的写法更加简单易读。

17.2 巧用 RAND()提取随机行

大多数数据库都会提供产生随机数的包或者函数，通过这些包或者函数可以产生用户需要的随机数，也可以用来从数据表中抽取随机产生的记录，这对一些抽样分析统计是非常有用的。例如ORACLE中用DBMS_RANDOM包产生随机数，而在MySQL中，产生随机数的方法是RAND()函数。可以利用这个函数与ORDER BY子句一起完成随机抽取某些行的功能。它的原理其实就是ORDER BY RAND()能够把数据随机排序。

例如，可按照随机顺序检索数据行：

```
mysql> select * from sales2 order by rand();
+------+------------+--------+------+
| id   | company_id | moneys | year |
+------+------------+--------+------+
| 106  | 106        | 106    | 2007 |
| 362  | 362        | 362    | 0000 |
| 702  | 702        | 702    | 0000 |
| 871  | 871        | 871    | 0000 |
| 398  | 398        | 398    | 0000 |
| 639  | 639        | 639    | 0000 |
| 45   | 45         | 45     | 1946 |
| 129  | 129        | 129    | 2030 |
...
...
```

这样的话，如果想随机抽取一部分样本的时候，就可以把数据随机排序后再抽取前 n 条记录就可以了，比如：

```
mysql> select * from sales2 order by rand() limit 5;
+------+------------+--------+------+
| id   | company_id | moneys | year |
+------+------------+--------+------+
| 876  | 876        | 876    | 0000 |
| 283  | 283        | 283    | 0000 |
| 442  | 442        | 442    | 0000 |
| 874  | 874        | 874    | 0000 |
| 849  | 849        | 849    | 0000 |
+------+------------+--------+------+
5 rows in set (0.00 sec);
```

上面的例子从 sales2 表中随机抽取了 5 个样本，随机抽取样本对总体的统计具有十分重要的

意义，因此这个函数非常有用。

17.3 利用 GROUP BY 的 WITH ROLLUP 子句做统计

在SQL语句中，使用GROUP BY的WITH ROLLUP字句可以检索出更多的分组聚合信息，它不仅仅能像一般的GROUP BY语句那样检索出各组的聚合信息，还能检索出本组类的整体聚合信息，具体如下例所示。

（1）创建表sales并初始化数据：

```
create table sales
(
      year       int not null,
      country varchar(20) not null,
      product varchar(32) not null,
      profit    int
);
 insert into sales values(2004,'china','tnt1',2001);
 insert into sales values(2004,'china','tnt2',2002);
 insert into sales values(2004,'china','tnt3',2003);
 insert into sales values(2005,'china','tnt1',2004);
 insert into sales values(2005,'china','tnt2',2005);
 insert into sales values(2005,'china','tnt3',2006);
 insert into sales values(2005,'china','tnt1',2007);
 insert into sales values(2005,'china','tnt2',2008);
 insert into sales values(2005,'china','tnt3',2009);
 insert into sales values(2006,'china','tnt1',2010);
 insert into sales values(2006,'china','tnt2',2011);
 insert into sales values(2006,'china','tnt3',2012);
```

（2）按照year、country、product列分组对profit列进行聚合计算如下：

```
mysql> select year, country, product, sum(profit) from sales group by year, country, product;
+------+---------+---------+-------------+
| year | country | product | sum(profit) |
+------+---------+---------+-------------+
| 2004 | china   | tnt1    | 2001        |
| 2004 | china   | tnt2    | 2002        |
| 2004 | china   | tnt3    | 2003        |
| 2005 | china   | tnt1    | 4011        |
| 2005 | china   | tnt2    | 4013        |
| 2005 | china   | tnt3    | 4015        |
| 2006 | china   | tnt1    | 2010        |
| 2006 | china   | tnt2    | 2011        |
| 2006 | china   | tnt3    | 2012        |
+------+---------+---------+-------------+
```

```
    9 rows in set (0.00 sec)

    mysql> select year, country, product, sum(profit) from sales group by year, country, product
with rollup;
    +------+---------+---------+-------------+
    | year | country | product | sum(profit) |
    +------+---------+---------+-------------+
    | 2004 | china   | tnt1    | 2001        |
    | 2004 | china   | tnt2    | 2002        |
    | 2004 | china   | tnt3    | 2003        |
    | 2004 | china   |         | 6006        |
    | 2004 |         |         | 6006        |
    | 2005 | china   | tnt1    | 4011        |
    | 2005 | china   | tnt2    | 4013        |
    | 2005 | china   | tnt3    | 4015        |
    | 2005 | china   |         | 12039       |
    | 2005 |         |         | 12039       |
    | 2006 | china   | tnt1    | 2010        |
    | 2006 | china   | tnt2    | 2011        |
    | 2006 | china   | tnt3    | 2012        |
    | 2006 | china   |         | 6033        |
    | 2006 |         |         | 6033        |
    | NULL |         |         | 24078       |
    +------+---------+---------+-------------+
    16 rows in set (0.00 sec)
```

从上面的例子中可以看到第 2 个 SQL 语句的结果比第一个 SQL 语句的结果多出了很多行，而这些行反映出了更多的信息，例如，第 2 个 SQL 语句的结果的前 3 行表示 2004 年在中国各个产品（tnt1、tnt2、tnt3）的利润，而第 4 行表示 2004 年在中国所有产品的利润是 6006，这个信息在第一个 SQL 语句中是不能反映出来的，第 5 行表示 2004 年全世界所有产品的利润是 6006（当然这里的 country 字段只有 china）。

其实 WITH ROLLUP 反映的是一种 OLAP 思想，也就是说这一个 GROUP BY 语句执行完成后可以满足用户想要得到的任何一个分组以及分组组合的聚合信息值。

> 注意：1、当使用 ROLLUP 时，不能同时使用 ORDER BY 子句进行结果排序。换言之，ROLLUP 和 ORDER BY 是互相排斥的
>
> 2、LIMIT 用在 ROLLUP 后面。

17.4 用 BIT GROUP FUNCTIONS 做统计

在本小节，主要介绍如何共同使用 GROUP BY 语句和 BIT_AND、BIT_OR 函数完成统计工作。这两个函数的一般用途就是做数值之间的逻辑位运算，但是，当把它们与 GROUP BY 子句联合使用的时候就可以做一些其他的任务。

假设现在有这样一个任务：一个超市需要记录每个用户每次来超市都购买了哪些商品。为了

将问题简单化，假设该超市只有面包、牛奶、饼干、啤酒 4 种商品。那么通常该怎么做呢？一般先建立一个购物单表，里面记录购物发生的时间、顾客信息等；然后再建立一个购物单明细表，里面记录该顾客所购买的商品。这样设计表结构的优点是顾客所购买的商品的详细信息可以记录下来，比如数量、单价等，但是如果目前的这个任务只需要知道用户购买商品的种类和每次购物总价等信息的话，那么这种数据库结构的设计就显得太复杂了。一般还可能会想到用一个表实现这个功能，并且用一个字段以字符串的形式记录顾客所购买的所有商品的商品号，这也是一种方法，但是如果顾客一次购买商品比较多的话，需要很大的存储空间，而且将来做各种统计的时候也会捉襟见肘。

下面给出一种新的解决办法，类似于上面讲到的第二种方案，仍然用一个字段表示顾客购买商品的信息，但是这个字段是数值型的而不是字符型的，该字段存储一个十进制数字，当它转换成二进制的时候，那么每一位代表一种商品，而且如果所在位是“1”那么表示顾客购买了该种商品，“0”表示没有购买该种商品。比如数值的第 1 位代表面包（规定从右向左开始计算）、第 2 位代表牛奶、第 3 位代表饼干、第 4 位代表啤酒，这样如果一个用户购物单的商品列的数值为 5，那么二进制表示为 0101，这样从右向左第 1 位和第 3 位是 1，那么就可以知道这个用户购买了面包和饼干，而如果这个客户有多个这样的购物单（在数据库中就是有多条记录），把这些购物单按用户分组做 BIT_OR()操作就可以知道这个用户都购买过什么商品。

下面举例说明一下这个操作，首先初始化一组数据：

```
mysql> create table order_rab (id int,customer_id int,kind int);
Query OK, 0 rows affected (0.05 sec)
mysql> insert into order_rab values (1,1,5),(2,1,4);
Query OK, 2 rows affected (0.00 sec)
mysql> insert into order_rab values (3,2,3),(4,2,4);
Query OK, 2 rows affected (0.00 sec)
mysql> select * from order_rab;
+------+-------------+------+
| id   | customer_id | kind |
+------+-------------+------+
| 1    | 1           | 5    |
| 2    | 1           | 4    |
| 3    | 2           | 3    |
| 4    | 2           | 4    |
+------+-------------+------+
4 rows in set (0.00 sec)
```

其中 customerid 是顾客编号，kind 是所购买的商品，初始化了两个顾客 1 和 2 的数据，他们每人购物两次，前者购买的商品数值是 5 和 4，转化为二进制分别为 0101、0100，表示这个顾客第一次购买了牛奶和啤酒，第二次购买了牛奶；后者购买的商品数值是 3 和 4，转化为二进制分别为 0011、0100，表示这个顾客第一次购买了饼干和啤酒，第二次购买了牛奶。

下面用 BIT_OR()函数与 GROUP BY 子句联合起来，统计一下这两个顾客在这个超市一共都购买过什么商品，如下例：

```
mysql> select customer_id,bit_or(kind) from order_rab group by customer_id;
+-------------+--------------+
| customer_id | bit_or(kind) |
```

```
+-------------+--------------+
| 1           | 5            |
| 2           | 7            |
+-------------+--------------+
2 rows in set (0.00 sec)
```

可以看到顾客 1 的 BIT_OR()结果是 5 即 0101，表示这个顾客在本超市购买过牛奶和啤酒；顾客 2 的 BIT_OR()结果是 7 即 0111，表示这个顾客在本超市购买过牛奶、饼干、啤酒。

下面解释一下数据库在处理这个逻辑时的计算过程，以第一个顾客举例，那么 BIT_OR(kind)就相当于把 kind 的各个值做了一个“或”操作，最终结果是十进制的 5。逻辑计算公式如下：

```
#     ..0101
#     ..0100
#  OR ..0000
#  ---------
#     ..0101
```

同理，可以用 BIT_AND()统计每个顾客每次来本超市都会购买的商品，具体如下：

```
mysql> select customer_id,bit_and(kind) from order_rab group by customer_id;
+-------------+---------------+
| customer_id | bit_and(kind) |
+-------------+---------------+
| 1           | 4             |
| 2           | 0             |
+-------------+---------------+
2 rows in set (0.01 sec)
```

顾客 1 的 BIT_AND()结果是 4 即 0100，表示顾客 1 每次来本超市都会购买牛奶；顾客 2 的 BIT_AND()结果是 0 即 0000，表示顾客 2 没有每次来本超市都会购买的商品。

数据库在处理 BIT_AND()的时候就是把 kind 的各个值做了一个“与”操作，拿顾客 1 举例说明一下，逻辑计算公式如下：

```
#     ..0101
#     ..0100
# AND ..1111
#  ----------
#     ..0100
```

从上面的例子可以看出，这种数据库结构设计的好处就是能用很简洁的数据表示很丰富的信息，这种方法能够大大地节省存储空间，而且能够提高部分统计计算的速度。不过需要注意的是，这种设计其实损失了顾客购买商品的详细信息，比如购买商品的数量、当时单价、是否有折扣、是否有促销等，因此还要根据应用的实际情况有选择地考虑数据库结构设计。

17.5 数据库名、表名大小写问题

在 MySQL 中，数据库对应操作系统下的数据目录。数据库中的每个表至少对应数据库目录中的一个文件（也可能是多个，这取决于存储引擎）。因此，所使用操作系统的大小写敏感性决定了数据库名和表名的大小写敏感性。在大多数 UNIX 环境中，由于操作系统对大小写的敏感性导致了数据库名和表名对大小写敏感性，而在 Windows 中由于操作系统本身对大

小写不敏感，因此在 Windows 下 MySQL 数据库名和表名对大小写也不敏感。
列、索引、存储子程序和触发器名在任何平台上对大小写不敏感。默认情况下，表别名在 UNIX 中对大小写敏感，但在 Windows 或 Mac OS X 中对大小写不敏感。下面的查询在 UNIX 中会报错，因为它同时引用了别名 a 和 A：

```
mysql> select id from order_rab a where A.id = 1;
ERROR 1054 (42S22): Unknown column 'A.id' in 'where clause'
```

然而，该查询在 Windows 中是可以的。要想避免出现差别，最好采用一致的转换，例如，总是用小写创建并引用数据库名和表名。
在 MySQL 中如何在硬盘上保存、使用表名和数据库名由 lower_case_tables_name 系统变量决定，可以在启动 mysqld 时设置这个系统变量。lower_case_tables_name 可以采用如表 17-2 所示的任一值。

表 17-2　　lower_case_tables_name 的取值范围

值	含义
0	使用 CREATE TABLE 或 CREATE DATABASE 语句指定的大写和小写在硬盘上保存表名和数据库名。名称对大小写敏感。在 UNIX 系统中的默认设置就是这个值
1	表名在硬盘上以小写保存，名称对大小写敏感。MySQL 将所有表名转换为小写以便存储和查找。该值为 Windows 和 Mac OS X 系统中的默认值
2	表名和数据库名在硬盘上使用 CREATE TABLE 或 CREATE DATABASE 语句指定的大小写进行保存，但 MySQL 将它们转换为小写以便查找。此值只在对大小写不敏感的文件系统上适用

如果只在一个平台上使用 MySQL，通常不需要更改 lower_case_tables_name 变量。然而，如果用户想要在对大小写敏感性不同的文件系统的平台之间转移表，就会遇到困难。例如，在 UNIX 中，my_tables 和 MY_tables 是两个不同的表，但在 Windows 中，这两个表名相同。
在 UNIX 中使用 lower_case_tables_name=0，而在 Windows 中使用 lower_case_tables_name=2，这样可以保留数据库名和表名的大小写。不利之处是必须确保在 Windows 中的所有 SQL 语句总是正确地使用大小写来引用数据库名和表名，如果 SQL 语句中没有正确引用数据库名和表名的大小写，那么虽然在 Windows 中能正确执行，但是如果将查询转移到 UNIX 中，大小写不正确，将会导致查询失败。

> 注意：1、在 UNIX 中将 lower_case_tables_name 设置为 1 并且重启 mysqld 之前，必须先将旧的数据库名和表名转换为小写。
>
> 2、尽管在某些平台中数据库名和表名对大小写不敏感，但是最好养成在同一查询中使用相同的大小写来引用给定的数据库名或表名的习惯。

17.6 使用外键需要注意的问题

在 MySQL 中，InnoDB 存储引擎支持对外部关键字约束条件的检查。而对于其他类型存储引擎的表，当使用 REFERENCES tbl_name(col_name)子句定义列时可以使用外部关键字，但是该子句没有实际的效果，只作为备忘录或注释来提醒用户目前正定义的列指向另一个表中的一个列。
例如，下面的 myisam 表外键就没有起作用：

```
mysql> create table users(id int,name varchar(10),primary key(id)) engine=myisam;
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> create table books(id int,bookname varchar(10),userid int ,primary key(id),constraint fk_userid_id foreign key(userid) references users(id)) engine=myisam;
Query OK, 0 rows affected (0.03 sec)

mysql> insert into books values(1,'book1',1);
Query OK, 1 row affected (0.00 sec)
```

如果用 InnoDB 存储引擎建表的话，外键就会起作用，具体如下：

```
mysql> create table users2(id int,name varchar(10),primary key(id)) engine=innodb;
Query OK, 0 rows affected (0.14 sec)

mysql> create table books2(id int,bookname varchar(10),userid int ,primary key(id),constraint fk_userid_id foreign key(userid) references users2(id)) engine=innodb;
Query OK, 0 rows affected (0.18 sec)

mysql> insert into books2 values(1,'book1',1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`sakila/books2`, CONSTRAINT `fk_userid_id` FOREIGN KEY (`userid`) REFERENCES `users2` (`id`))
```

而且，用 show create table 命令查看建表语句的时候，发现 MyISAM 存储引擎的并不显示外键的语句，而 InnoDB 存储引擎的就显示外键语句，具体如下：

```
mysql> show create table books\G;
*************************** 1. row ***************************
       Table: books
Create Table: CREATE TABLE `books` (
  `id` int(11) NOT NULL DEFAULT '0',
  `bookname` varchar(10) DEFAULT NULL,
  `userid` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_userid_id` (`userid`)
) ENGINE=MyISAM DEFAULT CHARSET=gbk
1 row in set (0.00 sec)

mysql> show create table books2\G;
*************************** 1. row ***************************
       Table: books2
Create Table: CREATE TABLE `books2` (
  `id` int(11) NOT NULL DEFAULT '0',
  `bookname` varchar(10) DEFAULT NULL,
  `userid` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `fk_userid_id` (`userid`),
  CONSTRAINT `fk_userid_id` FOREIGN KEY (`userid`) REFERENCES `users2` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk
1 row in set (0.00 sec)
```

17.7 小结

本章讲述了在数据库开发时会用到的一些 SQL 小技巧和需要注意的问题。

- 应用正则表达式可以使用户能够在一段很大的字符串中找到符合自己规则的一些语句，这个功能在进行匹配查找的时候非常有用。
- RAND()函数与 ORDER BY 字句的配合使用能够实现随机抽取样本的功能，这个技巧在进行数据统计的时候很方便。
- GROUP BY 的 WITH ROLLUP 字句能够实现类似于 OLAP 的查询。
- BIT 函数与 GROUP BY 子句的联合使用在某些应用场合可以大大降低存储量，提高统计查询效率。
- MySQL 数据库名和表名的大小写与操作系统对大小写的敏感性关系密切，因此需要格外引起用户的注意。
- MySQL 的外键功能仅对 InnoDB 存储引擎的表有作用，其他类型存储引擎的表虽然可以建立外键，但是并不能起到外键的作用。

第18章 SQL 优化

在应用的的开发过程中，由于初期数据量小，开发人员写 SQL 语句时更重视功能上的实现，但是当应用系统正式上线后，随着生产数据量的急剧增长，很多 SQL 语句开始逐渐显露出性能问题，对生产的影响也越来越大，此时这些有问题的 SQL 语句就成为整个系统性能的瓶颈，因此我们必须要对它们进行优化，本章将详细介绍在 MySQL 中优化 SQL 语句的方法。

18.1 优化 SQL 语句的一般步骤

当面对一个有 SQL 性能问题的数据库时，我们应该从何处入手来进行系统的分析，使得能够尽快定位问题 SQL 并尽快解决问题，本节将向读者介绍这个过程。

18.1.1 通过 show status 命令了解各种 SQL 的执行频率

MySQL 客户端连接成功后，通过 show [session|global]status 命令可以提供服务器状态信息，也可以在操作系统上使用 mysqladmin extended-status 命令获得这些消息。show [session|global] status 可以根据需要加上参数“session”或者“global”来显示 session 级（当前连接）的统计结果和 global 级（自数据库上次启动至今）的统计结果。如果不写，默认使用参数是“session”。

下面的命令显示了当前 session 中所有统计参数的值：

```
mysql> show status like 'Com_%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| Com_admin_commands        | 0     |
| Com_alter_db              | 0     |
```

```
| Com_alter_event          | 0     |
| Com_alter_table          | 0     |
| Com_analyze              | 0     |
| Com_backup_table         | 0     |
| Com_begin                | 0     |
| Com_change_db            | 1     |
| Com_change_master        | 0     |
| Com_check                | 0     |
| Com_checksum             | 0     |
| Com_commit               | 0     |
……
```

Com_xxx 表示每个 xxx 语句执行的次数，我们通常比较关心的是以下几个统计参数。

- Com_select：执行 select 操作的次数，一次查询只累加 1。
- Com_insert：执行 INSERT 操作的次数，对于批量插入的 INSERT 操作，只累加一次。
- Com_update：执行 UPDATE 操作的次数。
- Com_delete：执行 DELETE 操作的次数。

上面这些参数对于所有存储引擎的表操作都会进行累计。下面这几个参数只是针对 InnoDB 存储引擎的，累加的算法也略有不同。

- Innodb_rows_read：select 查询返回的行数。
- Innodb_rows_inserted：执行 INSERT 操作插入的行数。
- Innodb_rows_updated：执行 UPDATE 操作更新的行数。
- Innodb_rows_deleted：执行 DELETE 操作删除的行数。

通过以上几个参数，可以很容易地了解当前数据库的应用是以插入更新为主还是以查询操作为主，以及各种类型的 SQL 大致的执行比例是多少。对于更新操作的计数，是对执行次数的计数，不论提交还是回滚都会进行累加。

对于事务型的应用，通过 Com_commit 和 Com_rollback 可以了解事务提交和回滚的情况，对于回滚操作非常频繁的数据库，可能意味着应用编写存在问题。

此外，以下几个参数便于用户了解数据库的基本情况。

- Connections：试图连接 MySQL 服务器的次数。
- Uptime：服务器工作时间。
- Slow_queries：慢查询的次数。

18.1.2 定位执行效率较低的 SQL 语句

可以通过以下两种方式定位执行效率较低的 SQL 语句。

- 通过慢查询日志定位那些执行效率较低的 SQL 语句，用--log-slow-queries[=file_name]选项启动时，mysqld 写一个包含所有执行时间超过 long_query_time 秒的 SQL 语句的日志文件。具体可以查看本书第 26 章中日志管理的相关部分。
- 慢查询日志在查询结束以后才纪录，所以在应用反映执行效率出现问题的时候查询慢查询日志并不能定位问题，可以使用 show processlist 命令查看当前 MySQL 在进行的线程，包括线程的状态、是否锁表等，可以实时地查看 SQL 的执行情况，同时对一些锁表操作进行优化。

18.1.3 通过 EXPLAIN 分析低效 SQL 的执行计划

通过以上步骤查询到效率低的 SQL 语句后，可以通过 EXPLAIN 或者 DESC 命令获取 MySQL 如何执行 SELECT 语句的信息，包括在 SELECT 语句执行过程中表如何连接和连接的顺序，比如想计算 2006 年所有公司的销售额，需要关联 sales 表和 company 表，并且对 moneys 字段做求和（sum）操作，相应 SQL 的执行计划如下：

```
    mysql> explain select sum(moneys) from sales a,company b where a.company_id = b.id and a.year
= 2006\G;
    *************************** 1. row ***************************
               id: 1
      select_type: SIMPLE
            table: a
             type: ALL
    possible_keys: NULL
              key: NULL
          key_len: NULL
              ref: NULL
             rows: 1000
            Extra: Using where
    *************************** 2. row ***************************
               id: 1
      select_type: SIMPLE
            table: b
             type: ref
    possible_keys: ind_company_id
              key: ind_company_id
          key_len: 5
              ref: sakila.a.company_id
             rows: 1
            Extra: Using where; Using index
    2 rows in set (0.00 sec)
```

每个列的简单解释如下：

- select_type：表示 SELECT 的类型，常见的取值有 SIMPLE（简单表，即不使用表连接或者子查询）、PRIMARY（主查询，即外层的查询）、UNION（UNION 中的第二个或者后面的查询语句）、SUBQUERY（子查询中的第一个 SELECT）等。
- table：输出结果集的表。
- type：表示表的连接类型，性能由好到差的连接类型为 system（表中仅有一行，即常量表）、const（单表中最多有一个匹配行，例如 primary key 或者 unique index）、eq_ref（对于前面的每一行，在此表中只查询一条记录，简单来说，就是多表连接中使用 primary key 或者 unique index）、ref（与 eq_ref 类似，区别在于不是使用 primary key 或者 unique index，而是使用普通的索引）、ref_or_null（与 ref 类似，区别在于条件中包含对 NULL 的查询）、index_merge（索引合并优化）、unique_subquery（in 的后面是一个查询主键字段的子查询）、index_subquery（与 unique_subquery 类似，区别在于 in 的后面是查询非唯一索引字段的子查询）、range（单表中的范围查询）、index（对于前面的每一行，都通过查询索引来得到数据）、all（对于前面的每一行，

都通过全表扫描来得到数据）。

- possible_keys：表示查询时，可能使用的索引。
- key：表示实际使用的索引。
- key_len：索引字段的长度。
- rows：扫描行的数量。
- Extra：执行情况的说明和描述。

18.1.4 确定问题并采取相应的优化措施

经过以上步骤，基本就可以确认问题出现的原因。此时用户可以根据情况采取相应的措施，进行优化提高执行的效率。

在上面的例子中，已经可以确认是对 a 表的全表扫描导致效率的不理想，那么对 a 表的 year 字段创建索引，具体如下：

```
mysql> create index ind_sales2_year on sales2(year);
Query OK, 1000 rows affected (0.03 sec)
Records: 1000  Duplicates: 0  Warnings: 0
```

创建索引后，再看一下这条语句的执行计划，具体如下：

```
mysql> explain select sum(moneys) from sales2 a,company2 b where a.company_id = b.id and a.year = 2006\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: a
         type: ref
possible_keys: ind_sales2_year
          key: ind_sales2_year
      key_len: 2
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: b
         type: ref
possible_keys: ind_company2_id
          key: ind_company2_id
      key_len: 5
          ref: sakila.a.company_id
         rows: 1
        Extra: Using where; Using index
2 rows in set (0.00 sec)
```

可以发现建立索引后对 a 表需要扫描的行数明显减少（从 1000 行减少到 1 行），可见索引的使用可以大大提高数据库的访问速度，尤其在表很庞大的时候这种优势更为明显。

18.2 索引问题

索引是数据库优化中最常用也是最重要的手段之一，通过索引通常可以帮助用户解决大多数的 SQL 性能问题。本节将对 MySQL 中的索引的分类、存储、使用方法做详细的介绍。

18.2.1 索引的存储分类

MyISAM 存储引擎的表的数据和索引是自动分开存储的，各自是独立的一个文件；InnoDB 存储引擎的表的数据和索引是存储在同一个表空间里面，但可以有多个文件组成。

MySQL 中索引的存储类型目前只有两种（BTREE 和 HASH），具体和表的存储引擎相关：MyISAM 和 InnoDB 存储引擎都只支持 BTREE 索引；MEMORY/HEAP 存储引擎可以支持 HASH 和 BTREE 索引。

MySQL 目前不支持函数索引，但是能对列的前面某一部分进索引，例如 name 字段，可以只取 name 的前 4 个字符进行索引，这个特性可以大大缩小索引文件的大小，用户在设计表结构的时候也可以对文本列根据此特性进行灵活设计。下面是创建前缀索引的一个例子：

```
mysql> create index ind_company2_name on company2(name(4));
Query OK, 1000 rows affected (0.03 sec)
Records: 1000  Duplicates: 0  Warnings: 0
```

18.2.2 MySQL 如何使用索引

索引用于快速找出在某个列中有一特定值的行。对相关列使用索引是提高 SELECT 操作性能的最佳途径。

查询要使用索引最主要的条件是查询条件中需要使用索引关键字，如果是多列索引，那么只有查询条件使用了多列关键字最左边的前缀时，才可以使用索引，否则将不能使用索引。

1. 使用索引

在 MySQL 中，下列几种情况下有可能使用到索引。

（1）对于创建的多列索引，只要查询的条件中用到了最左边的列，索引一般就会被使用，举例说明如下。

首先按 company_id，moneys 的顺序创建一个复合索引，具体如下：

```
mysql> create index ind_sales2_companyid_moneys on sales2(company_id,moneys);
Query OK, 1000 rows affected (0.03 sec)
Records: 1000  Duplicates: 0  Warnings: 0
```

然后按 company_id 进行表查询，具体如下：

```
mysql> explain select * from sales2 where company_id = 2006\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ref
possible_keys: ind_sales2_companyid_moneys
```

```
          key: ind_sales2_companyid_moneys
      key_len: 5
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

可以发现即便 where 条件中不是用的 company_id 与 moneys 的组合条件，索引仍然能用到，这就是索引的前缀特性。但是如果只按 moneys 条件查询表，那么索引就不会被用到，具体如下：

```
mysql> explain select * from sales2 where moneys = 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra: Using where
1 row in set (0.00 sec)
```

（2）对于使用 like 的查询，后面如果是常量并且只有%号不在第一个字符，索引才可能会被使用，来看下面两个执行计划：

```
mysql> explain select * from company2 where name like '%3'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: company2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra: Using where
1 row in set (0.00 sec)

mysql> explain select * from company2 where name like '3%'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: company2
         type: range
```

```
possible_keys: ind_company2_name
          key: ind_company2_name
      key_len: 11
          ref: NULL
         rows: 103
        Extra: Using where
1 row in set (0.00 sec)
```

可以发现第一个例子没有使用索引，而第二例子就能够使用索引，区别就在于“%”的位置不同，前者把“%”放到第一位就不能用到索引，而后者没有放到第一位就使用了索引。

另外，如果如果 like 后面跟的是一个列的名字，那么索引也不会被使用。

（3）如果对大的文本进行搜索，使用全文索引而不用使用 like ‘%...%’。

（4）如果列名是索引，使用 column_name is null 将使用索引。如下例中查询 name 为 null 的记录就用到了索引：

```
mysql> explain select * from company2 where name is null\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: company2
         type: ref
possible_keys: ind_company2_name
          key: ind_company2_name
      key_len: 11
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

2. 存在索引但不使用索引

在下列情况下，虽然存在索引，但是 MySQL 并不会使用相应的索引。

（1） 如果 MySQL 估计使用索引比全表扫描更慢，则不使用索引。例如如果列 key_part1 均匀分布在 1 和 100 之间，下列查询中使用索引就不是很好：

```
SELECT * FROM table_name where key_part1 > 1 and key_part1 < 90;
```

（2） 如果使用 MEMORY/HEAP 表并且 where 条件中不使用“=”进行索引列，那么不会用到索引。heap 表只有在“=”的条件下才会使用索引。

（3） 用 or 分割开的条件，如果 or 前的条件中的列有索引，而后面的列中没有索引，那么涉及到的索引都不会被用到，例如：

```
mysql> show index from sales\G;
*************************** 1. row ***************************
       Table: sales
  Non_unique: 1
    Key_name: ind_sales_year
Seq_in_index: 1
 Column_name: year
```

```
       Collation: A
     Cardinality: NULL
        Sub_part: NULL
          Packed: NULL
            Null:
      Index_type: BTREE
         Comment:
1 row in set (0.00 sec)
```

从上面可以发现只有 year 列上面有索引，来看如下的执行计划：

```
mysql> explain select * from sales where year = 2001 or country = 'China'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales
         type: ALL
possible_keys: ind_sales_year
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 12
        Extra: Using where
1 row in set (0.00 sec)
```

可见虽然在 year 这个列上存在索引 ind_sales_year，但是这个 SQL 语句并没有用到这个索引，原因就是 or 中有一个条件中的列没有索引。

（4） 如果不是索引列的第一部分，如下例子：

```
mysql> explain select * from sales2 where moneys = 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra: Using where
1 row in set (0.00 sec)
```

可见虽然在 money 上面建有复合索引，但是由于 money 不是索引的第一列，那么在查询中这个索引也不会被 MySQL 采用。

（5） 如果 like 是以%开始，例如：

```
mysql> explain select * from company2 where name like '%3'\G;
*************************** 1. row ***************************
           id: 1
```

```
  select_type: SIMPLE
        table: company2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra: Using where
1 row in set (0.00 sec)
```

可见虽然在 name 上建有索引，但是由于 where 条件中 like 的值的“%”在第一位了，那么 MySQL 也不会采用这个索引。

（6） 如果列类型是字符串，那么一定记得在 where 条件中把字符常量值用引号引起来，否则的话即便这个列上有索引，MySQL 也不会用到的，因为，MySQL 默认把输入的常量值进行转换以后才进行检索。如下面的例子中 company2 表中的 name 字段是字符型的，但是 SQL 语句中的条件值 294 是一个数值型值，因此即便在 name 上有索引，MySQL 也不能正确地用上索引，而是继续进行全表扫描。

```
mysql> explain select * from company2 where name = 294\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: company2
         type: ALL
possible_keys: ind_company2_name
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra: Using where
1 row in set (0.00 sec)

mysql> explain select * from company2 where name = '294'\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: company2
         type: ref
possible_keys: ind_company2_name
          key: ind_company2_name
      key_len: 23
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

从上面的例子中可以看到，第一个 SQL 语句中把一个数值型常量赋值给了一个字符型的列 name，那么虽然在 name 列上有索引，但是也没有用到；而第二个 SQL 语句就可以正确使用索引。

18.2.3 查看索引使用情况

如果索引正在工作，Handler_read_key 的值将很高，这个值代表了一个行被索引值读的次数，很低的值表明增加索引得到的性能改善不高，因为索引并不经常使用。

Handler_read_rnd_next 的值高则意味着查询运行低效，并且应该建立索引补救。这个值的含义是在数据文件中读下一行的请求数。如果正进行大量的表扫描，Handler_read_rnd_next 的值较高，则通常说明表索引不正确或写入的查询没有利用索引，具体如下。

```
mysql> show status like 'Handler_read%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Handler_read_first    | 0     |
| Handler_read_key      | 5     |
| Handler_read_next     | 0     |
| Handler_read_prev     | 0     |
| Handler_read_rnd      | 0     |
| Handler_read_rnd_next | 2055  |
+-----------------------+-------+
6 rows in set (0.00 sec)
```

从上面的例子中可以看出，目前使用的 MySQL 数据库的索引情况并不理想。

18.3 两个简单实用的优化方法

对于大多数开发人员来说，可能只希望掌握一些简单实用的优化方法，对于更多更复杂的优化，更倾向于交给专业 DBA 来做。本节将向大家介绍两个简单适用的优化方法。

18.3.1 定期分析表和检查表

分析表的语法如下：

```
ANALYZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, tbl_name] ...
```

本语句用于分析和存储表的关键字分布，分析的结果将可以使得系统得到准确的统计信息，使得 SQL 能够生成正确的执行计划。如果用户感觉实际执行计划并不是预期的执行计划，执行一次分析表可能会解决问题。在分析期间，使用一个读取锁定对表进行锁定。这对于 MyISAM, BDB 和 InnoDB 表有作用。对于 MyISAM 表，本语句与使用 myisamchk -a 相当，下例中对表 sales 做了表分析：

```
mysql> analyze table sales;
+-------------+---------+----------+----------+
| Table       | Op      | Msg_type | Msg_text |
+-------------+---------+----------+----------+
```

```
| sakila.sales | analyze | status   | OK       |
+--------------+---------+----------+----------+
1 row in set (0.00 sec)
```

检查表的语法如下：

```
CHECK TABLE tbl_name [, tbl_name] ... [option] ... option = {QUICK | FAST | MEDIUM | EXTENDED
| CHANGED}
```

检查表的作用是检查一个或多个表是否有错误。CHECK TABLE 对 MyISAM 和 InnoDB 表有作用。对于 MyISAM 表，关键字统计数据被更新，例如：

```
mysql> check table sales;
+--------------+-------+----------+----------+
| Table        | Op    | Msg_type | Msg_text |
+--------------+-------+----------+----------+
| sakila.sales | check | status   | OK       |
+--------------+-------+----------+----------+
1 row in set (0.00 sec)
```

CHECK TABLE 也可以检查视图是否有错误，比如在视图定义中被引用的表已不存在，举例如下。

（1）首先我们创建一个视图。

```
mysql> create view sales_view3 as select *  from sales3;
Query OK, 0 rows affected (0.00 sec)
```

（2）然后 CHECK 一下该视图，发现没有问题。

```
mysql> check table sales_view3;
+-------------------+-------+----------+----------+
| Table             | Op    | Msg_type | Msg_text |
+-------------------+-------+----------+----------+
| sakila.sales_view3 | check | status   | OK       |
+-------------------+-------+----------+----------+
1 row in set (0.00 sec)
```

（3）现在删除掉视图依赖的表。

```
mysql> drop table sales3;
Query OK, 0 rows affected (0.00 sec)
```

- （4）再来 CHECK 一下刚才的视图，发现报错了。

```
mysql> check table sales_view3\G;
*************************** 1. row ***************************
   Table: sakila.sales_view3
      Op: check
Msg_type: error
Msg_text: View 'sakila.sales_view3' references invalid table(s) or column(s) or function(s)
or definer/invoker of view lack rights to use them
1 row in set (0.00 sec)
```

18.3.2 定期优化表

优化表的语法如下：

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG] TABLE tbl_name [, tbl_name] ...
```

如果已经删除了表的一大部分，或者如果已经对含有可变长度行的表（含有 VARCHAR、BLOB 或 TEXT 列的表）进行了很多更改，则应使用 OPTIMIZE TABLE 命令来进行表优化。这个命令可以将表中的空间碎片进行合并，并且可以消除由于删除或者更新造成的空间浪费，但 OPTIMIZE TABLE 命令只对 MyISAM、BDB 和 InnoDB 表起作用。

以下例子显示了优化表 sales 的过程：

```
mysql> optimize table sales;
+--------------+----------+----------+----------+
| Table        | Op       | Msg_type | Msg_text |
+--------------+----------+----------+----------+
| sakila.sales | optimize | status   | OK       |
+--------------+----------+----------+----------+
1 row in set (0.00 sec)
```

> 注意：ANALYZE、CHECK、OPTIMIZE 执行期间将对表进行锁定，因此一定注意要在数据库不繁忙的时候执行相关的操作。

18.4 常用 SQL 的优化

前面我们介绍了 MySQL 中怎么样通过索引来优化查询。日常开发中，除了使用查询外，我们还会使用一些其他的常用 SQL，比如 INSERT、GROUP BY 等。对于这些 SQL 语句，我们该怎么样进行优化呢？本节将针对这些 SQL 语句介绍一些优化的方法。

18.4.1 大批量插入数据

当用 load 命令导入数据的时候，适当的设置可以提高导入的速度。

对于 MyISAM 存储引擎的表，可以通过以下方式快速的导入大量的数据。

```
ALTER TABLE tbl_name DISABLE KEYS;
loading the data
ALTER TABLE tbl_name ENABLE KEYS;
```

DISABLE KEYS 和 ENABLE KEYS 用来打开或者关闭 MyISAM 表非唯一索引的更新。在导入大量的数据到一个非空的 MyISAM 表时，通过设置这两个命令，可以提高导入的效率。对于导入大量数据到一个空的 MyISAM 表，默认就是先导入数据然后才创建索引的，所以不用进行设置。

下面例子中，用 LOAD 语句导入数据耗时 115.12 秒：

```
mysql>  load data infile '/home/mysql/film_test.txt' into table film_test2;
Query OK, 529056 rows affected (1 min 55.12 sec)
Records: 529056  Deleted: 0  Skipped: 0  Warnings: 0
```

而用 alter table tbl_name disable keys 方式总耗时 6.34 + 12.25 = 18.59 秒，提高了 6 倍多。

```
mysql> alter table film_test2 disable keys;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql>  load data infile '/home/mysql/film_test.txt' into table film_test2;
Query OK, 529056 rows affected (6.34 sec)
Records: 529056  Deleted: 0  Skipped: 0  Warnings: 0

mysql> alter table film_test2 enable keys;
Query OK, 0 rows affected (12.25 sec)
```

上面是对MyISAM表进行数据导入时的优化措施，对于InnoDB类型的表，这种方式并不能提高导入数据的效率，可以有以下几种方式提高InnoDB表的导入效率。

（1）因为 InnoDB 类型的表是按照主键的顺序保存的，所以将导入的数据按照主键的顺序排列，可以有效地提高导入数据的效率。

例如，下面文本film_test3.txt是按表film_test4的主键存储的，那么导入的时候共耗时27.92秒。

```
mysql> load data infile '/home/mysql/film_test3.txt' into table film_test4;
Query OK, 1587168 rows affected (22.92 sec)
Records: 1587168  Deleted: 0  Skipped: 0  Warnings: 0
```

而下面的 film_test4.txt 是没有任何顺序的文本，那么导入的时候共耗时 31.16 秒。

```
mysql> load data infile '/home/mysql/film_test4.txt' into table film_test4;
Query OK, 1587168 rows affected (31.16 sec)
Records: 1587168  Deleted: 0  Skipped: 0  Warnings: 0
```

从上面例子可以看出当被导入的文件按表主键顺序存储的时候比不按主键顺序存储的时候快 1.12 倍。

（2）在导入数据前执行 SET UNIQUE_CHECKS=0，关闭唯一性校验，在导入结束后执行 SET UNIQUE_CHECKS=1，恢复唯一性校验，可以提高导入的效率。

例如，当 UNIQUE_CHECKS=1 时：

```
mysql> load data infile '/home/mysql/film_test3.txt' into table film_test4;
Query OK, 1587168 rows affected (22.92 sec)
Records: 1587168  Deleted: 0  Skipped: 0  Warnings: 0
```

当 SET UNIQUE_CHECKS=0 时：

```
mysql> load data infile '/home/mysql/film_test3.txt' into table film_test4;
Query OK, 1587168 rows affected (19.92 sec)
Records: 1587168  Deleted: 0  Skipped: 0  Warnings: 0
```

可见比 UNIQUE_CHECKS=0 的时候比 SET UNIQUE_CHECKS=1 的时候要快一些。

（3）如果应用使用自动提交的方式，建议在导入前执行 SET AUTOCOMMIT=0，关闭自动提交，导入结束后再执行 SET AUTOCOMMIT=1，打开自动提交，也可以提高导入的效率。

例如，当 AUTOCOMMIT=1 时：

```
mysql> load data infile '/home/mysql/film_test3.txt' into table film_test4;
Query OK, 1587168 rows affected (22.92 sec)
Records: 1587168  Deleted: 0  Skipped: 0  Warnings: 0
```

当 AUTOCOMMIT=0 时：

```
mysql> load data infile '/home/mysql/film_test3.txt' into table film_test4;
Query OK, 1587168 rows affected (20.87 sec)
Records: 1587168  Deleted: 0  Skipped: 0  Warnings: 0
```

对比一下可以知道，当 AUTOCOMMIT=0 时比 AUTOCOMMIT=1 时导入数据要快一些。

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

18.4.2 优化 INSERT 语句

当进行数据 INSERT 的时候，可以考虑采用以下几种优化方式。

- 如果同时从同一客户插入很多行，尽量使用多个值表的 INSERT 语句，这种方式将大大缩减客户端与数据库之间的连接、关闭等消耗，使得效率比分开执行的单个 INSERT 语句快(在一些情况中几倍)。下面是一次插入多值的一个例子：

```
insert into test values(1,2),(1,3),(1,4)…
```

- 如果从不同客户插入很多行，能通过使用 INSERT DELAYED 语句得到更高的速度。DELAYED 的含义是让 INSERT 语句马上执行，其实数据都被放在内存的队列中，并没有真正写入磁盘，这比每条语句分别插入要快的多；LOW_PRIORITY 刚好相反，在所有其他用户对表的读写完后才进行插入；
- 将索引文件和数据文件分在不同的磁盘上存放（利用建表中的选项）；
- 如果进行批量插入，可以增加 bulk_insert_buffer_size 变量值的方法来提高速度，但是，这只能对 MyISAM 表使用；
- 当从一个文本文件装载一个表时，使用 LOAD DATA INFILE。这通常比使用很多 INSERT 语句快 20 倍。

18.4.3 优化 GROUP BY 语句

默认情况下，MySQL 对所有 GROUP BY col1，col2....的字段进行排序。这与在查询中指定 ORDER BY col1，col2...类似。因此，如果显式包括一个包含相同的列的 ORDER BY 子句，则对 MySQL 的实际执行性能没有什么影响。

如果查询包括 GROUP BY 但用户想要避免排序结果的消耗，则可以指定 ORDER BY NULL 禁止排序，如下面的例子：

```
mysql> explain select id,sum(moneys) from sales2 group by id\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra: Using temporary; Using filesort
1 row in set (0.00 sec)

mysql> explain select id,sum(moneys) from sales2 group by id order by null\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ALL
possible_keys: NULL
```

```
             key: NULL
         key_len: NULL
             ref: NULL
            rows: 1000
           Extra: Using temporary
1 row in set (0.00 sec)
```

从上面的例子可以看出第一个 SQL 语句需要进行“filesort”，而第二个 SQL 由于 ORDER BY NULL 不需要进行“filesort”，而 filesort 往往非常耗费时间。

18.4.4 优化 ORDER BY 语句：

在某些情况中，MySQL 可以使用一个索引来满足 ORDER BY 子句，而不需要额外的排序。WHERE 条件和 ORDER BY 使用相同的索引，并且 ORDER BY 的顺序和索引顺序相同，并且 ORDER BY 的字段都是升序或者都是降序。

例如，下列 SQL 可以使用索引。

```
SELECT * FROM t1 ORDER BY key_part1,key_part2,... ;
SELECT * FROM t1 WHERE key_part1=1 ORDER BY key_part1 DESC, key_part2 DESC;
SELECT * FROM t1 ORDER BY key_part1 DESC, key_part2 DESC;
```

但是在以下几种情况下则不使用索引：

```
SELECT * FROM t1 ORDER BY key_part1 DESC, key_part2 ASC;
--order by 的字段混合 ASC 和 DESC
SELECT * FROM t1 WHERE key2=constant ORDER BY key1;
--用于查询行的关键字与 ORDER BY 中所使用的不相同
SELECT * FROM t1 ORDER BY key1, key2;
--对不同的关键字使用 ORDER BY：
```

18.4.5 优化嵌套查询

MySQL 4.1 开始支持 SQL 的子查询。这个技术可以使用 SELECT 语句来创建一个单列的查询结果，然后把这个结果作为过滤条件用在另一个查询中。使用子查询可以一次性地完成很多逻辑上需要多个步骤才能完成的 SQL 操作，同时也可以避免事务或者表锁死，并且写起来也很容易。但是，有些情况下，子查询可以被更有效率的连接（JOIN）替代。

在下面的例子中，要从 sales2 表中找到那些在 company2 表中不存在的所有公司的信息：

```
mysql> explain select * from sales2 where company_id not in ( select id from
company2 )\G;
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: sales2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
```

```
           rows: 1000
          Extra: Using where
*************************** 2. row ***************************
             id: 2
    select_type: DEPENDENT SUBQUERY
          table: company2
           type: index_subquery
  possible_keys: ind_company2_id
            key: ind_company2_id
        key_len: 5
            ref: func
           rows: 2
          Extra: Using index
2 rows in set (0.00 sec)
```

如果使用连接（JOIN）来完成这个查询工作，速度将会快很多。尤其是当 company2 表中对 id 建有索引的话，性能将会更好，具体查询如下：

```
mysql> explain select * from sales2 left join company2 on sales2.company_id = company2.id where sales2.company_id is null\G;
*************************** 1. row ***************************
             id: 1
    select_type: SIMPLE
          table: sales2
           type: ref
  possible_keys: ind_sales2_companyid_moneys
            key: ind_sales2_companyid_moneys
        key_len: 5
            ref: const
           rows: 1
          Extra: Using where
*************************** 2. row ***************************
             id: 1
    select_type: SIMPLE
          table: company2
           type: ref
  possible_keys: ind_company2_id
            key: ind_company2_id
        key_len: 5
            ref: sakila.sales2.company_id
           rows: 1
          Extra:
2 rows in set (0.00 sec)
```

从执行计划中可以明显看出查询扫描的记录范围和使用索引的情况都有了很大的改善。

连接（JOIN）之所以更有效率一些，是因为 MySQL 不需要在内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。

18.4.6 MySQL 如何优化 OR 条件

对于含有 OR 的查询子句，如果要利用索引，则 OR 之间的每个条件列都必须用到索引；如果没有索引，则应该考虑增加索引。

例如，首先使用 show index 命令查看表 sales2 的索引，可知它有 3 个索引，在 id、year 两个字段上分别有 1 个独立的索引，在 company_id 和 year 字段上有 1 个复合索引。

```
    mysql> show index from sales2\G;
*************************** 1. row ***************************
       Table: sales2
  Non_unique: 1
    Key_name: ind_sales2_id
Seq_in_index: 1
 Column_name: id
   Collation: A
 Cardinality: 1000
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: sales2
  Non_unique: 1
    Key_name: ind_sales2_year
Seq_in_index: 1
 Column_name: year
   Collation: A
 Cardinality: 250
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
*************************** 3. row ***************************
       Table: sales2
  Non_unique: 1
    Key_name: ind_sales2_companyid_moneys
Seq_in_index: 1
 Column_name: company_id
   Collation: A
 Cardinality: 1000
    Sub_part: NULL
      Packed: NULL
        Null: YES
```

```
  Index_type: BTREE
     Comment:
*************************** 4. row ***************************
       Table: sales2
  Non_unique: 1
    Key_name: ind_sales2_companyid_moneys
Seq_in_index: 2
 Column_name: year
   Collation: A
 Cardinality: 1000
    Sub_part: NULL
      Packed: NULL
        Null: YES
  Index_type: BTREE
     Comment:
4 rows in set (0.00 sec)
```

然后在两个独立索引上面做 OR 操作，具体如下：

```
mysql> explain select * from sales2 where id = 2 or year = 1998\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: index_merge
possible_keys: ind_sales2_id,ind_sales2_year
          key: ind_sales2_id,ind_sales2_year
      key_len: 5,2
          ref: NULL
         rows: 2
        Extra: Using union(ind_sales2_id,ind_sales2_year); Using where
1 row in set (0.00 sec)
```

可以发现查询正确的用到了索引，并且从执行计划的描述中，发现 MySQL 在处理含有 OR 字句的查询时，实际是对 OR 的各个字段分别查询后的结果进行了 UNION。

但是当在建有复合索引的列 company_id 和 moneys 上面做 OR 操作的时候，却不能用到索引，具体结果如下：

```
mysql> explain select * from sales2 where company_id = 3 or moneys = 100\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ALL
possible_keys: ind_sales2_companyid_moneys
          key: NULL
      key_len: NULL
          ref: NULL
```

```
           rows: 1000
          Extra: Using where
1 row in set (0.00 sec)
```

18.4.7 使用 SQL 提示

SQL 提示（SQL HINT）是优化数据库的一个重要手段，简单来说就是在 SQL 语句中加入一些人为的提示来达到优化操作的目的。

下面是一个使用 SQL 提示的例子：

```
SELECT SQL_BUFFER_RESULTS * FROM...
```

这个语句将强制 MySQL 生成一个临时结果集。只要临时结果集生成后，所有表上的锁定均被释放。这能在遇到表锁定问题时或要花很长时间将结果传给客户端时有所帮助，因为可以尽快释放锁资源。

下面是一些在 MySQL 中常用的 SQL 提示。

1. USE INDEX

在查询语句中表名的后面，添加 USE INDEX 来提供希望 MySQL 去参考的索引列表，就可以让 MySQL 不再考虑其他可用的索引。

```
mysql> explain select * from sales2 use index (ind_sales2_id) where id = 3\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ref
possible_keys: ind_sales2_id
          key: ind_sales2_id
      key_len: 5
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec).
```

2. IGNORE INDEX

如果用户只是单纯地想让 MySQL 忽略一个或者多个索引，则可以使用 IGNORE INDEX 作为 HINT。同样是上面的例子，这次来看一下查询过程忽略索引 ind_sales2_id 的情况：

```
mysql> explain select * from sales2 ignore index (ind_sales2_id) where id = 3\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sales2
         type: ALL
```

```
    possible_keys: NULL
              key: NULL
          key_len: NULL
              ref: NULL
             rows: 1000
            Extra: Using where
    1 row in set (0.00 sec).
```

从执行计划可以看出，系统忽略了指定的索引，而使用了全表扫描。

3. FORCE INDEX

为强制 MySQL 使用一个特定的索引，可在查询中使用 FORCE INDEX 作为 HINT。例如，当不强制使用索引的时候，因为 id 的值都是大于 0 的，因此 MySQL 会默认进行全表扫描，而不使用索引，如下所示：

```
    mysql> explain select * from sales2 where id  > 0 \G;
    *************************** 1. row ***************************
               id: 1
      select_type: SIMPLE
            table: sales2
             type: ALL
    possible_keys: ind_sales2_id
              key: NULL
          key_len: NULL
              ref: NULL
             rows: 1000
            Extra: Using where
    1 row in set (0.00 sec)
```

但是，当使用 FORCE INDEX 进行提示时，即便使用索引的效率不是最高，MySQL 还是选择使用了索引，这是 MySQL 留给用户的一个自行选择执行计划的权力。加入 FORCE INDEX 提示后再次执行上面的 SQL：

```
    mysql> explain select * from sales2 force index (ind_sales2_id) where id > 0
\G;
    *************************** 1. row ***************************
               id: 1
      select_type: SIMPLE
            table: sales2
             type: range
    possible_keys: ind_sales2_id
              key: ind_sales2_id
          key_len: 5
              ref: NULL
             rows: 1000
```

```
            Extra: Using where
    1 row in set (0.00 sec).
```

果然，执行计划中使用了 FORCE INDEX 后的索引。

18.5 小结

SQL 优化问题是数据库性能优化最基础也是最重要的一个问题，实践表明很多数据库性能问题都是由不合适的 SQL 语句造成。本章通过实例描述了 SQL 优化的一般过程，从定位一个有性能问题的 SQL 语句到分析产生性能问题的原因，最后到采取什么措施优化 SQL 语句的性能。另外还介绍了优化 SQL 语句经常需要考虑的几个方面，比如索引、表分析、排序等。

第19章　优化数据库对象

在数据库设计过程中，用户可能会经常遇到这种问题：是否应该把所有表都按照第三范式来设计？表里面的字段到底改设置为多大长度合适？这些问题虽然很小，但是如果设计不当则可能会给将来的应用带来很多的性能问题。本章中将介绍 MySQL 中一些数据库对象的优化方法，其中一些方法不仅仅适用于 MySQL，也适用于其他类型的数据库管理系统。

19.1 优化表的数据类型

表需要使用何种数据类型，是需要根据应用来判断的。虽然应用设计的时候需要考虑字段的长度留有一定的冗余，但是不推荐让很多字段都留有大量的冗余，这样即浪费磁盘存储空间，同时在应用程序操作时也浪费物理内存。

在 MySQL 中，可以使用函数 PROCEDURE ANALYSE()对当前应用的表进行分析，该函数可以对数据表中列的数据类型提出优化建议，用户可以根据应用的实际情况酌情考虑是否实施优化。

以下是函数 PROCEDURE ANALYSE()的使用方法：

```
SELECT * FROM tbl_name PROCEDURE ANALYSE();
SELECT * FROM tbl_name PROCEDURE ANALYSE(16,256);
```

输出的每一列信息都会对数据表中的列的数据类型提出优化建议。以上第二个语句告诉 PROCEDURE ANALYSE()不要为那些包含的值多于 16 个或者 256 字节的 ENUM 类型提出建议。如果没有这样的限制，输出信息可能很长；ENUM 定义通常很难阅读。

根据 PROCEDURE ANALYSE()函数的输出信息，用户可能会发现，一些表中的字段可以修改为效率更高的数据类型。如果决定改变某个字段的类型，则需要使用 ALTER TABLE 语句。

下面分析一下表 duck_cust 的数据类型是否需要优化。

（1）首先创建测试表 duck_cust，duck_cust 表中记录了客户的一些基本信息：

```
drop table duck_cust;
CREATE TABLE duck_cust(
cust_num MEDIUMINT  AUTO_INCREMENT, --客户编号
cust_title TINYINT, --客户标题号
```

```
    cust_last CHAR(20) NOT NULL,  --客户姓氏
    cust_first CHAR(15) NOT NULL, --客户名
    cust_suffix ENUM('Jr.', 'II', 'III','IV', 'V', 'M.D.','PhD'), --附加码
    cust_add1 CHAR(30) NOT NULL, --客户地址
    cust_add2 CHAR(10), --客户地址
    cust_city CHAR(18) NOT NULL, --客户所在城市
    cust_state CHAR(2) NOT NULL, --客户所在州
    cust_zip1 CHAR(5)NOT NULL, --客户邮编
    cust_zip2 CHAR(4), --客户邮编
    cust_duckname CHAR(25) NOT NULL, --客户名称
    cust_duckbday DATE, --客户生日
    PRIMARY KEY (cust_num)
    )TYPE=MyISAM;
```

