数据库（database） 保存有组织的数据的容器（通常是一个文 件或一组文件）

表（table） 某种特定类型数据的结构化清单

模式（schema） 关于数据库和表的布局及特性的信息

列（column） 表中的一个字段。所有表都是由一个或多个列组 成的。

行（row） 表中的一个记录

主键（primary key）①一一列（或一组列），其值能够唯一区分表中每个行

> □ 任意两行都不具有相同的主键值；
>
> □ 每个行都必须具有一个主键值（主键列不允许`NULL`值）。
>
> 主键通常定义在表的一列上，但这并不是必需的，也可以一起使用 多个列作为主键。
>
> 好习惯：
>
> □ 不更新主键列中的值；
>
> □ 不重用主键列的值；
>
> □ 不在主键列中使用可能会更改的值。（例如，如果使用一个 名字作为主键以标识某个供应商，当该供应商合并和更改其 名字时，必须更改这个主键。）

> 命令用;或\g结束，换句话说，仅按Enter不执行命令；
>
> 输入quit或exit退出命令行实用程序

# 基础

SQL语句不区分大小写

## SHOW

`SHOW DATABASES;` 返回可用数据库的一个列表。包含在这个列 表中的可能是MySQL内部使用的数据库

`SHOW TABLES;` 返回当前选择的数据库内可用表的列表。

`SHOW COLUMNS` 要求给出一个表名，它对每个字段返回一行，行中包含字段名、数据类型、是否允许NULL、键信息、默认值以及其他信息（如字段cust_id 的auto_increment）。

## SELECT

### 检索单个列

所需的列名在SELECT关键字之后给出，FROM 关键字指出从其中检索数据的表名。

```
SELECT prod_id FROM products;
```

### 检索多个列

在SELECT关键字后给出多个列名，列名之间必须以逗号分隔

```
SELECT prod_id, prod_name, prod_price FROM products;
```

### 检索所有列

如果给定一个通配符（`*`），则返回表中所有列

```
SELECT * FROM products;
```

但检索不需要的列通常会降低检索和应用程序的性能

使用通配符有一个大优点。由于不明确指定列名（因为星号检索每个列），所以能检索出名字未知的列。

### 检索不同的行

`DISTINCT`此关键字指示MySQL 只返回不同的值

```
SELECT DISTINCT vend_id FROM products;
```

如果使用 `DISTINCT`关键字，它必须直接放在列名的前面

`DISTINCT`关键字应用于所有列而 不仅是前置它的列。除非指定的两个列都不同，否则所有行都将被检索出来

### 限制结果

为了返回第一行或前几行，可使用LIMIT子句。

`LIMIT 5, 5`指示MySQL返回从行5开始的5行。第一个数为开始 位置，第二个数为要检索的行数。

> **检索出来的第一行为行0而不是行1。因此，LIMIT 1, 1 将检索出第二行而不是第一行**
>
> **如果没有足够的行（例如，给出LIMIT 10, 5，但只有13 行），MySQL将只返回它能返回的那么多行。**
>
> **MYSQL5 -> LIMIT 4 OFFSET 3意为从行3开始取4行，就像LIMIT 3, 4一样**

### 完全限定符

列名：表名`.`列名，如 `products.vend_id`

表名：数据库名`.`表名，如 `example.products`

## ORDER BY

检索出的数据并不是以纯粹的随机顺序显示的。如果不排序，数据一般将以它在底层表中出现的顺序显示。这可以是数据最初添加到表中的顺序。但是，如果数据后来进行过更新或删除，则此顺序将会受到MySQL重用回收存储空间的影响。

子句：SQL语句由子句构成，有些子句是必需的，而有的是可选的。一个子句通常由一个关键字和所提供的数据组成。子句的例子有SELECT语句的FROM子句

**排序数据**

```
SELECT prod_name FROM products ORDER BY prod_name;
```

以**字母顺序排序**数据的`ORDER BY`子句

`ORDER BY`子句中使用的列将是为显示所选择的列。用非检索的列排序数据是完全合法的

**按多个列排序**

```
SELECT prod_id, prod_price, prod_name FROM products ORDER BY prod_price,
prod_name;
```

**在按多个列排序时，排序完全按所规定的顺序进行**

**指定排序方向**

`按prod_price降序`

```
SELECT prod_id, prod_price, prod_name FROM products ORDER BY prod_price DESC,
prod_name;
```

