

# 《MySql 基础教程》

## 目 录

第 一 章.....2

第 二 章.....13

第 三 章.....24

第 四 章.....34

第 五 章.....47

第 六 章.....55

第 七 章.....65

第 八 章.....73

第 九 章.....76

第 十 章.....82

第 十 一 章.....88

第 十 二 章.....96

第 十 三 章.....101

第 十 四 章.....119

第 十 五 章.....122

# 第一章

SQL 全称是“结构化查询语言(Structured Query Language)”，最早的是 IBM 的圣约瑟研究实验室为其关系数据库管理系统 SYSTEM R 开发的一种查询语言，它的前身是 SQUARE 语言。SQL 语言结构简洁，功能强大，简单易学，所以自从 IBM 公司 1981 年推出以来，SQL 语言，得到了广泛的应用。如今无论是像 Oracle ,Sybase,Informix,SQL server 这些大型的数据库管理系统，还是像 Visual Foxpro,PowerBuilder 这些微机上常用的数据库开发系统，都支持 SQL 语言作为查询语言。

SQL 是高级的非过程化编程语言，允许用户在高层数据结构上工作。他不要求用户指定对数据的存放方法，也不需要用户了解具体的数据存放方式，所以具有完全不同底层结构的不同数据库系统可以使用相同的 SQL 语言作为数据输入与管理的接口。它以记录集合作为操纵对象，所有 SQL 语句接受集合作为输入，返回集合作为输出，这种集合特性允许一条 SQL 语句的输出作为另一条 SQL 语句的输入，所以 SQL 语言可以嵌套，这使他具有极大的灵活性和强大的功能，在多数情况下，在其他语言中需要一大段程序实现的一个单独事件只需要一个 SQL 语句就可以达到目的，这也意味着用 SQL 语言可以写出非常复杂的语句。

SQL 同时也是数据库文件格式的扩展名。

SQL 语言包含 4 个部分：

数据查询语言（SELECT 语句）

数据操纵语言（INSERT, UPDATE, DELETE 语句）

数据定义语言（如 CREATE, DROP 等语句）

数据控制语言（如 COMMIT, ROLLBACK 等语句）

SQL 语言是结构化语言（Structure Query Language）的缩写，是一种用于数据库查询和编程的语言，已经成为关系型数据库普遍使用的标准，使用这种标准数据库语言对程序设计和数据库的维护都带来了极大的方便，广泛地应用于各种数据查询。VB 和其他的应用程序包括 Access、Foxpro、Oracle、SQL Server 等都支持 SQL 语言。

SQL 语言的常用操作有：建立数据库数据表（CREATE TABLE），如本系统中的学生及成绩备份就用到该语句；从数据库中筛选一个记录集（SELECT），这是最常用的一个语句，功能强大，能有效地对数据库中一个或多个数据表中的数据进行访问，并兼有排序、分组等功能；在数据表中添加一个记录（INSERT）；删除符合条件的记录（DELETE）；更改符合条件的记录（UPDATE）；

VB 中的数据库操作对象都提供了对 SQL 语句的支持。其一般的用法是以 VB 的各种控件接收用户对数据库访问的请求，在事件响应程序代码中将其转换成对数据库的 SQL 查询语句，并以字符串的形式存在，然后将其传递给相应的数据库操作对象，最终完成对数据库的访问数据库，顾名思义，是存入数据的仓库。只不过这个仓库是在计算机存储设备上的，而且数据是按一定格式存放的。

当人们收集了大量的数据后,应该把它们保存起来进入近一步的处理,进一步的抽取有用的信息。当年人们把数据存放在文件柜中,可现在随着社会的发展,数据量急剧增长,现在人们就借助计算机和数据库技术科学的保存大量的数据,以便能更好的利用这些数据资源。

要是下定义的话,就应该是:指长期储存在计算机内的、有组织的、可共享的数据集合。

数据库包含关系数据库、面向对象数据库及新兴的 XML 数据库等多种,目前应用最广泛的是关系数据库,若在关系数据库基础上提供部分面向对象数据库功能的对象关系数据库。在数据库技术的早期还曾经流行过层次数据库与网状数据库,但这两类数据库目前已经极少使用。

## 数据库管理

数据库管理(Database Administration)是有关建立、存储、修改和存取数据库中信息的技术,是指为保证数据库系统的正常运行和服务质量,有关人员须进行的技术管理工作。负责这些技术管理工作的个人或集体称为数据库管理员(DBA)。数据库管理的主要内容有:数据库的建立、数据库的调整、数据库的重组、数据库的重构、数据库的安全控制、数据的完整性控制和对用户提供技术支持。

数据库的建立:数据库的设计只是提供了数据的类型、逻辑结构、联系、约束和存储结构等有关数据的描述。这些描述称为数据模式。要建立可运行的数据库,还需进行下列工作:

(1)选定数据库的各种参数,例如最大的数据存储空间、缓冲池的数量、并发度等。这些参数可以由用户设置,也可以由系统按默认值设置。

(2)定义数据库,利用数据库管理系统(DBMS)所提供的数据库定义语言和命令,定义数据库名、数据库模式、索引等。

(3)准备和装入数据,定义数据库仅仅建立了数据库的框架,要建成数据库还必须装入大量的数据,这是一项浩繁的工作。在数据的准备和录入过程中,必须在技术和制度上采取措施,保证装入数据的正确性。计算机系统中原已积累的数据,要充分利用,尽可能转换成数据库的数据。

注:"数据库"这个词对于不同的人应该给予不同的感觉。如果你是一个最终用户,你根本就不关心数据存储和维护的细节,数据库也不应该拿这些事情来烦你。但是如果你是一个数据库管理员,那么有些细节上的东西你就必须要清楚。数据库管理系统可以为不同的用户提供不同的视图,也就是他们所看到的数据库是不一样的。这就需要进行数据抽象,以形成这些不同的视图。

最早是在 CODASYL 的 DBTG 报告中完整地给出了数据抽象的三个层次。ANSI/SPARC 报告中也提出了类似的建议,这个报告中抽象的层次为内部层、概念层和外部层。但是,现在的数据库管理系统是根据 DBTG 的报告从三个层次来进行抽象的,它们分别是物理层、逻辑层和视图层(概念层)。

## 数据库的种类

大型数据库有: Oracle、Sybase、DB2、SQL server

小型数据库有: Access、MySQL 等。

在接下来的章节内,我们将详细学习 MySQL 的相关知识。

## 一、什么是 MySQL ?

MySQL 是一个广受 Linux 社区人们喜爱的半商业的数据库。MySQL 是可运行在大多数的 Linux 平台(i386, Sparc, etc), 以及少许非 Linux 甚至非 Unix 平台。

### 1、许可费用

MySQL 的普及很大程度上源于它的宽松, 除了略显不寻常的许可费用。MySQL 的价格随平台和安装方式变化。MySQL 的 Windows 版本 (NT 和 9X) 在任何情况下都不免费, 而任何 Unix 变种 (包括 Linux) 的 MySQL 如果由用户自己或系统管理员而不是第三方安装则是免费的, 第三方案庄则必须付许可费。

### 2、价格

平台 安装方式 价格

Windows NT,9X 任何 200 美元

Unix 或 Linux 自行安装 免费

Unix 或 Linux 第三方安装 200 美元

需要一个应用组件 200 美元

可以得到多种支持合同, 内容太多不再罗列, 最新报价可咨询 MySQL 站点。

### 3、安装

可以在 MySQL 站点上获得大多数主要的软件包格式 (RPM、DBE、TGZ), 客户端库和各种语言“包装”(Wrapper) 可以分开的 RPM 格式获得。RPM 格式的安装没有多大麻烦, 并且无需初始配置。在 rc3.d (以 RedHat RPM 为例) 生成一个初始脚本, 故 MySQL 守护进程在多用户模式下重启时被启动运行。MySQL 的守护进程 (mysqld) 消耗很少的内存 (在运行 RedHat 5.1 的奔腾 133 上, 每个守护进程使用 500K 内存和另外 4M 共享内存的开销) 并在只有在执行真正的查询时才装载到处理器上, 这意味着对小型数据库来说, MySQL 可以相当轻松地使用而不会对其他系统功能有太大的影响。

### 4、数据类型

字段支持大量数据类型是件好事。通常的整数、浮点数、字符串和数字均以多种长度表示, 并支持变长的 BLOB (Binary Large Object) 类型。对整数字段由自动增量选项, 日期时间字段也能很好的表示。

MySQL 与大多数其他数据库系统不同的是提供两个相对不常用的字段类型: ENUM 和 SET。ENUM 是一个枚举类型, 非常类适于 Pascal 语言的枚举类型, 它允许程序员看到类似于 'red'、'green'、'blue' 的字段值, 而 MySQL 只将这些值存储为一个字节。SET 也是从 Pascal 借用的, 它也是一个枚举类型, 但一个单独字段一次可存储多个值, 这种存储多个枚举值的能力也许不会给你一些印象 (并可能威胁第三范式定义), 但正确使用 SET 和 CONTAINS 关键字可以省去很多表连接, 能获得很好的性能提高。

### 5、SQL 兼容性

MySQL 包含一些与 SQL 标准不同的转变, 他们的大多数被设计成是对 SQL 语言脚本语言的不足的一种补偿。然而, 另一些扩展确实使 MySQL 与众不同, 例如, LINK 子句搜索是自动地忽略大小写的。MySQL 也允许用户自定义的 SQL 函数, 换句话说, 一个程序员可以编写一个函数然后集成到 MySQL

中, 并且其表现的与任何基本函数如 **SUM()**或 **AVG ()**没有什么不同。函数必须被编译进一个共享库文件中(.so 文件), 然后用一个 **LOAD FUNCTION** 命令装载。

它也缺乏一些常用的 **SQL** 功能, 没有子选择(在查询中的查询)。视图(**View**)也没了。当然大多数子查询可以用简单的连接(**join**)子句重写, 但有时用两个嵌套的查询思考问题比一个大连接容易。同样, 视图仅仅为程序员隐蔽 **where** 子句, 但这正是程序员们期望的另一种便利。

## 6、存储过程和触发器

**MySQL** 没有一种存储过程(**Stored Procedure**)语言, 这是对习惯于企业级数据库的程序员的最大限制。多语句 **SQL** 命令必须通过客户方代码来协调, 这种情形是借助于相当健全的查询语言和赋予客户端锁定和解锁表的能力, 这样才允许的多语句运行。

## 7、参考完整性 (Referential Integrity-RI)

**MySQL** 的主要的缺陷之一是缺乏标准的 **RI** 机制; 然而, **MySQL** 的创造者也不是对其用户的愿望置若罔闻, 并且提供了一些解决办法。其中之一是支持唯一索引。**Rule** 限制的缺乏(在给订字段域上的一种固定的范围限制)通过大量的数据类型来补偿。不简单地提供检查约束(一个字段相对于同一行的另一个字段的之值的限制)、外部关键字和经常与 **RI** 相关的“级联删除”功能。有趣的是, 当不支持这些功能时, **SQL** 分析器容忍这些语句的句法。这样做目的是易于移植数据库到 **MySQL** 中。这是一个很好的尝试, 并且它确实未来支持该功能留下方便之门; 然而, 那些没有仔细阅读文档的人可能误以为这些功能实际上是存在的。

## 8、安全性

自始至终我对 **MySQL** 最大的抱怨是其安全系统, 它唯一的缺点是复杂而非标准, 另外只有到调用 **mysqladmin** 来重读用户权限时才发生改变。通常的 **SQL GRANT/REVOKE** 语句到最近的版本才被支持, 但是至少他们现在有了。 **MySQL** 的编写者广泛地记载了其特定的安全性系统, 但是它确实需要一条可能是别无它法的学习过程。

## 9、备份和恢复、数据导入/导出

强制参考一致性的缺乏显著地简化备份和恢复, 单靠数据导入/导出就可完美复制这一功能。**LOAD DATA INFILE** 命令给了数据导入很大的灵活性。**SELECT INTO** 命令实现了数据导出的相等功能。另外, 既然 **MySQL** 不使用原始的分区, 所有的数据库数据能用一个文件系统备份保存。数据库活动能被记载。与通常的数据库日志不同(存储记录变化或在记录映像之前/之后), **MySQL** 记载实际的 **SQL** 语句。这允许数据库被恢复到失败前的那一点, 但是不允许提交(**commit**)和回卷(**rollback**)操作。

## 9、连接性

**MySQL** 客户库是客户/服务器结构的 **C** 语言库, 它意味着一个客户能查询驻留在另一台机器的一个数据库。然而 **MySQL** 真正的强项处于该库中的语言“包装器(wrapper)”, **Perl**、**Python** 和 **PHP** 只是一部分。**Apache** 的 **Web** 服务器也有许多模块例如目录存取文件等允许各种各样的 **Apache** 配置信息(例如目录存取文件)使用 **MySQL**, 应用程序接口简单、一致并且相对完整。另外、多平台 **ODBC** 驱动程序可自由获得。

## 10、未来

MySQL 的开发继续以快速进行着。事实上, 开发步伐对大多数开放源代码是一种挑战。本文提到的几个抱怨中有很多新功能正在解决, 然而, 我将不对还没确实存在的特征做评价。开发者们向我表明了在未来的开发中把增加查询功能和提高查询速度作为最高优先级。

## 11、总结

MySQL 是数据库领域的中间派。它缺乏一个全功能数据库的大多数主要特征, 但是又有比类似 Xbase 记录存储引擎更多的特征。它象企业级 RDBMS 那样需要一个积极的服务者守护程序, 但是不能象他们那样消费资源。查询语言允许复杂的连接(join)查询, 但是所有的参考完整必须由程序员强制保证。

MySQL 在 Linux 世界里找到一个位置—提供简洁和速度, 同时仍然提供足够的功能使程序员高兴。数据库程序员将喜欢其查询功能和广泛的客户库, 数据库管理员会觉得系统缺乏主要数据库功能, 他们会发觉它对简单数据库(在不能保证购买大牌数据库时)是有价值的。

# 二、安装 MySQL

### 步骤 1 下载 MySQL

下载地址: <http://dev.mysql.com/downloads/mysql/5.0.html#downloads>

### 步骤 2 安装 MySQL

双击 MySQLSetup.exe, 单击 Next 继续, 选择安装类型, 如图 1.18 所示。有” Typical (默认)”、” Complete (完全)”、” Custom (用户自定义)” 三个选项, 在这里请选择” Custom”, 这样可以在后面的安装过程中设置相关的选项, 单击 Next 继续安装。

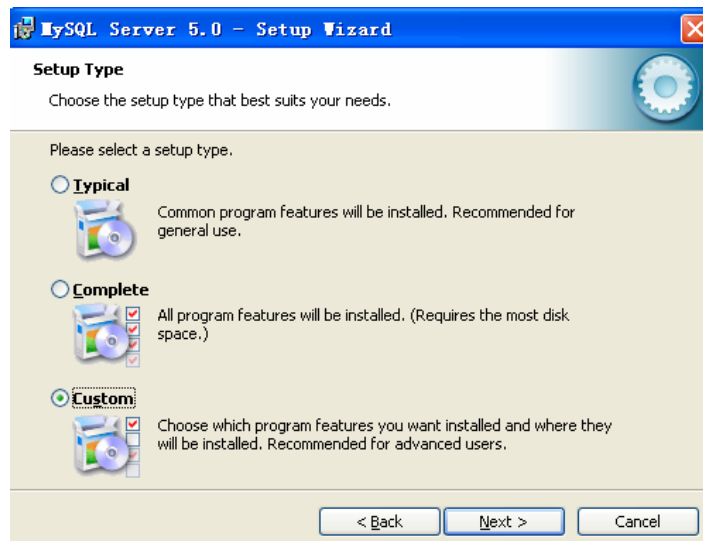


图 1.18MySQL 的安装类型选项图

上一步选择了 Custom 安装, 这里将设定 MySQL 的组件包和安装路径, 如图 1.18 所示。设定好之后单击 Next 继续安装。



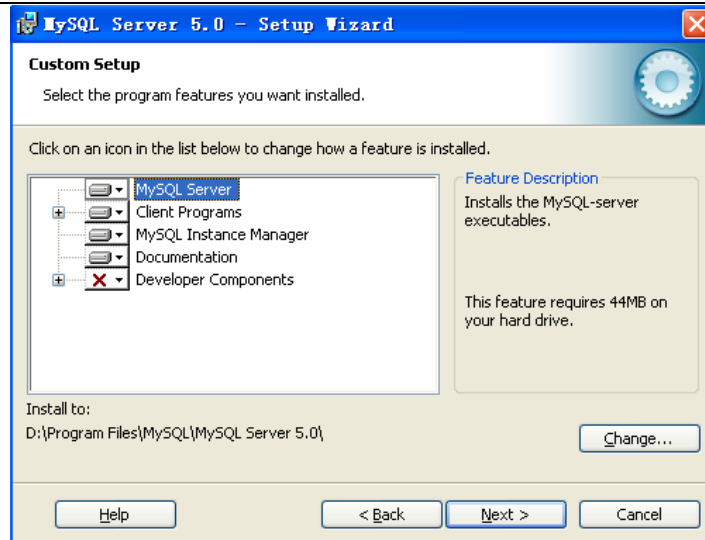


图 1.19 自定义 MySQL 安装路径

信息配置完成, 单击 **Install** 进行安装, 然后一路单击 **Next**, 直到出现如图 1.19 所示界面。单击 **Finish**, 将结束 Mysql 的安装。如果在单击 **Finish** 时, 选中 “configure the MySQL Server now” 项, 将启动 mysql 配置向导, 如图 1.20 所示。

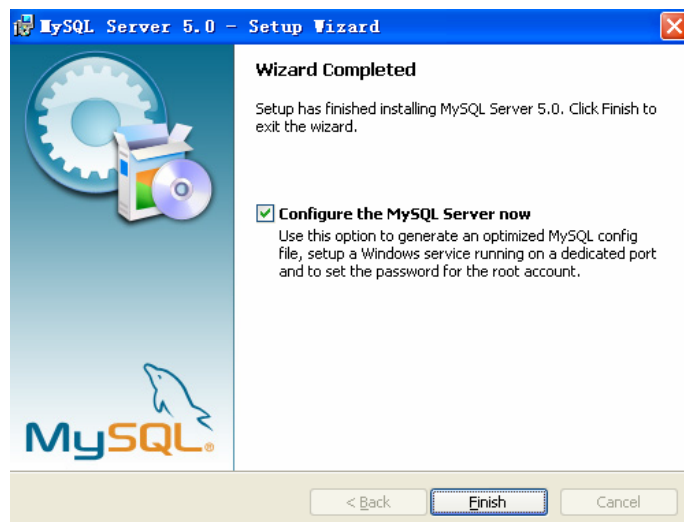


图 1.20 提示是否配置服务

### 步骤 3 配置 MySQL 服务器

mysql 配置向导启动界面, 单击 **Next**, 选择配置方式, “Detailed Configuration(手动精确配置)”、“Standard Configuration (标准配置)”, 单击 “Detailed Configuration” 选项, 这个选项可以让使用者熟悉配置过程, 如图 1.21 所示

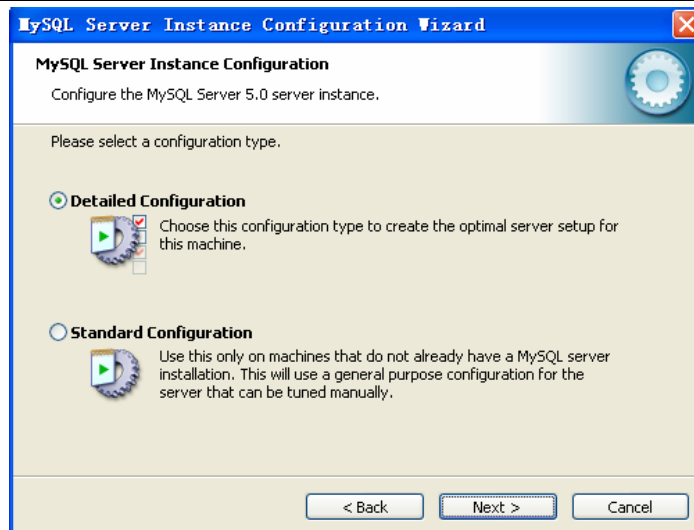


图 1.21 安装方式设置

选择服务器类型，“Developer Machine（开发测试类，mysql 占用很少资源）”、“Server Machine（服务器类型，mysql 占用较多资源）”、“Dedicated MySQL Server Machine（专门的数据库服务器，mysql 占用所有可用资源）”，一般选“Server Machine”，如图 1.22 所示。

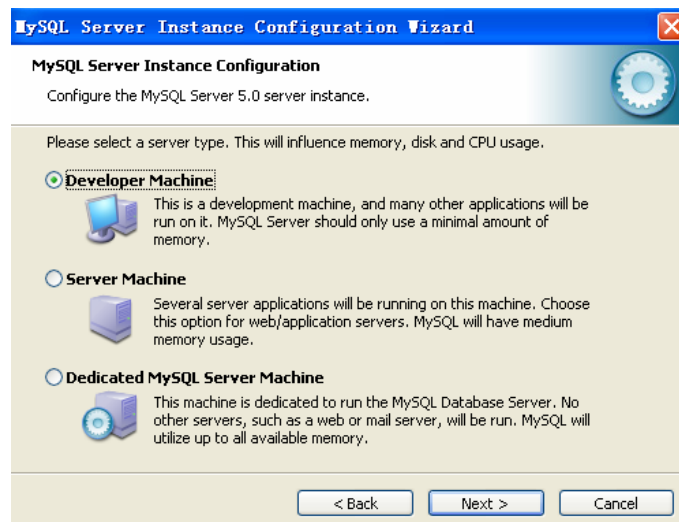


图 1.22 安装类型设置

#### 步骤 4 安装类型设置

“Multifunctional Database（通用多功能型，好）”、“Transactional Database Only（服务器类型，专注于事务处理，一般）”、“Non-Transactional Database Only”（非事务处理型，较简单，主要做一些监控、记数用，对 MyISAM 数据类型的支持仅限于 non-transactional）。这里选择“Transactional Database Only”，单击 Next 继续安装。

#### 步骤 5 设置网站允许链接 mysql 的最大数目

“Decision Support(DSS)/OLAP(20 个左右)”、“Online Transaction Processing(OLTP)(500 个左右)”、“Manual Setting（手动设置，输一个数）”，这里选“Online Transaction Processing(OLTP)”，单击 Next 如图 1.23 所示。



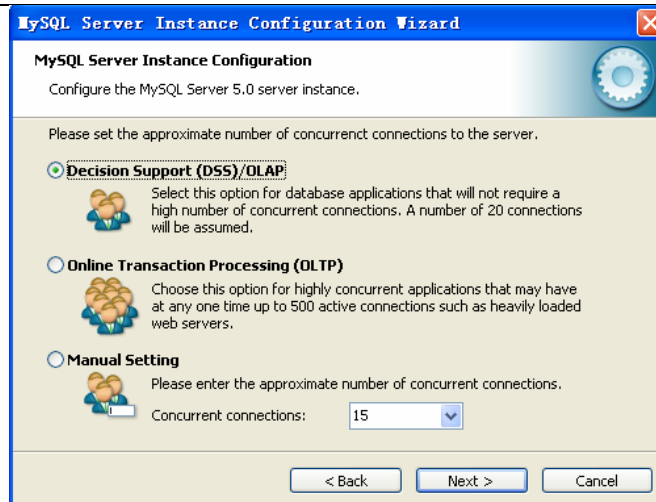


图 1.23 连接方式设置

### 步骤 6 MySQL 端口的设置

是否启用 TCP/IP 连接, 设定端口, 如果不启用, 就只能在本地的机器上访问 mysql 数据库。这里选择启用, 选中“Enable TCP/IP Networking”选项。设置 Port Number 的值为 3306, 单击 Next 如图 1.24 所示。

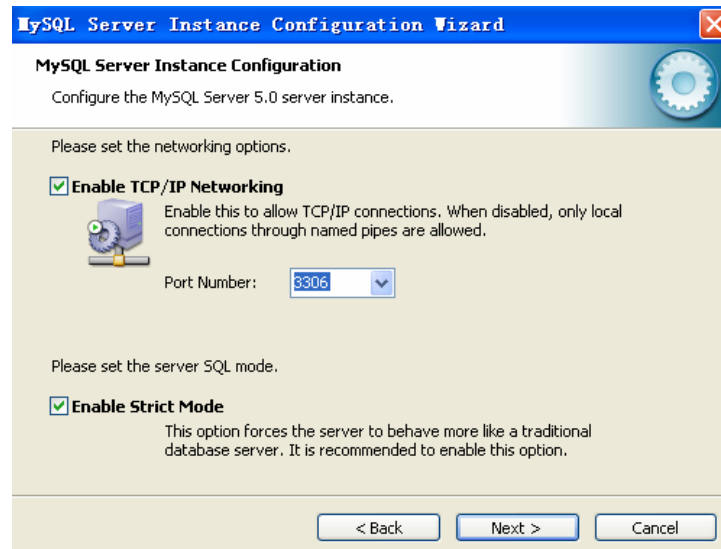


图 1.24 对 MySQL 的端口进行设置, 保持默认 3306

### 步骤 7 设置 MySQL 的字符集

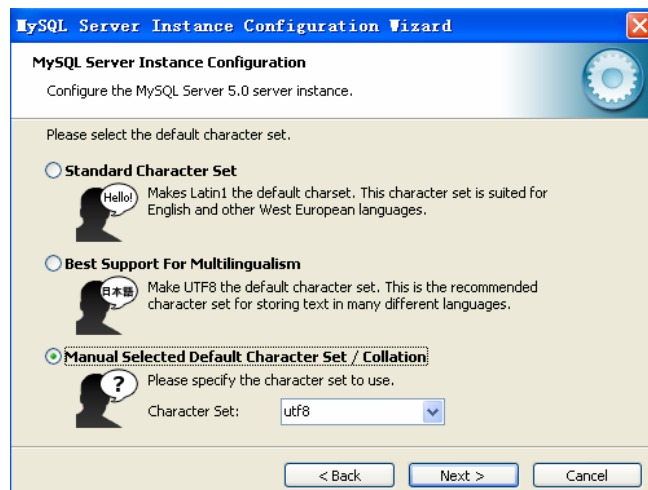


图 1.25 字符集设置

此步骤比较重要, 将对 mysql 默认数据库语言编码进行设置, 第一个是西文编码, 第二个是多字节的 utf8 编码, 这两项都不是通用的编码, 所以建议选择第三项。然后在 Character Set 那里选择或填写“gbk”或“gb2312”, 这两者的区别就是 GBK 的字库容量大, 包括了 gb2312 的所有汉字, 并且加上了繁体字等。单击 Next 按钮继续, 如图 1.25 所示。

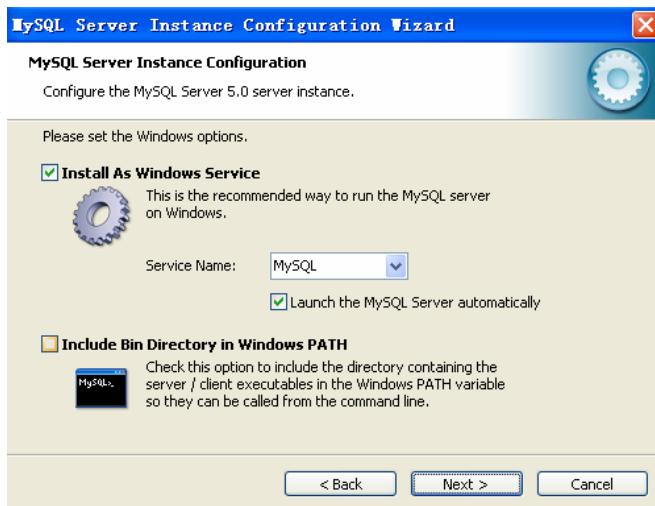


图 1.26 数据库注册

### 步骤 8 数据库注册

本步骤可以指定 Service Name (服务标识名称), 将 mysql 的 bin 目录加入到 Windows PATH (加入后, 就可以直接使用 bin 下的文件, 而不用指出目录名, 比如连接数据库。输入“mysql.exe -uusername -ppassword;”就可以, 不用指出 mysql.exe 的完整地址, 很方便), 在这里建议选中“Install As Windows Service”选项。Service Name 按默认提供的即可。如图 1.26 所示。单击 Next 继续安装。



图 1.27 权限设置

### 步骤 9 权限设置

询问是否要修改默认 root 用户 (超级管理) 的密码 (默认为空), “New root password”项可以填写新密码 (如果是重装, 并且之前已经设置了密码, 在这里更改密码可能会出错, 请留空, 安装配置完成后另行修改密码), “Confirm (再输一遍)”选项内提示再重输一次密码, 防止输错。如图 1.27 所示。“Enable root access from remote machines”选项的表示是否允许 root 用户在其它的机器上登陆, 如果要只允许本地用户访问, 就不能选中, 如果允许远程用户访问请选中此项。“Create An Anonymous Account”表示是否新建一个匿名用户, 匿名用户可以连接数据库, 不能操作数据或查询数据。一般无须选中此项。设置

完毕, 单击【Next】按钮, 将显示出如图 1.28 所示的界面。

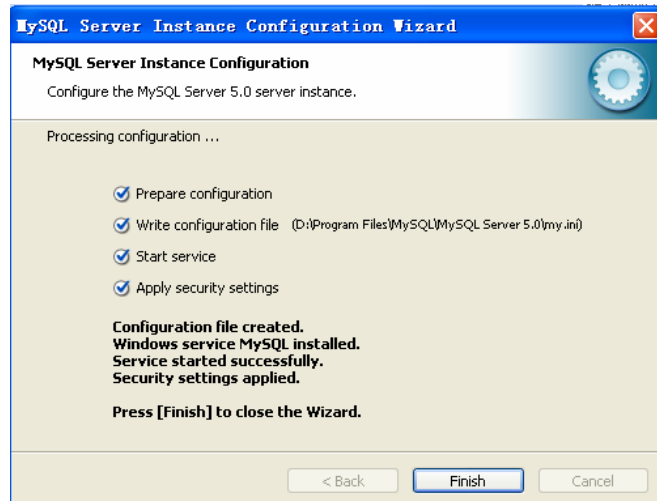
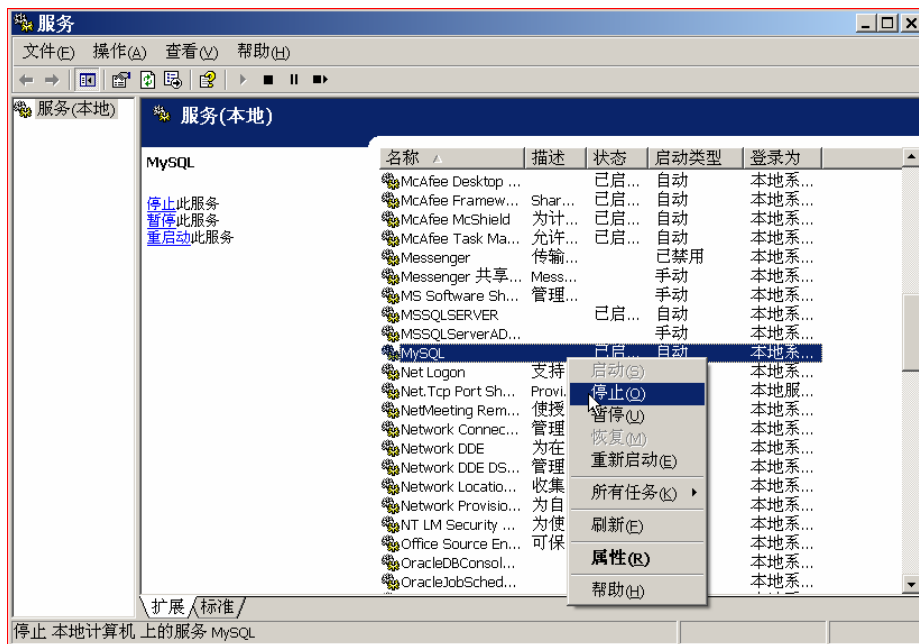