（2）然后生成一些测试数据：

```
    INSERT INTO duck_cust VALUES(NULL, 1, 'Irishlord', 'Red', 'III', '1022 N.E. Sea of Rye',
'A207', 'Seacouver', 'WA', '98601', '3464', 'Netrek Rules', '1967:10:21');
    INSERT INTO duck_cust VALUES(NULL, 4, 'Thegreat', 'Vicki', 0, '2004 Singleton Dr.', 0,
'Freedom', 'KS', '67209', '4321', 'Frida Kahlo de Tomayo', '1948:03:21');
    INSERT INTO duck_cust VALUES(NULL, 9, 'Montgomery', 'Chantel', 0, '1567 Terra Cotta Way',
0,  'Chicago', 'IL', '89129', '4444', 'Bianca', '1971:07:29');
    INSERT INTO duck_cust VALUES(NULL, 7, 'Robert', 'David', 'Sr.', '20113 Open Road Highway',
'#6', 'Blacktop', 'AZ', '00606', '1952', 'Harley', '1949:08:00');
    INSERT INTO duck_cust VALUES(NULL, 5, 'Kazui', 'Wonko', 'PhD', '42 Cube Farm Lane',
'Gatehouse',  'Vlimpt', 'CA', '45362', 0, 'Fitzwhistle', '1961:12:04');
    INSERT INTO duck_cust VALUES(NULL, 6, 'Gashlycrumb', 'Karen', 0, '3113 Picket Fence Lane',
0, 'Fedora', 'VT', '41927', '5698', 'Tess D''urberville', '1948:08:19');
```

这时，查看一下表结构：

```
    desc duck_cust;
    +-------------------+-----------------------------------------+-------+------+---------+--------+
    | Field             | Type                                    | Null  | Key  | Default | Extra  |
    +-------------------+-----------------------------------------+-------+------+---------+--------+
    | cust_num          | mediumint(9)                            |       | MUL | 0       |        |
    | cust_title        | tinyint(4)                              | YES   |      | NULL    |        |
    | cust_last         | char(20)                                |       |      |         |        |
    | cust_first        | char(15)                                |       |      |         |        |
    | cust_suffix       | enum('Jr.','II','III','IV','V','M.D.','PhD') | YES   |      | NULL    |        |
    | cust_add1         | char(30)                                |       |      |         |        |
    | cust_add2         | char(10)                                | YES   |      | NULL    |        |
    | cust_city         | char(18)                                |       |      |         |        |
    | cust_state        | char(2)                                 |       |      |         |        |
    | cust_zip1         | char(5)                                 |       |      |         |        |
    | cust_zip2         | char(4)                                 | YES   |      | NULL |        |
    | cust_duckname     | char(25)                                |       |      |         |        |
```

```
| cust_duckbday | date                                  | YES  |      | NULL |        |
+-------------------+---------------------------------------+------+------+---------+--------+
```

（3）使用 PROCEDURE ANALYSE()函数确定要优化的列：

```
    mysql> SELECT * FROM duck_cust PROCEDURE ANALYSE()\G;
*************************** 1. row ***************************
             Field_name: sakila.duck_cust.cust_num
              Min_value: 1
              Max_value: 6
             Min_length: 1
             Max_length: 1
       Empties_or_zeros: 0
                  Nulls: 0
Avg_value_or_avg_length: 3.5000
                    Std: 1.7078
      Optimal_fieldtype: ENUM('1','2','3','4','5','6') NOT NULL
*************************** 2. row ***************************
…
…
```

从结果中，可以看到 test.duck_cust.cust_num 列的 Min_length、Max_length、Avg_value_or_avg_length，根据这些统计值,可以对列进行优化，例如，插入的数据最大长度和最小长度都是 1，所以，可以优化字段 cust_num 为 mediumint(2)；同时，上面的结果也给出了优化建议“Optimal_fieldtype: ENUM('1','2','3','4','5','6') NOT NULL”。

看到这个建议读者可能会觉得很奇怪，怎么给出了枚举类型？而不是我们预期的整型。因为这时分析的测试表记录数太少，使得 cust_name 的唯一值太少，因此函数觉得用枚举类型会更合理。如果是对一个大表进行分析，提出的建议会更准确。

根据给出的统计信息和优化建议，可以使用如下语句进行字段类型的更改：

```
mysql> alter table  duck_cust  modify cust_num mediumint(2);
Query OK, 6 rows affected (0.03 sec)
Records: 6  Duplicates: 0  Warnings: 0
```

19.2 通过拆分提高表的访问效率

这里所说的“拆分”，是指对数据表进行拆分。如果针对 MyISAM 类型的表进行，那么有两种拆分方法。

- 第一种方法是垂直拆分，即把主码和一些列放到一个表，然后把主码和另外的列放到另一个表中。

  如果一个表中某些列常用，而另外一些列不常用，则可以采用垂直拆分，另外垂直拆分可以使得数据行变小，一个数据页就能存放更多的数据，在查询时就会减少 I/O 次数。其缺点是需要管理冗余列，查询所有数据需要联合（JOIN）操作。

- 第二种方法是水平拆分，即根据一列或多列数据的值把数据行放到两个独立的表中。水平拆分通常在以下几种情况下使用：

  - 表很大，分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询速度。

- 表中的数据本来就有独立性，例如，表中分别记录各个地区的数据或不同时期的数据，特别是有些数据常用，而另外一些数据不常用。
- 需要把数据存放到多个介质上。

例如，移动电话的账单表就可以分成两个表或多个表。最近 3 个月的账单数据存在一个表中，3 个月前的历史账单存放在另外一个表中，超过 1 年的历史账单可以存储到单独的存储介质上，这种拆分是最常使用的水平拆分方法。

水平拆分会给应用增加复杂度，它通常在查询时需要多个表名，查询所有数据需要 UNION 操作。在许多数据库应用中，这种复杂性会超过它带来的优点，因为只要索引关键字不大，则在索引用于查询时，表中增加 2 至 3 倍数据量，查询时也就增加读一个索引层的磁盘次数，所以水平拆分要考虑数据量的增长速度，根据实际情况决定是否需要对表进行水平拆分。

19.3 逆规范化

数据库设计时要满足规范化这个道理大家都非常清楚，但是否数据的规范化程度越高越好呢？这还是由实际需求来决定。因为规范化越高，那么产生的关系就越多，关系过多的直接结果就是导致表之间的连接操作越频繁，而表之间的连接操作是性能较低的操作，直接影响到查询的速度，所以对于查询较多的应用就需要根据实际情况运用逆规范化对数据进行设计，通过逆规范化来提高查询的性能。

例如，移动电话的用户每月都会查询自己的账单，账单信息一般包含用户的名字和本月消费总金额，设想一下，如果用户的姓名和属性信息存放在一个表中，假设表名为 A，而用户的编号和他对应的账单信息存放在另外一张 B 表中，那么，用户每次查询自己的月账单时，数据库查询时都要进行表连接，因为账单表 B 中并不包含用户的名字，所以必须通过关联 A 表取过来，如果在数据库设计时考虑到这一点，就可以在 B 表增加一个冗余字段存放用户的名字，这样在查询账单时就不用再做表关联，可以使查询有更好的性能。

反规范的好处是降低连接操作的需求、降低外码和索引的数目，还可能减少表的数目，相应带来的问题是可能出现数据的完整性问题。加快查询速度，但会降低修改速度。因此决定做反规范时，一定要权衡利弊，仔细分析应用的数据存取需求和实际的性能特点，好的索引和其他方法经常能够解决性能问题，而不必采用反规范这种方法。

在进行反规范操作之前，要充分考虑数据的存取需求、常用表的大小、一些特殊的计算（例如合计）、数据的物理存储位置等。常用的反规范技术有增加冗余列、增加派生列、重新组表和分割表。

- 增加冗余列：指在多个表中具有相同的列，它常用来在查询时避免连接操作。
- 增加派生列：指增加的列来自其他表中的数据，由其他表中的数据经过计算生成。增加的派生列其作用是在查询时减少连接操作，避免使用集函数。
- 重新组表：指如果许多用户需要查看两个表连接出来的结果数据，则把这两个表重新组成一个表来减少连接而提高性能。
- 分割表：可以参见 19.2 小节的内容。

另外，逆规范技术需要维护数据的完整性。无论使用何种反规范技术，都需要一

定的管理来维护数据的完整性，常用的方法是批处理维护、应用逻辑和触发器。

- 批处理维护是指对复制列或派生列的修改积累一定的时间后，运行一批处理作业或存储过程对复制或派生列进行修改，这只能在对实时性要求不高的情况下使用。
- 数据的完整性也可由应用逻辑来实现，这就要求必须在同一事务中对所有涉及的表进行增、删、改操作。用应用逻辑来实现数据的完整性风险较大，因为同一逻辑必须在所有的应用中使用和维护，容易遗漏，特别是在需求变化时，不易于维护。
- 另一种方式就是使用触发器，对数据的任何修改立即触发对复制列或派生列的相应修改。触发器是实时的，而且相应的处理逻辑只在一个地方出现，易于维护。一般来说，是解决这类问题比较好的办法。

19.4 使用中间表提高统计查询速度

对于数据量较大的表，在其上进行统计查询通常会效率很低，并且还要考虑统计查询是否会对在线的应用产生负面影响。通常在这种情况下，使用中间表可以提高统计查询的效率，下面通过对 session 表的统计来介绍中间表的使用：

（1）session 表记录了客户每天的消费记录，表结构如下：

```
CREATE  TABLE session (
cust_id  varchar(10) ,     --客户编号
cust_amount DECIMAL(16,2), --客户消费金额
cust_date  DATE, --客户消费时间
cust_ip  varchar(20)  -客户 IP 地址
)
```

（2）由于每天都会产生大量的客户消费记录,所以 session 表的数据量很大,现在业务部门有一具体的需求：希望了解最近一周客户的消费总金额和近一周每天不同时段用户的消费总金额。针对这一需求我们通过 2 种方法来得出业务部门想要的结果。

方法 1：在 session 表上直接进行统计,得出想要的结果。

```
mysql> select sum(cust_amount) from session where cust_date>adddate(now(),-7);
+------------------+
| sum(cust_amount) |
+------------------+
|     161699200.64 |
+------------------+
1 row in set (3.95 sec)
```

方法 2：创建中间表 tmp_session，表结构和源表结构完全相同。

```
CREATE  TABLE  tmp_session (
cust_id  varchar(10) ,     --客户编号
cust_amount DECIMAL(16,2), --客户消费金额
cust_date  DATE, --客户消费时间
cust_ip  varchar(20)  -客户 IP 地址
) ;
```

转移要统计的数据到中间表,然后在中间表上进行统计，得出想要的结果。

```
mysql> insert into  tmp_session  select * from session  where cust_date>adddate(now(),-7);
Query OK, 1573328 rows affected (6.67 sec)
Records: 1573328  Duplicates: 0  Warnings: 0

mysql> select sum(cust_amount) from tmp_session;
+------------------+
| sum(cust_amount) |
+------------------+
|     161699200.64 |
+------------------+
1 row in set (0.73 sec)
```

从上面的 2 种实现方法上看,在中间表中做统计花费的时间很少(这里不计算转移数据花费的时间)，另外，针对业务部门想了解“近一周每天不同时段用户的消费总金额”这一需求，在中间表上给出统计结果更为合适,原因是源数据表(session 表) cust_date 字段没有索引并且源表的数据量较大，所以在按时间进行分时段统计时效率很低，这时可以在中间表上对 cust_date 字段创建单独的索引来提高统计查询的速度。

中间表在统计查询中经常会用到，其优点如下:

- 中间表复制源表部分数据，并且与源表相“隔离”，在中间表上做统计查询不会对在线应用产生负面影响。
- 中间表上可以灵活的添加索引或增加临时用的新字段,从而达到提高统计查询效率和辅助统计查询作用。

19.5 小结

本章介绍了对数据库对象的优化，数据库对象设计的好坏是一个数据库设计的基础，而且一旦数据库对象设计完毕并投入使用，将来再进行修改就比较麻烦，因此在进行数据库设计的时候一定要尽可能地考虑周到。

第20章　锁问题

锁是计算机协调多个进程或线程并发访问某一资源的机制。在数据库中，除传统的计算资源（如 CPU、RAM、I/O 等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说,锁对数据库而言显得尤其重要,也更加复杂。本章我们着重讨论 MySQL 锁机制的特点，常见的锁问题，以及解决 MySQL 锁问题的一些方法或建议。

20.1 MySQL 锁概述

相对其他数据库而言，MySQL的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。比如，MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；BDB存储引擎采用的是页面锁（page-level locking），但也支持表级锁；InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。

MySQL这3种锁的特性可大致归纳如下。

- 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。
- 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。
- 页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

从上述特点可见，很难笼统地说哪种锁更好，只能就具体应用的特点来说哪种锁更合适！仅从锁的角度来说：表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web 应用；而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，如一些在线事务处理（OLTP）系统。这一点在本书的“开发篇”介绍表类型的选择时，也曾提到过。下面几节我们重点介绍 MySQL 表锁和 InnoDB 行锁的问题，由于BDB 已经被 InnoDB 取代，即将成为历史，在此就不做进一步的讨论了。

20.2 MyISAM 表锁

MyISAM 存储引擎只支持表锁，这也是 MySQL 开始几个版本中唯一支持的锁类型。随着应用对事务完整性和并发性要求的不断提高，MySQL 才开始开发基于事务的存储引擎，后来慢慢出现了支持页锁的 BDB 存储引擎和支持行锁的 InnoDB 存储引擎（实际 InnoDB 是单独的一个公司，现在已经被 Oracle 公司收购）。但是 MyISAM 的表锁依然是使用最为广泛的锁类型。本节将详细介绍 MyISAM 表锁的使用。

20.2.1 查询表级锁争用情况

可以通过检查table_locks_waited和table_locks_immediate状态变量来分析系统上的表锁定争夺：

```
mysql> show status like 'table%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Table_locks_immediate | 2979  |
| Table_locks_waited    | 0     |
+-----------------------+-------+
2 rows in set (0.00 sec))
```

如果 Table_locks_waited 的值比较高，则说明存在着较严重的表级锁争用情况。

20.2.2 MySQL 表级锁的锁模式

MySQL 的表级锁有两种模式：表共享读锁（Table Read Lock）和表独占写锁（Table Write Lock）。锁模式的兼容性如表 20-1 所示。

表 20-1　　MySQL 中的表锁兼容性

请求锁模式 / 是否兼容 / 当前锁模式	None	读锁	写锁
读锁	是	是	否
写锁	是	否	否

可见，对 MyISAM 表的读操作，不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求；对 MyISAM 表的写操作，则会阻塞其他用户对同一表的读和写操作；MyISAM 表的读操作与写操作之间，以及写操作之间是串行的！根据如表 20-2 所示的例子可以知道，当一个线程获得对一个表的写锁后，只有持有锁的线程可以对表进行更新操作。其他线程的读、写操作都会等待，直到锁被释放为止。

表 20-2　　MyISAM 存储引擎的写阻塞读例子

session_1	session_2
获得表 film_text 的 WRITE 锁定 mysql> lock table film_text write; Query OK, 0 rows affected (0.00 sec)	

当前 session 对锁定表的查询、更新、插入操作都可以执行：
mysql> select film_id,title from film_text where film_id = 1001;
+---------+-------------+
| film_id | title |
+---------+-------------+
| 1001 | Update Test |
+---------+-------------+
1 row in set (0.00 sec)
 	
mysql> insert into film_text (film_id,title) values(1003,'Test');
Query OK, 1 row affected (0.00 sec)
 	
mysql> update film_text set title = 'Test' where film_id = 1001;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
	其他 session 对锁定表的查询被阻塞，需要等待锁被释放：
	mysql> select film_id,title from film_text where film_id = 1001;
	等待

释放锁： mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待

	Session2 获得锁，查询返回： mysql> select film_id, title from film_text where film_id = 1001; +---------+-------+ | film_id | title | +---------+-------+ | 1001 | Test | +---------+-------+ 1 row in set (57.59 sec)

20.2.3 如何加表锁

MyISAM 在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT 等）前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用 LOCK TABLE 命令给 MyISAM 表显式加锁。在本书的示例中，显式加锁基本上都是为了方便而已，并非必须如此。

给 MyISAM 表显示加锁，一般是为了在一定程度模拟事务操作，实现对某一时间点多个表的一致性读取。例如，有一个订单表 orders，其中记录有各订单的总金额 total，同时还有一个订单明细表 order_detail，其中记录有各订单每一产品的金额小计 subtotal，假设我们需要检查这两个表的金额合计是否相符，可能就需要执行如下两条 SQL：

```
Select sum(total) from orders;
Select sum(subtotal) from order_detail;
```

这时，如果不先给两个表加锁，就可能产生错误的结果，因为第一条语句执行过程中，order_detail 表可能已经发生了改变。因此，正确的方法应该是：

```
Lock tables orders read local, order_detail read local;
Select sum(total) from orders;
Select sum(subtotal) from order_detail;
Unlock tables;
```

要特别说明以下两点内容。

- 上面的例子在 LOCK TABLES 时加了“local”选项，其作用就是在满足 MyISAM 表并发插入条件的情况下，允许其他用户在表尾并发插入记录，有关 MyISAM 表的并发插入问题，在后面的章节中还会进一步介绍。
- 在用 LOCK TABLES 给表显式加表锁时，必须同时取得所有涉及到表的锁，并且 MySQL 不支持锁升级。也就是说，在执行 LOCK TABLES 后，只能访问显式加锁的这些表，不能访问未加锁的表；同时，如果加的是读锁，那么只能执行查询操作，而不能执行更新操作。其实，在自动加锁的情况下也基本如此，MyISAM 总是一次获得 SQL 语句所需要的全部锁。这也正是 MyISAM 表不会出现死锁（Deadlock Free）的原因。

在如表 20-3 所示的例子中，一个 session 使用 LOCK TABLE 命令给表 film_text 加了读锁，这个 session 可以查询锁定表中的记录，但更新或访问其他表都会提示错误；同时，另外一个 session 可以查询表中的记录，但更新就会出现锁等待。

表 20-3　　MyISAM 存储引擎的读阻塞写例子

session_1	session_2
获得表 film_text 的 READ 锁定 mysql> lock table film_text read;	

Query OK, 0 rows affected (0.00 sec)	
当前 session 可以查询该表记录 mysql> select film_id,title from film_text where film_id = 1001; +---------+-----------------+ | film_id | title | +---------+-----------------+ | 1001 | ACADEMY DINOSAUR | +---------+-----------------+ 1 row in set (0.00 sec)	其他 session 也可以查询该表的记录 mysql> select film_id,title from film_text where film_id = 1001; +---------+-----------------+ | film_id | title | +---------+-----------------+ | 1001 | ACADEMY DINOSAUR | +---------+-----------------+ 1 row in set (0.00 sec)
当前 session 不能查询没有锁定的表 mysql> select film_id,title from film where film_id = 1001; ERROR 1100 (HY000): Table 'film' was not locked with LOCK TABLES	其他 session 可以查询或者更新未锁定的表 mysql> select film_id,title from film where film_id = 1001; +---------+---------------+ | film_id | title | +---------+---------------+ | 1001 | update record | +---------+---------------+ 1 row in set (0.00 sec) mysql> update film set title = 'Test' where film_id = 1001; Query OK, 1 row affected (0.04 sec) Rows matched: 1 Changed: 1 Warnings: 0

当前 session 中插入或者更新锁定的表都会提示错误：
mysql> insert into film_text (film_id,title) values(1002,'Test');
ERROR 1099 (HY000): Table 'film_text' was locked with a READ lock and can't be updated
mysql> update film_text set title = 'Test' where film_id = 1001;
ERROR 1099 (HY000): Table 'film_text' was locked with a READ lock and can't be updated
	其他 session 更新锁定表会等待获得锁：
	mysql> update film_text set title = 'Test' where film_id = 1001;
	等待

释放锁 mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待
	Session 获得锁，更新操作完成： mysql> update film_text set title = 'Test' where film_id = 1001; Query OK, 1 row affected (1 min 0.71 sec) Rows matched: 1 Changed: 1 Warnings: 0

当使用 LOCK TABLES 时，不仅需要一次锁定用到的所有表，而且，同一个表在 SQL 语句中出现多少次，就要通过与 SQL 语句中相同的别名锁定多少次，否则也会出错！举例说明

如下。

（1）对 actor 表获得读锁：

```
mysql> lock table actor read;
Query OK, 0 rows affected (0.00 sec)
```

（2）但是通过别名访问会提示错误：

```
mysql> select a.first_name,a.last_name,b.first_name,b.last_name from actor a,actor b where a.first_name = b.first_name and a.first_name = 'Lisa' and a.last_name = 'Tom' and a.last_name <> b.last_name;
ERROR 1100 (HY000): Table 'a' was not locked with LOCK TABLES
```

（3）需要对别名分别锁定：

```
mysql> lock table actor as a read,actor as b read;
Query OK, 0 rows affected (0.00 sec)
```

（4）按照别名的查询可以正确执行：

```
mysql> select a.first_name,a.last_name,b.first_name,b.last_name from actor a,actor b where a.first_name = b.first_name and a.first_name = 'Lisa' and a.last_name = 'Tom' and a.last_name <> b.last_name;
+------------+-----------+------------+-----------+
| first_name | last_name | first_name | last_name |
+------------+-----------+------------+-----------+
| Lisa       | Tom       | LISA       | MONROE    |
+------------+-----------+------------+-----------+
1 row in set (0.00 sec)
```

20.2.4 并发插入（Concurrent Inserts）

上文提到过 MyISAM 表的读和写是串行的，但这是就总体而言的。在一定条件下，MyISAM 表也支持查询和插入操作的并发进行。

MyISAM存储引擎有一个系统变量concurrent_insert，专门用以控制其并发插入的行为，其值分别可以为0、1或2。

- 当concurrent_insert设置为0时，不允许并发插入。
- 当concurrent_insert设置为1时，如果MyISAM表中没有空洞（即表的中间没有被删除的行），MyISAM允许在一个进程读表的同时，另一个进程从表尾插入记录。这也是MySQL的默认设置。
- 当concurrent_insert设置为2时，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录。

在如表 20-4 所示的例子中，session_1 获得了一个表的 READ LOCAL 锁，该线程可以对表进行查询操作，但不能对表进行更新操作；其他的线程（session_2），虽然不能对表进行删除和更新操作，但却可以对该表进行并发插入操作，这里假设该表中间不存在空洞。

表 20-4　　MyISAM 存储引擎的读写（INSERT）并发例子

session_1	session_2

获得表 film_text 的 READ LOCAL 锁定 mysql> lock table film_text read local; Query OK, 0 rows affected (0.00 sec)	

当前 session 不能对锁定表进行更新或者插入操作：
mysql> insert into film_text (film_id,title) values(1002,'Test');
ERROR 1099 (HY000): Table 'film_text' was locked with a READ lock and can't be updated
mysql> update film_text set title = 'Test' where film_id = 1001;
ERROR 1099 (HY000): Table 'film_text' was locked with a READ lock and can't be updated
	其他 session 可以进行插入操作，但是更新会等待：
	mysql> insert into film_text (film_id,title) values(1002,'Test');
	Query OK, 1 row affected (0.00 sec)
	mysql> update film_text set title = 'Update Test' where film_id = 1001;
	等待

当前 session 不能访问其他 session 插入的记录： mysql> select film_id,title from film_text where film_id = 1002; Empty set (0.00 sec)	
释放锁： mysql> unlock tables; Query OK, 0 rows affected (0.00 sec)	等待
当前 session 解锁后可以获得其他 session 插入的记录： mysql> select film_id,title from film_text where film_id = 1002; +---------+-------+ | film_id | title | +---------+-------+ | 1002 | Test | +---------+-------+ 1 row in set (0.00 sec)	Session2 获得锁，更新操作完成： mysql> update film_text set title = 'Update Test' where film_id = 1001; Query OK, 1 row affected (1 min 17.75 sec) Rows matched: 1 Changed: 1 Warnings: 0

可以利用MyISAM存储引擎的并发插入特性，来解决应用中对同一表查询和插入的锁争用。例如，将concurrent_insert系统变量设为2，总是允许并发插入；同时，通过定期在系统空闲时段执行OPTIMIZE TABLE语句来整理空间碎片，收回因删除记录而产生的中间空洞。有关OPTIMIZE TABLE语句的详细介绍，可以参见第18章中“两个简单实用的优化方法”一节的内容。

20.2.5 MyISAM 的锁调度

前面讲过，MyISAM 存储引擎的读锁和写锁是互斥的，读写操作是串行的。那么，一个进程请求某个 MyISAM 表的读锁，同时另一个进程也请求同一表的写锁，MySQL 如何处理呢？答案是写进程先获得锁。不仅如此，即使读请求先到锁等待队列，写请求后到，写锁也会插到读锁请求之前！这是因为 MySQL 认为写请求一般比读请求要重要。这也正是 MyISAM 表不太适合于有大量更新操作和查询操作应用的原因，因为，大量的更新操作会造成查询操作

很难获得读锁，从而可能永远阻塞。这种情况有时可能会变得非常糟糕！幸好我们可以通过一些设置来调节 MyISAM 的调度行为。

- 通过指定启动参数low-priority-updates，使MyISAM引擎默认给予读请求以优先的权利。
- 通过执行命令SET LOW_PRIORITY_UPDATES=1，使该连接发出的更新请求优先级降低。
- 通过指定INSERT、UPDATE、DELETE语句的LOW_PRIORITY属性，降低该语句的优先级。

虽然上面 3 种方法都是要么更新优先，要么查询优先的方法，但还是可以用其来解决查询相对重要的应用（如用户登录系统）中，读锁等待严重的问题。

另外，MySQL也提供了一种折中的办法来调节读写冲突，即给系统参数max_write_lock_count设置一个合适的值，当一个表的读锁达到这个值后，MySQL就暂时将写请求的优先级降低，给读进程一定获得锁的机会。

上面已经讨论了写优先调度机制带来的问题和解决办法。这里还要强调一点：一些需要长时间运行的查询操作，也会使写进程“饿死”！因此，应用中应尽量避免出现长时间运行的查询操作，不要总想用一条SELECT语句来解决问题，因为这种看似巧妙的SQL语句，往往比较复杂，执行时间较长，在可能的情况下可以通过使用中间表等措施对SQL语句做一定的“分解”，使每一步查询都能在较短时间完成，从而减少锁冲突。如果复杂查询不可避免，应尽量安排在数据库空闲时段执行，比如一些定期统计可以安排在夜间执行。

20.3 InnoDB 锁问题

InnoDB 与 MyISAM 的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁。行级锁与表级锁本来就有许多不同之处，另外，事务的引入也带来了一些新问题。下面我们先介绍一点背景知识，然后详细讨论 InnoDB 的锁问题。

20.3.1 背景知识

1. 事务（Transaction）及其 ACID 属性

事务是由一组 SQL 语句组成的逻辑处理单元，事务具有以下 4 个属性，通常简称为事务的 ACID 属性。

- 原子性（Atomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
- 一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（如 B 树索引或双向链表）也都必须是正确的。
- 隔离性（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。
- 持久性（Durable）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

银行转帐就是事务的一个典型例子。

2. 并发事务处理带来的问题

相对于串行处理来说，并发事务处理能大大增加数据库资源的利用率，提高数据库系统的事务吞吐量，从而可以支持更多的用户。但并发事务处理也会带来一些问题，主要包括以下几种情况。

- 更新丢失（Lost Update）：当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题——最后的更新覆盖了由其他事务所做的更新。例如，两个编辑人员制作了同一文档的电子副本。每个编辑人员独立地更改其副本，然后保存更改后的副本，这样就覆盖了原始文档。最后保存其更改副本的编辑人员覆盖另一个编辑人员所做的更改。如果在一个编辑人员完成并提交事务之前，另一个编辑人员不能访问同一文件，则可避免此问题。
- 脏读（Dirty Reads）：一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做"脏读"。
- 不可重复读（Non-Repeatable Reads）：一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现其读出的数据已经发生了改变、或某些记录已经被删除了！这种现象就叫做“不可重复读”。
- 幻读（Phantom Reads）：一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读”。

3. 事务隔离级别

在上面讲到的并发事务处理带来的问题中，“更新丢失”通常是应该完全避免的。但防止更新丢失，并不能单靠数据库事务控制器来解决，需要应用程序对要更新的数据加必要的锁来解决，因此，防止更新丢失应该是应用的责任。

“脏读”、“不可重复读”和“幻读”，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。数据库实现事务隔离的方式，基本上可分为以下两种。

- 一种是在读取数据前，对其加锁，阻止其他事务对数据进行修改。
- 另一种是不用加任何锁，通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot），并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好象是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据多版本并发控制（MultiVersion Concurrency Control，简称 MVCC 或 MCC），也经常称为多版本数据库。

数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上“串行化”进行，这显然与“并发”是矛盾的。同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。

为了解决“隔离”与“并发”的矛盾，ISO/ANSI SQL92 定义了 4 个事务隔离级别，每个级别的隔离程度不同，允许出现的副作用也不同，应用可以根据自己的业务逻辑要求，通过选择不同的隔离级别来平衡“隔离”与“并发”的矛盾。表 20-5 很好地概括了这 4 个隔离级别的特性。

表 20-5　　4 种隔离级别比较

读数据一致性及允许的并发副作用 / 隔离级别	读数据一致性	脏读	不可重复读	幻读
未提交读（Read uncommitted）	最低级别，只能保证不读取物理上损坏的数据	是	是	是
已提交度（Read committed）	语句级	否	是	是
可重复读（Repeatable read）	事务级	否	否	是
可序列化（Serializable）	最高级别，事务级	否	否	否

最后要说明的是：各具体数据库并不一定完全实现了上述 4 个隔离级别，例如，Oracle 只提供 Read committed 和 Serializable 两个标准隔离级别，另外还提供自己定义的 Read only 隔离级别；SQL Server 除支持上述 ISO/ANSI SQL92 定义的 4 个隔离级别外，还支持一个叫做“快照”的隔离级别，但严格来说它是一个用 MVCC 实现的 Serializable 隔离级别。MySQL 支持全部 4 个隔离级别，但在具体实现时，有一些特点，比如在一些隔离级别下是采用 MVCC 一致性读，但某些情况下又不是，这些内容在后面的章节中将会做进一步介绍。

20.3.2 获取 InnoDB 行锁争用情况

可以通过检查 InnoDB_row_lock 状态变量来分析系统上的行锁的争夺情况：

```
mysql> show status like 'innodb_row_lock%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| InnoDB_row_lock_current_waits | 0     |
| InnoDB_row_lock_time          | 0     |
| InnoDB_row_lock_time_avg      | 0     |
| InnoDB_row_lock_time_max      | 0     |
| InnoDB_row_lock_waits         | 0     |
+-------------------------------+-------+
5 rows in set (0.01 sec)
```

如果发现锁争用比较严重，如 InnoDB_row_lock_waits 和 InnoDB_row_lock_time_avg 的值比较高，还可以通过设置 InnoDB Monitors 来进一步观察发生锁冲突的表、数据行等，并分析锁争用的原因。

具体方法如下：

```
mysql> CREATE TABLE innodb_monitor(a INT) ENGINE=INNODB;
Query OK, 0 rows affected (0.14 sec)
```

然后就可以用下面的语句来进行查看：

```
mysql> Show innodb status\G;
```

```
*************************** 1. row ***************************
  Type: InnoDB
  Name:
Status:
…
…
------------
TRANSACTIONS
------------
Trx id counter 0 117472192
Purge done for trx's n:o < 0 117472190 undo n:o < 0 0
History list length 17
Total number of lock structs in row lock hash table 0
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 117472185, not started, process no 11052, OS thread id 1158191456
MySQL thread id 200610, query id 291197 localhost root
---TRANSACTION 0 117472183, not started, process no 11052, OS thread id 1158723936
MySQL thread id 199285, query id 291199 localhost root
Show innodb status
…
```

监视器可以通过发出下列语句来停止查看：

```
mysql> DROP TABLE innodb_monitor;
Query OK, 0 rows affected (0.05 sec)
```

设置监视器后，在 SHOW INNODB STATUS 的显示内容中，会有详细的当前锁等待的信息，包括表名、锁类型、锁定记录的情况等，便于进行进一步的分析和问题的确定。打开监视器以后，默认情况下每 15 秒会向日志中记录监控的内容，如果长时间打开会导致.err 文件变得非常的巨大，所以用户在确认问题原因之后，要记得删除监控表以关闭监视器，或者通过使用“--console”选项来启动服务器以关闭写日志文件。

20.3.3 InnoDB 的行锁模式及加锁方法

InnoDB 实现了以下两种类型的行锁。

- 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
- 排他锁（X）：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

- 意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。
- 意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。

上述锁模式的兼容情况具体如表 20-6 所示。

表 20-6　　InnoDB 行锁模式兼容性列表

请求锁模式 是否兼容 当前锁模式	X	IX	S	IS
X	冲突	冲突	冲突	冲突
IX	冲突	兼容	冲突	兼容
S	冲突	冲突	兼容	兼容
IS	冲突	兼容	兼容	兼容

如果一个事务请求的锁模式与当前的锁兼容，InnoDB 就将请求的锁授予该事务；反之，如果两者不兼容，该事务就要等待锁释放。

意向锁是 InnoDB 自动加的，不需用户干预。对于 UPDATE、DELETE 和 INSERT 语句，InnoDB 会自动给涉及数据集加排他锁（X）；对于普通 SELECT 语句，InnoDB 不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。

- 共享锁（S）：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE。
- 排他锁（X)：SELECT * FROM table_name WHERE ... FOR UPDATE。

用 SELECT ... IN SHARE MODE 获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行 UPDATE 或者 DELETE 操作。但是如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用 SELECT... FOR UPDATE 方式获得排他锁。

在如表 20-7 所示的例子中，使用了 SELECT ... IN SHARE MODE 加锁后再更新记录，看看会出现什么情况，其中 actor 表的 actor_id 字段为主键。

表 20-7　　InnoDB 存储引擎的共享锁例子

session_1	session_2
mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select actor_id,first_name,last_name from actor where actor_id = 178; +----------+-----------+-----------+ | actor_id | first_name | last_name | +----------+-----------+-----------+ | 178 | LISA | MONROE | +----------+-----------+-----------+ 1 row in set (0.00 sec)	mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select actor_id,first_name,last_name from actor where actor_id = 178; +----------+-----------+-----------+ | actor_id | first_name | last_name | +----------+-----------+-----------+ | 178 | LISA | MONROE | +----------+-----------+-----------+ 1 row in set (0.00 sec)
当前 session 对 actor_id=178 的记录加 share mode 的共享锁： mysql> select actor_id,first_name,last_name from actor where actor_id = 178 lock in share mode; +----------+-----------+-----------+ | actor_id | first_name | last_name | +----------+-----------+-----------+	

| 178 | LISA | MONROE | +----------+-----------+-----------+ 1 row in set (0.01 sec)	
	其他 session 仍然可以查询记录，并也可以对该记录加 share mode 的共享锁： mysql> select actor_id,first_name,last_name from actor where actor_id = 178 lock in share mode; +----------+-----------+-----------+ | actor_id | first_name | last_name | +----------+-----------+-----------+ | 178 | LISA | MONROE | +----------+-----------+-----------+ 1 row in set (0.01 sec)
当前 session 对锁定的记录进行更新操作，等待锁： mysql> update actor set last_name = 'MONROE T' where actor_id = 178; 等待	
	其他 session 也对该记录进行更新操作，则会导致死锁退出： mysql> update actor set last_name = 'MONROE T' where actor_id = 178; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
获得锁后，可以成功更新： mysql> update actor set last_name = 'MONROE T' where actor_id = 178; Query OK, 1 row affected (17.67 sec) Rows matched: 1 Changed: 1 Warnings: 0	

当使用SELECT...FOR UPDATE加锁后再更新记录，出现如表20-8所示的情况。

表20-8 InnoDB存储引擎的排他锁例子

session_1	session_2
mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select actor_id,first_name,last_name from actor where actor_id = 178; +----------+-----------+-----------+ | actor_id | first_name | last_name | +----------+-----------+-----------+ | 178 | LISA | MONROE | +----------+-----------+-----------+ 1 row in set (0.00 sec)	mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select actor_id,first_name,last_name from actor where actor_id = 178; +----------+-----------+-----------+ | actor_id | first_name | last_name | +----------+-----------+-----------+ | 178 | LISA | MONROE | +----------+-----------+-----------+ 1 row in set (0.00 sec)

当前 session 对 actor_id=178 的记录加 for update 的共享锁： mysql> select actor_id,first_name,last_name from actor where actor_id = 178 for update; +----------+------------+-----------+ | actor_id | first_name | last_name | +----------+------------+-----------+ | 178 | LISA | MONROE | +----------+------------+-----------+ 1 row in set (0.00 sec)	
	其他 session 可以查询该记录，但是不能对该记录加共享锁，会等待获得锁： mysql> select actor_id,first_name,last_name from actor where actor_id = 178; +----------+------------+-----------+ | actor_id | first_name | last_name | +----------+------------+-----------+ | 178 | LISA | MONROE | +----------+------------+-----------+ 1 row in set (0.00 sec) mysql> select actor_id,first_name,last_name from actor where actor_id = 178 for update; 等待
当前 session 可以对锁定的记录进行更新操作，更新后释放锁： mysql> update actor set last_name = 'MONROE T' where actor_id = 178; Query OK, 1 row affected (0.00 sec) Rows matched: 1 Changed: 1 Warnings: 0 mysql> commit; Query OK, 0 rows affected (0.01 sec)	
	其他 session 获得锁，得到其他 session 提交的记录： mysql> select actor_id,first_name,last_name from actor where actor_id = 178 for update; +----------+------------+-----------+ | actor_id | first_name | last_name | +----------+------------+-----------+ | 178 | LISA | MONROE T | +----------+------------+-----------+ 1 row in set (9.59 sec)

20.3.4 InnoDB 行锁实现方式

InnoDB 行锁是通过给索引上的索引项加锁来实现的，这一点 MySQL 与 Oracle 不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB 这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁！
在实际应用中，要特别注意 InnoDB 行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。下面通过一些实际例子来加以说明。
（1）在不通过索引条件查询的时候，InnoDB 确实使用的是表锁，而不是行锁。
在如表 20-9 所示的例子中，开始 tab_no_index 表没有索引：

```
mysql> create table tab_no_index(id int,name varchar(10)) engine=innodb;
Query OK, 0 rows affected (0.15 sec)

mysql> insert into tab_no_index values(1,'1'),(2,'2'),(3,'3'),(4,'4');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

表 20-9　　InnoDB 存储引擎的表在不使用索引时使用表锁例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_no_index where id = 1 ; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_no_index where id = 2 ; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)
mysql> select * from tab_no_index where id = 1 for update; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	
	mysql> select * from tab_no_index where id = 2 for update; 等待

在如表 20-9 所示的例子中，看起来 session_1 只给一行加了排他锁，但 session_2 在请求其他行的排他锁时，却出现了锁等待！原因就是在没有索引的情况下，InnoDB 只能使用表锁。当我们给其增加一个索引后，InnoDB 就只锁定了符合条件的行，如表 20-10 所示。
创建 tab_with_index 表，id 字段有普通索引：

```
mysql> create table tab_with_index(id int,name varchar(10)) engine=innodb;
```

```
Query OK, 0 rows affected (0.15 sec)
mysql> alter table tab_with_index add index id(id);
Query OK, 4 rows affected (0.24 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

表 20-10　　InnoDB 存储引擎的表在使用索引时使用行锁例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_with_index where id = 1 ; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec) mysql> select * from tab_with_index where id = 2 ; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)
mysql> select * from tab_with_index where id = 1 for update; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	
	mysql> select * from tab_with_index where id = 2 for update; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)

（2）由于 MySQL 的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。应用设计的时候要注意这一点。

在如表 20-11 所示的例子中，表 tab_with_index 的 id 字段有索引，name 字段没有索引：

```
mysql> alter table tab_with_index drop index name;
Query OK, 4 rows affected (0.22 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> insert into tab_with_index  values(1,'4');
Query OK, 1 row affected (0.00 sec)

mysql> select * from tab_with_index where id = 1;
```

```
+------+------+
| id   | name |
+------+------+
| 1    | 1    |
| 1    | 4    |
+------+------+
2 rows in set (0.00 sec)
```

表 20-11　　InnoDB 存储引擎使用相同索引键的阻塞例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)
mysql> select * from tab_with_index where id = 1 and name = '1' for update; +------+------+ | id | name | +------+------+ | 1 | 1 | +------+------+ 1 row in set (0.00 sec)	
	虽然 session_2 访问的是和 session_1 不同的记录，但是因为使用了相同的索引，所以需要等待锁： mysql> select * from tab_with_index where id = 1 and name = '4' for update; 等待

（3）当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行，另外，不论是使用主键索引、唯一索引或普通索引，InnoDB 都会使用行锁来对数据加锁。

在如表 20-12 所示的例子中，表 tab_with_index 的 id 字段有主键索引，name 字段有普通索引：

```
mysql> alter table tab_with_index add index name(name);
Query OK, 5 rows affected (0.23 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

表 20-12　　InnoDB 存储引擎的表使用不同索引的阻塞例子

• session_1	• session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)
mysql> select * from tab_with_index where id = 1 for update; +------+------+ | id | name | +------+------+ | 1 | 1 | | 1 | 4 | +------+------+	

2 rows in set (0.00 sec)	
	Session_2 使用 name 的索引访问记录，因为记录没有被索引，所以可以获得锁： mysql> select * from tab_with_index where name = '2' for update; +------+------+ | id | name | +------+------+ | 2 | 2 | +------+------+ 1 row in set (0.00 sec)
	由于访问的记录已经被 session_1 锁定，所以等待获得锁。： mysql> select * from tab_with_index where name = '4' for update;

（4）即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查 SQL 的执行计划，以确认是否真正使用了索引。关于 MySQL 在什么情况下不使用索引的详细讨论，参见本章“索引问题”一节的介绍。

在下面的例子中，检索值的数据类型与索引字段不同，虽然 MySQL 能够进行数据类型转换，但却不会使用索引，从而导致 InnoDB 使用表锁。通过用 explain 检查两条 SQL 的执行计划，我们可以清楚地看到了这一点。

例子中 tab_with_index 表的 name 字段有索引，但是 name 字段是 varchar 类型的，如果 where 条件中不是和 varchar 类型进行比较，则会对 name 进行类型转换，而执行的全表扫描。

```
mysql> alter table tab_no_index add index name(name);
Query OK, 4 rows affected (8.06 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> explain select * from tab_with_index where name = 1 \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tab_with_index
         type: ALL
possible_keys: name
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
        Extra: Using where
1 row in set (0.00 sec)
```

```
mysql> explain select * from tab_with_index where name = '1' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: tab_with_index
         type: ref
possible_keys: name
          key: name
      key_len: 23
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

20.3.5 间隙锁（Next-Key 锁）

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB 也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key 锁）。举例来说，假如 emp 表中只有 101 条记录，其 empid 的值分别是 1,2,...,100,101，下面的 SQL：

```
Select * from  emp where empid > 100 for update;
```

是一个范围条件的检索，InnoDB 不仅会对符合条件的 empid 值为 101 的记录加锁，也会对 empid 大于 101（这些记录并不存在）的“间隙”加锁。

InnoDB 使用间隙锁的目的，一方面是为了防止幻读，以满足相关隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了 empid 大于 100 的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；另外一方面，是为了满足其恢复和复制的需要。有关其恢复和复制对锁机制的影响，以及不同隔离级别下 InnoDB 使用间隙锁的情况，在后续的章节中会做进一步介绍。

很显然，在使用范围条件检索并锁定记录时，InnoDB 这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。因此，在实际应用开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

还要特别说明的是，InnoDB 除了通过范围条件加锁时使用间隙锁外，如果使用相等条件请求给一个不存在的记录加锁，InnoDB 也会使用间隙锁！

在如表 20-13 所示的例子中，假如 emp 表中只有 101 条记录，其 empid 的值分别是 1,2,.....,100,101。

表 20-13　　InnoDB 存储引擎的间隙锁阻塞例子

session_1	session_2
mysql> select @@tx_isolation; +-----------------+ | @@tx_isolation | +-----------------+ | REPEATABLE-READ |	mysql> select @@tx_isolation; +-----------------+ | @@tx_isolation | +-----------------+ | REPEATABLE-READ |

+----------------+ 1 row in set (0.00 sec) mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec)	+----------------+ 1 row in set (0.00 sec) mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec)
当前 session 对不存在的记录加 for update 的锁： mysql> select * from emp where empid = 102 for update; Empty set (0.00 sec)	
	这时，如果其他 session 插入 empid 为 201 的记录（注意：这条记录并不存在），也会出现锁等待： mysql>insert into emp(empid,...) values(201,...); 阻塞等待
Session_1 执行 rollback： mysql> rollback; Query OK, 0 rows affected (13.04 sec)	
	由于其他 session_1 回退后释放了 Next-Key 锁，当前 session 可以获得锁并成功插入记录： mysql>insert into emp(empid,...) values(201,...); Query OK, 1 row affected (13.35 sec)

20.3.6 恢复和复制的需要，对 InnoDB 锁机制的影响

MySQL 通过 BINLOG 录执行成功的 INSERT、UPDATE、DELETE 等更新数据的 SQL 语句，并由此实现 MySQL 数据库的恢复和主从复制（可以参见本书“管理篇”的介绍）。MySQL 的恢复机制（复制其实就是在 Slave Mysql 不断做基于 BINLOG 的恢复）有以下特点。

- 一是 MySQL 的恢复是 SQL 语句级的，也就是重新执行 BINLOG 中的 SQL 语句。这与 Oracle 数据库不同，Oracle 是基于数据库文件块的。
- 二是 MySQL 的 Binlog 是按照事务提交的先后顺序记录的，恢复也是按这个顺序进行的。这点也与 Oralce 不同，Oracle 是按照系统更新号（System Change Number，SCN）来恢复数据的，每个事务开始时，Oracle 都会分配一个全局唯一的 SCN，SCN 的顺序与事务开始的时间顺序是一致的。

从上面两点可知，MySQL 的恢复机制要求：在一个事务未提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读，这已经超过了 ISO/ANSI SQL92“可重复读”隔离级别的要求，实际上是要求事务要串行化。这也是许多情况下，InnoDB 要用到间隙锁的原因，比如在用范围条件更新记录时，无论在 Read Commited 或是 Repeatable Read 隔离级别下，InnoDB 都要使用间隙锁，但这并不是隔离级别要求的，有关 InnoDB 在不同隔离级别下加锁的差异在下一小节还会介绍。

另外，对于“insert into target_tab select * from source_tab where ...”和“create table new_tab ...select ... From source_tab where ...(CTAS)”这种 SQL 语句，用户并没有对 source_tab 做任何更新操作，但 MySQL 对这种 SQL 语句做了特别处理。先来看如表 20-14

的例子。

表 20-14　　CTAS 操作给原表加锁例子

session_1	session_2
mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select * from target_tab; Empty set (0.00 sec) mysql> select * from source_tab where name = '1'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 1 | 1 | | 5 | 1 | 1 | | 6 | 1 | 1 | | 7 | 1 | 1 | | 8 | 1 | 1 | +----+------+----+ 5 rows in set (0.00 sec)	mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select * from target_tab; Empty set (0.00 sec) mysql> select * from source_tab where name = '1'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 1 | 1 | | 5 | 1 | 1 | | 6 | 1 | 1 | | 7 | 1 | 1 | | 8 | 1 | 1 | +----+------+----+ 5 rows in set (0.00 sec)
mysql> insert into target_tab select d1,name from source_tab where name = '1'; Query OK, 5 rows affected (0.00 sec) Records: 5　Duplicates: 0　Warnings: 0	
	mysql> update source_tab set name = '1' where name = '8'; 等待
commit;	
	返回结果 commit;

在上面的例子中，只是简单地读 source_tab 表的数据，相当于执行一个普通的 SELECT 语句，用一致性读就可以了。ORACLE 正是这么做的，它通过 MVCC 技术实现的多版本数据来实现一致性读，不需要给 source_tab 加任何锁。我们知道 InnoDB 也实现了多版本数据，对普通的 SELECT 一致性读，也不需要加任何锁；但这里 InnoDB 却给 source_tab 加了共享锁，并没有使用多版本数据一致性读技术！

MySQL 为什么要这么做呢？其原因还是为了保证恢复和复制的正确性。因为不加锁的话，如果在上述语句执行过程中，其他事务对 source_tab 做了更新操作，就可能导致数据恢复的结果错误。为了演示这一点，我们再重复一下前面的例子，不同的是在 session_1 执行事务前，先将系统变量 innodb_locks_unsafe_for_binlog 的值设置为“on”（其默认值为 off），具体结果如表 20-15 所示。

表 20-15　　CTAS 操作不给原表加锁带来的安全问题例子

session_1	session_2
mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql>set innodb_locks_unsafe_for_binlog='on' Query OK, 0 rows affected (0.00 sec) mysql> select * from target_tab; Empty set (0.00 sec) mysql> select * from source_tab where name = '1'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 1 | 1 | | 5 | 1 | 1 | | 6 | 1 | 1 | | 7 | 1 | 1 | | 8 | 1 | 1 | +----+------+----+ 5 rows in set (0.00 sec)	mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec) mysql> select * from target_tab; Empty set (0.00 sec) mysql> select * from source_tab where name = '1'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 1 | 1 | | 5 | 1 | 1 | | 6 | 1 | 1 | | 7 | 1 | 1 | | 8 | 1 | 1 | +----+------+----+ 5 rows in set (0.00 sec)
mysql> insert into target_tab select d1,name from source_tab where name = '1'; Query OK, 5 rows affected (0.00 sec) Records: 5 Duplicates: 0 Warnings: 0	

	session_1 未提交，可以对 session_1 的 select 的记录进行更新操作。 mysql> update source_tab set name = '8' where name = '1'; Query OK, 5 rows affected (0.00 sec) Rows matched: 5 Changed: 5 Warnings: 0 mysql> select * from source_tab where name = '8'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 8 | 1 | | 5 | 8 | 1 | | 6 | 8 | 1 | | 7 | 8 | 1 | | 8 | 8 | 1 | +----+------+----+ 5 rows in set (0.00 sec)
	更新操作先提交 mysql> commit; Query OK, 0 rows affected (0.05 sec)
插入操作后提交 mysql> commit; Query OK, 0 rows affected (0.07 sec)	

此时查看数据，target_tab 中可以插入source_tab 更新前的结果，这符合应用逻辑： mysql> select * from source_tab where name = '8'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 8 | 1 | | 5 | 8 | 1 | | 6 | 8 | 1 | | 7 | 8 | 1 | | 8 | 8 | 1 | +----+------+----+ 5 rows in set (0.00 sec) mysql> select * from target_tab; +------+------+ | id | name | +------+------+ | 4 | 1.00 | | 5 | 1.00 | | 6 | 1.00 | | 7 | 1.00 | | 8 | 1.00 | +------+------+ 5 rows in set (0.00 sec)	mysql> select * from tt1 where name = '1'; Empty set (0.00 sec) mysql> select * from source_tab where name = '8'; +----+------+----+ | d1 | name | d2 | +----+------+----+ | 4 | 8 | 1 | | 5 | 8 | 1 | | 6 | 8 | 1 | | 7 | 8 | 1 | | 8 | 8 | 1 | +----+------+----+ 5 rows in set (0.00 sec) mysql> select * from target_tab; +------+------+ | id | name | +------+------+ | 4 | 1.00 | | 5 | 1.00 | | 6 | 1.00 | | 7 | 1.00 | | 8 | 1.00 | +------+------+ 5 rows in set (0.00 sec)

从上可见，设置系统变量innodb_locks_unsafe_for_binlog的值为“on”后，InnoDB不再对source_tab加锁，结果也符合应用逻辑，但是如果分析BINLOG的内容：

```
.....
SET TIMESTAMP=1169175130;
BEGIN;
# at 274
#070119 10:51:57 server id 1 end_log_pos 105 Query thread_id=1 exec_time=0
error_code=0
SET TIMESTAMP=1169175117;
update source_tab set name = '8' where name = '1';
# at 379
#070119 10:52:10 server id 1 end_log_pos 406 Xid = 5
COMMIT;
# at 406
#070119 10:52:14 server id 1 end_log_pos 474 Query thread_id=2 exec_time=0
error_code=0
SET TIMESTAMP=1169175134;
BEGIN;
```

```
# at 474
#070119 10:51:29 server id 1 end_log_pos 119 Query thread_id=2 exec_time=0
error_code=0
SET TIMESTAMP=1169175089;
insert into target_tab select d1,name from source_tab where name = '1';
# at 593
#070119 10:52:14 server id 1 end_log_pos 620 Xid = 7
COMMIT;
.....
```

可以发现，在BINLOG中，更新操作的位置在INSERT...SELECT之前，如果使用这个BINLOG进行数据库恢复，恢复的结果与实际的应用逻辑不符；如果进行复制，就会导致主从数据库不一致！

通过上面的例子，我们就不难理解为什么 MySQL 在处理“Insert into target_tab select * from source_tab where ...”和“create table new_tab ...select ... From source_tab where ...”时要给 source_tab 加锁，而不是使用对并发影响最小的多版本数据来实现一致性读。还要特别说明的是，如果上述语句的 SELECT 是范围条件，InnoDB 还会给源表加间隙锁（Next-Lock）。

因此，INSERT...SELECT...和CREATE TABLE...SELECT...语句，可能会阻止对源表的并发更新，造成对源表锁的等待。如果查询比较复杂的话，会造成严重的性能问题，我们在应用中应尽量避免使用。实际上，MySQL将这种SQL叫作不确定（non-deterministic）的SQL，不推荐使用。

如果应用中一定要用这种 SQL 来实现业务逻辑，又不希望对源表的并发更新产生影响，可以采取以下两种措施：

- 一是采取上面示例中的做法，将 innodb_locks_unsafe_for_binlog 的值设置为“on”，强制 MySQL 使用多版本数据一致性读。但付出的代价是可能无法用 binlog 正确地恢复或复制数据，因此，不推荐使用这种方式。
- 二是通过使用“select * from source_tab ... Into outfile”和“load data infile ...”语句组合来间接实现，采用这种方式 MySQL 不会给 source_tab 加锁。

20.3.7 InnoDB 在不同隔离级别下的一致性读及锁的差异

前面讲过，锁和多版本数据是 InnoDB 实现一致性读和 ISO/ANSI SQL92 隔离级别的手段，因此，在不同的隔离级别下，InnoDB 处理 SQL 时采用的一致性读策略和需要的锁是不同的。同时，数据恢复和复制机制的特点，也对一些 SQL 的一致性读策略和锁策略有很大影响。将这些特性归纳成如表 20-16 所示的内容，以便读者查阅。

表 20-16 InnoDB 存储引擎中不同 SQL 在不同隔离级别下锁比较

隔离级别 / 一致性读和锁 / SQL		Read Uncommited	Read Commited	Repeatable Read	Serializable
SQL	条件				
select	相等	None locks	Consisten read/None lock	Consisten read/None lock	Share locks

	范围	None locks	Consisten read/None lock	Consisten read/None lock	Share Next-Key
update	相等	exclusive locks	exclusive locks	exclusive locks	Exclusive locks
	范围	exclusive next-key	exclusive next-key	exclusive next-key	exclusive next-key
Insert	N/A	exclusive locks	exclusive locks	exclusive locks	exclusive locks
replace	无键冲突	exclusive locks	exclusive locks	exclusive locks	exclusive locks
	键冲突	exclusive next-key	exclusive next-key	exclusive next-key	exclusive next-key
delete	相等	exclusive locks	exclusive locks	exclusive locks	exclusive locks
	范围	exclusive next-key	exclusive next-key	exclusive next-key	exclusive next-key
Select ... from ... Lock in share mode	相等	Share locks	Share locks	Share locks	Share locks
	范围	Share locks	Share locks	Share Next-Key	Share Next-Key
Select * from ... For update	相等	exclusive locks	exclusive locks	exclusive locks	exclusive locks
	范围	exclusive locks	Share locks	exclusive next-key	exclusive next-key
Insert into ... Select ...（指源表锁）	innodb_locks_unsafe_for_binlog=off	Share Next-Key	Share Next-Key	Share Next-Key	Share Next-Key
	innodb_locks_unsafe_for_binlog=on	None locks	Consisten read/None lock	Consisten read/None lock	Share Next-Key
create table ... Select ...（指源表锁）	innodb_locks_unsafe_for_binlog=off	Share Next-Key	Share Next-Key	Share Next-Key	Share Next-Key
	innodb_locks_unsafe_for_binlog=on	None locks	Consisten read/None lock	Consisten read/None lock	Share Next-Key