`DESC`关键字只应用到直接位于其前面的列名。

`DESC`降序，`ASC`升序（默认）

> **在多个列上降序排序** 如果想在多个列上进行降序排序，必须对每个列指定DESC关键字。
>
> 在字典（dictionary）排序顺序中，**A被视为与a相同**，这是MySQL （和大多数数据库管理系统）的默认行为。

**ORDER BY 和 LIMIT 结合**

```
SELECT prod_price FROM products ORDER BY prod_price DESC LIMIT 1;
+------------+
| prod_price |
+------------+
|      55.00 |
+------------+
```

**在给出`ORDER BY`子句时，应该保证它位于`FROM`子句之后。如果使用`LIMIT`，它必须位于`ORDER BY`之后。**

## WHERE

> 在SELECT语句中，数据根据WHERE子句中指定的搜索条件进行过滤。WHERE子句在表名（FROM子句）之后给出
>
> **在同时使用ORDER BY和WHERE子句时，应该让ORDER BY位于WHERE之后**

表6-1　WHERE子句操作符

| 操 作 符 | 说　明 |
| --- | --- |
| = | 等于 |
| <> | 不等于 |
| != | 不等于 |
| < | 小于 |
| <= | 小于等于 |
| > | 大于 |
| >= | 大于等于 |
| BETWEEN | 在指定的两个值之间 |

在使用 `BETWEEN` 时，必须指定两个值——所需范围的**低端值**和**高端值**。这两个值必须用 `AND` 关键字分隔。

`BETWEEN` 匹配范围中所有的值，**包括指定的开始值和结束值**

### 空值检查

在创建表时，表设计人员可以指定其中的列是否可以不包含值。在一个列不包含值时，称其为包含空值NULL。

> NULL 无值（no value），它与字段包含0、空字符串或仅仅包含 空格不同。

- SELECT语句有一个特殊的WHERE子句，可用来检查具有NULL值的列。 这个WHERE子句就是 `IS NULL` 子句。

```
SELECT cust_id FROM customers WHERE cust_email IS NULL;
```

- WHERE子句还可以 `IS NOT NULL` 筛选出特定列不是空的

```
SELECT cust_id FROM customers WHERE cust_email IS NOT NULL;
```

> **通过过滤选择出不具有特定值的行时，你可能希望返回具有NULL值的行。但是，不行。因为未知具有特殊的含义，数据库不知道它们是否匹配，所以在匹配过滤 或不匹配过滤时不返回它们。**

### AND 和 OR

MYSQL还可以多个WHERE子句。这些子句可以两种方式使用：以 `AND` 子句的方式或 `OR` 子句的方式使用。

- `AND` 用在 `WHERE` 子句中的关键字，用来指示检索满足所有给定条件的行

- `OR` 用在 `WHERE` 子句中的关键字，用来表示检索匹配任一给定条件的行
- `AND` 优先级比 `OR` 高， 用 () 分组操作符

### IN

- `IN` 操作符用来指定条件范围，范围中的每个条件都可以进行匹配。`IN` 取合法值的由逗号分隔的清单，全都括在圆括号中。

> - 在使用长的合法选项清单时，`IN` 操作符的语法更清楚且更直观。
> - 在使用IN时，计算的次序更容易管理（因为使用的操作符更少）。
> - `IN` 操作符一般比 `OR` 操作符清单执行更快。
> - `IN` 的最大优点是可以包含其他 `SELECT` 语句，使得能够更动态地建立 `WHERE` 子句。

`SELECT prod_name, prod_price FROM products WHERE vend_id IN(1002, 1003) ORDER BY prod_name;`

等于

`SELECT prod_name, prod_price FROM products WHERE vend_id = 1002 OR vend_id = 1003 ORDER BY prod_name;`

### NOT

- `NOT` WHERE子句中用来否定后跟条件的关键字。

MySQL中的 `NOT` MySQL支持使用 `NOT` 对IN、BETWEEN和 EXISTS子句取反

## LIKE

- LIKE指示MySQL， 后跟的搜索模式利用通配符匹配而不是直接相等匹配进行比较

> 通配符（wildcard）　　用来匹配值的一部分的特殊字符。
>
> 搜索模式（search pattern）　　由字面值、通配符或两者组合构成的搜索条件

### %

- %表示任何字符出现任意次数

```
SELECT prod_id, prod_name FROM products WHERE prod_name LIKE 'jet%';
```