图 1.28 安装成功

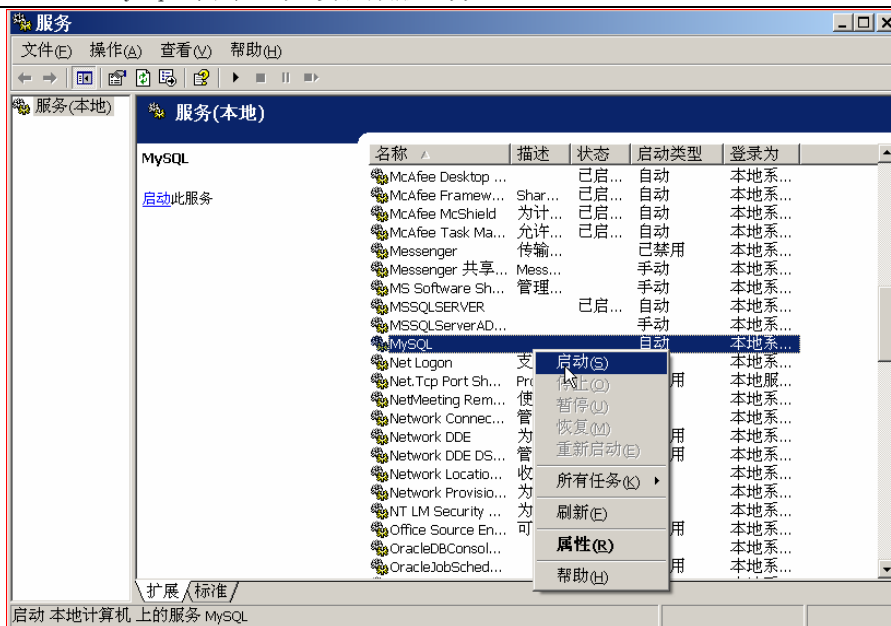
### 三、控制服务

可以通过服务管理器管理 MYSQL 的服务。

停止 MYSQL 的服务。



启动 MYSQL 的服务。



也可以在 DOS 中直接通过命令行的形式进行控制。  
启动 MYSQL 的服务。

```
D:\>net start mysql

MySQL 服务已经启动成功。

D:\>
```

停止 MYSQL 的服务。

```
C:\>NET STOP MYSQL

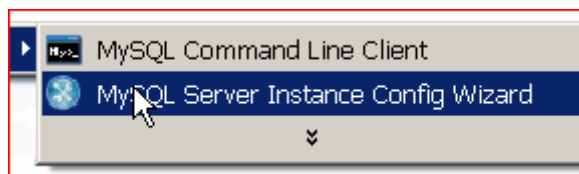
MySQL 服务正在停止。
MySQL 服务已成功停止。
```

如果在命令中出现“服务名无效”的提示，如下图所示

```
C:\>net start MYSQL;
服务名无效。

请键入 NET HELPMSG 2185 以获得更多的帮助。
```

请重新执行 MYSQL 服务配置向导。

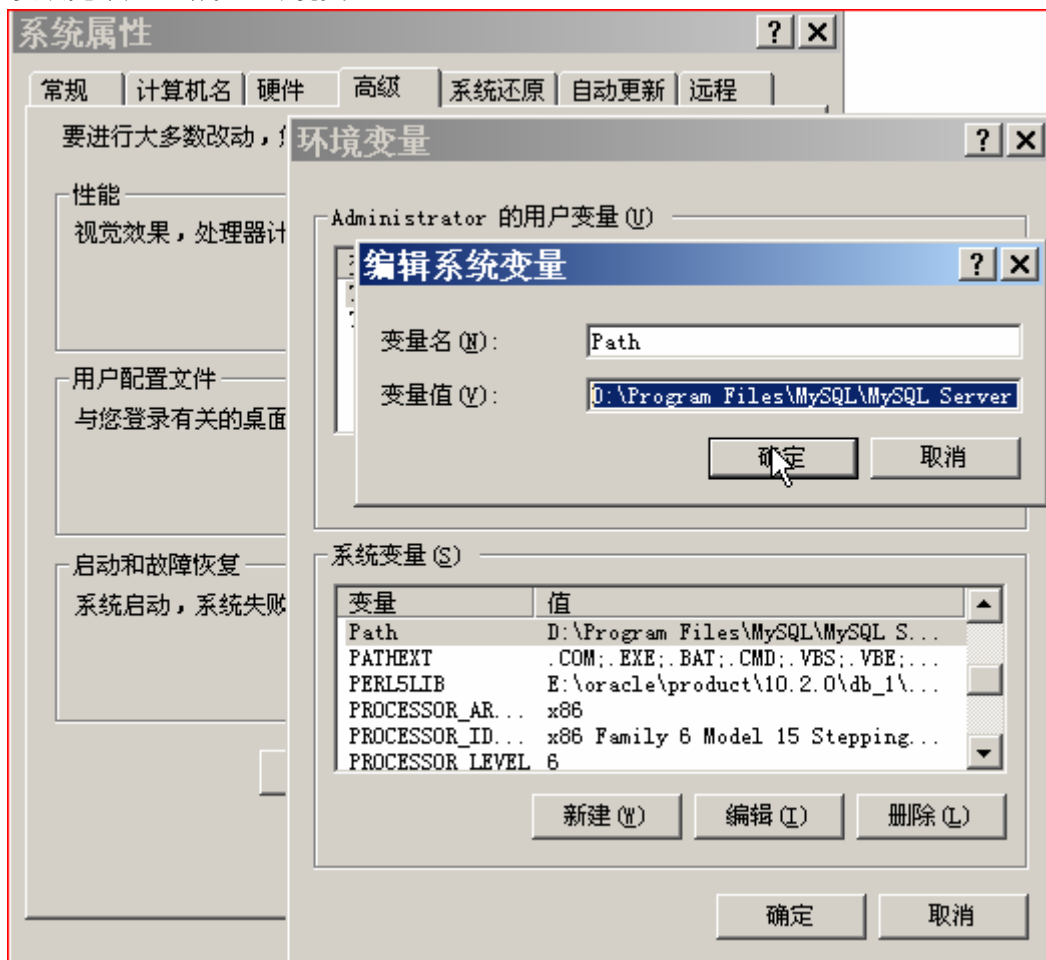


## 第二章

DDL 语句:

DDL 是 Data definition Language 的缩写, 意为数据定义语言, 是 SQL 语言的四大功能之一。用于定义数据库的三级结构, 包括外模式、概念模式、内模式及其相互之间的映像, 定义数据的完整性、安全控制等约束。

MySQL 安装完毕后, 请设置环境变量:



### 1、连接 MYSQL。

格式: `mysql -h 主机地址 -u 用户名 -p 用户密码`

#### a、例 1: 连接到本机上的 MYSQL。

首先在打开 DOS 窗口, 然后进入目录 `mysqlbin`(如果设置了环境变量就不用再进入相关目录), 再键入命令 `mysql -uroot -p`, 回车后提示你输密码, 如果刚安装好 MYSQL, 超级用户 root 是没有密码的, 故直接回车即可进入到 MYSQL 中了, MYSQL 的提示符是: `mysql>`;

```
C:\Documents and Settings\Administrator>mysql -uroot -p3239436
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 5.0.24a-community-nt

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

mysql> _
```

Welcome to the MySQL monitor. Commands end with ; or \g. 说明命令的结束符, 用;或\g 结束。

Your MySQL connection id is 35 to server version: 5.0.24a-community-nt. 说明客户端的连接 ID, 这个数字记录了 MySQL 服务到目前为止的连接次数, 每次连接都会自动加 1, 本例中是 35 次。后面的语句是说明 MySQL 的版本号。

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.通过“help”或“\h”命令来显示帮助内容, 通过“\c”命令来清除命令行缓存.\c 不是清屏的, 它是不执行\c 前面的命令。比如你执行 select \* from user \c select \* from message;得到的结果与 select \* from message 的结果是一致的。

注意这里的语句后面不能加“;”号, 否则会引起错误。因为它把分号误以为是密码的一部分。

```
C:\>mysql -uroot -p3239436;
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)
```

也可以把用户名与密码分开的方式创建与数据库的连接。

```
C:\>mysql -uroot -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 5.0.24a-community-nt

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

还可以用下列的一种方式联接数据库。

```
C:\>mysql -uroot --password=3239436
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8 to server version: 5.0.24a-community-nt

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

通过上述练习, 请注意, 在命令行中指定的程序选项遵从下述规则:

- 在命令名后面紧跟选项。
- 选项参量以一个和两个破折号开始, 取决于它具有短名还是长名。许多选项有两种形式。例如, -?和--help 是指导 MySQL 程序显示帮助消息的选项的短名和长名。
- 选项名对大小写敏感。-v 和-V 均有效, 但具有不同的含义。(它们是--verbose 和--version 选项的短名)。
- 部分选项在选项名后面紧随选项值。例如, -h localhost 或--host=localhost 表示客户程序的 MySQL 服务器主机。选项值可以告诉程序 MySQL 服务器运行的主机名。
- 对于带选项值的长选项, 通过一个 '=' 将选项名和值隔离开来。对于带选项值的短选项, 选项值可以紧随选项字母后面, 或者二者之间可以用一个空格隔开。(-hlocalhost 和-h localhost 是等效的)。该规则的例外情况是指定 MySQL 密码的选项。该选项的形式可以为--password=pass\_val 或--password。在后一种情况(未给出 密码值), 程序将提示输入密码。也可以给出密码选项, 短形式为-ppass\_val 或-p。然而, 对于短形式, 如果给出了密码值, 必须紧跟在选项后面, 中间不能插入空格。这样要求的原因是如果选项后面有空格, 程序没有办法来告知后面的参量是 密码值还是其它某种参量。



b、例 2: 连接到远程主机上的 MYSQL。假设远程主机的 IP 为: 110.110.110.110, 用户名为 root, 密码为 abcd123。则键入以下命令:

mysql -h110.110.110.110 -uroot -pabcd123 (注:u 与 root 可以不用加空格, 其它也一样)

```
C:\>mysql -h127.0.0.1 -uroot -p3239436
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 5.0.24a-community-nt

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

mysql>
```

c、退出 MYSQL 命令: exit (回车) (注意: 在 MYSQL 环境中的命令, 所有操作后面都带一个分号作为命令结束符)

```
mysql> exit;
Bye
C:\>
```

## 2、增加新用户

数据库在创建之时, 权限为 Root, 然而每个普通用户并不需要这么高的权限, 基于安全考虑也不能分配这样的权限给使用者, 所以增加新用户的命令就显得重要了。

格式: grant select on 数据库.\* to 用户名@登录主机 identified by "密码"

```
mysql> grant select,insert,update,delete on *.* to xmh@"%" identified by "xmh";
Query OK, 0 rows affected (0.13 sec)

mysql> _
```

例 1、增加一个用户 xmh 密码为 xmh, 让他可以在任何主机上登录, 并对所有数据库有查询、插入、修改、删除的权限。首先用以 root 用户连入 MYSQL, 然后键入以下命令:

grant select,insert,update,delete on \*.\* to xmh@"%" identified by "xmh";

但这样增加的用户是十分危险的, 你想如某个人知道 xmh 的密码, 那么他就可以在互联网上的任何一台电脑上登录你的 mysql 数据库并对你的数据可以为所欲为了, 如何解决呢?

例 2、增加一个用户 zah 密码为 zah, 让他只可以在 localhost 上登录, 并可以对数据库 test 进行查询、插入、修改、删除的操作 (localhost 指本地主机, 即 MYSQL 数据库所在的那台主机), 这样用户即使用知道 zah 的密码, 他也无法从 internet 上直接访问数据库, 只能通过 MYSQL 主机上的 web 页来访问了。

```
mysql> grant select,insert,update,delete on test.* to zah@localhost identified by "zah";
Query OK, 0 rows affected (0.03 sec)
```

grant select,insert,update,delete on test.\* to zah@localhost identified by "zah";

如果你不想 zah 有密码, 可以再打一个命令将密码消掉。

grant select,insert,update,delete on test.\* to zah@localhost identified by "";

MySQL 可以为不同的用户分配严格的、复杂的权限。这些操作大多都可以用 SQL 指令 Grant (分配权限) 和 Revoke (回收权限) 来实现。Grant 可以把指定的权限分配给特定的用户, 如果这个用户不存在, 则会创建一个用户。

Grant 常用格式:

`grant` 权限 1,权限 2,...权限 n on 数据库名称.表名称 to 用户名@用户地址 identified by ‘连接口令’;

权限 1,权限 2,...权限 n 代表 `select,insert,update,delete,create,drop,index,alter,grant,references,reload,shutdown,process,file` 等 14 个权限。

当权限 1,权限 2,...权限 n 被 `all privileges` 或者 `all` 代替,表示赋予用户全部权限。

当数据库名称.表名称被`*,*`代替,表示赋予用户操作服务器上所有数据库所有表的权限。

用户地址可以是 `localhost`,也可以是 ip 地址、机器名字、域名。也可以用 `'%'`表示从任何地址连接。

‘连接口令’不能为空,否则创建失败。

比较重要的是 `priveleges` (权限)。

普通用户的权限权限应用于描述

`SELECT` 表,列允许用户从表中选择行(记录)

`INSERT` 表,列允许用户在表中插入新行

`UPDATE` 表,列允许用户修改现存表里行中的值

`DELETE` 表允许用户删除现存表的行

`INDEX` 表允许用户创建和拖动特定表索引

`ALTER` 表允许用户改变现存表的结构。例如,可添加列、重名列或表、修改列的数据类型

`CREATE` 数据库,表允许用户创建新数据库或表。如果在 `GRANT` 中指定了一个特定的数据库或表,他们只能够创建该数据库或表,即他们必须首先删除(`Drop`)它

`DROP` 数据库,表允许用户拖动(删除)数据库或表

管理员权限权限描述

`CREATE TEMPORARY TABLES` 允许管理员在 `CREATE TABLE` 语句中使用 `TEMPORARY` 关键字

`FILE` 允许将数据从文件读入表,或从表读入文件

`LOCK TABLES` 允许使用 `LOCK TABLES` 语句

`PROCESS` 允许管理员查看属于所有用户的服务器进程

`RELOAD` 允许管理员重新载入授权表、清空授权、主机、日志和表格

`REPLICATION CLIENT` 允许在复制主机(Master)和从机(Slave)上使用 `SHOW STATUS`

`REPLICATION SLAVE` 允许复制从服务器连接到主服务器

`SHOW DATABASES` 允许使用 `SHOW DATABASES` 语句查看所有的数据库列表。没有这个权限,用户只能看到他们能够看到的数据库

`SHUTDOWN` 允许管理员关闭 MySQL 服务器

`SUPER` 允许管理员关闭属于任何用户的线程

特别的权限权限描述

`ALL` (或 `ALL PRIVILEGES`) 授予所有权限

`USAGE` 不授予权限。这将创建一个用户并允许他登录,但不允许其他操作,如 `update/select` 等

实例:

例如:

```
mysql>grant select,insert,update,delete on test.user to mql@localhost identified by '123456' ;
```

给本地的用户 `mql` 分配可对数据库 `test` 的 `user` 表进行 `select,insert,update,delete` 操作的权限,并设定口令为 `123456`。若 `mql` 用户不存在,则将自动创建此用户。具体的权限控制在 `mysql.db` 表中可以查看到。也可直接对这个表进行更新操作进行权限的修改。

```
mysql>grant all privileges on test.* to mql@localhost identified by '123456' ;
```

给本地用户 `mql` 分配可对数据库 `test` 所有表进行所有操作的权限,并设定口令为 `123456`。

```
mysql>grant all privileges on *.* to mql@localhost identified by '123456' ;
```

给本地用户 `mql` 分配可对所有数据库的所有表进行所有操作的权限, 并设定口令为 123456。

```
mysql>grant all privileges on *.* to mql2@61.127.46.128 identified by '123456' ;
```

给来自 61.127.46.128 的用户 `mql2` 分配可对所有数据库的所有表进行所有操作的权限, 并设定口令为 123456

3、修改密码(注意这是在 DOS 环境下进行的)

格式: `mysqladmin -u 用户名 -p 旧密码 password 新密码`

1、例 1: 给 `zah` 加个密码 `zyj`。首先在 DOS 下进入目录 `mysqlbin`, 然后键入以下命令

```
C:\>mysqladmin -uzah password "zyj"
C:\>_
```

适应于给空密码的用户添加密码

```
mysqladmin -uzah password zyj
```

注: 因为开始时 `zah` 没有密码, 所以 `-p` 旧密码一项就可以省略了, 再者新密码加不加引号都一样。

2、例 2: 再将 `xmh` 的密码改为 `zyj`。

```
C:\>mysqladmin -uxmh -pxmh password zyj
C:\>_
```

```
mysqladmin -uxmh -pxmh password zyj
```

3、更改用户权限

查看用户权限 (查看当前登录用户的权限)

```
mysql> show grants for "zah"@'localhost';
+-----+
| Grants for zah@localhost |
+-----+
| GRANT SELECT, INSERT, UPDATE, DELETE ON *.* TO 'zah'@'localhost' IDENTIFIED BY |
| PASSWORD '*700FF6A5221599687B18B43D80553FDAEC143498' |
+-----+
1 row in set (0.00 sec)

mysql> show grants;
+-----+
| Grants for root@localhost |
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED BY PASSWORD '*732 |
| 0726F8389D92B765B5914B5FEDF919A82F433' WITH GRANT OPTION |
+-----+
1 row in set (0.00 sec)
```

使用 `show grants` 语句可以查看当前登陆用户的权限, 也可以指定相应的用户进行查询。

解除用户权限

REVOKE 和作用和 GRANT 相反, 语法格式为:

REVOKE privileges ON 数据库名[.表名] FROM user\_name

```
mysql> revoke all on *.* from zah@localhost ;
Query OK, 0 rows affected (0.00 sec)

mysql> show grants for "zah"@localhost";
+-----+
| Grants for zah@localhost |
+-----+
| GRANT USAGE ON *.* TO 'zah'@'localhost' IDENTIFIED BY PASSWORD '*700FF6A5221599687B18B43D80553FDAEC143498' |
+-----+
1 row in set (0.00 sec)
```

可以比较执行操作前后显示的 zah 权限, 可以发现 select、insert、update、delete 等权限已被解除。

REVOKE all...仅仅是回收用户的权限, 并不删除用户。在 MySQL 中, 用户信息存放在 mysql.User 中。MySQL 可以通过 DROP USER 来彻底删除一个用户, 具体操作请见下文。

#### 4、查看用户列表

```
mysql> use mysql;
Database changed
mysql> select user from user;
+-----+
| user |
+-----+
| root |
| scott |
| root |
| scott |
| zah |
+-----+
5 rows in set (0.00 sec)
```

#### 5、删除用户

```
mysql> drop user zah@localhost;
Query OK, 0 rows affected (0.09 sec)
```

使用 drop user 语句把用户从 user 表中删除。

#### 5、创建数据库

在此将利用 zah 的帐户进行相关操作, 所以先给 zah 帐户赋予操作权限。

```
mysql> grant select,insert,update,delete,create on *.* to zah@localhost identified by "zah";
Query OK, 0 rows affected (0.02 sec)

mysql> show grants for "zah"@localhost";
+-----+
| Grants for zah@localhost |
+-----+
| GRANT SELECT, INSERT, UPDATE, DELETE, CREATE ON *.* TO 'zah'@'localhost' IDENTIFIED BY PASSWORD '*700FF6A5221599687B18B43D80553FDAEC143498' |
+-----+
1 row in set (0.00 sec)
```

由于前面的并未给用户设置 create 权限，所以如上图所示，添加 create 权限。  
create database dbname;

```
mysql> create database mysqldemo;
Query OK, 1 row affected (0.03 sec)

mysql>
```

“Query OK, 1 row affected (0.03 sec)”表示上面的命令执行成功。可能有读者会表示理解，明明是 create 命令，怎么会是 query 呢？其实在 MySQL 中，DDL 和 DML（不包括 SELECT）操作执行成功后都显示“Query OK”。“1 row affected”表示操作只影响了数据库一行的记录，“0.03 sec”记录了操作执行的时间。

如果数据库已存在，则会如下提示：

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

为数据库改名

```
mysql> rename database beijing to newbeijing;
Query OK, 0 rows affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| auto |
| mysql |
| newbeijing |
| pharsedatabase |
| test |
+-----+
6 rows in set (0.00 sec)
```

## 6、查看数据库

```
mysql> show databases ;
+-----+
| Database |
+-----+
| information_schema |
| bbcs |
| bbcs8 |
| bj2008 |
| blog |
| blogonline |
| catalog |
| cdshop |
| digitstore |
| dlog4j |
| friends |
| gbooks |
| jpetstore |
| myblog |
| mynews |
| mysql |
| mysqldemo |
| new2007 |
| newsboard |
| registerbook |
| roller |
| stunews |
| stuscore |
| tdbcrm |
| whilvydy |
+-----+
25 rows in set (0.00 sec)
```

上面列出的表中有四张表是系统自动创建的。

INFORMATION\_SCHEMA 是信息数据库，其中保存着关于 MySQL 服务器所维护的所有其他数据库的信息。在 INFORMATION\_SCHEMA 中，有数个只读表。它们实际上是视图，而不是基本表，因此，你将无法看到与之相关的任何文件。

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。

查看一个具体数据库的信息

```
mysql> show create database new2007;
+-----+-----+
| Database | Create Database |
+-----+-----+
| new2007 | CREATE DATABASE `new2007` /*!40100 DEFAULT CHARACTER SET utf8 */ |
+-----+-----+
1 row in set (0.00 sec)
```

## 7、显示库中的表



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

## 8、删除数据库

```
mysql> drop database mysqldemo;
Query OK, 0 rows affected (0.08 sec)
```

“Query OK, 0 rows affected (0.08 sec)”表示删除成功，请不要着眼于这里的“0 rows affected”，所有的drop语句都是如此。

删除数据库后将删除其中所有的表。

## 9、表的相关操作

### ● 创建表

```
mysql> create table stu(id int(11),name varchar(50),address text);
Query OK, 0 rows affected (1.30 sec)
```

表是数据库的最基本元素之一，表与表之间可以相互独立，也可以相互关联。创建表的基本语法如下：

```
create table table_name
(column_name column_type{identity |null|not null},
...)
```

其中参数 table\_name 和 column\_name 必须满足用户数据库中的识别器(identifier)的要求，参数 column\_name 是一个标准的 SQL 类型或由用户数据库提供的类型。用户要使用 non-null 从句为各字段输入数据。

create table 还有一些其他选项，如创建临时表和使用 select 子句从其他的表中读取某些字段组成新表等。还有，在创建表是可用 PRIMARY KEY、KEY、INDEX 等标识符设定某些字段为主键或索引等。

书写上要注意：

在一对圆括号里的列出完整的字段清单。

字段名间用逗号隔开。

字段名间的逗号后要加一个空格。

最后一个字段名后不用逗号。

所有的 SQL 陈述都以分号";"结束。

- 查看表结构:

desc tablename

```
mysql> desc stu ;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id    | int(11)| YES  |     | NULL    |       |
| name  | varchar(50)| YES |     | NULL    |       |
| address | text   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.05 sec)
```

如果要查看更全面的表定义信息,则需要通过查看创建表的 SQL 语句来得到。

```
mysql> show create table stu \G;
***** 1. row *****
      Table: stu
Create Table: CREATE TABLE `stu` (
  `id` int(11) default NULL,
  `name` varchar(50) default NULL,
  `address` text
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)

ERROR:
No query specified
```

从上面除了看到表定义语句以外,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分行排列,对于长记录的文件易于显示。

- 修改表结构

ALTER TABLE 变更一个现存表的定义. ADD COLUMN 形式用与 CREATE TABLE 一样的语法向表中增加一个新列/字段. ALTER COLUMN 形式允许你从列/字段中设置或者删除缺省(值)。注意缺省(值)只对新插入的行有效。RENAME 子句可以在不影响相关表中任何数据的情况下更改一个表或者列/字段的名称。因此,表或列/字段在此命令执行后仍将是相同尺寸和类型。ADD table constraint definition 子句用与 CREATE TABLE 一样的语法向表中增加一个新的约束。

如果要改变表的属性,你必须是表的所有者。

向表中增加一个 VARCHAR 列:

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

给前面的学生表添加一年龄列。

```
mysql> alter table stu add column age int(5);
Query OK, 0 rows affected (0.25 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

修改现存列的类型:

```
ALTER TABLE tablename MODIFY COLUMN_NAME COLUMN_TYPE [FIRST | AFTER COLUMN_NAME];
```

```
mysql> alter table stu modify id int(3);  
Query OK, 0 rows affected (0.22 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

修改表字段的顺序:

在修改或添加字段的时候就可以指定它们的位置。

将 age 字段放置于 name 字段之后。

```
mysql> alter table student modify age int(5) after name;  
Query OK, 0 rows affected (0.22 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

注意, 更改列名或列的顺序用的关键词为 change。

对现存列改名:

ALTER TABLE tablename change oldCOLUMN newCOLUMN COLUMNTYPE;

将学生表中的列名为“address”改为“intro”;

```
mysql> alter table stu change address intro text;  
Query OK, 0 rows affected (0.19 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

注意, 更改列名用的关键词为 change。

对现存表改名:

ALTER TABLE tablename RENAME TO suppliers;

```
mysql> alter table stu rename student;  
Query OK, 0 rows affected (0.06 sec)
```

删除表字段:

ALTER TABLE tablename DROP COLUMN\_NAME;

```
mysql> alter table student drop age;  
Query OK, 0 rows affected (0.19 sec)  
Records: 0 Duplicates: 0 Warnings: 0
```

向表中增加一个外键约束:

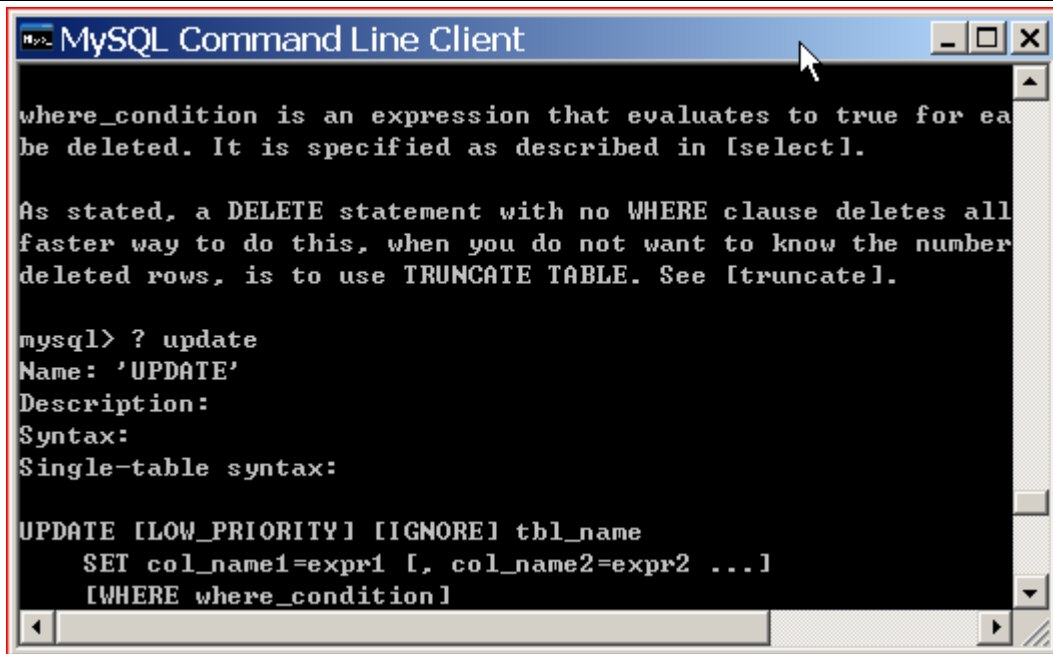
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses(address) MATCH FULL

删除表:

drop TABLE tablename;

注意: 因为 MYSQL 可以使用在不同的平台之上, 而 UNIX 与 WINDOWS 总是有差别的, 前者对大小写敏感, 后者却不敏感, 所以为了避免出现差别, 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。

用? + 关键词可以查询具体的帮助文档。



## 第三章

DML 语句:

DML 是 Data Manipulation Language 的缩写, 意为数据操纵语言, 是 SQL 语言的四大功能之一。由 DBMS 提供, 用于让用户或程序员使用, 实现对数据库中数据的操作。DML 分成交互型 DML 和嵌入式 DML 两类。依据语言的级别, DML 又可分成过程性 DML 和非过程性 DML 两种。

在进行学习之前, 先创建一个名为 test 的数据库, 在里面添加名为 student 的表, 具体操作如下图所示:

```
mysql> use test;  
Database changed  
mysql> create table student(  
  -> id int(5),  
  -> name varchar(30),  
  -> intro varchar(120)  
  -> );  
Query OK, 0 rows affected (0.13 sec)
```

插入记录:

表创建好了, 就可以向里面插入记录了。插入记录的基本语法是:

insert into tablename(field1,field2...) values(value1,vaule2...);

向 student 表中插入记录:

```
mysql> insert into student(id,name,intro) values(1,'xmh','good study');  
Query OK, 1 row affected (0.08 sec)
```

请注意在 values 对应的值中, 如果类型为 varchar 的需要加上单引号。

对于一些可为空的字段, 在不需要添加值的前提下, 可以不用在添加时列出。

```
mysql> insert into student(id,name)
-> values(2,'zxx')
-> ;
Query OK, 1 row affected (0.05 sec)
```

请注意列数要与 values 一一对应。

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

```
mysql> insert into student values(2,'zyj','nice ');
Query OK, 1 row affected (0.05 sec)
```

含有可空字段

非空字段但含有默认值的字段、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, 这样可以缩短 SQL 的复杂性。

例如对 student 表中的 name 字段进行修改, 为其设定默认值为 “zah” 操作如下:

```
mysql> alter table student modify name varchar(30) default 'zah';
Query OK, 1 row affected (0.25 sec)
Records: 1 Duplicates: 0 Warnings: 0
```

此时执行 insert into student(id,intro) values(3,'test'); 语句后, 将发现未赋值的 name 字段将会填写默认值。

用 set 方式插入值。

```
mysql> insert into student set id=5,name='njy',intro='qwe';
Query OK, 1 row affected (0.03 sec)
```

一次插入多条数据

```
mysql> insert into student values(2,'zyj','nice')
-> ,(3,'yzz','deph')
-> ,(4,'zxx','tree')
-> ;
Query OK, 3 rows affected (0.03 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

每条记录之间都用逗号进行了分隔, 这样在插入大量记录的时候, 节省很多网络开销, 大大提高插入效率。

**更新记录:**

将第 3 条记录的姓名更改为 “zah”:

```
mysql> update student set name='zah' where id=3;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

如果没有 where 条件, 将更改所有记录。

如果更改的数据并不存在, 也不会抛错, 只是提示影响的行数为 0;

```
mysql> update student set name='qhxx' where id=7;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0 Changed: 0 Warnings: 0
```

将第 4 条记录的姓名更改为 “zdq”, 简介更改为 “the best”:

```
mysql> update student set name='zdq',intro='the best' where id=4;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

**删除记录:**

如果记录不再需要, 可以删除记录, 语法如下:

DELETE FROM tablename [WHERE CONDITION]

在 student 表中, 删除 id 为 4 的全部记录。

```
mysql> delete from student where id=4;  
Query OK, 1 row affected (0.05 sec)
```

同时删除多表中的数据:

DELETE t1,t2 FROM t1,t2 [WHERE CONDITION]

```
mysql> select * from floatdemo;  
+-----+  
| f      |  
+-----+  
| 5.2    |  
| 5.6    |  
+-----+  
2 rows in set (0.02 sec)  
  
mysql> select * from chardemo;  
+-----+-----+  
| v      | c      |  
+-----+-----+  
| ad     | ab     |  
+-----+-----+  
1 row in set (0.00 sec)  
  
mysql> delete a,b from chardemo a,floatdemo b;  
Query OK, 3 rows affected (0.05 sec)
```

如果 from 后面的表名用别名, 则 delete 后面也要用相应的别名(可用表名 空格 别名或 表名 as 别名), 否则会提示语法错误。

TRUNCATE 清空表数据