从表 20-16 可以看出：对于许多 SQL，隔离级别越高，InnoDB 给记录集加的锁就越严格（尤其是使用范围条件的时候），产生锁冲突的可能性也就越高，从而对并发性事务处理性能的影响也就越大。因此，我们在应用中，应该尽量使用较低的隔离级别，以减少锁争用的机率。实际上，通过优化事务逻辑，大部分应用使用 Read Commited 隔离级别就足够了。对于一些确实需要更高隔离级别的事务，可以通过在程序中执行 SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ 或 SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE 动态改变隔离级别的方式满足需求。

20.3.8 什么时候使用表锁

对于 InnoDB 表，在绝大部分情况下都应该使用行级锁，因为事务和行锁往往是我们之所以选择 InnoDB 表的理由。但在个别特殊事务中，也可以考虑使用表级锁。

- 第一种情况是：事务需要更新大部分或全部数据，表又比较大，如果使用默认的行锁，不仅这个事务执行效率低，而且可能造成其他事务长时间锁等待和锁冲突，这种情况下可以考虑使用表锁来提高该事务的执行速度。
- 第二种情况是：事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚。这种情况也可以考虑一次性锁定事务涉及的表，从而避免死锁、减少数据库因事务回滚带来的开销。

当然，应用中这两种事务不能太多，否则，就应该考虑使用 MyISAM 表了。

在 InnoDB 下，使用表锁要注意以下两点。

（1）使用 LOCK TABLES 虽然可以给 InnoDB 加表级锁，但必须说明的是，表锁不是由 InnoDB 存储引擎层管理的，而是由其上一层——MySQL Server 负责的，仅当 autocommit=0、innodb_table_locks=1（默认设置）时，InnoDB 层才能知道 MySQL 加的表锁，MySQL Server 也才能感知 InnoDB 加的行锁，这种情况下，InnoDB 才能自动识别涉及表级锁的死锁；否则，InnoDB 将无法自动检测并处理这种死锁。有关死锁，下一小节还会继续讨论。

（2）在用 LOCK TABLES 对 InnoDB 表加锁时要注意，要将 AUTOCOMMIT 设为 0，否则 MySQL 不会给表加锁；事务结束前，不要用 UNLOCK TABLES 释放表锁，因为 UNLOCK TABLES 会隐含地提交事务；COMMIT 或 ROLLBACK 并不能释放用 LOCK TABLES 加的表级锁，必须用 UNLOCK TABLES 释放表锁。正确的方式见如下语句：

例如，如果需要写表 t1 并从表 t 读，可以按如下做：

```
SET AUTOCOMMIT=0;
LOCK TABLES t1 WRITE, t2 READ, ...;
[do something with tables t1 and t2 here];
COMMIT;
UNLOCK TABLES;
```

20.3.9 关于死锁

上文讲过，MyISAM 表锁是 deadlock free 的，这是因为 MyISAM 总是一次获得所需的全部锁，要么全部满足，要么等待，因此不会出现死锁。但在 InnoDB 中，除单个 SQL 组成的事务外，锁是逐步获得的，这就决定了在 InnoDB 中发生死锁是可能的。如表 20-17 所示的就是一个发生死锁的例子。

表 20-17　　InnoDB 存储引擎中的死锁例子

session_1	session_2

mysql> set autocommit = 0;
Query OK, 0 rows affected (0.00 sec)
 	
mysql> select * from table_1 where where id=1 for update;
...
做一些其他处理...
	mysql> set autocommit = 0;
	Query OK, 0 rows affected (0.00 sec)
	
	mysql> select * from table_2 where id=1 for update;
	...

select * from table_2 where id =1 for update; 因 session_2 已取得排他锁，等待	做一些其他处理...
	mysql> select * from table_1 where where id=1 for update; 死锁

在上面的例子中，两个事务都需要获得对方持有的排他锁才能继续完成事务，这种循环锁等待就是典型的死锁。

发生死锁后，InnoDB 一般都能自动检测到，并使一个事务释放锁并回退，另一个事务获得锁，继续完成事务。但在涉及外部锁，或涉及表锁的情况下，InnoDB 并不能完全自动检测到死锁，这需要通过设置锁等待超时参数 innodb_lock_wait_timeout 来解决。需要说明的是，这个参数并不是只用来解决死锁问题，在并发访问比较高的情况下，如果大量事务因无法立即获得所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖跨数据库。我们通过设置合适的锁等待超时阈值，可以避免这种情况发生。

通常来说，死锁都是应用设计的问题，通过调整业务流程、数据库对象设计、事务大小，以及访问数据库的 SQL 语句，绝大部分死锁都可以避免。下面就通过实例来介绍几种避免死锁的常用方法。

（1）在应用中，如果不同的程序会并发存取多个表，应尽量约定以相同的顺序来访问表，这样可以大大降低产生死锁的机会。在下面的例子中，由于两个 session 访问两个表的顺序不同，发生死锁的机会就非常高！但如果以相同的顺序来访问，死锁就可以避免。

表 20-18　　InnoDB 存储引擎中表顺序造成的死锁例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)
mysql> select first_name,last_name from actor where actor_id = 1 for update; +------------+-----------+ | first_name | last_name | +------------+-----------+ | PENELOPE | GUINESS | +------------+-----------+ 1 row in set (0.00 sec)	
	mysql> insert into country (country_id,country) values(110,'Test'); Query OK, 1 row affected (0.00 sec)
mysql> insert into country	

(country_id,country) values(110,'Test'); 等待	
	mysql> select first_name,last_name from actor where actor_id = 1 for update; +------------+-----------+ | first_name | last_name | +------------+-----------+ | PENELOPE | GUINESS | +------------+-----------+ 1 row in set (0.00 sec)
mysql> insert into country (country_id,country) values(110,'Test'); ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	

（2）在程序以批量方式处理数据的时候，如果事先对数据排序，保证每个线程按固定的顺序来处理记录，也可以大大降低出现死锁的可能。

表 20-19　　InnoDB 存储引擎中表数据操作顺序不一致造成的死锁例子

session_1	session_2
mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)	mysql> set autocommit=0; Query OK, 0 rows affected (0.00 sec)
mysql> select first_name,last_name from actor where actor_id = 1 for update; +------------+-----------+ | first_name | last_name | +------------+-----------+ | PENELOPE | GUINESS | +------------+-----------+ 1 row in set (0.00 sec)	
	mysql> select first_name,last_name from actor where actor_id = 3 for update; +------------+-----------+ | first_name | last_name | +------------+-----------+ | ED | CHASE | +------------+-----------+ 1 row in set (0.00 sec)
mysql> select first_name,last_name from actor where actor_id = 3 for update; 等待	
	mysql> select first_name,last_name from actor where actor_id = 1 for update; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

mysql> select first_name,last_name from actor where actor_id = 3 for update; +------------+-----------+ | first_name | last_name | +------------+-----------+ | ED | CHASE | +------------+-----------+ 1 row in set (4.71 sec)	

（3）在事务中，如果要更新记录，应该直接申请足够级别的锁，即排他锁，而不应先申请共享锁，更新时再申请排他锁，因为当用户申请排他锁时，其他事务可能又已经获得了相同记录的共享锁，从而造成锁冲突，甚至死锁。具体演示可参见 20.3.3 小节中的例子。

（4）前面讲过，在 REPEATABLE-READ 隔离级别下，如果两个线程同时对相同条件记录用 SELECT...FOR UPDATE 加排他锁，在没有符合该条件记录情况下，两个线程都会加锁成功。程序发现记录尚不存在，就试图插入一条新记录，如果两个线程都这么做，就会出现死锁。这种情况下，将隔离级别改成 READ COMMITTED，就可避免问题，如表 20-20 所示。

表 20-20　　InnoDB 存储引擎中隔离级别引起的死锁例子 1

session_1	session_2
mysql> select @@tx_isolation; +-----------------+ | @@tx_isolation | +-----------------+ | REPEATABLE-READ | +-----------------+ 1 row in set (0.00 sec) mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec)	mysql> select @@tx_isolation; +-----------------+ | @@tx_isolation | +-----------------+ | REPEATABLE-READ | +-----------------+ 1 row in set (0.00 sec) mysql> set autocommit = 0; Query OK, 0 rows affected (0.00 sec)
当前 session 对不存在的记录加 for update 的锁： mysql> select actor_id,first_name,last_name from actor where actor_id = 201 for update; Empty set (0.00 sec)	
	其他 session 也可以对不存在的记录加 for update 的锁： mysql> select actor_id,first_name,last_name from actor where actor_id = 201 for update; Empty set (0.00 sec)
因为其他 session 也对该记录加了锁，所以当前的插入会等待： mysql> insert into actor (actor_id, first_name, last_name) values(201,'Lisa','Tom'); 等待	
	因为其他 session 已经对记录进行了更新，这时候

	再插入记录就会提示死锁并退出： mysql> insert into actor (actor_id, first_name , last_name) values(201,'Lisa','Tom'); ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
由于其他 session 已经退出，当前 session 可以获得锁并成功插入记录： mysql> insert into actor (actor_id, first_name, last_name) values(201,'Lisa','Tom'); Query OK, 1 row affected (13.35 sec)	

（5）当隔离级别为 READ COMMITTED 时，如果两个线程都先执行 SELECT...FOR UPDATE，判断是否存在符合条件的记录，如果没有，就插入记录。此时，只有一个线程能插入成功，另一个线程会出现锁等待，当第 1 个线程提交后，第 2 个线程会因主键重出错，但虽然这个线程出错了，却会获得一个排他锁！这时如果有第 3 个线程又来申请排他锁，也会出现死锁。

对于这种情况，可以直接做插入操作，然后再捕获主键重异常，或者在遇到主键重错误时，总是执行 ROLLBACK 释放获得的排他锁，如表 20-21 所示。

表 20-21　　InnoDB 存储引擎中隔离级别引起的死锁例子 2

session_1	session_2	session_3
mysql> select @@tx_isolation; +----------------+ | @@tx_isolation | +----------------+ | READ-COMMITTED | +----------------+ 1 row in set (0.00 sec) mysql> set autocommit=0; Query OK, 0 rows affected (0.01 sec)	mysql> select @@tx_isolation; +----------------+ | @@tx_isolation | +----------------+ | READ-COMMITTED | +----------------+ 1 row in set (0.00 sec) mysql> set autocommit=0; Query OK, 0 rows affected (0.01 sec)	mysql> select @@tx_isolation; +----------------+ | @@tx_isolation | +----------------+ | READ-COMMITTED | +----------------+ 1 row in set (0.00 sec) mysql> set autocommit=0; Query OK, 0 rows affected (0.01 sec)
Session_1 获得 for update 的共享锁： mysql> select actor_id, first_name,last_name from actor where actor_id = 201 for update; Empty set (0.00 sec)	由于记录不存在，session_2 也可以获得 for update 的共享锁： mysql> select actor_id, first_name,last_name from actor where actor_id = 201 for update; Empty set (0.00 sec)	
Session_1 可以成功插入记录： mysql> insert into actor (actor_id,first_name,last_name) values(201,'Lisa','Tom'); Query OK, 1 row affected (0.00 sec)		

	Session_2 插入申请等待获得锁： mysql> insert into actor (actor_id,first_name,last_name) values(201,'Lisa','Tom'); 等待	
Session_1 成功提交： mysql> commit; Query OK, 0 rows affected (0.04 sec)		
	Session_2 获得锁，发现插入记录主键重，这个时候抛出了异常，但是并没有释放共享锁： mysql> insert into actor (actor_id,first_name,last_name) values(201,'Lisa','Tom'); ERROR 1062 (23000): Duplicate entry '201' for key 'PRIMARY'	
		Session_3 申请获得共享锁，因为 session_2 已经锁定该记录，所以 session_3 需要等待： mysql> select actor_id, first_name,last_name from actor where actor_id = 201 for update; 等待
	这个时候，如果 session_2 直接对记录进行更新操作，则会抛出死锁的异常： mysql> update actor set last_name='Lan' where actor_id = 201; ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction	
		Session_2 释放锁后，session_3 获得锁： mysql> select first_name, last_name from actor where actor_id = 201 for update; +------------+-----------+ | first_name | last_name | +------------+-----------+ | Lisa | Tom | +------------+-----------+

		1 row in set (31.12 sec)

尽管通过上面介绍的设计和 SQL 优化等措施，可以大大减少死锁，但死锁很难完全避免。因此，在程序设计中总是捕获并处理死锁异常是一个很好的编程习惯。

如果出现死锁，可以用 SHOW INNODB STATUS 命令来确定最后一个死锁产生的原因。返回结果中包括死锁相关事务的详细信息，如引发死锁的 SQL 语句，事务已经获得的锁，正在等待什么锁，以及被回滚的事务等。据此可以分析死锁产生的原因和改进措施。下面是一段 SHOW INNODB STATUS 输出的样例：

```
mysql> show innodb status \G
…….
------------------------
LATEST DETECTED DEADLOCK
------------------------
070710 14:05:16
*** (1) TRANSACTION:
TRANSACTION 0 117470078, ACTIVE 117 sec, process no 1468, OS thread id 1197328736 inserting
mysql tables in use 1, locked 1
LOCK WAIT 5 lock struct(s), heap size 1216
MySQL thread id 7521657, query id 673468054 localhost root update
insert into country (country_id,country) values(110,'Test')
……

*** (2) TRANSACTION:
TRANSACTION 0 117470079, ACTIVE 39 sec, process no 1468, OS thread id 1164048736 starting
index read, thread declared inside InnoDB 500
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1216, undo log entries 1
MySQL thread id 7521664, query id 673468058 localhost root statistics
select first_name,last_name from actor where actor_id = 1 for update
*** (2) HOLDS THE LOCK(S):
……

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
……

*** WE ROLL BACK TRANSACTION (1)
……
```

20.4 小结

本章重点介绍了 MySQL 中 MyISAM 表级锁和 InnoDB 行级锁的实现特点，并讨论了两种存储引擎经常遇到的锁问题和解决办法。

对于 MyISAM 的表锁，主要讨论了以下几点：

（1）共享读锁（S）之间是兼容的，但共享读锁（S）与排他写锁（X）之间，以及排他写锁（X）之间是互斥的，也就是说读和写是串行的。
（2）在一定条件下，MyISAM 允许查询和插入并发执行，我们可以利用这一点来解决应用中对同一表查询和插入的锁争用问题。
（3）MyISAM 默认的锁调度机制是写优先，这并不一定适合所有应用，用户可以通过设置 LOW_PRIORITY_UPDATES 参数，或在 INSERT、UPDATE、DELETE 语句中指定 LOW_PRIORITY 选项来调节读写锁的争用。
（4）由于表锁的锁定粒度大，读写之间又是串行的，因此，如果更新操作较多，MyISAM 表可能会出现严重的锁等待，可以考虑采用 InnoDB 表来减少锁冲突。

对于 InnoDB 表，本章主要讨论了以下几项内容。

- InnoDB 的行锁是基于锁引实现的，如果不通过索引访问数据，InnoDB 会使用表锁。
- 介绍了 InnoDB 间隙锁（Next-key)机制，以及 InnoDB 使用间隙锁的原因。
- 在不同的隔离级别下，InnoDB 的锁机制和一致性读策略不同。
- MySQL 的恢复和复制对 InnoDB 锁机制和一致性读策略也有较大影响。
- 锁冲突甚至死锁很难完全避免。

在了解 InnoDB 锁特性后，用户可以通过设计和 SQL 调整等措施减少锁冲突和死锁，包括：

- 尽量使用较低的隔离级别；
- 精心设计索引，并尽量使用索引访问数据，使加锁更精确，从而减少锁冲突的机会；
- 选择合理的事务大小，小事务发生锁冲突的几率也更小；
- 给记录集显示加锁时，最好一次性请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁；
- 不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大大减少死锁的机会；
- 尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响；
- 不要申请超过实际需要的锁级别；除非必须，查询时不要显示加锁；
- 对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能。

第21章　优化 MySQL Server

和大多数数据库一样，MySQL 也提供了很多参数来进行服务器的设置。当服务第一次启动的时候，所有的启动参数值都是系统默认的。这些参数在很多生产环境下并不能完全满足实际的应用需求，所以用户就需要按照实际情况进行调整。本章将分为两部分，第一部分介绍 MySQL Server 端参数的查看方法；第二部分将详细介绍 MySQL Server 中一些与性能相关的重要参数的优化设置方法。本章所有实例都在 Linux AS4 平台上的 MySQL 5.0.41 下测试通过。

21.1 查看 MySQL Server 参数

MySQL 服务启动后，我们可以用 SHOW VARIABLES 和 SHOW STATUS 命令查看 MySQL 的服务器静态参数值和动态运行状态信息。其中前者是在数据库启动后不会动态更改的值，比

如缓冲区大小、字符集、数据文件名称等；后者是数据库运行期间的动态变化的信息，比如锁等待、当前连接数等。以下分别对这两个命令的使用进行举例说明。

查看服务器参数值：

```
mysql> show variables;
+-------------------------------+----------------------------+
| Variable_name                 | Value                      |
+-------------------------------+----------------------------+
| auto_increment_increment      | 1                          |
| auto_increment_offset         | 1                          |
| automatic_sp_privileges       | ON                         |
| back_log                      | 50                         |
| basedir                       | /home/mysql/mysql_home/    |
| bdb_cache_parts               | 0                          |
| bdb_cache_size                | 0                          |
| bdb_home                      |
...
...
```

查看服务器运行状态值：

```
mysql> show status;
+----------------------------------+-----------+
| Variable_name                    | Value     |
+----------------------------------+-----------+
| Aborted_clients                  | 0         |
| Aborted_connects                 | 2         |
| Binlog_cache_disk_use            | 5         |
| Binlog_cache_use                 | 22        |
| Bytes_received                   | 163       |
| Bytes_sent                       | 10533     |
| Com_admin_commands               | 0         |
| Com_alter_db                     | 0         |
| Com_alter_event                  | 0         |
| Com_alter_table                  | 0         |
| Com_analyze                      | 0         |
| Com_backup_table                 | 0         |
| Com_begin                        | 0         |
| Com_change_db                    | 1         |
| Com_change_master                | 0         |
...
...
```

也可以在操作系统下直接查看数据库参数或数据库状态信息，具体命令如下：

```
[mysql@db3 bin]$ mysqladmin -uroot variables
+-------------------------------+-----------------------------------+
| Variable_name                 | Value                             |
+-------------------------------+-----------------------------------+
```

```
| auto_increment_increment        | 1                                |
| auto_increment_offset           | 1                                |
| automatic_sp_privileges         | ON                               |
| back_log                        | 50                               |
| basedir                         | /home/mysql/mysql_home/          |
| bdb_cache_parts                 | 0                                |
| bdb_cache_size                  | 0                                |
| bdb_home                        |                                  |
| bdb_log_buffer_size             | 0                                |
| bdb_logdir                      |

...

[mysql@db3 bin]$ mysqladmin -uroot status
Uptime: 327065  Threads: 1  Questions: 157054  Slow queries: 0  Opens: 0  Flush tables: 1
Open tables: 15  Queries per second avg: 0.480
```

MySQL 服务器的参数很多，如果需要了解某个参数的详细定义，可以使用以下命令：

```
[mysql@bj72 mysql]$ mysqld --verbose --help|more
mysqld  Ver 5.0.27-standard-log for pc-linux-gnu on i686 (MySQL Community Edition - Standard
(GPL))
Copyright (C) 2000 MySQL AB, by Monty and others
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL license

Starts the MySQL database server

Usage: mysqld [OPTIONS]

Default options are read from the following files in the given order:
/etc/my.cnf /home/mysql/mysql/my.cnf ~/.my.cnf /usr/local/mysql/etc/my.cnf
The following groups are read: mysqld server mysqld-5.0
The following options may be given as the first argument:
--print-defaults        Print the program argument list and exit
--no-defaults           Don't read default options from any options file
--defaults-file=#       Only read default options from the given file #
--defaults-extra-file=# Read this file after the global files are read

  -?, --help          Display this help and exit.
  --abort-slave-event-count=#
                      Option used by mysql-test for debugging and testing of
                      replication.
  --allow-suspicious-udfs
                      Allows use of UDFs consisting of only one symbol xxx()
```

```
                      without corresponding xxx_init() or xxx_deinit(). That
                      also means that one can load any function from any
                      library, for example exit() from libc.so
 --automatic-sp-privileges
                      Creating and dropping stored procedures alters ACLs.
                      Disable with --skip-automatic-sp-privileges.
…
…
…
Variables (--variable-name=value)
and boolean options {FALSE|TRUE}  Value (after reading options)
--------------------------------- ----------------------------
help                              TRUE
abort-slave-event-count           0
allow-suspicious-udfs             FALSE
auto-increment-increment          1
auto-increment-offset             1
automatic-sp-privileges           TRUE
basedir                           /usr/local/mysql/
bdb                               FALSE
bind-address                      (No default value)
character-set-client-handshake    TRUE
character-set-filesystem          binary
character-set-server              gbk
character-sets-dir                /usr/local/mysql/share/mysql/charsets/
chroot                            (No default value)
collation-server                  gbk_chinese_ci
completion-type                   0

…
…
…
innodb_buffer_pool_size           629145600
innodb_commit_concurrency         0
innodb_concurrency_tickets        500
innodb_file_io_threads            4
innodb_force_recovery             0
innodb_lock_wait_timeout          50
innodb_log_buffer_size            8388608
innodb_log_file_size              536870912
innodb_log_files_in_group         2
innodb_mirrored_log_groups        1
innodb_open_files                 300
innodb_sync_spin_loops            20
```

```
innodb_thread_concurrency          8
innodb_thre

...
```

从输出结果中可以看出，输出结果分为两部分：第一部分是对服务器参数的介绍，第二部分是当前服务器的实际参数值。如果需要查询某个参数的定义和当前值，可以用操作系统命令进行过滤。比如，想知道当前服务器字符集的设置，可以用如下命令查看：

```
[root@localhost zzx]# mysqld --verbose --help|grep character-set-server
  -C, --character-set-server=name
                      --character-set-server instead).
character-set-server               gbk
```

21.2 影响 MySQL 性能的重要参数

上一节我们介绍了 MySQL Server 端的参数查看方法。但是对于这么多参数，初学者往往不知所措。实际上，绝大多数参数也不是经常需要用户调整的。那么其中哪些是对服务器性能影响最大呢？哪些是经常需要调整的呢？本节将介绍这些重要的参数。这些参数分为两部分，前两节介绍的参数“key_buffer_size”和“table_cache”仅仅适用于 MyISAM 存储引擎；后面几节介绍的也仅仅适用于 InnoDB 存储引擎，这些参数很容易辨认，因为它们都是以“innodb_”开始。

21.2.1 key_buffer_size 的设置

首先看看 mysqld（MySQL 服务器启动命令，加“--verbose –help”显示全部启动选项）中是如何定义 key_buffer_size 参数的：

```
[root@localhost zzx]# mysqld --verbose --help|grep key_buffer_size=
  --key_buffer_size=# The size of the buffer used for index blocks for MyISAM
```

从以上定义可以看出，这个参数是用来设置索引块（Index Blocks）缓存的大小，它被所有线程共享，此参数只适用于 MyISAM 存储引擎。MySQL 5.1 以前只允许使用一个系统默认的 key_buffer，MySQL 5.1 以后提供了多个 key_buffer，可以将指定的表索引缓存入指定的 key_buffer，这样可以更小地降低线程之间的竞争。

可以这样建立一个索引缓存：

```
mysql> set global hot_cache2.key_buffer_size=128*1024;
Query OK, 0 rows affected (0.01 sec)
```

其中，global 表示对每一个新的连接，此参数都将生效。hot_cache2 是新的 key_buffer 名称。如果需要更改参数值，可以随时进行重建，例如：

```
mysql> set global hot_cache2.key_buffer_size=200*1024;
Query OK, 0 rows affected (0.00 sec)
```

然后可以把相关表的索引放到指定的索引缓存中，如下：

```
mysql> cache index sales,sales2 in hot_cache2;
+--------------+------------------+----------+----------+
| Table        | Op               | Msg_type | Msg_text |
+--------------+------------------+----------+----------+
```

```
| sakila.sales  | assign_to_keycache | status   | OK       |
| sakila.sales2 | assign_to_keycache | status   | OK       |
+---------------+--------------------+----------+----------+
2 rows in set (0.04 sec)
```

要想将索引预装到默认 key_buffer 中，可以使用 LOAD INDEX INTO CACHE 语句。例如，下面的语句可以预装表 sales 的所有索引：

```
mysql> load index into cache sales ;
+--------------+--------------+----------+----------+
| Table        | Op           | Msg_type | Msg_text |
+--------------+--------------+----------+----------+
| sakila.sales | preload_keys | status   | OK       |
+--------------+--------------+----------+----------+
1 row in set (0.00 sec)
```

如果需要删除索引缓存，则要使用下面命令：

```
mysql> set global hot_cache2.key_buffer_size=0;
Query OK, 0 rows affected (0.00 sec)
```

请注意不能删除默认 key_buffer。来看一下实际删除结果：

```
mysql>  show variables like 'key_buffer_size';
+-----------------+---------+
| Variable_name   | Value   |
+-----------------+---------+
| key_buffer_size | 8388600 |
+-----------------+---------+
1 row in set (0.00 sec)
mysql> set global key_buffer_size=0;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> show warnings;
+---------+------+-----------------------------+
| Level   | Code | Message                     |
+---------+------+-----------------------------+
| Warning | 1438 | Cannot drop default keycache |
+---------+------+-----------------------------+
1 row in set (0.01 sec)
```

可以看出，虽然提示设置成功，但是有一个 warning“Cannot drop default keycache”，提示不能删除默认 key_buffer。重新创建一个连接后，参数值果然没有更改：

```
[zzx@localhost ~]$ mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql>  show variables like 'key_buffer_size';
```

```
+-----------------+---------+
| Variable_name   | Value   |
+-----------------+---------+
| key_buffer_size | 8388600 |
+-----------------+---------+
1 row in set (0.00 sec)
```

cache index 命令在一个表和 key_buffer 之间建立一种联系，但每次服务器重启时 key_buffer 中的数据将清空。如果想要每次服务器重启时相应表的索引能自动放到 key_buffer 中，可以在配置文件中设置 init-file 选项来指定包含 cache index 语句的文件路径，然后在对应的文件中写入 cache index 语句。下面是一个例子：

```
[zzx@localhost ~]$ more /etc/my.cnf
...
key_buffer_size = 4G
hot_cache.key_buffer_size = 2G
cold_cache.key_buffer_size = 2G
init_file=/path/to/data-directory/mysqld_init.sql
...
```

每次服务器启动时执行 mysqld_init.sql 中的语句,该文件每行应包含一个 SQL 语句。下面的例子分配几个表，分别对应 hot_cache 和 cold_cache：

```
CACHE INDEX a.t1, a.t2, b.t3 IN hot_cache;
CACHE INDEX a.t4, b.t5, b.t6 IN cold_cache;
```

21.2.2 table_cache 的设置

在 mysqld 中对 table_cache 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep table_cache=
  --table_cache=#     The number of open tables for all threads.
```

这个参数表示数据库用户打开表的缓存数量。每个连接进来，都会至少打开一个表缓存。因此，table_cache 与 max_connections 有关，例如，对于 200 个并行运行的连接，应该让表的缓存至少有 200×N，这里 N 是可以执行的查询的一个联接中表的最大数量。此外，还需要为临时表和文件保留一些额外的文件描述符。

可以通过检查 mysqld 的状态变量 open_tables 和 opened_tables 确定这个参数是否过小，这两个参数的区别是前者表示当前打开的表缓存数，如果执行 FLUSH TABLES 操作，则此系统会关闭一些当前没有使用的表缓存而使得此状态值减小；后者表示曾经打开的表缓存数，会一直进行累加，如果执行 FLUSH TABLES 操作，值不会减少。下面的例子验证了这个过程。

（1）首先清空表缓存，记录两个状态的值：

```
mysql> flush tables;
Query OK, 0 rows affected (0.00 sec)

mysql> show global status like 'open_tables';;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 0     |
+---------------+-------+
```

```
1 row in set (0.00 sec)

ERROR:
No query specified

mysql> show global status like 'opened_tables';;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Opened_tables | 35    |
+---------------+-------+
1 row in set (0.00 sec)
```

（2）然后，执行一个 SQL，对表 t 进行查询：

```
mysql> select count(1) from t;
+----------+
| count(1) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)
```

（3）接着再查看这两个参数的值：

```
mysql> show global status like 'open_tables';;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 1     |
+---------------+-------+
1 row in set (0.00 sec)

ERROR:
No query specified

mysql> show global status like 'opened_tables';;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Opened_tables | 36    |
+---------------+-------+
1 row in set (0.00 sec)
```

（4）可以发现，两个参数值都因为对表 t 的查询而状态加 1。这时，再次执行刚才对表 t 的查询：

```
mysql> select count(1) from t;
+----------+
| count(1) |
```

```
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)

mysql> show global status like 'open_tables';;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Open_tables   | 1     |
+---------------+-------+
1 row in set (0.00 sec)

ERROR:
No query specified

mysql> show global status like 'opened_tables';;
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Opened_tables | 36    |
+---------------+-------+
1 row in set (0.00 sec)
```

（5）此时这两个参数的值并没有变化，因为表 t 的描述符已经在此连接中打开过一次，因此保存在了表缓存中。因此，状态值“open_tables”对于设置 table_cache 值有着更有价值的参考。

21.2.3 innodb_buffer_pool_size 的设置

mysqld 中对 innodb_buffer_pool_size 参数的定义如下（grep 后面的“-A 2”表示显示包含指定字符串的行和此行后面的 2 行）：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_buffer_pool_size" -A 2
  --innodb_buffer_pool_size=#
                      The size of the memory buffer InnoDB uses to cache data
                      and indexes of its tables.
```

这个参数定义了 InnoDB 存储引擎的表数据和索引数据的最大内存缓冲区大小。和 MyISAM 存储引擎不同，MyISAM 的 key_buffer_size 只缓存索引键，而 innodb_buffer_pool_size 却是同时为数据块和索引块做缓存，这个特性和 Oracle 是一样的。这个值设得越高，访问表中数据需要的磁盘 I/O 就越少。在一个专用的数据库服务器上，可以设置这个参数达机器物理内存大小的 80%。尽管如此，还是建议用户不要把它设置得太大，因为对物理内存的竞争可能在操作系统上导致内存调度。

21.2.4 innodb_flush_log_at_trx_commit 的设置

在 mysqld 中对 innodb_flush_log_at_trx_commit 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_flush_log_at_trx_commit" -A 3
  --innodb_flush_log_at_trx_commit[=#]
                      Set to 0 (write and flush once per second), 1 (write and
                      flush at each commit) or 2 (write at commit, flush once
                      per second).
```

这个参数是用来控制缓冲区中的数据写入到日志文件以及日志文件数据刷新到磁盘的操作时机。对这个参数的设置可以对数据库在性能与数据安全之间进行折中。

- 当这个参数是 0 的时候，日志缓冲每秒一次地被写到日志文件，并且对日志文件做向磁盘刷新的操作，但是在一个事务提交不做任何操作。
- 当这个参数是 1 的时候，在每个事务提交时，日志缓冲被写到日志文件，并且对日志文件做向磁盘刷新的操作。
- 当这个参数是 2 的时候，在每个事务提交时，日志缓冲被写到日志文件，但不对日志文件做向磁盘刷新的操作，对日志文件每秒向磁盘做一次刷新操作。

innodb_flush_log_at_trx_commit 参数的默认值是 1，也是最安全的设置，即每个事务提交的时候都会从 log buffer 写到日志文件，而且会实际刷新磁盘，但是这样性能有一定的损失。如果可以容忍在数据库崩溃的时候损失一部分数据，那么设置成 0 或者 2 都会有所改善。设置成 0，则在数据库崩溃的时候会丢失那些没有被写入日志文件的事务，最多丢失 1 秒钟的事务，这种方式是最不安全的，也是效率最高的。设置成 2 的时候，因为只是没有刷新到磁盘，但是已经写入日志文件，所以只要操作系统没有崩溃，那么并没有丢失数据，比设置成 0 更安全一些。

在 MySQL 官方手册中，为了确保事务的持久性和复制设置的一致性，都是建议将这个参数设置为 1 的。

21.2.5 innodb_additional_mem_pool_size 的设置

在 mysqld 中，对 innodb_additional_mem_pool_size 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_additional_mem_pool_size" -A
2
  --innodb_additional_mem_pool_size=#
                      Size of a memory pool InnoDB uses to store data
                      dictionary information and other internal data
```

这个参数是 InnoDB 存储引擎用来存储数据库结构和其他内部数据结构的内存池的大小，其默认值是 1MB。应用程序里的表越多，则需要在这里分配越多的内存。如果 InnoDB 用光了这个池内的内存，则 InnoDB 开始从操作系统分配内存，并且往 MySQL 错误日志写警告信息。没有必要给这个缓冲池分配非常大的空间，在应用相对稳定的情况下，这个缓冲池的大小也相对稳定，系统默认值是 1MB。

21.2.6 innodb_lock_wait_timeout 的设置

在 mysqld 中，对 innodb_lock_wait_timeout 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_lock_wait_timeout" -A 2
  --innodb_lock_wait_timeout=#
                      Timeout in seconds an InnoDB transaction may wait for a
                      lock before being rolled back.
```

MySQL 可以自动地监测行锁导致的死锁并进行相应的处理，但是对于表锁导致的死锁不能自动的监测，所以该参数主要被用于在出现类似情况的时候等待指定的时间后回滚。系统默认值是 50 秒，用户可以根据应用的需要进行调整。

21.2.7 innodb_support_xa 的设置

在 mysqld 中，对 innodb_support_xa 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_support_xa"
  --innodb_support_xa Enable InnoDB support for the XA two-phase commit
```

通过该参数设置是否支持分布式事务，默认值是 ON 或者 1，表示支持分布式事务。如果确认应用中不需要使用分布式事务，则可以关闭这个参数，减少磁盘刷新的次数并获得更好的 InnoDB 性能。

21.2.8 innodb_log_buffer_size 的设置

在 mysqld 中，对 innodb_log_buffer_size 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_log_buffer_size" -A2
  --innodb_log_buffer_size=#
                      The size of the buffer which InnoDB uses to write log to
                      the log files on disk.
```

从参数名称可以显而易见看出，含义是日志缓存的大小。默认的设置在中等强度写入负载以及较短事务的情况下，一般都可以满足服务器的性能要求。如果存在更新操作峰值或者负载较大，就应该考虑加大它的值了。如果它的值设置太高了，可能会浪费内存，因为它每秒都会刷新一次，因此无需设置超过 1 秒所需的内存空间。通常设置为 8～16MB 就足够了。越小的系统它的值越小。系统默认值是 1MB。

21.2.9 innodb_log_file_size 的设置

在 mysqld 中，对 innodb_log_file_size 参数的定义如下：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep "\-\-innodb_log_file_size" -A1
  --innodb_log_file_size=#
                      Size of each log file in a log group.
```

该参数含义是一个日志组（log group）中每个日志文件的大小。此参数在高写入负载尤其是大数据集的情况下很重要。这个值越大则性能相对越高，但是带来的副作用是，当系统灾难时恢复时间会加大。系统默认值是 5MB。

21.3 小结

本章的题目是优化 MySQL Server，实际上更多地介绍了一些服务器性能相关的参数。对服务器的优化，实际上也就是对这些参数的不断调整。对于初学者，MySQL 提供了一些针对不同级别应用的参数例子可供参考。对于更高级别的使用者，则需要根据不同的应用更加灵活地设置这些参数，并在实践中不断尝试摸索，使得服务器的性能与安全达到最佳组合。

第22章　磁盘 I/O 问题

作为应用系统的持久化层，不管数据库采取了什么样的 Cache 机制，但数据库最终总是要将数据储存到可以长久保存的 I/O 设备——磁盘上，但磁盘的存取速度显然要比 CPU、RAM 的速度慢很多，因此，对于比较大的数据库，磁盘 I/O 一般总会成为数据库的一个性能瓶颈！

实际上，我们前面提到的 SQL 优化、数据库对象优化、数据库参数优化，以及应用程序优化等，大部分都是想通过减少或延缓磁盘读写来减轻磁盘 I/O 的压力及其对性能的影响。解决磁盘 I/O 问题，减少或延缓磁盘操作肯定是一个重要方面，但磁盘 I/O 是不可避免的，因此，增强磁盘 I/O 本身的性能和吞吐量也是一个重要方面。本章将从磁盘阵列、符号链接、裸设备等更底层的方面来介绍提高磁盘 I/O 能力的一些技术和方法。

22.1 使用磁盘阵列

RAID 是 Redundant Array of Inexpensive Disks 的缩写，翻译成中文就是“廉价磁盘冗余阵列”，通常就叫做磁盘阵列。RAID 就是按照一定策略将数据分布到若干物理磁盘上，这样不仅增强了数据存储的可靠性，而且可以提高数据读写的整体性能，因为通过分布实现了数据的“并行”读写。

RAID 最早是用来取代大型计算机上高档存储设备的，相对于那些高档存储设备而言，RAID 的价格很便宜，这也是其名称中带有“廉价”一词的原因。但其实很长时间，对于 PC 机而言，其价格远远谈不上“廉价”！近几年，随着存储技术的发展，RAID 开始变得真正物美价廉了。

22.1.1 常见 RAID 级别及其特性

根据数据分布和冗余方式，RAID 分为许多级别，不同存储厂商提供的 RAID 卡或设备支持的 RAID 级别也不尽相同，这里只介绍最常见也是最基本的几种，其他 RAID 级别基本上都是在这几种基础上的改进。

表 22-1　　常见 RAID 级别比较

RAID 级别	特性	优点	缺点
RAID 0	也叫条带化（Stripe），按一定的条带大小（Chunk Size)将数据依次分布到各个磁盘，没有数据冗余	数据并发读写速度快，无额外磁盘空间开销，投资省	数据无冗余保护，可靠性差
RAID 1	也叫磁盘镜像（Mirror），两个磁盘一组，所有数据都同时写入两个磁盘，但读时从任一磁盘读都可以	数据有完全冗余保护，只要不出现两块镜像磁盘同时损坏，不会影响使用；可以提高并发读性能	容量一定的话，需要 2 陪的磁盘，投资比较大
RAID 10	是 RAID 1 和 RAID 0 的结合，也叫 RAID 1+0。先对磁盘做镜像，再条带话，使其兼具 RAID 1 的	可靠性高，并发读写性能优良	容量一定的话，需要 2 陪的磁盘，投资比较大

	可靠性和RAID 0的优良并发读写性能		
RAID 4	象 RAID 0 一样对磁盘组条带化，不同的是：需要额外增加一个磁盘，用来写各 Stripe 的校验纠错数据	RAID 中的一个磁盘损坏，其数据可以通过校验纠错数据计算出来，具有一定容错保护能力；读数据速度快	每个 Stripe 上数据的修改都要写校验纠错块，写性能受影响；所有纠错数据都在同一磁盘上，风险大，也会形成一个性能瓶颈；在出现坏盘时，读性能会下降
RAID 5	是对 RAID 4 的改进：将每一个条带（Stripe）的校验纠错数据块也分布写到各个磁盘，而不是写到一个特定的磁盘	基本同 RAID 4，只是其写性能和数据保护能力要更强一点	写性能不及 RAID 0、RAID 1 和 RAID 10，容错能力也不及 RAID 1；在出现坏盘时，读性能会下降

22.1.2 如何选择 RAID 级别

了解各种 RAID 级别的特性后，我们就可以根据数据读写的特点、可靠性要求，以及投资预算等来选择合适的 RAID 级别，比如：

- 数据读写都很频繁，可靠性要求也很高，最好选择 RAID 10；
- 数据读很频繁，写相对较少，对可靠性有一定要求，可以选择 RAID 5；
- 数据读写都很频繁，但可靠性要求不高，可以选择 RAID 0。

22.1.3 虚拟文件卷或软 RAID

最初，RAID 都是由硬件实现的，要使用 RAID，至少需要有一个 RAID 卡。但现在，一些操作系统中提供的软件包，也模拟实现了一些 RAID 的特性，虽然性能上不如硬 RAID，但相比单个磁盘，性能和可靠性都有所改善。比如，Linux 下的逻辑卷（Logical Volume）系统 lvm2，支持条带化（Stripe）；Linux 下的 MD（Multiple Device）驱动，支持 RAID 0、RAID 1、RAID 4、RAID 5、RAID 6 等。在不具备硬件条件的情况下，可以考虑使用上述虚拟文件卷或软 RAID 技术，具体配置方法可参见 Linux 帮助文档。

22.2 使用 Symbolic Links 分布 I/O

MySQL 的数据库名和表名是与文件系统的目录名和文件名对应的，默认情况下，创建的数据库和表都存放在参数 datadir 定义的目录下。这样如果不使用 RAID 或逻辑卷，所有的表都存放在一个磁盘设备上，无法发挥多磁盘并行读写的优势！在这种情况下，我们就可以利用操作系统的符号连接（Symbolic Links）将不同的数据库或表、索引指向不同的物理磁盘，从而达到分布磁盘 I/O 的目的。

（1）将一个数据库指向其他物理磁盘。

其方法是先在目标磁盘上创建目录，然后再创建从 MySQL 数据目录到目标目录的符号连接：

```
shell> mkdir /otherdisk/databases/test
shell> ln -s /otherdisk/databases/test /path/to/datadir
```

（2）将MyISAM（其他存储引擎的表不支持）表的数据文件或索引文件指向其他物理磁盘。

- 对于新建的表，可以通过在CREATE TABLE语句中增加DATA DIRECTORY和INDEX DIRECTORY选项来完成，例如：

```
Create table test(id int primary key,
Name varchar(20))
Type = myisam
DATA DIRECTORY = '/disk2/data'
INDEX DIRECTORY = '/disk3/index'
```

- 对于已有的表，可以先将其数据文件（.MYD）或索引文件（.MYI）转移到目标磁盘，然后再建立符号连接即可。需要说明的是表定义文件（.frm）必须位于MySQL数据文件目录下，不能用符号连接。

（3）在Windows下使用符号连接。

以上介绍的是Linux/UNIX下符号连接的使用方法，在Windows下，是通过在MySQL数据文件目录下创建包含目标路径并以“.sym”结尾的文本文件来实现的。例如，假设MySQL的数据文件目录是C:\mysql\data，要把数据库foo存放到D:\data\foo，可以按以下步骤操做：

- 创建目录D:\data\foo；
- 创建文件C:\mysql\data\foo.sym，在其中输入D:\data\foo。

这样在数据库foo创建的表都会存储到D:\data\foo目录下。

注意：使用Symbolic Links存在一定的安全风险，如果不使用Symbolic Links，应通过启动参数skip-symbolic-links禁用这一功能。

22.3 禁止操作系统更新文件的 atime 属性

atime 是 Linux/UNIX 系统下的一个文件属性，每当读取文件时，操作系统都会将读操作发生的时间回写到磁盘上。对于读写频繁的数据库文件来说，记录文件的访问时间一般没有任何用处，却会增加磁盘系统的负担，影响 I/O 的性能！因此，可以通过设置文件系统的 mount 属性，阻止操作系统写 atime 信息，以减轻磁盘 I/O 的负担。在 Linux 下的具体做法是：
修改文件系统配置文件/etc/fstab，指定 noatime 选项：

```
    LABEL=/home             /home             ext3     noatime        1 2
```

然后重新 mount 文件系统：

```
    #mount -oremount /home
```

完成上述操作，以后读/home 下文件就不会再写磁盘了。

22.4 用裸设备（Raw Device）存放 InnoDB 的共享表空间

MyISAM 存储引擎有自己的索引缓存机制，但数据文件的读写完全依赖于操作系统，操作系统磁盘 I/O 缓存对 MyISAM 表的存取很重要。但 InnoDB 存储引擎与 MyISAM 不同，

它采用类似 Oracle 的数据缓存机制来 Cache 索引和数据，操作系统的磁盘 I/O 缓存对其性能不仅没有帮助，甚至还有反作用。因此，在 InnoDB 缓存充足的情况下，可以考虑使用 Raw Device 来存放 InnoDB 共享表空间，具体操作方法如下。

（1）修改MySQL配置文件，在innodb_data_file_path参数中增加裸设备文件名并指定newraw属性：

```
……
[mysqld]
innodb_data_home_dir=
innodb_data_file_path=/dev/hdd1:3Gnewraw;/dev/hdd2:2Gnewraw
……
```

（2）启动MySQL，使其完成分区初始化工作，然后关闭MySQL。此时还不能创建或修改InnoDB表。

（3）将innodb_data_file_path中的newraw改成raw：

```
……
class=programlisting[mysqld]
innodb_data_home_dir=
innodb_data_file_path=/dev/hdd1:3Graw;/dev/hdd2:2Graw
……
```

（4）重新启动即可开始使用。

22.5 小结

本章站在操作系统的角度介绍了如何对 MySQL 数据库进行优化，主要讨论了 I/O 的优化问题、文件系统分布的优化问题等。在大多数的数据库系统中，磁盘 I/O 都会是影响系统性能的瓶颈，希望通过本章，读者能够掌握一些减少磁盘 I/O 以提高系统性能的方法。

第23章　应用优化

前面章节介绍了很多数据库的优化措施。但是在实际生产环境中，由于数据库服务器本身的性能局限，就必须要对前台的应用来进行一些优化，使得前台访问数据库的压力能够减到最小。本章将介绍一些常用的应用优化方法。

23.1 使用连接池

对于访问数据库来说，建立连接的代价比较昂贵，因此，我们有必要建立“连接池”以提高访问的性能。从名字上理解，“连接池”是一个存放“连接”的“池子”，再具体一些，我们可以把连接当作对象或者设备，统一放在一个“池子”里面，以前需要直接访问数据库的地方，现在都改为从这个“池子”里面获取连接来使用。因为“池子”中的连接都已经预

先创建好，可以直接分配给应用使用，因此大大减少了创建新连接所耗费的资源。连接返回后，本次访问将连接交还给“连接池”，以供新的访问使用。

23.2 减少对 MySQL 的访问

在实际应用中，我们的硬件资源通常是有限的、无法扩充的。这种情况下，应用有什么措施能减少对数据库的访问呢？本节将向大家介绍一些简单的方法。

23.2.1 避免对同一数据做重复检索

应用中需要理清对数据库的访问逻辑。能够一次连接就能够提取出所有结果的，就不用两次连接，这样可以大大减少对数据库无谓的重复访问。

例如，在某应用中需要检索某人的年龄和性别，那么就可以执行以下查询：

```
Select old,gender from users where userid = 231;
```

之后又需要这个人的家庭住址，可以又执行：

```
Select address from users where userid = 231;
```

这样的话，就需要向数据库提交两次请求，数据库就要做两次查询操作，其实完全可以用一句 SQL 语句得到想要的结果，然后把得到的结果放到变量中已备后用，比如：

```
Select old,gender,address from users where userid = 231;
```

不管读者是否相信，由于上面原因导致的性能问题，在很多应用系统中都存在，因此在理清应用逻辑并向数据库提交请求前进行深思熟虑是很有必要的。

23.2.2 使用查询缓存

MySQL 的查询缓存（MySQL Query Cache）是在 4.1 版本以后新增的功能，它的作用是存储 SELECT 查询的文本以及相应结果。如果随后收到一个相同的查询，服务器会从查询缓存中重新得到查询结果，而不再需要解析和执行查询。

查询缓存的适用对象是更新不频繁的表，当表更改（包括表结构和表数据）后，查询缓存值的相关条目被清空。

查询缓存相关的参数主要有以下几个：

```
mysql> show variables like '%query_cache%';
+------------------------------+---------+
| Variable_name                | Value   |
+------------------------------+---------+
| have_query_cache             | YES     |
| query_cache_limit            | 1048576 |
| query_cache_min_res_unit     | 4096    |
| query_cache_size             | 0       |
| query_cache_type             | OFF     |
| query_cache_wlock_invalidate | OFF     |
+------------------------------+---------+
6 rows in set (0.00 sec)
```

对于以上几个参数，具体解释如下。

- have_query_cache 表明服务器在安装是否已经配置了高速缓存

- query_cache_size 表明缓存区大小，单位为 M。
- query_cache_type 的变量值从 0 到 2，含义分别为：

  0 或者 off（缓存关闭）

  1 或者 on（缓存打开，使用 SQL_NO_CACHE 提示的 SELECT 除外）

  2 或者 demand（只有带 SQL_CACHE 的 SELECT 语句提供高速缓存）

通过 SHOW STATUS 命令，可以监视查询缓存的使用状况，如表 23-1 所示。

表 23-1　　MySQL 查询缓存的性能监控参数

变量	含义
Qcache_queries_in_cache	在缓存中已注册的查询数目
Qcache_inserts	被加入到缓存中的查询数目
Qcache_hits	缓存采样数数目
Qcache_lowmem_prunes	因为缺少内存而被从缓存中删除的查询数目
Qcache_not_cached	没有被缓存的查询数目（不能被缓存的，或由于 QUERY_CACHE_TYPE）
Qcache_free_memory	查询缓存的空闲内存总数
Qcache_free_blocks	查询缓存中的空闲内存块的数目
Qcache_total_blocks	查询缓存中的块的总数目

23.2.3 增加 CACHE 层

在应用中，我们可以在应用端加 CACHE 层来达到减轻数据库的负担的目的。CACHE 层有很多种，也有很多种实现的方式，只要能达到降低数据库的负担，又能满足应用就可以，这就需要根据应用的实际情况进行特殊处理。

比如，可以把部分数据从数据库中抽取出来放到应用端以文本方式存储，然后有查询需求的话，可以直接从这个“CACHE”中检索，由于这里的数据量小所以能够达到很高的查询效率，而且也减轻了数据库的负担。当然这种方案还涉及很多其他问题，比如如果有数据更新怎么办、多长时间刷新一次“CACHE”等，都需要根据具体应用环境进行相应的处理。

再比如用户可以在应用端建立一个二级数据库，把访问频度非常大的数据放到二级库上，然后设定一个机制与主数据库进行同步，这样用户的主要操作都在二级数据库上进行，大大地降低了主数据库的压力。

各种“CACHE”层的实现方式不同，这里只是抛砖引玉，给读者一个解决问题的思路，不可千篇一律地照搬照抄。

23.3 负载均衡

负载均衡（Load Balance）是实际应用中使用非常普遍的一种优化方法，它的机制就是利用某种均衡算法，将固定的负载量分布到不同的服务器上，以此来减轻单台服务器的负载，达到优化的目的。负载均衡可以用在系统中的各个层面中，从前台的 Web 服务器到中间层的应用服务器，最后到数据层的数据库服务器，都可以使用。本节主要介绍 MySQL 数据库端的一些负载均衡方法。

23.3.1 利用 MySQL 复制分流查询操作

利用 MySQL 的主从复制（具体介绍见第 29 章）可以有效地分流更新操作和查询操作，

具体的实现是一个主服务器承担更新操作，而多台从服务器承担查询操作，主从之间通过复制实现数据的同步。多台从服务器一方面用来确保可用性，一方面可以创建不同的索引以满足不同查询的需要。

对于主从之间不需要复制全部表的情况，可以通过在主服务器上搭建一个虚拟的从服务器，将需要复制到从服务器的表设置成 BlackHole 引擎，然后定义 replicate-do-table 参数只复制这些表，这样就过滤出需要复制的 BINLOG，减少了传输 BINLOG 的带宽。因为搭建的虚拟从服务器只起到过滤 BINLOG 的作用，并没有实际记录任何数据，所以对主数据库服务器的性能影响也非常得有限。

通过复制来分流查询是减少主数据库负载的一个常用方法，但是这种办法也存在一些问题，最主要的问题是当主数据库上更新频繁或者网络出现问题的时候，主从之间的数据可能存在比较大的延迟更新，从而造成查询结果和主数据库上有所差异。因此应用在设计的时候需要有所考虑。

23.3.2 采用分布式数据库架构

分布式的数据库架构适合大数据量、负载高的情况，它具有良好的扩展性和高可用性。通过在多台服务器之间分布数据，可以实现在多台服务器之间的负载平均，提高了访问的执行效率。具体实现的时候，可以使用 MySQL 的 CLUSTER 功能（具体介绍见第 30 章）或者通过用户自己编写的程序来实现全局事务。需要注意的是当前分布式事务只支持 InnoDB 存储引擎，因此如果自己编写程序来实现分布式架构数据库的话，那么就必须采用 InnoDB 存储引擎。

23.4 其他优化措施

除了上面介绍的优化方法，还有很多我们日常不太注意，但是很有用的优化方法，这里总结了几条，希望读者能从实践中进行更多的总结。

- 对于没有删除行操作的 MyISAM 表，插入操作和查询操作可以并行进行，因为没有删除操作的表查询期间不会阻塞插入操作。对于确实需要执行删除操作的表，尽量在空闲时间进行批量删除操作，并且在进行删除操作之后应该进行 OPTIMIZE 操作来消除由于删除操作带来的空洞，以避免将来的更新操作阻塞其他操作。
- 充分利用列有默认值的事实。只有当插入的值不同于默认值时，才明确地插入值。这会减少 MySQL 需要做的语法分析从而提高插入速度。
- 表的字段尽量不使用自增长变量，在高并发情况下该字段的自增可能对效率有比较大的影响，推荐通过应用来实现字段的自增长。

23.5 小结

应用的优化也是数据库优化的重要组成部分，数据库本身的优化是有一定的局限性的，到了一定程度就很难再有大的提升，但是如果从应用端进行考虑，那么还有很多可以优化的内容。本章从连接池、减少数据库访问、负载均衡以及 CACHE 层等方面进行了讨论。应用层优化的方面还有很多与应用本身密切相关，这些东西需要读者在实际应用环境中不断总结、不断积累经验才能达到更好的优化效果。

第24章　MySQL 高级安装和升级

在第 1 章中已经介绍了 MySQL 在 Windows 平台上安装 NOINSTALL 包和图形化包的方法，以及在 Linux/UNIX 平台上安装 RPM 包的方法。对于 Linux/UNIX 平台来说，用户还可以考虑采用另外两种安装包来进行安装，一种是二进制包（Binary Package），另一种是源码包（Source Package），这两种包都可以从 MySQL 的官方网站下载（http://dev.mysql.com/downloads/mysql/5.0.html），因为针对不同的硬件和操作系统安装包有所不同，所以大家在下载时请根据实际安装环境选择相应的包。这两种安装包相对于 RPM 包的最大优点是安装配置更灵活，更适合于中高级用户，因此称为“高级”安装。本章将主要对这两种安装包的使用进行详细的介绍。

24.1 Linux/UNIX 平台下的安装

本节主要介绍了 Linux/UNIX 平台下 3 种安装包之间的区别以及各自的安装步骤，并对其参数文件的设置方法做了详细的介绍。

24.1.1 安装包比较

Linux 的安装包分为 RPM 包、二进制包和源码包，表 24-1 中简单描述了 3 种安装包之间的主要差异，其中文件布局指的是 MySQL 安装完毕后生成的各个目录和用途。

表 24-1　　Linux 平台下的 3 种安装包比较

	RPM	二进制	源码
优点	安装简单，适合初学者学习使用	安装简单；可以安装到任何路径下，灵活性好；一台服务器可以安装多个 MySQL	在实际安装的操作系统进行可根据需要定制编译，最灵活；性能最好；一台服务器可以安装多个 MySQL
缺点	需要单独下载客户端和服务器；安装路径不灵活，默认路径不能修改，一台服务器只能安装一个 MySQL	已经经过编译，性能不如源码编译得好；不能灵活定制编译参数	安装过程较复杂；编译时间长
文件布局	/usr/bin（客户端程序和脚本） /usr/sbin（mysqld 服务器） /var/lib/mysql（日志文件和数据库） /usr/share/doc/packages（文档） /usr/include/mysql（包含(头)文件） /usr/lib/mysql（库文件） /usr/share/mysql（错误消息	bin（客户端程序和 mysqld 服务器） data（日志文件和数据库） docs（文档和 ChangeLog） include（包含(头)文件）lib（库文件） scripts（mysql_install_db 脚本，用来安装系统数据库） share/mysql（错误消息文件） sql-bench（基准程序）	bin（客户端程序和脚本） include/mysql（包含(头)文件） info（Info 格式的文档） lib/mysql（库文件） libexec（mysqld 服务器）、 share/mysql（错误消息文件） sql-bench（基准程序和 crash-me 测试） var（数据库和日志文件）