在执行这条子句时，将检索任 意以 jet 起头的词

> 根据MySQL的配置方式，**搜索可以是区分大小写的**。如果区分大小写，'jet%'与JetPack 1000将不匹配。

```
SELECT prod_id, prod_name FROM products WHERE prod_name LIKE '%anvil%';
```

搜索模式 `'%anvil%'` 表示匹配任何位置包含文本anvil的值，而不论它之前或之后出现什么字符

```
SELECT prod_id, prod_name FROM products WHERE prod_name LIKE 's%e';
```

找出以 s 起头以 e 结尾的所有产品

> **尾空格可能会干扰通配符匹配。**
>
> **即使是`WHERE prod_name LIKE '%'`也不能匹配用值NULL作为产品名的行**

_

- 下划线 `_` 只匹配单个字符而不是多个字符

```
SELECT prod_id, prod_name FROM products WHERE prod_name LIKE '_ ton anvil';
```

> - 通配符搜索的处理一般要比前面讨论的其他搜索所花时间更长
> - 不要过度使用通配符
> - 把通配符置于搜索模式的开始处，搜索起来是最慢的
> - 仔细注意通配符的位置

## 正则表达式

使用 `REGEXP` 来使用正则表达式

`.`

- `.` 它表示匹配任意一个字符

#### LIKE和REGEXP的区别

```
mysql> SELECT prod_name FROM products WHERE prod_name LIKE '1000';
Empty set (0.00 sec)

mysql> SELECT prod_name FROM products WHERE prod_name REGEXP '1000';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
+--------------+
1 row in set (0.00 sec)
```

`LIKE` **匹配整个列。**如果被匹配的文本在列值中出现，`LIKE`将不会找到它，相应的行也不被返回（除非使用通配符）。

而`REGEXP`**在列值内进行匹配**，如果被匹配的文本在列值中出现，`REGEXP`将会找到它，相应的行将被返回。这是一个非常重要的差别。

> MySQL中的正则表达式匹配（自版本 3.23.4后）**不区分大小写**（即，大写和小写都匹配）。为区分大小写，可使用BINARY关键字

|

- 为搜索两个串之一（或者为这个串，或者为另一个串），使用 `|`

```
SELECT prod_name FROM products WHERE prod_name REGEXP '1000|2000';
```

[]

- 匹配任何单一字符
- `[123]` 定义一组字符，它的意思是匹配1或2或3
- 正则表达式 `[123]Ton` 为 `[1|2|3]Ton` 的缩写

**匹配范围( - )**

- 集合可用来定义要匹配的一个或多个字符。

`[0123456789]` 等于 `[0-9]`

**匹配特殊字符**

- 为了匹配特殊字符，必须用 `\\` 为前导。`\\-` 表示查找 `-`，`\\.` 表示查找 `.`。

\\也用来引用元字符（具有特殊含义的字符），如表9-1所列。

表9-1　空白元字符

| 元　字　符 | 说　　明 |
| --- | --- |
| \\f | 换页 |
| \\n | 换行 |
| \\r | 回车 |
| \\t | 制表 |
| \\v | 纵向制表 |

> - 为了匹配反斜杠（`\`）字符本身，需要使用 `\\\`。
> - 多数正则表达式实现使用单个反斜杠转义特殊字符，以便能使用这些字符本身。但MySQL要求两个反斜杠（MySQL 自己解释一个，正则表达式库解释另一个

**预定义的字符集**

表9-2　字符类

| 类 | 说　明 |
| --- | --- |
| [:alnum:] | 任意字母和数字（同[a-zA-Z0-9]） |
| [:alpha:] | 任意字符（同[a-zA-Z]） |
| [:blank:] | 空格和制表（同[\\t]） |
| [:cntrl:] | ASCII控制字符（ASCII 0到31和127） |
| [:digit:] | 任意数字（同[0-9]） |
| [:graph:] | 与[:print:]相同，但不包括空格 |
| [:lower:] | 任意小写字母（同[a-z]） |
| [:print:] | 任意可打印字符 |
| [:punct:] | 既不在[:alnum:]又不在[:cntrl:]中的任意字符 |
| [:space:] | 包括空格在内的任意空白字符（同[\\f\\n\\r\\t\\v]） |
| [:upper:] | 任意大写字母（同[A-Z]） |
| [:xdigit:] | 任意十六进制数字（同[a-fA-F0-9]） |