```
mysql> truncate student;  
Query OK, 4 rows affected (0.05 sec)
```

TRUNCATE TABLE 用于完全清空一个表。从逻辑上说, 该语句与用于删除所有行的 DELETE 语句等同, 但是在有些情况下, 两者在使用上有所不同。TRUNCATE 效率要高一些, 但很难恢复数据。

### 简单查询

简单的 Transact-SQL 查询只包括选择列表、FROM 子句和 WHERE 子句。它们分别说明所查询列、查询的表或视图、以及搜索条件等。

例如, 下面的语句查询 student 表中 id 为 2 的学生信息。

```
mysql> select name,intro from student where id=2;  
+-----+-----+  
| name | intro |  
+-----+-----+  
| zyj  | nice  |  
+-----+-----+  
1 row in set (0.00 sec)
```

#### (一) 选择列表

选择列表(select\_list)指出所查询列, 它可以是一组列名列表、星号、表达式、变量(包括局部变量和全局变量)等构成。



### 1、选择所有列

例如, 下面语句显示 student 表中所有列的数据:

```
SELECT * FROM student
```

```
mysql> select * from student;
+-----+-----+-----+
| id    | name  | intro    |
+-----+-----+-----+
| 1     | xmh   | good study |
| 3     | zah   | deph     |
| 2     | zyj   | nice     |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

### 2、选择部分列并指定它们的显示次序

查询结果集合中数据的排列顺序与选择列表中所指定的列名排列顺序相同。

例如: SELECT intro,name FROM student

```
mysql> select intro,name from student;
+-----+-----+
| intro    | name  |
+-----+-----+
| good study | xmh   |
| deph     | zah   |
| nice     | zyj   |
+-----+-----+
3 rows in set (0.00 sec)
```

### 3、更改列标题

在选择列表中, 可重新指定列标题。定义格式为:

列标题 列名(列标题就是以前的列名, 后面的列名指更换后的名称)

列标题 as 列名

```
mysql> select name student_name, intro student_intro from student;
+-----+-----+
| student_name | student_intro |
+-----+-----+
| xmh          | good study    |
| zah          | deph          |
| zyj          | nice          |
+-----+-----+
3 rows in set (0.00 sec)
```

如果指定的列标题不是标准的标识符格式时, 应使用引号定界符, 例如, 下列语句使用汉字显示列标题:

```
mysql> select name '姓名', intro '简介' from student;
+-----+-----+
| 姓名 | 简介 |
+-----+-----+
| xmh   | good study |
| zah   | deph     |
| zyj   | nice     |
+-----+-----+
3 rows in set (0.00 sec)
```

更改列的显示顺序,就是对 select 后面出现的列名重新排序即可。

```
mysql> select intro,name,id from student;
+-----+-----+-----+
| intro | name | id |
+-----+-----+-----+
| good study | xmh | 1 |
| dep | zah | 2 |
| nice | zyj | 3 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

#### 4、删除重复行

SELECT 语句中使用 ALL 或 DISTINCT 选项来显示表中符合条件的所有行或删除其中重复的数据行,默认为 ALL。使用 DISTINCT 选项时,对于所有重复的数据行在 SELECT 返回的结果集合中只保留一行。

在学生表中插入一条数据: insert into student values(2,'zyj','nice');

```
mysql> select all * from student;
+-----+-----+-----+
| id | name | intro |
+-----+-----+-----+
| 1 | xmh | good study |
| 2 | zyj | nice |
| 2 | zyj | nice |
+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> select distinct * from student;
+-----+-----+-----+
| id | name | intro |
+-----+-----+-----+
| 1 | xmh | good study |
| 2 | zyj | nice |
+-----+-----+-----+
2 rows in set (0.01 sec)
```

#### 5、使用表达式

```
mysql> select 2<<2;
+-----+
| 2<<2 |
+-----+
| 8 |
+-----+
1 row in set (0.00 sec)
```

#### 6、使用函数

```
mysql> select bin(123);
+-----+
| bin(123) |
+-----+
| 1111011 |
+-----+
1 row in set (0.00 sec)
```

#### 使用 WHERE 子句设置查询条件

WHERE 子句设置查询条件,过滤掉不需要的数据行。例如下面语句查询年龄大于 20 的数据:

查询 id>2 的所有学员。

```
SELECT *  
FROM student  
WHERE id>2
```

WHERE 子句可包括各种条件运算符:

比较运算符(大小比较): >、>=、=、<、<=、<>、!>、!<

范围运算符(表达式值是否在指定的范围): BETWEEN...AND...

NOT BETWEEN...AND...

列表运算符(判断表达式是否为列表中的指定项): IN (项 1,项 2.....)

NOT IN (项 1,项 2.....)

模式匹配符(判断值是否与指定的字符通配格式相符):LIKE、NOT LIKE

空值判断符(判断表达式是否为空): IS NULL、NOT IS NULL

逻辑运算符(用于多条件的逻辑连接): NOT、AND、OR

1、范围运算符例: id BETWEEN 2 AND 5 相当于 id>=2 AND id<=5

```
mysql> select * from student  
-> where id between 2 and 5;  
+-----+-----+-----+  
| id  | name | intro |  
+-----+-----+-----+  
| 3  | zah  | deph  |  
| 2  | zyj  | nice  |  
| 2  | zqy  | nice  |  
| 5  | zxx  | NULL  |  
+-----+-----+-----+  
4 rows in set (0.00 sec)
```

2、列表运算符例: name IN (zah,zxx)相当于 name='zah' or name='zxx'

```
mysql> select * from student  
-> where name in ('zah','zxx');  
+-----+-----+-----+  
| id  | name | intro |  
+-----+-----+-----+  
| 3  | zah  | deph  |  
| 5  | zxx  | NULL  |  
+-----+-----+-----+  
2 rows in set (0.03 sec)
```

3、模式匹配符例: 常用于模糊查找, 它判断列值是否与指定的字符串格式相匹配。可用于 char、varchar、text、ntext、datetime 和 smalldatetime 等类型查询。

可使用以下通配字符:

百分号%: 可匹配任意类型和长度的字符, 如果是中文, 请使用两个百分号即%%。

下划线\_: 匹配单个任意字符, 它常用来限制表达式的字符长度。

例如:

限制以 xx 结尾, 使用 LIKE '%xx'

```
mysql> select * from student
-> where name like 'zxx';
+-----+-----+-----+
| id  | name | intro |
+-----+-----+-----+
| 5   | zxx  | NULL  |
+-----+-----+-----+
1 row in set (0.00 sec)
```

限制以 z 开头: LIKE 'z%'

```
mysql> select * from student
-> where name like 'z%';
+-----+-----+-----+
| id  | name | intro |
+-----+-----+-----+
| 3   | zah  | deph  |
| 2   | zyj  | nice  |
| 2   | zgy  | nice  |
| 5   | zxx  | NULL  |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

空值判断符例 IS NULL

NULL 值表示“没有数据”。NULL 可以写成大写或小写。请注意 NULL 值不同于数字类型的 0 或字符串类型的空字符串。

对于 SQL 的新手, NULL 值的概念常常会造成混淆, 他们常认为 NULL 是与空字符串"相同的事。情况并非如此。例如, 下述语句是完全不同的:

```
mysql> INSERT INTO my_table (phone) VALUES (NULL);
```

```
mysql> INSERT INTO my_table (phone) VALUES ("");
```

这两条语句均会将值插入 phone (电话) 列, 但第 1 条语句插入的是 NULL 值, 第 2 条语句插入的是空字符串。第 1 种情况的含义可被解释为“电话号码未知”, 而第 2 种情况的含义可被解释为“该人员没有电话, 因此没有电话号码”。

为了进行 NULL 处理, 可使用 IS NULL 和 IS NOT NULL 操作符以及 IFNULL() 函数。

在 SQL 中, NULL 值与任何其它值的比较 (即使是 NULL) 永远不会为“真”。包含 NULL 的表达式总是会导出 NULL 值, 除非在关于操作符的文档中以及表达式的函数中作了其他规定。下述示例中的所有列均返回 NULL:

```
mysql> SELECT NULL, 1+NULL, CONCAT('Invisible',NULL);
```

如果打算搜索列值为 NULL 的列, 不能使用 expr = NULL 测试。下述语句不返回任何行, 这是因为, 对于任何表达式, expr = NULL 永远不为“真”: 要想查找 NULL 值, 必须使用 IS NULL 测试。在下面的语句中, 介绍了查找 NULL 简介的方式:

```
mysql> select * from student where intro is null;
+-----+-----+-----+
| id  | name | intro |
+-----+-----+-----+
| 5   | zxx  | NULL  |
+-----+-----+-----+
1 row in set (0.02 sec)
```

4、逻辑运算符: 优先级为 NOT、AND、OR

```
mysql> select * from student where name not in('zah','zxx');
+-----+-----+-----+
| id    | name  | intro    |
+-----+-----+-----+
| 1     | xmh   | good study |
| 2     | zyj   | nice     |
| 2     | zqy   | nice     |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

### 查询结果排序

使用 ORDER BY 子句对查询返回的结果按一列或多列排序。ORDER BY 子句的语法格式为:

ORDER BY {column\_name [ASC|DESC]} [,...n]

其中 ASC 表示升序, 为默认值, DESC 为降序。ORDER BY 不能按 ntext、text 和 image 数据类型进行排序。

指定为升序排列效果如下图所示

```
mysql> select * from student order by intro asc;
+-----+-----+-----+
| id    | name  | intro    |
+-----+-----+-----+
| 5     | zxx   | NULL     |
| 3     | zah   | deph     |
| 1     | xmh   | good study |
| 2     | zyj   | nice     |
| 2     | zqy   | nice     |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

指定为降序排列效果如下图所示

```
mysql> select * from student order by intro desc;
+-----+-----+-----+
| id    | name  | intro    |
+-----+-----+-----+
| 2     | zyj   | nice     |
| 2     | zqy   | nice     |
| 1     | xmh   | good study |
| 3     | zah   | deph     |
| 5     | zxx   | NULL     |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

对于排序后的记录, 如果只希望显示一部分而不是全部, 则使用关键字 LIMIT。它的语法如下:

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。在默认的情况下, 起始偏移量为 0, 只需要写记录数就可以, 这时候, 显示的实际就是前 n 条记录。

对学生表按 id 排序后取前 2 条记录。

```
mysql> select * from student order by id desc limit 2;
+-----+-----+-----+
| id    | name  | intro    |
+-----+-----+-----+
| 3     | zah   | deph     |
| 2     | zyj   | nice     |
+-----+-----+-----+
2 rows in set (0.02 sec)
```

如果给定 2 个参数，第一个指定要返回的第一行的偏移量，第二个指定返回行的最大数目。初始行的偏移量是 0（不是 1）。如果给定一个参数，它指出偏移量为 0 的返回行的最大数目。也就是说 limit 2 和 limit 0,2 完全等价。

对学生表按 id 排序后从第 2 条记录开始取 2 条记录。

```
mysql> select * from student order by id desc limit 1,2;
+-----+-----+-----+
| id    | name  | intro |
+-----+-----+-----+
|      2 | zyj   | nice  |
|      2 | zqy   | nice  |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

### 正则表达式的使用

日常开发过程中，经常会遇到一些常规方法很难完成的工作，其实使用正则表达式就可以快捷地解决相应的难题。MySQL 采用 Henry Spencer 的正则表达式实施，其目标是符合 POSIX 1003.2。请参见附录 C：感谢。MySQL 采用了扩展的版本，以支持在 SQL 语句中与 REGEXP 操作符一起使用的模式匹配操作。

序列	序列说明
^	在字符串的开始处进行匹配
\$	在字符串的末尾始处进行匹配
.	匹配任意单个字符，包括换行符
[...]	匹配出括号内任意字符
[^...]	匹配不出括号内任意字符
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 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

判断“fo\nfo”串是否以“fo”开头和结尾，判断“fofo”串是否以“fo”开头。

```
mysql> select 'fo\nfo' regexp '^fo$',
-> , 'fofo' regexp '^fo';
+-----+-----+-----+
| 'fo\nfo' regexp '^fo$' | 'fofo' regexp '^fo' |
+-----+-----+-----+
|                        |                      |
|                        0 |                      1 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

返回结果为 1 的表示匹配，返回结果为 0 的表示不匹配。

如果用 like 来匹配则将如下所示：



```
mysql> select 'fo\info' like 'fo','fo\info' like '%fo';
+-----+-----+
| 'fo\info' like 'fo' | 'fo\info' like '%fo' |
+-----+-----+
| 0 | 1 |
+-----+-----+
1 row in set (0.00 sec)
```

判断字符串中是否匹配单个字符如下所示。

```
mysql> select 'xmhazh' regexp '.h'
-> , 'xmhazh' regexp '.w';
+-----+-----+
| 'xmhazh' regexp '.h' | 'xmhazh' regexp '.w' |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)
```

判断“xmhazh”是否与“abc”中的任意一个字符匹配，如果有一个匹配上了，返回值为 1；判断“xmh”是否与“xmhazh”中的任意一个字符匹配不上，如果有一个匹配上了，返回值为 0；

```
mysql> select 'xmhazh' regexp "[abc]"
-> , 'xmh' regexp "[^xmhazh]";
+-----+-----+
| 'xmhazh' regexp "[abc]" | 'xmh' regexp "[^xmhazh]" |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)
```

在前面的练习中，如果显示以不以 z 开头学员信息如何写呢？

```
mysql> select * from student where name regexp "^[^z]";
+-----+-----+
| id | name | intro |
+-----+-----+
| 1 | xmh | good study |
+-----+-----+
1 row in set (0.00 sec)
```

匹配姓名中倒数第二个字母含有 a 或 m 的学员信息。

```
mysql> select * from student where name regexp "[am].$";
+-----+-----+
| id | name | intro |
+-----+-----+
| 1 | xmh | good study |
| 3 | zah | deph |
+-----+-----+
2 rows in set (0.00 sec)
```

## 本章小结:

本章对增、删、改、查分别进行了讲解，分别针对相关的用法列举了示例进行演示。通过本章节的学习，对这些用法有了较清晰的了解。并且针对查询引入了正则表达式的相关用法。

## 第四章

DML 语句:

数据库的一个最大的特点就是将各种分散的数据按照一定规律、条件进行分类组合,最后得出统计结果。常用的几个聚合函数如下所示:

函数	说明
AVG	求平均值
COUNT	返回组中项目的数量,返回值为 int 类型
MAX	求最大值
MIN	求最小值
SUN	求和
STDEV	计算统计标准偏差

聚合函数对一组值执行计算并返回单一的值。除 COUNT 函数之外,聚合函数忽略空值 (NULL)。聚合函数经常与 SELECT 语句的 GROUP BY 子句一同使用。所有聚合函数都具有确定性。任何时候用一组给定的输入值调用它们时,都返回相同的值。

聚合函数仅在下列项中允许作为表达式使用:

- SELECT 语句的选择列表 (子查询或外部查询)
- HAVING 子句

首先在 student 表中插入一些数据,如下所示:

```
mysql> select * from student;
+----+-----+-----+-----+
| id  | name | intro      | score |
+----+-----+-----+-----+
| 1   | xmh  | good study | 64    |
| 3   | zah  | depb       | 72    |
| 2   | zyj  | nice       | 68    |
| 2   | zqy  | nice       | 68    |
| 5   | zxx  | NULL       | 80    |
+----+-----+-----+-----+
5 rows in set (0.00 sec)
```

### SUM

SUM 返回表达式中所有数值的总和, SUM 只能用于数字类型的列,不能够汇总字符、日期等其他类型。

下面的示例计算学生的总分:

```
mysql> select sum(score) from student;
+-----+
| sum(score) |
+-----+
|          352 |
+-----+
1 row in set (0.06 sec)
```

注意这种查询只能返回一个数值,因此,不能够直接与可能返回多行的列一起使用参与查询。如下列操作将报告错误,但是,在一个查询中可以使用多个聚合函数。

```
mysql> select sum(score),name from student;
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GR
OUP columns is illegal if there is no GROUP BY clause
```

使用下面的查询，若将价格乘以二，可以得到所有书籍的平均价格：

### AVG

AVG 函数返回表达式中所有数值的平均值，AVG 函数也只能用于数字类型的列，例如求学生的平均成绩。

```
mysql> select avg(score) from student;
+-----+
| avg(score) |
+-----+
|      70.4000 |
+-----+
1 row in set (0.00 sec)
```

### MAX 和 MIN

MAX 和 MIN 函数返回表达式中的最大值和最小值，它们可以用于数字类型、字符型、日期/时间类型的列。

```
mysql> select max(score) as '最高分',min(score) as '最低分' from student;
+-----+-----+
| 最高分 | 最低分 |
+-----+-----+
|      80 |      64 |
+-----+-----+
1 row in set (0.00 sec)
```

### COUNT

COUNT 返回提供的表达式中非空值的计数，COUNT 可以用于数字和字符类型的列。另外，也可以使用星号 (\*) 作为 COUNT 的表达式，使用星号可以不必指定特定的列而计算所有的行数。

```
mysql> select count(*) ,count(intro) from student;
+-----+-----+
| count(*) | count(intro) |
+-----+-----+
|        5 |          4 |
+-----+-----+
1 row in set (0.00 sec)
```

COUNT 函数还可以嵌套另一个函数 DISTINCT，表示求出字段值非空并且惟一的记录个数。

```
mysql> select count(distinct(id)) ,count(id) from student;
+-----+-----+
| count(distinct(id)) | count(id) |
+-----+-----+
|          4 |          5 |
+-----+-----+
1 row in set (0.00 sec)
```

AVG 函数也可以嵌套函数 DISTINCT，表示求出非空且非重复字段的平均值。

```
mysql> select avg(distinct(id)) ,avg(id) from student;
+-----+-----+
| avg(distinct(id)) | avg(id) |
+-----+-----+
|                2.7500 | 2.6000 |
+-----+-----+
1 row in set (0.00 sec)
```

注意: SUM 和 AVG 只能对数字列使用, 例如 int、smallint、tinyint、decimal、numeric、float、real、money 和 smallmoney 数据类型。MIN 和 MAX 不能对 bit 数据类型使用。除 COUNT(\*) 外, 其它聚合函数均不能对 text 和 image 数据类型使用。

### 分组查询

分组函数对每一组数据进行计算, 得到对应的计算数据。在前面的例子中, 返回结果都只有一行, 因为在使用了分组函数的查询语句中并没有对数据进行分组。这意味着整个表中的所有数据被作为一组进行了统计计算, 所以最终得到了一行结果。

除了对整个表的数据进行统计计算外, 更多的需求是根据用户的实际需要对表中的数据进行分组, 然后对每个分组进行组函数计算。使用 Group by 子句可以对表中的数据进行分组。

```
SELECT COLUMN, GROUP_FUNCTION
FROM TABLE
[WHERE CONDITION]
[GROUP BY GROUP_BY_EXPRESSION]
[ORDER BY COLUMN];
```

规则 1: 出现在 SELECT 后面的字段, 如果出现的位置不在分组函数中, 那么必须要出现在 GROUP BY 子句中。

```
mysql> select avg(id), score from student group by score;
+-----+-----+
| avg(id) | score |
+-----+-----+
| 1.0000 | 64 |
| 2.0000 | 68 |
| 3.0000 | 72 |
| 5.0000 | 80 |
+-----+-----+
4 rows in set (0.00 sec)
```

规则 2: 出现在 GROUP BY 子句中的字段并不一定要出现在 SELECT 后面。

```
mysql> select avg(id) from student group by score;
+-----+
| avg(id) |
+-----+
| 1.0000 |
| 2.0000 |
| 3.0000 |
| 5.0000 |
+-----+
4 rows in set (0.00 sec)
```

在 GROUP BY 子句中, 可以按单列进行分组, 也可以在多列上进行分组, 多列分组是按照多个字段的组合进行分组, 最终的结果也会按照分组字段进行排序显示。

```
mysql> select avg(id) from student group by score,name;
+-----+
| avg(id) |
+-----+
| 1.0000 |
| 2.0000 |
| 2.0000 |
| 3.0000 |
| 5.0000 |
+-----+
5 rows in set (0.00 sec)
```

规则 3: WHERE 子句中不能使用聚合函数。

使用 Having 子句对分组结果进行限制

```
SELECT COLUMN, GROUP_FUNCTION
FROM TABLE
[WHERE CONDITION]
[GROUP BY GROUP_BY_EXPRESSION]
[HAVING GROUP_CONDITION]
[ORDER BY COLUMN];
```

这是一个完整 SELECT 查询语句，在整个语句的执行过程中，最先执行的是 WHERE 子句，在对表数据进行过滤后，符合条件的数据通过 GROUP BY 进行分组，分组数据通过 HAVING 子句进行组函数过滤，最终的结果通过 ORDER BY 子句进行排序，排序的结果被返回给用户。

对分数高于 65 分的学生进行分组，统计平均分高于 70 分组人数和平均分数。

```
mysql> select count(id), score from student
-> where score > 65
-> group by score
-> having avg(score) > 70
-> order by id;
+-----+-----+
| count(id) | score |
+-----+-----+
| 1 | 72 |
| 1 | 80 |
+-----+-----+
2 rows in set (0.00 sec)
```

## 二、联合查询

UNION 运算符可以将两个或两个以上 SELECT 语句的查询结果集合并成一个结果集显示，即执行联合查询。UNION 的语法格式为：

select\_statement

UNION [ALL] selectstatement

[UNION [ALL] selectstatement][...n]

其中 selectstatement 为待联合的 SELECT 查询语句。

ALL 选项表示将所有行合并到结果集中。不指定该项时，被联合查询结果集中的重复行将只保留一行。

联合查询时，查询结果的列标题为第一个查询语句的列标题。因此，要定义列标题必须在第一个查询语句中定义。要对联合查询结果排序时，也必须使用第一查询语句中的列名、列标题或者列序号。

在使用 UNION 运算符时, 应保证每个联合查询语句的选择列表中有相同数量的表达式, 并且每个查询选择表达式应具有相同的数据类型, 或是可以自动将它们转换为相同的数据类型。在自动转换时, 对于数值类型, 系统将低精度的数据类型转换为高精度的数据类型。

在包括多个查询的 UNION 语句中, 其执行顺序是自左至右, 使用括号可以改变这一执行顺序。例如:

查询 1 UNION (查询 2 UNION 查询 3)

表 I student 表

```
mysql> desc student;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(8)        | YES  |     | NULL    |       |
| name  | varchar(50)   | YES  |     | NULL    |       |
| intro | text          | YES  |     | NULL    |       |
| score | int(5)        | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

表 II stu 表

```
mysql> desc stu;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(5)        | YES  |     | NULL    |       |
| name  | varchar(30)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.02 sec)
```

执行联合查询的效果:

```
mysql> select id,name from student
-> union
-> select * from stu;
+-----+-----+
| id  | name |
+-----+-----+
| 1   | xmh  |
| 3   | zah  |
| 2   | zyj  |
| 2   | zqy  |
| 5   | zxx  |
| 1   | xdds  |
| 2   | wer  |
+-----+-----+
7 rows in set (0.00 sec)
```

通过上例观察, 列字段大小不一致也不会影响联合查询的执行。这里可能会产生一个疑问, 如果列名不一致, 是否可以用 UNION 查询?

对 stu 表进行更改列名:

```
mysql> alter table stu change name na varchar(50);
Query OK, 2 rows affected (0.59 sec)
Records: 2 Duplicates: 0 Warnings: 0
```

然后执行 UNION 查询结果与前面一样。

如果改变列的类型呢?

```
mysql> alter table stu modify na text;  
Query OK, 2 rows affected (0.20 sec)  
Records: 2 Duplicates: 0 Warnings: 0
```

执行 UNION 查询结果与前面一样。

通过以上示例,说明 UNION 查询只需要相关的表具有相同的列数即可。

### 三、连接查询

通过连接运算符可以实现多个表查询。连接是关系数据库模型的主要特点,也是它区别于其它类型数据库管理系统的一个标志。

在关系数据库管理系统中,表建立时各数据之间的关系不必确定,常把一个实体的所有信息存放在一个表中。当检索数据时,通过连接操作查询出存放在多个表中的不同实体的信息。连接操作给用户带来很大的灵活性,他们可以在任何时候增加新的数据类型。为不同实体创建新的表,尔后通过连接进行查询。

SQL-92 标准所定义的 FROM 子句的连接语法格式为:

Select columns FROM join\_table join\_type join\_table [ON (join\_condition)]

其中 join\_table 指出参与连接操作的表名,连接可以对同一个表操作,也可以对多表操作,对同一个表操作的连接又称做自连接。

join\_type 指出连接类型,可分为三种:内连接、外连接和交叉连接。内连接(INNER JOIN)使用比较运算符进行表间某(些)列数据的比较操作,并列出行中与连接条件相匹配的数据行。根据所使用的比较方式不同,内连接又分为等值连接、自然连接和不等连接三种。外连接分为左外连接(LEFT OUTER JOIN 或 LEFT JOIN)、右外连接(RIGHT OUTER JOIN 或 RIGHT JOIN)和全外连接(FULL OUTER JOIN 或 FULL JOIN)三种。与内连接不同的是,外连接不只列出与连接条件相匹配的行,而是列出左表(左外连接时)、右表(右外连接时)或两个表(全外连接时)中所有符合搜索条件的数据行。

交叉连接(CROSS JOIN)没有 WHERE 子句,它返回连接表中所有数据行的笛卡尔积,其结果集合中的数据行数等于第一个表中符合查询条件的数据行数乘以第二个表中符合查询条件的数据行数。

如对 student 和 stu 表执行交叉连接,效果如下图所示。

```
mysql> select t.id,t.name from student as t  
-> cross join stu;  
+-----+-----+  
| id | name |  
+-----+-----+  
| 1 | xmh |  
| 1 | xmh |  
| 3 | zah |  
| 3 | zah |  
| 2 | zyj |  
| 2 | zyj |  
| 2 | zqy |  
| 2 | zqy |  
| 5 | zxx |  
| 5 | zxx |  
+-----+-----+  
10 rows in set (0.00 sec)
```

连接操作中的 ON (join\_condition) 子句指出连接条件,它由被连接表中的列和比较运算符、逻辑运算符等构成。

#### 内连接

内连接查询操作列出与连接条件匹配的数据行,它使用比较运算符比较被连接列的列值。内连接



分三种:

1、等值连接: 在连接条件中使用等于号(=)运算符比较被连接列的列值, 其查询结果中列出被连接表中的所有列, 包括其中的重复列。

2、不等连接: 在连接条件使用除等于运算符以外的其它比较运算符比较被连接的列的列值。这些运算符包括>、>=、<=、<、!=、!<和!>。

3、自然连接: 在连接条件中使用等于(=)运算符比较被连接列的列值, 但它使用选择列表指出查询结果集合中所包括的列, 并删除连接表中的重复列。

### 外连接

内连接时, 返回查询结果集合中的仅是符合查询条件( WHERE 搜索条件或 HAVING 条件)和连接条件的行。而采用外连接时, 它返回到查询结果集合中的不仅包含符合连接条件的行, 而且还包括左表(左外连接时)、右表(右外连接时)中的所有数据行。简言之即左外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, 右外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。右连接和左连接类似, 两者是可以相互转化的。

下面通过一个示例来演示内连接和外连接的相关用法。

创建两个表:

Employee (员工信息表)

员工编号(eid)	姓名(name)	部门编号(did)
1	李小飞	1
2	郑 波	1
3	关思宇	2
4	戊卫成	2
5	朱妙妙	NULL

Dept (部门信息表)

部门编号(id)	部门名称(dname)
1	技术部
2	市场部
3	工程部

```
create table employee(
  eid int(5),
  name varchar(30),
  did int(5)
);
create table dept(
  id int(5),
  dname varchar(30)character set utf8 collate utf8_general_ci
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into dept values(1,'技术部'),(2,'市场部'),(3,'工程部');
insert into Employee values(1,'李小飞',1),(2,'郑波',1),(3,'关思宇',2),(4,'戊卫成',2),(5,'朱妙妙',null);
如果插入中文时提示长度不够时, 请先执行 set names gbk;
```

注意: 朱妙妙不属于任何部门(新来的员工, 还没有分配到任何的部门), 而工程部不存在任何的员工(比如是一个新成立的部门, 还没有员工)

## 1、内连接查询

查询出各员工的信息以及所在的部门名称

我们可以有两种方式，这两种是等效的

一种是：

```
select e.eid '员工 ID',  
e.name '员工姓名',  
d.dname '部门名称'  
from employee e,dept d  
where e.did=d.id
```

另外一个：

```
select e.eid '员工 ID',  
e.name '员工姓名',  
d.dname '部门名称'  
from employee e inner join dept d  
on e.did=d.id
```

检索的结果都是：

```
+-----+-----+-----+  
| 员工ID | 员工姓名 | 部门名称 |  
+-----+-----+-----+  
|      1 | 李小飞   | 技术部   |  
|      2 | 郑波     | 技术部   |  
|      3 | 关思宇   | 市场部   |  
|      4 | 戊卫成   | 市场部   |  
+-----+-----+-----+  
4 rows in set (0.00 sec)
```

而“朱妙妙”和“工程部”的信息是不会检索出来。因为采用内连接计算的时候必须要保证连接的条件 `e.deptid=d.deptid` 匹配，结果才会被检索出来。当我们连接两张检索数据的时候，检索的方式是首先逐行扫描“员工信息表”中的记录，然后根据连接条件来决定此记录是否被检索。比如对于李小飞，这条记录的 `deptid` 是 1（部门编号），它在部门表中能找到和它匹配的编号 1，而编号 1 的部门名称（`deptname`）是“技术部”所以张三这条记录会被检索，最终的结果肯定是：

0001 李小飞 技术部

同样，郑波、关思宇等都被检索出来。但是朱妙妙的部门编号是 `NULL`，它在部门信息表中找不到匹配的项（因为部门信息表中不存在部门编号为 `NULL` 的部门），所以朱妙妙不会被检索。

同理，没有任何人员的部门编号为 03，所以工程部的记录也不会被检索。

## 2、左外联结

但是有些情况下，我们需要知道所有员工的信息，即使他不属于任何部门。这样我们就可以采用外连接，在这里为左外连接，也就是连接中的左表的表中的记录，无论能不能在右表中找到匹配的项，都要检索，如果没有匹配的项目，那么右表中的字段值为 `NULL`（空），在这里就代表，此员工不属于任何部门。

显示所有的员工和部门名称，包括新员工。

检索语句为：

```
select e.eid '员工 ID',  
e.name '员工姓名',  
d.dname '部门名称'  
from employee e left join dept d  
on e.did=d.id
```

检索的结果是：

```

+-----+-----+-----+
| 员工ID | 员工姓名 | 部门名称 |
+-----+-----+-----+
| 1 | 李小飞 | 技术部 |
| 2 | 郑波 | 技术部 |
| 3 | 关思宇 | 市场部 |
| 4 | 戊卫成 | 市场部 |
| 5 | 朱妙妙 | NULL |
+-----+-----+-----+
5 rows in set (0.00 sec)

```

但是在这里，工程部同样不会被检索，因为，deptname 是在连接的右边的表中，“工程部”在左表中不存在任何的记录，所以不会被检索。这里关注的是“连接中的左边的表”

### 3、右外连接

有时，我们需要知道，全部部门的信息，即使它没有任何的员工。在我们的查询中部门表在连接的右边，如果我们想知道右边表中的所有记录信息，那么就可以采用右外连接，如果此记录在左边的表中找不到匹配项，则相应字段 (employeeid,employeenam)为 NULL

检索语句为：

```

select e.eid '员工 ID',
e.name '员工姓名',
d.dname '部门名称'
from employee e right join dept d
on e.did=d.id

```

检索的结果是：

```

+-----+-----+-----+
| 员工ID | 员工姓名 | 部门名称 |
+-----+-----+-----+
| 1 | 李小飞 | 技术部 |
| 2 | 郑波 | 技术部 |
| 3 | 关思宇 | 市场部 |
| 4 | 戊卫成 | 市场部 |
| NULL | NULL | 工程部 |
+-----+-----+-----+
5 rows in set (0.00 sec)

```

但在这里，朱妙妙是不会被检索了，因为它在右表中找不到匹配项，这里关注的是“连接中的右边的表”

这里考虑一下，如果要统计出各部门里员工的人数，如何操作呢？

```

select d.dname as '部门名称',count(e.eid) as '员工人数'
from employee as e right join dept as d
on e.did=d.id
group by did;