	和字符集文件） /usr/share/sql-bench（基准程序）		

24.1.2 安装 RPM 包

RPM 包的安装步骤在本书的第 1 章中已经详细介绍过了，这里再简单说一下。大多数情况下，下载 Server 包和 Client 包就可以满足大部分应用，下载的文件名通常是以下格式：

```
MySQL-包类型(例如 Server 或者 Client)-版本类型(例如社区版或者企业版)-版本号(例如 5.0.41)
-0.操作系统类型(rhel3).CPU 类型（例如 i386）.rpm
```

例如，下面两个文件分别是基于 X86 架构的 Redhat Linux 操作系统上的社区版 MySQL 的 Server 包和 Client 包，版本都为 5.0.41。

- MySQL-server-community-5.0.41-0.rhel3.i386.rpm
- MySQL-client-community-5.0.41-0.rhel3.i386.rpm

对于 RPM 文件一般使用“rpm –ivh 文件名”的方式进行安装，其中 rpm 是 RPM 包的管理工具，-ivh 分别是 rpm 的 3 个选项，具体含义如表 24-2 所示。

表 24-2 rpm 选项及其说明

选项	说明
-i, --install	表示对后面的 RPM 包进行安装
-v, --verbose	安装过程中提供更多的输出信息
-h, --hash	打印字符“#”来进行安装进度的提示

rpm 工具更详细的参数使用方法请用 rpm --help 命令来查看。

例如，对上例中 RPM 包的安装方法如下：

```
shell> rpm -ivh MySQL-server-community-5.0.41-0.rhel3.i386.rpm
shell> rpm -ivh MySQL-client-community-5.0.41-0.rhel3.i386.rpm
```

24.1.3 安装二进制包

如果用户既不想安装最简单却不够灵活的 RPM 包，又不想安装复杂费时的源码包，那么，已经编译好的二进制包将是最好的选择。

具体安装步骤如下：

（1）用 root 登录操作系统，增加 mysql 用户和组，数据库将安装在此用户下：

```
shell> groupadd mysql
shell> useradd -g mysql mysql
```

（2）解压二进制安装包，假设安装文件放在/home/mysql，并对解压后的 mysql 目录加一个符号链接“mysql”，这样对 mysql 目录的操作会更方便：

```
shell> cd /home/mysql
shell>tar -xzvf  /home/mysql/mysql-VERSION-OS.tar.gz
shell> ln -s mysql-VERSION-OS mysql
```

（3）在数据目录下创建系统数据库和系统表，--user 表示这些数据库和表的 owner 是此用户：

```
shell> cd mysql
shell> scripts/mysql_install_db --user=mysql
```

（4）设置目录权限，将 data 目录 owner 改为 mysql，其他目录和文件为 root：

```
shell> chown -R root:mysql  .
shell> chown -R mysql:mysql data
```

（5）启动 MySQL：

```
shell> bin/mysqld_safe --user=mysql &
```

24.1.4 安装源码包

如果对数据库的性能要求很高，并且希望能够灵活地定制安装选项，安装源码包将是最好的选择。其安装步骤与安装二进制包非常类似，具体如下。

（1）用 root 登录操作系统，增加 mysql 用户和组，数据库将安装在此用户下：

```
shell> groupadd mysql
shell> useradd -g mysql mysql
```

（2）解压源码安装文件 mysql-VERSION.tar.gz，并进入解压后的目录：

```
shell> gunzip < mysql-VERSION.tar.gz | tar -xvf -
shell> cd mysql-VERSION
```

（3）用 configure 工具来编译源码，这里可以选择很多编译参数，具体可以用 configure --help 来进行查看。这里假设 MySQL 安装在/usr/local/mysql 下：

```
shell> ./configure --prefix=/usr/local/mysql
shell> make
shell> make install
```

（4）选择一个 MySQL 自带的样例配置文件（比如 my-medium.cnf），并 cp 到/etc 下改名为 my.cnf。当然，这个配置文件也可以自己来编写。

```
shell> cp support-files/my-medium.cnf /etc/my.cnf
```

（5）在数据目录下创建系统数据库和系统表，--user 表示这些数据库和表的 owner 是此用户。

```
shell> cd /usr/local/mysql
shell> bin/mysql_install_db --user=mysql
```

（6）设置目录权限，将 var 目录 owner 改为 mysql（源码安装，默认数据目录为 var），其他目录和文件为 root。

```
shell> chown -R root  .
shell> chown -R mysql var
shell> chgrp -R mysql .
```

（7）启动 MySQL：

```
shell> bin/mysqld_safe --user=mysql &
```

24.1.5 参数设置方法

在 MySQL 中，参数的初始化是通过参数文件来进行设置，如果不设置参数文件，MySQL 就按照系统中所有参数的默认值来进行启动，通过“mysqld –verbose –help”命令可以来查看参数文件中所有参数的当前设置值。

在 Windows 和 Linux 上，参数文件可以被放在多个位置，数据库启动的时候将按照不同的顺序来搜索，如果在多个位置都有参数文件，则搜索顺序靠后的文件中的参数将覆盖靠前的参数。表 24-3 和表 24-4 分别给出了在不同操作系统中数据库启动时 MySQL 搜索参数文件的顺序。

表 24-3　　Windows 平台上 MySQL 参数文件的读取顺序

文件名	备注
WINDIR\my.ini	全局选项
C:\my.cnf	全局选项
INSTALLDIR\my.ini	全局选项
defaults-extra-file	用--defaults-extra-file=path 指定的文件

- WINDIR 典型名称为 C:\WINDOWS 或 C:\WINNT。用户可以使用以下命令从 WINDIR 环境变量值确定自己的确切位置：

```
C：\> echo %WINDIR%
```

- INSTALLDIR 是 MySQL 的安装目录，比如 c:\mysql。
- defaults-extra-file 是 MySQL 启动时候可选择的附带选项，用此参数可以指定任何路径下的配置文件。
- “全局选项”表示如果一台服务器上安装了多个 MySQL，则每个 MySQL 服务启动的时候都会首先从此选项中读取参数。

表 24-4　　Linux 平台上 MySQL 参数文件读取顺序

文件名	备注
/etc/my.cnf	全局选项
$MYSQL_HOME/my.cnf	服务器相关选项，其中$MYSQL_HOME 为环境变量中指定的 MySQL 安装目录
defaults-extra-file	用--defaults-extra-file=path 指定的文件
~/.my.cnf	用户相关选项

> 注意：不管在 Windows 还是 Linux 平台上，为了避免混淆，建议最好只在一个位置指定配置文件。

对于初学者，建议从$MYSQL_HOME/support-files 下面按照需要 cp 合适的配置文件为数据库配置文件，例如：

```
cp my-large.cnf    /etc/my.cnf
```

当参数需要修改时，可以选择以下 3 种修改方式（命令行中 para_name 表示要修改的参数名，value 表示要修改的目标参数值）。

- session 级修改（只对本 session 有效），在 mysql 提示符下执行如下命令：

```
mysql>set para_name=value;
```

- 全局级修改（对所有新的连接都有效,但是对本 session 无效，数据库重启后失效），在 mysql 提示符下执行如下命令：

```
mysql>set global    para_name=value;
```

- 永久修改。将参数在 my.cnf 中增加或者修改，数据库重启后生效。

24.2 源码包安装的性能考虑

很多对 MySQL 很熟悉的用户都喜欢采用源码包来进行安装，因为在安装源码包的过程中能够提供更灵活的安装选项和更多的性能设置。本节就源码安装的常用选项进行简单的介绍，有兴趣的读者可以参考 MySQL 官方文档以获得更详细的信息。

24.2.1 去掉不需要的模块

源码安装由于可以灵活地进行数据库的定制编译，因此有更强的灵活性。某些编译选项可以大大增强用户数据库的性能。

执行如下命令可以看到所有编译的配置选项：

```
shell> ./configure --help
```

如果只安装客户端，则可以执行如下命令：

```
shell> ./configure --without-server
```

如果不想要位于“/usr/local/var”目录下面的日志（log）文件和数据库，可以使用类似于下列 configure 命令的一个：

```
shell>./configure  --prefix=/usr/local/mysql
shell>./configure  --prefix=/usr/local localstatedir=/usr/local/mysql/data
```

第一个命令改变安装前缀以便将所有内容安装到“/usr/local/mysql”下面而非默认的“/usr/local”。第二个命令保留默认安装前缀，但是覆盖了数据库目录默认目录（通常是“/usr/local/var”）并且把它改为/usr/local/mysql/data。编译完 MySQL 后，可以通过选项文件更改这些选项。

修改 socket 的默认位置：

```
shell> ./configure --with-unix-socket-path=/usr/local/mysql/tmp/mysql.sock
```

24.2.2 只选择要使用的字符集

MySQL 使用 LATIN1 和 LATIN1_SWEDISH_CI 作为默认的字符集和校对规则。如果想改变安装后的默认字符集和默认排序规则，可以使用如下编译选项：

```
shell> ./configure -- with-charset=CHARSET
./configure --with-collation=COLLATION
```

如果不需要安装所有的字符集，那么编译的时候可以选择只安装用户需要的字符集。这样可以节省更多的系统资源，并且使得安装后的 MySQL 速度更快。编译选项如下：

```
shell>./configure  --with-extra-charsets=LIST
```

LIST 可以是下面任何一项：

- 以空格为间隔的一系列字符集名；
- complex，以包括不能动态装载的所有字符集；
- all，包括所有字符集。

24.2.3 使用静态编译以提高性能

使用静态编译将可以大大提高 MySQL 的性能，编译选项如下：

```
shell>./configure --with-client-ldflags=-all-static --with-mysqld-ldflags=-all-static
```

其中的选项含义如下：

- --with-client-ldflags=-all-static 以纯静态方式编译客户端；
- --with-mysqld-ldflags=-all-static 以纯静态方式编译服务端。

24.3 升级 MySQL

MySQL 的版本更新很快，新版本中往往包含了很多新功能，并且解决了很多旧版本中的 BUG，因此很多情况下用户需要对数据库进行升级。

MySQL 的升级很简单，以下给出了几种不同的升级方法，每种升级方法都有一定的优缺点，用户可以按照实际需求选择合适的方法进行操作。

方法一：最简单，适合于任何存储引擎（不一定速度最快）。

（1） 在目标服务器上安装新版本的 MySQL。

（2） 在新版本 MySQL 上创建和老版本同名的数据库。命令如下：

```
shell> mysqladmin -h hostname -P port -u user -p passwd  create db_name
```

（3） 将老版本 MySQL 上的数据库通过管道导入到新版本数据库中。命令如下：

```
shell> mysqldump --opt db_name | mysql -h  hostname -P port -u user -p passwd
db_name
```

这里的--opt 选项表明采用优化（Optimize）方式进行导出。

> 注意：如果网络较慢，可以在导出选项中加上--compress 来减少网络传输。

对于不支持管道操作符（|）的操作系统，可以先用 mysqldump 工具将旧版本的数据导出为文本文件，然后再往新版本 MySQL 中导入此文件。其实就是把上面操作分为两步执行，具体操作如下：

```
shell> mysqldump --opt db_name > filename（旧版本 MySQL 上执行）
shell> mysql  -u user  -p passwd db_name < filename（新版本 MySQL 上执行）
```

（4）将旧版本 MySQL 中的 mysql 数据库目录全部 cp 过来覆盖新版本 MySQL 中的 mysql 数据库。例如将/home/mysql_old/data/mysql 目录覆盖掉 /home/mysql_new/data/mysql，可以使用如下命令。

```
shell>cp  -R /home/mysql_old/data/mysql /home/mysql_new/data
```

这里，-R 选项表示 cp 整个目录下内容，包括嵌套的所有子目录。

（5）新版本服务器的 shell 里面执行 mysql_fix_privilege_tables 命令升级权限表。

```
shell>mysql_fix_privilege_tables
```

（6）重启新版本 MySQL 服务。

至此，升级完毕。

方法二：适合于任何存储引擎，速度较快。

（1）参照方法一中的步骤（1）安装新版本 MySQL。

（2）在旧版本 MySQL 中，创建用来保存输出文件的目录并用 mysqldump 备份数据库：

```
shell> mkdir DUMPDIR
shell>mysqldump --tab=DUMPDIR db_name
```

这里使用--tab 选项不会生成 SQL 文本。而是在备份目录下对每个表分别生成了.sql 和.txt 文件，其中.sql 保存了表的创建语句；.txt 保存了用默认分隔符生成的纯数据文本。

（3）将 DUMPDIR 目录中的文件转移到目标服务器上相应的目录中并将文件装载到新

版本的 MySQL 中，具体操作如下（以下命令都在新版本服务器中执行）：

```
shell> mysqladmin create db_name          # 创建数据库
shell> cat DUMPDIR/*.sql | mysql db_name  # 创建数据库表
shell> mysqlimport db_name DUMPDIR/*.txt  # 加载数据
```

（4）参照方法一中的步骤（4）、（5）、（6）升级权限表，并重启 MySQL 服务。

方法三：适合于 MyISAM 存储引擎的表，速度最快。

（1）参照方法一中的步骤（1）安装新数据库。

（2）将旧版本 MySQL 中的数据目录下的所有文件（.frm、.MYD 和.MYI）cp 到新版本 MySQL 下的相应目录下。

（3）参照方法一中的步骤（4）、（5）、（6）升级权限表，并重启 MySQL 服务。

从上面 3 种方法中可以看出，其实升级的步骤都大同小异，区别仅仅在于数据迁移方法的不同。这里需要提醒读者的有两点：

- 上面的升级方法都是假设升级期间旧版本 MySQL 不再进行数据更新，否则，迁移过去的数据将不能保证和原数据库一致。
- 迁移前后的数据库字符集最好能保持一致，否则可能会出现各种各样的乱码问题。

24.4 MySQL 降级

MySQL 一般很少降级使用（降到低版本）。对于 MyISAM 存储引擎，要想实现 MySQL 降级，可以直接将数据文件 cp 到低版本数据库上的数据目录下。如果发生表格式冲突，或者是其他存储引擎的表，则可以先使用 mysqldump 命令导出文本后再将其导入低版本的数库。

24.5 小结

本章主要对 Linux/UNIX 平台下 MySQL 的 3 种安装包（即 RPM、二进制和源码）进行了比较，并详细介绍了每一种安装包的安装步骤。尤其对于源码包的安装，还介绍了一些常用的编译选项，用户可以通过这些选项灵活进行安装定制。在本章的最后还介绍了 MySQL 的升级和降级，重点介绍了几种不同的升级方法，用户可以根据实际需求进行选择。

第25章　MySQL 中的常用工具

在 MySQL 的日常工作和管理中，用户经常会用到 MySQL 提供的各种管理工具，比如对象查看、数据备份、日志分析等，熟练使用这些工具将会大大提高工作效率。本章将给大家介绍这些常用工具的使用，因为这些工具一般都有很多的选项参数，限于篇幅限制，不可能一一详细介绍，只选择最常用的选项。如果读者希望了解更多的选项，可以参考相应工具的帮助文档。

25.1 mysql（客户端连接工具）

在 MySQL 提供的工具中，DBA（数据库管理员）使用最频繁的莫过于 mysql。这里的“mysql”不是指 MySQL 服务，也不是指 mysql 数据库，而是连接数据库的客户端工具。类似于 Oracle 数据库里的 sqlplus，是操作者和数据库之间的纽带和桥梁。

在前面章节的例子中，曾多次使用过 mysql 进行数据库的连接，大多数情况下，它的使用都非常简单，语法如下：

```
mysql [OPTIONS] [database]
```

这里的 OPTIONS 表示 mysql 的可用选项，可以一次写一个或者多个，甚至可以不写；database 表示连接的数据库，一次只能写一个或者不写，如果不写，连接成功后需要用“use dbname”命令来进入要操作的数据库。

下面介绍 mysql 的一些常用选项，这些选项通常有两种表达方式，一种是“-”+选项单词的缩写字符+选项值；另外一种是“--”+选项的完整单词+“=”+选项的实际值。例如，下面两种写法是完全等价的：

- mysql --uroot
- mysql --user=root

在下面的介绍中，如果有两种表达方式，都会用逗号隔开进行列出；否则将只显示一种表达方式。要了解更多的选项，读者可以用 mysql --help 命令进行查看。

25.1.1 连接选项

```
-u, --user=name     指定用户名
-p, --password[=name]  指定密码
-h, --host=name     指定服务器 IP 或者域名
-P, --port=#  指定连接端口
```

这 4 个选项经常一起配合使用。默认情况下，如果这些选项都不写，mysql 将会使用'用户'@'localhost'和空密码连接本机（localhost）上的 3306 端口。空用户在 MySQL 刚刚安装完毕后会自动生成，这也就是我们只使用一个“mysql”命令就可以连接到数据库的原因。如下例中，使用“mysql”命令就可以直接到数据库。

```
[root@localhost mysql]# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 71
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql>
```

可以查看一下当前的连接用户：

```
mysql> select current_user();
+----------------+
| current_user() |
+----------------+
| @localhost     |
+----------------+
```

```
1 row in set (0.04 sec)
```

果然，系统使用了空用户进行了连接。大家可能会想，如果删除了空用户，那么用单纯的“mysql”命令是不是就永远不能登录了呢？不是的，如果空用户被删除，mysql 会接着去 my.cnf 里面去找[client]组内的用户名和密码，如果有则按照此用户名和密码进行登录；如果没有记录此选项，则系统会使用'root'@'localhost'用户进行登录。来看下面的例子，my.cnf 中有用户 z1，密码为空：

```
[root@localhost mysql]# more /etc/my.cnf
[mysqld]
default-character-set=gbk
log-bin
datadir=/home/zzx
log
log-slow-queries
innodb_data_file_path=ibdata1:58M;ibdata2:20M:autoextend
[mysql]
default-character-set=gbk
[client]
user=z1
password=
```

使用 mysql 命令直接登录后查看当前用户：

```
[root@localhost mysql]# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 71
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> select  current_user();
+----------------+
| current_user() |
+----------------+
| z1@localhost   |
+----------------+
1 row in set (0.00 sec)
```

果然，此时的默认登录用户换成了 z1@localhost。这里将[client]中选项注释后，重启服务器，再次用 mysql 命令直接登录：

```
[root@localhost mysql]# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.41-community MySQL Community Edition (GPL)

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

mysql> select current_user();
```

```
+----------------+
| current_user() |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.00 sec)
```

正如我们所料，此时的登录用户已经变成了'root'@'localhost'。

如果客户端和服务器位于同一台机器上，通常不需要指定-h 选项，否则要指定 MySQL 服务所在的 IP 或者主机名。如果不指定端口，默认连接到 3306 端口。以下是一个远程用户用 root 帐号成功连接到服务器 192.168.7.55 上 3306 端口的例子：

```
C:\mysql\bin>mysql -h 192.168.7.55  -P 3306 -uroot -p
Enter password: **********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8 to server version: 4.1.13-standard-log

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

mysql>
```

> 注意：在正式的生产环境中，为了安全起见，一般需要创建应用账号并赋予适当权限，而不会用 root 直接操纵数据库；默认端口（3306）一般不要使用，可以改为任意操作系统未占用的端口。

25.1.2 客户端字符集选项

```
--default-character-set=charset-name
```

细心的读者可能发现，作为服务器的字符集选项，这个选项也可以配置在 my.cnf 的[mysqld]组中。同样，作为客户端字符集选项，也可以配置在 my.cnf 的[mysql]组中，这样每次用 mysql 工具连接数据库的时候就会自动使用此客户端字符集。当然，也可以在 mysql 的命令行中手工指定客户端字符集，如下所示：

```
shell>mysql  -u user --default-character-set=charset
```

相当于在 mysql 客户端连接成功后执行：

```
set names charset;
```

下例描述了此选项使用前后客户端字符集的变化。

（1）正常连接到 MySQL 服务后的客户端字符集（粗体显示）。

```
[zzx@localhost ~]$ mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show variables like 'chara%';
+--------------------------+----------------------------+
| Variable_name            | Value                      |
```

```
+--------------------------+----------------------------+
| character_set_client     | gbk                        |
| character_set_connection | gbk                        |
| character_set_database   | gbk                        |
| character_set_filesystem | binary                     |
| character_set_results    | gbk                        |
| character_set_server     | gbk                        |
| character_set_system     | utf8                       |
| character_sets_dir       | /usr/share/mysql/charsets/ |
+--------------------------+----------------------------+
8 rows in set (0.00 sec)
```

（2）加上参数重新连接后再次观察客户端字符集（粗体显示）。

```
[zzx@localhost ~]$ mysql -uroot --default-character-set=utf8
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show variables like 'chara%';
+--------------------------+----------------------------+
| Variable_name            | Value                      |
+--------------------------+----------------------------+
| character_set_client     | utf8                       |
| character_set_connection | utf8                       |
| character_set_database   | gbk                        |
| character_set_filesystem | binary                     |
| character_set_results    | utf8                       |
| character_set_server     | gbk                        |
| character_set_system     | utf8                       |
| character_sets_dir       | /usr/share/mysql/charsets/ |
+--------------------------+----------------------------+
8 rows in set (0.00 sec)
```

果然，系统的客户端字符集按照之前的设置发生了变化。

25.1.3 执行选项

```
-e, --execute=name          执行 SQL 语句并退出
```

此选项可以直接在 MySQL 客户端执行 SQL 语句，而不用连接到 MySQL 数据库后再执行，对于一些批处理脚本，这种方式尤其方便。下面的例子从客户端直接查询 mysql 数据库中 user 表中的 User 和 Host 字段：

```
[zzx@localhost ~]$ mysql -u root -p mysql -e "SELECT User, Host FROM user"
Enter password:
+------+----------------------+
| User | Host                 |
```

```
+------+----------------------+
| z1   | %                    |
| z2   | %                    |
| z3   | %                    |
| root | 127.0.0.1            |
| z1   | 192.168              |
| z1   | 1921168              |
| root | localhost            |
| z1   | localhost            |
| z10  | localhost            |
| z3   | localhost            |
| z4   | localhost            |
| root | localhost.localdomain |
| z1   | test_hostname        |
```

可以按这种方式连续执行多个 SQL 语句，用英文分号（;）隔开，下面例子中连续执行了两个 SQL 语句：

```
shell> mysql -u root -p -e "SELECT Name FROM Country WHERE Name LIKE 'AU%';SELECT COUNT(*) FROM City" world
Enter password: ******
+-----------+
| Name      |
+-----------+
| Australia |
| Austria   |
+-----------+
+----------+
| COUNT(*) |
+----------+
|     4079 |
```

25.1.4 格式化选项

```
-E , --vertical     将输出方式按照字段顺序竖着显示
-s , --silent       去掉 mysql 中的线条框显示
```

“-E”选项类似于 mysql 里面执行 SQL 语句后加“\G”，将输出内容比较多的行能够更清晰完整的进行显示，经常和“-e”选项一起使用。下例中在 shell 命令行直接对数据库做查询，并将结果格式化输出：

```
[zzx@localhost ~]$ mysql -u root -p mysql -e "SELECT User, Host FROM user"  -E
Enter password:
*************************** 1. row ***************************
User: z1
Host: %
*************************** 2. row ***************************
User: z2
Host: %
```

```
*************************** 3. row ***************************
User: z3
Host: %
```

在 mysql 的安静模式下，“-s”选项可以将输出中讨厌的线条框去掉，字段之间用 tab 进行分隔，没条记录显示一行。此选项对于只显示数据的情况很有用，下例中是此选项的显示结果：

```
[zzx@localhost ~]$ mysql -s -uroot test
mysql> select * from emp;
id      name    content
1       z1      aa
2       z1      aa
3       z1      aa
4       z1      aa
```

25.1.5 错误处理选项

```
-f, --force             强制执行 SQL
-v, --verbose           显示更多信息
--show-warnings         显示警告信息
```

在一个批量执行的 SQL 中，如果有其中一个 SQL 执行出错，正常情况下，该批处理将停止退出。加上-f 选项，则跳过出错 SQL，强制执行后面 SQL；加上-v 选项，则显示出错的 SQL 语句；加上--show-warnings，则会显示全部错误信息。

这 3 个参数经常一起使用，在很多情况下会对用户很有帮助，比如加载数据。如果数据中有语法错误的地方，则会将出错信息记录在日志中，而不会停止使得后面的正常 SQL 无法执行；而出错的语句，也可以在日志中得以查看，进行修复。下面是一个例子。

（1）设置测试 SQL 文本 a.sql 和测试表 t2，a.sql 中记录了 3 条 insert 语句，t2 为只有一个 int 类型字段的空表。

```
[zzx@localhost ~]$ more a.sql
insert into t2 values(1);
insert into t2 values(2a);
insert into t2 values(3);

mysql> desc t2;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| id    | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
1 row in set (0.00 sec)

mysql> select * from t2;
Empty set (0.00 sec)
```

（2）不加任何参数将数据导入表 t2。

```
[zzx@localhost ~]$ mysql -uroot test  < a.sql
ERROR 1054 (42S22) at line 2: Unknown column '2a' in 'field list'
```

可以发现，在导入过程中出错。查看一下实际导入的记录。

```
[zzx@localhost ~]$ mysql -uroot test -e 'select * from t2'
+------+
| id   |
+------+
|    1 |
+------+
```

可以发现，由于第二条记录出现语法错误，无法插入表 t2，系统自动退出，第一条记录成功插入，第三条记录没有插入。

（3）加参数-f 重新导入。

```
[zzx@localhost ~]$ mysql -uroot test -f < a.sql
ERROR 1054 (42S22) at line 2: Unknown column '2a' in 'field list'
[zzx@localhost ~]$ mysql -uroot test -e 'select * from t2'
+------+
| id   |
+------+
|    1 |
|    1 |
|    3 |
+------+
```

可以发现，虽然导入过程依旧报错，但是出错后的记录 3 却已经成功插入。但是我们只能看到部分出错信息，无法定位出错 SQL 语句。

（4）加参数-v 后重新导入。

```
[zzx@localhost ~]$ mysql -uroot test -f -v< a.sql
--------------
insert into t2 values(1)
--------------


--------------
insert into t2 values(2a)
--------------

ERROR 1054 (42S22) at line 2: Unknown column '2a' in 'field list'
--------------
insert into t2 values(3)
--------------

[zzx@localhost ~]$ mysql -uroot test -e  'select * from t2'
+------+
| id   |
+------+
|    1 |
|    1 |
|    3 |
```

```
|    1 |
|    3 |
+------+
```

这时候发现，出错后的 SQL 依然能正确插入，并且对出错 SQL 和错误内容都进行了提示，我们可以很容易的定位并解决问题。

（5）修改测试数据，将第二条记录中的“2a”改为“2222222222222222222”，显然，后者超出了 int 数据类型的范围。

```
[zzx@localhost ~]$ more a.sql
insert into t2 values(1);
insert into t2 values(2222222222222222222);
insert into t2 values(3);
```

（6）再次将 a.sql 导入表 t2。

```
[zzx@localhost ~]$ mysql -uroot test  < a.sql
```

这次没有出现任何错误提示，此时没有设置 SQL_MODE，默认是非严格的数据校验，也就是第二条记录虽然可以插入表 t2，但是插入的数据是错误的。我们看一下 t2 的数据：

```
mysql> select * from t2;
+------------+
| id         |
+------------+
|          1 |
| 2147483647 |
|          3 |
+------------+
3 rows in set (0.00 sec
```

果然，插入的数据是 int 类型的最大值“2147483647”，而非“2222222222222222222”。但是由于只是警告，并没有停止出错 SQL 的运行，这样会导致插入错误的数据，而我们却无法得知。所以下面尝试加入-v 参数再次导入。

（7）加-v 参数，再次导入 t2。

```
[zzx@localhost ~]$ mysql -uroot -v test  < a.sql
--------------
insert into t2 values(1)
--------------


--------------
insert into t2 values(2222222222222222222)
--------------


--------------
insert into t2 values(3)
```

结果显示了所有 SQL 语句的执行情况，但是对第二条出错语句并没有任何报警提示，我们仍然无法得知数据出错。最后加上--show-warnings 选项试试。

（8）加上--show-warnings 选项再次导入表 t2。

```
[zzx@localhost ~]$ mysql -uroot -v --show-warnings test  < a.sql
--------------
```

```
insert into t2 values(1)
--------------


--------------
insert into t2 values(2222222222222222222)
--------------

Warning (Code 1264): Out of range value adjusted for column 'id' at row 1
--------------
insert into t2 values(3)
```

结果发现，第二条错误数据报警，提示插入的值超出了字段的范围，可以从这个日志中很容易地找到错误数据。

通过上面的测试例子，可以发现-f、-v、--show-warnings 选项在执行一些可能含有语法错误或者数据错误的批处理作业中，可以记录比较完整的日志，从而帮助用户发现并解决这些错误。

25.2 myisampack（MyISAM 表压缩工具）

myisampack 是一个表压缩工具，可以使用很高的压缩率来对 MyISAM 存储引擎的表进行压缩，使得压缩后的表占用比压缩前小得多的磁盘空间。但是压缩后的表也将成为一个只读表，不能进行 DML 操作。

此工具的用法如下：

```
shell> myisampack [options] filename
```

下面是一个使用 myisampack 工具前后的对比示例。

（1）压缩前，粗体显示表 t2 的数据文件大小为 1146894 字节。

```
[root@localhost test]# ls -ltr
total 440
-rw-rw----  1 mysql mysql      59 Aug 11 06:38 db.opt
-rw-rw----  1 mysql mysql    1024 Aug 11 06:38 dept.MYI
-rw-rw----  1 mysql mysql   10240 Aug 11 06:38 dept.MYD
-rw-rw----  1 mysql mysql    8586 Aug 11 06:38 dept.frm
-rw-rw----  1 mysql mysql    8556 Aug 17 04:21 t2.frm
-rw-rw----  1 mysql mysql    8622 Aug 27 06:53 emp.frm
-rw-rw----  1 mysql mysql    8622 Aug 27 07:13 emp1.frm
-rw-rw----  1 mysql mysql    1024 Aug 30 03:11 t2.MYI
-rw-rw----  1 mysql mysql 1146894 Aug 30 03:11 t2.MYD
```

（2）对表 t2 进行压缩。

```
[root@localhost test]# myisampack t2
Compressing t2.MYD: (163842 records)
- Calculating statistics
- Compressing file
```

```
93.46%
```

（3）压缩后 t2 的数据文件大小变为 75016 字节（粗体显示），压缩后的文件仅占原文件的 6.54%（1-93.46%）。

```
[root@localhost test]# ls -ltr
total 188
-rw-rw----  1 mysql mysql    59 Aug 11 06:38 db.opt
-rw-rw----  1 mysql mysql  1024 Aug 11 06:38 dept.MYI
-rw-rw----  1 mysql mysql 10240 Aug 11 06:38 dept.MYD
-rw-rw----  1 mysql mysql  8586 Aug 11 06:38 dept.frm
-rw-rw----  1 mysql mysql  8556 Aug 17 04:21 t2.frm
-rw-rw----  1 mysql mysql  8622 Aug 27 06:53 emp.frm
-rw-rw----  1 mysql mysql  8622 Aug 27 07:13 emp1.frm
-rw-rw----  1 mysql mysql 75016 Aug 30 03:11 t2.MYD
-rw-rw----  1 mysql mysql  1024 Aug 30 03:11 t2.MYI
```

（4）测试压缩后的表功能：往表 t2 中做插入和查询操作。

```
[root@localhost test]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 47
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> flush tables;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t2 values(1);
ERROR 1036 (HY000): Table 't2' is read only
mysql> select count(1) from t2;
+----------+
| count(1) |
+----------+
|   163842 |
+----------+
1 row in set (0.00 sec)
```

显然，查询正常，却无法进行更新操作。

25.3 mysqladmin（MySQL 管理工具）

mysqladmin 是一个执行管理操作的客户端程序。可以用它来检查服务器的配置和当前的状态，创建并删除数据库等。它的功能和 mysql 客户端非常类似，主要区别在于它更侧重于一些管理方面的功能，比如关闭数据库。

mysqladmin 的用法如下：

```
shell> mysqladmin [options] command [command-options]
      [command [command-options]] ...
```

使用方法和常用的选项和 mysql 非常类似，这里就不再赘述。这里将可以执行的命令行简单列举如下：

```
create databasename   Create a new database
debug                 Instruct server to write debug information to log
drop databasename     Delete a database and all its tables
extended-status       Gives an extended status message from the server
flush-hosts           Flush all cached hosts
flush-logs            Flush all logs
flush-status          Clear status variables
flush-tables          Flush all tables
flush-threads         Flush the thread cache
flush-privileges      Reload grant tables (same as reload)
kill id,id,...        Kill mysql threads
password new-password Change old password to new-password, MySQL 4.1 hashing.
old-password new-password Change old password to new-password in old format.

ping                  Check if mysqld is alive
processlist           Show list of active threads in server
reload                Reload grant tables
refresh               Flush all tables and close and open logfiles
shutdown              Take server down
status                Gives a short status message from the server
start-slave           Start slave
stop-slave            Stop slave
variables             Prints variables available
version               Get version info from server
```

这里简单举一个关闭数据库的例子：

```
[root@localhost test]# mysqladmin -uroot -p shutdown
Enter password:
```

25.4 mysqlbinlog（日志管理工具）

由于服务器生成的二进制日志文件以二进制格式保存，所以如果要想检查这些文件的文本格式，就会用到 mysqlbinlog 日志管理工具。

mysqlbinlog 的具体用法如下：

```
shell> mysqlbinlog [options] log-files1 log-files2...
```

option 有很多选项，常用的如下：

- -d, --database=name　　指定数据库名称，只列出指定的数据库相关操作。
- -o, --offset=#　忽略掉日志中的前 n 行命令
- -r, --result-file=name 将输出的文本格式日志输出到指定文件
- -s, --short-form　　显示简单格式，省略掉一些信息
- --set-charset=char-name：在输出为文本格式时，在文件第一行加上 set names char-name，这个选项在某些情况下装载数据时，非常有用。
- --start-datetime=name –stop-datetime=name：指定日期间隔内的所有日志
- --start-position=#　--stop-position=#：指定位置间隔内的所有日志

下面举一个例子说明这些选项的使用：

（1）创建新日志，对 mysql 和 test 数据库做不同的 DML 操作。

```
[root@localhost mysql]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> flush logs;
Query OK, 0 rows affected (0.10 sec)

mysql> use mysql
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> revoke process  on *.* from z3@'%';
Query OK, 0 rows affected (0.00 sec)

mysql> use test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> truncate table t2;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t2 values(1);
Query OK, 1 row affected (0.00 sec)
```

（2）不加任何参数，显示所有日志（粗体字显示上一步执行过的 SQL）。

```
[root@localhost mysql]# mysqlbinlog localhost-bin.000033
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
```

```
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#070830  5:04:24 server id 1  end_log_pos 98  Start: binlog v 4, server v 5.0.41-community-log created 070830  5:04:24
# Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
# at 98
#070830  5:06:09 server id 1  end_log_pos 196  Query  thread_id=2  exec_time=0 error_code=0
use mysql/*!*/;
SET TIMESTAMP=1188421569/*!*/;
SET  @@session.foreign_key_checks=1,  @@session.sql_auto_is_null=1, @@session.unique_checks=1/*!*/;
SET @@session.sql_mode=0/*!*/;
/*!\C gbk *//*!*/;
SET @@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_server=28/*!*/;
revoke process  on *.* from z3@'%'/*!*/;
# at 196
#070830  5:06:28 server id 1  end_log_pos 276  Query  thread_id=2  exec_time=0 error_code=0
use test/*!*/;
SET TIMESTAMP=1188421588/*!*/;
truncate table t2/*!*/;
# at 276
#070830  5:06:35 server id 1  end_log_pos 363  Query  thread_id=2  exec_time=0 error_code=0
SET TIMESTAMP=1188421595/*!*/;
insert into t2 values(1)/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

（3）加-d 选项，将只显示对 test 数据库的操作日志。

```
[root@localhost mysql]# mysqlbinlog localhost-bin.000033 -d test
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#070830  5:04:24 server id 1  end_log_pos 98  Start: binlog v 4, server v 5.0.41-community-log created 070830  5:04:24
# Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
# at 98
```

```
    # at 196
    #070830  5:06:28 server id 1  end_log_pos 276    Query    thread_id=2      exec_time=0
error_code=0
    use test/*!*/;
    SET TIMESTAMP=1188421588/*!*/;
    SET            @@session.foreign_key_checks=1,            @@session.sql_auto_is_null=1,
@@session.unique_checks=1/*!*/;
    SET @@session.sql_mode=0/*!*/;
    /*!\C gbk *//*!*/;
    SET
@@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_serv
er=28/*!*/;
    truncate table t2/*!*/;
    # at 276
    #070830  5:06:35 server id 1  end_log_pos 363    Query    thread_id=2      exec_time=0
error_code=0
    SET TIMESTAMP=1188421595/*!*/;
    insert into t2 values(1)/*!*/;
    DELIMITER ;
    # End of log file
    ROLLBACK /* added by mysqlbinlog */;
    /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

日志中的粗体字显示，输出中仅仅包含了对 test 数据库的操作部分。

（4）加-o 选项，忽略掉前 3 个操作，只剩下对 t2 的 insert 操作。

```
    [root@localhost mysql]# mysqlbinlog localhost-bin.000033 -o 3
    /*!40019 SET @@session.max_insert_delayed_threads=0*/;
    /*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
    DELIMITER /*!*/;
    # at 4
    #070830   5:04:24  server  id  1   end_log_pos  98      Start:  binlog  v  4,  server  v
5.0.41-community-log created 070830  5:04:24
    # Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
    # at 276
    #070830  5:06:35 server id 1  end_log_pos 363    Query    thread_id=2      exec_time=0
error_code=0
    use test/*!*/;
    SET TIMESTAMP=1188421595/*!*/;
    SET            @@session.foreign_key_checks=1,            @@session.sql_auto_is_null=1,
@@session.unique_checks=1/*!*/;
    SET @@session.sql_mode=0/*!*/;
    /*!\C gbk *//*!*/;
    SET
@@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_serv
er=28/*!*/;
```

```
insert into t2 values(1)/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

（5）加-r 选项，将上面的结果输出到文件 resultfile 中。

```
[root@localhost mysql]# mysqlbinlog localhost-bin.000033 -o 3 -r resultfile
[root@localhost mysql]# more resultfile
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
# at 4
#070830  5:04:24 server id 1  end_log_pos 98  Start: binlog v 4, server v
5.0.41-community-log created 070830  5:04:24
# Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
# at 276
#070830  5:06:35 server id 1  end_log_pos 363  Query  thread_id=2  exec_time=0
error_code=0
use test/*!*/;
SET TIMESTAMP=1188421595/*!*/;
SET  @@session.foreign_key_checks=1,  @@session.sql_auto_is_null=1,
@@session.unique_checks=1/*!*/;
SET @@session.sql_mode=0/*!*/;
/*!\C gbk *//*!*/;
SET
@@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_serv
er=28/*!*/;
insert into t2 values(1)/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

（6）结果显示内容较多，显得比较乱，加-s 选项将上面的内容进行简单显示。

```
[root@localhost mysql]# mysqlbinlog localhost-bin.000033 -o 3 -s
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
DELIMITER /*!*/;
use test/*!*/;
SET TIMESTAMP=1188421595/*!*/;
SET  @@session.foreign_key_checks=1,  @@session.sql_auto_is_null=1,
@@session.unique_checks=1/*!*/;
SET @@session.sql_mode=0/*!*/;
/*!\C gbk *//*!*/;
SET
```

```
@@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_server=28/*!*/;
    insert into t2 values(1)/*!*/;
    DELIMITER ;
    # End of log file
    ROLLBACK /* added by mysqlbinlog */;
    /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

可以发现，内容的确比上面的精简了一些，但粗体字显示的主要内容都没有少。

（7）加“--start-datetime --stop-datetime”选项显示 5:00:00～5:06:20 之间的日志。

```
    [root@localhost mysql]# mysqlbinlog localhost-bin.000033 --start-datetime='2007/08/30 05:00:00' --stop-datetime='2007/08/30 05:06:20'
    /*!40019 SET @@session.max_insert_delayed_threads=0*/;
    /*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
    DELIMITER /*!*/;
    # at 4
    #070830  5:04:24 server id 1  end_log_pos 98  Start: binlog v 4, server v 5.0.41-community-log created 070830  5:04:24
    # Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
    # at 98
    #070830  5:06:09 server id 1  end_log_pos 196  Query  thread_id=2  exec_time=0 error_code=0
    use mysql/*!*/;
    SET TIMESTAMP=1188421569/*!*/;
    SET  @@session.foreign_key_checks=1,  @@session.sql_auto_is_null=1, @@session.unique_checks=1/*!*/;
    SET @@session.sql_mode=0/*!*/;
    /*!\C gbk *//*!*/;
    SET @@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_server=28/*!*/;
    revoke process  on *.* from z3@'%'/*!*/;
    DELIMITER ;
    # End of log file
    ROLLBACK /* added by mysqlbinlog */;
    /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

开始日期和结束日期可以只写一个。如果只写开始日期，表示范围是开始日期到日志结束；如果只写结束日期，表示日志开始到指定的结束日期。

（8）--start-position=#和—stop-position=#和日期范围类似，不过可以更精确地表示范围。例如，在上面例子中，改成位置范围后如下：

```
    [root@localhost mysql]# mysqlbinlog localhost-bin.000033 --start-position=4 --stop-position=196
    /*!40019 SET @@session.max_insert_delayed_threads=0*/;
    /*!50003 SET @OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
    DELIMITER /*!*/;
```

```
    # at 4
    #070830   5:04:24  server  id  1   end_log_pos  98       Start:  binlog  v  4,  server  v
5.0.41-community-log created 070830  5:04:24
    # Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
    # at 98
    #070830  5:06:09 server id 1  end_log_pos 196   Query    thread_id=2      exec_time=0
error_code=0
    use mysql/*!*/;
    SET TIMESTAMP=1188421569/*!*/;
    SET            @@session.foreign_key_checks=1,            @@session.sql_auto_is_null=1,
@@session.unique_checks=1/*!*/;
    SET @@session.sql_mode=0/*!*/;
    /*!\C gbk *//*!*/;
    SET
@@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_serv
er=28/*!*/;
    revoke process  on *.* from z3@'%'/*!*/;
    DELIMITER ;
    # End of log file
    ROLLBACK /* added by mysqlbinlog */;
    /*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

25.5 mysqlcheck（MyISAM 表维护工具）

mysqlcheck 客户端工具可以检查和修复 MyISAM 表，还可以优化和分析表。实际上，它集成了 mysql 工具中 check、repair、analyze、optimize 的功能。

有 3 种方式可以来调用 mysqlcheck：

```
    shell> mysqlcheck[options] db_name [tables]
    shell> mysqlcheck[options] ---database DB1 [DB2 DB3...]
    shell> mysqlcheck[options] --all—database
```

option 中有以下常用选项：

- -c, --check　　检查表
- -r, --repair　　修复表
- -a, --analyze　　分析表
- -o, --optimize　　优化表

其中，默认选项是-c（检查表）。

下面对这些选项依次进行了举例说明。

（1）检查表（check）：

```
    [root@localhost mysql]# mysqlcheck -uroot -c test
    test.dept                                          OK
    test.emp                                           OK
    test.emp1                                          OK
    test.t2                                            OK
```

（2）修复表（repair）：

```
[root@localhost mysql]# mysqlcheck -uroot -r test
test.dept                                          OK
test.emp
note     : The storage engine for the table doesn't support repair
test.emp1
note     : The storage engine for the table doesn't support repair
test.t2                                            OK
```

emp 和 emp1 表的存储引擎为 InnoDB，不支持 repair。

（3）分析表（analyze）：

```
[root@localhost mysql]# mysqlcheck -uroot -a test
test.dept                                          OK
test.emp                                           OK
test.emp1                                          OK
test.t2                                            OK
```

（4）优化表（optimize）：

```
[root@localhost mysql]# mysqlcheck -uroot -o test
test.dept                                          OK
test.emp                                           OK
test.emp1                                          OK
test.t2                                            OK
```

25.6 mysqldump（数据导出工具）

mysqldump 客户端工具用来备份数据库或在不同数据库之间进行数据迁移。备份内容包含创建表或装载表的 SQL 语句。mysqldump 目前是 MySQL 中最常用的备份工具。

有 3 种方式来调用 mysqldump：

```
shell> mysqldump [options] db_name [tables] #备份单个数据库或者库中部分数据表
shell> mysqldump [options] ---database DB1 [DB2 DB3...] #备份指定的一个或者多个数据库
shell> mysqldump [options] --all—database #备份所有数据库
```

下面是 mysqldump 的一些常用选项，要查阅更详细的功能，请用“mysqldump –help”查看。

1．连接选项

```
-u , --user=name        指定用户名
–p , --password[=name]      指定密码
–h , --host=name指定服务器 IP 或者域名
–P , --port=#      指定连接端口
```

这 4 个选项经常一起配合使用，如果客户端位于服务器上，通常不需要指定 host。如果不指定端口，默认连接到 3306 端口，以下是一个远程客户端连接到服务器的例子：

```
C:\mysql\bin>mysqldump -h192.168.7.55  -P3306 -uroot -p test > test.txt
Enter password: **********
```

2．输出内容选项

```
--add-drop-database  每个数据库创建语句前加上 DROP DATABASE 语句
--add-drop-table      在每个表创建语句前加上 DROP TABLE 语句
```

这两个选项可以在导入数据库的时候不用先手工删除旧的数据库，而是会自动删除，提高导入效率，但是导入前一定要做好备份并且确认旧数据库的确已经可以删除，否则误操作将会造成数据的损失。在默认情况下，这两个参数都自动加上。

```
-n, --no-create-db      不包含数据库的创建语句
-t, --no-create-info    不包含数据表的创建语句
-d, --no-data     不包含数据
```

这 3 个选项分别表示备份文件中不包含数据库的创建语句、不包含数据表的创建语句、不包含数据，在不同的场合下，用户可以根据实际需求来进行选择。

下例中只导出表的创建语句，不包含任何其他信息：

```
[zzx@localhost ~]$ mysqldump -uroot  --compact   -d test emp >a
[zzx@localhost ~]$ more a
CREATE TABLE `emp` (
  `id` int(11) NOT NULL default '0',
  `name` varchar(200) default NULL,
  `content` text,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

3．输出格式选项

--compact 选项使得输出结果简洁，不包括默认选项中的各种注释。下例中对 test 数据库中的表 emp 进行简洁导出：

```
[zzx@localhost ~]$ mysqldump -uroot --compact test emp >a

[zzx@localhost ~]$ more a
CREATE TABLE `emp` (
  `id` int(11) NOT NULL default '0',
  `name` varchar(200) default NULL,
  `content` text,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO `emp` VALUES (1,'z1','aa'),(2,'z1','aa'),(3,'z1','aa'),(4,'z1','aa');
```

-c --complete-insert 选项使得输出文件中的 insert 语句包括字段名称，默认是不包括字段名称的。下例中对 test 数据库中的表 emp 使用此选项进行导出：

```
[root@localhost test]# mysqldump -uroot test emp -c --compact >a
[root@localhost test]# more a
CREATE TABLE `emp` (
  `id` int(11) NOT NULL default '0',
  `name` varchar(200) default NULL,
  `content` text,
  PRIMARY KEY  (`id`),
  KEY `idx_name` (`name`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
INSERT INTO `emp` (`id`, `name`, `content`) VALUES
(1,'z1','aa'),(2,'z1','aa'),(3,'z1','aa'),(4,'z1','aa');
```

-T 选项将指定数据表中的数据备份为单纯的数据文本和建表 SQL 两个文件，经常和下面几个选项一起配合使用，将数据导出为指定格式显示。下面几个选项的具体使用方法，在第 27 章“备份与恢复”中将会进行详细的讲解。

- -T,--tab=name（备份数据和建表语句）；
- --fields-terminated-by=name（域分隔符）；
- --fields-enclosed-by=name（域引用符）；
- --fields-optionally-enclosed-by=name（域可选引用符）；
- --fields-escaped-by=name（转义字符）。

在下面的例子中，将 test 数据库中的表 t2 导出为单纯的数据文本和建表 SQL 两个文件，并存放在当前路径下的 bak 子目录下。

（1）创建备份目录。

```
[mysql@localhost ~]$ mkdir bak
```

（2）将 test1 数据库下的表 t2 备份到 bak 目录下。

```
[mysql@localhost ~]$ mysqldump -uroot -p test1 t2 -T ./bak
Enter password:
```

（3）进入 bak 目录，发现生成了两个文件，一个是.sql，另一个是.txt。

```
[mysql@localhost ~]$ cd bak
[mysql@localhost bak]$ ls
t2.sql  t2.txt
```

（4）查看两个文件的内容，.sql 文件存放了建表语句，而.txt 文件存放了实际的数据。

```
[mysql@localhost bak]$ more t2.sql
-- MySQL dump 10.9
--
-- Host: localhost    Database: test1
-- ------------------------------------------------------
-- Server version       4.1.13-standard-log

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `t2`
--

DROP TABLE IF EXISTS `t2`;
CREATE TABLE `t2` (
  `id` int(11) default NULL,
  `name` varchar(10) default NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

```
/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

[mysql@localhost bak]$ more t2.txt
1       z1
2       z2
```

4．字符集选项

--default-character-set=name 选项可以设置导出的客户端字符集。系统默认的客户端字符集可以通过以下命令来查看：

```
[zzx@localhost ~]$ mysqld --verbose --help|grep 'default-character-set'|grep -v name
default-character-set             gbk
```

这个选项在导出数据库的时候非常重要，如果客户端字符集和数据库字符集不一致，数据在导出的时候就需要进行字符集转换，将数据库字符集转换为客户端字符集，经过转换后的数据很可能成为乱码或者“？”等特殊字符，使得备份文件无法恢复。

下面是一个字符集导出中文的例子。

（1）测试表 emp 字符集为 latin1，插入测试记录。

```
mysql> show create table emp \G;
*************************** 1. row ***************************
       Table: emp
Create Table: CREATE TABLE `emp` (
  `id` int(11) NOT NULL default '0',
  `name` varchar(200) default NULL,
  `content` text,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
mysql> set names latin1;
Query OK, 0 rows affected (0.00 sec)
mysql> insert into emp values(5,'中国','test');
Query OK, 1 row affected (0.05 sec)
```

（2）用默认客户端字符集导出表 emp。

```
[zzx@localhost ~]$ mysqld --verbose --help|grep 'default-character-set'|grep -v name
default-character-set             gbk
[zzx@localhost ~]$ mysqldump -uroot  --compact  test emp >a
[zzx@localhost ~]$ more a
CREATE TABLE `emp` (
  `id` int(11) NOT NULL default '0',
  `name` varchar(200) default NULL,
  `content` text,
  PRIMARY KEY  (`id`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO `emp` VALUES (1,'z1','aa'),(2,'z1','aa'),(3,'z1','aa'),(4,'z1','aa'),(5,'胆腺鹿煤','test');
```

可以发现“中国”这两个汉字已经变成了乱码。

（2）手工设置客户端字符集为 latin1，重新导出。

```
[zzx@localhost ~]$ mysqldump -uroot  --compact  --default-character-set=latin1 test emp >a
[zzx@localhost ~]$ more a
CREATE TABLE `emp` (
  `id` int(11) NOT NULL default '0',
  `name` varchar(200) default NULL,
  `content` text,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
INSERT INTO `emp` VALUES (1,'z1','aa'),(2,'z1','aa'),(3,'z1','aa'),(4,'z1','aa'),(5,'中国','test');
```

这次，中文字符可以正确地导出。

5．其他常用选项

- -F --flush-logs（备份前刷新日志）。

加上此选项后，备份前将关闭旧日志，生成新日志。使得进行恢复的时候直接从新日志开始进行重做，大大方便了恢复过程。

- -l --lock-tables（给所有表加读锁）。

可以在备份期间使用，使得数据无法被更新，从而使备份的数据保持一致性，可以配合-F 选项一起使用。

25.7 mysqlhotcopy（MyISAM 表热备份工具）

mysqlhotcopy 是一个 Perl 脚本，它使用 LOCK TABLES、FLUSH TABLES、cp 或 scp 来快速备份数据库。它是备份数据库或单个表的最快途径，其缺点是 mysqlhotcopy 只用于备份 MyISAM，而且它需要运行在 Linux/UNIX 环境中。

需要注意的是，mysqlhotcopy 是 Perl 脚本，因此需要安装 Perl 的 MySQL 数据库接口包，我们可以从 Linux 的官方 FTP（ftp.redhat.com）上下载它，以笔者的测试环境（Linux AS4，INTEL 32 位处理器）为例，从目录/pub/redhat/linux/enterprise/4/en/os/i386/SRPMS 中下载包 perl-DBD-MySQL-2.9004-3.1.src.rpm。而此包的安装又依赖于 MySQL 的开发包，因此还需要下载 MySQL 相应版本的开发包，笔者测试环境中下载的包名为：MySQL-devel-community-5.0.41-0.rhel4.i386.rpm。后者的安装过程和普通 rpm 一样，这里不再赘述，而前者从 FTP 站点下载的是源码包，需要重新编译后才可以安装。编译的过程简单描述如下。

（1）在操作系统上 su 到 root 用户。

（2）执行如下命令，生成.spec 文件。

```
[root@localhost ~]# rpm -i perl-DBD-MySQL-2.9004-3.1.src.rpm
warning: perl-DBD-MySQL-2.9004-3.1.src.rpm: V3 DSA signature: NOKEY, key ID db42a60e
```

（3）此时进入 cd /usr/src/redhat/SPECS，可以看到 perl-DBD-MySQL.spec。

```
[root@localhost ~]# cd /usr/src/redhat/SPECS
```

```
[root@localhost SPECS]# ls -l
total 32
-rw-rw-r--  1 root root 3703 Nov 27  2004 perl-DBD-MySQL.spec
-rw-rw-r--  1 root root 4369 Jun 15  2004 perl-DBI.spec
-rw-rw-r--  1 root root 7683 May 28  2003 telnet.spec
```

（4）执行如下命令，将 spec 文件编译为 RPM 安装文件。

```
[root@localhost SPECS]# rpmbuild -bb perl-DBD-MySQL.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.905
......省略中间日志
+ cd /usr/src/redhat/BUILD
+ cd DBD-mysql-2.9004
+ rm -rf /var/tmp/perl-DBD-MySQL-2.9004-root
+ exit 0
```

（5）进入 RPM 最后放置目录，一般为/usr/src/redhat/RPMS/i386。

```
[root@localhost SPECS]# cd /usr/src/redhat/RPMS/i386
[root@localhost i386]# ls -ltr
total 2068
-rw-r--r--  1 root root  29385 Mar 17  2006 telnet-server-0.17-26.i386.rpm
-rw-r--r--  1 root root 171509 Mar 17  2006 telnet-debuginfo-0.17-26.i386.rpm
-rw-r--r--  1 root root  46858 Mar 17  2006 telnet-0.17-26.i386.rpm
-rw-r--r--  1 root root 974749 Dec 29 15:42 perl-DBD-MySQL-2.9004-3.1.i386.rpm
-rw-r--r--  1 root root 855503 Dec 29 15:42 perl-DBD-MySQL-debuginfo-2.9004-3.1.i386.rpm
```

（6）安装生成的 rpm 包。

```
rpm ivh perl-DBD-MySQL-2.9004-3.1.i386.rpm
```

至此，源码包安装完毕。

mysqlhotcopy 的用法如下：

```
shell> mysqlhotcopy db_name [/path/to/new_directory]
shell> mysqlhotcopy db_name_1 ... db_name_n        /path/to/new_directory
```

下面的例子中将 mysql 数据库备份到到当前目录下的 backup 下：

```
[root@localhost mysql]# mysqlhotcopy -u root mysql ./backup/
Existing hotcopy directory renamed to './backup//mysql_old'
Locked 18 tables in 0 seconds.
Flushed      tables      (`mysql`.`columns_priv`,      `mysql`.`db`,      `mysql`.`func`,
`mysql`.`help_category`,           `mysql`.`help_keyword`,           `mysql`.`help_relation`,
`mysql`.`help_topic`,  `mysql`.`host`,  `mysql`.`proc`,  `mysql`.`procs_priv`,  `mysql`.`t1`,
`mysql`.`tables_priv`,          `mysql`.`time_zone`,          `mysql`.`time_zone_leap_second`,
`mysql`.`time_zone_name`, `mysql`.`time_zone_transition`, `mysql`.`time_zone_transition_type`,
`mysql`.`user`) in 0 seconds.
Copying 54 files...
Copying indices for 0 files...
Unlocked tables.
mysqlhotcopy copied 18 tables (54 files) in 0 seconds (0 seconds overall).
```

mysqlhotcopy 的常用选项如下。

- --allowold：如果备份路径下中含有同名备份，则将旧的备份目录 rename 为目录名

_old。

- --addtodest：如果备份路径下存在同名目录，则仅仅将新的文件加入目录。
- --noindices：不备份所有的索引文件。
- --flushlog：表被锁定后刷新日志。

这些选项的含义都很简单，读者可以下去自己测试，这里就不再举例。更多的选项，可以用 mysqlhotcopy –help 命令或者 perldoc /usr/bin/mysqlhotcopy 命令进行查询。

25.8 mysqlimport（数据导入工具）

mysqlimport 是客户端数据导入工具，用来导入 mysqldump 加-T 选项后导出的文本文件。它实际上是客户端提供了 LOAD DATA INFILEQL 语句的一个命令行接口。用法和 LOAD DATA INFILE 子句非常类似，我们在第 27 章（备份与恢复）中对 mysqlimport 和 LOAD DATA INFILE 的用法都举例进行了详细的介绍，这里就不再赘述。

mysqlimport 的基本用法如下：