**匹配多个实例**

- 针对前一个字符的多次匹配

表9-3　重复元字符

| 元　字　符 | 说　明 |
| --- | --- |
| * | 0个或多个匹配 |
| + | 1个或多个匹配（等于{1,}） |
| ? | 0个或1个匹配（等于{0,1}） |
| {n} | 指定数目的匹配 |
| {n,} | 不少于指定数目的匹配 |
| {n,m} | 匹配数目的范围（m不超过255） |

```
SELECT prod_name FROM products WHERE prod_name REGEXP '\\([0-9] sticks?\\)';
```

`s` 匹配 0 次或 1 次。s 后的`?`使 s 可选，因为?匹配它前面的任何字符的 0 次或 1 次出现

```
SELECT prod_name FROM products WHERE prod_name REGEXP '[[:digit:]]{4}' ORDER BY
prod_name;
```

**匹配特定位置**

之前说的都是匹配任意位置的，如果要使用特定位置，则使用下面的定位符

表9-4　定位元字符

| 元　字　符 | 说　明 |
| --- | --- |
| ^ | 文本的开始 |
| $ | 文本的结尾 |
| [[:<:]] | 词的开始 |
| [[:>:]] | 词的结尾 |

```
SELECT prod_name FROM products WHERE prod_name REGEXP '^[0-9\\.]';
```

`^[0-9\\.]`只在`.`或任意数字为串中第一个字符时才匹配它们

> `^`**的双重用途** ^有两种用法。在集合中（用[和]定义），用它来否定该集合，否则，用来指串的开始处。
>
> `[^a-zA-Z0-9]`表示找到一个非字母也非数字的字符

前面说过，`LIKE`匹配整个串而`REGEXP`匹配子串。利用定位符，通过用`^`开始每个表达式，用`$`结束每个表达式，可以使`REGEXP`的作用与`LIKE`一样

**测试**：REGEXP检查总是返回0（没有匹配） 或1（匹配）。可以用带文字串的REGEXP来测试表达式，并试验它们。

```
SELECT 'hello' REGEXP '[0-9]';
```

返回0（因为文本hello中没有数字）

## 计算字段

### 拼接字段

- 计算字段是运行时在`SELECT`语句内创建的

> 字段：基本上与列（column）的意思相同，经常互换使用，不过数据库列一般称为列，而术语字段通常用在计算字段的连接上。
>
> 只有数据库知道SELECT语句中哪些列是实际的表列，哪些列是计算字段
>
> 可在SQL语句内完成的许多转换和格式化工作都可以直接在客户机应用程序内完成。但在数据库服务器上完成这些操作比在客户机中完成要快得多

- 拼接（concatenate） 将值联结到一起构成单个值。在MySQL的`SELECT`语句中，可使用`Concat()`函数来拼接两个列

  > 多数DBMS使用`+`或`||`来实现拼接， MySQL则使用`Concat()`函数来实现。

```
SELECT Concat(vend_name, ' (', vend_country, ')') FROM vendors ORDER BY
vend_name;
```

> Trim函数MySQL除了支持`RTrim()`（它去掉串右边的空格），
>
> 还支持`LTrim()`（去掉串左边的空格）以及
>
> `Trim()`（去掉串左右两边的空格）

- 别名（alias）是一个字段或值的替换名。别名用`AS`关键字赋予。

```
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country), ')') AS vend_title
FROM vendors ORDER BY vend_name;
```

> 别名（导出列 ）的作用: 任何客户机应用都可以按名引用 这个列，就像它是一个实际的表列一样。
>
> 常见的用途包括在实际的表列名包含不符合规定的字符（如空格）时重新命名它，在原来的名字含混或容易误解时扩充它，等等

## 执行算数计算

```
SELECT prod_id, quantity, item_price, quantity*item_price AS expanded_price FROM
orderitems WHERE order_num=20005;
```