```

检索的结果是：

```

+-----+-----+
| 部门名称 | 员工人数 |
+-----+-----+
| 工程部 | 0 |
| 技术部 | 2 |
| 市场部 | 2 |
+-----+-----+
3 rows in set (0.00 sec)

```

子查询:

某些情况下, 当进行查询的时候, 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。

例如要查询所有部门的员工信息

```
select * from employee
where did in(
select id from dept)
;
```

检索的结果是:

```
+-----+-----+-----+
| eid | name | did |
+-----+-----+-----+
| 1 | 李小飞 | 1 |
| 2 | 郑波 | 1 |
| 3 | 关思宇 | 2 |
| 4 | 戊卫成 | 2 |
+-----+-----+-----+
4 rows in set (0.14 sec)
```

如果子查询的结果是唯一的, 还可以用=代替 in。

```
select * from employee
where did =(
select id from dept)
;
```

结果不唯一时会报错。

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

```
select * from employee
where did =(
select id from dept limit 1)
;
```

检索的结果是:

```
+-----+-----+-----+
| eid | name | did |
+-----+-----+-----+
| 1 | 李小飞 | 1 |
| 2 | 郑波 | 1 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

某些情况下, 子查询是可以转化为表连接的, 例如前面的查询所有部门的员工信息就可以转化为表连接。

```
select employee.* from employee
,dept where
employee.did=dept.id;
```

注意:

MYSQL4.1 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。

练习:

1、某公司产品的年销量报表如下所示:

年	产品	销量
2005	a	700
2005	b	550
2005	c	600
2006	a	340
2006	b	500
2007	a	500
2007	b	800

请按销量统计出各年销售最好的产品信息, 结果如下表所示:

年	产品	销量
2005	a	700
2006	b	500
2007	b	800

答案: 注意要屏掉重复的数据。

```
select year,prods,max(num) from products
where num in(
select max(num)
from products group by year)
group by year;
```

```
mysql> select year,prods,max(num) from products
-> where num in(
-> select max(num)
-> from products group by year)
-> group by year;
+-----+-----+-----+
| year | prods | max(num) |
+-----+-----+-----+
| 2005 | a     | 700      |
| 2006 | b     | 500      |
| 2007 | a     | 800      |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

2、新建的 class 表和 person 表:

```
mysql> desc class;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| classid | int(4)        | YES  |     | NULL    |       |
| name    | varchar(10)   | YES  |     | NULL    |       |
| intro   | varchar(20)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> desc person;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(4)        | YES  |     | NULL    |       |
| name  | varchar(10)   | YES  |     | NULL    |       |
| age   | int(4)        | YES  |     | NULL    |       |
| cid   | int(4)        | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

插入一些数据

```
create table class(classid int(4),name varchar(10),intro varchar(20));
create table person(id int(4),name varchar(10),age int(4),cid int(4));
```

```
insert into class values(1,'语文',null),(2,'数学',null),
(3,'英语',null),(4,'化学',null),
(5,'政治',null),(6,'历史',null),(7,'地理',null);
insert into person values(1,'张三',23,1),(1,'李四',23,1),
(1,'王五',20,2),(1,'周周',18,2),(1,'团团',19,3),(1,'贺小军',21,3),
(1,'王小庆',23,4),(1,'李洁',18,5),(1,'王敏',21,6),
(1,'刘华',27,4),(1,'王明',22,null);
```

1、查询出选择课程的所有学员？

```
mysql> select * from person p
-> inner join class c
-> on p.cid=c.classid;
+-----+-----+-----+-----+-----+-----+-----+
| id | name | age | cid | classid | name | intro |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 张三 | 23 | 1 | 1 | 语文 | NULL |
| 1 | 李四 | 23 | 1 | 1 | 语文 | NULL |
| 1 | 王五 | 20 | 2 | 2 | 数学 | NULL |
| 1 | 周周 | 18 | 2 | 2 | 数学 | NULL |
| 1 | 团团 | 19 | 3 | 3 | 英语 | NULL |
| 1 | 贺小军 | 21 | 3 | 3 | 英语 | NULL |
| 1 | 王小庆 | 23 | 4 | 4 | 化学 | NULL |
| 1 | 李洁 | 18 | 5 | 5 | 政治 | NULL |
| 1 | 王敏 | 21 | 6 | 6 | 历史 | NULL |
| 1 | 刘华 | 27 | 4 | 4 | 化学 | NULL |
+-----+-----+-----+-----+-----+-----+-----+
10 rows in set (0.00 sec)
```

2、查询出所有被学员选择的课程信息

```
mysql> select * from class c inner join person p on p.cid=c.classid;
+-----+-----+-----+-----+-----+-----+-----+
| classid | name | intro | id | name | age | cid |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 语文 | NULL | 1 | 张三 | 23 | 1 |
| 1 | 语文 | NULL | 1 | 李四 | 23 | 1 |
| 2 | 数学 | NULL | 1 | 王五 | 20 | 2 |
| 2 | 数学 | NULL | 1 | 周周 | 18 | 2 |
| 3 | 英语 | NULL | 1 | 团团 | 19 | 3 |
| 3 | 英语 | NULL | 1 | 贺小军 | 21 | 3 |
| 4 | 化学 | NULL | 1 | 王小庆 | 23 | 4 |
| 5 | 政治 | NULL | 1 | 李洁 | 18 | 5 |
| 6 | 历史 | NULL | 1 | 王敏 | 21 | 6 |
| 4 | 化学 | NULL | 1 | 刘华 | 27 | 4 |
+-----+-----+-----+-----+-----+-----+-----+
10 rows in set (0.00 sec)
```

以上两句都可以用 where 子句来改写, 不过那种方式的内联接被称为旧式内联接。

### 3、查询出所有的课程信息, 包括没有被学生选择的课程

```
mysql> select * from class c left join person p on p.cid=c.classid;
+-----+-----+-----+-----+-----+-----+-----+
| classid | name | intro | id | name | age | cid |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 语文 | NULL | 1 | 张三 | 23 | 1 |
| 1 | 语文 | NULL | 1 | 李四 | 23 | 1 |
| 2 | 数学 | NULL | 1 | 王五 | 20 | 2 |
| 2 | 数学 | NULL | 1 | 周周 | 18 | 2 |
| 3 | 英语 | NULL | 1 | 团团 | 19 | 3 |
| 3 | 英语 | NULL | 1 | 贺小军 | 21 | 3 |
| 4 | 化学 | NULL | 1 | 王小庆 | 23 | 4 |
| 4 | 化学 | NULL | 1 | 刘华 | 27 | 4 |
| 5 | 政治 | NULL | 1 | 李洁 | 18 | 5 |
| 6 | 历史 | NULL | 1 | 王敏 | 21 | 6 |
| 7 | 地理 | NULL | NULL | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

### 4、查询出所有的学生信息, 包括没有选择课程的学生

```
mysql> select * from class c right join person p on p.cid=c.classid ;
+-----+-----+-----+-----+-----+-----+-----+
| classid | name | intro | id | name | age | cid |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | 语文 | NULL | 1 | 张三 | 23 | 1 |
| 1 | 语文 | NULL | 1 | 李四 | 23 | 1 |
| 2 | 数学 | NULL | 1 | 王五 | 20 | 2 |
| 2 | 数学 | NULL | 1 | 周周 | 18 | 2 |
| 3 | 英语 | NULL | 1 | 团团 | 19 | 3 |
| 3 | 英语 | NULL | 1 | 贺小军 | 21 | 3 |
| 4 | 化学 | NULL | 1 | 王小庆 | 23 | 4 |
| 5 | 政治 | NULL | 1 | 李洁 | 18 | 5 |
| 6 | 历史 | NULL | 1 | 王敏 | 21 | 6 |
| 4 | 化学 | NULL | 1 | 刘华 | 27 | 4 |
| NULL | NULL | NULL | 1 | 王明 | 22 | NULL |
+-----+-----+-----+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```



5、查询哪些学生没有选择课程？

```
mysql> select * from person where cid is null;
+-----+-----+-----+-----+
| id   | name | age  | cid  |
+-----+-----+-----+-----+
| 1    | 王明 | 22   | NULL |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

6、查询哪些课程没有被学生选择？

```
mysql> select * from class where classid not in
-> (select cid from person p where p.cid is not null);
+-----+-----+-----+
| classid | name | intro |
+-----+-----+-----+
| 7       | 地理 | NULL  |
+-----+-----+-----+
1 row in set (0.00 sec)
```

本章小结：

本章对聚合函数与分组函数分别进行了讲解，分别针对相关的用法列举了示例进行演示。通过本章的学习，对这些用法有了较清晰的了解。

## 第 五 章

我们要把现实世界中的各种信息转换成计算机能理解的东西，这些转换后的信息就形成了数据。例如，某人的出生日期是“1987年5月23日”，他的身高是170厘米，等等。数据不仅包括数字、字母、文字和其他特殊字符组成的文本形式的数据，而且还包括图形、图像、动画、影像、声音等多媒体数据。但使用最多、最基本的仍然是文本数据。

我们用 Create Table 语句创建一个表（参看前面的章节），这个表中包含列的定义。例如我们在前面创建了一个 joke 表，这个表中有 content 和 writer 两个列：

```
CREATE TABLE student (
  id int(8) default NULL,
  name varchar(50) default NULL,
  intro text,
  score int(5) default NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ;
```

定义一个列的语法如下：

```
col_name col_type [col_attributes] [general_attributes]
```

其中列名由 col\_name 给出。列名可最多包含 64 个字符，字符包括字母、数字、下划线及美元符号。列名可以名字中合法的任何符号（包括数字）开头。但列名不能完全由数字组成，因为那样可能使其与数据分不开。MySQL 保留诸如 SELECT、DELETE 和 CREATE 这样的词，这些词不能用做列名，但是函数名（如 POS 和 MIN）是可以使用的。

列类型 col\_type 表示列可存储的特定值。列类型说明符还能表示存放在列中的值的最大长度。对于

某些类型,可用一个数值明确地说明其长度。而另外一些值,其长度由类型名蕴含。例如, **CHAR(10)** 明确指定了 10 个字符的长度,而 **TINYBLOB** 值隐含最大长度为 255 个字符。有的类型说明符允许指定最大的显示宽度(即显示值时使用多少个字符)。浮点类型允许指定小数位数,所以能控制浮点数的精度值为多少。

可以在列类型之后指定可选的类型说明属性,以及指定更多的常见属性。属性起修饰类型的作用,并更改其处理列值的方式,属性有以下类型:

(1) 专用属性用于指定列。例如, **UNSIGNED** 属性只针对整型,而 **BINARY** 属性只用于 **CHAR** 和 **VARCHAR**。

(2) 通用属性除少数列之外可用于任意列。可以指定 **NULL** 或 **NOT NULL** 以表示某个列是否能够存放 **NULL**。还可以用 **DEFAULT**, **def\_value** 来表示在创建一个新行但未明确给出该列的值时,该列可赋予值 **def\_value**。**def\_value** 必须为一个常量;它不能是表达式,也不能引用其他列。不能对 **BLOB** 或 **TEXT** 列指定缺省值。

如果想给出多个列的专用属性,可按任意顺序指定它们,只要它们跟在列类型之后、通用属性之前即可。类似地,如果需要给出多个通用属性,也可按任意顺序给出它们,只要将它们放在列类型和可能给出的列专用属性之后即可。

### MySQL 的列(字段)类型

数据库中的每个表都是由一个或多个列(字段)构成的。在用 **CREATE TABLE** 语句创建一个表时,要为每列(字段)指定一个类型。列(字段)的类型比数据类型更为细化,它精确地描述了给定表列(字段)可能包含的值的种类,如是否带小数、是否文字很多。

## 2.1 数值列类型

MySQL 有整数和浮点数值类型的列类型,如表 1 所示。整数列类型可以有符号也可无符号。有一种特殊的属性允许整数列值自动生成,这对需要唯一序列或标识号的应用系统来说是非常有用的。

类型	字节	取值范围	说明
TINYINT	1	有符号值: -128 到 127 (- 27 到 27 - 1) 无符号值: 0 到 255 (0 到 28 - 1)	非常小的整数
SMALLINT	2	有符号值: -32768 到 32767 (- 215 到 215 - 1) 无符号值: 0 到 65535 (0 到 21 6 - 1)	较小整数
MEDIUMINT	3	有符号值: -8388608 到 8388607 (- 22 3 到 22 3 - 1 ) 无符号值: 0 到 16777215 (0 到 22 4 - 1)	中等大小整数
INT	4	有符号值: -2147683648 到 2147683647 (- 231 到 231- 1) 无符号值: 0 到 4294967295 (0 到 232 - 1)	标准整数
BIGINT	8	有符号值: -9223372036854775808 到 9223373036854775807 (- 263 到 263-1) 无符号值: 0 到 18446744073709551615 (0 到 264 - 1)	较大整数
FLOAT	4	最小非零值: $\pm 1.175494351E - 38$	单精度浮点数
DOUBLE	8	最小非零值: $\pm 2.2250738585072014E - 308$	双精度浮点数
DECIMAL (M, D)	M+2	可变;其值的范围依赖于 M 和 D	一个串的浮点数

表 4: MYSQL 中的数值类型

MySQL 提供了五种整型: **TINYINT**、**SMALLINT**、**MEDIUMINT**、**INT** 和 **BIGINT**。**INT** 为 **INTEGER** 的缩写。这些类型在可表示的取值范围上是不同的。整数列可定义为 **UNSIGNED** 从而禁用负值;这使列的取值范围为 0 以上。各种类型的存储量需求也是不同的。取值范围较大的类型所需的存储量较大。

```
mysql> use test;
Database changed
mysql> create table testnum(first int,second int<5>);
Query OK, 0 rows affected (0.27 sec)

mysql> desc testnum;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| first | int(11) | YES  |     | NULL    |       |
| second | int(5)  | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

在表中插入数据并显示

```
mysql> insert into testnum values(1,1);
Query OK, 1 row affected (0.05 sec)

mysql> select * from testnum;
+-----+-----+
| first | second |
+-----+-----+
| 1     | 1     |
+-----+-----+
1 row in set (0.03 sec)
```

修改字段类型, 加入 zerofill 参数

```
mysql> alter table testnum modify first int zerofill;
Query OK, 1 row affected (0.31 sec)
Records: 1 Duplicates: 0 Warnings: 0

mysql> select * from testnum;
+-----+-----+
| first      | second |
+-----+-----+
| 0000000001 | 1     |
+-----+-----+
1 row in set (0.00 sec)
```

如果插入的数超过了范围

```
mysql> insert into testnum values(1,1111111);
Query OK, 1 row affected (0.05 sec)

mysql> select * from testnum;
+-----+-----+
| first      | second |
+-----+-----+
| 0000000001 | 1     |
| 0000000001 | 1111111 |
+-----+-----+
2 rows in set (0.00 sec)
```

从结果可以看出, 如果插入大于宽度限制的值, 还是按照类型的实际精度进行保存。这是, 宽度格式实际没有意义, 左边不会再填充任何的“0”字符。

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, 上限是原值的 2 倍。例如 tinyint 无符号的取值范围是 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。它一般从 1 开始, 每行增加 1。一个表中最多只能有一个 `AUTO_INCREMENT` 列。对于任何想使用 `AUTO_INCREMENT` 的都需要定义为 `NOT NULL`, 并定义为 `PRIMARY KEY` 或定义为 `UNIQUE` 键。

MySQL 提供三种浮点类型: `FLOAT`、`DOUBLE` 和 `DECIMAL`。与整型不同, 浮点类型不能是 `UNSIGNED` 的, 其取值范围也与整型不同, 这种不同不仅在于这些类型有最大值, 而且还有最小非零值。最小值提供了相应类型精度的一种度量, 这对于记录科学数据来说是非常重要的(当然, 也有负的最大和最小值)。

```
mysql> create table testnum2(first float(5,2),second double(5,2),third decimal(5,2));
Query OK, 0 rows affected (0.09 sec)

mysql> insert into testnum2 values(1.24,1.24,1.24);
Query OK, 1 row affected (0.03 sec)

mysql> select * from testnum2;
+-----+-----+-----+
| first | second | third |
+-----+-----+-----+
| 1.24 | 1.24 | 1.24 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

如果插入超出它们所规定范围的值会如何呢?

```
mysql> insert into testnum2 values(1.245,1.245,1.245);
Query OK, 1 row affected, 1 warning (0.05 sec)

mysql> select * from testnum2;
+-----+-----+-----+
| first | second | third |
+-----+-----+-----+
| 1.24 | 1.24 | 1.24 |
| 1.25 | 1.25 | 1.25 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

由于精度的限制, 会进行四舍五入。

修改表结构, 将字段的精度和标度全部去掉。然后再插入一条数据, 看看结果将有何变化呢?

```
mysql> alter table testnum2 modify first float;
Query OK, 2 rows affected (0.31 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> alter table testnum2 modify second double;
Query OK, 2 rows affected (0.25 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> alter table testnum2 modify third decimal;
Query OK, 2 rows affected, 2 warnings (0.25 sec)
Records: 2  Duplicates: 0  Warnings: 2

mysql> insert into testnum2 values(1.245,1.245,1.245);
Query OK, 1 row affected, 1 warning (0.05 sec)

mysql> select * from testnum2;
+-----+-----+-----+
| first | second | third |
+-----+-----+-----+
| 1.24  | 1.24   | 1     |
| 1.25  | 1.25   | 1     |
| 1.245 | 1.245  | 1     |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

如果浮点数不写精度和标度,则会按照实际的精度值显示。如果有写精度和标度则会进行四舍五入插入,系统不会报错。定点数如何不写精度和标度,则按默认值 decimal(10,0)来进行操作,如果数值超过它的精度和标度范围,系统则会报错。

在选择了某种数值类型时,应该考虑所要表示的值的范围,只需选择能覆盖要取值的范围的最小类型即可。选择较大类型会对空间造成浪费,使表不必要地增大,处理起来没有选择较小类型那样有效。对于整型值,如果数据取值范围较小,如人员年龄或兄弟姐妹数,则 TINYINT 最合适。MEDIUMINT 能够表示数百万的值并且可用于更多类型的值,但存储代价较大。BIGINT 在全部整型中取值范围最大,而且需要的存储空间是表示范围次大的整型 INT 类型的两倍,因此只在确实需要时才用。对于浮点值,DOUBLE 占用 FLOAT 的两倍空间。除非特别需要高精度或范围极大的值,一般应使用只用一半存储代价的 FLOAT 型来表示数据。

在定义整型列时,可以指定可选的显示尺寸 M。如果这样,M 应该是一个 1 到 255 的整数。它表示用来显示列中值的字符数。例如,MEDIUMINT(4)指定了一个具有 4 个字符显示宽度的 MEDIUMINT 列。如果定义了一个没有明确宽度的整数列,将会自动分配给它一个缺省的宽度。缺省值为每种类型的“最长”值的长度。如果某个特定值的可打印表示需要不止 M 个字符,则显示完全的值;不会将值截断以适合 M 个字符。

对每种浮点类型,可指定一个最大的显示尺寸 M 和小数位数 D。M 的值应该取 1 到 255。D 的值可为 0 到 30,但是不应大于 M - 2(如果熟悉 ODBC 术语,就会知道 M 和 D 对应于 ODBC 概念的“精度”和“小数点位数”)。M 和 D 对 FLOAT 和 DOUBLE 都是可选的,但对于 DECIMAL 是必须的。在选项 M 和 D 时,如果省略了它们,则使用缺省值。

BIT(位)类型,用于存放位字段,BIT(M)可以用来存放多位二进制,M 的范围从 1~64,如果不写就默认为 1。对于位字段,直接用 SELECT 命令将不会看到结果。可以用 bin()或者 hex()函数进行读取。

```
mysql> create table tbit(b BIT(8));
Query OK, 0 rows affected (0.13 sec)

mysql> insert into tbit values(11);
Query OK, 1 row affected (0.05 sec)
```

输入的数据会以二进制的形式存储，所以以二进制、八进制和十六进制的方式查询。

```
mysql> SELECT b+0, BIN(b+0), OCT(b+0), HEX(b+0) FROM tbit;
+-----+-----+-----+-----+
| b+0 | BIN(b+0) | OCT(b+0) | HEX(b+0) |
+-----+-----+-----+-----+
| 11 | 1011 | 13 | B |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

## 2.2 字符串列类型

MySQL 提供了几种存放字符数据的串类型，其类型如下：

类型名	最大尺寸	存储需求	说明
CHAR	M 字节	M 字节	定长字符串
VARCHAR	M 字节	L + 1 字节	可变长字符串
TINYBLOB	2 <sup>8</sup> - 1 字节	L + 1 字节	非常小的 BLOB（二进制大对象）
BLOB	2 <sup>16</sup> - 1 字节	L + 2 字节	小 BLOB
MEDIUMBLOB	2 <sup>24</sup> - 1 字节	L + 3 字节	中等的 BLOB
LONGBLOB	2 <sup>32</sup> - 1 字节	L + 4 字节	大 BLOB
TINYTEXT	65535 个成员	1 或 2 字节	非常小的文本串
TEXT	64 个成员	1、2、3、4 或 8 字节	小文本串
MEDIUMTEXT			中等文本串
LONGTEXT			大文本串
ENUM			枚举；列可赋予某个枚举成员
SET			集合；列可赋予多个集合成员

上表给出了 MySQL 定义串值列的类型，以及每种类型的最大尺寸和存储需求。对于可变长的列类型，各行的值所占的存储量是不同的，这取决于实际存放在列中的值的长度。这个长度在表中用 L 表示。

L 以外所需的额外字节为存放该值的长度所需的字节数。MySQL 通过存储值的内容及其长度来处理可变长度的值。这些额外的字节是无符号整数。请注意，可变长类型的最大长度、此类型所需的额外字节数以及占用相同字节数的无符号整数之间的对应关系。例如，MEDIUMBLOB 值可能最多 224 - 1 字节长并需要 3 个字节记录其结果。3 个字节的整数类型 MEDIUMINT 的最大无符号值为 224 - 1。这并非偶然。

### 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 或者 65535 之间的值。在检索的时候，CHAR 列删除了尾部的空格，而 VARCHAR 则保留了这些空格。下面通过示例来展示它们的区别。

```
mysql> create table testvar(
  -> fir varchar(4),
  -> sec char(4)
  -> );
Query OK, 0 rows affected (0.11 sec)
```

在创建好的表中插入数据

```
mysql> insert into testvar values('ab ','ab ');
Query OK, 1 row affected, 1 warning (0.05 sec)
```

查询字符串的长度

```
mysql> select length(fir),length(sec) from testvar;
+-----+-----+
| length(fir) | length(sec) |
+-----+-----+
|          4 |          2 |
+-----+-----+
1 row in set (0.00 sec)
```

为了更清晰地反应它们间的区别，给两个字段分别追加一个“=”字符。

```
mysql> select concat(fir,'=') as varchar_test,
  -> concat(sec,'=') as char_test from testvar;
+-----+-----+
| varchar_test | char_test |
+-----+-----+
| ab =       | ab=      |
+-----+-----+
1 row in set (0.00 sec)
```

BINARY 和 VARBINARY 类型

这两种类型包含的是二进制字符串，它们之间的区别与 CHAR 和 VARCHAR 类型类似。

```
mysql> create table tbinary(
  -> biy BINARY(3)
  -> );
Query OK, 0 rows affected (0.09 sec)
```

往表中插入一条记录

```
mysql> insert into tbinary set biy='c';
Query OK, 1 row affected (0.05 sec)
```

分别用以下几种模式来查看数据

```
mysql> select *,hex(biy) from tbinary;
+-----+-----+
| biy | hex(biy) |
+-----+-----+
| c   | 630000   |
+-----+-----+
1 row in set (0.00 sec)
```

ENUM 类型

ENUM 是一个字符串对象，其值来自表创建时在列规定中显式枚举的一系列值。



```
mysql> create table tenum(  
  -> gender enum('M','F')  
  -> );  
Query OK, 0 rows affected (0.11 sec)
```

插入几条记录;

```
mysql> insert into tenum values('M')  
  -> ,('1')  
  -> ,('f')  
  -> ,(NULL)  
  -> ;  
Query OK, 4 rows affected (0.03 sec)  
Records: 4 Duplicates: 0 Warnings: 0
```

查询结果

```
mysql> select * from tenum;  
+-----+  
| gender |  
+-----+  
| M      |  
| M      |  
| F      |  
| NULL   |  
+-----+  
4 rows in set (0.00 sec)
```

ENUM 类型是忽略大小写的, 插入时均转成了大写, 对于插入时不在指定范围内的值, 而是选择了枚举中位于第一的值, 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。

### 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。

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。

```
mysql> create table tset(  
  -> col set('a','b','c','d')  
  -> );  
Query OK, 0 rows affected (0.11 sec)
```

插入一些数据, 并显示出结果。

```
mysql> insert into tset values('a,b')
-> ,('a,d,a')
-> ,('a,b')
-> ,('a,c')
-> ,('a')
-> ;
Query OK, 5 rows affected (0.05 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> select * from tset;
+-----+
| 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。

## 第 六 章

字符串类

CHARSET(str) //返回字符串字符集

```
mysql> select charset('abc'),charset(convert('abc' using gbk));
+-----+-----+
| charset('abc') | charset(convert('abc' using gbk)) |
+-----+-----+
| utf8          | gbk                                |
+-----+-----+
1 row in set (0.00 sec)
```

CONCAT(str1,str2,...)

返回结果为连接参数产生的字符串。如有任何一个参数为 NULL，则返回值为 NULL。或许有一个或多个参数。

```
mysql> select concat('my','s','ql'),concat('my',NULL,'ql');
+-----+-----+
| concat('my','s','ql') | concat('my',NULL,'ql') |
+-----+-----+
| mysql                  | NULL                    |
+-----+-----+
1 row in set (0.00 sec)
```

INSTR (string ,substring ) //返回 substring 首次在 string 中出现的位置,不存在返回 0

```
mysql> select instr('hello my friend','o');
+-----+
| instr('hello my friend','o') |
+-----+
|                               5 |
+-----+
1 row in set (0.00 sec)
```

UCASE (string2 ) //转换成大写, LCASE (string2 ) //转换成小写

```
mysql> select lc case('HELLO,MY DEAR'),ucase('hello my friend');
+-----+-----+
| lc case('HELLO,MY DEAR') | uc case('hello my friend') |
+-----+-----+
| hello,my dear           | HELLO MY FRIEND           |
+-----+-----+
1 row in set (0.00 sec)
```

LEFT (string2 ,length ) //从 string2 中的左边起取 length 个字符, LENGTH (string ) //string 长度

```
mysql> select left('hello my friend',3),length('hello my friend');
+-----+-----+
| left('hello my friend',3) | length('hello my friend') |
+-----+-----+
| hel                       | 15                         |
+-----+-----+
1 row in set (0.00 sec)
```

LOAD\_FILE (file\_name ) //从文件读取内容

在 D 盘新建一个 TXT 文本文件, 文本的内容将会被查询出来。但用户必须具有读取 file 的权限。

```
mysql> select load_file('D:\mysql.txt');
+-----+
| load_file('D:\mysql.txt') |
+-----+
| 欢迎你访问                |
+-----+
1 row in set (0.00 sec)
```

LOCATE(substr,str) , LOCATE(substr,str,pos)

第一个语法返回字符串 str 中子字符串 substr 的第一个出现位置。第二个语法返回字符串 str 中子字符串 substr 的第一个出现位置, 起始位置在 pos。如若 substr 不在 str 中, 则返回值为 0。

```
mysql> select locate('e','hello my friend'), locate('e','hello my friend',5);
+-----+-----+
| locate('e','hello my friend') | locate('e','hello my friend',5) |
+-----+-----+
|                               2 |                               13 |
+-----+-----+
1 row in set (0.00 sec)
```

LPAD(str,len,padstr)返回字符串 str, 其左边由字符串 padstr 填补到 len 字符长度。假如 str 的长度大于 len, 则返回值被缩短至 len 字符。

```
mysql> select lpad('hi',4,'??'),lpad('hi',1,'??');
+-----+
| lpad('hi',4,'??') | lpad('hi',1,'??') |
+-----+
| ??hi              | h                  |
+-----+
1 row in set (0.05 sec)
```

LTRIM (string2) //去除前端空格

```
mysql> select '  hello,my friend',ltrim('  hello,my friend');
+-----+
| hello,my friend | ltrim('  hello,my friend') |
+-----+
|  hello,my friend | hello,my friend            |
+-----+
1 row in set (0.00 sec)
```

REPEAT (string2 ,count) //重复 count 次

```
mysql> select repeat('hello ',10);
+-----+
| repeat('hello ',10) |
+-----+
| hello hello hello hello hello hello hello hello hello hello |
+-----+
1 row in set (0.00 sec)
```

REPLACE (str ,search\_str ,replace\_str) //在 str 中用 replace\_str 替换 search\_str

```
mysql> select replace('hello ','l','=-');
+-----+
| replace('hello ','l','=-') |
+-----+
| he==o                      |
+-----+
1 row in set (0.00 sec)
```

RPAD (string2 ,length ,pad) //在 str 后用 pad 补充，直到长度为 length

```
mysql> select rpad('hello',10,'y');
+-----+
| rpad('hello',10,'y') |
+-----+
| helloyyyyy          |
+-----+
1 row in set (0.00 sec)
```

RTRIM (string2) //去除后端空格

```
mysql> select 'hello,my friend ',rtrim('hello,my friend ');
+-----+
| hello,my friend | rtrim('hello,my friend ') |
+-----+
| hello,my friend | hello,my friend           |
+-----+
1 row in set (0.00 sec)
```

STRCMP (string1 ,string2) //逐字符比较两字符串大小，字符串均相同，则返回 0，第一个参数小于第二个，则返回 -1，其它情况返回 1。

```
mysql> select strcmp('hi','hi'),strcmp('ab','cd'),strcmp('cd','ab');
+-----+-----+-----+
| strcmp('hi','hi') | strcmp('ab','cd') | strcmp('cd','ab') |
+-----+-----+-----+
| 0 | -1 | 1 |
+-----+-----+-----+
1 row in set (0.03 sec)
```

SUBSTRING(str, position [,length]) //从 str 的 position 开始,取 length 个字符,

```
mysql> select substring('my friend',2),substring('my friend',2,5);
+-----+-----+
| substring('my friend',2) | substring('my friend',2,5) |
+-----+-----+
| y friend | y fri |
+-----+-----+
1 row in set (0.00 sec)
```

TRIM([[[BOTH|LEADING|TRAILING] [padding] FROM]string2) 返回字符串 str, 其中所有 remstr 前缀和/或后缀都已被删除。若分类符 BOTH、LEADIN 或 TRAILING 中没有一个是给定的,则假设为 BOTH。remstr 为可选项,在未指定情况下,可删除空格。

```
mysql> select
-> trim(' hello '),
-> trim(leading 'x' from 'xxxhelloxxx'),
-> trim(both 'x' from 'xxxhelloxxx'),
-> trim(trailing 'xyz' from 'helloxxxxyz')
-> ;
```

trim 是去除两边的空格, leading 是将指定的字符从原字符串中从左边除掉, both 是将指定的字符从原字符串中从两边除掉, trailing 是将指定的字符从原字符串中从尾部除掉。

所以运行结果为:“hello”、“helloxxx”、“hello”、“helloxx”。

RIGHT(string2,length) //取 string2 最后 length 个字符

```
mysql> select right('hello my friend',5);
+-----+
| right('hello my friend',5) |
+-----+
| riend |
+-----+
1 row in set (0.00 sec)
```

SPACE(count) //生成 count 个空格

```
mysql> select concat('hello',space(10),'my dear friend');
+-----+
| concat('hello',space(10),'my dear friend') |
+-----+
| hello my dear friend |
+-----+
1 row in set (0.00 sec)
```

## 数学类

ABS (number2) 绝对值, BIN (decimal\_number) 十进制转二进制; HEX (DecimalNumber) 转十六进制

```
mysql> select abs(-3),bin(15),hex(15);
+-----+-----+
| abs(-3) | bin(15) | hex(15) |
+-----+-----+
|      3 | 1111    | F       |
+-----+-----+
1 row in set (0.00 sec)
```

CEILING (number2) //向上取整, ROUND (number [,decimals]) //四舍五入,decimals 为小数位数  
 , FLOOR (number2) //向下取整。

```
mysql> select ceiling(3.4),round(3.5),floor(3.5);
+-----+-----+-----+
| ceiling(3.4) | round(3.5) | floor(3.5) |
+-----+-----+-----+
|           4 |           4 |           3 |
+-----+-----+-----+
1 row in set (0.05 sec)
```