```
shell> mysqlimport [options] db_name textfile1          [textfile2 ...]
```

25.9 mysqlshow（数据库对象查看工具）

mysqlshow 客户端对象查找工具，用来很快地查找存在哪些数据库、数据库中的表、表中的列或索引。和 mysql 客户端工具很类似，不过有些特性是 mysql 客户端工具所不具备的。

mysqlshow 的使用方法如下：

```
shell> mysqlshow[option] [db_name [tbl_name [col_name]]]
```

如果不加任何选项，默认情况下，会显示所有数据库。下例中显示了当前 MySQL 中的所有数据库：

```
[zzx@localhost ~]$ mysqlshow -uroot
+--------------------+
|     Databases      |
+--------------------+
| information_schema |
| backup             |
| data               |
| index              |
| mysql              |
| test               |
| test1              |
+--------------------+
```

下面是 mysqlshow 的一些常用选项。

- --count（显示数据库和表的统计信息）。

如果不指定数据库，则显示每个数据库的名称、表数量、记录数量；如果指定数据库，则显示指定数据库的每个表名、字段数量，记录数量；如果指定具体数据库中的具体表，则显示表的字段信息，如下例所示。

（1）不指定数据库：

```
[zzx@localhost mysql]$ mysqlshow -uroot --count
```

```
+--------------------+--------+-------------+
|     Databases      | Tables |  Total Rows  |
+--------------------+--------+-------------+
| information_schema |     17 |          887 |
| bak                |      0 |            0 |
| mysql              |     18 |         1685 |
| test               |      6 |          522 |
| test1              |      3 |            4 |
+--------------------+--------+-------------+
5 rows in set.
```

（2）指定数据库：

```
[zzx@localhost mysql]$ mysqlshow -uroot test --count
Database: test
+--------+----------+------------+
| Tables | Columns  | Total Rows |
+--------+----------+------------+
| books2 |        3 |          1 |
| dept   |        2 |        512 |
| emp    |        3 |          5 |
| emp1   |        3 |          3 |
| t2     |        1 |          1 |
| users2 |        2 |          0 |
+--------+----------+------------+
6 rows in set.
```

（3）指定数据库和表：

```
[zzx@localhost mysql]$ mysqlshow -uroot test emp --count
Database: test  Table: emp  Rows: 5
+---------+-------------+-----------------+------+-----+---------+-------+----------
----------------------+---------+
| Field   | Type        | Collation       | Null | Key | Default | Extra | Privileges
| Comment |
+---------+-------------+-----------------+------+-----+---------+-------+----------
----------------------+---------+
| id      | int(11)     |                 | NO   | PRI | 0       |       |
select,insert,update,references |         |
| name    | varchar(200) | latin1_swedish_ci | YES  |     |         |       |
select,insert,update,references |         |
| content | text        | latin1_swedish_ci | YES  |     |         |       |
select,insert,update,references |         |
+---------+-------------+-----------------+------+-----+---------+-------+----------
----------------------+---------+
```

- -k –keys（显示指定表中的所有索引）。

此选项显示了两部分内容，一部分是指定表的表结构，另外一部分是指定表的当前索引信息。下例中显示了 test 库中表 emp 的表结构和当前索引信息：

```
[zzx@localhost mysql]$ mysqlshow -uroot  test emp -k
Database: test  Table: emp
+---------+-------------+-----------------+------+-----+---------+-------+----------
----------------------+---------+
| Field   | Type        | Collation       | Null | Key | Default | Extra | Privileges
| Comment |
+---------+-------------+-----------------+------+-----+---------+-------+----------
----------------------+---------+
| id      | int(11)     |                 | NO   | PRI | 0       |       |
select,insert,update,references |         |
| name    | varchar(200) | latin1_swedish_ci | YES  |     |         |       |
select,insert,update,references |         |
| content | text        | latin1_swedish_ci | YES  |     |         |       |
select,insert,update,references |         |
+---------+-------------+-----------------+------+-----+---------+-------+----------
----------------------+---------+

+-------+-----------+----------+-------------+-------------+-----------+-------------
+----------+--------+------+------------+---------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality |
Sub_part | Packed | Null | Index_type | Comment |
+-------+-----------+----------+-------------+-------------+-----------+-------------
+----------+--------+------+------------+---------+
| emp   | 0         | PRIMARY  | 1           | id          | A         | 5           |
|       |      | BTREE      |         |
+-------+-----------+----------+-------------+-------------+-----------+-------------
+----------+--------+------+------------+---------+
```

细心的读者可能发现，显示的内容实际上和在 mysql 客户端执行“show full columns from emp”和“show index from emp”的结果完全一致。

```
[zzx@localhost ~]$ mysql -uroot test -e 'show full columns from emp;show index from emp'
```

- -i –status（显示表的一些状态信息）。

下例中显示了 test 数据库中 emp 表的一些状态信息：

```
[zzx@localhost mysql]$ mysqlshow -uroot  test emp  -i
Database: test  Wildcard: emp
+------+--------+---------+------------+------+----------------+-------------+---------
--------+-------------+-----------+----------------+---------------------+-------------+---
--------+-----------------+----------+----------------+---------------------+
| Name | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length
| Index_length | Data_free | Auto_increment | Create_time         | Update_time | Check_time |
Collation         | Checksum | Create_options | Comment               |
+------+--------+---------+------------+------+----------------+-------------+---------
--------+-------------+-----------+----------------+---------------------+-------------+---
--------+-----------------+----------+----------------+---------------------+
| emp  | InnoDB | 10      | Compact    | 5    | 3276           | 16384       | 0
| 0            | 0         |                | 2007-08-30 06:47:22 |             |            |
```

```
latin1_swedish_ci |          |               | InnoDB free: 97280 kB |
    +------+--------+---------+-----------+------+---------------+------------+---------
--------+-------------+----------+---------------+--------------------+------------+---
---------+-----------------+----------+---------------+----------------------+
```

此命令和 mysql 客户端执行“show table status from test like 'emp'”的结果完全一致。

25.10 perror（错误代码查看工具）

在 MySQL 的使用过程中，可能会出现各种各样的 error，这些 error 有些是由于操作系统引起的，比如文件或者目录不存在；有些则是由于存储引擎使用不当引起的。这些 error 一般都有一个代码，类似于“error：#”或者“Errcode：#”，“#”代表具体的错误号。perror 的作用就是解释这些错误代码的详细含义。

perror 的用法很简单，如下所示：

```
    perror [OPTIONS] [ERRORCODE [ERRORCODE...]]
```

在下面例子中，可以看一下错误号 30 和 60 分别是指什么错误：

```
    [zzx@localhost mysql]$ perror 30 60
    OS error code  30:  Read-only file system
    OS error code  60:  Device not a stream
```

25.11 replace（文本替换工具）

replace 是 MySQL 自带的一个对文件中的字符串进行替换的工具，类似于 Linux 下的 sed，不过它的使用更加简单灵活。

具体使用方法如下：

```
    shell> replace from to [from to] ... -- file [file] ...
    shell> replace from to [from to] ... < file
```

其中--表示字符串结束，文件的开始，可跟多个源文件，替换完毕后会覆盖原文件。<表示后面的文件作为输入，替换后的文本显示在标准输出上，不会覆盖原文件。

下面对两种方式分别举一个例子进行说明。

- 覆盖方式（--）

（1）查看原文件 a 的内容：

```
    [zzx@localhost ~]$ more a
    a1 a2 a3
    b1 b2 b3
```

（2）将文件 a 中的 a1 和 b1 分别替换为 aa1 和 bb1：

```
    [zzx@localhost ~]$ replace a1 aa1 b1 bb1 -- a
    a converted
```

（3）查看替换后的文件：

```
    [zzx@localhost ~]$ more a
    aa1 a2 a3
    bb1 b2 b3
```

可以发现，文件 a 的内容已经是替换后的内容。

- 非覆盖方式（<）

（1）查看原文件 a 的内容：

```
[zzx@localhost ~]$ more a
aa1 a2 a3
bb1 b2 b3
```

（2）将文件 a 中的 a 和 b 分别替换为 c 和 d：

```
[zzx@localhost ~]$ replace a c b d < a
cc1 c2 c3
dd1 d2 d3
```

（3）查看替换后的文件：

```
[zzx@localhost ~]$ more a
aa1 a2 a3
bb1 b2 b3
```

可以发现，文件 a 的内容并没有改变，仍然是替换前的内容，替换后的内容只显示在标准输出上。

25.12 小结

本章简单了介绍了 MySQL 中的一些常用工具的作用，并举例说明了基本的使用方法。熟练使用这些工具，将会给我们工作带来很大的便利。由于这些工具参数众多，这里没有一一列举，如果要了解更详细的使用方法，读者可以参考相关的帮助文档。

第26章 MySQL 日志

在任何一种数据库中，都会有各种各样的日志，记录着数据库工作的方方面面，以帮助数据库管理员追踪数据库曾经发生过的各种事件。MySQL 也不例外，在 MySQL 中，有 4 种不同的日志，分别是错误日志、二进制日志（BINLOG 日志）、查询日志和慢查询日志，这些日志记录着数据库在不同方面的踪迹。本章将详细介绍这几种日志的作用和使用方法，希望读者能充分利用这些日志对数据库进行各种维护和调优。

26.1 错误日志

错误日志是 MySQL 中最重要的日志之一，它记录了当 mysqld 启动和停止时，以及服务器在运行过程中发生任何严重错误时的相关信息。当数据库出现任何故障导致无法正常使用时，可以首先查看此日志。

可以用--log-error[=file_name]选项来指定 mysqld（MySQL 服务器）保存错误日志文件的位置。如果没有给定 file_name 值，mysqld 使用错误日志名 host_name.err（host_name 为主机名）并默认在参数 DATADIR（数据目录）指定的目录中写入日志文件。

以下是 MySQL 正常启动和关闭的一段日志，不同的版本可能略有不同：

```
[root@localhost mysql]# more localhost.localdomain.err
```

```
    070727 04:24:46  mysqld started
    InnoDB: The first specified data file ./ibdata1 did not exist:
    InnoDB: a new database to be created!
    070727  4:24:46  InnoDB: Setting file ./ibdata1 size to 10 MB
    InnoDB: Database physically writes the file full: wait...
    070727  4:24:46  InnoDB: Log file ./ib_logfile0 did not exist: new to be created
    InnoDB: Setting log file ./ib_logfile0 size to 5 MB
    InnoDB: Database physically writes the file full: wait...
    070727  4:24:46  InnoDB: Log file ./ib_logfile1 did not exist: new to be created
    InnoDB: Setting log file ./ib_logfile1 size to 5 MB
    InnoDB: Database physically writes the file full: wait...
    InnoDB: Doublewrite buffer not found: creating new
    InnoDB: Doublewrite buffer created
    InnoDB: Creating foreign key constraint system tables
    InnoDB: Foreign key constraint system tables created
    070727  4:24:47  InnoDB: Started; log sequence number 0 0
    070727  4:24:47 [Note] /usr/sbin/mysqld: ready for connections.
    Version: '5.0.41-community'   socket: '/var/lib/mysql/mysql.sock'   port: 3306   MySQL
Community Edition (GPL)
    070727  6:20:27 [Note] /usr/sbin/mysqld: Normal shutdown

    070727  6:20:27  InnoDB: Starting shutdown...
    070727  6:20:28  InnoDB: Shutdown completed; log sequence number 0 43655
    070727  6:20:28 [Note] /usr/sbin/mysqld: Shutdown complete

    070727 06:20:28  mysqld ended
```

26.2 二进制日志

二进制日志（BINLOG）记录了所有的 DDL（数据定义语言）语句和 DML（数据操纵语言）语句，但是不包括数据查询语句。语句以“事件”的形式保存，它描述了数据的更改过程。此日志对于灾难时的数据恢复起着极其重要的作用。

26.2.1 日志的位置和格式

当用--log-bin[=file_name]选项启动时，mysqld 将包含所有更新数据的 SQL 命令写入日志文件。如果没有给出 file_name 值，默认名为主机名后面跟“-bin”。如果给出了文件名，但没有包含路径，则文件默认被写入参数 DATADIR（数据目录）指定的目录。

26.2.2 日志的读取

由于日志以二进制方式存储，不能直接读取，需要用 mysqlbinlog 工具来查看，语法如下：

```
    shell> mysqlbinlog log-file;
```

mysqlbinlog 的用法在上一章中已经详细介绍过，因此这里不再赘述。下面的例子演示了二

进制日志的读取过程。

（1）往测试表 emp 中插入两条测试记录。

```
[root@localhost mysql]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> insert into emp values(1,'z1');
Query OK, 1 row affected (0.00 sec)

mysql> insert into emp values(1,'z2');
Query OK, 1 row affected (0.00 sec)

mysql> exit
Bye
```

（2）使用 mysqlbinlog 工具进行日志查看，粗体字显示了步骤（1）中所做的操作。

```
[root@localhost mysql]# mysqlbinlog localhost-bin.000007
……
use test/*!*/;
SET TIMESTAMP=1186691584/*!*/;
SET            @@session.foreign_key_checks=1,            @@session.sql_auto_is_null=1,
@@session.unique_checks=1/*!*/;
SET @@session.sql_mode=0/*!*/;
/*!\C gbk *//*!*/;
SET
@@session.character_set_client=28,@@session.collation_connection=28,@@session.collation_serv
er=28/*!*/;
insert into emp values(1,'z1')/*!*/;
# at 191
#070810  4:33:07 server id 1  end_log_pos 284   Query   thread_id=8     exec_time=0
error_code=0
SET TIMESTAMP=1186691587/*!*/;
insert into emp values(1,'z2')/*!*/;
DELIMITER ;
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;
```

```
……
```

26.2.3 日志的删除

对于比较繁忙的 OLTP（在线事务处理）系统，由于每天生成日志量大，这些日志如果长时间不清除，将会对磁盘空间带来很大的浪费。因此，定期删除日志是 DBA 维护 MySQL 数据库的一个重要工作内容。下面将介绍几种删除日志的常见方法。

1．方法 1

执行“RESET MASTER;”命令，该命令将删除所有 BINLOG 日志，新日志编号从“000001”开始。下例中删除了当前的所有日志。

（1）查看删除前日志。

```
mysql> system ls -ltr localhost-bin*
-rw-rw----  1 mysql mysql 145 Aug 10 04:04 localhost-bin.000006
-rw-rw----  1 mysql mysql 144 Aug 10 04:04 localhost-bin.index
-rw-rw----  1 mysql mysql  98 Aug 10 04:04 localhost-bin.000009
-rw-rw----  1 mysql mysql 145 Aug 10 04:04 localhost-bin.000008
-rw-rw----  1 mysql mysql 145 Aug 10 04:04 localhost-bin.000007
```

在测试环境中，二进制日志格式为“hostname-bin.”后跟日志序号。而测试机的 hostname 设置为“localhost”，因此这里用“localhost-bin.*”来表示所有的日志。结果中的“localhost-bin.index”是日志的索引文件，记录了最大的日志序号，本例中可以将此文件忽略。

（2）用 RESET MASTER 命令进行日志删除。

```
mysql> reset master;
Query OK, 0 rows affected (0.08 sec)
```

（3）查看删除后的日志。

```
mysql> system ls -ltr localhost-bin*
-rw-rw----  1 mysql mysql 36 Aug 10 04:04 localhost-bin.index
-rw-rw----  1 mysql mysql 98 Aug 10 04:04 localhost-bin.000001
```

可以发现，以前的日志全部被清空。新日志重新从“000001”开始编号

2．方法 2

执行“PURGE MASTER LOGS TO 'mysql-bin.******'”命令，该命令将删除“******”编号之前的所有日志。下例中删除了“localhost-bin.000006”之前编号的所有日志。

（1）查看删除前日志。

```
mysql> system ls -ltr localhost-bin*
-rw-rw----  1 mysql mysql 145 Aug 10 04:00 localhost-bin.000004
-rw-rw----  1 mysql mysql  98 Aug 10 04:00 localhost-bin.000006
-rw-rw----  1 mysql mysql 145 Aug 10 04:00 localhost-bin.000005
-rw-rw----  1 mysql mysql 108 Aug 10 04:00 localhost-bin.index
```

（2）用 PURGE 命令进行删除。

```
mysql> purge master logs to 'localhost-bin.000006';
Query OK, 0 rows affected (0.04 sec)
```

（3）查看删除后日志。

```
mysql> system ls -ltr localhost-bin*
```

```
-rw-rw----  1 mysql mysql 98 Aug 10 04:00 localhost-bin.000006
-rw-rw----  1 mysql mysql 36 Aug 10 04:03 localhost-bin.index
```

从结果中发现，编号“000006”之前的所有日志都已经被删除。

3．方法 3

执行“PURGE MASTER LOGS BEFORE 'yyyy-mm-dd hh24:mi:ss'”命令，该命令将删除日期为“yyyy-mm-dd hh24:mi:ss”之前产生的所有日志。下例中删除了日期在“2007-08-10 04:07:00”之前的所有日志。

（1）查看删除前日志。

```
mysql> system ls -ltr localhost-bin*
-rw-rw----  1 mysql mysql 145 Aug 10 04:06 localhost-bin.000001
-rw-rw----  1 mysql mysql 145 Aug 10 04:06 localhost-bin.000002
-rw-rw----  1 mysql mysql 145 Aug 10 04:06 localhost-bin.000003
-rw-rw----  1 mysql mysql 145 Aug 10 04:06 localhost-bin.000004
-rw-rw----  1 mysql mysql 145 Aug 10 04:06 localhost-bin.000005
-rw-rw----  1 mysql mysql 252 Aug 10 04:07 localhost-bin.index
-rw-rw----  1 mysql mysql  98 Aug 10 04:07 localhost-bin.000007
-rw-rw----  1 mysql mysql 145 Aug 10 04:07 localhost-bin.000006
```

（2）用 PURGE 命令删除“2007-08-10 04:07:00”之前的所有日志。

```
mysql> purge master logs before '2007-08-10 04:07:00';
Query OK, 0 rows affected (0.04 sec)
```

（3）查看删除后日志，果然，系统只保留了两个指定删除日期后的日志。

```
mysql> system ls -ltr localhost-bin*
-rw-rw----  1 mysql mysql  98 Aug 10 04:07 localhost-bin.000007
-rw-rw----  1 mysql mysql 145 Aug 10 04:07 localhost-bin.000006
-rw-rw----  1 mysql mysql  72 Aug 10 04:08 localhost-bin.index
```

4．方法 4

设置参数--expire_logs_days=#，此参数的含义是设置日志的过期天数，过了指定的天数后日志将会被自动删除，这样将将有利于减少 DBA 管理日志的工作量。下例中通过手工更改系统日期来测试此参数的使用。

（1）查看删除前日志。

```
mysql> system ls -ltr localhost-bin.*
-rw-rw----  1 mysql mysql 145 Dec 25 00:00 localhost-bin.000041
-rw-rw----  1 mysql mysql 145 Dec 25 00:06 localhost-bin.000042
-rw-rw----  1 mysql mysql 144 Dec 25 00:06 localhost-bin.index
-rw-rw----  1 mysql mysql  98 Dec 25 00:06 localhost-bin.000044
-rw-rw----  1 mysql mysql 145 Dec 25 00:06 localhost-bin.000043
```

（2）在 my.cnf 的[mysqld]中加入“expire_logs_day=3”，然后重新启动 MySQL 服务。

```
[root@localhost zzx]# more /etc/my.cnf
[mysqld]
ndbcluster
log-bin
expire_logs_day=3
```

```
[root@localhost log]# service mysql restart
Shutting down MySQL.[  OK  ]
Starting MySQL[  OK  ]
```

（3）将系统时间改为 1 天以后。

```
[root@localhost mysql]# date
Tue Dec 25 01:14:19 CST 2007
[root@localhost mysql]# date -s '20071226 14:00:00'
Wed Dec 26 14:00:00 CST 2007
```

（4）用“flush logs”触发日志文件更新，这是由于没有到 3 天，所有日志将不会被删除。

```
[root@localhost mysql]# mysqladmin  flush-log
[root@localhost mysql]# ls -ltr |grep 'localhost-bin*'
-rw-rw----  1 mysql mysql      145 Dec 25 00:00 localhost-bin.000041
-rw-rw----  1 mysql mysql      145 Dec 25 00:06 localhost-bin.000042
-rw-rw----  1 mysql mysql      145 Dec 25 00:06 localhost-bin.000043
-rw-rw----  1 mysql mysql      145 Dec 26 14:00 localhost-bin.000044
-rw-rw----  1 mysql mysql      216 Dec 26 14:00 localhost-bin.index
-rw-rw----  1 mysql mysql       98 Dec 26 14:00 localhost-bin.000046
-rw-rw----  1 mysql mysql      145 Dec 26 14:00 localhost-bin.000045
```

（5）将日期改为 3 天以后，再次执行“flush logs”触发日志文件更新。

```
[root@localhost mysql]# date -s '20071228 14:00:00'
Fri Dec 28 14:00:00 CST 2007
[root@localhost mysql]# mysqladmin  flush-log
[root@localhost mysql]# ls -ltr |grep 'localhost-bin*'
-rw-rw----  1 mysql mysql      145 Dec 26 14:00 localhost-bin.000044
-rw-rw----  1 mysql mysql      145 Dec 26 14:00 localhost-bin.000045
-rw-rw----  1 mysql mysql      144 Dec 28 14:00 localhost-bin.index
-rw-rw----  1 mysql mysql       98 Dec 28 14:00 localhost-bin.000047
-rw-rw----  1 mysql mysql      145 Dec 28 14:00 localhost-bin.000046
```

从结果中可以看出，3 天前日志 localhost-bin.000041-- localhost-bin.000043 都已经被删除。

26.2.4 其他选项

二进制日志由于记录了数据的变化过程，对于数据的完整性和安全性起着非常重要的作用。因此，MySQL 还提供了一些其他参数选项来进行更小粒度的管理，具体介绍如下。

- --binlog-do-db=db_name

该选项告诉主服务器，如果当前的数据库(即 USE 选定的数据库)是 db_name，应将更新记录到二进制日志中。其他所有没有显式指定的数据库更新将被忽略，不记录在日志中。

- --binlog-ignore-db=db_name

该选项告诉主服务器，如果当前的数据库(即 USE 选定的数据库)是 db_name，不应将更新保存到二进制日志中，其他没有显式忽略的数据库都将进行记录。

如果想记录或忽略多个数据库，可以对上面两个选项分别使用多次，即对每个数据库指定相应的选项。例如，如果只想记录数据库 db1 和 db2 的日志，可以在参数文件中设置两行：

```
--binlog-do-db=db1
--binlog-do-db=db2
```

- --innodb-safe-binlog

此选项经常和--sync-binlog＝N（每写 N 次日志同步磁盘）一起配合使用，使得事务在日志中的记录更加安全。

- SET SQL_LOG_BIN=0

具有 SUPER 权限的客户端可以通过此语句禁止将自己的语句记入二进制记录。这个选项在某些环境下是有用的，但是使用时一定要小心，因为它很可能造成日志记录的不完整或者在复制环境中造成主从数据的不一致。

26.3 查询日志

查询日志记录了客户端的所有语句，而二进制日志不包含只查询数据的语句。

26.3.1 日志的位置和格式

当用--log[=file_name]或-l [file_name]选项启动 mysqld（MySQL 服务器）时，查询日志开始被记录。和其他日志一样，如果没有给定 *file_name* 的值，日志将写入参数 DATADIR（数据目录）指定的路径下，默认文件名是 *host_name*.log。

26.3.2 日志的读取

因为查询日志记录的格式是纯文本，因此可以直接进行读取。下面是一个读取查询日志的例子。

（1）首先在客户端对数据库做一些简单操作，包括查询和插入。

```
[root@localhost mysql]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from dept;
Empty set (0.00 sec)

mysql> insert into emp values(1,'z2');
Query OK, 1 row affected (0.01 sec)
```

（2）然后查看查询日志中记录的客户端的所有操作，对应的内容如下：

```
[root@localhost mysql]# more localhost.log
……
070810  6:52:02       9 Query       select * from dept
```

```
070810  6:52:27        9 Query       insert into emp values(1,'z2')
070810  6:53:50        9 Quit
```

> 注意：log 日志中记录了所有数据库的操作，对于访问频繁的系统，此日志对系统性能的影响较大，建议一般情况下关。

26.4 慢查询日志

慢查询日志记录了包含所有执行时间超过参数 long_query_time（单位：秒）所设置值的 SQL 语句的日志。获得表锁定的时间不算作执行时间。

26.4.1 文件位置和格式

当用--log-slow-queries[=file_name]选项启动 mysqld（MySQL 服务器）时，慢查询日志开始被记录。和前面几种日志一样，如果没有给定 file_name 的值，日志将写入参数 DATADIR（数据目录）指定的路径下，默认文件名是 host_name-slow.log。

26.4.2 日志的读取

和错误日志、查询日志一样，慢查询日志记录的格式也是纯文本，可以被直接读取。下例中演示了慢查询日志的设置和读取过程。

（1）首先查询一下 long_query_time 的值。

```
mysql> show variables like 'long%';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| long_query_time | 10    |
+-----------------+-------+
1 row in set (0.00 sec)
```

（2）为了方便测试，将修改慢查询时间为 2 秒。

```
mysql> set long_query_time=2;
Query OK, 0 rows affected (0.02 sec)
```

（3）依次执行下面两个查询语句。

第一个查询因为查询时间低于 2 秒而不会出现在慢查询日志中：

```
mysql> select count(1) from emp;
+----------+
| count(1) |
+----------+
|   131075 |
+----------+
1 row in set (0.00 sec)
```

第二个查询因为查询时间大于 2 秒而应该出现在慢查询日志中：

```
mysql> select count(1) from emp t1,dept t2 where t1.id=t2.id;
```

```
+----------+
| count(1) |
+----------+
| 33555200 |
+----------+
1 row in set (11.31 sec)
```

（4）查看慢查询日志。

```
[root@localhost mysql]# more localhost-slow.log
/usr/sbin/mysqld, Version: 5.0.41-community-log (MySQL Community Edition (GPL)). started
with:
Tcp port: 3306  Unix socket: /var/lib/mysql/mysql.sock
Time                 Id Command    Argument
# Time: 070810 23:43:55
# User@Host: root[root] @ localhost []
# Query_time: 297  Lock_time: 0  Rows_sent: 0  Rows_examined: 26214
use test;
# Query_time: 11  Lock_time: 0  Rows_sent: 1  Rows_examined: 512
select count(1) from emp t1,dept t2 where t1.id=t2.id;
```

从上面日志中，可以发现查询时间超过 2 秒的 SQL，而小于 2 秒的则没有出现在此日志中。如果慢查询日志中记录内容很多，可以使用 mysqldumpslow 工具（MySQL 客户端安装自带）来对慢查询日志进行分类汇总。下例中对日志文件 bj37-slow.log 进行了分类汇总，只显示汇总后摘要结果：

```
[zzx@bj37 data]$ mysqldumpslow bj37-slow.log
[root@localhost mysql]# mysqldumpslow localhost-slow.log

Reading mysql slow query log from localhost-slow.log
Count: 1  Time=297.00s (297s)  Lock=0.00s (0s)  Rows=0.0 (0), root[root]@localhost
  select count(N) from emp t1,emp t2 where t1.id<>t2.id

Count: 2  Time=11.00s (22s)  Lock=0.00s (0s)  Rows=1.0 (2), root[root]@localhost
  select count(N) from emp t1,dept t2 where t1.id=t2.id

Count: 1  Time=9.00s (9s)  Lock=0.00s (0s)  Rows=0.0 (0), root[root]@localhost
  select count(N) from emp t1,emp t2 where t1.id=t2.id

Count: 2  Time=3.00s (6s)  Lock=0.00s (0s)  Rows=1.0 (2), root[root]@localhost
  select count(N) from emp t1,dept t2 where t1.id=t2.id and t1.id=N
```

对于 SQL 文本完全一致，只是变量不同的语句，mysqldumpslow 将会自动视为同一个语句进行统计，变量值用 N 来代替。这个统计结果将大大增加用户阅读慢查询日志的效率，并迅速定位系统的 SQL 瓶颈。

> 注意：慢查询日志对于我们发现应用中有性能问题的 SQL 很有帮助，建议正常情况下，打开此日志并经常查看分析。

26.4.3 其他选项

在 MySQL 5.1 中，通过--log-slow-admin-statements 服务器选项，可以请求将慢管理语句，例如 OPTIMIZE TABLE、ANALYZE TABLE 和 ALTER TABLE 写入慢查询日志。

26.5 小结

日志是数据库中很重要的记录内容，它可以帮助我们诊断数据库出现的各种问题。本章主要介绍了 MySQL 最常用的 4 种日志类型：错误日志、二进制日志、查询日志和慢查询日志。这 4 种日志各有不同的用途。

- 系统故障时，建议首先查看错误日志，以帮助用户迅速定位故障原因。
- 如果要记录数据的变更、数据的备份、数据的复制等操作时，二进制日志必须打开，以帮助用户进行数据恢复等操作。默认不记录此日志，建议通过--log-bin 选项将此日志打开。
- 如果希望记录数据库发生的任何操作，包括 SELECT，则需要用--log 将查询日志打开，此日志默认关闭，一般情况下建议不要打开此日志，以免影响系统整体性能。
- 如果希望查看系统的性能问题，希望找到有性能问题的 SQL 语句，则需要用--log-slow-queries 打开慢查询日志。对于大量的慢查询日志，建议使用 mysqldumpslow 工具来进行汇总查看。

第27章　备份与恢复

备份和恢复在任何数据库里面都是非常重要的内容，好的备份方法和备份策略将会使得数据库中的数据更加高效和安全。和很多数据库类似，MySQL 的备份也主要分为逻辑备份和物理备份。本章将重点来介绍这两种方式的备份和恢复方法（本章例子全部在 Linux AS4、MySQL 5.0.41 平台下测试通过）。

27.1 备份/恢复策略

对于一个 DBA 来说，定制合理的备份策略无疑是很重要的，以下是我们在进行备份或恢复操作时需要考虑的一些因素。

- 确定要备份的表的存储引擎是事务型还是非事务性，两种不同的存储引擎备份方式在处理数据一致性方面是不太一样的。
- 确定使用全备份还是增量备份。全备份的优点是备份保持最新备份，恢复的时候可以花费更少的时间；缺点是如果数据量大，将会花费很多的时间，并对系统造成较长时间的压力。增量备份则恰恰相反，只需要备份每天的增量日志，备份时间少，对负载压力也小；缺点就是恢复的时候需要全备份加上次备份到故障前的所有日志，恢复时间会长些。
- 可以考虑采取复制的方法来做异地备份，但是记住，复制不能代替备份，它对数据库的误操作也无能为力。
- 要定期做备份，备份的周期要充分考虑系统可以承受的恢复时间。备份要在系统负

载较小的时候进行。

- 确保 MySQL 打开 log-bin 选项，有了 BINLOG，MySQL 才可以在必要的时候做完整恢复，或基于时间点的恢复，或基于位置的恢复。
- 要经常做备份恢复测试，确保备份是有效的，并且是可以恢复的。

27.2 逻辑备份和恢复

在 MySQL 里面，逻辑备份的最大优点是对于各种存储引擎，都可以用同样的方法来备份；而物理备份则不同，不同的存储引擎有着不同的备份方法。因此，对于不同存储引擎混合的数据库，用逻辑备份会更简单一些。本节将详细介绍逻辑备份以及相应的恢复方法。

27.2.1 备份

MySQL 中的逻辑备份是将数据库中的数据备份为一个文本文件，备份的文件可以被查看和编辑。在 MySQL 中，使用 mysqldump 工具来完成逻辑备份。有以下 3 种方法来调用 mysqldump：

- 备份指定的数据库，或者此数据库中某些表。

```
shell> mysqldump [options] db_name [tables]
```

- 备份指定的一个或多个数据库。

```
shell> mysqldump [options] ---database DB1 [DB2 DB3...]
```

- 备份所有数据库。

```
shell> mysqldump [options] --all--database
```

如果没有指定数据库中的任何表，默认导出所有数据库中所有表。以下给出一些使用 mysqldump 工具进行备份的例子。

（1）备份所有数据库：

```
[zzx@localhost ~]$ mysqldump -uroot -p --all-database > all.sql
Enter password:
```

（2）备份数据库 test：

```
[zzx@localhost ~]$ mysqldump -uroot -p test > test.sql
Enter password:
```

（3）备份数据库 test 下的表 emp：

```
[zzx@localhost ~]$ mysqldump -uroot -p test emp > emp.sql
Enter password:
```

（4）备份数据库 test 下的表 emp 和 dept：

```
[zzx@localhost ~]$ mysqldump -uroot -p test emp dept > emp_dept.sql
Enter password:
```

（5）备份数据库 test 下的所有表为逗号分割的文本，备份到/tmp：

```
[root@localhost tmp]# mysqldump -uroot -T /tmp test emp --fields-terminated-by ','
[root@localhost tmp]# more emp.txt
1,z1
2,z2
3,z3
4,z4
1,z1
```

其中 mysqldump 的选项很多，具体可以使用“--help”参数查看帮助：

```
mysqldump --help
```

需要强调的是，为了保证数据备份的一致性，MyISAM 存储引擎在备份的时候需要加上-l 参数，表示将所有表加上读锁，在备份期间，所有表将只能读而不能进行数据更新。但是对于事务存储引擎（InnoDB 和 BDB）来说，可以采用更好的选项--single-transaction，此选项将使得 InnoDB 存储引擎得到一个快照（Snapshot），使得备份的数据能够保证一致性。

27.2.2 完全恢复

mysqldump 的恢复也很简单，将备份作为输入执行即可，具体语法如下：

```
mysql -uroot -p dbname < bakfile
```

注意，将备份恢复后数据并不完整，还需要将备份后执行的日志进行重做，语法如下：

```
mysqlbinlog binlog-file | mysql -u root -p***
```

以下是一个完整的 mysqldump 备份与恢复的例子。

（1）上午 9 点，备份数据库：

```
[root@localhost mysql]# mysqldump -uroot -p -l -F test >test.dmp
Enter password:
```

其中-l 参数表示给所有表加读锁，-F 表示生成一个新的日志文件，此时，test 中 emp 表的数据如下：

```
mysql> select * from emp order by id;
+------+------+
| id   | name |
+------+------+
|    1 | z1   |
|    2 | z2   |
|    3 | z3   |
|    4 | z4   |
+------+------+
4 rows in set (0.00 sec)
```

（2）9 点半备份完毕，然后，插入新的数据：

```
mysql> insert into emp values(5,'z5');
Query OK, 1 row affected (0.04 sec)

mysql> insert into emp values(6,'z6');
Query OK, 1 row affected (0.04 sec)
```

（3）10 点，数据库突然故障，数据无法访问。需要恢复备份：

```
[root@localhost mysql]# mysql -uroot -p test < test.dmp
Enter password:
```

恢复后的数据如下：

```
mysql> select * from emp order by id;
+------+------+
| id   | name |
+------+------+
|    1 | z1   |
|    2 | z2   |
```

```
|    3 | z3   |
|    4 | z4   |
+------+------+
4 rows in set (0.00 sec)
```

（4）使用 mysqlbinlog 恢复自 mysqldump 备份以来的 BINLOG。

```
[root@localhost mysql]# mysqlbinlog  localhost-bin.000015 | mysql -u root  -p test
Enter password:
```

查询完全恢复的数据如下：

```
mysql> select * from emp order by id;
+------+------+
| id   | name |
+------+------+
|    1 | z1   |
|    2 | z2   |
|    3 | z3   |
|    4 | z4   |
|    5 | z5   |
|    6 | z6   |
+------+------+
6 rows in set (0.00 sec)
```

至此，数据库完全恢复。

27.2.3 基于时间点恢复

由于误操作，比如误删除了一张表，这时使用完全恢复是没有用的，因为日志里面还存在误操作的语句，我们需要的是恢复到误操作之前的状态，然后跳过误操作语句，再恢复后面执行的语句，完成我们的恢复。这种恢复叫不完全恢复，在 MySQL 中，不完全恢复分为基于时间点的恢复和基于位置的恢复。

以下是基于时间点恢复的操作步骤。

（1）如果上午 10 点发生了误操作，可以用以下语句用备份和 BINLOG 将数据恢复到故障前：

```
shell>mysqlbinlog --stop-date="2005-04-20 9:59:59" /var/log/mysql/bin.123456 | mysql -u root  -pmypwd
```

（2）跳过故障时的时间点，继续执行后面的 BINLOG，完成恢复。

```
shell>mysqlbinlog --start-date="2005-04-20 10:01:00" /var/log/mysql/bin.123456| mysql -u root -pmypwd \
```

27.2.4 基于位置恢复

和基于时间点的恢复类似，但是更精确，因为同一个时间点可能有很多条 SQL 语句同时执行。恢复的操作步骤如下。

（1）在 shell 下执行如下命令：

```
shell>mysqlbinlog  --start-date="2005-04-20  9:55:00"  --stop-date="2005-04-20  10:05:00" /var/log/mysql/bin.123456 > /tmp/mysql_restore.sql
```

该命令将在/tmp 目录创建小的文本文件，编辑此文件，找到出错语句前后的位置号，

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com

例如前后位置号分别是 368312 和 368315。

（2）恢复了以前的备份文件后，应从命令行输入下面内容：

```
shell>mysqlbinlog --stop-position="368312" /var/log/mysql/bin.123456 \
  | mysql -u root -pmypwd
shell>mysqlbinlog --start-position="368315" /var/log/mysql/bin.123456 \
  | mysql -u root -pmypwd \
```

上面的第一行将恢复到停止位置为止的所有事务。下一行将恢复从给定的起始位置直到二进制日志结束的所有事务。因为 mysqlbinlog 的输出包括每个 SQL 语句记录之前的 SET TIMESTAMP 语句，因此恢复的数据和相关 MySQL 日志将反应事务执行的原时间。

27.3 物理备份和恢复

物理备份又分为冷备份和热备份两种，和逻辑备份相比，它的最大优点是备份和恢复的速度更快，因为物理备份的原理都是基于文件的 cp。本节将介绍 MySQL 中的物理备份及其恢复的方法。

27.3.1 冷备份

冷备份其实就是停掉数据库服务，cp 数据文件的方法。这种方法对 MyISAM 和 InnoDB 存储引擎都适合，但是一般很少使用，因为很多应用是不允许长时间停机的。

进行备份的操作如下：停掉 MySQL 服务，在操作系统级别备份 MySQL 的数据文件和日志文件到备份目录。

进行恢复的操作如下：首先停掉 MySQL 服务，在操作系统级别恢复 MySQL 的数据文件；然后重启 MySQL 服务，使用 mysqlbinlog 工具恢复自备份以来的所有 BINLOG。

27.3.2 热备份

MySQL 中，对于不同的存储引擎热备份方法也有所不同，下面主要介绍 MyISAM 和 InnoDB 两种最常用的存储引擎的热备份方法。

1. MyISAM 存储引擎

MyISAM 存储引擎的热备份有很多方法，本质其实就是将要备份的表加读锁，然后再 cp 数据文件到备份目录。常用的有以下两种方法。

方法 1：使用 mysqlhotcopy 工具

mysqlhotcopy 是 MySQL 自带的一个热备份工具，使用方法很简单：

```
shell> mysqlhotcopy db_name [/path/to/new_directory]
```

mysqlhotcopy 有很多选项，具体可以使用“--help”查看帮助：

```
mysqlhotcopy --help
```

方法 2：手工锁表 copy

在 mysqlhotcopy 使用不正常的情况下，可以手工来做热备份，操作步骤如下：

首先数据库中所有表加读锁：

```
mysql>flush tables for read ;
```

然后 cp 数据文件到备份目录即可。

2. InnoDB 存储引擎

ibbackup 是 Innobase 公司（www.innodb.com）的一个热备份工具，专门对 InnoDB 存储引擎进行物理热备份，此工具是收费的，但可以免费使用 1 个月。Innobase 公司已经于 2005 年被 Oracle 公司所收购。

下面简单介绍一下使用 ibbackup 工具的备份步骤。

（1）编辑用于启动的配置文件 my.cnf 和用于备份的配置文件 backup-my.cnf。

```
my.cnf 的例子如下：
[mysqld]
datadir = /home/heikki/data
innodb_data_home_dir = /home/heikki/data
innodb_data_file_path = ibdata1:100M;ibdata2:200M;ibdata3:500M:autoextend
innodb_log_group_home_dir = /home/heikki/data
set-variable = innodb_log_files_in_group=2
set-variable = innodb_log_file_size=20M
```

（2）如果想备份到/home/heikki/backup，则 backup-my.cnf 的例子如下。

```
[mysqld]
datadir = /home/heikki/backup
innodb_data_home_dir = /home/heikki/backup
innodb_data_file_path = ibdata1:100M;ibdata2:200M;ibdata3:500M:autoextend
innodb_log_group_home_dir = /home/heikki/backup
set-variable = innodb_log_files_in_group=2
set-variable = innodb_log_file_size=20M
```

（3）开始备份，具体如下：

```
$ ibbackup /home/pekka/my.cnf /home/pekka/backup-my.cnf
InnoDB Hot Backup version 2.0-beta3; Copyright 2003 Innobase Oy
License A00001 is granted to Innobase Oy
Type ibbackup -license for detailed license terms, -help for help

Contents of /home/pekka/my.cnf:
innodb_data_home_dir got value /home/heikki/data
innodb_data_file_path got value ibdata1:100M;ibdata2:200M;ibdata3:500M:autoextend
datadir got value /home/heikki/data
innodb_log_group_home_dir got value /home/heikki/data
innodb_log_files_in_group got value 3
innodb_log_file_size got value 10485760

Contents of /home/pekka/backup-my.cnf:
innodb_data_home_dir got value /home/heikki/backup
innodb_data_file_path got value ibdata1:100M;ibdata2:200M;ibdata3:500M:autoextend
datadir got value /home/heikki/backup
```

```
innodb_log_group_home_dir got value /home/heikki/backup
innodb_log_files_in_group got value 3
innodb_log_file_size got value 10485760

ibbackup: Found checkpoint at lsn 0 268331310
ibbackup: Starting log scan from lsn 0 268331008
040121 17:35:46  ibbackup: Copying log…

040121 17:35:47  ibbackup: Switching to log file 2, lsn 0 272584704
040121 17:35:49  ibbackup: Log copied, lsn 0 282171935
ibbackup: We wait 10 seconds before starting copying the data files…
040121 17:35:59  ibbackup: Copying /home/heikki/data/ibdata1

040121 17:35:59  ibbackup: Switching to log file 0, lsn 0 283068416
040121 17:36:42  ibbackup: Copying /home/heikki/data/ibdata2
040121 17:38:19  ibbackup: Copying /home/heikki/data/ibdata3
ibbackup: A copied database page was modified at 0 284263243
ibbackup: Scanned log up to lsn 0 291666654
ibbackup: Was able to parse the log up to lsn 0 291666654
ibbackup: Maximum page number for a log record 3127
040121 17:42:15  ibbackup: Full backup completed!
```

注意：ibbackup 工具不会覆盖任何重名的文件，因此在新的备份开始之前，需要确保备份目录中没有重名文件，否则备份很可能会失败。

（4）备份完成后，备份目录下包含有数据文件和日志文件，如下所示：

```
$ ls -lh  /home/heikki/backup
total 824M
-rw-r—-    1 pekka    dev          22M Jan 21 17:42 ibbackup_logfile
-rw-r—-    1 pekka    dev         100M Jan 21 17:36 ibdata1
-rw-r—-    1 pekka    dev         200M Jan 21 17:38 ibdata2
-rw-r—-    1 pekka    dev         500M Jan 21 17:42 ibdata3
```

因为在 cp 数据文件时，文件内容在不断地变化，因此在不同的时间点 cp 的数据块中的数据很可能是不一致的。因此，ibbackup 在备份期间用一个日志文件 ibbackup_logfile 记录了备份期间数据的变化，在恢复的时候就可以用此日志文件对备份的数据文件进行日志重做，使得备份的数据能够保持完整性和一致性。

当主数据库出现故障时，我们需要用备份进行恢复，恢复的步骤如下。

（1）进行日志重做。如前面所述，利用下面的命令对备份数据进行日志重做。

```
shell>ibbackup --apply-log /home/pekka/backup-my.cnf
```

（2）恢复后重启数据库服务：

```
shell>./bin/mysqld_saft  --defaults-file=/home/pekka/backup-my.cnf  &
```

（3）服务重启后，利用 BINLOG 日志将备份点与故障点之间的剩余数据进行恢复。

```
mysqlbinlog  binlog-file | mysql -u root  -p***
```

ibbackup 还有一些其他的功能，比如压缩备份、不完全恢复等，这里就不再赘述。更详细的

使用方法读者可以用“ibbackup --help”命令进行查看，或者参阅官方帮助文档（http://www.innodb.com/support/documentation/innodb-hot-backup-manual/）。对于 InnoDB 和 MyISAM 混合的数据库，Innobase 公司还提供了一个开源的 Perl 脚本 innobackup，它可以将两种存储引擎的表一起进行备份，具体使用方法读者可以参阅上述链接中的文档。

27.4 表的导入导出

在数据库的日常维护中，表的导入导出是很频繁的一类操作。本节将对 MySQL 中这类操作进行详细的介绍，希望读者能够熟练掌握。

27.4.1 导出

在某些情况下，为了一些特定的目的，经常需要将表里的数据导出为某些符号分割的纯数据文本，而不是 SQL 语句。这些应用可能有以下一些：

- 用来作为 EXCEL 显示；
- 单纯为了节省备份空间；
- 为了快速的加载数据，LOAD DATA 的加载速度比普通的 SQL 加载要快 20 倍以上。

为了满足这些应用，可以使用以下两种办法来实现。

方法 1：使用 SELECT ...INTO OUTFILE ...命令来导出数据，具体语法如下。

```
    mysql> SELECT * FROM tablename INTO OUTFILE 'target_file' [option];
```

其中 option 参数可以是以下选项：

```
    FIELDS  TERMINATED BY 'string' （字段分隔符，默认为制表符'\t'）;
    FIELDS  [OPTIONALLY] ENCLOSED BY 'char'（字段引用符，如果加 OPTIONALLY 选项则只用在 char、varchar 和 text 等字符型字段上。默认不使用引用符）;
    FIELDS  ESCAPED BY 'char' （转义字符，默认为'\'）;
    LINES     STARTING BY 'string' （每行前都加此字符串，默认''）;
    LINES     TERMINATED BY 'string'（行结束符，默认为'\n'）;
```

其中 char 表示此符号只能是单个字符，string 表示可以是字符串。

例如，将 emp 表中数据导出为数据文本，其中，字段分隔符为“,”，字段引用符为“”（双引号），记录结束符为回车符，具体实现如下：

```
    mysql> select * from emp into outfile '/tmp/emp.txt' fields terminated by ',' enclosed by '"' ;
    Query OK, 5 rows affected (0.00 sec)

    mysql>
    mysql> system more /tmp/emp.txt
    "1","z1","aa"
    "2","z1","aa"
    "3","z1","aa"
    "4","z1","aa"
    "1","z1","aa"
```

发现第一列是数值型，如果不希望字段两边用引号引起，则语句改为：

```
    mysql> select * from emp into outfile '/tmp/emp.txt' fields terminated by ',' optionally
```

```
enclosed by '"' ;
    Query OK, 5 rows affected (0.00 sec)

    mysql> system more /tmp/emp.txt
    1,"z1","aa"
    2,"z1","aa"
    3,"z1","aa"
    4,"z1","aa"
    1,"z1","aa"
```

结果如我们所愿，第一列的双引号被去掉。

下面来测试一下转义字符。转义字符，顾名思义，就是由于含义模糊而需要特殊进行转换的字符，在不同的情况下，需要转义的字符是不一样的。MySQL 导出的数据中需要转义的字符主要包括以下 3 类：

- 转义字符本身；
- 字段分隔符；
- 记录分隔符。

在下面的例子中，对表 emp 中的 name 更新为含“\”，字段分隔符，记录分隔符的数据，然后导出：

```
    mysql> update emp set name='\\"##!aa' where id=1;
    Query OK, 2 rows affected (0.04 sec)
    Rows matched: 2  Changed: 2  Warnings: 0

    mysql> system rm /tmp/emp.txt
    mysql> select * from emp into outfile '/tmp/emp.txt' fields terminated by "," optionally
enclosed by '"' ;
    Query OK, 5 rows affected (0.00 sec)

    mysql> system more /tmp/emp.txt
    1,"\\\"##!aa","aa"
    2,"z1","aa"
    3,"z1","aa"
    4,"z1","aa"
    1,"\\\"##!aa","aa"
    mysql>
```

以上例子中，name 中含有转义字符本身“\”，域引用符“””，因此，在输出的数据中我们发现这两种字符前面都加上了转义字符“\”，“\#”变成了“\\\#”。继续进行测试，将 id 为 1 的 name 更新为含有字段分隔符“,”的字符串：

```
    mysql> update emp set name='\\"#,#,!aa' where id=1;
    Query OK, 2 rows affected (0.04 sec)
    Rows matched: 2  Changed: 2  Warnings: 0

    mysql> system rm /tmp/emp.txt
    mysql> select * from emp into outfile '/tmp/emp.txt' fields terminated by "," optionally
enclosed by '"' ;
```

```
Query OK, 5 rows affected (0.00 sec)

mysql> system more /tmp/emp.txt
1,"\\\"#,#,!aa",aa"
2,"z1","aa"
3,"z1","aa"
4,"z1","aa"
1,"\\\"#,#,!aa",aa"
```

> 注意：在 MySQL 客户端连接成功后，如果要执行操作系统的命令，可以用“system+操作系统命令”来进行执行。

这个时候，发现数据中的字符“,”并没有被转义，这是为什么呢？其实仔细想想就明白了，因为每个字符串的两边带有引用符“"”（双引号），所以当 MySQL 看到数据中的“,”时，由于它处在前半个引用分隔符之后，后半个引用分隔符之前，所以并没有将它作为字段分隔符，而只是作为普通的一个数据字符来对待，因而不需要转义。继续做测试，将输出文件的字段引用符去掉，这个时候，我们的预期是数据中的“,”将成为转义字符而需要加上“\”：

```
mysql> system rm /tmp/emp.txt
mysql> select * from emp into outfile '/tmp/emp.txt' fields terminated by "," ;
Query OK, 5 rows affected (0.00 sec)

mysql> system more /tmp/emp.txt
1,\\"#\,#\,!aa,aa
2,z1,aa
3,z1,aa
4,z1,aa
1,\\"#\,#\,!aa,aa
```

果然，现在的“,”前面加上了转义字符“\”。而刚才的引用符“”“”却没有被转义，因为它已经没有什么歧义，不需要被转义。

通过上面的测试，可以得出以下结论：

- 当导出命令中包含字段引用符时，数据中含有转义字符本身和字段引用符的字符需要被转义；
- 当导出命令中不包含字段引用符时，数据中含有转义字符本身和字段分隔符的字符需要被转义。

> 注意：SELECT…INTO OUTFILE…产生的输出文件如果在目标目录下有重名文件，将不会创建成功，源文件不能被自动覆盖。

方法 2：用 mysqldump 导出数据为文本。

```
mysqldump -u username -T target_dir dbname tablename [option]
```

其中 option 参数可以是以下选项：

- --fields-terminated-by=name（字段分隔符）；
- --fields-enclosed-by=name（字段引用符）；
- --fields-optionally-enclosed-by=name（字段引用符，只用在 char、varchar 和 text 等字符型字段上）；
- --fields-escaped-by=name（转义字符）；

- --lines-terminated-by=name（记录结束符）。

下面的例子中，采用 mysqldump 生成了指定分隔符分隔的文本：

```
    [root@localhost tmp]# mysqldump -uroot -T /tmp test emp --fields-terminated-by ','
--fields-optionally-enclosed-by '"'
    [root@localhost tmp]# more /tmp/emp.txt
    1,"\\\"#,#,!aa",aa"
    2,"z1","aa"
    3,"z1","aa"
    4,"z1","aa"
    1,"\\\"#,#,!aa",aa"
```

除了生成数据文件 emp.txt 之外，还生成一个 emp.sql 文件，里面记录了 emp 表的创建脚本，记录的内容如下：

```
    [root@localhost tmp]# more emp.sql
    -- MySQL dump 10.11
    --
    -- Host: localhost    Database: test
    -- ------------------------------------------------------
    -- Server version       5.0.41-community-log

    /*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
    /*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
    /*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
    /*!40101 SET NAMES utf8 */;
    /*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
    /*!40103 SET TIME_ZONE='+00:00' */;
    /*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='' */;
    /*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

    --
    -- Table structure for table `emp`
    --

    DROP TABLE IF EXISTS `emp`;
    CREATE TABLE `emp` (
      `id` int(11) default NULL,
      `name` varchar(10) default NULL,
      `content` text
    ) ENGINE=InnoDB DEFAULT CHARSET=latin1;

    /*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

    /*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
    /*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
    /*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
```

```
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2007-08-11 20:27:32
```

可以发现，除多了一个表的创建脚本文件外，mysqldump 和 SELEC...INTO OUTFILE...的选项和语法非常类似。其实，mysqldump 实际调用的就是后者提供的接口，并在其上面添加了一些新的功能而已。

27.4.2 导入

本节只讨论用 SELECT... INTO OUTFILE 或者 mysqldump 导出的纯数据文本的导入方法。和导出类似，导入也有两种不同的方法，分别是 LOAD DATA INFILE...和 mysqlimport，它们的本质是一样的，区别只是在于一个在 MySQL 内部执行，另一个在 MySQL 外部执行。

方法 1：使用“LOAD DATA INFILE...”命令。

```
mysql > LOAD DATA [LOCAL] INFILE  'filename'  INTO TABLE tablename [option]
```

option 可以是以下选项：

- FIELDS TERMINATED BY 'string'（字段分隔符，默认为制表符'\t'）；
- FIELDS [OPTIONALLY] ENCLOSED BY 'char'（字段引用符，如果加 OPTIONALLY 选项则只用在 char、varchar 和 text 等字符型字段上。默认不使用引用符）；
- FIELDS ESCAPED BY 'char'（转义字符，默认为'\'）；
- LINES STARTING BY 'string'（每行前都加此字符串，默认''）；
- LINES TERMINATED BY 'string'（行结束符，默认为'\n'）；
- IGNORE number LINES（忽略输入文件中的前 n 行数据）；
- (col_name_or_user_var,...) （按照列出的字段顺序和字段数量加载数据）；
- SET col_name = expr,... 将列做一定的数值转换后再加载。

其中 char 表示此符号只能是单个字符，string 表示可以是字符串。

FILELD 和 LINES 和前面 SELECT ...INTO OUTFILE...的含义完全相同，不同的是多了几个不同的选项，下面的例子将文件“/tmp/emp.txt”中的数据加载到表 emp 中：

```
mysql> load data infile '/tmp/emp.txt' into table emp fields terminated by ',' enclosed by
'"' ;
Query OK, 4 rows affected (0.03 sec)
Records: 4  Deleted: 0  Skipped: 0  Warnings: 0

mysql> select * from emp;
+------+------+---------+
| id   | name | content |
+------+------+---------+
|    2 | z1   | aa      |
|    3 | z1   | aa      |
|    4 | z1   | aa      |
|    1 | z1   | aa      |
+------+------+---------+
4 rows in set (0.00 sec)
```

如果不希望加载文件中的前两行，可以如下操作：

```
mysql> load data infile '/tmp/emp.txt' into table emp fields terminated by ',' enclosed by
```