输出中显示的`expanded_price`列为一个计算字段，此计算为`quantity*item_price`。

客户机应用现在可以使用这个新计算列，就像使用其他列一样

表10-1　MySQL算术操作符

| 操 作 符 | 说　　明 |
| --- | --- |
| + | 加 |
| - | 减 |
| * | 乘 |
| / | 除 |

### 圆括号可用来区分优先顺序

> 如何测试计算`SELECT`提供了测试和试验函数与计算的一个很好的办法。虽然`SELECT`通常用来从表中检索数据，但可以 省略`FROM`子句以便简单地访问和处理表达式。
>
> 例如，`SELECT 3*2;`将返回 6，`SELECT Trim('abc');`将返回abc，而`SELECT Now()`利用`Now()`函数返回当前日期和时间。通过这些例子， 可以明白如何根据需要使用`SELECT`进行试验

## 数据处理函数

- 函数一般是在**数据**上执行的，它给数据的转换和处理提供了方便。
- 函数移植性不强，应该写好注释

大多数SQL支持以下类型的函数

- 用于处理文本串（如删除或填充值，转换值为大写或小写）的文本函数。
- 用于在数值数据上进行算术操作（如返回绝对值，进行代数运算）的数值函数。
- 用于处理日期和时间值并从这些值中提取特定成分（例如，返回 两个日期之差，检查日期有效性等）的日期和时间函数。
- 返回DBMS正使用的特殊信息（如返回用户登录信息，检查版本 细节）的系统函数。

### 文本处理函数

Upper()将文本转换为大写

```
SELECT vend_name, Upper(vend_name) AS vend_name_upcase FROM vendors ORDER BY
vend_name;
```

表11-1　常用的文本处理函数

| 函　数 | 说　明 |
| --- | --- |
| Left() | 返回串左边的字符 |
| Length() | 返回串的长度 |
| Locate() | 找出串的一个子串 |
| Lower() | 将串转换为小写 |
| LTrim() | 去掉串左边的空格 |
| Right() | 返回串右边的字符 |

| 函　数 | 说　明 |
| --- | --- |
| RTrim() | 去掉串右边的空格 |
| Soundex() | 返回串的SOUNDEX值 |
| SubString() | 返回子串的字符 |
| Upper() | 将串转换为大写 |

- `SOUNDEX`是一个将任何文本串转换为描述其语音表示的字母数字模式的算法

```
mysql> SELECT cust_name, cust_contact FROM customers WHERE cust_contact = 'Y.Lie';
Empty set (0.00 sec)

mysql> SELECT cust_name, cust_contact FROM customers WHERE Soundex(cust_contact) = Soundex('Y.Lie');
+-------------+--------------+
| cust_name   | cust_contact |
+-------------+--------------+
| Coyote Inc. | Y Lee        |
+-------------+--------------+
1 row in set (0.01 sec)
```

**时间处理函数**

- 日期和时间采用相应的数据类型和特殊的格式存储，以便能快速和有效地排序或过滤，并且节省物理存储空间

表11-2　常用日期和时间处理函数

| 函　数 | 说　明 |
| --- | --- |
| AddDate() | 增加一个日期（天、周等） |
| AddTime() | 增加一个时间（时、分等） |
| CurDate() | 返回当前日期 |
| CurTime() | 返回当前时间 |
| Date() | 返回日期时间的日期部分 |
| DateDiff() | 计算两个日期之差 |
| Date_Add() | 高度灵活的日期运算函数 |
| Date_Format() | 返回一个格式化的日期或时间串 |
| Day() | 返回一个日期的天数部分 |
| DayOfWeek() | 对于一个日期，返回对应的星期几 |
| Hour() | 返回一个时间的小时部分 |
| Minute() | 返回一个时间的分钟部分 |
| Month() | 返回一个日期的月份部分 |
| Now() | 返回当前日期和时间 |
| Second() | 返回一个时间的秒部分 |
| Time() | 返回一个日期时间的时间部分 |
| Year() | 返回一个日期的年份部分 |

- **日期必须为格式yyyy-mm-dd**

```
SELECT cust_id, order_num FROM orders WHERE order_date = '2005-09-01'
```

`order_date`的数据类型为`datetime`。这种类型存储日期及时间值。样例表中的值全都具有时间值`00:00:00`，但实际中很可能并不总是这样。如果用当前日期和时间存储订单日期（因此你不仅知道订单日期，还知道下订单当天的时间）。

比如：存储的`order_date`值为`2005-09-01 11:30:05`，则`WHERE order_date = '2005-09-01'`失败。 即使给出具有该日期的一行，也不会把它检索出来，因为WHERE匹配失败。