MOD (numerator ,denominator) //求余, SQRT(number2) //开平方, POWER (number ,power) //求指数

```
mysql> select mod(13,4),sqrt(16),power(2,4);
+-----+-----+-----+
| mod(13,4) | sqrt(16) | power(2,4) |
+-----+-----+-----+
|          1 |          4 |          16 |
+-----+-----+-----+
1 row in set (0.03 sec)
```

LEAST (number , number2 [...]) //求最小值, SIGN (number2) //返回符号,正负或 0

```
mysql> select least(23,32,14),sign(5),sign(-5),sign(0);
+-----+-----+-----+-----+
| least(23,32,14) | sign(5) | sign(-5) | sign(0) |
+-----+-----+-----+-----+
|           14 |        1 |        -1 |         0 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

FORMAT(X,D) //保留小数位数, 将数字 X 的格式写为 '#,###,###.##',以四舍五入的方式保留小数点后 D 位, 并将结果以字符串的形式返回。若 D 为 0, 则返回结果不带有小数点, 或不含小数部分。

```
mysql> select format(123332.123456,4),format(1203.1,4),format(123.2,0);
+-----+-----+-----+
| format(123332.123456,4) | format(1203.1,4) | format(123.2,0) |
+-----+-----+-----+
| 123,332.1235           | 1,203.1000       | 123              |
+-----+-----+-----+
1 row in set (0.00 sec)
```

RAND() RAND(N) //返回一个随机浮点值 v , 范围在 0 到 1 之间 (即, 其范围为  $0 \leq v \leq 1.0$ )。若已指定一个整数参数 N , 则它被用作种子值, 用来产生重复序列。

```
mysql> select rand(),rand(20),floor(0+(rand()*10));
+-----+-----+-----+
| rand() | rand(20) | floor(0+(rand()*10)) |
+-----+-----+-----+
| 0.55745815910162 | 0.15888261251047 | 5                    |
+-----+-----+-----+
1 row in set (0.00 sec)
```

## 日期时间类

ADDTIME (date2 ,time\_interval ) //将 time\_interval 加到 date2

CONVERT\_TZ (datetime2 ,fromTZ ,toTZ ) //转换时区

CURRENT\_DATE ( ) //当前日期

CURRENT\_TIME ( ) //当前时间

CURRENT\_TIMESTAMP ( ) //当前时间戳

DATE (datetime ) //返回 datetime 的日期部分

DATE\_ADD (date2 , INTERVAL d\_value d\_type ) //在 date2 中加上日期或时间

DATE\_FORMAT (datetime ,FormatCodes ) //使用 formatcodes 格式显示 datetime

DATE\_SUB (date2 , INTERVAL d\_value d\_type ) //在 date2 上减去一个时间

DATEDIFF (date1 ,date2 ) //两个日期差

DAY (date ) //返回日期的天

DAYNAME (date ) //英文星期

```
mysql> select DAY<'2008-08-08'> , DAYNAME<'2008-08-08'>;
+-----+-----+
| DAY<'2008-08-08'> | DAYNAME<'2008-08-08'> |
+-----+-----+
| 8 | Friday |
+-----+-----+
1 row in set (0.00 sec)
```

DAYOFWEEK (date ) //星期(1-7) ,1 为星期天

DAYOFYEAR (date ) //一年中的第几天, 返回 date 在一年中的日数, 在 1 到 366 范围内。

```
mysql> select DAYOFYEAR<'2008-08-08'> , DAYOFYEAR<NOW(>>;
+-----+-----+
| DAYOFYEAR<'2008-08-08'> | DAYOFYEAR<NOW(>> |
+-----+-----+
| 221 | 160 |
+-----+-----+
1 row in set (0.00 sec)
```

EXTRACT (interval\_name FROM date ) //从 date 中提取日期的指定部分

MAKEDATE (year ,day ) //给出年及年中的第几天,生成日期串

MAKETIME (hour ,minute ,second ) //生成时间串

NOW ( ) //当前时间

SEC\_TO\_TIME (seconds ) //秒数转成时间

STR\_TO\_DATE (string ,format ) //字串转成时间,以 format 格式显示

TIMEDIFF (datetime1 ,datetime2 ) //两个时间差

TIME\_TO\_SEC (time ) //时间转秒数]

WEEK (date\_time [,start\_of\_week ]) //第几周

WEEKDAY(date)

返回 date 的星期索引(0=星期一, 1=星期二, .....6= 星期天)。



```
mysql> select WEEKDAY('2008-08-08 22:23:00'),WEEKDAY('2008-08-08');
+-----+-----+
| WEEKDAY('2008-08-08 22:23:00') | WEEKDAY('2008-08-08') |
+-----+-----+
| 4 | 4 |
+-----+-----+
1 row in set (0.00 sec)
```

YEAR(datetime) //年份, 返回 date 的年份, 范围在 1000 到 9999。

```
mysql> select YEAR('2008-08-08');
+-----+
| YEAR('2008-08-08') |
+-----+
| 2008 |
+-----+
1 row in set (0.00 sec)
```

DAYOFWEEK(date) 返回日期 date 的星期索引(1=星期天, 2=星期一, ……7=星期六)。

DAYOFMONTH(datetime) //月的第几天, 返回 date 的月份中日期, 在 1 到 31 范围内。

```
mysql> select DAYOFWEEK('2008-08-08'),DAYOFMONTH('2008-08-08');
+-----+-----+
| DAYOFWEEK('2008-08-08') | DAYOFMONTH('2008-08-08') |
+-----+-----+
| 6 | 8 |
+-----+-----+
1 row in set (0.06 sec)
```

这个周的第六天（即星期五），这个月的第 8 天。

HOURL(datetime) //返回 time 的小时, 范围是 0 到 23,

MINUTE(datetime) //返回 time 的分钟, 范围是 0 到 59。

SECOND(datetime) //返回 time 的秒数, 范围是 0 到 59。

```
mysql> select HOUR('10:20:30'),MINUTE('10:20:30'),SECOND('10:20:30');
+-----+-----+-----+
| HOUR('10:20:30') | MINUTE('10:20:30') | SECOND('10:20:30') |
+-----+-----+-----+
| 10 | 20 | 30 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

LAST\_DAY(date) //date 的月的最后日期

MICROSECOND(datetime) //微秒

MONTH(datetime) //返回 date 的月份, 范围 1 到 12。MONTHNAME(date) 返回 date 的月份名字。

```
mysql> select MONTH('2008-08-08'),MONTHNAME('2008-08-08');
+-----+-----+
| MONTH('2008-08-08') | MONTHNAME('2008-08-08') |
+-----+-----+
| 8 | August |
+-----+-----+
1 row in set (0.00 sec)
```

QUARTER(date)

返回 date 一年中的季度, 范围 1 到 4。

```
mysql> select quarter('2008-08-08');
+-----+
| quarter('2008-08-08') |
+-----+
| 3 |
+-----+
1 row in set (0.00 sec)
```

WEEK(date)

WEEK(date,first)

对于星期天是一周的第一天的地方，有一个单个参数，返回 date 的周数，范围在 0 到 52。2 个参数形式 WEEK() 允许你指定星期是否开始于星期天或星期一。如果第二个参数是 0，星期从星期天开始，如果第二个参数是 1，从星期一开始。

```
mysql> select week('1998-02-20',1), week('1998-02-20',0);
+-----+-----+
| week('1998-02-20',1) | week('1998-02-20',0) |
+-----+-----+
| 8 | 7 |
+-----+-----+
1 row in set (0.00 sec)
```

PERIOD\_ADD(P,N)

增加 N 个月到阶段 P (以格式 YYMM 或 YYYYMM)。以格式 YYYYMM 返回值。注意阶段参数 P 不是日期值。

PERIOD\_DIFF(P1,P2)

返回在时期 P1 和 P2 之间月数，P1 和 P2 应该以格式 YYMM 或 YYYYMM。注意，时期参数 P1 和 P2 不是日期值。

```
mysql> select period_add(9801,2),period_diff(9802,9703);
+-----+-----+
| period_add(9801,2) | period_diff(9802,9703) |
+-----+-----+
| 199803 | 11 |
+-----+-----+
1 row in set (0.00 sec)
```

DATE\_ADD(date,INTERVAL expr type)

DATE\_SUB(date,INTERVAL expr type)

```
mysql> select date_add('2008-06-08',interval 31 day),
-> date_sub('2008-06-08',interval 31 day);
```

执行的结果为

2008-07-09 , 2008-05-08

ADDDATE(date,INTERVAL expr type), SUBDATE(date,INTERVAL expr type)

ADDDATE()和 SUBDATE()是 DATE\_ADD()和 DATE\_SUB()的同义词。

```
mysql> select adddate('2008-06-08',31),subdate('2008-06-08',31);
+-----+-----+
| adddate('2008-06-08',31) | subdate('2008-06-08',31) |
+-----+-----+
| 2008-07-09 | 2008-05-08 |
+-----+-----+
1 row in set (0.00 sec)
```

Interval 用于表示时间尺度。

```
mysql> select '2008-08-08 23:59:59'+interval 1 second as add_second,
-> '2008-08-08 23:59:59'-interval 1 second as sub_second;
+-----+-----+
| add_second          | sub_second          |
+-----+-----+
| 2008-08-09 00:00:00 | 2008-08-08 23:59:58 |
+-----+-----+
1 row in set (0.02 sec)
```

EXTRACT()函数所使用的时间间隔类型说明符同 DATE\_ADD()或 DATE\_SUB()的相同,但它从日期中提取其部分,而不是执行日期运算。

```
mysql> select extract<year from '2008-08-08'>,
-> extract<year_month from '2008-08-08 01:02:03'>,
-> extract<day_minute from '2008-08-08 01:02:03'>;
```

结果为: 2008, 200808, 80102

TO\_DAYS(date) 给出一个日期 date, 返回一个天数(从 0 年的天数)。TO\_DAYS() 不用于阳历出现(1582)前的值,原因是当日历改变时,遗失的日期不会被考虑在内。

```
mysql> select to_days(20080808);
+-----+
| to_days(20080808) |
+-----+
|          733627   |
+-----+
1 row in set (0.00 sec)
```

FROM\_DAYS(N) 给出一个天数 N, 返回一个 DATE 值。

```
mysql> select from_days(733627);
+-----+
| from_days(733627) |
+-----+
| 2008-08-08        |
+-----+
1 row in set (0.00 sec)
```

DATE\_FORMAT(date,format) 根据 format 字符串格式化 date 值。下列修饰符可以被用在 format 字符串中:

代号	说明	代号	说明
%a	工作日的缩写名称 (Sun..Sat)	%i	分钟, 数字形式 (00..59)
%b	月份的缩写名称 (Jan..Dec)	%j	一年中的天数 (001..366)
%c	月份, 数字形式(0..12)	%k	小时 (0..23)
%D	带有英语后缀的该月日期 (0th, 1st, 2nd, 3rd, ...)	%l	小时 (1..12)
%d	该月日期, 数字形式 (00..31)	%M	月份名称 (January..December)
%e	该月日期, 数字形式(0..31)	%m	月份, 数字形式 (00..12)
%f	微秒 (000000..999999)	%p	上午 (AM) 或下午 (PM)
%H	小时(00..23)	%r	时间, 12 小时制 (小时 hh:分钟 mm:秒数 ss 后加 AM 或 PM)

%h	小时 (01..12)	%S	秒 (00..59)
%I	小时 (01..12)	%s	秒 (00..59)
%T	时间 , 24 小时制 (小时 hh:分钟 mm:秒数 ss)	%w	一周中的每日 (0=周日..6=周六)
%U	周 (00..53), 其中周日为每周的第一天	%X	该周的年份, 其中周日为每周的第一天, 数字形式, 4 位数;和%V 同时使用
%u	周 (00..53), 其中周一为每周的第一天	%x	该周的年份, 其中周一为每周的第一天, 数字形式, 4 位数;和%v 同时使用
%V	周 (01..53), 其中周日为每周的第一天 ; 和 %X 同时使用	%Y	年份, 数字形式, 4 位数
%v	周 (01..53), 其中周一为每周的第一天 ; 和 %x 同时使用	%y	年份, 数字形式 (2 位数)
%W	工作日名称 (周日..周六)	%%	'%' 文字字符

所有的其他字符不做解释被复制到结果中。

```
mysql> SELECT DATE_FORMAT('2008-08-08 22:23:00', '%W %M %Y');
+-----+
| DATE_FORMAT('2008-08-08 22:23:00', '%W %M %Y') |
+-----+
| Friday August 2008                               |
+-----+
1 row in set (0.00 sec)
```

CURDATE()与 CURRENT\_DATE 。以'YYYY-MM-DD'或 YYYYMMDD 格式返回今天日期值, 取决于函数是在一个字符串还是数字上下文被使用。

```
mysql> select curdate(),current_date(),current_time(),now();
+-----+-----+-----+-----+
| curdate() | current_date() | current_time() | now() |
+-----+-----+-----+-----+
| 2008-06-08 | 2008-06-08      | 23:08:56       | 2008-06-08 23:08:56 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

SEC\_TO\_TIME(seconds)

返回 seconds 参数, 变换成小时、分钟和秒, 值以'HH:MM:SS'或 HHMMSS 格式化, 取决于函数是在一个字符串还是在数字上下文中被使用。

TIME\_TO\_SEC(time) 返回 time 参数, 转换成秒。

```
mysql> select time_to_sec('22:23:58'),sec_to_time(80638);
+-----+-----+
| time_to_sec('22:23:58') | sec_to_time(80638) |
+-----+-----+
| 80638 | 22:23:58          |
+-----+-----+
1 row in set (0.00 sec)
```

## 第七章

### MYSQL 表达式与运算符

MySQL 允许编写包括常量、函数调用和表列引用的表达式。这些值可利用不同类型的运算符进行组合, 诸如算术运算符或比较运算符。表达式的项可用圆括号来分组。表达式在 **SELECT** 语句的列选择列表和 **WHERE** 子句中出现得最为频繁, 如下所示

```
mysql> select
  -> concat(name,',',intro)
  -> from student
  -> where
  -> score>65 and intro IS NOT NULL;
```

所选择的列给出了一个表达式, 如 **WHERE** 子句中所示的那样。表达式也出现在 **DELETE** 和 **UPDATE** 语句的 **WHERE** 子句中, 以及出现在 **INSERT** 语句的 **VALUES()** 子句中。

在 MySQL 遇到一个表达式时, 它对其求值得出结果。例如,  $(4 * 3) / (4 - 2)$  求值得 6。表达式求值可能涉及类型转换。例如, MySQL 在数 960821 用于需要日期值的环境时, 将其转换为日期 “1996-08-21”。

```
mysql> SELECT DATE(960821);
+-----+
| DATE(960821) |
+-----+
| 1996-08-21   |
+-----+
1 row in set (0.03 sec)
```

本章讨论怎样编写 MySQL 的表达式, 以及在表达式求值中 MySQL 所使用的类型转换规则。每个 MySQL 的运算符都介绍过了, 但 MySQL 有那么多函数, 我们只接触过几个。每个运算符和函数的进一步介绍可参阅附录 C。

#### 撰写表达式

表达式可以只是一个简单的常量, 如数值常量 0 和字符串常量 ‘abc’。

```
mysql> SELECT 1, 'ABC';
+---+-----+
| 1 | ABC |
+---+-----+
| 1 | ABC |
+---+-----+
1 row in set (0.00 sec)
```

表达式可以进行函数调用。有的函数需要参数 (圆括号中有值), 而有的不需要。多个参数应该用逗号分隔。在调用一个函数时, 参数旁边可以有空格, 但在函数名与圆括号间不能有空格。下面是一些函数例子:

**NOW()** 无参数函数

**STRCMP**(“abc”, “def”) 有两个参数的函数

**STRCMP**(“abc”, “def”) 参数旁边有空格是合法的

**STRCMP** (“abc”, “def”) 函数名后跟空格是不合法的

如果函数名后有一个空格, MySQL 的分析程序可能会将函数名解释为一个列名 (函数名不是保留字, 如果需要的话, 可将它们用作列名)。其结果是出现一个语法错误。表达式中可使用表列。最简单

的情形是, 当某个列所属的表在上下文中是明确的, 则可简单地给出列名对该列进行引用。下面的每个 SELECT 语句中惟一地出了一个表名, 因此, 列的引用无歧义:

```
SELECT * FROM CLASS;  
SELECT * FROM EMPLOYEE;
```

如果使用哪个表的列不明确, 可在列名前加上表名。如果使用哪个数据库中的表也不明确的话, 可在表名前加上数据库名。如果只是希望意思更明显, 也可以在无歧义的上下文中利用这种更为具体的表示形式, 如:

```
SELECT E.NAME,C.NAME  
FROM EMPLOYEE E,CLASS C  
WHERE E.DID=C.CLASSID;
```

总之, 可以组合所有这些值以得到更为复杂的表达式。

MySQL 有几种类型的运算符可用来连接表达式的项。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。

### 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。

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

下面通过示例来简单展示几种运算符的使用方法:

```
mysql> select 517+518,518-517,517*518,517/518,517%518;  
+-----+  
| 517+518 | 518-517 | 517*518 | 517/518 | 517%518 |  
+-----+  
|    1035 |        1 | 267806 | 0.9981 |    517 |  
+-----+  
1 row in set (0.00 sec)
```

如果除数为 0, 将是非法除数, 返回结果为 NULL。

```
mysql> select 517/0;  
+-----+  
| 517/0 |  
+-----+  
|  NULL |  
+-----+  
1 row in set (0.00 sec)
```

取模的时候, % 和 MOD 的功能是一样的

```
mysql> select 517%518, mod(517,518);  
+-----+  
| 517%518 | mod(517,518) |  
+-----+  
|    517 |           517 |  
+-----+  
1 row in set (0.00 sec)
```

### 逻辑运算符

逻辑运算符表

运算符	语法	说明
AND,&&	a AND b, a && b	逻辑与; 如果两操作数为真, 结果为真
OR,	a OR, a    b	逻辑与; 任一操作数为真, 结果为真
NOT,!	NOT a,! a	逻辑与; 如果两操作数为假, 结果为真

逻辑运算符对表达式进行估计以确定其为真（非零）或假（零）。MySQL 包含有 C 风格的“&&”、“||”和“!”运算符，可替换 AND、OR 和 NOT。要特别注意“||”运算符，ANSI SQL 指定“||”作为串连接符，但在 MySQL 中，它表示一个逻辑或运算。如果执行下面的查询，则返回数 0:

```
mysql> SELECT "ABC"||"DEF";
+-----+
| "ABC"||"DEF" |
+-----+
|                0 |
+-----+
1 row in set, 2 warnings (0.00 sec)
```

MySQL 为进行运算，将“abc”和“def”转换为整数，且两者都转换为 0，0 与 0 进行或运算，结果为 0。在 MySQL 中，必须用 CONCAT (“abc”，“def”) 来完成串的连接。

位运算符表

运算符	语法	说明
&	a & b	按位与; 如果两个操作的对应位为 1, 则该结果为 1
	a   b	按位或; 如果两个操作的对应位中有一位为 1, 则该结果为 1
<<	a<<b	将 a 左移 b 个二进制位
>>	a>>b	将 a 右移 b 个二进制位

比较运算符如表

其中包括测试相对大小或数和串的顺序的运算符，以及完成模式匹配和测试 NULL 值的运算符。“<=>”运算符是 MySQL 特有的。

比较运算符表

运算符	语法	说明
=	a = b	如果两个操作数相等, 为真
!=,<>	a!=b,a<>b	如果两个操作数不等, 为真
<	a<b	如果 a 小于等于 b, 为真
<=	a<=b	如果 a 小于等于 b, 为真
>=	a>= b	如果 a 大于等于 b, 为真
>	a  b	如果 a 大于 b, 为真
IN	a IN(b1,b2...)	如果 a 为 b1,b2 中的任意一个, 为真
BETWEEN	a BETWEEN b AND c	将 a 的值在 b 和 c 之间包括 b 和 c, 为真
LIKE	a LIKE b	SQL 模式匹配: 如果 a 与 b 匹配, 为真
NOT LIKE	a NOT LIKE b	SQL 模式匹配: 如果 a 与 b 不匹配, 为真
REGEXP	a REGEXP b	正则表达式匹配: 如果 a 与 b 匹配, 为真
NOT REGEXP	a NOT REGEXP b	正则表达式匹配: 如果 a 与 b 不匹配, 为真
<=>	a<=>b	如果操作数相同 (即使为 NULL), 为真
IS NULL	a is null	如果操作数为 NULL, 为真
IS NOT NULL	a is not null	如果操作数不为 NULL, 为真



要注意 **BINARY** 运行符, 此运算符可用来将一个串转换为一个二进制串, 这个串在比较中是区分大小写的。它没有相应的 **NOT BINARY** 计算。如果希望在不区分大小写的环境中使用区分大小写的功能, 则使用 **BINARY**。对于利用二进制串类型 (**CHAR BINARY**、**VARCHAR BINARY** 和 **BLOB** 类型) 定义的列, 其比较总是区分大小写的。

```
mysql> SELECT 'ABC'=BINARY<'ABC'>,
-> 'abc'=BINARY<'Abc'>,
-> BINARY<'Abc'>='abc',
-> ;
```

'ABC'=BINARY<'ABC'>	'abc'=BINARY<'Abc'>	BINARY<'Abc'>='abc'
1	0	0

1 row in set (0.00 sec)

为了对这样的列类型实现不区分大小写的比较, 可利用 **UPPER()** 或 **LOWER()** 来转换成相同的大小写。

通配符 “%” 与任何串匹配, 其中包括与空字符序列匹配, 但是, “%” 不与 **NULL** 匹配。事实上, 具有 **NULL** 操作数的任何模式匹配都将失败。MySQL 的 **LIKE** 运算符是不区分大小写的, 除非它至少有一个操作数是二进制串。

```
mysql> SELECT 'ABC' LIKE 'abc%',
-> 'ABC' LIKE NULL,
-> BINARY 'ABC' LIKE 'abc%';
```

'ABC' LIKE 'abc%'	'ABC' LIKE NULL	BINARY 'ABC' LIKE 'abc%'
1	NULL	0

1 row in set (0.00 sec)

**LIKE** 所允许的另一个通配符是 “\_”, 它与单个字符匹配。“\_ \_ \_” 与三个字符的串匹配。“c \_ t” 与 “cat”、“cut” 甚至 “c \_ t” 匹配 (因为 “\_” 与自身匹配)。为了关掉 “%” 或 “\_” 的特殊含义, 与这些字符的直接实例相匹配, 需要在它们前面放置一个斜杠 (“%” 或 “\_”)。

```
mysql> SELECT 'ABC' LIKE 'ab%c',
-> 'ABC' LIKE 'a\%c',
-> 'A% C' LIKE 'a\%c',
-> ;
```

'ABC' LIKE 'ab%c'	'ABC' LIKE 'a\%c'	'A% C' LIKE 'a\%c'
1	0	1

1 row in set (0.00 sec)

MySQL 的另一种形式的模式匹配使用了正则表达式。运算符为 **REGEXP** 而不是 **LIKE** (**RLIKE** 为 **REGEXP** 的同义词)。最常用的正则表达式模式字符如下: ‘.’ 与任意单个字符匹配:

```
mysql> SELECT "ABC" REGEXP "A.C";
```

"ABC" REGEXP "A.C"
1

1 row in set (0.05 sec)

‘[...]’ 与方括号中任意字符匹配。可列出由短划线 ‘-’ 分隔的范围端点指定一个字符范围。

为了否定这种区间的意义（即与未列出的任何字符匹配），指定 ‘^’ 作为该区间的第一个字符即可。

```
mysql> SELECT "ABC" REGEXP "[A-Z]",
-> "ABC" REGEXP "[^A-Z]";
+-----+-----+
| "ABC" REGEXP "[A-Z]" | "ABC" REGEXP "[^A-Z]" |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)
```

上面的语句就是说[A-Z]中有一个字符被“ABC”包含即为真。

‘\*’ 表示“与其前面字符的任意数目的字符匹配”，因此，如 ‘x\*’ 与任意数目的 ‘x’ 字符匹配，例如：

```
mysql> SELECT "ABCDEF" REGEXP "A.*F",
-> "ABC" REGEXP "[0-9]*ABC",
-> "ABC" REGEXP "[0-9][0-9]*";
```

执行结果为 1, 1, 0;

“任意数目”包括 0 个实例，这也就是为什么第二个表达式匹配成功的原因。‘^pat’ 和 ‘pat\$’ 固定了一种模式匹配，从而模式 pat 只在它出现在串的前头时匹配，而 ‘^pat\$’ 只在 pat 匹配整个串时匹配，例如：

```
mysql> SELECT "ABC" REGEXP "b",
-> "ABC" REGEXP "^b",
-> "ABC" REGEXP "b$",
-> "ABC" REGEXP "^abc$",
-> "ABCD" REGEXP "^abc$";
```

执行结果为 1, 0, 0, 1, 0; (这里的^是表示从哪儿开始匹配的意思)

REGEXP 模式可从某个表列中取出，虽然如果该列包含几个不同的值时，这样做比常量模式慢。每当列值更改时，必须对模式进行检查并转换成内部形式。MySQL 的正则表达式匹配还有一些特殊的模式字符。要了解更详细信息请参阅帮助文档。

### 运算符的优先级

当求一个表达式的值时，首先查看运算符以决定运算的先后次序。有的运算符具有较高的优先级；例如，乘和除比加和减的优先级更高。下面的两个表达式是等价的，因为 “\*” 和 “/” 先于 “+” 和 “-” 计算：

```
mysql> SELECT 1+2*3-4/5,1+6-0.8;
+-----+-----+
| 1+2*3-4/5 | 1+6-0.8 |
+-----+-----+
| 6.2000 | 6.2 |
+-----+-----+
1 row in set (0.00 sec)
```

可用圆括号来忽略运算符的优先级并改变表达式的求值顺序，如：

```
mysql> SELECT 1+2*3-4/5,(1+2)*(3-4)/5;
+-----+-----+
| 1+2*3-4/5 | (1+2)*(3-4)/5 |
+-----+-----+
| 6.2000 | -0.6000 |
+-----+-----+
1 row in set (0.00 sec)
```

### 表达式中的 NULL 值

请注意, 在表达式中使用 NULL 值时, 其结果有可能出现意外。下列准则将有助于避免出问题。如果将 NULL 作为算术运算或位运算符的一个操作数, 其结果为 NULL, 如果将 NULL 用于逻辑运算符, NULL 被认为是假。

```
mysql> SELECT 1+NULL, 1 AND NULL, 1 OR NULL;
+-----+-----+-----+
| 1+NULL | 1 AND NULL | 1 OR NULL |
+-----+-----+-----+
| NULL   | NULL      | 1         |
+-----+-----+-----+
1 row in set (0.00 sec)
```

NULL 作为任意比较运算符的操作数, 除 <=>、IS NULL 和 IS NOT NULL 运算符(它们是专门扩展来处理 NULL 值的)外, 将产生一个 NULL 结果。如:

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

如果给函数一个 NULL 参数, 除了那些处理 NULL 参数的函数外, 一般返回一个 NULL 结果。例如, IFNULL() 能够处理 NULL 参数并适当地返回真或假。在升序排序中, NULL 将排在所有非 NULL 值之前(包括空串), 而在降序排序中, NULL 将排在所有非 NULL 值之后。

```
mysql> SELECT IFNULL(1,0), IFNULL(1/0,1);
+-----+-----+
| IFNULL(1,0) | IFNULL(1/0,1) |
+-----+-----+
| 1           | 1.0000        |
+-----+-----+
1 row in set (0.00 sec)
```

### 类型转换

MySQL 根据所执行的操作类型, 自动地进行大量的类型转换, 任何时候, 只要将一个类型的值用于需要另一类型值的场合, 就会进行这种转换。下面是需要进行类型转换的原因:

- 操作数转换为适合于某种运算符求值的类型。

- 函数参数转换为函数所需的类型。

- 转换某个值以便赋给一个具有不同类型的表列。下列表达式涉及类型转换。它由加运算符“+”和两个操作数 1 和“2”组成:

1+"2"

其中操作数的类型不同, 一个是数, 另一个是串, 因此, MySQL 对其中之一进行转换以便使它们两个具有相同的类型。但是应该转换哪一个呢? 因为, “+”是一个数值运算符, 所以 MySQL 希望操作数为数, 因此, 将串“2”转换为数 2。然后求此表达式的值得出 3。再举一例。CONCAT() 函数连接串产生一个更长的串作为结果。为了完成此工作, 它将参数解释为串, 而不管参数实际是何类型。如果传递给 CONCAT() 几个数, 则它将把它们转换成串, 然后返回这些串的连接, 如:

```
mysql> SELECT 1+"2",CONCAT(1,2,3),DATE(20080808);
+-----+-----+-----+
| 1+"2" | CONCAT(1,2,3) | DATE(20080808) |
+-----+-----+-----+
|      3 | 123           | 2008-08-08     |
+-----+-----+-----+
1 row in set (0.00 sec)
```

某些运算符可将操作数强制转换为它们所要的类型，而不管操作数是什么类型。例如，算术运算符需要数，并按此对操作数进行转换，参考如下运算：

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

MySQL 不对整个串进行寻找一个数的查找；它只查看串的起始处。如果一个串不以数作为前导部分，其转换结果为 0。

```
mysql> SELECT 0+"2008-6-8","12:14:01"+0,"23-skil"+0,"xmh-18"+0;
+-----+-----+-----+-----+
| 0+"2008-6-8" | "12:14:01"+0 | "23-skil"+0 | "xmh-18"+0 |
+-----+-----+-----+-----+
|          2008 |          12 |          23 |           0 |
+-----+-----+-----+-----+
1 row in set, 4 warnings (0.00 sec)
```

逻辑和位运算符比算术运算符要求更为严格。它们要求操作数都为数，否则各操作数不被认为是真。

```
mysql> SELECT 0.3 OR 'XZ',1.3 OR .04,0.3 AND 'XZZ','XZZ' AND .04;
+-----+-----+-----+-----+
| 0.3 OR 'XZ' | 1.3 OR .04 | 0.3 AND 'XZZ' | 'XZZ' AND .04 |
+-----+-----+-----+-----+
|           1 |           1 |           0 |           0 |
+-----+-----+-----+-----+
1 row in set, 2 warnings (0.00 sec)
```

模式匹配运算符要求对串进行处理。这表示可将 MySQL 的模式匹配运算符用于数，因为 MySQL 会在试图进行的匹配中将它们转换成串。例如：

```
mysql> SELECT 12345 LIKE "1%",12345 REGEXP "1.*5";
+-----+-----+
| 12345 LIKE "1%" | 12345 REGEXP "1.*5" |
+-----+-----+
|           1 |           1 |
+-----+-----+
1 row in set (0.00 sec)
```

大小比较运算符（“<”、“<=”、“=” 等等）是上下文相关的；即，它们根据操作数的类型求值。如果两个操作数都是串，则按串进行字典顺序的比较；如果两个操作数都为整数，则按整数进行数的比较。

```
mysql> SELECT "2"<"1",2<"11","2"<11,2<11;
+-----+-----+-----+-----+
| "2"<"1" | 2<"11" | "2"<11 | 2<11 |
+-----+-----+-----+-----+
|      0 |      1 |      1 |      1 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

考虑以下执行语句后的结果为什么会是这样?

```
mysql> SELECT * FROM STUDENT WHERE NAME=00;
+-----+-----+-----+-----+
| id | name | intro | score |
+-----+-----+-----+-----+
| 1 | xmh | good study | 65 |
| 3 | zah | dep | 75 |
| 2 | zyj | nice | 70 |
| 2 | zqy | nice | 70 |
| 5 | zxx | NULL | 85 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

原来的打算大概是想查询姓名为包含值“00”的那行。但实际作用却是查询了所有的行。之所以这样是由于 MySQL 的比较规则在起作用。name 为一个串列, 但 00 没有用引号括起来, 因此, 它被作为数对待了。按 MySQL 的比较规则, 涉及一个串与一个数的比较按两个数的比较来求值。随着查询的执行, name 的每个值被转换为 0, “00” 也被转换为 0, 因此, 所有不类似数的串都转换成 0。从而, 对于每一行, WHERE 子句都为真, 因此, 查询出了所有的行, 试想如果换成是 DELETE 或 UPDATE 语句, 结果将如何呢?

如果不能肯定某个值的使用方式, 可以利用 MySQL 的表达式求值机制将该值强制转换为特定的类型:

- 操作数转换为适合于某种运算符求值的类型。
- 利用 CONCAT() 将值转换为串
- 利用 ASCII() 得到字符的 ASCII 值
- 利用 DATE() 强制转换串或数为日期