```
'"' ignore 2 lines;
    Query OK, 2 rows affected (0.04 sec)
    Records: 2  Deleted: 0  Skipped: 0  Warnings: 0

    mysql> select * from emp;
    +------+------+---------+
    | id   | name | content |
    +------+------+---------+
    |    4 | z1   | aa      |
    |    1 | z1   | aa      |
    +------+------+---------+
    2 rows in set (0.00 sec)
```

此时数据只加载了两行，对比一下数据文件，可以发现的确只加载了后两行：

```
    mysql> system /tmp/emp.txt
    sh: /tmp/emp.txt: Permission denied
    mysql> system more /tmp/emp.txt
    2,"z1","aa"
    3,"z1","aa"
    4,"z1","aa"
    1,"z1","aa"
```

如果发现文件中的列顺序和表中的列顺序不符，或者只想加载部分列，可以在命令行中加上列的顺序，如下例所示：

```
    mysql> load data infile '/tmp/emp.txt' into table emp fields terminated by ',' enclosed by
'"' ignore 2 lines (id,content,name);
    Query OK, 2 rows affected (0.05 sec)
    Records: 2  Deleted: 0  Skipped: 0  Warnings: 0

    mysql> select * from emp;
    +------+------+---------+
    | id   | name | content |
    +------+------+---------+
    |    4 | aa   | z1      |
    |    1 | aa   | z1      |
    +------+------+---------+
    2 rows in set (0.00 sec)
```

可以发现，文件中第二列的内容放到了 content 里面，第三列的内容放到了 name 里面。

如果只想加载第一列，字段的列表里面可以只加第一列的名称：

```
    mysql> load data infile '/tmp/emp.txt' into table emp fields terminated by ',' enclosed by
'"' ignore 2 lines (id);
    Query OK, 2 rows affected, 2 warnings (0.04 sec)
    Records: 2  Deleted: 0  Skipped: 0  Warnings: 2

    mysql> select * from emp;
    +------+------+---------+
```

```
| id   | name | content |
+------+------+---------+
|    4 | NULL | NULL    |
|    1 | NULL | NULL    |
+------+------+---------+
2 rows in set (0.00 sec)
```

如果希望将 id 列的内容+10 后再加载到表中，可以如下操作：

```
mysql> load data infile '/tmp/emp.txt' into table emp fields terminated by ',' enclosed by
'"' set id=id+10;
Query OK, 4 rows affected (0.03 sec)
Records: 4  Deleted: 0  Skipped: 0  Warnings: 0

mysql> select * from emp;
+------+------+---------+
| id   | name | content |
+------+------+---------+
|   12 | z1   | aa      |
|   13 | z1   | aa      |
|   14 | z1   | aa      |
|   11 | z1   | aa      |
+------+------+---------+
4 rows in set (0.00 sec)
```

方法 2：用 mysqlimport 来实现，具体命令如下。

```
shell>mysqlimport -u root -p*** [--LOCAL] dbname order_tab.txt [option]
```

其中 option 参数可以是以下选项：

- --fields-terminated-by=name（字段分隔符）；
- --fields-enclosed-by=name（字段引用符）；
- --fields-optionally-enclosed-by=name（字段引用符，只用在 char、varchar 和 text 等字符型字段上）；
- --fields-escaped-by=name（转义字符）；
- --lines-terminated-by=name（记录结束符）；
- -- ignore-lines=number（或略前几行）。

这与 mysqldump 的选项几乎完全相同，这里不再详细介绍，简单来看一个例子：

```
[root@localhost tmp]# mysqlimport -uroot test /tmp/emp.txt --fields-terminated-by=','
--fields-enclosed-by='"'
test.emp: Records: 4  Deleted: 0  Skipped: 0  Warnings: 0
[root@localhost tmp]#
[root@localhost tmp]# mysql -uroot test -e 'select count(10) from emp'
+-----------+
| count(10) |
+-----------+
|         4 |
+-----------+
```

```
[root@localhost tmp]# mysql -uroot test -e 'select * from emp'
+------+------+---------+
| id   | name | content |
+------+------+---------+
|    2 | z1   | aa      |
|    3 | z1   | aa      |
|    4 | z1   | aa      |
|    1 | z1   | aa      |
+------+------+---------+
```

注意： 如果导入和导出是跨平台操作的（Windows 和 Linux），那么要注意设置参数 line-terminated-by，Windows 上设置为 line-terminated-by='\r\n'，Linux 上设置为 line-terminated-by='\n'。

27.5 小结

本章主要介绍了 MySQL 的备份和恢复方法。和其他数据库类似，MySQL 也分为逻辑备份和物理备份。两种备份方法各有优缺点，逻辑备份保存的是 SQL 文本，可以在各种条件下恢复，但是对于大数据量的系统，备份和恢复的时间都比较长；物理备份恰恰相反，由于是文件的物理 cp，备份和恢复时间都比较短，但是备份的文件在不同的平台上不一定兼容。
其中，mysqldump 是最常用的逻辑备份工具，适合各种存储引擎，希望大家重点掌握。物理备份对于不同的存储引擎，备份方式有所不同，大家主要掌握 mysqlhotcopy 和手工热备份的方法；对于 InnoDB 的热备份工具，读者可以了解一下 ibbackup 的使用方法，由于此工具是收费的，因此普及率不太高，而 MySQL 官方目前没有提供免费的 InnoDB 热备份工具。
对于数据表的导入导出方法，应重点掌握 SELECT …INTO OUTFILE 和 LOAD DATA INFILE 的使用，mysqldump 和 mysqlimport 实际上是调用了前两种方法接口，只不过是在 mysql 外部执行罢了。数据的导入导出在数据库的管理维护中使用非常频繁，而 LOAD DATA INFILE 是加载数据最快的方法，因此读者应重点掌握。

第28章　MySQL 权限与安全

MySQL 的权限系统主要用来对连接到数据库的用户进行权限的验证，以此来判断此用户是否属于合法的用户，如果是合法用户则赋予相应的数据库权限。
数据库的权限和数据库的安全是息息相关的，不当的权限设置可能会导致各种各样的安全隐患，操作系统的某些设置也会对 MySQL 的安全造成影响。
本章对 MySQL 的权限系统以及相应的安全问题进行了一些探讨，希望能够帮助读者对这些方面有深入的认识。

28.1 MySQL 权限管理

本节从 MySQL 权限系统的工作原理和账号管理两个方面来进行介绍。读完本节后，希望读者能够在了解权限系统的工作原理基础上，熟练掌握账号的管理和使用方法。

28.1.1 权限系统的工作原理

MySQL 权限系统通过下面两个阶段进行认证：

（1）对连接的用户进行身份认证，合法的用户通过认证，不合法的用户拒绝连接；

（2）对通过认证的合法用户赋予相应的权限，用户可以在这些权限范围内对数据库做相应的操作。

对于身份的认证，MySQL 是通过 IP 地址和用户名联合进行确认的，例如 MySQL 安装后默认创建的用户 root@localhost 表示用户 root 只能从本地（localhost）进行连接才可以通过认证，此用户从其他任何主机对数据库进行的连接都将被拒绝。也就是说，同样的一个用户名，如果来自不同的 IP 地址，则 MySQL 将其视为不同的用户。

MySQL 的权限表在数据库启动的时候就载入内存，当用户通过身份认证后，就在内存中进行相应权限的存取，这样，此用户就可以在数据库中做权限范围内的各种操作了。

28.1.2 权限表的存取

在权限存取的两个过程中，系统会用到“mysql”数据库（安装 MySQL 时被创建，数据库名称叫“mysql”）中 user、host 和 db 这 3 个最重要的权限表，表定义如表 28-1 所示。

表 28-1　　mysql 数据库中的 3 个权限表定义

表名	user	db	host
用户列	Host	Host	Host
	User	Db	Db
	Password	User	
权限列	Select_priv	Select_priv	Select_priv
	Insert_priv	Insert_priv	Insert_priv
	Update_priv	Update_priv	Update_priv
	Delete_priv	Delete_priv	Delete_priv
	Index_priv	Index_priv	Index_priv
	Alter_priv	Alter_priv	Alter_priv
	Create_priv	Create_priv	Create_priv
	Drop_priv	Drop_priv	Drop_priv
	Grant_priv	Grant_priv	Grant_priv
	Create_view_priv	Create_view_priv	Create_view_priv
	Show_view_priv	Show_view_priv	Show_view_priv
	Create_routine_priv	Create_routine_priv	
	Alter_routine_priv	Alter_routine_priv	
	References_priv	References_priv	References_priv
	Reload_priv		
	Shutdown_priv		
	Process_priv		

	File_priv		
	Show_db_priv		
	Super_priv		
	Create_tmp_table_priv	Create_tmp_table_priv	Create_tmp_table_priv
	Lock_tables_priv	Lock_tables_priv	Lock_tables_priv
	Execute_priv		
	Repl_slave_priv		
	Repl_client_priv		
安全列	ssl_type		
	ssl_cipher		
	x509_issuer		
	x509_subject		
资源控制列	max_questions		
	max_updates		
	max_connections		
	max_user_connections		

在这 3 个表中，最重要的表是 user 表，其次是 db 表，host 表在大多数情况下并不使用。user 中的列主要分为 4 个部分：用户列、权限列、安全列和资源控制列。

其中，通常用得最多的是用户列和权限列，其中权限列有分为普通权限和管理权限。普通权限主要用于数据库的操作，比如 select_priv、create_priv 等；而管理权限主要用来对数据库进行管理的操作，比如 process_priv、super_priv 等。

当用户进行连接的时候，权限表的存取过程有以下两个阶段。

- 先从 user 表中的 host、user 和 password 这 3 个字段中判断连接的 IP、用户名和密码是否存在于表中，如果存在，则通过身份验证，否则拒绝连接。
- 如果通过身份验证，则按照以下权限表的顺序得到数据库权限：user→db→tables_priv→columns_priv。

在这几个权限表中，权限范围依次递减，全局权限覆盖局部权限。

上面的第一阶段好理解，下面以一个例子来详细解释一下第二阶段。

（1）创建用户 z1@localhost，并赋予所有数据库上的所有表的 select 权限。

```
mysql> grant select on *.* to z1@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from user where user='z1' and host='localhost' \G;
*************************** 1. row ***************************
                 Host: localhost
                 User: z1
             Password:
          Select_priv: Y
          Insert_priv: N
          Update_priv: N
          Delete_priv: N
          Create_priv: N
            Drop_priv: N
          ......
```

（2）再来看 db 表：

```
mysql> select * from db;
Empty set (0.00 sec)
```

可以发现，user 表的的 select_priv 列是“Y”，而 db 表中并没有记录，也就是说，对所有数据库都具有相同权限的用户记录并不需要记入 db 表，而仅仅需要将 user 表中的 select_priv 改为“Y”即可。换句话说，user 表中的每个权限都代表了对所有数据库都有的权限。

（3）将 z1@localhost 上的权限改为只对 test1 数据库上所有表的 select 权限。

```
mysql> revoke select on *.* from z1@localhost;
Query OK, 0 rows affected (0.00 sec)
mysql> grant select on test1.* to z1@localhost;
Query OK, 0 rows affected (0.01 sec)
mysql> select * from user where user='z1' and host='localhost' \G;
*************************** 1. row ***************************
                 Host: localhost
                 User: z1
             Password:
          Select_priv: N
          Insert_priv: N
          Update_priv: N
          Delete_priv: N
          Create_priv: N
            Drop_priv: N
mysql> select * from db \G;
*************************** 1. row ***************************
                 Host: localhost
                   Db: test1
                 User: z1
          Select_priv: Y
          Insert_priv: N
          Update_priv: N
          Delete_priv: N
          Create_priv: N
            Drop_priv: N
           Grant_priv: N
      References_priv: N
           Index_priv: N
           Alter_priv: N
Create_tmp_table_priv: N
     Lock_tables_priv: N
     Create_view_priv: N
       Show_view_priv: N
  Create_routine_priv: N
   Alter_routine_priv: N
```

```
            Execute_priv: N
1 row in set (0.00 sec)
```

这个时候发现，user 表中的 select_priv 变为“N”，而 db 表中则增加了 db 为 test1 的一条记录。也就是说，当只授予部分数据库某些权限时，user 表中的相应权限列保持“N”，而将具体的数据库权限写入 db 表。

table 和 column 的权限机制和 db 类似，这里就不再赘述。

从上面例子可以看出，当用户通过权限认证，进行权限分配时，将按照 user→db→tables_priv→columns_priv 的顺序进行权限分配，即先检查全局权限表 user，如果 user 中对应权限为“Y”，则此用户对所有数据库的权限都为“Y”，将不再检查 db、tables_priv 和 columns_priv；如果为“N”，则到 db 表中检查此用户对应的具体数据库，并得到 db 中为“Y”的权限；如果 db 中相应权限为“N”，则检查 tables_priv 中此数据库对应的具体表，取得表中为“Y”的权限；如果 tables_priv 中相应权限为“N”，则检查 columns_priv 中此表对应的具体列，取得列中为“Y”的权限。

28.1.3 账号管理

理解了权限系统的工作原理后，本节开始介绍账号的管理。账号管理也是 DBA 日常工作中很重要的工作之一，主要包括账号的创建、权限更改和账号的删除。用户连接数据库的第一步都从账号创建开始。

1. 创建账号

有两种方法可以用来创建账号：使用 GRANT 语法创建或者直接操作授权表，但更推荐使用第一种方法，因为操作简单，出错几率更少。下面将详细讲述这两种方式的使用方法。

GRANT 的常用语法如下：

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ...
    ON [object_type] {tbl_name | * | *.* | db_name.*}
    TO user [IDENTIFIED BY [PASSWORD] 'password']
        [, user [IDENTIFIED BY [PASSWORD] 'password']] ...
 [WITH GRANT OPTION]

object_type =
    TABLE
  | FUNCTION
  | PROCEDURE
```

来看下面几个例子。

例 1：创建用户 z1，权限为可以在所有数据库上执行所有权限，只能从本地进行连接。

```
mysql> grant all privileges on *.* to z1@localhost;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from user where user='z1' and host='localhost' \G;
*************************** 1. row ***************************
                Host: localhost
                User: z1
            Password:
```

```
            Select_priv: Y
            Insert_priv: Y
            Update_priv: Y
            Delete_priv: Y
            Create_priv: Y
              Drop_priv: Y
            Reload_priv: Y
          Shutdown_priv: Y
           Process_priv: Y
              File_priv: Y
             Grant_priv: N
        References_priv: Y
             Index_priv: Y
             Alter_priv: Y
           Show_db_priv: Y
             Super_priv: Y
  Create_tmp_table_priv: Y
       Lock_tables_priv: Y
           Execute_priv: Y
        Repl_slave_priv: Y
       Repl_client_priv: Y
       Create_view_priv: Y
         Show_view_priv: Y
    Create_routine_priv: Y
     Alter_routine_priv: Y
       Create_user_priv: Y
            ......
```

可以发现，除了 Grant_priv 权限外，所有权限在 user 表里面都是“Y”。

例 2：在例 1 基础上，增加对 z1 的 grant 权限。

```
mysql> grant all privileges on *.* to z1@localhost with grant option;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from user where user='z1' and host='localhost' \G;
*************************** 1. row ***************************
                 Host: localhost
                 User: z1
             Password:
            Select_priv: Y
            Insert_priv: Y
            Update_priv: Y
            Delete_priv: Y
            Create_priv: Y
              Drop_priv: Y
            Reload_priv: Y
```

```
            Shutdown_priv: Y
             Process_priv: Y
                File_priv: Y
               Grant_priv: Y
                      ......
```

例 3：在例 2 基础上，设置密码为“123”。

```
mysql> grant all privileges on *.* to z1@localhost identified by '123' with grant option;
Query OK, 0 rows affected (0.00 sec)
```

从 user 表中查看修改的密码：

```
mysql> select * from user where user='z1' and host='localhost' \G;
*************************** 1. row ***************************
                 Host: localhost
                 User: z1
             Password: *23AE809DDACAF96AF0FD78ED04B6A265E05AA257
          Select_priv: Y
          Insert_priv: Y
          Update_priv: Y
          Delete_priv: Y
          Create_priv: Y
                      ......
```

可以发现，密码变成了一堆加密后的字符串。在 MySQL 5.0 里面，密码的算法是生成一个以*开始的 41 位的字符串，而 MySQL 4.0 之前是 16 位，因此安全性大大提高。

例 4：创建新用户 z2，可以从任何 IP 进行连接，权限为对 test1 数据库里的所有表进行 SELECT、UPDATE、INSERT 和 DELETE 操作，初始密码为“123”。

```
mysql> grant select,insert,update,delete on test1.* to 'z2'@'%' identified by '123';
Query OK, 0 rows affected (0.00 sec)
mysql> select * from user where user='z2' and host='%' \G;
*************************** 1. row ***************************
                 Host: %
                 User: z2
             Password: *23AE809DDACAF96AF0FD78ED04B6A265E05AA257
          Select_priv: N
          Insert_priv: N
          Update_priv: N
          Delete_priv: N
                      ......
mysql> select * from db where user='z2' and host='%' \G;
*************************** 1. row ***************************
                 Host: %
                   Db: test1
                 User: z2
          Select_priv: Y
          Insert_priv: Y
```

```
            Update_priv: Y
            Delete_priv: Y
                  ......
```

如上文所述，user 表中的权限都是“N”，db 表中增加的记录权限则都是“Y”。一般地，我们只授予用户适当的权限，而一般不会授予过多的权限，本例中的权限适合于大多数应用账号。

本例中的 IP 限制为所有 IP 都可以连接，因此设置为“*”，mysql 数据库中是通过 user 表的 host 字段来进行控制，host 可以是以下类型的赋值。

- Host 值可以是主机名或 IP 号，或“localhost”指出本地主机。
- 可以在 Host 列值使用通配符字符“%”和“_”。
- Host 值“%”匹配任何主机名，空 Host 值等价于“%”。它们的含义与 LIKE 操作符的模式匹配操作相同。例如，“%”的 Host 值与所有主机名匹配，而“%.mysql.com”匹配 mysql.com 域的所有主机。

表 28-2　　host 和 user 组合进行连接的例子

Host 值	User 值	被条目匹配的连接
'thomas.loc.gov'	'fred'	fred，从 thomas.loc.gov 连接
'thomas.loc.gov'	''	任何用户，从 thomas.loc.gov 连接
'%'	'fred'	fred，从任何主机连接
'%'	''	任何用户，从任何主机连接
'%.loc.gov'	'fred'	fred，从在 loc.gov 域的任何主机连接
'x.y.%'	'fred'	fred，从 x.y.net、x.y.com、x.y.edu 等联接
'144.155.166.177'	'fred'	fred，从有 144.155.166.177 IP 地址的主机连接
'144.155.166.%'	'fred'	fred，从 144.155.166 C 类子网的任何主机连接

可能大家会有这样的疑问，如果权限表中的 host 既有“thomas.loc.gov”，又有“%”，而此时，连接从主机 thomas.loc.gov 过来。显然，user 表里面这两条记录都符合匹配条件，那系统会选择哪一个呢？

如果有多个匹配，服务器必须选择使用哪个条目。按照下述原则来解决：

- 服务器在启动时读入 user 表后进行排序；
- 然后当用户试图连接时，以排序的顺序浏览条目；
- 服务器使用与客户端和用户名匹配的第一行。

当服务器读取表时，它首先以最具体的 Host 值排序。主机名和 IP 号是最具体的。“%”意味着“任何主机”并且是最不特定的。有相同 Host 值的条目首先以最具体的 User 值排序（空 User 值意味着“任何用户”并且是最不特定的）。下例是排序前和排序后的结果。

排序前：

```
 +-----------+----------+-
| Host      | User     | …
+-----------+----------+-
| %         | root     | …
| %         | jeffrey  | …
| localhost | root     | …
| localhost |          | …
```

```
+-----------+----------+-
```

排序后：

```
+-----------+----------+-
| Host      | User     | …
+-----------+----------+-
| localhost | root     | … ...
| localhost |          | … ...
| %         | jeffrey  | … ...
| %         | root     | … ...
+-----------+----------+-
```

很显然，在上面的例子中应该匹配 host 为“thomas.loc.gov”所对应的权限。

> 注意：mysql 数据库的 user 表中 host 的值为*或者空，表示所有外部 IP 都可以连接，但是不包括本地服务器 localhost，因此，如果要包括本地服务器，必须单独为 localhost 赋予权限。

例 5：授予 SUPER、PROCESS、FILE 权限给用户 z3@%。

```
mysql> grant super,process,file on *.* to 'z3'@'%';
Query OK, 0 rows affected (0.00 sec)
```

因为这几个权限都属于管理权限，因此不能够指定某个数据库，on 后面必须跟“*.*”，下面的语法将提示错误：

```
mysql> grant super,process,file on test1.* to 'z3'@'%';
ERROR 1221 (HY000): Incorrect usage of DB GRANT and GLOBAL PRIVILEGES
```

例 6：只授予登录权限给 z4@localhost。

```
mysql> grant usage on *.* to 'z4'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
[zzx@localhost ~]$ mysql -uz4
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
+--------------------+
1 row in set (0.00 sec)
```

usage 权限只能用于数据库登录，不能执行任何操作。
直接操作权限表也可以进行权限的创建，其实 GRANT 操作的本质就是修改权限表后进行权限的刷新，因此，GRANT 比操作权限表更简单，下面继续以上文的例 4 为例子来说明一下更新权限表的用法。
创建新用户 z2，可以从任何 IP 进行连接，权限为对 test1 数据库里的所有表进行 SELECT、UPDATE、INSERT 和 DELETE，初始密码为 123，用 grant 实现如下：

```
mysql> grant select,insert,update,delete on test1.* to 'z2'@'%' identified by '123';
Query OK, 0 rows affected (0.00 sec)
```

直接操作权限表如下：

```
[zzx@localhost ~]$ mysql -uz2
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 20
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
+--------------------+
1 row in set (0.00 sec)
mysql> exit
Bye
[zzx@localhost ~]$ mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> insert into db (host,db,user,select_priv,insert_priv,update_priv,delete_priv)
values('%','test1','z2','Y','Y','Y','Y');
Query OK, 1 row affected (0.00 sec)
mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
[zzx@localhost ~]$ mysql -uz2
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 22
Server version: 5.0.41-community-log MySQL Community Edition (GPL)
```

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

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| test1              |
+--------------------+
2 rows in set (0.01 sec)
```

2. 查看和更改账号权限

创建完账号后，时间长了可能就会忘记分配的权限而需要查看账号权限，也有可能会经过一段时间后需要更改以前的账号权限，下面将介绍查看和更改这两种常用操作的命令。

- 查看权限。

账号创建好后，可以通过如下命令进行查看权限：

```
show grants for user@host;
```

如下例所示：

```
mysql> show grants for z1@localhost;
+-----------------------------------------------------+
| Grants for z1@localhost                             |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'localhost'              |
| GRANT SELECT, INSERT ON `test1`.* TO 'z1'@'localhost' |
+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

host 可以不写，默认是“%”，如下所示：

```
mysql> show grants for z1;
ERROR 1141 (42000): There is no such grant defined for user 'z1' on host '%'

mysql> grant select on test1.* to 'z1'@'%';
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for z1;
+-------------------------------------+
| Grants for z1@%                     |
+-------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'%'      |
| GRANT SELECT ON `test1`.* TO 'z1'@'%' |
+-------------------------------------+
2 rows in set (0.00 sec)
```

对于 MySQL 5.0 以后的版本，也可以利用新增的 information_schema 数据库进行权限的查看：

```
mysql> use information_schema;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from SCHEMA_PRIVILEGES where grantee="'z1'@'localhost'";
+------------------+---------------+--------------+----------------+--------------+
| GRANTEE          | TABLE_CATALOG | TABLE_SCHEMA | PRIVILEGE_TYPE | IS_GRANTABLE |
+------------------+---------------+--------------+----------------+--------------+
| 'z1'@'localhost' | NULL          | test1        | SELECT         | NO           |
| 'z1'@'localhost' | NULL          | test1        | INSERT         | NO           |
+------------------+---------------+--------------+----------------+--------------+
2 rows in set (0.00 sec)
```

- 更改权限。

可以进行权限的的新增和回收。和账号创建一样，权限变更也有两种方法：使用 GRANT（新增）和 REVOKE（回收）语句，或者更改权限表。

第二种方法和前面一样，直接对 user、db、tables_priv 和 columns_priv 中的权限列进行更新即可，这里重点介绍第一种方法。

和创建账号语法完全一样，GRANT 可以直接用来对账号进行增加。其实 GRANT 语句在执行的时候，如果权限表中不存在目标账号，则创建账号；如果已经存在，则执行权限的新增。来看下面的一个例子。

（1）z2@localhost 目前只有登录权限。

```
mysql> show grants for z2@localhost;
+---------------------------------------+
| Grants for z2@localhost               |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'z2'@'localhost' |
+---------------------------------------+
1   row in set (0.00 sec)
```

（2）赋予 z2@localhost 所有数据库上的所有表的 SELECT 权限。

```
mysql> grant select on *.* to 'z2'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for z2@localhost;
+---------------------------------------+
| Grants for z2@localhost               |
+---------------------------------------+
| GRANT SELECT ON *.* TO 'z2'@'localhost' |
+---------------------------------------+
1   row in set (0.00 sec)
```

（3）继续给 z2@localhost 赋予 SELECT 和 INSERT 权限，和已有的 SELECT 权限进行合并。

```
mysql> grant select,insert on *.* to 'z2'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for z2@localhost;
+-----------------------------------------------+
| Grants for z2@localhost                       |
+-----------------------------------------------+
| GRANT SELECT, INSERT ON *.* TO 'z2'@'localhost' |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

REVOKE 语句可以回收已经赋予的权限，语法如下：

```
REVOKE priv_type [(column_list)] [, priv_type [(column_list)]] ...
    ON [object_type] {tbl_name | * | *.* | db_name.*}
    FROM user [, user] ...

REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

对于上面的例子，这里决定要收回 z2@localhost 上的 INSERT 和 SELECT 权限：

```
mysql> revoke select,insert on *.* from z2@localhost;
Query OK, 0 rows affected (0.00 sec)
mysql> show grants for z2@localhost;
+---------------------------------------+
| Grants for z2@localhost               |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'z2'@'localhost' |
+---------------------------------------+
1 row in set (0.00 sec)
```

usage 权限不能被回收，也就是说，REVOKE 用户并不能删除用户。

```
mysql> show grants for z2@localhost;
+---------------------------------------+
| Grants for z2@localhost               |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'z2'@'localhost' |
+---------------------------------------+
1 row in set (0.00 sec)

mysql> revoke usage on *.* from z2@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for z2@localhost;
+---------------------------------------+
| Grants for z2@localhost               |
+---------------------------------------+
```

```
| GRANT USAGE ON *.* TO 'z2'@'localhost' |
+----------------------------------------+
1 row in set (0.00 sec)
```

- 修改密码。

  方法 1：可以用 mysqladmin 命令在命令行指定密码。

```
shell> mysqladmin -u user_name -h host_name password "newpwd"
```

  方法 2：执行 SET PASSWORD 语句。下例中将账号'jeffrey'@'%'的密码改为'biscuit'。

```
mysql> SET PASSWORD FOR 'jeffrey'@'%' = PASSWORD('biscuit');
```

如果是更改自己的密码，可以省略 for 语句：

```
mysql> SET PASSWORD = PASSWORD('biscuit');
```

  方法 3：还可以在全局级别使用 GRANT USAGE 语句(在*.*)来指定某个账户的密码而不影响账户当前的权限。

```
mysql> GRANT USAGE ON *.* TO 'jeffrey'@'%' IDENTIFIED BY 'biscuit';
```

  方法 4：直接更改数据库的 user 表。

```
shell> mysql -u root mysql
mysql> INSERT INTO user (Host,User,Password)
    -> VALUES('%','jeffrey',PASSWORD('biscuit'));
mysql> FLUSH PRIVILEGES;

shell> mysql -u root mysql
mysql> UPDATE user SET Password = PASSWORD('bagel')
    -> WHERE Host = '%' AND User = 'francis';
mysql> FLUSH PRIVILEGES;
```

> 注意：更改密码时候一定要使用 PASSWORD 函数（mysqladmin 和 GRANT 两种方式不用写，会自动加上）。

3. 删除账号

要彻底删除账号，同样也有两种方法：DROP USER 命令和修改权限表。

DROP USER 语法非常简单，具体如下：

```
DROP USER user [, user] ...
```

举一个简单的例子，将 z2@localhost 用户删除：

```
mysql> show grants for z2@localhost ;
+----------------------------------------+
| Grants for z2@localhost                |
+----------------------------------------+
| GRANT USAGE ON *.* TO 'z2'@'localhost' |
+----------------------------------------+
1 row in set (0.00 sec)

mysql> drop user z2@localhost;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> show grants for z2@localhost ;
ERROR 1141 (42000): There is no such grant defined for user 'z2' on host 'localhost'
```

修改权限表方法只要把 user 用户中的用户记录删除即可，这里不再演示。

28.2 MySQL 安全问题

对于任何一种数据库来说，安全问题都是非常重要的。如果数据库出现安全漏洞，轻则数据被窃取，重则数据被破坏，这些后果对于一些重要的数据库都是非常严重的。本节将从操作系统和数据库两个层面对 MySQL 的安全问题进行探讨。

28.2.1 操作系统相关的安全问题

本节介绍一些常见的操作系统安全问题，这些问题主要出现在 MySQL 的安装和启动过程中，希望读者能够从安装开始就重视安全问题。

1. 严格控制操作系统账号和权限

在数据库服务器上要严格控制操作系统的账号和权限，比如：

- 锁定 mysql 用户；
- 其他任何用户都采取独立的账号登录，管理员通过 mysql 专有用户管理 MySQL，或者通过 root su 到 mysql 用户下进行管理；
- mysql 用户目录下，除了数据文件目录，其他文件和目录属主都改为 root。

2. 尽量避免以 root 权限运行 MySQL

MySQL 安装完毕后，一般会将数据目录属主设置为 mysql 用户，而将 MySQL 软件目录的属主设置为 root，这样做的目的是当使用 mysql 用户启动数据库时，可以防止任何具有 FILE 权限的用户能够用 root 创建文件。而如果使用 root 用户启动数据库，则任何具有 FILE 权限的用户都可以读写 root 用户的文件，这样会给系统造成严重的安全隐患。来看下面一个例子。

（1）MySQL 以用户 zzx 启动：

```
[root@localhost mysql]# ./bin/mysqld_safe  --user=zzx &
[1] 705
[root@localhost mysql]# Starting mysqld daemon with databases from /home/zzx/mysql/data
```

（2）将 mysql 数据库下的表 user 写到根目录/下的 user.txt：

```
[root@localhost mysql]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 5.1.11-beta-log

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

```
mysql> use mysql
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from user into outfile '/user.txt';
ERROR 1 (HY000): Can't create/write to file '/user.txt' (Errcode: 13)
```

由于是用非 root 权限启动 MySQL，则文件无法写入。

（3）将数据库关闭，然后以 root 启动。

```
[root@localhost mysql]# mysqladmin -uroot shutdown
STOPPING server from pid file /home/zzx/mysql/data/localhost.localdomain.pid
070723 12:32:49  mysqld ended

[1]+  Done                    ./bin/mysqld_safe --user=zzx
[root@localhost mysql]# ./bin/mysqld_safe  --user=root &
[1] 783
[root@localhost mysql]# Starting mysqld daemon with databases from /home/zzx/mysql/data
```

（4）重新执行将 mysql 数据库下的表 user 写到根目录/下的 user.txt：

```
mysql> select * from user into outfile '/user.txt';
Query OK, 9 rows affected (0.01 sec)
mysql> system more /user.txt
localhost       root            Y       Y       Y       Y       Y       Y       Y
Y       Y       Y       Y       Y       YY      Y       Y       Y       Y       Y       Y
Y       Y       Y       Y       Y       Y       Y       Y0      0       0       0
localhost.localdomain   root            Y       Y       Y       Y       Y       Y
Y       Y       Y       Y       Y       YY      Y       Y       Y       Y       Y       Y
Y       Y       Y       Y       Y       Y       Y       Y       Y0      0       0       0
```

此时文件成功写入根目录下。

由上例可以看出，以 root 用户启动 MySQL，任何只要有 FILE 权限的用户都可以在任意目录写出文件，这样对操作系统的安全将造成严重隐患。

3. 防止 DNS 欺骗

创建用户时，host 可以指定域名或者 IP 地址。但是，如果指定域名，就可能带来如下安全隐患：如果域名对应的 IP 地址被恶意修改，则数据库就会被恶意的 IP 地址进行访问，导致安全隐患。

在下例中，尝试改变域名对应的 IP 地址，以此来观察一下对连接的影响。

（1）创建测试用户 z1，域名指定为 test_hostname。

```
mysql> grant select on test1.* to z1@test_hostname;
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for z1@test_hostname;
+-------------------------------------------------+
```

```
| Grants for z1@test_hostname                          |
+------------------------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'test_hostname'            |
| GRANT SELECT ON `test1`.* TO 'z1'@'test_hostname'     |
+------------------------------------------------------+
```

（2）编辑 hosts 文件，增加此域名和 IP 地址的对应关系：

```
[root@localhost mysql]# vi /etc/hosts

# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1               localhost.localdomain localhost
192.168.7.55            localhost.localdomain
192.168.7.187           zzx
192.168.52.24           test_hostname
```

（3）客户端尝试连接成功：

```
C:\mysql\bin>ipconfig

Windows IP Configuration


Ethernet adapter 无线网络连接:

        Media State . . . . . . . . . . . : Media disconnected

Ethernet adapter 本地连接:

        Connection-specific DNS Suffix  . : netease.internal
        IP Address. . . . . . . . . . . . : 192.168.52.24
        Subnet Mask . . . . . . . . . . . : 255.255.254.0
        Default Gateway . . . . . . . . . : 192.168.52.254

C:\mysql\bin>mysql -h192.168.7.55 -P3311 -uz1 -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 5.1.11-beta-log

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

mysql> exit
```

（4）修改域名 IP 地址的对应关系，将 192.168.52.24 改为 192.168.52.23：

```
[root@localhost mysql]# vi /etc/hosts

# Do not remove the following line, or various programs
# that require network functionality will fail.
```

```
127.0.0.1               localhost.localdomain localhost
192.168.7.55            localhost.localdomain
192.168.7.187           zzx
28.2.1.23         test_hostname
```

（5）客户端再次尝试连接失败：

```
C:\mysql\bin>mysql -h192.168.7.55 -P3311 -uz1 -p
Enter password:
ERROR 1045 (28000): Access denied for user 'z1'@'192.168.52.24' (using
 NO)
```

28.2.2 数据库相关的安全问题

本节介绍一些常见的数据库安全问题，这些问题大多数是由于账号的管理不当造成的。希望读者读完本节后能够认识到账号管理的重要性，同时加强对账号管理的安全意识。

1. 删除匿名账号

在某些版本中，安装完毕 MySQL 后，会自动安装一个空账号，此账号具有对 test 数据库的全部权限，如下所示：

```
[zzx@bj34 zzx]$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 56064 to server version: 4.1.13-standard-log

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

mysql> use mysql
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from user \G;
……
*************************** 2. row ***************************
                Host: localhost
                User:
            Password:
          Select_priv: N
          Insert_priv: N
          Update_priv: N
          Delete_priv: N
          Create_priv: N
            Drop_priv: N
          Reload_priv: N
```

```
           Shutdown_priv: N
            Process_priv: N
               File_priv: N
              Grant_priv: N
         References_priv: N
              Index_priv: N
              Alter_priv: N
            Show_db_priv: N
              Super_priv: N
    Create_tmp_table_priv: N
         Lock_tables_priv: N
             Execute_priv: N
          Repl_slave_priv: N
         Repl_client_priv: N
                 ssl_type:
               ssl_cipher:
              x509_issuer:
             x509_subject:
            max_questions: 0
              max_updates: 0
          max_connections: 0
    ......
    mysql> select * from db \G;
    *************************** 1. row ***************************
                    Host: %
                      Db: test
                    User:
              Select_priv: Y
              Insert_priv: Y
              Update_priv: Y
              Delete_priv: Y
              Create_priv: Y
                Drop_priv: Y
              Grant_priv: N
         References_priv: Y
              Index_priv: Y
              Alter_priv: Y
    Create_tmp_table_priv: Y
         Lock_tables_priv: Y
```

普通用户只需要执行 mysql 命令即可登录 MySQL 数据库，这个时候默认使用了空用户，可以在 test 数据库里面做各种操作，比如可以创建一个大表，占用大量磁盘空间，这样将给系统造成安全隐患，如下所示：

```
    [zzx@bj34 zzx]$ mysql
    Welcome to the MySQL monitor.  Commands end with ; or \g.
```

```
Your MySQL connection id is 56073 to server version: 4.1.13-standard-log

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

mysql> use test
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> create table t1(id int);
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t1 values(1);
Query OK, 1 row affected (0.00 sec)
```

建议删除此空账号，或者对此账号加密码：

```
[zzx@bj34 zzx]$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 56074 to server version: 4.1.13-standard-log

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

mysql> drop user ''@'localhost';
Query OK, 0 rows affected (0.00 sec)
[zzx@bj34 zzx]$ mysql
'ERROR 1045 (28000): Access denied for user 'zzx'@'localhost' (using password: NO)
```

2. 给 root 账号设置口令

MySQL 安装完毕后，root 默认口令为空，需要马上修改 root 口令：

```
[root@localhost zzx]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> set password=password('newpassword');
Query OK, 0 rows affected (0.00 sec)
```

不写密码登录将被拒绝：

```
[root@localhost zzx]# mysql -uroot
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
```

3. 设置安全密码

密码的安全体现在以下两个方面：

- 设置安全的密码，建议使用 6 位以上字母、数字、下画线和一些特殊字符组合而成的字符串；
- 使用上的安全，使用密码期间尽量保证使用过程安全，不会被别人窃取。

第一点就不说了，越长越复杂越没有规律的密码越安全。对于第二点，可以总结一下，在日常工作中，使用密码一般是采用以下几种方式：

（1）直接将密码写在命令行中。

```
[root@localhost zzx]# mysql -uroot -p123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

（2）交互式方式输入密码。

```
[root@localhost zzx]# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql>
```

（3）将用户名和密码写在配置文件里面，连接的时候自动读取。比如应用连接数据库或者执行一些批处理脚本。对于这种方式，MySQL 供了一种方法，在 my.cnf 里面写入连接信息。

```
[client]
user=username
password=password
```

然后对配置文件进行严格的权限限制，例如：

```
chmod +600 my.cnf
```

以上是 3 种常见的密码使用方式。很显然，第 1 种最不安全，因为它将密码写成为明文；第 2 种比较安全，但是只能使用在交互式的界面下；第 3 种使用比较方便，但是需要将配置文件设置严格的存取权限，而且任何只要可以登录操作系统的用户都可以自动登录，存在一定的安全隐患。

第 3 种方法通常使用不多，下面举一个例子。

（1）输入 mysql 无法登录。

```
[root@localhost zzx]# mysql
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
```

（2）修改配置文件，加入连接信息。

```
[root@localhost zzx]# vi /etc/my.cnf
……
[client]
user=root
password=123
```

（3）重启数据库后，输入 mysql。

```
[root@localhost zzx]# service mysql restart
Shutting down MySQL.[  OK  ]
Starting MySQL[  OK  ]
[root@localhost zzx]# mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> select current_user();
+----------------+
| current_user() |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.00 sec)
```

4. 只授予账号必须的权限

只需要赋予普通用户必须的权限，比如：

```
Grant select,insert,update,delete on tablename to ‘username’@’hostname’;
```

在很多情况下，DBA 由于图方便，而经常赋予用户 all privileges 权限，这个 all privileges 到底具体包含哪些权限呢？来看下面的例子：

```
mysql> grant all privileges on test1.* to 'z1'@'localhost';
Query OK, 0 rows affected (0.00 sec)
mysql> select * from db where user='z1' \G;
*************************** 1. row ***************************
                 Host: localhost
                   Db: test1
                 User: z1
          Select_priv: Y
          Insert_priv: Y
          Update_priv: Y
          Delete_priv: Y
          Create_priv: Y
            Drop_priv: Y
           Grant_priv: N
```

```
      References_priv: Y
           Index_priv: Y
           Alter_priv: Y
Create_tmp_table_priv: Y
     Lock_tables_priv: Y
     Create_view_priv: Y
       Show_view_priv: Y
  Create_routine_priv: Y
   Alter_routine_priv: Y
         Execute_priv: Y
1 row in set (0.00 sec)
```

all privileges 里面的权限，远远超过了我们一般应用所需要的权限。而且，有些权限如果误操作，将会产生非常严重的后果，比如 drop_priv 等。因此，赋予用户权限的时候越具体，则对数据库越安全。

5. 除 root 外，任何用户不应有 mysql 库 user 表的存取权限

由于 MySQL 中可以通过更改 mysql 数据库的 user 表进行权限的增加、删除、变更等操作，因此，除了 root 以外，任何用户都不应该拥有对 user 表的存取权限（SELECT、UPDATE、INSERT、DELETE 等），造成系统的安全隐患。下例中对普通用户 z3 授予了 user 表的存取权限，看看会对系统产生了怎样的安全隐患。

（1）创建普通用户 z3，拥有对 mysql 数据库中 user 表的各种权限。

```
[root@localhost zzx]# mysql -uroot -pabc
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 36
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> grant select,update,insert,delete  on mysql.user to z3@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
```

（2）用 z3 来更新 root 权限。

```
[root@localhost zzx]# mysql -uz3
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 37
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use mysql
Reading table information for completion of table and column names
```

```
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> update user set password=password('abcd') where user='root' and host='localhost';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

（3）当数据库重启或者 root 刷新权限表后，root 登录时密码已经被更改。

```
[root@localhost zzx]# mysql -uroot  -pabc
Enter password:
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)

[root@localhost zzx]# mysql -uroot  -pabcd
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> exit
```

6. 不要把 FILE、PROCESS 或 SUPER 权限授予管理员以外的账号

FILE 权限主要以下作用：

- 将数据库的信息通过 SELECT ...INTO OUTFILE...写到服务器上有写权限的目录下，作为文本格式存放。具有权限的目录也就是启动 MySQL 时的用户权限目录。
- 可以将有读权限的文本文件通过 LOAD DATA INFILE...命令写入数据库表，如果这些表中存放了很重要的信息，将对系统造成很大的安全隐患。

在下例中详细描述了 FILE 权限可能造成的隐患。

（1）连接数据库并创建测试表 t。

```
[root@localhost zzx]# mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 43
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use mysql
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
mysql> create table t1 (name varchar(500));
Query OK, 0 rows affected (0.04 sec)
```

（2）将/etc/passwd 文件加载到表 t1 中。

```
mysql> load data infile '/etc/passwd' into table t1;
Query OK, 35 rows affected (0.00 sec)
Records: 35  Deleted: 0  Skipped: 0  Warnings: 0
```

（3）查看 t1 内容。

```
mysql> select * from t1;
+------------------------------------------------------------------------------------------------------
------+
|                                                                                                  name
|
+------------------------------------------------------------------------------------------------------
------+
|root:x:0:0:root:/root:/bin/bash
|
| bin:x:1:1:bin:/bin:/sbin/nologin
|
| daemon:x:2:2:daemon:/sbin:/sbin/nologin
|
| adm:x:3:4:adm:/var/adm:/sbin/nologin
|
| lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
|
| sync:x:5:0:sync:/sbin:/bin/sync
|
| shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
|
|……                                                  |
|                                          |
| oracle:x:501:502::/home/oracle:/bin/bash
|
| zzx:x:502:503::/home/zzx:/bin/bash
|
| mysql:x:102:102:MySQL server:/var/lib/mysql:/bin/bash
|
+------------------------------------------------------------------------------------------------------
------+
35 rows in set (0.00 sec)
```

这样，重要的用户信息/etc/passwd 内容将被写入表 t1，造成系统安全隐患。

PROCESS 权限能被用来执行“show processlist”命令，查看当前所有用户执行的查询的明文文本，包括设定或改变密码的查询。在默认情况下，每个用户都可以执行“show processlist”

命令，但是只能查询本用户的进程。因此，对 PROCESS 权限管理不当，有可能会使得普通用户能够看到管理员执行的命令。

下例中对普通用户赋予了 PROCESS 权限，来看看会造成什么安全隐患。

（1）将 PROCESS 权限授予普通用户：

```
[zzx@localhost ~]$ mysql -uroot -p
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 51
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show processlist;
mysql> grant process on *.* to  'z1'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

（2）锁定表 user，可以让进程阻塞，以方便用户看到进程内容：

```
mysql> lock table user read;
Query OK, 0 rows affected (0.00 sec)
```

（3）打开另外一个 session，用 root 执行修改密码操作，此时因为 user 表被锁定，此进程被阻塞挂起：

```
[zzx@localhost ~]$ mysql -uroot -p
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 51
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> set password=password('123');
```

（4）打开第 3 个 session，用 z1 登录，执行 show processlist 语句：

```
[zzx@localhost ~]$ mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 59
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show processlist;
+----+------+-----------+-------+---------+------+--------+---------------------------
--+
| Id | User | Host      | db    | Command | Time | State  | Info                        |
+----+------+-----------+-------+---------+------+--------+---------------------------
--+
| 51 | root | localhost | mysql | Query   |  157 | Locked | set password=password('123')
|
| 58 | root | localhost | mysql | Sleep   |  341 |        | NULL                        |
```

```
| 59 | z1   | localhost | NULL | Query   |    0 | NULL   | show processlist             |
+----+------+-----------+------+---------+------+--------+------------------------------
--+
3 rows in set (0.00 sec)
```

可以发现，z1 显示的进程中清楚地看到了 root 的修改密码操作，并看到了明文的密码，这将对系统造成严重的安全隐患。

SUPER 权限能执行 kill 命令，终止其他用户进程。下面例子中，普通用户拥有了 SUPER 权限后，便可以任意 kill 任何用户的进程。

（1）z1 登录后想 kill 掉 root 修改密码进程（进程号 51）：

```
[zzx@localhost ~]$ mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 59
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show processlist;
+----+------+-----------+------+---------+------+--------+------------------------------
--+
| Id | User | Host      | db   | Command | Time | State  | Info                         |
+----+------+-----------+------+---------+------+--------+------------------------------
--+
| 51 | root | localhost | mysql | Query   |  157 | Locked | set password=password('123')
|
| 58 | root | localhost | mysql | Sleep   |  341 |        | NULL                         |
| 59 | z1   | localhost | NULL | Query   |    0 | NULL   | show processlist             |
+----+------+-----------+------+---------+------+--------+------------------------------
--+
3 rows in set (0.00 sec)

mysql> kill 51;
ERROR 1095 (HY000): You are not owner of thread 51
mysql>
```

（2）kill 失败后，root 将 super 权限赋予 z1：

```
mysql> grant super on *.* to z1@localhost;
Query OK, 0 rows affected (0.00 sec)
mysql> show grants for z1@localhost;
+-----------------------------------------------+
| Grants for z1@localhost                       |
+-----------------------------------------------+
| GRANT PROCESS, SUPER ON *.* TO 'z1'@'localhost' |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

（3）重新 kill root 的进程成功：

```
[zzx@localhost ~]$ mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 60
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show processlist;
+----+------+-----------+-------+---------+------+--------+---------------------------
--+
| Id | User | Host      | db    | Command | Time | State  | Info                        |
+----+------+-----------+-------+---------+------+--------+---------------------------
--+
| 51 | root | localhost | mysql | Query   |   23 | Locked | set password=password('123')
|
| 58 | root | localhost | mysql | Sleep   |   26 |        | NULL                        |
| 60 | z1   | localhost | NULL  | Query   |    0 | NULL   | show processlist            |
+----+------+-----------+-------+---------+------+--------+---------------------------
--+
3 rows in set (0.00 sec)

mysql> kill 51;
Query OK, 0 rows affected (0.00 sec)
```

从上面的例子中，可以看到了 FILE、PROCESS、SUPER 三个管理权限可能带来的安全隐患，因此，除了管理员外，不要把这些权限赋予普通用户。

7. LOAD DATA LOCAL 带来的安全问题

LOAD DATA 默认读的是服务器上的文件，但是加上 LOCAL 参数后，就可以将本地具有访问权限的文件加载到数据库中。这在带来方便的同时，也带来了以下安全问题。

- 可以任意加载本地文件到数据库。
- 在 Web 环境中，客户从 Web 服务器连接，用户可以使用 LOAD DATA LOCAL 语句来读取 Web 服务器进程有读访问权限的任何文件（假定用户可以运行 SQL 服务器的任何命令）。在这种环境中，MySQL 服务器的客户实际上是 Web 服务器，而不是连接 Web 服务器的用户运行的程序。

解决方法是，可以用--local-infile=0 选项启动 mysqld 从服务器端禁用所有 LOAD DATA LOCAL 命令。

对于 mysql 命令行客户端，可以通过指定--local-infile[=1]选项启用 LOAD DATA LOCAL，或通过--local-infile=0 选项禁用。类似地，对于 mysqlimport，--local or -L 选项启用本地数据文件装载。在任何情况下，成功进行本地装载需要服务器启用相关选项。

8. 使用 MERGE 存储引擎潜藏的安全漏洞

MERGE 存储引擎的表在某些版本中可能存在以下安全漏洞：

- 用户 A 赋予表 T 的权限给用户 B；
- 用户 B 创建一个包含 T 的 MERGE 表，做各种操作；
- 用户 A 收回对 T 的权限。

存在的安全隐患是用户 B 通过 merge 表仍然可以访问表 A 中的数据。下面例子描述了这个过程。

（1）用 root 创建用户 z1，拥有数据库 test1 的所有权限：

```
mysql> grant all privileges on test1.* to z1@localhost;
Query OK, 0 rows affected (0.00 sec)
```

（2）z1 登录后创建表 t2 并插入测试数据：

```
mysql> create table t2 (id int);
Query OK, 0 rows affected (0.01 sec)
mysql> insert into t2 values(2);
Query OK, 1 row affected (0.00 sec)
```

（3）z1 在 t1 和 t2 上创建 MERGE 表：

```
mysql> create table t12 (id int) engine=merge union=(t1,t2);
Query OK, 0 rows affected (0.01 sec)
mysql> select * from t12;
+------+
| id   |
+------+
|    1 |
|    1 |
|    2 |
+------+
3 rows in set (0.00 sec)
```

（4）root 收回 z1 在表 t1 上的所有权限：

```
mysql> revoke all privileges on test1.t1  from  z1@localhost;
Query OK, 0 rows affected (0.00 sec)
mysql> show grants for z1@localhost;
+---------------------------------------+
| Grants for z1@localhost               |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'localhost' |
+---------------------------------------+
1 row in set (0.00 sec)
```

（5）z1 通过 t12 依然能够访问 t1 数据：

```
[root@localhost test1]# mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 21
Server version: 5.0.41-community-log MySQL Community Edition (GPL)
```

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

mysql> use test1
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----------------+
| Tables_in_test1 |
+-----------------+
| t12             |
| t2              |
+-----------------+
2 rows in set (0.00 sec)

mysql> select * from t2;
+------+
| id   |
+------+
|    2 |
+------+
1 row in set (0.00 sec)

mysql> select * from t12;
+------+
| id   |
+------+
|    1 |
|    1 |
|    2 |
+------+
3 rows in set (0.00 sec)
```

9. DROP TABLE 命令并不收回以前的相关访问授权

DROP 表的时候，其他用户对此表的权限并没有被收回，这样导致重新创建同名的表时，以前其他用户对此表的权限会自动赋予，进而产生权限外流。因此，在删除表时，要同时取消其他用户在此表上的相应权限。

下面的例子说明了不收回相关访问授权的隐患。

（1）用 root 创建用户 z1，授予对 test1 下所有表的 select 权限：

```
mysql> grant select on test1.* to z1@localhost;
```

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

mysql> show grants for z1@localhost;
+-----------------------------------------------+
| Grants for z1@localhost                       |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'localhost'        |
| GRANT SELECT ON `test1`.* TO 'z1'@'localhost' |
+-----------------------------------------------+
2 rows in set (0.00 sec)
```

（2）z1 登录，测试权限：

```
[root@localhost test1]# mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 43
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test1
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----------------+
| Tables_in_test1 |
+-----------------+
| t1              |
| t12             |
| t2              |
+-----------------+
3 rows in set (0.00 sec)
```

（3）root 登录，删除表 t1：

```
[root@localhost test1]# mysql -uroot
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
[root@localhost test1]# mysql -uroot -p123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test1
Reading table information for completion of table and column names
```

```
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> drop table t1;
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
```

（4）z1 登录，再次测试权限：

```
[root@localhost test1]# mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 46
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test1
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------+
| Tables_in_test1 |
+----------------+
| t12            |
| t2             |
+----------------+
2 rows in set (0.00 sec)
```

（5）此时 t1 表已经看不到了。

```
mysql> show grants for z1@localhost;
+-----------------------------------------------+
| Grants for z1@localhost                       |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'localhost'        |
| GRANT SELECT ON `test1`.* TO 'z1'@'localhost' |
+-----------------------------------------------+
2 rows in set (0.00 sec)
```

权限仍然显示对 test1 下所有表的 SELECT（安全漏洞）。

（6）root 再次登录，创建 t1 表：

```
[root@localhost test1]# mysql -uroot -p123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 48
Server version: 5.0.41-community-log MySQL Community Edition (GPL)
```

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

mysql> create table t1(id int);
Query OK, 0 rows affected (0.03 sec)

mysql> exit
```

（7）z1 登录，对 t1 权限依旧存在：

```
[root@localhost test1]# mysql -uz1 test1
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 49
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show tables;
+-----------------+
| Tables_in_test1 |
+-----------------+
| t1              |
| t12             |
| t2              |
+-----------------+
3 rows in set (0.00 sec)

mysql> select * from t1;
Empty set (0.00 sec)
```

注意：对表做删除后，其他用户对此表的权限不会自动收回，一定记住要手工收回。

10．使用 SSL

SSL（Secure Socket Layer，安全套接字层）是一种安全传输协议，最初由 Netscape 公司所开发，用以保障在 Internet 上数据传输之安全，利用数据加密（Encryption）技术，可确保数据在网络上之传输过程中不会被截取及窃听。

SSL 协议提供的服务主要有：

（1）认证用户和服务器，确保数据发送到正确的客户机和服务器；

（2）加密数据以防止数据中途被窃取；

（3）维护数据的完整性，确保数据在传输过程中不被改变。

在 MySQL 中，要想使用 SSL 进行安全传输，需要在命令行中或选项文件中设置“--ssl”选项。

对于服务器，“--ssl”选项规定该服务器允许 SSL 连接。对于客户端程序，它允许客户使用 SSL 连接服务器。单单该选项不足以使用 SSL 连接。还必须指定--ssl-ca、--ssl-cert 和--ssl-key 选项。如果不想启用 SSL，可以将选项指定为--skip-ssl 或--ssl=0。
请注意，如果编译的服务器或客户端不支持 SSL，则使用普通的未加密的连接。
确保使用 SSL 连接的安全方式是，使用含 REQUIRE SSL 子句的 GRANT 语句在服务器上创建一个账户，然后使用该账户来连接服务器，服务器和客户端均应启用 SSL 支持。下面例子创建了一个含 REQUIRE SSL 子句的账号：

```
mysql> grant select on *.* to z4 identified by '123' REQUIRE ssl;
Query OK, 0 rows affected (0.03 sec)
```

- --ssl-ca=file_name　含可信 SSL CA 的清单的文件的路径。
- --ssl-cert=file_name SSL 证书文件名，用于建立安全连接。
- --ssl-key=file_name SSL 密钥文件名，用于建立安全连接。

11．如果可能，给所有用户加上访问 IP 限制

对数据库来说，我们希望客户端过来的连接都是安全的，因此，就很有必要在创建用户的时候指定可以进行连接的服务器 IP 或者 HOSTNAME，只有符合授权的 IP 或者 HOSTNAME 才可以进行数据库访问。

12．REVOKE 命令的漏洞

当用户对多次赋予权限后，由于各种原因，需要将此用户的权限全部取消，此时，REVOKE 命令可能并不会按照我们的意愿执行，来看下面的例子。
（1）连续赋予用户两次权限，其中，第 2 次是对所有数据库的所有权限。

```
mysql> grant select,insert on test1.* to z1@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> grant all privileges  on *.* to z1@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for z1@localhost;
+-----------------------------------------------------+
| Grants for z1@localhost                             |
+-----------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'z1'@'localhost'     |
| GRANT SELECT, INSERT ON `test1`.* TO 'z1'@'localhost' |
+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

（2）此时，需要取消此用户的所有权限。