解决办法是指示MySQL仅将**给出的日期与列中的日期部分进行比较，而不是将给出的日期与整个列值进行比较。**为此，必须使用`Date()`函数。`Date(order_date)`指示MySQL仅提取列的日期部分

> **如果要的是日期，请使用`Date()`**

```
SELECT cust_id, order_num FROM orders WHERE Date(order_date) BETWEEN '2005-09-
01' AND '2005-09-30';
#日期不对会warning，检索不出
或
SELECT cust_id, order_num FROM orders WHERE Year(order_date) = 2005 AND
Month(order_date) = 9;
```

查找2005年9月的所有订单

**数值处理函数**

表11-3　常用数值处理函数

| 函　数 | 说　明 |
| --- | --- |
| Abs() | 返回一个数的绝对值 |
| Cos() | 返回一个角度的余弦 |
| Exp() | 返回一个数的指数值 |
| Mod() | 返回除操作的余数 |
| Pi() | 返回圆周率 |
| Rand() | 返回一个随机数 |
| Sin() | 返回一个角度的正弦 |
| Sqrt() | 返回一个数的平方根 |
| Tan() | 返回一个角度的正切 |

## 汇总数据

**聚集函数**

- 确定表中行数（或者满足某个条件或包含某个特定值的行数）。
- 获得表中行组的和。
- 找出表列（或所有行或某些特定的行）的最大值、最小值和平均值

> **聚集函数（aggregate function） 运行在行组上，计算和返回单个值的函数。**

表12-1　SQL聚集函数

| 函　数 | 说　明 |
| --- | --- |
| AVG() | 返回某列的平均值 |
| COUNT() | 返回某列的行数 |
| MAX() | 返回某列的最大值 |
| MIN() | 返回某列的最小值 |
| SUM() | 返回某列值之和 |

`AVG`

- `AVG()` 通过对表中行数计数并计算特定列值之和，求得该列的平均值。`AVG()` 可用来返回所有列的平均值，也可以用来返回特定列或行的平均值。

```
SELECT AVG(prod_price) AS avg_price FROM products;
```

`AVG()` 返回 `products` 表中所有产品的平均价格

```
SELECT AVG(prod_price) AS avg_price FROM products WHERE vend_id = 1003;
```

返回特定供应商所提供产品的平均价格

> `AVG()` **只能用来确定特定数值列 的平均值，而且列名必须作为函数参数给出。**

> **为了获得多个列的平均值， 必须使用多个`AVG()`函数**
>
> - **`AVG()`函数忽略列值为`NULL`的行**

`COUNT`

- `COUNT()`函数进行计数

**两种使用方式**

- 使用`COUNT(*)`对表中行的数目进行计数，不管表列中包含的是空值（`NULL`）还是非空值。
- 使用`COUNT(column)`对特定列中具有值的行进行计数，忽略`NULL`值。

```
SELECT COUNT(*) AS num_cust FROM customers;
```

返回`customers`表中客户的总数

```
SELECT COUNT(cust_email) AS num_cust FROM customers;
```

对具有电子邮件地址的客户计数

> **如果指定列名，则指定列的值为空的行被`COUNT()`函数忽略，但如果`COUNT()`函数中用的是星号（`*`），则不忽略**

`MAX`

- `MAX()`返回指定列中的最大值。`MAX()`要求指定列名

```
SELECT MAX(prod_price) AS max_price FROM products;
```

`MAX()`返回products表中最贵的物品的价格

> **虽然MAX()一般用来找出最大的 数值或日期值，但MySQL允许将它用来返回任意列中的最大值，包括返回文本列中的最大值。在用于文本数据时，如果数据按相应的列排序，则`MAX()`返回最后一行。**
>
> **`MAX()`函数忽略列值为NULL的行。**

`MIN`

- 它返回指定列的最小值。与`MAX()`一样，`MIN()`要求指定列名

```
SELECT MIN(prod_price) AS min_price FROM products;
```

`MIN()`返回products表中最便宜物品的价格

> **`MIN()`函数与MAX()函数类似， MySQL允许将它用来返回任意列中的最小值，包括返回文本列中的最小值。在用于文本数据时，如果数据按相应的列排序， 则`MIN()`返回最前面的行。**
>
> **MIN()函数忽略列值为NULL的行。**