```
mysql> SELECT 0x65, 0x65+0, CONCAT(14), ASCII('A'), DATE(20080808);
+-----+-----+-----+-----+-----+
| 0x65 | 0x65+0 | CONCAT(14) | ASCII('A') | DATE(20080808) |
+-----+-----+-----+-----+-----+
| e | 101 | 14 | 65 | 2008-08-08 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

### 3. 超范围值或非法值的转换

超范围值或非法值的转换的基本原则为: 无用输入, 无用输出。如果不在存储日期前对其进行验证, 那么可能会得到不喜欢的东西。下面给出一些 MySQL 处理超范围值或不合适值的一般原则, 这些内容曾经在前面介绍过:

■ 对于数值或 TIME 列, 超出合法范围的值被剪裁为相应取值范围的最接近的数值并作为结果值存储。

■ 对于非 ENUM 或 SET 的串列, 太长的串被截为适合该列存储的最大长度的串。ENUM 或 SET 列的赋值依赖于定义列时给出的合法值。如果赋予 ENUM 列一个未作为枚举成员给出的值, 将会赋予一个错误成员 (即, 对应于零值成员的空串)。如果赋予 SET 列一个包含未作为集合成员给出的子串的值, 那么, 那些未作为集合成员给出的子串将被删除, 并将剩余成员构成的值赋给该列。

■ 对于日期或时间列, 非法值被转换为该类型适当的“零”值 (参阅表 2-11)。对于非 TIME 的日期和时间列, 超出取值范围的值可转换为“零”值、NULL 或某种其他的值 (换句话说, 结果是不可预料的)。

## 第八章

### MYSQL 表类型与注释及命名规范

设计数据库管理系统由很多折衷。一些任务必须用 `transaction-safe` 的方法完成, 但是这样增加了时间, 磁盘, 内存的开销。

表类型又成为 `storage engines`. 它揭示了一些表类型依靠大量单独的源代码来管理自己的 `caching, indexing, locking, and disk access`. 也揭示数据库的根本目标: 存储数据。

#### ISAM 表

**ISAM** 是一个定义明确且历经时间考验的数据表格管理方法, 它在设计之时就考虑到数据库被查询的次数要远大于更新的次数。因此, **ISAM** 执行读取操作的速度很快, 而且不占用大量的内存和存储资源。**ISAM** 的两个主要不足之处在于, 它不支持事务处理, 也不能够容错: 如果你的硬盘崩溃了, 那么数据文件就无法恢复了。如果你正在把 **ISAM** 用在关键任务应用程序里, 那就必须经常备份你所有的实时数据, 通过其复制特性, **MySQL** 能够支持这样的备份应用程序。

```
create table asset
(
  assetID int not null,
  description varchar(255)
) type=ISAM;
```

5.0 已经不支持这个类型。

#### MYISAM 表

**MyISAM** 是 **MySQL** 的 **ISAM** 扩展格式和缺省的数据库引擎。除了提供 **ISAM** 里所没有的索引和字段管理的大量功能, **MyISAM** 还使用一种表格锁定的机制, 来优化多个并发的读写操作。其代价是你需要经常运行 `OPTIMIZE TABLE` 命令, 来恢复被更新机制所浪费的空间。**MyISAM** 还有一些有用的扩展, 例如用来修复数据库文件的 **MyISAMChk** 工具和用来恢复浪费空间的 **MyISAMPack** 工具。

**MyISAM** 强调了快速读取操作, 这可能就是为什么 **MySQL** 受到了 **Web** 开发如此青睐的主要原因: 在 **Web** 开发中你所进行的大量数据操作都是读取操作。所以, 大多数虚拟主机提供商和 **Internet** 平台提供商(**Internet Presence Provider, IPP**)只允许使用 **MyISAM** 格式。

```
create table article (
  articleID int not null auto_increment primary key,
  title varchar(255),
  body text
);
```

后面可以添加) `type=MyISAM`; 不过默认也是这样的。

#### InnoDB 表

**InnoDB** 和 **Berkley DB(BDB)**数据库引擎都是造就 **MySQL** 灵活性的技术的直接产品, 这项技术就是 **MySQL++ API**。在使用 **MySQL** 的时候, 你所面对的每一个挑战几乎都源于 **ISAM** 和 **MyISAM** 数据库引擎不支持事务处理也不支持外来键。尽管要比 **ISAM** 和 **MyISAM** 引擎慢很多, 但是 **InnoDB** 和 **BDB** 包括了对事务处理和外来键的支持, 这两点都是前两个引擎所没有的。如前所述, 如果你的设计需要这些特性中的一者或者两者, 那你就被迫使用后两个引擎中的一个了。



## 加注释

MySQL 允许在 SQL 代码中使用注释。这对于说明存放在文件中的查询很有用处。可用两个方式编写注释。以“#”号开头直到行尾的所有内容都认为是注释。另一种为 C 风格的注释。即, 以“/\*”开始, 以“\*/”结束的所有内容都认为是注释。C 风格的注释可跨多行, 如:

```
mysql> # this is a single line comment;
mysql> /* this is also a single line comment */
mysql> /* this, however,
/*> is a multiple line
/*> comment
/*> */
```

自 MySQL3 版以来, 可在 C 风格的注释中“隐藏” MySQL 特有的关键字, 注释以“/\*!”而不是以“/\*”起头。MySQL 查看这种特殊类型注释的内部并使用这些关键字, 但其他数据库服务器将这些关键字作为注释的一部分忽略。这样有助于编写由 MySQL 执行时利用 MySQL 特有功能的代码, 而且该代码也可以不用修改就用于其他数据库服务器。下面的两条语句对于非 MySQL 的数据库服务器是等价的, 但如果是 MySQL 服务器, 将在第二条语句中执行一个 INSERT DELAYED 操作:

```
mysql> insert into student values(517,'517','','90'),
-> insert /*! delayed */into student values(518,'518','','92');
```

自 MySQL3 以来, 除了刚才介绍的注释风格外, 还可以用两个短划线和一個空格(“--”)来开始注释; 从这两个短划线到行的结束的所有内容都作为注释处理。有的数据库以双短划线作为注释的起始。MySQL 也允许这样, 但需要加一个空格以免产生混淆。例如, 带有如像 5--7 这样的表达式的语句有可能被认为包含一个注释, 但不可能写 5--7 这样的表达式, 因此, 这是一个很有用的探索。然而, 这仅仅是一个探索, 最好不用这种风格的注释。

```
mysql> -- insert delayed */into student values(518,'518','','92');
mysql> _
```

## MySQL 命名

### 3.2 MySQL 的命名规则

几乎每条 SQL 语句都在某种程度上涉及一个数据库或其组成成分。本节介绍引用数据库、表、列、索引和别名的语法规则。名称是区分大小写的, 这里也对其进行了介绍。

#### 3.2.1 引用数据库的成分

在用名称引用数据库的成分时, 受到可使用的字符以及名称可具有的长度的限制。名称的形式还依赖于使用它们的上下文环境:

- 名称中可用的字符。名称可由服务器所采用的字符集中任意字母、数字、“\_”和“\$”组成。名称可按上述任意字符包括数字起头。但是名称不能单独由数字组成, 因为那样会使其与数值相混。MySQL 所提供的名称用一个数起始的能力是很不寻常的。如果使用了这样的一个名称, 要特别注意包含“E”和“e”的名称, 因为这两个字符可能会导致与表达式的混淆。23e + 14 表示列 23e 加 14, 但是 23e+14 又表示什么? 它表示一个科学表示法表示的数吗?

- 名称的长度。数据库、表、列和索引的名称最多可由 64 个字符组成。别名最多可长达 256 个字符。

- 名称限定词。为了引用一个数据库, 只要指定其名称即可, 如:

```
USE db_name
SHOW TABLES FROM db name
```

其中 db\_name 为所要引用的数据库名。要想引用一个表, 可有两种选择。一种选择是使用由数据库名和表名组成的完全限定的表名, 例如:

```
SHOW TABLES FROM db_name, tbl_name
```



```
SELECT * FROM db_name, tbl_name
```

其中, `tbl_name` 为要引用的表名。另一种选择是由表名自身来引用缺省(当前)数据库中的一个表。如果 `samp_db` 为缺省数据库中的一个表, 下面的两个语句是等价的:

```
SELECT * FROM member
```

```
SELECT * FROM samp_db.member
```

其中 `member` 为数据库 `samp_db` 中的一个表。要引用一个列, 有三种选择, 它们分别为: 完全限定、部分限定和非限定。完全限定名(如 `db_name.tbl_name.col_name`)是完全地指定。部分限定名(如 `tbl_name.col_name`)引用指定表中的列。非限定名(如 `col_name`)引用由环境上下文给出的表中的列。下面两个查询使用了相同的列名, 但是 `FROM` 子句提供的上下文指定了从哪个表中选择列:

```
SELECT last_name, first_name FROM president
```

```
SELECT last_name, first_name FROM members
```

虽然愿意的话, 提供完全限定名也是合法的, 但是一般不需要提供完全限定名, 如果用 `USE` 语句选择了一个数据库, 则该数据库将成为缺省数据库并在每一个非限定表引用中都隐含指向它。如果正使用一条 `SELECT` 语句, 此语句只引用了一个表, 那么该语句中的每个列引用都隐含指向这个表。只在所引用的表或数据库不能从上下文中确定时, 才需要对名称进行限定。下面是一些会出现混淆的情形:

■ 从多个数据库中引用表的查询。任何不在缺省数据库中的表都必须用“数据库名表名”的形式引用, 以便让 `MySQL` 知道在哪个数据库中找到该表。

■ 从多个表中选择一列的查询, 其中不止一个表含有具有该名称的列。

### 3.2.2 SQL 语句中的大小写规则

`SQL` 中的大小写规则在语句的不同部分是不同的, 而且还取决于所引用的东西以及运行的操作系统。下面给出相应的说明:

■ `SQL` 关键字和函数名。关键字与函数名是不区分大小写的。可按任意的大小写字符给出。下面的三条语句是等价的:

```
SELECT NOW();
```

```
Select now();
```

```
SELECT nOw();
```

■ 数据库与表名。`MySQL` 中数据库和表名对应于服务器主机上的基本文件系统中的目录和文件。因此, 数据库与表名是否区分大小写取决于主机上的操作系统处理文件名的方式。运行在 `UNIX` 上的服务器处理数据库名和表名是区分大小写的, 因为 `UNIX` 的文件名是区分大小写的。而 `Windows` 文件名是不区分大小写的, 所以运行在 `Windows` 上的服务器处理数据库名和表名也是不区分大小写的。如果在 `UNIX` 服务器上创建一个某天可能会移到 `Windows` 服务器上的数据库, 应该意识到这个特性: 如果现在创建了两个分别名为 `abc` 和 `ABC` 的表, 它们在 `Windows` 机器上将是没有区别的。避免这种情况发生的一种方法是选择一种字符(如小写), 总是以这种字符创建数据库和表名。这样, 在将数据库移到不同的服务器时, 名称的大小写便不会产生问题。

■ 列与索引名。`MySQL` 中列和索引名是不区分大小写的。下面的查询都是等价的:

```
SELECT name FROM student
```

```
SELECT NAME FROM student
```

```
SELECT nAmE FROM student
```

■ 别名。别名是区分大小写的。可按任意的大小写字符说明一个别名(大写、小写或大小写混合), 但是必须在任何查询中都以相同的大小写对其进行引用。不管数据库、表或别名是否是区分大小写的, 在同一个查询中的任何地方引用同一个名称都必须使用相同的大小写。对于 `SQL` 关键字、函数名或列名和索引名没有这个要求。可在同一个查询中多个地方用不同的大小写对它们进行引用。当然, 如果使用一致的大小写而不是“胡乱写”的风格(如 `SeLEct NamE FrOm ...`), 相应的查询可读性要强得多。

## 第九章

### 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类型来定义, 比如 int、float、double、decimal 等。其实正因为可选择的类型太多, 才需要依据一些原则来挑选最适合的数据类型。本章节将详细介绍字符、数据、日期数据类型的一些选择原则。

#### CHAR 和 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。

CHAR 和 VARCHAR 列检索的值并不是总相同, 因为检索 CHAR 列它将删除尾部的空格, 下面通过一个示例说明差别。

```
mysql> use mysqldemo;
Database changed
mysql> create table chardemo(v varchar(4),c char(4));
Query OK, 0 rows affected (0.53 sec)

mysql> insert into chardemo values('ad ','ab ');
Query OK, 1 row affected (0.05 sec)

mysql> select concat(v,'+') as varcharcolumn,concat(c,'+') as charcolumn from chardemo;
+-----+-----+
| varcharcolumn | charcolumn |
+-----+-----+
| ad +         | ab+        |
+-----+-----+
1 row in set (0.06 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的速度有较高要求的数据可以考虑使用 CHAR 类型来存储。

现在 MySQL 版本升级很快, 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 的使用原则不同, 下面做简单的概括。

**MyISAM 存储引擎:** 建议使用固定长度的数据表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 VARCHAR 或 CHAR 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, 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的 VARCHAR 列性能要好。因而, 主要的性能因素是数据行使用的存储问题。由于 CHAR 平均用的空间多于 VARCHAR,

因为使用 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。

### Text 和 Blob

一般在保存少量字符串的时候,我们会选择 VARCHAR 或者 CHAR;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。

```
mysql> create table blobdemo(id varchar(100),context text);
Query OK, 0 rows affected (0.09 sec)
```

往表中插入大量记录,这里使用 repeat 函数插入大字符串:

```
mysql> insert into blobdemo values(1,repeat('xmh',100));
Query OK, 1 row affected (0.05 sec)

mysql> insert into blobdemo values(2,repeat('zah',100));
Query OK, 1 row affected (0.03 sec)

mysql> insert into blobdemo values(3,repeat('zjy',100));
Query OK, 1 row affected (0.03 sec)

mysql> insert into blobdemo select * from blobdemo;
Query OK, 3 rows affected (0.05 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

### 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,如果插入的数据精度超过了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的形式存放的,所以定点数可以更精确地保存数据。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。在 MySQL 中 decimal、(或 numeric)用来表示定点数。

在简单了解浮点数和定点数的区别之后,下面通过示例来回顾一下浮点数的精确性问题。

创建一个表,表里只有一个 float 类型的列。

```
mysql> create table floatdemo(f float(8,1));
Query OK, 0 rows affected (0.14 sec)

mysql> desc floatdemo;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| f      | float(8,1)| YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.06 sec)
```

往里面插入一条数据(小数点后的第一个值小于五)。

```
mysql> insert into floatdemo values(5.17890);
Query OK, 1 row affected (0.08 sec)

mysql> select * from floatdemo;
+-----+
| f      |
+-----+
| 5.2    |
+-----+
1 row in set (0.00 sec)
```

往里面插入一条数据（小数点后的第一个值不小于五）。

```
mysql> select * from floatdemo;
+-----+
| f      |
+-----+
| 5.2    |
| 5.6    |
+-----+
2 rows in set (0.00 sec)
```

从上面可以看了，两次的值均被截断，并被四舍五入，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。

为了加深对浮点数与定点数的区别，请看如下示例。

```
mysql> create table decimaldemo(fnum float(10,2),dnum decimal(10,2));
Query OK, 0 rows affected (0.17 sec)

mysql> insert into decimaldemo values(517518.32,517518.32);
Query OK, 1 row affected (0.05 sec)

mysql> select * from decimaldemo;
+-----+-----+
| fnum      | dnum      |
+-----+-----+
| 517518.31 | 517518.32 |
+-----+-----+
1 row in set (0.00 sec)
```

上述数据在使用单精度浮点数时会产生误差，这是浮点数特有的问题，因此在精度要求较高的应用中（比如货币）要使用定点数保存数据。

有关浮点数的问题在 JAVA 中也有讲述，再回顾一下以前的问题。

```
public class Change{
    public static void main(String args[]){
        System.out.println(2.00 - 1.10);
    }
}
```

因为浮点型运算不会产生精确的结果，如果需要精确结果，请用 `BigDecimal`，参数为 `String` 而非 `double` 类型。改成

```
import java.math.*;
public class Anova{
    public static void main(String args[]){
        System.out.println(new BigDecimal("2.00").
            subtract(new BigDecimal("1.10")));
    }
}
```

```
}
}
```

在具体的工作中，需要考虑以下问题：

浮点数存在误差问题；

对货币等精度敏感的数据，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。

## 日期类型选择

常用的时间类型有 DATETIME、DATE、TIMESTAMP、TIME、YEAR。

MySQL 中的日期和时间类型

日期和时间类型	字节	最小值	零值	最大值
DATE	4	1000-01-01	0000-00-00 00:00:00	9999-12-31
DATETIME	8	1000-01-01 00:00:00	0000-00-00	9999-12-31 23:59:59
TIMESTAMP	4	19700101080001	00000000000000	2038 年的某个时刻
TIME	3	-838:59:59	00:00:00	838:59:59
YEAR	1	1901	0000	2155

这些类型的区别是：

要用来表示年月日，通常用 DATE 来表示

要用来表示年月日时分秒，通常用 DATETIME 表示

要用来表示时分秒，通常用 TIME 表示

要经常插入或更新日期为当前系统时间，通常用 TIMESTAMP

要用来表示年份，则用 YEAR 表示

每种日期类型都有一个范围，如果超出这个有效范围，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。

DATETIME、DATE、TIME 是最常用的 3 种日期类型，下面通过示例来讲解。

创建表 tdemo，然后创建三种日期类型字段。

```
mysql> create table tdemo(d date,t time,dt datetime);
Query OK, 0 rows affected (0.09 sec)

mysql> desc tdemo;
+-----+-----+-----+-----+-----+-----+
| 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()函数插入日期。

```
mysql> insert into tdemo values(now(),now(),now());
Query OK, 1 row affected (0.09 sec)

mysql> select * from tdemo;
+-----+-----+-----+
| d      | t      | dt      |
+-----+-----+-----+
| 2008-05-29 | 16:21:52 | 2008-05-29 16:21:52 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

从上图可清晰看出, 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, 下面创建一个示例。

```
mysql> create table t2demo(id1 timestamp);
Query OK, 0 rows affected (0.20 sec)

mysql> desc t2demo;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default      | Extra |
+-----+-----+-----+-----+-----+-----+
| id1   | timestamp | YES  |     | CURRENT_TIMESTAMP |      |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

可以发现, 系统给 t2demo 表自动创建了默认值 (系统日期), 下面插入一个空值试试。

```
mysql> insert into t2demo values(null);
Query OK, 1 row affected (0.06 sec)

mysql> select * from t2demo;
+-----+
| id1 |
+-----+
| 2008-05-29 16:27:32 |
+-----+
1 row in set (0.00 sec)
```

在 MySQL 表中, TIMESTAMP 类型字段只能有一列的默认值为 current\_timestmap, 如果有第二个 TIMESTAMP 类型, 则默认值为设置为 0。

在前面表的基础上增加一列。

```
mysql> alter table t add id2 timestamp;
ERROR 1146 (42S02): Table 'mysqldemo.t' doesn't exist
mysql> alter table t2demo add id2 timestamp;
Query OK, 1 row affected (0.28 sec)
Records: 1 Duplicates: 0 Warnings: 0
```

查看创建表的语句:

```
mysql> show create table t2demo \G;
***** 1. row *****
      Table: t2demo
Create Table: CREATE TABLE `t2demo` (
  `id1` timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,
  `id2` timestamp NOT NULL default '0000-00-00 00:00:00'
) ENGINE=InnoDB DEFAULT CHARSET=utf8
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\_timestamp。

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。

查看当前时区

```
mysql> show variables like 'time_zone';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| time_zone     | SYSTEM |
+-----+-----+
1 row in set (0.03 sec)
```

时区值为“SYSTEM”，这个值默认是和主机的时区值一致的，在中国“SYSTEM”实际是东八区(+8:00)。

```
mysql> select * from t2demo;
+-----+-----+
| id1          | id2          |
+-----+-----+
| 2008-05-29 16:43:05 | 2008-05-29 16:43:05 |
+-----+-----+
1 row in set (0.00 sec)
```

修改时区为东九区，再次执行插入操作，然后查看表中数据，会发现 id 的值不一样，就是时区差别所致。

```
mysql> set time_zone='+8:00';
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t2demo values(now(),now());
Query OK, 1 row affected (0.05 sec)

mysql> select * from t2demo;
+-----+-----+
| id1          | id2          |
+-----+-----+
| 2008-05-29 17:09:18 | 2008-05-29 18:09:18 |
| 2008-05-29 17:10:36 | 2008-05-29 17:10:36 |
+-----+-----+
2 rows in set (0.00 sec)
```

当插入时间值超过其范围后，值将归零处理。



```
mysql> insert into t2demo values(null,19700101080001);
Query OK, 1 row affected, 1 warning (0.05 sec)

mysql> select * from t2demo;
+-----+-----+
| id1          | id2          |
+-----+-----+
| 2008-05-29 17:43:05 | 2008-05-29 17:43:05 |
| 2008-05-29 17:51:01 | 0000-00-00 00:00:00 |
+-----+-----+
2 rows in set (0.00 sec)
```

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。

## 第 十 章

### 字符集概述

字符 (Character) 是文字与符号的总称，包括文字、图形符号、数学符号等。一组抽象字符的集合就是字符集 (Charset)。字符集常常和一种具体的语言文字对应起来，该文字中的所有字符或者大部分常用字符就构成了该文字的字符集，比如英文字符集。一组有共同特征的字符也可以组成字符集，比如繁体汉字字符集、日文汉字字符集。字符集的子集也是字符集。

20 世纪 60 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 (America Standard Code for Information Interchange)，即美国信息交换标准码。它已被国际标准化组织 (ISO) 定为国际标准，称为 ISO 646 标准。适用于所有拉丁文字字母，ASCII 码有 7 位码和 8 位码两种形式。ASCII 码于 1968 年提出，用于在不同计算机硬件和软件系统中实现数据传输标准化，在大多数的小型机和全部的个人计算机都使用此码。ASCII 码划分为两个集合：128 个字符的标准 ASCII 码和附加的 128 个字符的扩充和 ASCII 码。

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，包括 ISO-8859、GB2312、GBK 等。在追求贸易国际化的今天，一个软件要在不同文字的国家或地区发布，必须要进行本地化开发，所以统一字符集问题就显得迫切需要。

### Unicode 支持世界上所有语言的编码和转换。

事实上，字母、数字和标点符号等所有字符在计算机中都是用数字来表示的。前面也谈论到对字符

进行编码有多种不同的方式。

随着计算机应用变得越来越全球化, 推动了支持全世界各种语言字符的需求, 很多人越来越清楚一个单一的统一编码方案是必要的。Unicode 标准就是在这样的背景下诞生的。Unicode 是 XML 的默认编码方案, 是 LDAP 要求的编码方案, 也是 Java 和 Windows XP 使用的基础字符集。多年前, 你需要理解 ASCII 码才能成为一名成功的程序员。而今天, 你只需要理解 Unicode。

什么是 Unicode?

Unicode 是一个涵盖了目前全世界使用的所有已知字符的单一编码方案。每个字符都被分配了一个称为码点的不同数值。Unicode 标准将字符组织成相关字符块。例如, Unicode 使用从 0x0400 到 0x04FF 的码点块来表示俄国人和乌克兰人使用的西里尔文字及相关字母表。(符号 0x 表示后面的字符是一个十六进制的数值。Unicode 码点通常以十六进制表示。)文字是一个或多个相关书写系统或语言中使用的符号集合。因此, 西里尔文字是不同西里尔字母表使用的所有字符的一个超集。Unicode 还在不断发展, 以增加它所支持的文字的数量, 同时还包括新的字符, 如欧元符号(?)。

### 常见字符集

ISO 8859-1 (大多数西欧语言, 如英语、法语、西班牙语和德语)

ISO 8859-2 (大多数中欧语言, 如捷克语、波兰语和匈牙利语)

ISO 8859-4 (斯堪的纳维亚和波罗的海语)

ISO 8859-5 (俄语)

ISO 8859-6 (阿拉伯语, 包括许多具有更多表示形式的字符字型)

ISO 8859-7 (希腊语)

ISO 8859-8 (希伯来语)

ISO 8859-9 (土耳其语)

TIS 620.2533 (泰语, 包括许多具有更多表示形式的字符字型)

ISO 8859-15 (大多数带有欧元符号的西欧语言)

GB 2312-1980 (简体中文)

JIS X 0201-1976, JIS X 0208-1990 (日语)

KSC 5601-1992 附件 3 (朝鲜语)

GB 18030 (简体中文) 全称是《信息技术交换用汉字编码字符集、基本集的扩充》

HKSCS (繁体中文, 中国香港特别行政区)

Big5 (繁体中文, 中国台湾地区)

IS 13194.1991, 也称为 ISCII (印度语, 包括许多具有更多表示形式的字符字型)

常用字符集归纳:

UTF-8: 互联网和 UNIX/LINUX 广泛支持的 Unicode 字符集;

GBK : 不是国标, 但支持的系统较多

GB2312: 早期标准, 不再推荐使用

ISO-8859-1/latin1: 西欧字符集, 经常被一些程序员用来转码

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。

MySQL5.0 目前支持几十种字符集, UTF-8 是 MySQL5.0 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) 如是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。这样可以避免因为字符集转换带来的性能开销和数据损失。

### MySQL 支持的字符集简介

MySQL 服务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，相比其它类型的数据库而言，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 |
+-----+-----+-----+-----+
| big5    | Big5 Traditional Chinese | big5_chinese_ci | 2 |
| latin1  | cp1252 West European | latin1_swedish_ci | 1 |
| gbk     | GBK Simplified Chinese | gbk_chinese_ci | 2 |
| latin5  | ISO 8859-9 Turkish | latin5_turkish_ci | 1 |
+-----+-----+-----+-----+
| utf8    | UTF-8 Unicode | utf8_general_ci | 3 |
+-----+-----+-----+-----+
36 rows in set (0.13 sec)
```

或者查看 information\_schema.character\_set，可以显示所有的字符集和该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 (0.08 sec)
```

MySQL 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 (0.09 sec)

mysql>
```

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（大小不敏感）、\_cs（大小敏感）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。

下面这个例子中，如果是指定“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，如果是按照 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。

在进行下列练习前，请确认当前数据库的字符集，例如本机默认的字符集为 UTF8，请根据本地机器安装的数据库默认字符集进行调整。

```
mysql> select case when 'A' collate utf8_unicode_ci='a' collate utf8_unicode_ci
then 1 else 0 end;
+-----+
| case when 'A' collate utf8_unicode_ci='a' collate utf8_unicode_ci then 1 else 0 end |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)
```

需要区分大小写时：

```
mysql> select case when 'A' collate utf8_bin='a' collate utf8_bin then 1 else 0
end;
+-----+
| case when 'A' collate utf8_bin='a' collate utf8_bin then 1 else 0 end |
+-----+
| 0 |
+-----+
1 row in set (0.00 sec)
```

## 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。

### 1、服务器字符集和校对规则

服务器的字符集和校对，在 MySql 服务启动的时候确定。

- 在 my.cnf 中设置 `default-character-set=gbk`
- 在启动选项中设置 `mysql -uroot -p3239436 --default-character-set=gbk`
- 在编译的时候指定

如果没有特别的指定服务器的字符集，默认的为 `latin1`。上面三种只是指定了字符集，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'`” 查看当前服务器的字符集

```
mysql> show variables like 'character_set_server';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| character_set_server | utf8 |
+-----+-----+
1 row in set (0.00 sec)
```

可以用 “`show variables like 'collation_server'`” 查看当前服务器的校对规则

```
mysql> show variables like 'collation_server';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| collation_server | utf8_general_ci |
+-----+-----+
1 row in set (0.00 sec)
```

### 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。建议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使用 “`show variables like 'character_set_database'`” 查看。

```
mysql> show variables like 'character_set_database';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| character_set_database | utf8 |
+-----+-----+
1 row in set (0.00 sec)
```

校对规则使用 “`show variables like 'collation_database'`” 命令查看。

### 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。

建议在创建数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。

```
mysql> show create table student \G;
***** 1. row *****
      Table: student
Create Table: CREATE TABLE `student` (
  `id` int(8) NOT NULL auto_increment,
  `name` varchar(50) default NULL,
  `intro` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

#### 4、列字符集和校对规则

每一个“字符”列（即，CHAR、VARCHAR 或 TEXT 类型的列）有一个列字符集和一个列 校对规则，它不能为空。列定义语法有一个可选子句来指定列字符集和校对规则：

```
create table table1(column1 varchar(5) character set latin1 collate latin1_german1_ci);
```

MySQL 按照下面的方式选择列字符集和校对规则：

- 如果指定了 CHARACTER SET X 和 COLLATE Y, 那么采用 CHARACTER SET X 和 COLLATE Y。
- 如果指定了 CHARACTER SET X 而没有指定 COLLATE Y, 那么采用 CHARACTER SET X 和 CHARACTER SET X 的默认校对规则。
- 否则，采用表字符集和服务器的校对规则。

#### 5、连接字符集和校对规则

前面讲的四种设置方式，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端之间的交互的字符集和校对规则的设置。

mysql4.1 及其之后的版本，对字符集的支持分为四个层次：服务器(server)，数据库(database)，数据表(table)和连接(connection)：

character\_set\_server：这是设置服务器使用的字符集

character\_set\_client：这是设置客户端发送查询使用的字符集

character\_set\_connection：这是设置服务器需要将收到的查询串转换成的字符集

character\_set\_results：这是设置服务器要将结果数据转换到的字符集，转换后才发送给客户端

整个过程：

- client(如 php 程序)发送一个查询；
- 服务器收到查询，将查询串从 character\_set\_client 转换到 character\_set\_connection，然后执行转换后的查询；
- 服务器将结果数据转换到 character\_set\_results 字符集后发送回客户端。

你可以用下边两条命令查看一下系统的字符集和排序方式设定：

```
mysql> SHOW VARIABLES LIKE 'character_set_%';
```



```
mysql> SHOW VARIABLES LIKE 'character_set_%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| character_set_client | gbk |
| character_set_connection | gbk |
| character_set_database | utf8 |
| character_set_filesystem | binary |
| character_set_results | gbk |
| character_set_server | utf8 |
| character_set_system | utf8 |
+-----+-----+
```

```
mysql> SHOW VARIABLES LIKE 'collation_%';
```

```
mysql> SHOW VARIABLES LIKE 'collation_%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| collation_connection | gbk_chinese_ci |
| collation_database | utf8_general_ci |
| collation_server | utf8_general_ci |
+-----+-----+
3 rows in set (0.00 sec)
```

mysql 默认用的字符集是 latin1,连接校对用的 latin1\_swedish\_ci。看到这儿你应试有点明白了,

我们通过 JSP 发送的查询一般是 utf8 或者 GBK,GB2312,而它默认的是 latin1,所以直接进数据库里查看数据,都是一些乱码。(存取出来放在网页上大多是正常的。)只要把上述变量设置一致了,不管是直接在数据库里查看,还是在客户端里查看都不会出现中文乱码了。

## 第十一章

索引是提高查询速度的最重要的工具。当然还有其它的一些技术可供使用,但是一般来说引起最大性能差异的都是索引的正确使用。在 MySQL 邮件列表中,人们经常询问那些让查询运行得更快的方法。