```
mysql> revoke all privileges on *.* from z1@localhost;
Query OK, 0 rows affected (0.00 sec)
```

（3）我们很可能以为，此时用户已经没有任何权限了，而不会再去查看他的权限表。而实际上，此时的用户依然拥有 test1 上的 SELECT 和 INSERT 权限。

```
mysql> show grants for z1@localhost;
+-----------------------------------------------------+
| Grants for z1@localhost                             |
+-----------------------------------------------------+
| GRANT USAGE ON *.* TO 'z1'@'localhost'              |
| GRANT SELECT, INSERT ON `test1`.* TO 'z1'@'localhost' |
+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

（4）此时，再次用 z1 登录，测试一下是否能对 test1 数据库做操作。

```
[zzx@localhost ~]$ mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 26
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> use test1
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------+
| Tables_in_test1 |
+----------------+
| t1             |
| t12            |
| t2             |
+----------------+
3 rows in set (0.00 sec)

mysql> insert into t1 values(1);
Query OK, 1 row affected (0.00 sec)
```

这个是 MySQL 权限机制造成的隐患，在一个数据库上多次赋予权限，权限会自动合并；但是在多个数据库上多次赋予权限，每个数据库上都会认为是单独的一组权限，必须在此数据库上用 REVOKE 命令来单独进行权限收回，而 REVOKE ALL PRIVILEGES ON *.* 并不会替用户自动完成这个各种。

28.3 其他安全设置选项

除了上面介绍的那些需要注意的安全隐患外，MySQL 本身还带着一些选项，适当地使用这些选项将会使数据库更加安全。

28.3.1 old-passwords

在 MySQL 4.1 版本之前，PASSWORD 函数生成的密码是 16 位。4.1 以后，MySQL 改进了密码算法，生成的函数值变成了 41 位，如下所示。

MySQL 3.23 中执行结果如下：

```
mysql> SELECT PASSWORD('mypass');
+--------------------+
| PASSWORD('mypass') |
+--------------------+
| 6f8c114b58f2ce9e   |
+--------------------+
```

MySQL 5.1 中执行结果如下：

```
mysql> SELECT PASSWORD('mypass');
+-------------------------------------------+
| PASSWORD('mypass')                        |
+-------------------------------------------+
| *6C8989366EAF75BB670AD8EA7A7FC1176A95CEF4 |
+-------------------------------------------+
```

这样就会出现一个问题，当 4.1 以后的客户端连接 4.1 以前的客户端时，没有问题，因为新客户端可以理解新旧两种加密算法。但是反过来，当 4.1 以前的客户端需要连接 4.1 以后的服务器时候，由于无法理解新的密码算法，发到服务器端的密码还是旧的算法加密后的结果，于是导致在新的服务器上出现下面无法认证的情况：

```
shell> mysql -h localhost -u root
Client does not support authentication protocol requested
by server; consider upgrading MySQL client
```

对于这个问题，可以采用以下办法解决。

（1）在服务器端用 OLD_PASSWORD 函数更改密码为旧密码格式，客户端先可以进行正常连接：

```
mysql> SET PASSWORD FOR 'some_user'@'some_host' = OLD_PASSWORD('mypass');
```

（2）在 my.cnf 的[mysqld]中增加 old-passwords 参数并重启服务器，这样新的数据库连接成功之后做的 set password、grant、password()操作后，生成的新密码全部变成旧的密码格式。

> 注意：这个参数只是为了支持 4.1 版本前的客户端才进行设置，但是这将使得新建或者修改的用户密码全部变成旧的格式，降低了系统的安全性。

28.3.2 safe-user-create

此参数如果启用，用户将不能用 GRANT 语句创建新用户，除非用户有 mysql 数据库中 user 表的 INSERT 权限。如果想让用户具有授权权限来创建新用户，应给用户授予下面的权限：

```
mysql> GRANT INSERT(user) ON mysql.user TO 'user_name'@'host_name';
```

这样确保用户不能直接更改权限列，必须使用 GRANT 语句给其他用户授予该权限。以下例子描述了这个过程。

（1）用 root 创建用户 z1，z1 可以将权限授予其他用户：

```
[zzx@localhost ~]$ mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 57
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> grant select,insert on test1.* to z1@localhost with grant option;
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
```

（2）使用 z1 创建新用户成功：

```
[zzx@localhost ~]$ mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 58
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> grant select on test1.* to z3@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> exit
```

（3）用 safe-user-create 选项重启数据库：

```
[root@localhost bin]# ./mysqld_safe --safe-user-create &
[1] 32422
[root@localhost bin]# Starting mysqld daemon with databases from /var/lib/mysql
```

（4）重新用 z1 创建用户失败：

```
[root@localhost bin]# mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> grant select on test1.* to 'z4'@'192.168';
ERROR 1410 (42000): You are not allowed to create a user with GRANT
mysql> exit
```

（5）用 root 登录给 z1 赋予 mysql 数据库中 user 表的 insert 权限：

```
[root@localhost bin]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> grant insert on mysql.user to z1@localhost;
Query OK, 0 rows affected (0.00 sec)

mysql> exit
Bye
```

（6）用 z1 重新登录，授权用户成功：

```
[root@localhost bin]# mysql -uz1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> grant select on test1.* to 'z4'@localhost;
Query OK, 0 rows affected (0.00 sec)
```

28.3.3 secure-auth

secure-auth 参数的作用是让 MySQL 4.1 以前客户端无法进行用户认证。即使使用了 old-passwords 参数也行。此参数的作用就是为了防止低版本的客户端使用旧的密码认证方式连接高版本的服务器，从而引起安全隐患。因为 4.1 版本以前客户端已经很少有人使用，因而此参数也很少被使用，读者可以下去自行测试，这里不再赘述。

28.3.4 skip-grant-tables

skip-grant-tables 这个选项导致服务器根本不使用权限系统，从而给每个人以完全访问所有数据库的权力。通过执行 mysqladmin flush-privileges 或 mysqladmin reload 命令，或执行 flush privileges 语句，都可以让一个正在运行的服务器再次开始使用授权表。
下面的例子演示了此参数的使用：

- 使用--skip-grant-tables 启动数据库。

```
[root@localhost bin]# mysqld_safe --skip-grant-tables &
[1] 15298
[root@localhost bin]# Starting mysqld daemon with databases from /var/lib/mysql
```

- 此时用 root 用户可以直接登录，而不需要密码。

```
[root@localhost bin]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.0.41-community-log MySQL Community Edition (GPL)
```

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

mysql>
```

- 此时执行 flush privileges 命令，重新使用权限系统。

```
mysql>  flush privileges;
Query OK, 0 rows affected (0.00 sec)
mysql> exit
Bye
```

- 退出后再次登录，将无法登录成功。

```
[root@localhost bin]# mysql -uroot
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
```

28.3.5 skip-network

在网络上不允许 TCP/IP 连接，所有到数据库的连接必须经由命名管道（Named Pipes）或共享内存（Shared Memory）或 UNIX 套接字（SOCKET）文件进行。这个选项适合应用和数据库共用一台服务器的情况，其他客户端将无法通过网络远程访问数据库，大大增强了数据库的安全性，但同时也带来了管理维护上的不方便，来看下面的例子。

- 服务器上打开此选项（默认关闭）并重启 MySQL 服务。

```
[mysqld]
skip-networking
port            = 3311
......
```

- 远程客户端进行连接

```
C:\mysql\bin>mysql -h192.168.7.55 -P3311 -uz2 -p
Enter password: ***
ERROR 2003 (HY000): Can't connect to MySQL server on '192.168.7.55' (10061)
```

- 关闭此选项后重启服务器。

```
[mysqld]
#skip-networking
port            = 3311
......
```

- 远程客户端进行连接

```
C:\mysql\bin>mysql -h192.168.7.55 -P3311 -uz2 -p
Enter password: ***
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 5.1.11-beta-log

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

28.3.6 skip-show-database

使用 skip-show-database 选项，只允许有 show databases 权限的用户执行 show databases 语句，该语句显示所有数据库名。不使用该选项，允许所有用户执行 show databases，但

只显示用户有 show databases 权限或部分数据库权限的数据库名。下面例子显示了启用此选项后 show databases 的执行结果：

```
[root@localhost bin]# mysqld_safe --skip-show-database &
[1] 15027
[root@localhost bin]# Starting mysqld daemon with databases from /var/lib/mysql
[root@localhost bin]# mysql -uz2
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql> show databases;
ERROR 1227 (42000): Access denied; you need the SHOW DATABASES privilege for this operation
```

28.4 小结

权限和安全问题在任何数据库中都是非常重要的。本章重点介绍了 MySQL 中的权限管理以及可能存在的一些安全问题，并给出了很多例子加以详细说明。最后还讨论了 MySQL 提供的一些安全方面的参数，用户可以根据实际情况选择使用。

第29章　MySQL 复制

MySQL 从 3.23 版本开始提供复制的功能。复制是指将主数据库的 DDL 和 DML 操作通过二进制日志传到复制服务器（也叫从服务器）上，然后在从服务器上对这些日志重新执行（也叫重做），从而使得从服务器和主服务器的数据保持同步。

MySQL 支持一台主服务器同时向多台从服务器进行复制，从服务器同时也可以作为其他服务器的主服务器，实现链状的复制。

MySQL 复制的优点主要包括以下 3 个方面：

- 如果主服务器出现问题，可以快速切换到从服务器提供服务；
- 可以在从服务器上执行查询操作，降低主服务器的访问压力；
- 可以在从服务器上执行备份，以避免备份期间影响主服务器的服务。

> 注意：由于 MySQL 实现的是异步的复制，所以主从服务器之间存在一定的差距，在从服务器上进行的查询操作需要考虑到这些数据的差异，一般只有更新不频繁的数据或者对实时性要求不高的数据可以通过从服务器查询，实时性要求高的数据仍然需要从主数据库获得。

29.1 安装配置

MySQL 的复制至少需要两个 MySQL 服务，这些 MySQL 服务可以分布在不同的服务器上，也

可以在一台服务器上启动多个服务。复制配置的步骤比较简单，下面进行详细介绍。

（1）确保主从服务器上安装了相同版本的数据库。因为复制的功能在持续的改进中，所以在可能的情况下推荐安装最新的稳定版本。

（2）在主服务器上，设置一个复制使用的账户，并授予 REPLICATION SLAVE 权限。这里创建一个复制用户 rep1，可以从 IP 为 192.168.1.101 的主机进行连接：

```
mysql> GRANT REPLICATION SLAVE ON *.* TO 'repl'@'192.168.1.101' IDENTIFIED BY '1234test';
Query OK, 0 rows affected (0.00 sec)
```

（3）修改主数据库服务器的配置文件 my.cnf，开启 BINLOG，并设置 server-id 的值。这两个参数的修改需要重新启动数据库服务才可以生效。

```
My.cnf 中修改：
[mysqld]
log-bin    = /home/mysql/log/mysql-bin.log
server-id = 1
```

（4）在主服务器上，设置读锁定有效，这个操作是为了确保没有数据库操作，以便获得一个一致性的快照：

```
mysql>  flush tables with read lock;
Query OK, 0 rows affected (0.00 sec)
```

（5）然后得到主服务器上当前的二进制日志名和偏移量值。这个操作的目的是为了在从数据库启动以后，从这个点开始进行数据的恢复。

```
mysql> show master status;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000039 | 102      |              |                  |
+------------------+----------+--------------+------------------+
1   row in set (0.00 sec)
```

（6）现在主数据库服务器已经停止了更新操作，需要生成主数据库的备份，备份的方式有很多种，可以直接在操作系统下 cp 全部的数据文件到从数据库服务器上，也可以通过 mysqldump 导出数据或者使用 ibbackup 工具进行数据库的备份，这些备份操作的步骤已经在第 27 章中有详细介绍，这里就不再一一说明。如果主数据库的服务可以停止，那么直接 cp 数据文件应该是最快的生成快照的方法：

```
[mysql@db3 db]$ tar -cvf data.tar data
data/
data/test1/
data/test1/db.opt
……
```

（7）主数据库的备份完毕后，主数据库可以恢复写操作，剩下的操作只需要在从服务器上执行：

```
mysql> unlock tables;
Query OK, 0 rows affected (0.00 sec)
```

（8）将主数据库的一致性备份恢复到从数据库上。如果是使用.tar 打包的文件包，只需要解开到相应的目录即可。

（9）修改从数据库的配置文件 my.cnf，增加 server-id 参数。注意 server-id 的值必须是唯一的，不能和主数据库的配置相同，如果有多个从数据库服务器，每个从数据库服务器必

须有自己唯一的 server-id 值。

```
my.cnf 中修改：
[mysqld]
server-id = 2
```

（10）在从服务器上，使用--skip-slave-start 选项启动从数据库，这样不会立即启动从数据库服务上的复制进程，方便我们对从数据库的服务进行进一步的配置：

```
[mysql@master1 mysql_home]$ ./bin/mysqld_safe --skip-slave-start &
[1] 8768
[mysql@master1 mysql_home]$ Starting mysqld daemon with databases from /home/mysql/sysdb/data
```

（11）对从数据库服务器做相应设置，指定复制使用的用户，主数据库服务器的 IP、端口以及开始执行复制的日志文件和位置等，具体语法如下：

```
mysql> CHANGE MASTER TO
-> MASTER_HOST='master_host_name',
-> MASTER_USER='replication_user_name',
-> MASTER_PASSWORD='replication_password',
-> MASTER_LOG_FILE='recorded_log_file_name',
-> MASTER_LOG_POS=recorded_log_position;
```

举例说明如下：

```
mysql> CHANGE MASTER TO
    -> MASTER_HOST='192.168.1.100',
    -> MASTER_PORT=3306,
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='1234test',
    -> MASTER_LOG_FILE='mysql-bin.000039',
    -> MASTER_LOG_POS=102;
Query OK, 0 rows affected (0.10 sec)
```

（12）在从服务器上，启动 slave 线程：

```
mysql> start slave;
Query OK, 0 rows affected (0.00 sec)
```

（13）这时 slave 上执行 show processlist 命令将显示类似如下进程：

```
mysql> show processlist \G
*************************** 1. row ***************************
     Id: 1
   User: root
   Host: localhost
     db: NULL
Command: Query
   Time: 0
  State: NULL
   Info: show processlist
*************************** 2. row ***************************
     Id: 2
   User: system user
```

```
    Host:
      db: NULL
Command: Connect
    Time: 68
   State: Waiting for master to send event
    Info: NULL
*************************** 3. row ***************************
      Id: 3
    User: system user
    Host:
      db: NULL
Command: Connect
    Time: 168
   State: Has read all relay log; waiting for the slave I/O thread to update it
    Info: NULL
3 rows in set (0.00 sec)
```

这表明 slave 已经连接上 master，并开始接受并执行日志。

（14）也可以测试复制服务的正确性，在主数据库上执行一个更新操作，观察是否在从数据库上同步。在主数据库上 test 数据库创建一个测试表，插入数据：

```
mysql> use test
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> create table repl_test (id int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into repl_test values(1),(2),(3),(4),(5);
Query OK, 5 rows affected (0.00 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

（15）在从数据库上检查新表是否被创建，数据是否同步：

```
mysql> use test
Database changed
mysql> show tables;
+----------------+
| Tables_in_test |
+----------------+
| repl_test      |
+----------------+
1 row in set (0.00 sec)

mysql> select * from repl_test;
+------+
| id   |
```

```
+------+
| 1    |
| 2    |
| 3    |
| 4    |
| 5    |
+------+
5 rows in set (0.00 sec)
```

可以看到数据可以正确同步到从数据库上，复制服务配置成功完成。

29.2 主要复制启动选项

上一节介绍安装配置的时候，已经介绍了几个启动时的复制参数，包括 MASTER_HOST、MASTER_PORT、MASTER_USER、MASTER_PASSWORD、MASTER_LOG_FILE、MASTER_LOG_POS。这几个参数需要在从服务器上配置，用来记录需要复制的主数据库的地址、端口、访问的用户等，这里不再赘述。下面再介绍几个常用的启动选项，如 log-slave-updates、master-connect-retry、read-only 等。

29.2.1 log-slave-updates

log-slave-updates 这个参数用来配置从服务器上的更新操作是否写二进制日志，默认是不打开的。但是，如果这个从服务器同时也要作为其他服务器的主服务器，搭建一个链式的复制，那么就需要打开这个选项，这样它的从服务器将获得它的二进制日志以进行同步操作。
这个启动参数需要和--logs-bin 参数一起使用。

29.2.2 master-connect-retry

master-connect-retry 这个参数用来设置在和主服务器的连接丢失的时候，重试的时间间隔，默认是 60 秒，即每 60 秒重试一次。

29.2.3 read-only

read-only 该参数用来设置从服务器只能接受超级用户的更新操作，从而限制应用程序错误的对从服务器的更新操作。下面创建了一个普通用户，在默认情况下，该用户是可以更新从数据库中的数据的，但是使用 read-only 选项启动从数据库以后，该用户对从数据库的更新会提示错误。

（1）主数据库中创建用户：

```
mysql> grant ALL PRIVILEGES on test.* to 'lisa'@'%' identified by '1234';
Query OK, 0 rows affected (0.00 sec)
```

（2）从服务器使用该用户登录，可以删除表中的记录：

```
[mysql@master1 log]$ mysql -ulisa -p1234 test
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12 to server version: 5.1.9-beta-log
```

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

mysql> select * from repl_test;
+------+---------------------+
| id   | createtime          |
+------+---------------------+
| 5    | 2007-07-24 15:31:37 |
| 5    | 2007-07-24 15:36:47 |
+------+---------------------+
2 rows in set (0.00 sec)

mysql> delete from repl_test;
Query OK, 2 rows affected (0.00 sec)

mysql> select * from repl_test;
Empty set (0.00 sec)
```

（3）然后关闭从服务器，使用 read-only 选项启动从数据库：

```
[mysql@master1 log]$ mysqladmin -uroot -p shutdown
Enter password:
STOPPING server from pid file /home/mysql/sysdb/mysqld.pid
070724 17:13:06  mysqld ended

[mysql@master1 mysql_home]$ ./bin/mysqld_safe --read-only&
[1] 9814
[mysql@master1    mysql_home]$    Starting    mysqld    daemon    with    databases    from
/home/mysql/sysdb/data
```

（4）然后再使用 Lisa 用户登录数据库，进行删除操作：

```
[mysql@master1 data]$ mysql -ulisa -p1234 test
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 5.1.9-beta-log

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

mysql> select * from repl_test;
Empty set (0.00 sec)

mysql> select * from repl_test;
+------+---------------------+
| id   | createtime          |
+------+---------------------+
| 5    | 2007-07-24 16:29:49 |
+------+---------------------+
1 row in set (0.00 sec)
```

```
mysql> delete from repl_test;
ERROR 1290 (HY000): The MySQL server is running with the --read-only option so it cannot execute this statement
```

可以看到，使用 read-only 启动的从数据库会拒绝普通用户的更新操作，以确保从数据库的安全性。

29.2.4 指定复制的数据库或者表

可以使用 replicate-do-db、replicate-do-table、replicate-ignore-db、replicate-ignore-table 或 replicate-wild-do-table 来指定从主数据库复制到从数据库的数据库或者表。有些时候用户只需要将关键表备份到从服务器上，或者只需要将提供查询操作的表复制到从服务器上，这样就可以通过配置这几个参数来筛选进行同步的数据库和表。

下面演示的例子是设置 replicate-do-table 的情况，首先在主数据库的 test 数据库中创建两个表 repl_test 和 repl_test_1，然后从数据库启动的时候指定 replicate-do-table=test.repl_test，即只复制 repl_test 表。最后在主数据库中更新两个表，检查从数据库中数据复制的情况。

（1）首先检查主从数据库上两个表的记录，主从数据库是相同的：

```
mysql> select * from repl_test;
+------+---------------------+
| id   | createtime          |
+------+---------------------+
| 5    | 2007-07-24 16:29:49 |
+------+---------------------+
1 row in set (0.00 sec)

mysql> select * from repl_test_1;
Empty set (0.01 sec)
```

（2）然后关闭从数据库，然后重新启动时指定复制其中一个表：

```
[mysql@master1 data]$ mysqladmin -uroot -p shutdown
Enter password:
STOPPING server from pid file /home/mysql/sysdb/mysqld.pid
070724 17:31:11  mysqld ended

[mysql@master1 mysql_home]$ ./bin/mysqld_safe --replicate-do-table=test.repl_test &
[1] 9884
[mysql@master1 mysql_home]$ Starting mysqld daemon with databases from /home/mysql/sysdb/data
```

（3）下面更新主数据库上的两个表 repl_test 和 repl_test_1：

```
mysql> insert into repl_test values(1,now());
Query OK, 1 row affected (0.00 sec)

mysql> insert into repl_test_1 values(1,now());
Query OK, 1 row affected (0.00 sec)

mysql> select * from repl_test;
+------+---------------------+
```

```
| id   | createtime          |
+------+---------------------+
| 5    | 2007-07-24 16:29:49 |
| 1    | 2007-07-24 16:47:43 |
+------+---------------------+
2 rows in set (0.00 sec)

mysql> select * from repl_test_1;
+------+---------------------+
| id   | create_time         |
+------+---------------------+
| 1    | 2007-07-24 16:47:47 |
+------+---------------------+
1 row in set (0.00 sec)
```

（4）再检查从数据库的复制情况：

```
mysql> select * from repl_test;
+------+---------------------+
| id   | createtime          |
+------+---------------------+
| 5    | 2007-07-24 16:29:49 |
| 1    | 2007-07-24 16:47:43 |
+------+---------------------+
2 rows in set (0.00 sec)

mysql> select * from repl_test_1;
Empty set (0.00 sec)
```

从测试的结果可以看到，只有 repl_test 表的记录被复制到从服务器上，而 repl_test_1 表的记录没有复制。

29.2.5 slave-skip-errors

在复制过程中，由于各种原因，从服务器可能会遇到执行 BINLOG 中的 SQL 出错的情况（比如主键冲突），默认情况下，从服务器将会停止复制进程，不再进行同步，等待用户介入处理。这种问题如果不能及时发现，将会对应用或者备份产生影响。此参数的作用就是用来定义复制过程中从服务器可以自动跳过的错误号，这样当复制过程中遇到定义中的错误号时，便可以自动跳过，直接执行后面的 SQL 语句，以此来最大限度地减少人工干预。此参数可以定义多个错误号，或者通过定义成 all 跳过全部的错误。具体语法如下：

```
--slave-skip-errors=[err_code1,err_code2,... | all]
```

如果从数据库主要是作为主数据库的备份，那么就不应该使用这个启动参数，设置不当，很可能造成主从数据库的数据不同步。但是，如果从数据库仅仅是为了分担主数据库的查询压力，且对数据的完整性要求不是很严格，那么这个选项的确可以减轻数据库管理员维护从数据库的工作量。

29.3 日常管理维护

复制环境配置完成后，数据库管理员需要经常进行一些日常监控和管理维护工作，以便能及时发现一些复制中的问题，并尽快解决，以此来保持复制能够正常的工作。本节将向读者介绍一些常用的监控和管理维护方法。

29.3.1 查看从服务器状态

为了防止复制过程中出现故障从而导致复制进程停止，我们需要经常检查从服务器的复制状态。一般使用 show slave status 命令来检查，如下例所示：

```
mysql> show slave status \G
*************************** 1. row ***************************
             Slave_IO_State: Waiting for master to send event
                Master_Host: 192.168.1.100
                Master_User: repl
                Master_Port: 3306
              Connect_Retry: 60
            Master_Log_File: mysql-bin.000039
        Read_Master_Log_Pos: 382
             Relay_Log_File: master1-relay-bin.000002
              Relay_Log_Pos: 523
      Relay_Master_Log_File: mysql-bin.000039
           Slave_IO_Running: Yes
          Slave_SQL_Running: Yes
            Replicate_Do_DB:
        Replicate_Ignore_DB:
         Replicate_Do_Table:
     Replicate_Ignore_Table:
    Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
                 Last_Errno: 0
                 Last_Error:
               Skip_Counter: 0
        Exec_Master_Log_Pos: 382
            Relay_Log_Space: 523
            Until_Condition: None
             Until_Log_File:
              Until_Log_Pos: 0
         Master_SSL_Allowed: No
         Master_SSL_CA_File:
         Master_SSL_CA_Path:
            Master_SSL_Cert:
          Master_SSL_Cipher:
             Master_SSL_Key:
```

```
        Seconds_Behind_Master: 0
1 row in set (0.00 sec)
```

在显示的这些信息中，我们主要关心“Slave_IO_Running”和“Slave_SQL_Running”这两个进程状态是否是“yes”，这两个进程的含义分别如下。

- Slave_IO_Running：此进程负责从服务器（Slave）从主服务器（Master）上读取 BINLOG 日志，并写入从服务器上的中继日志中。
- Slave_SQL_Running：此进程负责读取并且执行中继日志中的 BINLOG 日志。

只要其中有一个进程的状态是 no，则表示复制进程停止，错误原因可以从“Last_Errno”字段的值中看到。

除了查看上面的信息，用户还可以通过这个命令了解从服务器的配置情况以及当前和主服务器的同步情况，包括指向那个主服务器，主服务器的端口，复制使用的用户，当前日志恢复到的位置等，这些信息都是记录在从服务器这一端的，主服务器上并没有相应的信息。

29.3.2 主从服务器同步维护

在某些繁忙的 OLTP(在线事务处理)系统上，由于主服务器更新频繁，而从服务器由于各种原因（比如硬件性能较差）导致更新速度较慢，从而使得主从服务器之间的数据差距越来越大，最终对某些应用产生影响。在这种情况下，我们就需要定期地进行主从服务器的数据同步，使得主从数据差距能够减到最小。常用的方法是：在负载较低的时候暂时阻塞主数据库的更新，强制主从数据库更新同步。具体操作步骤如下。(1) 在主服务器上，执行以下语句（注意，会阻塞主数据库的所有更新操作）：

```
mysql> FLUSH TABLES WITH READ LOCK;
Query OK, 0 rows affected (0.01 sec)

mysql> SHOW MASTER STATUS;
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000039 | 974      |              |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

记录 SHOW 语句的输出的日志名和偏移量，这些是从服务器复制的目的坐标。

(2) 在从服务器上，执行下面语句，其中 MASTER_POS_WAIT()函数的参数是前面步骤中得到的复制坐标值：

```
mysql> select MASTER_POS_WAIT('mysql-bin.000039','974');
+-----------------------------------------+
| MASTER_POS_WAIT('mysql-bin.000039','974') |
+-----------------------------------------+
| 0                                       |
+-----------------------------------------+
1 row in set (0.00 sec)
```

这个 SELECT 语句会阻塞直到从服务器达到指定的日志文件和偏移量后，返回 0，如果返回-1，则表示超时退出。查询返回 0 时，则从服务器与主服务器同步。

（3）在主服务器上，执行下面的语句允许主服务器重新开始处理更新：

```
mysql> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

29.3.3 从服务器复制出错的处理

上文提到过，在某些情况下，会出现从服务器更新失败，这时，首先需要确定是否是从服务器的表与主服务器的不同造成的。如果是表结构不同导致的，则修改从服务器的表与主服务器的相同，然后重新运行 START SLAVE 语句。

如果不是表结构不同导致的更新失败，则需要确认手动更新是否安全，然后忽视来自主服务器的更新失败的语句。跳过来自主服务器的语句的命令为 SET GLOBAL SQL_SLAVE_SKIP_COUNTER = n，其中 n 的取值为 1 或者 2。如果来自主服务器的更新语句不使用 AUTO_INCREMENT 或 LAST_INSERT_ID()，n 值应为 1，否则，值应为 2。原因是使用 AUTO_INCREMENT 或 LAST_INSERT_ID()的语句需要从二进制日志中取两个事件。以下例子就是在从服务器端模拟跳过主服务器的两个更新语句的效果。

（1）首先，在从服务器端先停止复制进程，并设置跳过两个语句：

```
mysql> select * from repl_test;
+------+
| id   |
+------+
| 1    |
+------+
1 row in set (0.00 sec)

mysql> stop slave;
Query OK, 0 rows affected (0.00 sec)

mysql> SET GLOBAL SQL_slave_SKIP_COUNTER = 2;
Query OK, 0 rows affected (0.01 sec)
```

（2）然后在主服务器端插入 3 条记录：

```
mysql> select * from repl_test;
+------+
| id   |
+------+
| 1    |
+------+
1 row in set (0.00 sec)

mysql> insert into repl_test values(2);
Query OK, 1 row affected (0.00 sec)

mysql> insert into repl_test values(3);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert into repl_test values(4);
Query OK, 1 row affected (0.00 sec)

mysql> select * from repl_test;
+------+
| id   |
+------+
| 1    |
| 2    |
| 3    |
| 4    |
+------+
4 rows in set (0.00 sec)
```

（3）从服务器端启动复制进程，检查测试的表，发现首先插入的两条记录被跳过了，只执行了第 3 条插入语句：

```
mysql> start slave;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from repl_test;
+------+
| id   |
+------+
| 1    |
| 4    |
+------+
2 rows in set (0.00 sec)
```

29.3.4 log event entry exceeded max_allowed_packet 的处理

如果应用中使用大的 BLOG 列或者长字符串，那么在从服务器上恢复的时候，可能会出现“log event entry exceeded max_allowed_packet”错误，这是因为含有大文本的记录无法通过网络进行传输导致。解决的办法就是在主从服务器上增加 max_allowed_packet 参数的大小，这个参数的默认值为 1MB，可以按照实际需要进行修改，比如下例中将其增大为 16MB：

```
mysql> show variables like 'max_allowed_packet';
+--------------------+---------+
| Variable_name      | Value   |
+--------------------+---------+
| max_allowed_packet | 1047552 |
+--------------------+---------+
1 row in set (0.00 sec)

mysql> SET @@global.max_allowed_packet=16777216;
Query OK, 0 rows affected (0.01 sec)
```

同时在 my.cnf 里，设置 max_allowed_packet = 16M，保证下次数据库重新启动后参数

继续有效。

29.3.5 多主复制时的自增长变量冲突问题

在大多数情况下，一般只使用单主复制（一台主服务器对一台或者多台从服务器）。但是在某些情况下，可能会需要使用多主复制（多台主服务器对一台从服务器）。这个时候，如果主服务器的表采用自动增长变量，那么复制到从服务器的同一张表后很可能会引起主键冲突，因为系统参数 auto_increment_increment 和 auto_increment_offset 默认值为 1，这样多台主服务器的自增变量列迟早会发生冲突。在单主复制时，可以采用默认设置，不会有主键冲突发生。但是使用多主复制时，就需要定制 auto_increment_increment 和 auto_increment_offset 的设置，保证多主之间复制到从数据库不会有重复冲突。比如，两个 master 的情况可以按照以下设置。

- Master1 上：auto_increment_increment = 2，auto_increment_offset = 1；（1,3,5,7…序列）。
- Master2 上：auto_increment_increment = 2，auto_increment_offset = 0；（0,2,4,6…序列）。

下面的例子中创建了测试表 ai，只有一个自增字段 i，我们开始演示修改这两个参数的效果。

首先在参数是默认值的时候，往表 ai 中插入记录，可以看到自动增长列的值是连续的。

```
mysql> CREATE TABLE ai (
    ->   i bigint(20) NOT NULL AUTO_INCREMENT,
    ->   PRIMARY KEY (i)
    -> ) ENGINE=MyISAM DEFAULT CHARSET=gbk;
Query OK, 0 rows affected (0.03 sec)

mysql> SHOW VARIABLES LIKE 'auto_inc%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| auto_increment_increment | 1     |
| auto_increment_offset    | 1     |
+--------------------------+-------+
2 rows in set (0.00 sec)

mysql> insert into ai values(null),(null),(null);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from ai;
+---+
| i |
+---+
| 1 |
| 2 |
| 3 |
```

```
+---+
3 rows in set (0.00 sec)
```

然后把参数 auto_increment_increment 的值修改成 10，再插入记录：

```
mysql> SET @@auto_increment_increment=10;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'auto_inc%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| auto_increment_increment | 10    |
| auto_increment_offset    | 1     |
+--------------------------+-------+
2 rows in set (0.00 sec)

mysql> insert into ai values(null),(null),(null);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from ai;
+----+
| i  |
+----+
| 1  |
| 2  |
| 3  |
| 11 |
| 21 |
| 31 |
+----+
6 rows in set (0.00 sec)
```

从测试的结果上看，新插入的记录不再连续了，每次增加 10。接着再修改 auto_increment_offset 参数，了解插入记录的效果：

```
mysql> SET @@auto_increment_offset=5;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'auto_inc%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| auto_increment_increment | 10    |
| auto_increment_offset    | 5     |
+--------------------------+-------+
```

```
2 rows in set (0.00 sec)

mysql> insert into ai values(null),(null),(null);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> select * from ai;
+----+
| i  |
+----+
| 1  |
| 2  |
| 3  |
| 11 |
| 21 |
| 31 |
| 35 |
| 45 |
| 55 |
+----+
9 rows in set (0.00 sec)
```

从插入记录的结果上可以了解，auto_increment_offset 参数设置的是每次增加后的偏移量，也就是每次按照 10 累加之后，还需要增加 5 个偏移量。

通过这两个参数可以方便地设置不同的主服务器上的自动增长列的值的范围，这样在这些数据复制到从服务器上时可以有效地避免主键的重复。

29.3.6 查看从服务器的复制进度

很多情况下，我们都想知道从服务器复制的进度如何。知道了这个差距，可以帮助我们判断是否需要手工来做主从的同步工作，也可以帮助我们判断从服务器上做统计的数据精度如何。这个值可以通过 SHOW PROCESSLIST 列表中的 Slave_SQL_Running 线程的 Time 值得到，它记录了从服务器当前执行的 SQL 时间戳与系统时间之间的差距，单位是秒。下面通过例子测试一下这个时间的准确性。

（1）首先在主服务器上插入一个包含当前时间戳的记录：

```
mysql> alter table repl_test add column createtime datetime;
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> insert into repl_test values(5,now());
Query OK, 1 row affected (0.00 sec)

mysql> select * from repl_test;
+------+---------------------+
| id   | createtime          |
+------+---------------------+
```

```
| 5    | 2007-07-24 15:36:47 |
+------+---------------------+
1 rows in set (0.00 sec)
```

（2）为了方便模拟时间，这里把从服务器上复制的 IO 进程（Slave_IO_Running）停下来，使得从服务器暂时不写中继日志，也就是最后执行的 SQL 就是当前中继日志中最后一个 SQL。

```
mysql> stop slave  IO_THREAD ;
Query OK, 0 rows affected (0.00 sec)
```

（3）一段时间之后，再从服务器端查询复制的情况：

```
mysql> select * from repl_test;
+------+---------------------+
| id   | createtime          |
+------+---------------------+
| 5    | 2007-07-24 15:36:47 |
+------+---------------------+
1 rows in set (0.00 sec)

mysql> select now();
+---------------------+
| now()               |
+---------------------+
| 2007-07-24 16:21:58 |
+---------------------+
1 row in set (0.00 sec)
```

（4）这时再查询 SQL 线程的时间，显示的 Time 值是 2705，也就是最后执行的复制操作大概是主服务器上 45 分钟前的更新。

```
mysql> show processlist \G
*************************** 1. row ***************************
......
*************************** 2. row ***************************
     Id: 5
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 2705
  State: Has read all relay log; waiting for the slave I/O thread to update it
   Info: NULL
*************************** 3. row ***************************
......
3 rows in set (0.00 sec)
```

由于 MySQL 复制的机制是执行主服务器传输过来的二进制日志，二进制日志中的每个语句通过设置时间戳来保证执行时间和顺序的正确性，所以每个语句执行之前都会首先设置时间

戳，而通过查询这个进程的 Time 就可以知道最后设置的时间戳和当前时间的差距。

29.4 切换主从服务器

假设有一个复制的环境，一个主数据库服务器 M，两个从数据库服务器 S1、S2 同时指向主数据库服务器 M。当主数据库 M 因为某种原因出现故障的时候，需要将其中的一个从数据库服务器（假设选中 S1）切换成主数据库服务器，同时修改另一个从数据库（S2）的配置，使其指向新的主数据库（S1）。此外还需要通知应用修改主数据库的 IP 地址，如果可能，将出现故障的主数据库（M）修复或者重置成新的从数据库。

下面详细介绍一下切换主从服务器的操作步骤。

（1）首先要确保所有的从数据库都已经执行了 relay log 中的全部更新，在每个从服务器上，执行 STOP SLAVE IO_THREAD，然后检查 SHOW PROCESSLIST 的输出，直到看到状态是 Has read all relay log，表示更新都执行完毕。

```
mysql> STOP SLAVE IO_THREAD;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW PROCESSLIST \G
*************************** 1. row ***************************
     Id: 2
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 4137
  State: Has read all relay log; waiting for the slave I/O thread to update it
   Info: NULL
*************************** 2. row ***************************
……
2 rows in set (0.00 sec)
```

（2）在从数据库 S1 上，执行 STOP SLAVE 停止从服务，然后 RESET MASTER 重置成主数据库。

```
mysql> STOP SLAVE;
Query OK, 0 rows affected (0.00 sec)

mysql> reset master;
Query OK, 0 rows affected (0.06 sec)
```

（3）在 S2 上，执行 STOP SLAVE 停止从服务，然后执行 CHANGE MASTER TO MASTER_HOST = 'S1'重新设置主数据库，然后再执行 START SLAVE 启动复制。

```
mysql> STOP SLAVE;
Query OK, 0 rows affected (0.00 sec)

mysql> CHANGE MASTER TO MASTER_HOST = '192.168.1.101';
Query OK, 0 rows affected (0.05 sec)
```

```
mysql> start slave;
Query OK, 0 rows affected (0.00 sec)
```

（4）通知所有的客户端将应用指向 S1，这样客户端发送的所有的更新语法写入到 S1 的二进制日志。

（5）删除新的主数据库服务器上的 master.info 和 relay-log.info 文件，否则下次重启的时候还会按照从服务器启动。

（6）最后，如果 M 服务器可以修复，则可以按照 S2 的方法配置成 S1 的从服务器。

> 注意：上面测试的步骤是默认 S1 是打开 log-bin 选项的，这样重置成主数据库后可以将二进制日志传输到其他从服务器。其次，S1 上没有打开 log-slave-updates 参数，否则重置成主数据库后，可能会将已经执行过的二进制日志重复传输给 S2，导致 S2 的同步错误。

29.5 小结

复制是 MySQL 数据库中经常使用的一个功能，它可以有效地保证主数据库的数据安全，并减轻主数据库的备份压力，以及分担主数据库的一部分查询压力。

MySQL 复制环境的搭建非常简单，在实际使用中，建议给重要的数据库配置复制，如果没有足够的服务器可以使用，那么也可以在一个主数据库上启动另外一个 MySQL 服务，以作为另外一个 MySQL 服务的从数据库。

本章重点介绍了复制环境的搭建、日常管理、主要启动选项和主从切换的操作步骤等，希望对读者在搭建和日常使用复制功能时有所帮助。由于 MySQL 的复制功能还在不断地完善中，更多的维护操作方法和启动选项的设置读者可以参考 MySQL 最新的官方文档。

第30章　MySQL Cluster

MySQL 自 4.1.x 版本开始推出 MySQL Cluster 功能。Cluster 简单地来说，就是一组“节点”的组合。这里的“节点”是一个逻辑概念，一台计算机上可以存放一个节点，也可以存放多个节点。这些节点的功能各不相同，有的用来存储数据（数据节点），有的用来存放表结构（SQL 节点），有的用来对其他节点进行管理（管理节点）。这些节点组合在一起，可以为应用提供具有高可用性、高性能和可缩放性的 Cluster 数据管理。

MySQL 使用 NDB 存储引擎来对数据节点的数据进行存储，以前版本的 NDB 存储引擎只支持基于内存的数据表，从 5.1 版本开始支持基于磁盘的数据表。

理论上，MySQL Cluster 通过数据的分布式存储和可扩展的系统架构，可以满足更大规模的应用；而且通过冗余策略，可以大大地提高系统的可靠性和数据的有效性。

虽然在 MySQL 5.0 版本时就有公司将 MySQL Cluster 用于正式生产环境，但是更多的测试表明，MySQL Cluster 在性能和可靠性上还有待于完善，我们期待 MySQL 5.1 正式版发布时，MySQL Cluster 在性能和可靠性上能够有更显著的改进。

30.1 MySQL Cluster 架构

MySQL Cluster 的系统架构如图 30-1 所示。

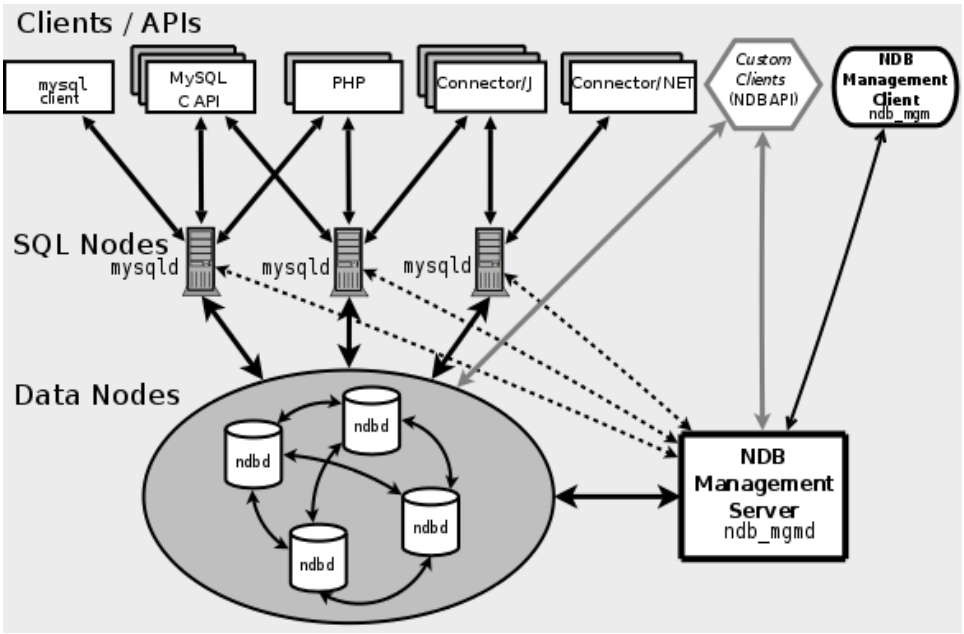


图 30-1　MySQL Cluster 架构

从图 30-1 中可以看出，MySQL Cluster 按照节点类型可以分为 3 部分。

■　管理节点

顾名思义，管理节点用来对其他节点进行管理。实际操作中，是通过对一个叫作config.ini的配置文件进行维护而起到管理的作用。该文件可以用来配置有多少需要维护的副本、需要在每个数据节点上为数据和索引分配多少内存、数据节点的位置、在每个数据节点上保存数据的磁盘位置、SQL节点的位置等信息。管理节点只能有一个，配置要求不高。

■　SQL 节点

SQL节点可以理解为应用和数据节点之间的一个桥梁。应用不能直接访问数据节点，只能先访问SQL节点，然后SQL节点再去访问数据节点来返回数据。Cluster中可以有多个SQL节点，通过每个SQL节点查询到的数据都是一致的，通常来说，SQL节点越多，分配到每个SQL节点的负载就越小，系统的整体性能就越好。

■　数据节点

用来存放Cluster里面的数据，可以有多个数据节点。每个数据节点可以有多个镜像节点。任何一个数据节点发生故障，只要它的镜像节点正常，Cluster就可以正常运行。

这 3 种逻辑上不同的节点物理上可以存放在不同的服务器上，也可以在同一台服务器上。通过架构中各个节点的介绍，可以总结一下 MySQL Cluster 的访问过程：前台应用利用一定的负载均衡算法将对数据库的访问分散到不同的 SQL 节点上，然后 SQL 节点对数据节点进行数据访问并从数据节点返回结果，最后 SQL 节点将收到的结果返给前台应用。而管理节点并不参与访问过程，它只用来对 SQL 节点和数据节点进行配置管理。

30.2 MySQL Cluster 的配置

MySQL 5.0 目前还不支持 Windows 平台上的 Cluster，本节将以 Linux AS4 上的 MySQL 5.1.11 Beta 版为测试环境，来介绍一下 MySQL Cluster 的配置方法，本章后面的其他部分也使用此测试环境为例。具体的节点配置如表 30-1 所示。

表 30-1　　节点配置说明

节点	对应的 IP 地址和端口
管理节点（1 个）	192.168.7.187
SQL 节点（2 个）	192.168.7.187:3331
	192.168.7.55:3331
数据节点（2 个）	192.168.7.187　datadir =/home/zzx2/mysql/data
	192.168.7.55　datadir=/home/zzx2/mysql/data

由于硬件环境限制，我们将管理节点、一个 SQL 节点和一个数据节点放在一台服务器上（192.168.7.187）；将另外一个 SQL 节点和一个数据节点放在另外一台服务器上（192.168.7.55）。

30.2.1 MySQL Cluster 的版本支持

对于 RPM 安装包，通常只下载 Server 和 Client 包就可满足大多数应用。但是，Server 包默认是不包括 Cluster 组件的，为了支持 Cluster 功能，还需要单独下载 Cluster 相关包，对于 5.0，可以到 http://dev.mysql.com/downloads/mysql/5.0.html 去下载最新版本的包，如图 30-2 所示。

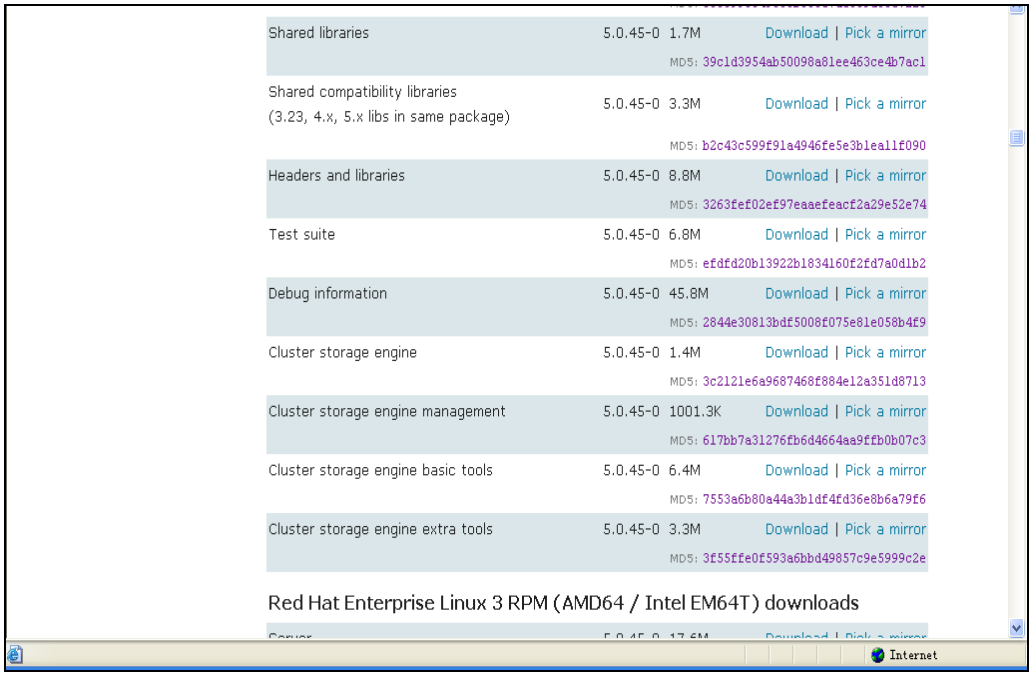


图 30-2　下载 RPM 包的 Cluster 组件界面

通常，对于 SQL 节点和数据节点，除了必须下载的 Server 包外，还需要下载 Cluster storage engine 包。如果是管理节点，则不用下载 Server 包，但是需要下载 Client 包，此外还需要下

载 Cluster storage engine management、Cluster storage engine basic tools 和 Cluster storage engine extra tools 3 个管理工具包。这 3 个工具包分别提供了 MySQL Cluster 管理服务器（ndb_mgmd）、客户端管理工具（最重要的是 ndb_mgm）、额外的集群测试和监控工具。其中，前两个包是必须的，最后一个包没有也可以，不会影响 Cluster 的正常运行和管理。下载后的包安装方法和其他 RPM 包完全一样，可以参考本书的第 1 章，这里就不再赘述。对于二进制包，如果要下载 5.0 版本，则需要下载 MAX 版才支持 Cluster。如果只是测试用，也可以只下载 5.1 的测试版（写作本书时 5.1 版本尚未正式发布）。
对于源码包，在编译时，务必使用“--with-ndbcluster”选项，使得编译后的二进制包支持 Cluster。

30.2.2 管理节点配置步骤

MySQL Cluster 安装完毕后，首先来配置管理节点。管理节点的配置很简单，但它是 Cluster 配置过程中最关键的一步。以上述测试节点为例，具体操作如下。

（1）在服务器 192.168.7.187 的/home/zzx2/下创建目录 mysql-cluster，并在目录中创建配置文件 config.ini。

```
mkdir /home/zzx2/mysql-cluster
cd /home/zzx2/mysql-cluster
touch config.ini
```

（2）根据这里的测试集群环境，config.ini 文件配置如下：

```
[NDBD DEFAULT]
NoOfReplicas=1          #每个数据节点的镜像数量
DataMemory=500M   # 每个数据节点中给数据分配的内存
IndexMemory=300M  #每个数据节点中给索引分配的内存
[TCP DEFAULT]
portnumber=2202     #数据节点的默认连接端口
[NDB_MGMD]   #配置管理节点
id=1
hostname=192.168.7.187   #管理节点 IP
datadir=/home/zzx2/mysql-cluster     #管理节点数据目录
[NDBD]
id=2
hostname =192.168.7.187
datadir =/home/zzx2/mysql/data
[NDBD]
id=3
hostname=192.168.7.55
datadir=/home/zzx2/mysql/data

[MYSQLD]
hostname=192.168.7.187
[MYSQLD]
hostname=192.168.7.55
[MYSQLD]# Options for mysqld process:
```

上面的配置文件中，包括很多的组，组名用“[]”括起来，这里我们最关心的是 3 类节点组的配置，分别定义如下。

- [NDB_MGMD]：表示管理节点的配置，只能有一个。
- [NDBD DEFAULT]：表示每个数据节点的默认配置，在每个节点的[NDBD]中不用再写这些选项。只能有一个。
- [NDBD]：表示每个数据节点的配置，可以有多个。
- [MYSQLD]：表示 SQL 节点的配置，可以有多个，分别写上不同 SQL 节点的 IP 地址；也可以不用写 IP 地址，只保留一个空节点，表示任意一个 IP 地址都可以进行访问。此节点的个数表明了可以用来连接数据节点的 SQL 节点总数。

每个节点都要有一个独立的 id 号，可以手工填写，比如“id=2”，也可以不写，系统会按照配置文件的填写顺序自动分配。

30.2.3 SQL 节点和数据节点的配置

SQL 节点和数据节点的配置非常简单，只需要在对 MySQL 的配置文件（my.cnf）中增加如下内容即可（参数含义见后面注释）：

```
# Options for mysqld process:
[MYSQLD]
ndbcluster                # 运行 NDB 存储引擎
ndb-connectstring=192.168.7.187  # 定位管理节点
# Options for ndbd process:
[MYSQL_CLUSTER]
ndb-connectstring=192.168.7.187  #定位管理节点
```

SQL 节点和数据节点的不同之处在于数据节点只需要配置上述选项即可，SQL 节点还需要配置 MySQL 服务器的其他选项。

30.3 开始使用 Cluster

MySQL Cluster 配置完毕后，本节将详细介绍 Cluster 的启动、关闭和使用方法。

30.3.1 Cluster 的启动

Cluster 需要各个节点都进行启动后才可以运行，节点的启动顺序为管理节点➔数据节点➔SQL 节点。以本章的测试环境为例，启动步骤如下。

（1）在管理节点上，从系统 shell 发出下述命令以启动管理节点进程：

```
[zzx2@zzx mysql-cluster]$ ndb_mgmd -f  ./config.ini
Cluster configuration warning:
  arbitrator with id 1 and db node with id 2 on same host 192.168.7.187
  Running arbitrator on the same host as a database node may
  cause complete cluster shutdown in case of host failure.
```

命令行中的 ndb_mgmd 是 MySQL Cluster 的管理服务器，后面的-f 表示后面的参数是启动的参数配置文件，也可以用--config-file=name 来表示，其他选项可以用 ndb_mgmd --help 命令来查看。如果没有任何提示，则集群管理进程成功启动，用 ps（进程查看）命令查看，可

以看到类似以下进程：

```
zzx2      7248     1  0 15:41 ?        00:00:00 ndb_mgmd -f ./config.ini
```

本例中出现了一个 warning，是因为管理节点和数据节点在同一台服务器上，如果主机出现故障，则整个 Cluster 会 shutdown。在实际生产环境下，最好将管理节点放到单独的服务器上。这里我们忽略这个 warning。

（2）在每台数据节点服务器上（本例为 192.168.7.55 和 192.168.7.187），运行下述命令启动 ndbd 进程：

```
shell> ndbd --initial --ndb-connectstring=192.168.7.187:1186
```

执行完毕后，查看系统进程，如果可以看到如下进程，则表示数据节点的 ndbd 进程启动成功：

```
[zzx2@zzx mysql-cluster]$ ps -ef
.....省略其他进程
zzx2            7862          1    0  18:19  ?                00:00:00  ndbd  --initial
--ndb-connectstring=192.168.7.187:1186
zzx2            7863     7862    2  18:19  ?                00:00:00  ndbd  --initial
--ndb-connectstring=192.168.7.187:1186
.....省略其他进程
```

ndbd 进程是使用 NDB 存储引擎处理表中数据的进程。通过该进程，存储节点能够实现分布式事务管理、节点恢复、在线备份等相关的任务。

> 注意：仅应在首次启动 ndbd 时，或在备份 / 恢复或配置变化后重启 ndbd 时使用“--initial”参数，这很重要。原因在于，该参数会使节点删除由早期 ndbd 实例创建的、用于恢复的任何文件，包括恢复用日志文件。

（3）依次启动 SQL 节点上的 MySQL 服务。

对于 node 1（192.168.7.187），启动其上的 MySQL 服务：

```
[zzx2@zzx mysql]$ ./bin/mysqld_safe &
[1] 29817
[zzx2@zzx mysql]$ Starting mysqld daemon with databases from /home/zzx2/mysql/data
```

对于 node 2（192.168.7.55），启动其上的 MySQL 服务：

```
[zzx2@test55 mysql]$ cd /home1/zzx2/mysql
[zzx2@test55 mysql]$ ./bin/mysqld_safe &
[1] 9762
[zzx2@test55 mysql]$ Starting mysqld daemon with databases from /home1/zzx2/mysql/data
```

（4）节点全部成功启动后，用 ndb_mgm 工具的 show 命令查看集群状态：

```
[zzx2@zzx data]$ ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm> show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=2    @192.168.7.187  (Version: 5.1.11, Nodegroup: 0, Master)
id=3    @192.168.7.55  (Version: 5.1.11, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
```

```
id=1    @192.168.7.187  (Version: 5.1.11)

[mysqld(API)]   3 node(s)
id=4    @192.168.7.187  (Version: 5.1.11)
id=5    @192.168.7.55  (Version: 5.1.11)
id=6 (not connected, accepting connect from any host)

ndb_mgm>
```

ndb_mgm 工具是 ndb_mgmd（MySQL Cluster Server）的客户端管理工具，通过它可以方便地检查 Cluster 的状态、启动备份、关闭 Cluster 等功能。更详细的使用方法，可以通过 ndb_mgm --help 命令来进行查看。

从上面显示的状态可以看出以下信息。

（1）集群目前的管理服务器端口是 1186。

```
Connected to Management Server at: localhost:1186
```

（2）集群的数据节点有 3 个，详细信息为：

```
[ndbd(NDB)]     2 node(s)
id=2    @192.168.7.187  (Version: 5.1.11, Nodegroup: 0, Master)
id=3    @192.168.7.55  (Version: 5.1.11, Nodegroup: 0)
```

（3）管理节点有一个，详细信息为：

```
[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.7.187  (Version: 5.1.11)
```