在大多数情况下,我们应该怀疑数据表上有没有索引,并且通常在添加索引之后立即解决了问题。当然,并不总是这样简单就可以解决问题的,因为优化技术本来就并非总是简单的。然而,如果没有使用索引,在很多情况下,你试图使用其它的方法来提高性能都是在浪费时间。首先使用索引来获取最大的性能提高,接着再看其它的技术是否有用。

id	name	intro	score
442	xmh883	no intro	60
442	xmh884	no intro	60
440	xmh879	no intro	60
440	xmh880	no intro	60
440	zyj879	no intro	60
439	xmh877	no intro	60
439	xmh878	no intro	60
439	zyj877	no intro	60
439	zyj878	no intro	60
443	zyj886	no intro	60
444	xmh887	no intro	60
438	zyj876	no intro	60
440	zyj880	no intro	60
441	xmh881	no intro	60
441	xmh882	no intro	60
441	zyj881	no intro	60
441	zyj882	no intro	60
444	xmh888	no intro	60
438	zyj875	no intro	60
442	zyj883	no intro	60
442	zyj884	no intro	60
443	xmh885	no intro	60
443	xmh886	no intro	60
443	zyj885	no intro	60

图 1: 无索引的 student 表

图 2 是同样的一张数据表, 但是增加了对 ad 表的 company\_num 数据列的索引。这个索引包含了 ad 表中的每个数据行的条目, 但是索引的条目是按照 company\_num 值排序的。现在, 我们不是逐行查看以搜寻匹配的数据项, 而是使用索引。假设我们查找公司 13 的所有数据行。我们开始扫描索引并找到了该公司的三个值。接着我们碰到了公司 14 的索引值, 它比我们正在搜寻的值大。索引值是排过序的, 因此当我们读取了包含 14 的索引记录的时候, 我们就知道再也不会会有更多的匹配记录, 可以结束查询操作了。因此使用索引获得的功效是: 我们找到了匹配的数据行在哪儿终止, 并能够忽略其它的数据行。另一个功效来自使用定位算法查找第一条匹配的条目, 而不需要从索引头开始执行线性扫描 (例如, 二分搜索就比线性扫描要快一些)。通过使用这种方法, 我们可以快速地定位第一个匹配的值, 节省了大量的搜索时间。数据库使用了多种技术来快速地定位索引值, 但是在本文中我们不关心这些技术。重点是它们能够实现, 并且索引是个好东西。



id	id	name	intro	score
1	442	xmh883	no intro	60
2	442	xmh884	no intro	60
3	440	xmh879	no intro	60
4	440	xmh880	no intro	60
5	440	zyj879	no intro	60
6	439	xmh877	no intro	60
7	439	xmh878	no intro	60
8	439	zyj877	no intro	60
9	439	zyj878	no intro	60
10	443	zyj886	no intro	60
11	444	xmh887	no intro	60
12	438	zyj876	no intro	60
13	440	zyj880	no intro	60
14	441	xmh881	no intro	60
15	441	xmh882	no intro	60
16	441	zyj881	no intro	60
17	441	zyj882	no intro	60
18	444	xmh888	no intro	60
19	438	zyj875	no intro	60
20	442	zyj883	no intro	60
21	442	zyj884	no intro	60
22	443	xmh885	no intro	60
23	443	xmh886	no intro	60
	443	zyj885	no intro	60

你可能要问, 我们为什么不对数据行进行排序从而省掉索引? 这样不是也能实现同样的搜索速度的改善吗? 是的, 如果表只有一个索引, 这样做也可能达到相同的效果。但是你可能添加第二个索引, 那么就无法一次使用两种不同方法对数据行进行排序了(例如, 你可能希望在顾客名称上建立一个索引, 在顾客 ID 号或电话号码上建立另外一个索引)。把与数据行相分离的条目作为索引解决了这个问题, 允许我们创建多个索引。此外, 索引中的行一般也比数据行短一些。当你插入或删除新的值的时候, 移动较短的索引值比移动较长数据行的排序次序更加容易。

不同的 MySQL 存储引擎的索引实现的具体细节信息是不同的。例如, 对于 MyISAM 数据表, 该表的数据行保存在一个数据文件中, 索引值保存在索引文件中。一个数据表上可能有多个索引, 但是它们都被存储在同一个索引文件中。索引文件中的每个索引都包含一个排序的键记录(它用于快速地访问数据文件)数组。

与此形成对照的是, BDB 和 InnoDB 存储引擎没有使用这种方法来分离数据行和索引值, 尽管它们也把索引作为排序后的值集合进行操作。在默认情况下, BDB 引擎使用单个文件存储数据和索引值。InnoDB 使用单个数据表空间 (tablespace), 在表空间中管理所有 InnoDB 表的数据和索引存储。我们可以把 InnoDB 配置为每个表都在自己的表空间中创建, 但是即使是这样, 数据表的数据和索引也存储在同一个表空间文件中。

前面的讨论描述了单个表查询环境下的索引的优点, 在这种情况下, 通过减少对整个表的扫描, 使用索引明显地提高了搜索的速度。当你运行涉及多表联结 (join) 查询的时候, 索引的价值就更高了。在单表查询中, 你需要在每个数据列上检查的值的数量是表中数据行的数量。在多表查询中, 这个数量可能大幅度上升, 因为这个数量是这些表中数据行的数量所产生的。

假设你拥有三个未索引的表 t1、t2 和 t3, 每个表都分别包含数据列 i1、i2 和 i3, 并且每个表都包含了 1000 条数据行, 其序号从 1 到 1000。查找某些值匹配的数据行组合的查询可能如下所示:

这一部分讲述了索引是什么以及索引是怎么样提高查询性能的。它还讨论了在某些环境中索引可能降低性能,并为你明智地选择数据表的索引提供了一些指导方针。在下一部分中我们将讨论 MySQL 查询优化器,它试图找到执行查询的效率最高的方法。了解一些优化器的知识,作为对如何建立索引的补充,对我们是有好处的,因为这样你才能更好地利用自己所建立的索引。某些编写查询的方法实际上让索引不起作用,在一般情况下你应该避免这种情形的发生。

为了演示索引的效果,向表 student 中先插入 10000 条数据。

```
delimiter $$
create procedure insertt()
begin
declare i int default 0;
ins:loop
set i = i+1;
if i>10000 then
leave ins;
end if;
insert into student values(i/2,concat('zyj',",",i),'no intro',60);
end loop ins;
end $$
```

如果出现 Column count doesn't match value count at row 1 错误,则是由于写的 SQL 语句里列的数目和后面的值的数目不一致,请查正修改。

然后执行如下语句,调用储存过程。

```
delimiter ;
call insertt();
```

在查找 name="zyj9991"的记录时,没有用索引时,MySQL 会扫描所有记录,即要查询 10000 次。查询所需时间为 0.03 秒。

```
mysql> select id,name from student where intro='no intro' and name='zyj9991';
+-----+-----+
| id    | name    |
+-----+-----+
| 4996  | zyj9991 |
+-----+-----+
1 row in set (0.03 sec)
```

如果在 name 上已经建立了索引,MySQL 无须任何扫描,即准确可找到该记录!

```
mysql> select id,name from student where intro='no intro' and name='zyj9999';
+-----+-----+
| id    | name    |
+-----+-----+
| 5000  | zyj9999 |
+-----+-----+
1 row in set (0.01 sec)
```

## 索引分单列索引和组合索引

单列索引:即一个索引只包含单个列,一个表可以有多个单列索引,但这不是组合索引。

组合索引:即一个索引包含多个列。

## 介绍一下索引的类型

## 1、普通索引。

这是最基本的索引，它没有任何限制。它有以下几种创建方式：

- 创建索引：

CREATE INDEX indexName ON tableName(tableColumns(length));如果是 CHAR,VARCHAR 类型，length 可以小于字段实际长度;如果是 BLOB 和 TEXT 类型，必须指定 length，下同。

```
mysql> create index sindex on student(id<4>);
Query OK, 11005 rows affected (0.81 sec)
Records: 11005 Duplicates: 0 Warnings: 0
```

- 修改表结构：

ALTER table tableName ADD INDEX [index\_name] [index\_type] (index\_col\_name,...)

在列 id 上添加索引

```
mysql> alter table student add index <id>;
Query OK, 11005 rows affected (0.66 sec)
Records: 11005 Duplicates: 0 Warnings: 0
```

如果不指定索引的名字，则与列名一致。

- 创建表的时候直接指定：

CREATE TABLE tableName ( [...], INDEX [indexName] (tableColumns(length)) );

```
mysql> create table pers<
-> id int(4) default 5,
-> name varchar(30),
-> index vindex(name)>;
Query OK, 0 rows affected (0.11 sec)
```

## 2、唯一索引。

它与前面的"普通索引"类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

- 创建索引：

CREATE UNIQUE INDEX indexName ON tableName(tableColumns(length))

```
mysql> create unique index unindex on dept(id);
Query OK, 3 rows affected (0.20 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

这里以前面所学的 dept 表为例，要注意表中的列要具有唯一值。

- 修改表结构：

ALTER table tableName ADD UNIQUE INDEX [index\_name] [index\_type] (index\_col\_name,...)

```
mysql> alter table dept add unique index unindex <id>;
Query OK, 3 rows affected (0.24 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

- 创建表的时候直接指定：

CREATE TABLE tableName ( [...], UNIQUE INDEX [indexName] (tableColumns(length)) );

```
create table pers(
id int(4) default 5,
name varchar(30),
unique index vindex(name));
```

## 3.主键索引

它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引：

```
create table testIndex(  
  id int not null auto_increment,  
  name varchar(20) not null,  
  primary key(id));
```

在 MySQL 中, 当你建立主键时, 索引同时也已建立起来了。不必重复设置。记住: 一个表只能有一个主键。当然也可以用 ALTER 命令来创建主键索引。

#### 4.全文索引

在 MySQL 中使用全文索引。在使用 like 进行模糊查询, 当数据量大到一定程度的时候, 我们会发现查询的效率是相当低下的。下面就此介绍一下 mysql 提供全文索引和搜索的功能。

很多互联网应用程序都提供了全文搜索功能, 用户可以使用一个词或者词语片断作为查询项目来定位匹配的记录。在后台, 这些程序使用在一个 SELECT 查询中的 LIKE 语句来执行这种查询, 尽管这种方法可行, 但对于全文查找而言, 这是一种效率极端低下的方法, 尤其在处理大量数据的时候。

MySQL 针对这一问题提供了一种基于内建的全文查找方式的解决方案。在此, 开发者只需要简单地标记出需要全文查找的字段, 然后使用特殊的 MySQL 方法在那些字段运行搜索, 这不仅仅提高了性能和效率 (因为 MySQL 对这些字段做了索引来优化搜索), 而且实现了更高质量的搜索, 因为 MySQL 使用自然语言来智能地对结果评级, 以去掉不相关的项目。

全文索引在 MySQL 中是一个 FULLTEXT 类型索引。FULLTEXT 索引用于 MyISAM 表, 可以在 CREATE TABLE 时或之后使用 ALTER TABLE 或 CREATE INDEX 在 CHAR、VARCHAR 或 TEXT 列上创建。对于大的数据库, 将数据装载到一个没有 FULLTEXT 索引的表中, 然后再使用 ALTER TABLE (或 CREATE INDEX) 创建索引, 这将是非常快的。将数据装载到一个已经有 FULLTEXT 索引的表中, 将是非常慢的。

全文搜索通过 MATCH() 函数完成。

创建全文索引的过程, 主要有两种办法:

第一种是创建表的时候就创建全文索引, 第二种类是在创建表以后再增加全文索引, 通过上面的引文我们知道后者比前者有些好处

##### 1、先创建表, 然后通过"ALTER TABLE"增加全文索引

```
--创建数据库  
create database ftt;  
--使用数据库  
use ftt;  
--创建表  
create table reviews  
(  
  id int(5) primary key not null auto_increment,  
  data text  
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

往表里插入数据

```
INSERT ignore INTO reviews (id, data) VALUES  
(1, 'Gingerboy has a new single out called Throwing Rocks. It's great!');  
INSERT ignore INTO reviews (id, data) VALUES  
(2, 'Hello all, I really like the new Madonna single. One of the hottest tracks currently playing...I've been listening to it all day');  
INSERT ignore INTO reviews (id, data) VALUES  
(3, 'Have you heard the new band Hotter Than Hell? They have five members and they burn their instruments when they play in concerts. These guys totally rock! Like, awesome, dude!');
```

通过 alter 增加全文索引,alter table reviews add fulltext index (data);

```
mysql> alter table reviews add fulltext index (data);
Query OK, 3 rows affected (0.20 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

通过 match 和 against 实现全文检索

select id from reviews where match (data) against ('Madonna');

```
mysql> select id from reviews where match (data) against ('Madonna');
+----+
| id |
+----+
| 2 |
+----+
1 row in set (0.00 sec)
```

搜索出包含 Madonna 或者 instruments 的所有记录的 id 号

select id from reviews where match (data) against ('+Madonna+instruments');

```
mysql> select id from reviews where match (data) against ('+Madonna+instruments'
);
+----+
| id |
+----+
| 2 |
| 3 |
+----+
2 rows in set (0.02 sec)
```

检索出包含 Mado 的所有记录的 id

select id from reviews where match (data) against ('Mado\*' in boolean mode);

```
mysql> select id from reviews where match (data) against ('Mado*' in boolean mod
e);
+----+
| id |
+----+
| 2 |
+----+
1 row in set (0.02 sec)
```

删除索引的语法:

DROP INDEX index\_name ON tableName

```
mysql> drop index data on reviews;
Query OK, 3 rows affected (0.09 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

单列索引和组合索引

为了形象地对比两者, 再建一个表:

```
CREATE TABLE myIndex ( i_testID INT NOT NULL AUTO_INCREMENT, vc_Name VARCHAR(50)
NOT NULL, vc_City VARCHAR(50) NOT NULL, i_Age INT NOT NULL, i_SchoolID INT NOT NULL,
PRIMARY KEY (i_testID) );
```

在这 10000 条记录里面 7 上 8 下地分布了 5 条 vc\_Name="erquan"的记录, 只不过 city,age,school 的

组合各不相同。

来看这条 T-SQL:

```
SELECT i_testID FROM myIndex WHERE vc_Name='erquan' AND vc_City='郑州' AND i_Age=25;
```

首先考虑建单列索引:

在 vc\_Name 列上建立了索引。执行 T-SQL 时, MYSQL 很快将目标锁定在了 vc\_Name=erquan 的 5 条记录上, 取出来放到一中间结果集。在这个结果集里, 先排除掉 vc\_City 不等于"郑州"的记录, 再排除 i\_Age 不等于 25 的记录, 最后筛选出唯一的符合条件的记录。

虽然在 vc\_Name 上建立了索引, 查询时 MYSQL 不用扫描整张表, 效率有所提高, 但离我们的要求还有一定的距离。同样的, 在 vc\_City 和 i\_Age 分别建立的单列索引的效率相似。

为了进一步榨取 MySQL 的效率, 就要考虑建立组合索引。就是将 vc\_Name,vc\_City,i\_Age 建到一个索引里:

```
ALTER TABLE myIndex ADD INDEX name_city_age (vc_Name(10),vc_City,i_Age);--注意了, 建表时, vc_Name 长度为 50, 这里为什么用 10 呢? 因为一般情况下名字的长度不会超过 10, 这样会加速索引查询速度, 还会减少索引文件的大小, 提高 INSERT 的更新速度。
```

执行 T-SQL 时, MySQL 无须扫描任何记录就到找到唯一的记录!!

肯定有人要问了, 如果分别在 vc\_Name,vc\_City,i\_Age 上建立单列索引, 让该表有 3 个单列索引, 查询时和上述的组合索引效率一样吧? 大不一样, 远远低于我们的组合索引~~虽然此时有了三个索引, 但 MySQL 只能用到其中的那个它认为似乎是最有效率的单列索引。

建立这样的组合索引, 其实是相当于分别建立了

vc\_Name,vc\_City,i\_Age

vc\_Name,vc\_City

vc\_Name

这样的三个组合索引! 为什么没有 vc\_City,i\_Age 等这样的组合索引呢? 这是因为 mysql 组合索引"最左前缀"的结果。简单的理解就是只从最左面的开始组合。并不是只要包含这三列的查询都会用到该组合索引, 下面的几个 T-SQL 会用到:

```
SELECT * FROM myIndex WHERE vc_Name="erquan" AND vc_City="郑州"
```

```
SELECT * FROM myIndex WHERE vc_Name="erquan"
```

而下面几个则不会用到:

```
SELECT * FROM myIndex WHERE i_Age=20 AND vc_City="郑州"
```

```
SELECT * FROM myIndex WHERE vc_City="郑州"
```

## 使用索引

到此你应该会建立、使用索引了吧? 但什么情况下需要建立索引呢? 一般来说, 在 WHERE 和 JOIN 中出现的列需要建立索引, 但也不完全如此, 因为 MySQL 只对 <, <=, =, >, >=, BETWEEN, IN, 以及某些时候的 LIKE(后面有说明)才会使用索引。

SELECT t.vc\_Name FROM testIndex t LEFT JOIN myIndex m ON t.vc\_Name=m.vc\_Name WHERE m.i\_Age=20 AND m.vc\_City='郑州' 时, 有对 myIndex 表的 vc\_City 和 i\_Age 建立索引的需要, 由于 testIndex 表的 vc\_Name 开出现在了 JOIN 子句中, 也有对它建立索引的必要。

刚才提到了, 只有某些时候的 LIKE 才需建立索引? 是的。因为在以通配符 % 和 \_ 开头作查询时, MySQL 不会使用索引, 如

```
SELECT * FROM myIndex WHERE vc_Name like'erquan%'
```

会使用索引, 而

```
SELECT * FROM myIndex WHERE vc_Name like'%erquan'
```

就不会使用索引了。



## 索引的不足之处

1.虽然索引大大提高了查询速度,同时却会降低更新表的速度,如对表进行 INSERT、UPDATE 和 DELETE。因为更新表时,MySQL 不仅要保存数据,还要保存一下索引文件

2.建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重,但如果你在一个大表上创建了多种组合索引,索引文件的会膨胀很快。

小结:

讲了这么多,无非是想利用索引提高数据库的执行效率。不过索引只是提高效率的一个因素。如果你的 MySQL 有大数据的表,就需要花时间研究建立最优秀的索引或优化查询语句。

## 第 十二 章

视图(view):从一个或几个基本表中根据用户需要而做成一个虚表

1:视图是虚表,它在存储时只存储视图的定义,而没有存储对应的数据

2:视图只在刚刚打开的一瞬间,通过定义从基表中搜集数据,并展现给用户

为什么有了表还要引入视图呢?这是因为视图具有以下几个优点:

1:能分割数据,简化观点

可以通过 select 和 where 来定义视图,从而可以分割数据基表中某些对于用户不关心的数据,使用户把注意力集中到所关心的数据列,进一步简化浏览数据工作。

2:为数据提供一定的逻辑独立性

如果为某一个基表定义一个视图,即使以后基本表的内容的发生改变了也不会影响“视图定义”所得到的数据

3:提供自动的安全保护功能

视图能像基本表一样授予或撤消访问许可权。

4:视图可以间接对表进行更新,因此视图的更新就是表的更新

### 创建视图

MySQL 中创建视图可以通过 create view 语句来实现,具体创建格式如下:

```
create [ or replace] [algorithm={merge | temptable | undefined}] view view_name  
[( column_list)] as select_statement [with [cascaded | local] check option]
```

algorithm={merge | temptable | undefined}属性用于优化 MySQL 视图的执行,该属性有 3 个可用的设置。下面将介绍这 3 个设置的使用方法。

merge: 该参数使 MySQL 执行视图时传入的任何子句合并到视图的查询定义中。

temptable: 如果视图底层表中的数据有变化,这些变化将在下次通过表时立即反映出来。

undefined: 当查询结果和视图结果为一一对应关系时, MySQL 将 algorithm 设定为 temptable。

view\_name: 新建视图的名称。

select\_statement: SQL 查询语句用于限定虚表的内容

根据 student 表, 创建一个名称为 v 的视图。具体语句如下所示:

```
mysql> create view v as select id,name,concat(name,'--',intro) from student;
Query OK, 0 rows affected (0.06 sec)

mysql> select * from v;
+-----+-----+-----+
| id | name | concat(name,'--',intro) |
+-----+-----+-----+
| 1 | xmh | xmh--the demo is good |
| 2 | zyj | zyj--the second demo |
| 3 | zyj | zyj--the third demo |
+-----+-----+-----+
3 rows in set (0.03 sec)
```

使用 algorithm 创建视图的实例。

```
mysql> create algorithm=merge view v3(xuhao,xinming) as select id,name from student;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from v3;
+-----+-----+
| xuhao | xinming |
+-----+-----+
| 1 | xmh |
| 2 | zyj |
| 3 | zyj |
+-----+-----+
3 rows in set (0.00 sec)
```

## 显示视图

Show tables 不仅显示表的名字, 也会显示视图的名字。

```
mysql> show tables;
+-----+
| Tables_in_mysql demo |
+-----+
| student |
| stuvview |
+-----+
2 rows in set (0.00 sec)
```

查表的状态信息显示视图的情况



```
mysql> show table status like 'st%' \G;
***** 1. row *****
      Name: student
      Engine: InnoDB
      Version: 10
      Row_format: Compact
      Rows: 3
      Avg_row_length: 5461
      Data_length: 16384
      Max_data_length: 0
      Index_length: 0
      Data_free: 0
      Auto_increment: 4
      Create_time: 2008-05-25 18:12:21
      Update_time: NULL
      Check_time: NULL
      Collation: utf8_general_ci
      Checksum: NULL
      Create_options:
      Comment: InnoDB free: 7168 kB
***** 2. row *****
      Name: stuvview
      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
2 rows in set (0.08 sec)
```

查询某一个视图

```
mysql> show create view stuview;
+-----+-----+
| View      | Create View |
+-----+-----+
| stuview   | CREATE ALGORITHM=UNDEFINED DEFINER='root'@'localhost' SQL SECURITY DEFINER VIEW 'stuview' AS select 'student'.'id' AS 'id','student'.'name' AS 'name',concat('student'.'name',_utf8'--','student'.'intro') AS 'concat(name,'--',intro)' from 'student' |
+-----+-----+
1 row in set (0.01 sec)
```

通过系统表查看视图

先进入系统的数据库

use information\_schema;

```
mysql> select * from views where table_name='stuview' \G ;
***** 1. row *****
TABLE_CATALOG: NULL
TABLE_SCHEMA: mysqldemo
TABLE_NAME: stuview
VIEW_DEFINITION: /* ALGORITHM=UNDEFINED */ select 'mysqldemo'.'student'.'id' AS 'id','mysqldemo'.'student'.'name' AS 'name',concat('mysqldemo'.'student'.'name',_utf8'--','mysqldemo'.'student'.'intro') AS 'concat(name,'--',intro)' from 'mysqldemo'.'student'
CHECK_OPTION: NONE
IS_UPDATABLE: YES
DEFINER: root@localhost
SECURITY_TYPE: DEFINER
1 row in set (0.09 sec)

ERROR:
No query specified
```

## 修改视图

MySQL 中修改视图可以通过 alter view 语句实现, alter view 语句具体使用说明如下:

```
alter view [algorithm={merge | temptable | undefined}] view view_name
[(column_list)] as select_statement[with [cascaded | local] check option]
```

algorithm: 该参数已经在创建视图中作了介绍, 这里不再赘述。

view\_name: 视图的名称。

select\_statement: SQL 语句用于限定视图。

修改 stuvview 视图，要求修改后只列出 username 字段。代码如下：

```
mysql> alter view stuvview as select * from student;
Query OK, 0 rows affected (0.06 sec)

mysql> select * from stuvview;
+----+-----+-----+
| id | name | intro |
+----+-----+-----+
| 1 | xmh | the demo is good |
| 2 | zyj | the second demo |
| 3 | zyj | the third demo |
+----+-----+-----+
3 rows in set (0.00 sec)
```

### 删除视图

```
mysql> DROP VIEW U;
Query OK, 0 rows affected (0.02 sec)
```

补充：

要强调的是视图创建的时候如果不要视图影响基表的数据应该指定 `algorithm=temptable`，如果不指定其默认为 `undefined`。

```
mysql> create algorithm=temptable view v2 as select id,name,concat(name,'--',intro) from student;
Query OK, 0 rows affected (0.02 sec)

mysql> select * from v2;
+----+-----+-----+
| id | name | concat(name,'--',intro) |
+----+-----+-----+
| 5 | zxx | NULL |
| 6 | xmh11 | xmh11--no intro |
| 6 | xmh12 | xmh12--no intro |
| 8 | xmh16 | xmh16--no intro |
| 9 | zxxyyyyy | zxxyyyyy--no intro |
| 6 | zyj11 | zyj11--no intro |
| 17 | TestProc | NULL |
+----+-----+-----+
7 rows in set (0.00 sec)

mysql> update v2 set name='rrr' where id=9;
ERROR 1288 (HY000): The target table v2 of the UPDATE is not updatable
```

## 第十三章

将常用的或很复杂的工作, 预先用 SQL 语句写好并用一个指定的名称存储起来, 那么以后要叫数据库提供与已定义好的存储过程的功能相同的服务时, 只需调用 `execute`, 即可自动完成命令。

存储过程的优点:

1. 存储过程只在创造时进行编译, 以后每次执行存储过程都不需再重新编译, 而一般 SQL 语句每执行一次就编译一次, 所以使用存储过程可提高数据库执行速度。
2. 当对数据库进行复杂操作时(如对多个表进行 `Update, Insert, Query, Delete` 时), 可将此复杂操作作用存储过程封装起来与数据库提供的事务处理结合一起使用。
3. 存储过程可以重复使用, 可减少数据库开发人员的工作量
4. 安全性高, 可设定只有某此用户才具有对指定存储过程的使用权

定义及实例存储过程是一种存储在书库中的程序(就像正规语言里的子程序一样), 准确的来说, MySQL 支持的“`routines` (例程)”有两种: 一是我们说的存储过程, 二是在其他 SQL 语句中可以返回值的函数(使用起来和 Mysql 预装载的函数一样, 如 `pi()`)。

一个存储过程包括名字, 参数列表, 以及可以包括很多 SQL 语句的 SQL 语句集。在这里对局部变量, 异常处理, 循环控制和 IF 条件句有新的语法定义。

### 为什么要用存储过程

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

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

所以存储过程是可复用的组件! 想象一下如果你改变了主机的语言, 这对存储过程不会产生影响, 因为它是数据库逻辑而不是应用程序。存储过程是可以移植的! 当你用 SQL 编写存储过程时, 你就知道它可以运行在 Mysql 支持的任何平台上, 不需要你额外添加运行环境包, 也不需要为程序在操作系统中执行设置许可, 或者为你的不同型号的电脑存储过程将被保存! 如果你编写好了一个程序, 例如显示银行事物处理中的支票撤消, 那想要了解支票的人就可以找到你的程序。

它会以源代码的形式保存在数据库中。这将使数据和处理数据的进程有意义的关联这可能跟你在课上听到的规划论中说的一样。存储过程可以迁移!

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

在继续下面的学习之前,要确保 MySQL 5.0 已经安装。为了确认使用的 MySQL 的版本是正确的,我们要查询版本。我有两种方法确认我使用的是 5.0 版本:

```
SHOW VARIABLES LIKE 'version';
```

or

```
SELECT VERSION();
```

例如:

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.0.24a-community-nt |
+-----+
1 row in set (0.09 sec)
```

当看见数字'5.0.x' 后就可以确认存储过程能够在这个客户端上正常工作。

### 存储过程入门

在进行存储过程的练习前,需要有一张较简单的表,在这里就用前面创建位于 mysqldemo 中的 student 表。因为在这里并不需要展示查询数据的技巧,而是教授存储过程,不需要使用大的数据表,因为它本身已经够复杂了。

这就是示例数据库,我们将从这个名字为 t 的只包含一列的表开始 Pick a Delimiter 选择分隔符 现在我们需要一个分隔符,实现这个步骤的 SQL 语句如下:

```
DELIMITER //
```

例如:

```
mysql> DELIMITER //
```

分隔符是你通知 mysql 客户端你已经完成输入一个 SQL 语句的字符或字符串符号。一直以来我们都使用分号“;”,但在存储过程中,这会产生不少问题,因为存储过程中有许多语句,所以每一个都需要一个分号因此你需要选择一个不太可能出现在你的语句或程序中的字符串作为分隔符。比如“//”、“\$\$”、“|”等。(分号)作为分隔符,创建一个简单的存储过程语句如下所示:

```
mysql> delimiter ;//
mysql> create procedure p1() select * from student;//
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

CREATE PROCEDURE 是关键词,p1 () 是过程名称,存储过程名对大小写不敏感,因此‘P1’和‘p1’是同一个名字,在同一个数据库中你将不能给两个存储过程取相同的名字。注意创建完了还是需要将结束符再改回来的。

建议使用格式如下所示:

```
delimiter $$
create procedure pro_name()
begin
----
end $$
delimiter ;
```

可以采取“数据库名.存储过程名”的方式来命名,存储过程名中可以包括空格符,其长度限制为 64 个字符,但注意不要使用 MySQL 内建函数的名字。

由于现在的数据库中还没有数据,所以然后执行:

```
insert into student values(1,'xmh','the demo is good');
```

这样表里就有了数据，再调用存储过程。

```
mysql> call p1(<);
+-----+-----+-----+
| id  | name | intro                |
+-----+-----+-----+
| 1   | xmh  | the demo is good    |
+-----+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> _
```

要注意，虽然参数列表为空，但仍然要保留空括号，这是必须的。存储过程的主体，是一般的 SQL 语句。例如此处的"SELECT \* FROM student;"，如果后面有语句结束符号 (//) 时可以不写这个分号。通常不会将 SELECT 语句用在存储过程中，这里只是为了演示。所以使用这样的语句，能在调用时更好的看出程序是否正常工作。

### 存储过程晋级

CREATE PROCEDURE 过程名 ([过程参数[,...]]) [特性 ...] 过程体

CREATE FUNCTION 函数名 ([函数参数[,...]]) RETURNS 返回类型 [特性 ...] 函数体

过程参数: [ IN | OUT | INOUT ] 参数名 参数类型

函数参数: 参数名 参数类型

返回类型: 有效的 MySQL 数据类型即可

特性: 内容较复杂，暂时不予以考虑。

过程体/函数体: 格式如下:

BEGIN

有效的 SQL 语句

END

IN 输入参数 表示该参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值

执行如下语句

insert into student values(2,'zyj','the second demo');

向数据库再插入一条数据。然后创建一个查询语句的存储过程。

```
mysql> delimiter $$
mysql> create procedure sp_demo_in_stu(IN s_in int)
-> begin
-> select * from student where id=s_in;
-> end $$
Query OK, 0 rows affected (0.06 sec)
```

调用存储过程

```
mysql> delimiter ;
mysql> call sp_demo_in_stu(1);
+-----+-----+-----+
| id   | name | intro                |
+-----+-----+-----+
| 1    | xmh  | the demo is good    |
+-----+-----+-----+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> call sp_demo_in_stu(2);
+-----+-----+-----+
| id   | name | intro                |
+-----+-----+-----+
| 2    | zyj  | the second demo     |
+-----+-----+-----+
1 row in set (0.00 sec)

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

OUT 输出参数 该值可在存储过程内部被改变, 并可返回

为了更好地演示效果, 这里请注意 select...into 的用法, 后续内容对此有相关解释。

```
mysql> delimiter $$
mysql> create procedure sp_demo_out_stu(out s_out varchar(50))
-> begin
-> select version() into s_out;
-> end $$
Query OK, 0 rows affected (0.00 sec)

mysql> delimiter ;
```

调用并显示结果的语句如下所示:

```
mysql> call sp_demo_out_stu(@a);
Query OK, 0 rows affected (0.00 sec)

mysql> select @a;
+-----+
| @a   |
+-----+
| 5.0.24a-community-nt |
+-----+
1 row in set (0.00 sec)
```

INOUT 输入输出参数 调用时指定, 并且可被改变和返回

```
mysql> delimiter $$
mysql> create procedure sp_demo_inout_stu(out s_out varchar(50), inout num int)
-> begin
-> select version() into s_out;
-> set num=num+1;
-> end $$
Query OK, 0 rows affected (0.00 sec)
```

设置参数, 调用存储过程结果如下所示:



```
mysql> delimiter ;
mysql> set @num=10;
Query OK, 0 rows affected (0.00 sec)

mysql> call sp_demo_inout_stu(@a,@num);
Query OK, 0 rows affected (0.03 sec)

mysql> select @a,@num;
+-----+-----+
| @a          | @num |
+-----+-----+
| 5.0.24a-community-nt | 11    |
+-----+-----+
1 row in set (0.00 sec)
```

查看存储过程或者函数

查看存储过程或者函数的状态:

Show {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']

```
mysql> show procedure status;
+-----+-----+-----+-----+-----+
| Db      | Name          | Type      | Definer      | Modified      |
| Created | Security_type | Comment    |              |              |
+-----+-----+-----+-----+-----+
| mysqldemo | p1            | PROCEDURE | root@localhost | 2008-05-26 10:16:33 |
| 2008-05-26 10:16:33 | DEFINER      |           |              |              |
| mysqldemo | sp_demo_inout_stu | PROCEDURE | root@localhost | 2008-05-26 14:23:14 |
| 2008-05-26 14:23:14 | DEFINER      |           |              |              |
| mysqldemo | sp_demo_in_stu   | PROCEDURE | root@localhost | 2008-05-26 13:21:06 |
| 2008-05-26 13:21:06 | DEFINER      |           |              |              |
| mysqldemo | sp_demo_out_stu  | PROCEDURE | root@localhost | 2008-05-26 14:08:57 |
| 2008-05-26 14:08:57 | DEFINER      |           |              |              |
+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

查看存储过程或者函数的定义

Show CREATE {PROCEDURE | FUNCTION} sp\_name

```
mysql> show create procedure p1 \G;
***** 1. row *****
      Procedure: p1
      sql_mode: NO_AUTO_CREATE_USER
Create Procedure: CREATE DEFINER='root'@'localhost' PROCEDURE `p1`()
select * from student
1 row in set (0.00 sec)

ERROR:
No query specified
```

通过查看 information\_schema.Routines 了解存储过程和函数的信息

要用到 INFORMATION\_SCHEMA 库中的 ROUTINES 表, 此表提供了关于存储子程序(存储程序和函数)的信息。此时, ROUTINES 表不包含自定义函数(UDF)。

```
mysql> select * from routines where ROUTINE_NAME = 'p1' \G ;
***** 1. row *****
SPECIFIC_NAME: p1
ROUTINE_CATALOG: NULL
ROUTINE_SCHEMA: mysqldemo
ROUTINE_NAME: p1
ROUTINE_TYPE: PROCEDURE
DTD_IDENTIFIER: NULL
ROUTINE_BODY: SQL
ROUTINE_DEFINITION: select * from student
EXTERNAL_NAME: NULL
EXTERNAL_LANGUAGE: NULL
PARAMETER_STYLE: SQL
IS_DETERMINISTIC: NO
SQL_DATA_ACCESS: CONTAINS SQL
SQL_PATH: NULL
SECURITY_TYPE: DEFINER
CREATED: 2008-05-26 10:16:33
LAST_ALTERED: 2008-05-26 10:16:33
SQL_MODE: NO_AUTO_CREATE_USER
ROUTINE_COMMENT:
DEFINER: root@localhost
1 row in set (0.01 sec)

ERROR:
No query specified
```

删除存储过程:

DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp\_name

```
mysql> use mysqldemo;
Database changed
mysql> drop procedure p1;
Query OK, 0 rows affected (0.01 sec)
```

删除的时候要确保在相应的数据库中, 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。

### 变量的使用

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 之中, 也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, 并且在任何其它语句的前面。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。

SET 变量名 = 表达式值 [,variable\_name = expression ...] , 也可以直接赋常量值。

```
mysql> delimiter $$
mysql> create procedure sp_declare_stu()
-> begin
-> declare a int;
-> declare b int;
-> set a=3;
-> set b=1;
-> insert into student values(a,'zyj','the third demo');
-> select * from student where id>b;
-> end $$
Query OK, 0 rows affected (0.00 sec)
```

执行这个存储过程的结果是:

```
mysql> delimiter ;
mysql> call sp_declare_stu(<>);
+-----+-----+
| id  | name | intro          |
+-----+-----+
| 2   | zyj  | the second demo |
| 3   | zyj  | the third demo  |
+-----+-----+
2 rows in set (0.06 sec)

Query OK, 0 rows affected (0.06 sec)
```

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

### 用户变量

- 1、在 mysql 客户端使用用户变量  
用 into 进行赋值

```
mysql> select 'greeting' into @a;
Query OK, 1 row affected (0.00 sec)

mysql> select @a;
+-----+
| @a      |
+-----+
| greeting |
+-----+
1 row in set (0.00 sec)
```

用 “=” 直接赋值

set @b="good morning!!";

```
mysql> select @b;
+-----+
| @b      |
+-----+
| good morning!! |
+-----+
1 row in set (0.00 sec)
```

可以用表达式赋值

```
mysql> set @c=5+1+7;
Query OK, 0 rows affected (0.00 sec)

mysql> select @c;
+-----+
| @c      |
+-----+
| 13      |
+-----+
1 row in set (0.00 sec)
```

- 2、在存储过程中使用用户变量

```
mysql> delimiter $$
mysql> create procedure helloworld()
-> begin
-> select concat(@hello,'BeiJing');
-> end $$
Query OK, 0 rows affected (0.00 sec)
```

运行后的结果为:

```
mysql> delimiter ;
mysql> set @hello='Welcome to ';
Query OK, 0 rows affected (0.00 sec)

mysql> call helloworld();
+-----+
| concat(@hello,'BeiJing') |
+-----+
| Welcome to BeiJing      |
+-----+
1 row in set (0.00 sec)

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

### 3、在存储过程间传递全局范围的用户变量

新建一个 P1 的存储过程，里面设置一个用户变量。

```
mysql> create procedure p1() set @last='p1';
Query OK, 0 rows affected (0.00 sec)
```

新建一个 P2 的存储过程，里面调用用户变量。

```
mysql> create procedure p2() select concat('welcome ',@last);
Query OK, 0 rows affected (0.00 sec)
```

调用 P1;

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

调用 P2;

```
mysql> call p2();
+-----+
| concat('welcome ',@last) |
+-----+
| welcome p1               |
+-----+
1 row in set (0.00 sec)

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

注意:

- ①用户变量名一般以@开头
- ②滥用用户变量会导致程序难以理解及管理

注释用法回顾

mysql 存储过程可使用两种风格的注释

双模杠: -- , #

该风格一般用于单行注释

风格: /\* 注释内容 \*/ 一般用于多行注释

## 流程控制

IF 语句:

If 语句的结构如下所示:

```
IF search_condition THEN statement_list
  [ELSEIF search_condition THEN statement_list] ...
  [ELSE statement_list]
END IF
```

IF 实现了一个基本的条件构造。如果 search\_condition 求值为真, 相应的 SQL 语句列表被执行。如果没有 search\_condition 匹配, 在 ELSE 子句里的语句列表被执行。statement\_list 可以包括一个或多个语句。

```
delimiter $$
create procedure ifdemo(in param int)
begin
  declare var1 int;
  set var1=param+1;
  if var1>5 then
    select * from student where name like 'z%';
  else
    select * from student where name like 'x%';
  end if;
end $$
delimiter ;
```

这里仍以 student 表为例, 如果用户传递的参数值经加 1 后大于 5, 则显示所有以“Z”开头的的数据, 反之显示所有以“X”开头的的数据。注意这里 end if 后要以分号结束。

查询操作如下:

```
mysql> call ifdemo(3);
+-----+-----+-----+-----+
| id  | name  | intro  | score |
+-----+-----+-----+-----+
| 6   | xmh11 | no intro | 60    |
| 6   | xmh12 | no intro | 60    |
| 8   | xmh16 | no intro | 60    |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)

mysql> call ifdemo(5);
+-----+-----+-----+-----+
| id  | name      | intro  | score |
+-----+-----+-----+-----+
| 5   | zxx       | NULL   | 85    |
| 9   | zxyyyyyy | no intro | 60    |
| 6   | zyjl1     | no intro | 60    |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)
```

## CASE 语句

实现比 IF 更复杂一些的条件构造, 用 CASE 语句对 IF 条件进行更换, 具体操作如下所示。

```
mysql> delimiter $$
mysql> create procedure casedemo(in param int)
-> begin
-> declare var1 int;
-> set var1=param+1;
-> case param
-> when 0 then select * from student where id=var1;
-> when 1 then select * from student where id=var1;
-> when 2 then select * from student where id=var1;
-> when 3 then select * from student where id=1;
-> end case;
-> end $$
Query OK, 0 rows affected (0.00 sec)
```

### LOOP 语句

LOOP 实现简单的循环, 在循环内的语句一直重复直循环被退出, 退出通常伴随着一个 LEAVE 语句。LEAVE 语句被用来退出任何被标注的流程控制构造。它和 BEGIN ... END 或循环一起被使用。

```
mysql> delimiter $$
mysql> create procedure loopedemo(in param int)
-> begin
-> set @i=0;
-> ins:loop
-> set @i = @i+1;
-> if @i>3 then
-> leave ins;
-> end if;
-> select * from student where id=@i;
-> end loop ins;
-> end $$
Query OK, 0 rows affected (0.00 sec)
```

### ITERATE 语句

ITERATE 只可以出现在 LOOP, REPEAT, 和 WHILE 语句内。ITERATE 意思为: “再次循环。”

验证角谷猜想: 给定一个整数  $x$ , 若  $x \% 2 = 1$ , 则  $x = 3 * x + 1$ , 否则  $x = x / 2$ , 如此循环下去, 经过有限步骤必能得到 1。

例如: 初始整数为 9, 则 9->28->14->7->22->11->.....->10->5->16->8->4->2->1

```
mysql> delimiter $$
mysql> create procedure iteratedemo(in number int)
-> begin
-> declare param1 int default 1;
-> set @a=concat(number);
-> xmh:loop
-> set param1=number%2;
-> if param1=1 then set number=number*3+1;
-> else set number=number/2;
-> end if;
-> set @a=concat(@a,'--->',number);
-> if number>1 then iterate xmh;
-> end if;
-> leave xmh;
-> end loop xmh;
-> end $$;
Query OK, 0 rows affected (0.06 sec)
```

执行的结果如下所示:

```
mysql> delimiter ;
mysql> call iteratedemo(11);
Query OK, 0 rows affected (0.00 sec)

mysql> select @a;
+-----+
----+
| @a
|
+-----+
----+
| 11--->34--->17--->52--->26--->13--->40--->20--->10--->5--->16--->8--->4--->2--->1 |
+-----+
1 row in set (0.00 sec)
```

### 异常处理

对于 MySQL 的异常处理，虽然不常用。但是还是有必要写下来。

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

这个语句指定每个可以处理一个或多个条件的处理程序。如果产生一个或多个条件，指定的语句被执行。

对于一个 CONTINUE 处理程序，当前子程序的执行在执行处理程序语句之后继续。对于 EXIT 处理程序，当前 BEGIN...END 复合语句的执行被终止。UNDO 处理程序类型语句还不被支持。

- 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。

除了 SQLSTATE 值，MySQL 错误代码也可以被支持。

设置 id 键为主键，添加一条数据（与原来的数据有重复时），会提示主键值重复，如下所示。

```
mysql> INSERT INTO student VALUES (1,'1','1'); $$
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

通过以上信息可以得到 sqlstate 值 23000，下面写一个处理机制，如果错误为 23000，则继续执行后面语句。



```
mysql> delimiter $$
mysql> CREATE PROCEDURE handlerdemo (<)
-> BEGIN
-> DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
-> SET @x = 1;
-> INSERT INTO student VALUES (1,'1','1');
-> SET @x = 2;
-> INSERT INTO student VALUES (1,'1','1');
-> SET @x = 3;
-> END;
-> $$
Query OK, 0 rows affected (0.00 sec)
```

当然也可以用错误的代码 1062 来进行处理。

```
mysql> CREATE PROCEDURE handlerdemo (<)
-> BEGIN
-> DECLARE CONTINUE HANDLER FOR 1062 SET @x2 = 1;
-> SET @x = 1;
-> INSERT INTO student VALUES (1,'1','1');
-> SET @x = 2;
-> INSERT INTO student VALUES (1,'1','1');
-> SET @x = 3;
-> END;
-> $$
Query OK, 0 rows affected (0.00 sec)
```

调用存储过程，执行 select @x 得到的值为 3。

```
mysql> delimiter $$
mysql> CREATE PROCEDURE handlerdemo (<)
-> BEGIN
-> DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET @x2 = 1;
-> SET @x = 1;
-> INSERT INTO student VALUES (1,'1','1');
-> SET @x = 2;
-> INSERT INTO student VALUES (1,'1','1');
-> SET @x = 3;
-> END;
-> $$
Query OK, 0 rows affected (0.00 sec)
```

调用存储过程，执行 select @x 得到的值为 1。

在 java 中如何调用存储过程？

因为涉及到连接数据，所以写一个公用类，提供一个获取 Connection 的公用方法，本章节是假设读者具备一定的 JAVA JDBC 基础知识为前提条件的，如果不熟悉 JDBC，请先学习 JDBC 的相关用法。

```
/*
 * Created on 2008-6-06
 */

import java.io.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

import java.util.Properties;
```

```
/**
 * Connection Factory.
 *
 * @author xmh
 * @version 1.0
 */
public class ConnectionFactory
{
    /** database driver class name */
    private String driver = "";

    /** database URL associated with the URL */
    private String dbURL = "";

    /** user name of the database */
    private String user = "";

    /** password for the current user */
    private String password = "";

    /** factory instance */
    private static ConnectionFactory    factory    = null;

    /**
     * constructor
     *
     * @throws Exception
     */
    private ConnectionFactory() throws Exception
    {
        driver = "com.mysql.jdbc.Driver";
        dbURL =
"jdbc:mysql://localhost:3306/mysqlDemo?useUnicode=true&characterEncoding=UTF-8";
        user = "root";
        password = "****";
    }

    /**
     * @return database url
     */
    public String getDbURL()
    {
        return dbURL;
    }

    /**
     * @return database driver class name
     */
    public String getDriver()
    {
        return driver;
    }
}
```

```
* @return password of the current user
*/
public String getPassword()
{
    return password;
}

/**
 * @return the database user
 */
public String getUser()
{
    return user;
}

public static Connection getConnection()
{
    Connection conn = null;

    if (factory == null)
    {
        try
        {
            factory = new ConnectionFactory();
        } catch (Exception e)
        {
            System.out.println(e.getMessage());
            e.printStackTrace();
            return null;
        }
    }

    try
    {
        Class.forName(factory.getDriver());
        conn = DriverManager.getConnection(factory.getDbURL(), factory.getUser(),
factory.getPassword());
    } catch (ClassNotFoundException e)
    {
        System.out.println(" No class " + factory.getDriver() + " found error");
        e.printStackTrace();
    } catch (SQLException e)
    {
        System.out.println("Failed to get connection :" + e.getMessage());
        e.printStackTrace();
    }

    return conn;
}
}
```

原始表如下所示:

```
mysql> select * from student;
+-----+-----+-----+-----+
| id    | name  | intro  | score |
+-----+-----+-----+-----+
| 5     | zxx   | NULL   | 85    |
| 6     | xmh11 | no intro | 60    |
| 6     | xmh12 | no intro | 60    |
| 8     | xmh16 | no intro | 60    |
| 9     | xmh17 | no intro | 60    |
| 6     | zyj11 | no intro | 60    |
+-----+-----+-----+-----+
6 rows in set (0.09 sec)
```

### 无返回值的存储过程

新建一个存储过程，执行插入数据命令。存储过程语句如下所示：

```
delimiter $$
create procedure addstu(IN p1 int,IN p2 varchar(30))
begin
insert into student values(p1,p2,NULL,80);
end $$
```

然后呢，在 java 里调用时就用下面的代码：

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestArr {

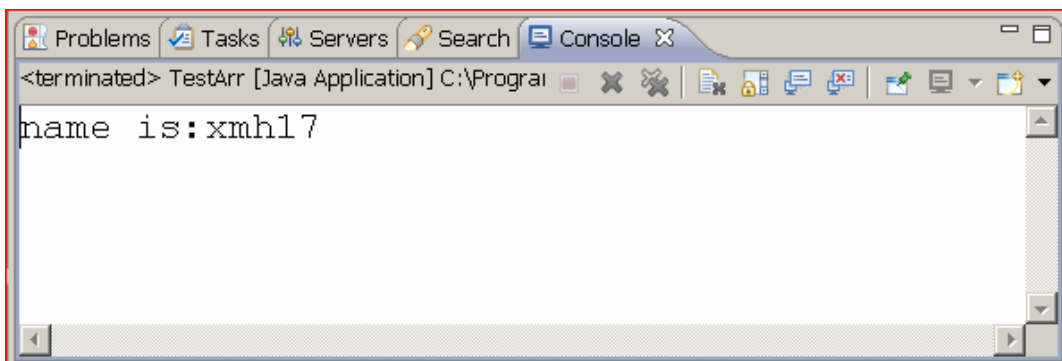
    public static void main(String[] args) {
        Statement stmt = null;
        ResultSet rs = null;
        Connection conn = null;
        try {
            conn = ConnectionFactory.getConnection();
            System.out.println(conn.toString());
            CallableStatement proc = null;
            proc = conn.prepareCall("{ call mysqldemo.addstu(?,?) }");
            proc.setString(1, "17");
            proc.setString(2, "TestProc");
            proc.execute();
        } catch (SQLException ex2) {
            ex2.printStackTrace();
        } catch (Exception ex2) {
            ex2.printStackTrace();
        } finally {
            try {
                if (rs != null) {
                    rs.close();
                }
                if (stmt != null) {
                    stmt.close();
                }
            }
            if (conn != null) {
```



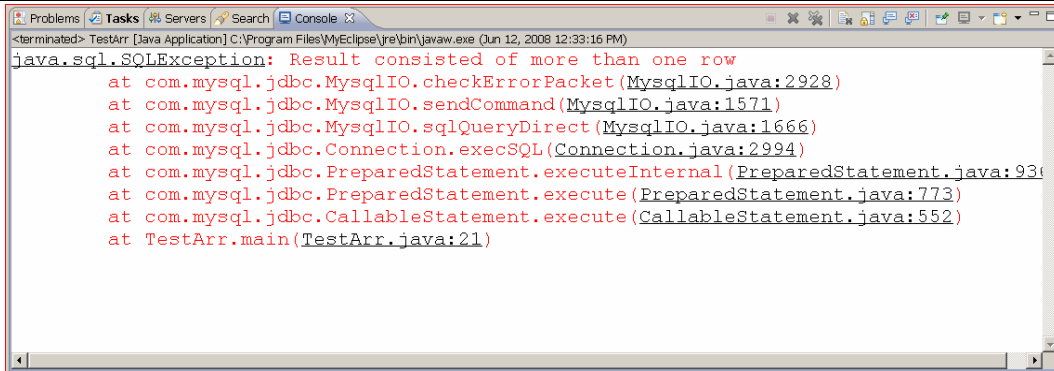
```
public static void main(String[] args) {
    Statement stmt = null;
    ResultSet rs = null;
    Connection conn = null;
    try {
        conn = ConnectionFactory.getConnection();
        System.out.println("conn:" + conn.toString());
        CallableStatement proc = null;
        proc = conn.prepareCall("{ call mysqldemo.queOne(?,?)");
        proc.setString(1, "9");
        proc.registerOutParameter(2, Types.VARCHAR);
        proc.execute();
        String name = proc.getString(2);
        System.out.println("name is:" + name);

    } catch (SQLException ex2) {
        ex2.printStackTrace();
    } catch (Exception ex2) {
        ex2.printStackTrace();
    } finally {
        try {
            if (rs != null) {
                rs.close();
            }
            if (stmt != null) {
                stmt.close();
            }
            if (conn != null) {
                conn.close();
            }
        }
    } catch (SQLException ex1) {
    }
}
}
```

注意，这里的 `proc.getString(2)` 中的数值 2 并非任意的，而是和存储过程中的 out 列对应的，如果 out 是在第一个位置，那就是 `proc.getString(1)`，如果是第三个位置，就是 `proc.getString(3)`，当然也可以同时有多个返回值，那就是再加几个 out 参数了。



对于查询的结果来讲，只能有一行，比如查询 id 号为 6 的将抛出异常。



```
<terminated> TestArr [Java Application] C:\Program Files\MyEclipse\jre\bin\javaw.exe (Jun 12, 2008 12:33:16 PM)
java.sql.SQLException: Result consisted of more than one row
    at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2928)
    at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1571)
    at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1666)
    at com.mysql.jdbc.Connection.execSQL(Connection.java:2994)
    at com.mysql.jdbc.PreparedStatement.executeInternal(PreparedStatement.java:936)
    at com.mysql.jdbc.PreparedStatement.execute(PreparedStatement.java:773)
    at com.mysql.jdbc.CallableStatement.execute(CallableStatement.java:552)
    at TestArr.main(TestArr.java:21)
```

### 有返回值的存储过程（列表）

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

存储过程为：

```
CREATE PROCEDURE sp_st(in_st_id INT)
BEGIN
    SELECT name,intro FROM student WHERE id=in_st_id;
END $$
```

JAVA 的调用过程是：

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestCallableStatement {

    String driver = "com.mysql.jdbc.Driver";
    String dbURL =
        "jdbc:mysql://localhost:3306/mysqlDemo?useUnicode=true&characterEncoding=UTF-8";
    String user = "root";
    String password = "*****";

    /**
     * @param args
     */
    public static void main(String[] args) {

        Statement stmt = null;
        ResultSet rs = null;
        Connection conn = null;
        TestCallableStatement tc = new TestCallableStatement();
        try {
            conn = ConnectionFactory.getConnection();
            tc.empsInDept(conn, 6);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```



```
        } finally {
            try {
                if (rs != null) {
                    rs.close();
                }
                if (stmt != null) {
                    stmt.close();
                }
                if (conn != null) {
                    conn.close();
                }
            }
        } catch (SQLException ex1) {
        }
    }
}

private void empsInDept(Connection myConnect, int id) throws SQLException {
    CallableStatement cStmt = myConnect
        .prepareCall("{CALL mysqldemo.sp_st(?)}");
    cStmt.setInt(1, id);
    cStmt.execute();
    ResultSet rs1 = cStmt.getResultSet();
    while (rs1.next()) {
        System.out.println(rs1.getString("name") + " "
            + rs1.getString("intro"));
    }
    rs1.close();
    /* process second result set */
    if (cStmt.getMoreResults()) {
        ResultSet rs2 = cStmt.getResultSet();
        while (rs2.next()) {
            System.out.println(rs2.getInt(1) + " " + rs2.getString(2) + " "
                + rs2.getString(3));
        }
        rs2.close();
    }
    cStmt.close();
}
}
```

## 第十四章

触发器的概念:“在数据库中为响应一个特殊表格中的某些事件而自动执行的程序代码。”(Wikipedia)

说得简单一些，它是在一个特殊的数据库事件，如 INSERT 或 DELETE 发生时，自动激活的一段代码。触发器可方便地用于日志记录、对单个表格到其他链接式表格进行自动的“层叠式”更改、或保证对表格关系进行自动更新。当一个新整数值增加到数据库域中时，自动更新运行的总数的代码段是一个触发器。自动记录对一个特殊数据库表格所作更改的 SQL 命令块也是一个触发器实例。

创建触发器的语法如下图所示：

```
CREATE TRIGGER <触发器名称>
{ BEFORE | AFTER }
{ INSERT | UPDATE | DELETE }
ON <表名称>
FOR EACH ROW
<触发的SQL语句>
```

触发器是 MySQL 5.x 的新功能，随着 5.x 代码树新版本的出现，这一功能也逐渐得到改善。在本文中，我将简单介绍如何定义并使用触发器，查看触发器状态，并如何在使用完毕后删除触发器。我还将为你展示一个触发器在现实世界中的应用实例，并检验它对数据库记录的改变。

例子

通过简单实例来说明是了解 MySQL 触发器应用的最佳办法。首先我们建立两个单域的表格。一个表格中为姓名列表（表格名：data），另一个表格中是所插入字符的字符数（表格名：chars）。我希望在 data 表格中定义一个触发器，每次在其中插入一个新姓名时，chars 表格中运行的总数就会根据新插入记录的字符数目进行自动更新。

```
CREATE TABLE chars (count INT(10));
CREATE TABLE data (name VARCHAR(255));
INSERT INTO chars (count) VALUES (0);
CREATE TRIGGER t1 AFTER INSERT ON
data FOR EACH ROW UPDATE chars SET count= count + CHAR_LENGTH(NEW.name);
```

理解上面代码的关键在于 CREATE TRIGGER 命令，它被用来定义一个新触发器。这个命令建立一个新触发器，假定的名称为 t1，每次有一个新记录插入到 data 表格中时，t1 就被激活。

在这个触发器中有两个重要的子句：

AFTER INSERT 子句表明触发器在新记录插入 data 表格后激活。

UPDATE chars SET count = count + CHAR\_LENGTH(NEW.name) 子句表示触发器激活后执行的 SQL 命令。在本例中，该命令表明用新插入的 data.name 域的字符数来更新 chars.count 栏。这一信息可通过内置的 MySQL 函数 CHAR\_LENGTH() 获得。

放在源表格域名前面的 NEW 关键字也值得注意。这个关键字表明触发器应考虑域的 new 值（也就是说，刚被插入到域中的值）。MySQL 还支持相应的 OLD 前缀，可用它来指域以前的值。

你可以通过调用 show triggers \G; 命令来检查触发器是否被激活。

```
mysql> show triggers \G;
***** 1. row *****
  Trigger: t1
    Event: INSERT
    Table: data
Statement: UPDATE chars SET count
= count + CHAR_LENGTH(NEW.name)
    Timing: AFTER
    Created: NULL
sql_mode: STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
Definer: root@localhost
***** 2. row *****
  Trigger: trg_del_org_user
    Event: DELETE
    Table: t_organization
Statement: BEGIN
          Delete FROM t_user Where t_user.org_id = OLD.org_id;
END
    Timing: BEFORE
    Created: NULL
sql_mode: STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
Definer: root@localhost
2 rows in set (0.05 sec)
```

激活触发器后, 开始对它进行测试。试着在 data 表格中插入几个记录:

```
INSERT INTO data (name) VALUES ('Sue'), ('Jane');
```

```
mysql> INSERT INTO data (name) VALUES ('Sue'), ('Jane');
Query OK, 2 rows affected (0.06 sec)
Records: 2 Duplicates: 0 Warnings: 0
```

然后检查 chars 表格看触发器是否完成它该完成的任务:

```
mysql> select * from chars;
+-----+
| count |
+-----+
|      ?      |
+-----+
1 row in set (0.00 sec)
```

触发器应用完毕后, 可有 DROP TRIGGER 命令轻松删除它。

```
mysql> drop trigger t1;
Query OK, 0 rows affected (0.00 sec)
```

注意: 理想情况下, 你还需要一个倒转触发器, 每当一个记录从源表格中删除时, 它从字符总数中减去记录的字符数。这很容易做到, 你可以把它当作练习来完成。提示: 应用 BEFORE DELETE ON 子句是其中一种方法。

现在, 我想建立一个审计记录来追踪对这个表格所做的改变。这个记录将反映表格的每项改变, 并向用户说明由谁做出改变以及改变的时间。我需要建立一个新表格来存储这一信息 (表格名: audit), 如下所示。

```
CREATE TABLE audit (id INT(7),
balance FLOAT, user VARCHAR(50)
NOT NULL, time TIMESTAMP NOT NULL);
```

接下来, 我将在 accounts 表格中定义一个触发器。

```
CREATE TRIGGER t1 AFTER UPDATE ON accounts
FOR EACH ROW INSERT INTO audit (id, balance, user, time)
VALUES (OLD.id, NEW.balance, CURRENT_USER(), NOW());
```

注释: accounts 表格每经历一次 UPDATE, 触发器插入 (INSERT) 对应记录的 id、新的余额、当前时间和登录 audit 表格的用户名称。

触发器是属于某一个表的: 当在这个表上执行插入、更新或删除操作的时候就导致触发器的激活, 我们不能给同一张表的同一个事件安排两个触发器。

## 第 十五 章

要使用 MySQL 中的事务处理, 首先需要创建使用事务表类型(如 BDB = Berkeley DB 或 InnoDB)的表。

MySQL 的数据表分为两类, 一类是传统的数据表, 另一类则是支持事务的数据表。支持事务的数据表分为两种: InnoDB 和 BerkeleyDB。我们可以通过以下的命令确认安装的 MySQL Server 是否支持这两种数据表:

```
SHOW VARIABLES LIKE 'have_innodb';
SHOW VARIABLES LIKE 'have_bdb';
```

即使 MySQL Server 本身是支持这两种数据表, 但是并不代表着创建新表的时候, 新表就是属于这两种的。通常, 在创建新表的时候, 如果没有加以特别的说明, 那么创建的新表则是传统的数据表, 是不会支持事务的。因此, 我的问题也就解决了, 因为以上的示例代码操作的是两张没有支持事务的数据表, 怎么可能去处理事务呢?

通常我们可以通过使用如下的方式来创建支持事务的数据表:

```
CREATE TABLE TABLE_NAME(FIELD1, FIELD2.... FIELDn) TYPE=INNODB;
```

如果原来建好的数据表, 我们也可以通过 ALTER TABLE 命令直接去改变数据表的类型, 而不需要备份数据, 然后 Drop 表, 再建立新表, 再导入数据这样烦琐的步骤。如:

```
ALTER TABLE TABLE_NAME TYPE=INNODB;
```

原始数据如下:

```
mysql> select * from student;
+----+-----+-----+-----+
| id  | name  | intro  | score |
+----+-----+-----+-----+
| 5   | zxx   | NULL   | 85    |
| 6   | xnh11 | no intro | 60    |
| 6   | xnh12 | no intro | 60    |
| 8   | xnh16 | no intro | 60    |
| 9   | xnh17 | no intro | 60    |
| 6   | zyj11 | no intro | 60    |
| 7   | zyj13 | no intro | 60    |
+----+-----+-----+-----+
7 rows in set (0.00 sec)
```

要在事务表上使用事务处理, 必须要首先关闭自动提交:

```
SET AUTOCOMMIT = 0;
```

事务处理以 BEGIN 命令开始:

BEGIN;

现在 mysql 客户处于服务器相关的事物上下文中。任何对事务表所做的改变在提交之前不会成为永久性的改变。

执行删除命令。

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> delete from student where id=7;
Query OK, 1 row affected (0.00 sec)

mysql> select * from student;
+----+-----+-----+-----+
| id  | name  | intro  | score |
+----+-----+-----+-----+
| 5   | zxx   | NULL   | 85    |
| 6   | xmh11 | no intro | 60    |
| 6   | xmh12 | no intro | 60    |
| 8   | xmh16 | no intro | 60    |
| 9   | xmh17 | no intro | 60    |
| 6   | zyjl1 | no intro | 60    |
+----+-----+-----+-----+
6 rows in set (0.00 sec)
```

若在提交前终止整个事务, 可以进行回滚操作:

执行回滚命令。

```
mysql> rollback;
Query OK, 0 rows affected (0.03 sec)

mysql> select * from student;
+----+-----+-----+-----+
| id  | name  | intro  | score |
+----+-----+-----+-----+
| 5   | zxx   | NULL   | 85    |
| 6   | xmh11 | no intro | 60    |
| 6   | xmh12 | no intro | 60    |
| 8   | xmh16 | no intro | 60    |
| 9   | xmh17 | no intro | 60    |
| 6   | zyjl1 | no intro | 60    |
| 7   | zyjl3 | no intro | 60    |
+----+-----+-----+-----+
7 rows in set (0.00 sec)
```

在做出所有的改变之后, 使用 COMMIT 命令完成事务处理:

执行提交命令。

```
mysql> delete from student where id=7;
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.03 sec)

mysql> select * from student;
+----+-----+-----+-----+
| id  | name  | intro  | score |
+----+-----+-----+-----+
| 5   | zxx   | NULL   | 85    |
| 6   | xmh11 | no intro | 60    |
| 6   | xmh12 | no intro | 60    |
| 8   | xmh16 | no intro | 60    |
| 9   | xmh17 | no intro | 60    |
| 6   | zyjl1 | no intro | 60    |
+----+-----+-----+-----+
6 rows in set (0.00 sec)
```

除了 COMMIT 命令外，下列命令也会自动结束当前事务：

ALTER TABLE

BEGIN

CREATE INDEX

DROP DATABASE

DROP TABLE

LOCK TABLES

RENAME TABLE

TRUNCATE

UNLOCK TABLES

## 第 十六 章

## 第 十七 章