（4）SQL 节点有 3 个，目前处于连接状态的有 2 个，详细信息为：

```
[mysqld(API)]   3 node(s)
id=4    @192.168.7.187  (Version: 5.1.11)
id=5    @192.168.7.55  (Version: 5.1.11)
id=6 (not connected, accepting connect from any host)
```

30.3.2 Cluster 的测试

成功启动后，下面来测试一下 Cluster 的功能。上文提到过，如果要使用 Cluster，则表的存储引擎必须为 NDB，其他类型存储引擎的数据将不会保存到数据节点中。对于 Cluster 的一个重要功能就是防止单点故障，本节将对这些问题分别进行测试。

1. NDB 存储引擎测试

（1）在任意一个 SQL 节点（这里用 192.168.7.187）的 test 库中创建测试表 t1，设置存储引擎为 NDB，并插入两条测试数据：

```
mysql> create table t1(id int) engine=ndb;
Query OK, 0 rows affected (0.92 sec)
mysql> insert into t1 values(1);
Query OK, 1 row affected (0.04 sec)

mysql> insert into t1 values(2);
Query OK, 1 row affected (0.01 sec)
```

（2）在另外一个 SQL 节点（192.168.7.55），查询 test 库中的 t1 表，结果如下：

```
mysql> select * from t1;
+------+
| id   |
+------+
|    1 |
|    2 |
+------+
2 rows in set (0.06 sec)
```

显然，两个 SQL 节点查询到的数据是一致的。

（3）在 SQL 节点 192.168.7.187 上将测试表 t1 的存储引擎改为 MyISAM，再次插入测试记录：

```
mysql> alter table t1 engine=myisam;
Query OK, 2 rows affected (0.89 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> insert into t1 values(3);
Query OK, 1 row affected (0.00 sec)
```

（4）在 SQL 节点 192.168.7.55 上再次查询表 t1，结果如下：

```
mysql> select * from t1;
ERROR 1412 (HY000): Table definition has changed, please retry transaction
```

可以发现，表 t1 已经无法查询。

（5）在 SQL 节点 192.168.7.187 上再次将 t1 的存储引擎改为 NDB：

```
mysql> alter table t1 engine=ndb;
Query OK, 3 rows affected (0.99 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

（6）在 SQL 节点 192.168.7.55 上再次查询，结果如下：

```
mysql> select * from t1;
+------+
| id   |
+------+
|    2 |
|    3 |
|    1 |
+------+
3 rows in set (0.00 sec)
```

显然，表 t1 的数据被再次同步到了数据节点。所有 SQL 节点又都可以正常查询数据。

2. 单点故障测试

对于任意一种节点，都存在单点故障的可能性。在 Cluster 的设置过程中，应该尽量对每一类节点设置冗余，以防止单点故障发生时造成的应用中断。对于管理节点，一般不需要特殊的配置，只需要将管理工具和配置文件放在多台主机上即可。下面将测试一下 SQL 节点和

数据节点的单点故障。

- SQL 节点发生单点故障。

对于上面的测试环境，我们设置了两个 SQL 节点，应用从两个节点对数据访问都可以得到一致的结果。如果有一个节点故障，系统还会正常运行吗？具体的测试过程如下。

（1）将 SQL 节点 192.168.7.187 上的 MySQL 服务停止。

```
[zzx2@zzx ndb_2_fs]$ mysqladmin -uroot shutdown
STOPPING server from pid file /home/zzx2/mysql/data/zzx.pid
071211 11:50:31  mysqld ended

[1]+  Done                    ./bin/mysqld_safe  (wd: ~/mysql)
(wd now: ~/mysql/data/ndb_2_fs)
```

（2）查看一下 Cluster 的状态。

```
[zzx2@zzx ndb_2_fs]$ ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm> show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=2    @192.168.7.187  (Version: 5.1.11, Nodegroup: 0, Master)
id=3    @192.168.7.55  (Version: 5.1.11, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.7.187  (Version: 5.1.11)

[mysqld(API)]   3 node(s)
id=4 (not connected, accepting connect from 192.168.7.187)
id=5    @192.168.7.55  (Version: 5.1.11)
id=6 (not connected, accepting connect from any host)
```

从粗体字中可以看出，SQL 节点 192.168.7.187 已经断开，但是另外一个 SQL 节点 192.168.7.55 仍然处于正常连接中。（3）从节点 192.168.7.55 上查看表 t1，结果如下。

```
mysql> select * from t1;
+------+
| id   |
+------+
|    2 |
|    4 |
|    3 |
|    1 |
+------+
4 rows in set (0.04 sec)
```

显然，SQL 节点的单点故障并没有引起数据的查询故障。对于应用说，需要改变的就是将以前对故障节点的访问改为对非故障节点的访问。

- 数据节点的单点故障。

在这个测试环境中，数据节点也是有两个，那么它们对数据的存储是互相镜像还是一份数据分成几块存储呢（就象是磁盘阵列的 RAID1 还是 RAID0）？这个答案关键在于配置文件中[NDBD DEFAULT]组中的 NoOfReplicas 参数，如果这个参数等于 1，表示只有一份数据，但是分成 n 块分别存储在 n 个数据节点上；如果等于 2，则表示数据被分成 n/2 块，每块数据都有两个备份，这样即使有任意一个节点发生故障，只要它的备份节点正常，系统就可以正常运行。

在下面例子中，先将两个数据节点之一停掉，观察一下表 t1 能否正常访问；然后将 NoOfReplicas 配置改为 2，这时，数据节点实际上已经互为镜像，保存了两份。这时再停掉任意一个数据节点，观察表 t1 还能否正常访问。

（1）将数据节点 192.168.7.187 上的 NDB 进程停止。

```
[zzx2@zzx ~]$ ps -ef|grep 'ndbd'
zzx2          31271          1    0  10:50  ?                00:00:00  ndbd  --initial
--ndb-connectstring=192.168.7.187:1186
zzx2          31272  31271    0  10:50  ?                00:00:06  ndbd  --initial
--ndb-connectstring=192.168.7.187:1186
zzx2      8536  8508  0 17:54 pts/1    00:00:00 grep ndbd
[zzx2@zzx ~]$ kill 31271 31272
```

（2）在任意一个 SQL 节点（这里用 192.168.7.55）查看表 t1 数据，结果如下。

```
mysql> select * from t1;
ERROR 1296 (HY000): Got error 4009 'Cluster Failure' from NDBCLUSTER
```

显然，这里的 Cluster 出现了故障，无法访问。

（3）将配置文件中的 NoOfReplicas 改为 2，按照前面所述步骤重新启动集群。

```
[zzx2@zzx mysql-cluster]$ more config.ini
[NDBD DEFAULT]
NoOfReplicas=2
DataMemory=500M
IndexMemory=300M
[TCP DEFAULT]
portnumber=2202
.....省略掉其他参数
```

（4）此时，停掉任何一个数据节点的数据（这里选 192.168.7.187）。

```
[zzx2@zzx mysql]$ ps -ef|grep ndbd
zzx2           5668          1    0  14:39  ?                00:00:00  ndbd  --initial
--ndb-connectstring=192.168.7.187:1186
zzx2           5669    5668    0  14:39  ?                00:00:00  ndbd  --initial
--ndb-connectstring=192.168.7.187:1186
zzx2      5875  5384  0 14:42 pts/2    00:00:00 grep ndbd
[zzx2@zzx mysql]$ kill 5668 5669
```

（5）再次从任意一个 SQL 节点查询表 t1。

```
mysql> select * from t1;
+------+
| id   |
+------+
|    3 |
```

```
|    2 |
|    1 |
|    4 |
+------+
4 rows in set (0.04 sec)
```

显然，结果依然正确。数据节点的冗余同样防止了单点故障的发生。

30.3.3 Cluster 的关闭

Cluster 的关闭命令很简单，只需要在 shell 下执行如下命令即可：

```
[zzx2@zzx mysql-cluster]$ ndb_mgm -e shutdown
Connected to Management Server at: localhost:1186
Node 2: Cluster shutdown initiated
Node 3: Cluster shutdown initiated
Node 3: Node shutdown completed.
Node 2: Node shutdown completed.
2 NDB Cluster node(s) have shutdown.
Shutdown of NDB Cluster management server failed.
*     0: No error
*        Executing: ndb_mgm_stop2
```

也可以用 ndb_mgm 工具进入管理界面后，使用 shutdown 命令关闭：

```
ndb_mgm> shutdown
Node 2: Cluster shutdown initiated
Node 3: Cluster shutdown initiated
Node 2: Node shutdown completed.
Node 3: Node shutdown completed.
2 NDB Cluster node(s) have shutdown.
Shutdown of NDB Cluster management server failed.
*     0: No error
*        Executing: ndb_mgm_stop2
```

需要注意的是，集群关闭后，SQL 节点的 MySQL 服务并不会停止。

30.4 维护 Cluster

经过前面的介绍，大家已经可以基本掌握 Cluster 的使用方法，本节将给大家介绍 Cluster 的日常维护。熟练掌握这些常用的维护方法，将会帮助用户更好地管理和使用 Cluster。

30.4.1 数据备份

在前面章节已经介绍过如何使用 mysqldump 工具对数据库进行逻辑备份。这个方法同样适合于 MySQL Cluster，备份方法和其他存储引擎一样，唯一的区别是在任意一个 SQL 节点上都可以执行，这里不再赘述。

本节主要介绍 Cluster 的物理备份方法。以上文中的测试环境为例，启动管理服务器（ndb_mgm），并执行“start backup”命令启动备份。

```
ndb_mgm> start backup;
Waiting for completed, this may take several minutes
Node 3: Backup 1 started from node 1
Node 3: Backup 1 started from node 1 completed
 StartGCP: 3010 StopGCP: 3013
 #Records: 4107 #LogRecords: 0
 Data: 67440 bytes Log: 0 bytes
```

在备份日志中，需要注意“Backup 1”，它表示该备份的唯一 ID，如果做第二次备份，备份 ID 会变成“Backup 2”。当日志中显示“Backup 1 started from node 1 completed”的时候，本次备份结束。备份的数据保存在每个数据节点下，具体备份路径是：$MYSQL_HOME/data/BACKUP/BACKUP-备份 ID。

上面的备份例子在两个数据节点下都可以看到备份数据。

对于数据节点 192.168.7.187，可以看到以下数据文件：

```
[zzx2@zzx BACKUP-1]$ pwd
/home/zzx2/mysql/data/BACKUP/BACKUP-1
[zzx2@zzx BACKUP-1]$ ls -ltr
total 60
-rw-rw-r--  1 zzx2 zzx2    44 Dec 12 16:48 BACKUP-1.2.log
-rw-rw-r--  1 zzx2 zzx2  6580 Dec 12 16:48 BACKUP-1.2.ctl
-rw-rw-r--  1 zzx2 zzx2 34920 Dec 12 16:48 BACKUP-1-0.2.Data
```

对于数据节点 192.168.7.55，可以看到以下数据文件：

```
[zzx2@test55 BACKUP-1]$ pwd
/home1/zzx2/mysql/data/BACKUP/BACKUP-1
[zzx2@test55 BACKUP-1]$ ls -ltr
total 60
-rw-rw-r--  1 zzx2 zzx2    44 Dec 12 17:50 BACKUP-1.3.log
-rw-rw-r--  1 zzx2 zzx2  6580 Dec 12 17:50 BACKUP-1.3.ctl
-rw-rw-r--  1 zzx2 zzx2 32936 Dec 12 17:50 BACKUP-1-0.3.Data
```

细心的读者可能发现，以上两个节点上的数据文件名不一样，一个包含“.2”，另一个包含“.3”，这个数字表明了此备份哪个数据节点上的数据（192.168.7.187 为节点 2，192.168.7.55 为节点 3）。

对于大数据量的备份，MySQL Cluster 还提供了几个备份的参数可供调整，这些参数需要写在 config.ini 的[NDBD DEFAULT]或者[NDBD]组中，对各参数的具体说明如下。

- BackupDataBufferSize：将数据写入磁盘之前用于对数据进行缓冲处理的内存量。
- BackupLogBufferSize： 将日志记录写入磁盘之前用于对其进行缓冲处理的内存量。
- BackupMemory：在数据库节点中为备份分配的总内存。它应是分配给备份数据缓冲的内存和分配给备份日志缓冲的内存之和。
- BackupWriteSize：每次写入磁盘的块大小，适用于备份数据缓冲和备份日志缓冲。

30.4.2 数据恢复

对于用“start backup”进行备份的 Cluster，必须使用 ndb_restore 工具进行数据恢复。ndb_restore 是 MySQL Cluster 带的管理工具，在 shell 中执行，而不是 ndb_mgm 工具中的一个命令。在前面的例子中，再增加一些测试数据，这样能更加真实地模拟实际环境。然后进

行备份，备份 ID 为 3。

备份前数据为：

```
mysql> select count(1) from t1;
+----------+
| count(1) |
+----------+
|    65536 |
+----------+
1 row in set (0.00 sec)
```

进行数据恢复的操作步骤如下。

（1）在数据节点 2（数据节点 192.168.7.187）的 shell 命令行中执行如下命令：

```
[zzx2@zzx mysql-cluster]$ ndb_restore -b 3 -n 2 -c host=192.168.7.187:1186 -m -r /home/zzx2/mysql/data/BACKUP/BACKUP-3
Backup Id = 3
Nodeid = 2
backup path = /home/zzx2/mysql/data/BACKUP/BACKUP-3
Ndb version in backup files: Version 5.1.11
Connected to ndb!!
Successfully restored table test/def/t1
Successfully restored table event REPL$test/t1
_____________________________________________________
Processing data in table: cluster/def/NDB$BLOB_2_3(3) fragment 0
_____________________________________________________
Processing data in table: sys/def/NDB$EVENTS_0(1) fragment 0
_____________________________________________________
Processing data in table: test/def/t1(5) fragment 0
_____________________________________________________
Processing data in table: cluster/def/schema(2) fragment 0
_____________________________________________________
Processing data in table: cluster/def/apply_status(4) fragment 0
_____________________________________________________
Processing data in table: sys/def/SYSTAB_0(0) fragment 0
Restored 32556 tuples and 0 log entries

NDBT_ProgramExit: 0 - OK
```

其中，命令行中的各参数含义分别如表 30-2 所示。

表 30-2　　相关参数及其说明

参数	说明
-b	备份 id
-n	节点 id
-m	恢复表定义
-r	恢复路径
-c	Cluster 管理器连接串

因为是第一个节点恢复，所以需要添加参数-m 来恢复表定义，这样在其他节点恢复的时候就不需要再加此参数，否则会报如下错误：

```
Table or index with given name already exists
Restore: Failed to restore table: cluster/def/NDB$BLOB_2_3 ... Exiting
```

（2）从节点 4（SQL 节点 192.168.7.187）登录数据库，查看表 t1 数据。

```
mysql> select count(1) from t1;
+----------+
| count(1) |
+----------+
|    32556 |
+----------+
1 row in set (0.01 sec)
```

可以发现，由于只从一个数据节点恢复了数据，所以数据量只有备份时的一半左右。此时再从另外一个数据节点恢复数据。

（3）在节点 3（数据节点 192.168.7.55）的 shell 命令行中执行如下命令：

```
[zzx2@test55 BACKUP-1]$ ndb_restore -b 3 -n 3 -c host=192.168.7.187:1186 -r /home1/zzx2/mysql/data/BACKUP/BACKUP-3
Backup Id = 3
Nodeid = 3
backup path = /home1/zzx2/mysql/data/BACKUP/BACKUP-3
Ndb version in backup files: Version 5.1.11
Connected to ndb!!
_____________________________________________________
Processing data in table: cluster/def/NDB$BLOB_2_3(3) fragment 1
_____________________________________________________
Processing data in table: sys/def/NDB$EVENTS_0(1) fragment 1
_____________________________________________________
Processing data in table: test/def/t1(5) fragment 1
_____________________________________________________
Processing data in table: cluster/def/schema(2) fragment 1
_____________________________________________________
Processing data in table: cluster/def/apply_status(4) fragment 1
_____________________________________________________
Processing data in table: sys/def/SYSTAB_0(0) fragment 1
Restored 32980 tuples and 0 log entries

NDBT_ProgramExit: 0 - OK
```

（4）再次从节点 4（SQL 节点 192.168.7.187）登录数据库，查看表 t1 数据。

```
mysql> select count(1) from t1;
+----------+
| count(1) |
+----------+
|    65536 |
+----------+
```

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

此时，数据已经完全恢复正常，恢复过程结束。

30.4.3 日志管理

MySQL Cluster 提供了两种日志，分别是集群日志（clusterlog）和节点日志（node log）。前者记录了所有 Cluster 节点生成的日志，后者仅仅记录了数据节点的本地事件。在大多数情况下，我们都推荐使用集群日志，因为它在一个地方记录了所有节点的数据，更便于进行管理。节点日志一般只在开发过程中使用，或者用来调试程序代码。

clusterlog 一般记录在和配置文件（config.ini）同一个目录下，文件名格式为 ndb_<nodeid>_cluster.log，其中 nodeid 为管理节点号。

下面是测试环境中摘录的一段 clusterlog：

```
2007-12-12 18:00:48 [MgmSrvr] ALERT    -- Node 2: Node 6 Disconnected
2007-12-12 18:00:48 [MgmSrvr] INFO     -- Node 2: Communication to Node 6 closed
2007-12-12 18:00:48 [MgmSrvr] ALERT    -- Node 3: Node 6 Disconnected
2007-12-12 18:00:48 [MgmSrvr] INFO     -- Node 3: Communication to Node 6 closed
2007-12-12 18:00:48 [MgmSrvr] INFO     -- Mgmt server state: nodeid 6 freed,
m_reserved_nodes 0000000000000012.
2007-12-12 18:00:51 [MgmSrvr] INFO     -- Node 3: Communication to Node 6 opened
2007-12-12 18:00:52 [MgmSrvr] INFO     -- Node 2: Communication to Node 6 opened
```

可以使用 ndb_mgm 客户端管理工具打开或者关闭日志，具体操作如下。

（1）在 shell 中执行 ndb_mgm 命令。

```
[zzx2@zzx mysql-cluster]$ ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm>
```

（2）执行 clusterlog info 命令查看当前日志状态。

```
ndb_mgm> clusterlog info
Connected to Management Server at: localhost:1186
Severities enabled: INFO WARNING ERROR CRITICAL ALERT
```

（3）当前日志是打开的，用 clusterlog off 命令关闭日志。

```
ndb_mgm> clusterlog off
Cluster logging is disabled
```

（4）再次查看日志状态，发现已经关闭。

```
ndb_mgm> clusterlog info
Cluster logging is disabled.
```

（5）执行 clusterlog on 命令将再次打开日志。

```
ndb_mgm> clusterlog on
Cluster logging is enabled.
ndb_mgm> clusterlog info
Severities enabled: INFO WARNING ERROR CRITICAL ALERT
```

Cluster 中的日志有很多类型，我们可以按照如下类别进行过滤，使得日志只记录我们关心的东西。

- Category（类别）：可以是下述值之一，STARTUP、SHUTDOWN、STATISTICS、CHECKPOINT、

NODERESTART、CONNECTION、ERROR 或 INFO。这些类别包含的事件很多，这里就不再赘述，有兴趣的读者可以查阅 MySQL 相关文档。

- Priority（优先级）：由从 1～15 的数字表示，“1”表示“最重要”，而“15”表示“最不重要”。每种 Category 都有一个默认的优先级阈值，如表 30-3 所示。优先级阈值以下的日志将被记录，反之，优先级阈值以上的日志不会被记录。

表 30-3　　MySQL Cluster 中不同类别日志的默认优先级阈值

类别	默认阈值（所有数据节点）
STARTUP	7
SHUTDOWN	7
STATISTICS	7
CHECKPOINT	7
NODERESTART	7
CONNECTION	7
ERROR	15
INFO	7

- Severity Level（严重级别）：可以是下述值之一，ALERT、CRITICAL、ERROR、WARNING、INFO 或 DEBUG。这些值的具体含义如表 30-4 所示。

表 30-4　　MySQL Cluster 中不同级别的事件定义

严重级别	事件定义
ALERT	应立刻更正的状况，如损坏的系统数据库
CRITICAL	临界状况，如设备错误或资源不足
ERROR	应予以更正的状况，如配置错误等
WARNING	不能称其为错误的状况，但仍需要特别处理
INFO	通报性消息
DEBUG	调试消息，用于 NDB Cluster 开发

这 3 种分类让用户可以从 3 个不同的角度来对日志进行过滤。过滤的方法可以用 ndb_mgm 工具来完成，具体设置方法如下。

- node_id CLUSTERLOG category=threshold：用小于或等于 threshold 的优先级将 category 事件记录到 Cluster 日志。node_id 可以为 ALL（所有节点）或者只指定某个节点。
- CLUSTERLOG TOGGLE severity_level：使得指定的 severity_level 打开或者关闭。

例如，要将测试环境中的节点 2 的 STARTUP 事件只记录级别为 3 以下的日志，可以进入 ndb_mgm 后执行如下命令：

```
ndb_mgm> 2 clusterlog startup=3
Executing CLUSTERLOG STARTUP=3 on node 2 OK!
```

如果要在 Cluster 日志中过滤掉 DEBUG 和 INFO 信息，可以执行如下命令：

```
ndb_mgm> clusterlog toggle  debug info
DEBUG disabled
INFO disabled
```

然后查看日志状态，发现 DEBUG 和 INFO 信息已经不存在了：

```
ndb_mgm> clusterlog info
Severities enabled: WARNING ERROR CRITICAL ALERT
```

30.5 小结

本章介绍了 MySQL Cluster（集群）的概念和架构，并通过几个实例详细讨论了 Cluster 的配置方法、启动关闭、常用维护方法等内容。MySQL Cluster 目前还有很多不足的地方，本章并没有深入讨论，因为它的版本更新速度非常快，功能也在不断地完善。期望 5.1 版本正式发行的时候能带给我们更多的惊喜！

第31章 MySQL 常见问题和应用技巧

在 MySQL 日常开发或者维护工作中，用户经常会遇到各种各样的故障或者问题，比如密码丢失、表损坏等。本章总结了一些常见的问题，希望对读者在遇到类似问题时有所帮助。

31.1 忘记 MySQL 的 root 密码

经常会有朋友或者同事问起，MySQL 的 root 密码忘了，不知道改怎么办。其实解决方法很简单，这在前面的章节中也曾提及，下面是详细的操作步骤。

（1）登录到数据库所在服务器，手工 kill 掉 MySQL 进程：

```
kill `cat /mysql-data-directory/hostname.pid`
```

其中，/mysql-data-directory/hostname.pid 指的是 MySQL 数据目录下的.pid 文件，它记录了 MySQL 服务的进程号。（2）使用--skip-grant-tables 选项重启 MySQL 服务：

```
[root@localhost mysql]#  ./bin/mysqld_safe --skip-grant-tables --user=zzx &
[1] 20881
[root@localhost mysql]# Starting mysqld daemon with databases from /home/zzx/mysql/data
```

其中--skip-grant-tables 选项前面曾经介绍过，意思是启动 MySQL 服务的时候跳过权限表认证。启动后，连接到 MySQL 的 root 将不需要口令。（3）用空密码的 root 用户连接到 MySQL，并且更改 root 口令：

```
[root@localhost mysql]# mysql -uroot
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 53
Server version: 5.0.41-community-log MySQL Community Edition (GPL)

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

mysql>
mysql> set password = password('123');
ERROR 1290 (HY000): The MySQL server is running with the --skip-grant-tables option so it
cannot execute this statement
```

```
mysql> update user set password=password('123') where user='root' and host='localhost';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

此时，由于使用了--skip-grant-tables 选项启动，使用“set password”命令更改密码失败，直接更新 user 表的 password 字段后更改密码成功。

（4）刷新权限表，使得权限认证重新生效：

```
mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

（5）重新用 root 登录时，必须输入新口令：

```
[root@localhost mysql]# mysql -uroot
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: NO)
[root@localhost mysql]# mysql -uroot -p123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6 to server version: 5.1.11-beta-log

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

mysql>
```

注意：在 MySQL 中，密码丢失后无法找回，只能通过上述方式修改密码。

31.2 如何处理 MyISAM 存储引擎的表损坏

如果大家经常使用MySQL，可能会遇到过MyISAM存储引擎的表损坏的情况。一张损坏的表的症状通常是查询意外中断并且能看到下述错误：

- “tbl_name.frm”被锁定不能更改；
- 不能找到文件“tbl_name.MYI”（Errcode：nnn）；
- 文件意外结束；
- 记录文件被毁坏；
- 从表处理器得到错误 nnn。

遇到这种问题，通常的解决方法有以下两种。

（1）方法一

使用 MySQL 自带的 myisamchk 工具进行修复，此工具前面曾经介绍过，是专门用来修复 MyISAM 的表的工具。恢复命令如下：

```
myisamchk -r tablename
```

其中-r 参数的含义是 recover，上面的方法几乎能解决所有问题，如果不行，则使用：

```
myisamchk -o tablename
```

其中-o 参数的含义是--safe-recover，可以进行更安全的修复。

（2）方法二

使用MySQL的CHECK TABLE和REPAIR TABLE命令一起进行修复，CHECK TABLE用来检查表

是否有损坏；REPAIR TABLE用来对坏表进行修复。这两个命令的语法如下：

- CHECK TABLE tbl_name [, tbl_name] ... [option] ...
  option = {QUICK | FAST | MEDIUM | EXTENDED | CHANGED}
- REPAIR [LOCAL | NO_WRITE_TO_BINLOG] TABLE
  tbl_name [, tbl_name] ... [QUICK] [EXTENDED] [USE_FRM]

关于以上选项的详细说明，有兴趣的读者可以参考 MySQL 的帮助文档。

31.3 MyISAM 表超过 4GB 无法访问的问题

在 MySQL 5.0 版本之前，MyISAM 存储引擎默认的表大小只支持到 4GB，可以用以下命令来查看：

```
[zzx@bj34 ams]$ myisamchk -dv t1

MyISAM file:         t1
Record format:       Packed
Character set:       gbk_chinese_ci (28)
File-version:        1
Creation time:       2007-07-10 16:50:47
Recover time:        2007-07-10 16:51:15
Status:              open,changed
Data records:              224299  Deleted blocks:                0
Datafile parts:            224299  Deleted data:                  0
Datafile pointer (bytes):       4  Keyfile pointer (bytes):       4
Datafile length:         63059248  Keyfile length:          9486336
Max datafile length:   4294967294  Max keyfile length: 4398046510079
Recordlength:              10284
```

粗体字的第一行显示了当前数据文件的大小（Datafile Length）和索引文件的大小（Keyfile Length），第二行显示了最大数据文件的大小（Max Datafile Length）和最大索引文件的大小（Max Keyfile Length）。可以看出，表的最大数据文件 size 是 4294967294 字节，也就是 4GB。可以用下面命令对数据文件的最大 size 进行扩充：

```
alter table tbl_name MAX_ROWS=1000000000 AVG_ROW_LENGTH=15000;
```

此命令可以修改表的最大记录数和平均记录长度，因此可以修改数据文件的最大 size。对上面的测试表修改后，再次进行查看：

```
mysql> alter table t1 MAX_ROWS=10000000000 AVG_ROW_LENGTH=15000;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> exit
Bye
[zzx@bj34 test]$ myisamchk -dv t1

MyISAM file:         t1
Record format:       Packed
```

```
Character set:        gbk_chinese_ci (28)
File-version:         1
Creation time:        2007-07-10 16:50:47
Recover time:         2007-07-10 16:51:15
Status:               open,changed
Data records:               224299  Deleted blocks:                0
Datafile parts:             224299  Deleted data:                  0
Datafile pointer (bytes):        4  Keyfile pointer (bytes):       4
Datafile length:          63059248  Keyfile length:           9486336
Max datafile length: 281474976710654  Max keyfile length:   17179868159
Recordlength:                10284
```

果然，数据文件的最大 size 已经变得相当大了（281474976710654 字节，约 280TB）。

31.4 数据目录磁盘空间不足的问题

很多系统在正式上线后，随着数据量的不断增加，会发现数据目录下的可用空间越来越小，从而给应用造成了安全隐患。对于这类问题，用户可以根据不同的情况采取不同的措施进行解决。

31.4.1 对于 MyISAM 存储引擎的表

对于 MyISAM 存储引擎的表，在建表时可以用如下选项分别指定数据目录和索引目录存储到不同的磁盘空间，而默认会同时放在数据目录下：

```
DATA DIRECTORY = 'absolute path to directory'
INDEX DIRECTORY = 'absolute path to directory'
```

下例中创建了表 t1，并使得数据目录和索引目录不同：

```
mysql> create table t1 (id int,name varchar(10))
    -> data directory='/var/lib/mysql/data'
    -> index directory='/var/lib/mysql/index';
Query OK, 0 rows affected (0.07 sec)
```

从操作系统上看表 t1 的物理文件如下：

```
lrwxrwxrwx  1 mysql mysql     27 Dec  4 14:29 t1.MYI -> /var/lib/mysql/index/t1.MYI
lrwxrwxrwx  1 mysql mysql     26 Dec  4 14:29 t1.MYD -> /var/lib/mysql/data/t1.MYD
-rw-rw----  1 mysql mysql   8586 Dec  4 14:29 t1.frm
```

很显然，对数据文件和索引文件指定了两个不同的存放目录，实际上是在操作系统上创建了两个符号链接，指向了不同的目录。读者可能会想到，既然创建的时候可以指定数据索引分离，那么在使用过程中是不是也可以随时对这些路径进行再次变更呢？答案是否定的。在下例中，将尝试把 t1 的数据和索引目录放在一起：

```
mysql> alter table t1 index directory='/var/lib/mysql/data';
Query OK, 0 rows affected, 1 warning (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
mysql> show warnings;
+---------+------+-------------------------------+
| Level   | Code | Message                       |
```

```
+---------+------+------------------------------+
| Warning |    0 | INDEX DIRECTORY option ignored |
+---------+------+------------------------------+
1 row in set (0.00 sec)
```

从 warning 中的提示可以看出，修改索引路径并没有成功。

对于上面这种情况，如果表已经创建，还可以将表的数据文件和索引文件 mv 到磁盘充足的分区上，然后在原文件处创建符号链接即可。当然，mv 前必须要停机或者将表锁定，以防止表的更改。

31.4.2 对于 InnoDB 存储引擎的表

对于 InnoDB 存储引擎的表，因为数据文件和索引文件是存放在一起的，所以无法将它们进行分离。当磁盘空间出现不足时，可以增加一个新的数据文件，这个文件放在有充足空间的磁盘上。具体实现方法是在参数 innodb_data_file_path 中增加此文件，路径写为新磁盘的绝对路径。例如，如果/home 下空间不足，希望在/home1 下新增加一个可自动扩充数据文件，那么参数就可以这么写：

```
innodb_data_file_path =     /home/ibdata1:2000M;/home1/ibdata2:2000M:autoextend
```

当然，参数修改后必须重启数据库才可以生效。

31.5 DNS 反向解析的问题

在 MySQL 5.0 以前的版本中执行 show processlist 命令的时候，有时会出现很多进程，类似于以下情况：

```
unauthenticated user | 192.168.5.71:57857 | NULL | Connect | NULL | login | NULL
```

这些进程会累计得越来越多，并且不会消失，应用无法正常响应，导致系统瘫痪。造成这种现象的原因是什么呢？

原来，MySQL 默认情况下，对于远程连接过来的 IP 地址，会进行域名的逆向解析，如果系统的 hosts 文件中没有与之对应的域名，MySQL 就会将此连接认为是无效用户，所以在进程中出现“unauthenticated user”并导致进程阻塞。

解决的方法很简单，在启动的时候加上--skip-name-resolve 选项，则 MySQL 就可以跳过域名解析过程，避免上述问题。在 MySQL 5.0 以后版本，默认都会跳过域名逆向解析。

31.6 mysql.sock 丢失后如何连接数据库

在 MySQL 服务器本机上连接数据库时，经常会出现 mysql.sock 不存在，导致无法连接的问题。这是因为如果指定 localhost 作为一个主机名，则 mysqladmin 默认使用 UNIX 套接字文件连接，而不是 TCP/IP。而这个套接字文件（一般命名为 mysql.sock）经常会因为各种原因而被删除。从 MySQL 4.1 开始，通过--protocol= TCP | SOCKET | PIPE | MEMORY}选项，用户可以显式地指定连接协议 ，下面演示了使用 UNIX 套接字失败后，使用 TCP 协议连接成功的例子：

```
UNIX 套接字连接：
[zzx@zzx mysql]$ mysql -uroot
```

```
    ERROR 2002 (HY000): Can't connect to local MySQL server through socket
'/home/zzx/mysql/mysql.sock' (2)

    TCP 连接:
    [zzx@zzx mysql]$ mysql --protocol=TCP -uroot -p -P3307 -hlocalhost
    Enter password:
    Welcome to the MySQL monitor.  Commands end with ; or \g.
    Your MySQL connection id is 73 to server version: 5.0.15-standard
    Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
    mysql>
```

31.7 同一台服务器运行多个 MySQL 数据库

在很多情况下，由于硬件资源的局限，用户通常需要在一台服务器上安装多个 MySQL 数据库，从而为多个应用提供服务。在这种情况下，通常可以使用以下方法进行安装。

将每个 MySQL 安装在不同的用户下面，例如 mysql1 和 mysql2，在每个用户下面，分别执行如下操作：

```
export MYSQL_HOME=/home/mysql1/mysql
shell> groupadd mysql
shell> useradd -g mysql mysql1
shell> cd /home/mysql1
shell>tar -xzvf    /home/mysql1/mysql-VERSION-OS.tar.gz
shell> ln -s mysql-VERSION-OS.tar.gz mysql
shell> cd mysql
cp support-files/my-large.cnf(根据实际情况选择)   ./my.cnf
vi my.cnf ，主要修改[client]和[mysqld]下面的 port 和 socket，并指定字符集，例如：
[client]
port                = 3307
socket              = /home/mysql1/mysql/data/mysql.sock
# The MySQL server
[mysqld]
default-character-set = utf8
port                = 3307
socket              = /home/mysql1/mysql/data/mysql.sock
。。。。。。
shell> scripts/mysql_install_db --user=mysql1
shell> chown -R root:mysql    .
shell> chown -R mysql1:mysql data
shell> bin/mysqld_safe --user=mysql &
```

mysql2 用户执行的和 mysql1 类似，区别在于指定不同的 MYSQL_HOME，以及不同的 port 和 socket 即可

31.8 客户端怎么访问内网数据库

在很多情况下，MySQL 数据库都会安装在企业内网，而有很多图形化客户端工具通常安装在本机，如果两个网络无法直接联通，那么客户端通常也无法正常连接数据库。很多数据库厂商都提供了相应的功能，例如 Oracle，客户端可以通过连接管理器（CMAN）来访问内网中的 Oracle 数据库，MySQL 能实现类似功能吗？答案是肯定的。来看下面的例子。

假设有如下服务器和客户端：

- 中转服务器 IP 为 202.108.15.169（192.168.161.43）。
- 内网服务器 IP 为 192.168.161.30，在端口 3313 上启动着 MySQL 服务。
- 客户端为 192.168.52.239（IP），操作系统为 Windows，且安装了 SecureCRT（一个远程连接工具）。

其中客户端可以和中转服务器联通，但无法和内网服务器连通。

首先，在客户端运行 MySQL Query Brower（一个图形化工具，在前面的章节中已介绍过），直接连接内网数据库，测试是否能连接。其连接界面和结果分别如图 31-1 和图 31-2 所示。

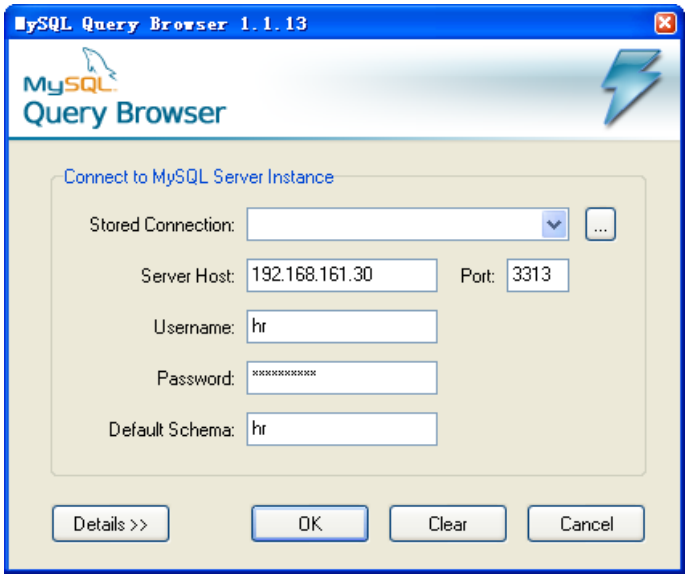


图 31-1　客户端登录界面

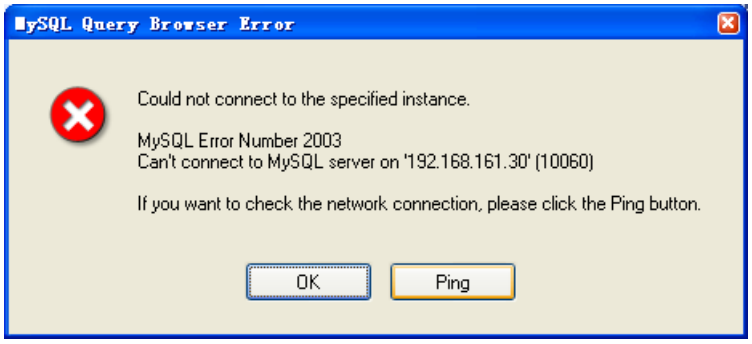


图 31-2　客户端登录失败界面

显然，因为网络不能直接连通，所以连接失败。

下面介绍两种方法来实现客户端对数据库的连接。

方法 1：使用 secureCRT 客户端工具。

secureCRT 是一个可以通过多种协议连接远程服务器的客户端工具，具体介绍可以到其官方网站（http://www.vandyke.com/）进行查看。通过 SecureCRT 工具连接到中转服务器，并创

建 SSH Turnal，具体操作步骤如下。

（1）单击 Session 的属性，选择“Connection”➔“Port Forwarding”选项，进入如图 31-3 所示的界面：

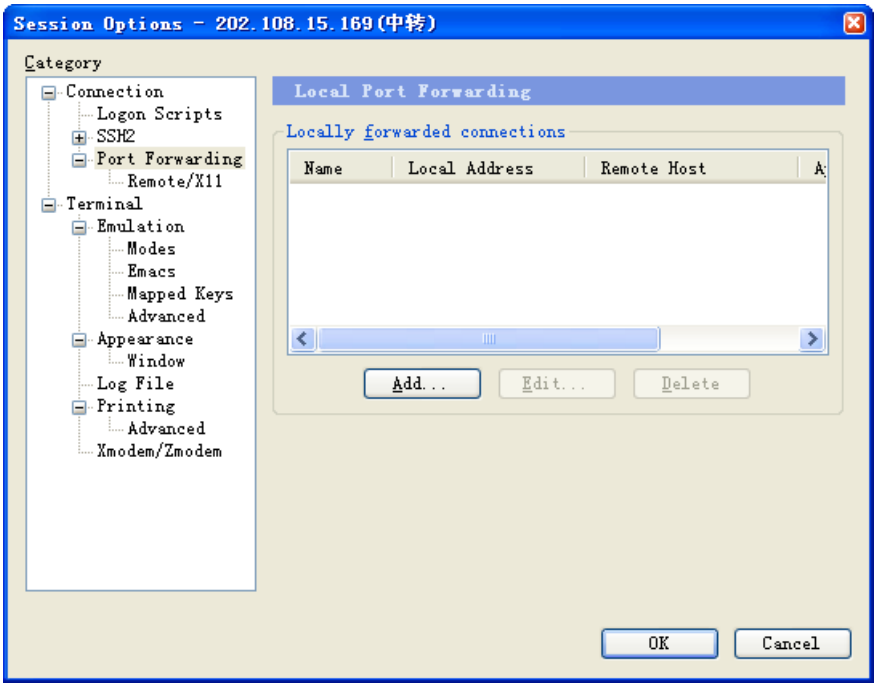


图 31-3　Session Options 设置界面

（2）单击“Add”按钮，出现如图 31-4 所示的“Local Port Forwarding Properties”对话框，在“Name”文本框中随便输入名字；“Local”栏下的“IP”文本框中输入 127.0.0.1，“Port”文本框中随便输入一个未使用的端口，例如 9987；“Remote”栏下的“Hostname”文本框中输入内网数据库的 IP 地址，这里是 192.168.161.30，而“Port”文本框中输入“3313”；单击“ok”按钮设置成功。

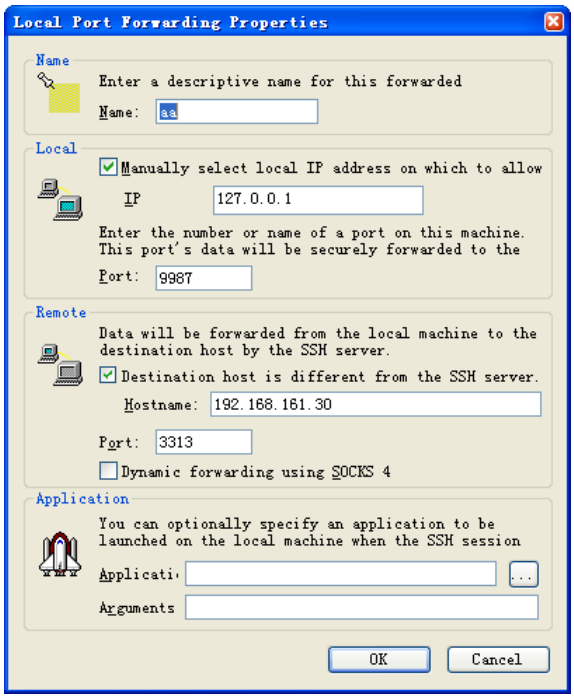


图 31-4 “Local Port Port Forwarding”对话框

（3）在内网数据库（其 IP 地址为 192.168.161.30）上增加一个用户 hr，“Host”设置为 192.168.161.43。

```
grant select on dbname.* to test@192.168.161.43 identified by '123';
```

（4）客户端重新登录，在“Server Host”文本框中填写本节 IP（127.0.0.1），“Port”文本框中填写“9987”，并在“Username”和“Password”文本框中输入上文中创建的用户名和密码，然后在“Default Schema”文本框中填入登录的数据库名称，如图 31-5 所示。

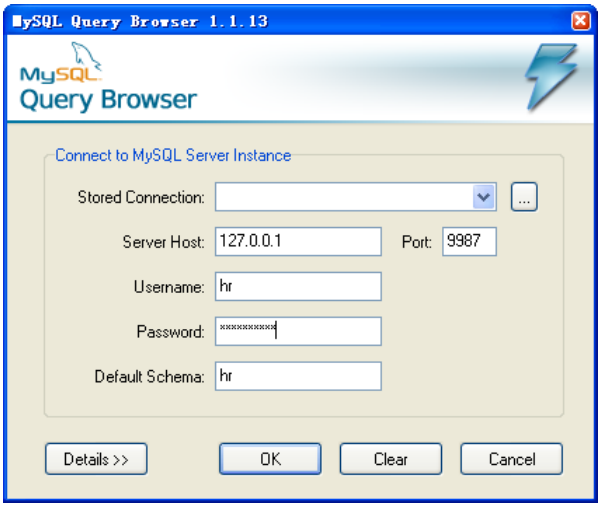


图 31-5 重新登录界面

（5）单击“OK”按钮，连接成功，进入数据库 hr，如图 31-6 所示。

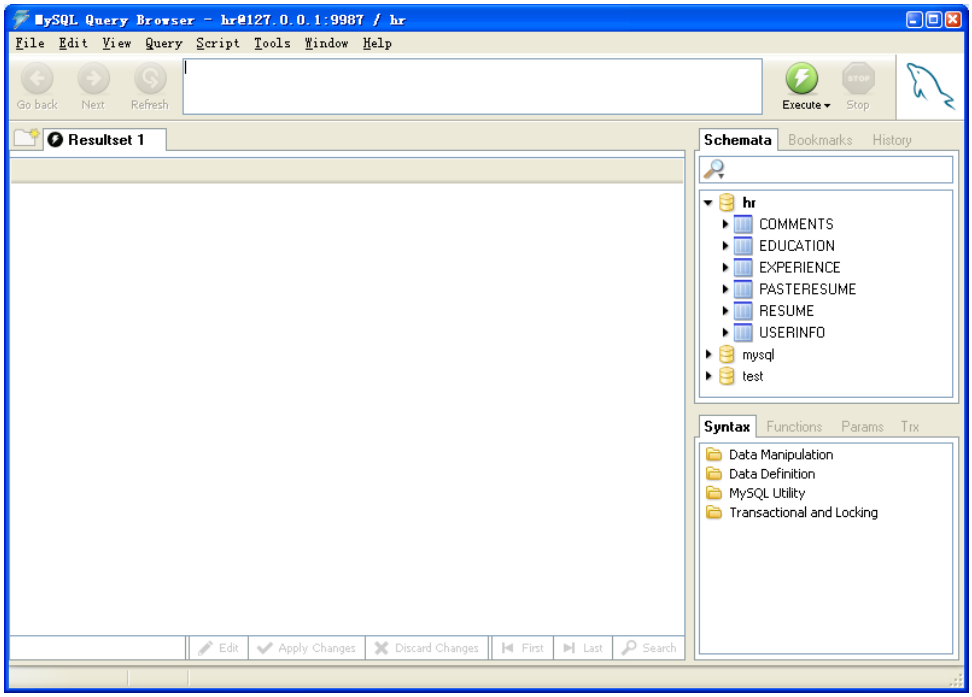


图 31-6 登录成功后的界面

方法 2：使用 MySQL Proxy（MySQL 代理）工具。

MySQL Proxy 是 MySQL AB 公司新开发的一个工具，写作本书的时候还是测试版。它的功能和 Oracle 的连接管理器（CMAN）有些类似，是位于客户端和服务器中间的一个小程序。但是它的功能比 CMAN 更多，除了可以进行连接的转发外，还可以进行查询的监控、过滤、

分析、负载均衡（Load Balance）、失败漂移（Failover）等其他更多的功能，具体功能的使用大家可以查看官方文档（http://forge.mysql.com/wiki/MySQL_Proxy）。这里我们仅仅介绍它对于连接转发功能的使用。

（1）在中转服务器（202.108.15.169）上下载 MySQL Proxy 的最新版本（http://dev.mysql.com/downloads/mysql-proxy/index.html），下载的方法和下载 MySQL 软件类似，这里就不再赘述。下载后的文件名为 mysql-proxy-0.6.0-linux-rhas4-x86.tar.gz。

（2） 解压下载的文件如下。

```
[zzx@zzx ~]$ tar xzvf mysql-proxy-0.6.0-linux-rhas4-x86.tar.gz
mysql-proxy-0.6.0-linux-rhas4-x86/
mysql-proxy-0.6.0-linux-rhas4-x86/sbin/
mysql-proxy-0.6.0-linux-rhas4-x86/sbin/mysql-proxy
mysql-proxy-0.6.0-linux-rhas4-x86/share/
. . . . . .
mysql-proxy-0.6.0-linux-rhas4-x86/share/mysql-proxy/xtab.lua
```

（3） 进入解压后的目录，发现有两个子目录 share 和 sbin。

```
[zzx ~]$ cd mysql-proxy-0.6.0-linux-rhas4-x86
 [zzx ~/mysql-proxy-0.6.0-linux-rhas4-x86]$ ls -l
总用量 16
drwxr-xr-x  4 zzx zzx 4096  9月 12 08:48 ./
drwx------  5 zzx zzx 4096  1月 11 15:33 ../
drwxr-xr-x  2 zzx zzx 4096  9月 12 08:48 sbin/
drwxr-xr-x  3 zzx zzx 4096  9月 12 08:48 share/
```

（4）其中 share 中有很多以.lua 为后缀的文件，这些是 MySQL Proxy 对客户端进行查询监控、分析等操作的脚本。sbin 目录中只有一个文件 mysql-proxy，用它来启动 MySQL Proxy 服务。

（5）执行./mysql-proxy --help-all 命令，查看 mysql-proxy 工具的使用方法。

```
[zzx ~/mysql-proxy-0.6.0-linux-rhas4-x86/sbin]$ ./mysql-proxy --help-all
Usage:
  mysql-proxy [OPTION...] - MySQL Proxy

Help Options:
  -?, --help                                  Show help options
  --help-all                                  Show all help options
  --help-admin                                Show options for the admin-module
  --help-proxy                                Show options for the proxy-module

admin module
  --admin-address=<host:port>                       listening address:port of internal
admin-server (default: :4041)

proxy-module
  --proxy-address=<host:port>                   listening address:port of the proxy-server
(default: :4040)
  --proxy-read-only-backend-addresses=<host:port>       address:port of the remote slave-server
```

```
(default: not set)
      --proxy-backend-addresses=<host:port>               address:port of the remote backend-servers
(default: 127.0.0.1:3306)
      --proxy-skip-profiling                                disables profiling of queries (default:
enabled)
      --proxy-fix-bug-25371                                 fix bug #25371 (mysqld > 5.1.12) for older
libmysql versions
      --proxy-lua-script=<file>                           filename of the lua script (default: not set)
      --no-proxy                                          Don't start proxy-server

    Application Options:
      -V, --version                                       Show version
      --daemon                                            Start in daemon-mode
      --pid-file=<file>                                   PID file in case we are started as daemon
```

这些参数中，对连接转发最重要的参数是--proxy-backend-addresses=<host:port>，它告诉 MySQL Proxy 要连接到的目标 MySQL 服务的 IP 和端口。

（6）启动 MySQL Proxy 服务，并指定--proxy-backend-addresses 为目标数据库的 IP 地址和端口。

```
    [zzx              ~/mysql-proxy-0.6.0-linux-rhas4-x86/sbin]$              ./mysql-proxy
--proxy-backend-addresses=192.168.161.30:3313 &
    [2] 3690
    [1]   Done                    ./mysql-proxy proxy-backend-addresses=192.168.161.30:3313
```

（7）用 netstat 观察一下启动的服务：

```
    [zzx ~/mysql-proxy-0.6.0-linux-rhas4-x86/sbin]$ netstat -nlp
    (Not all processes could be identified, non-owned process info
     will not be shown, you would have to be root to see it all.)
    Active Internet connections (only servers)
    Proto Recv-Q Send-Q Local Address               Foreign Address             State
PID/Program name
    tcp        0      0 0.0.0.0:4040                0.0.0.0:*                   LISTEN
3690/mysql-proxy
    tcp        0      0 0.0.0.0:4041                0.0.0.0:*                   LISTEN
3690/mysql-proxy
    ......
```

其中，4040 端口是默认的代理端口，客户端连接的时候直接指向此端口；4041 是默认的管理模块端口，这里不用管它。

（8）在 Windows 客户端启动 MySQL Query Browser，在登录界面的“Server Host”文本框中输入中转服务器的 IP 地址，“Port”文本框中输入 MySQL Proxy 的代理端口 4040，“Username”和“Password”输入实际数据库的用户名和密码，如图 31-7 所示。

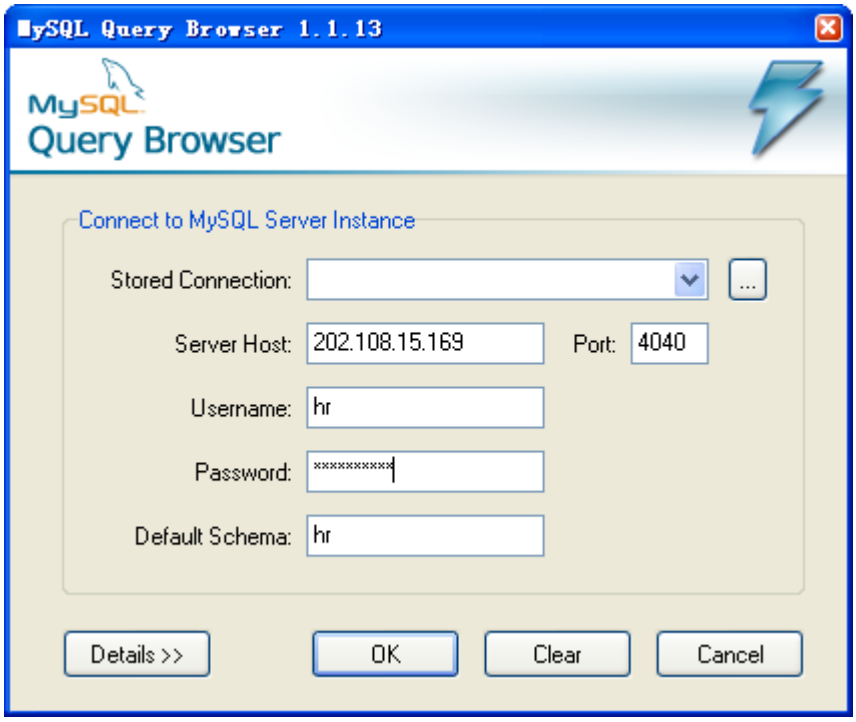


图 31-7　使用 MySQL Proxy 的登录界面

（9）单击“OK”按钮，成功登录数据库，如图 31-8 所示。

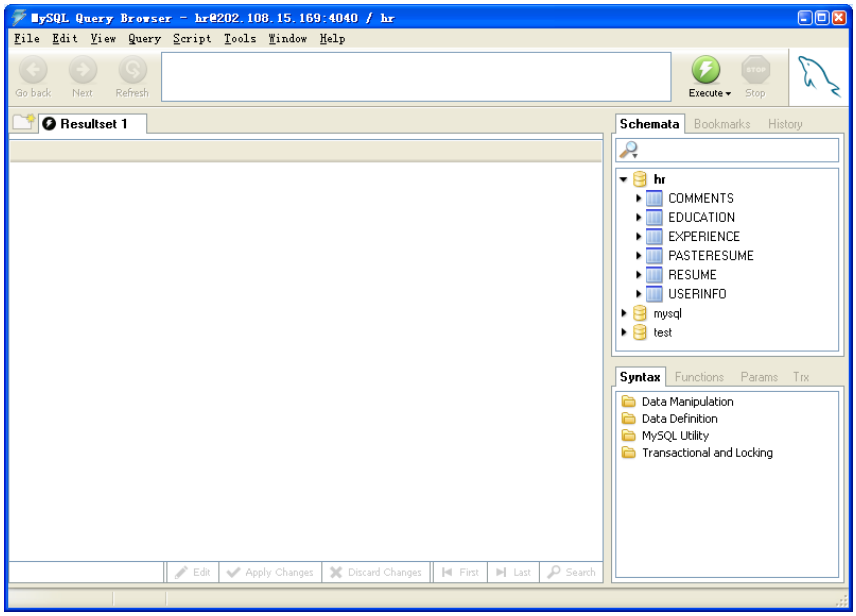


图 31-8　使用 MySQL Proxy 成功登录数据库后的界面

从方法 1 和方法 2 可以看出，方法 1 的优点是更通用，不仅能对 MySQL 服务进行连接的转发，而且还可以对其他类似的问题都可以采用这种办法进行解决，以达到客户端正常访问内网服务的目的。但是这种方法也有自己的缺点，就是必须要打开 secureCRT 相应的 SESSION，连接才可以生效；方法 2 的优点是不依赖于客户端的其他服务，只要中转服务器上启动代理服务，客户端就可以正常访问，缺点是只针对 MySQL 适用。大家可以根据自己的实际应用环境进行选择。

31.9 小结

本章详细介绍了在 MySQL 中经常会遇到的一些问题及其解决办法。对于实际应用来说，用户遇到的问题可能远远不止这些，希望大家在实践中能够多多总结，为管理和应用 MySQL 积累更多的经验。

欢迎点击这里的链接进入精彩的Linux公社网站

Linux公社（www.Linuxidc.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

Linux公社是专业的Linux系统门户网站，实时发布最新Linux资讯，包括Linux、Ubuntu、Fedora、RedHat、红旗Linux、Linux教程、Linux认证、SUSE Linux、Android、Oracle、Hadoop、CentOS、MySQL、Apache、Nginx、Tomcat、Python、Java、C语言、OpenStack、集群等技术。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

Linux公社 主站网址：www.linuxidc.com 旗下网站：www.linuxidc.net

包括：**Ubuntu 专题** **Fedora 专题** **Android 专题** **Oracle 专题** **Hadoop 专题** **RedHat 专题** **SUSE 专题** **红旗 Linux 专题** **CentOS 专题**

Linux 公社微信公众号：linuxidc_com