`SUM`

- `SUM()`用来返回指定列值的和（总计）。

```
SELECT SUM(quantity) AS items_ordered FROM orderitems WHERE order_num = 20005;
```

检索所订购物品的总数（所有quantity值之和）

```
SELECT SUM(quantity * item_price) AS total_price FROM orderitems WHERE order_num
= 20005;
```

合计每项物品的`item_price*quantity`，得出总的订单金额。`WHERE`子句同样保证只统计某个物品订单中的物品。

> **利用标准的算术操作符，所有聚集函数都可用来执行多个列上的计算。**
>
> **`SUM()`函数忽略列值为`NULL`的行。**

### 聚集不同值

> 聚集函数的 DISTINCT的使用，在MySQL 4.x中不能正常运行

- 对所有的行执行计算，指定`ALL`参数或不给参数（因为`ALL`是默认行为）；
- 只包含不同的值，指定`DISTINCT`参数。

```
mysql> SELECT AVG(prod_price) AS avg_price FROM products WHERE vend_id = 1003;
+------------+
| avg_price  |
+------------+
| 13.212857  |
+------------+
```

```
mysql> SELECT AVG(DISTINCT prod_price) AS avg_price FROM products WHERE vend_id = 1003;
+------------+
| avg_price  |
+------------+
| 15.998000  |
+------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM products WHERE vend_id = 1003;
+---------+---------+---------------+------------+----------------------------------------------+
| prod_id | vend_id | prod_name     | prod_price | prod_desc                                    |
+---------+---------+---------------+------------+----------------------------------------------+
| DTNTR   |    1003 | Detonator     |      13.00 | Detonator (plunger powered), fuses not included |
| FB      |    1003 | Bird seed     |      10.00 | Large bag (suitable for road runners)        |
| FC      |    1003 | Carrots       |       2.50 | Carrots (rabbit hunting season only)         |
| SAFE    |    1003 | Safe          |      50.00 | Safe with combination lock                   |
| SLING   |    1003 | Sling         |       4.49 | Sling, one size fits all                     |
| TNT1    |    1003 | TNT (1 stick) |       2.50 | TNT, red, single stick                       |
| TNT2    |    1003 | TNT (5 sticks) |      10.00 | TNT, red, pack of 10 sticks                  |
+---------+---------+---------------+------------+----------------------------------------------+
7 rows in set (0.00 sec)
```

它与上面的`SELECT`语句相同，但使用了`DISTINCT`参数，因此平均值只考虑各个不同的价格

> **如果指定列名，则`DISTINCT`只能用于`COUNT()`。`DISTINCT`不能用于`COUNT(*)`，因此不允许使用`COUNT（DISTINCT）`， 否则会产生错误。类似地，`DISTINCT`必须使用列名，不能用于计算或表达式。**
>
> **虽然`DISTINCT`从技术上可 用于`MIN()`和`MAX()`，但这样做实际上没有价值。一个列中的最小值和最大值不管是否包含不同值都是相同的**

**组合聚集函数**

```
SELECT   COUNT(*) AS num_items,
        MIN(prod_price) AS price_min,
        MAX(prod_price) AS price_max,
        AVG(prod_price) AS price_avg
FROM products;
```

返回4个值（products表中物品的数目，产品价格的最高、最低以及平均值）。

> 在指定别名以包含某个聚集函数的结果时，不应该使用表中实际的列名。虽然这样做并非不合法，但使用唯一的名字会使你的SQL更易于理解和使用（以及将来容易排除故障）。

**MySQL支持一系列聚集函数，可以用多种 方法使用它们以返回所需的结果。它们返回结果一般比你在自己的客户机应用程序中计算要快得多**

## 分组数据

### 创建分组

分组是在`SELECT`语句的`GROUP BY`子句中建立的。

```
SELECT vend_id, COUNT(*) AS num_prods FROM products GROUP BY vend_id;
```

这导致对每个vend_id而不是整个表计算num_prods一次

因为使用了`GROUP BY`，就不必指定要计算和估值的每个组了。系统 会自动完成。`GROUP BY`子句指示MySQL分组数据，然后对每个组而不是整个结果集进行聚集。

> - **`GROUP BY`子句可以包含任意数目的列。这使得能对分组进行嵌套， 为数据分组提供更细致的控制。**
> - **如果在GROUP BY子句中嵌套了分组，数据将在最后规定的分组上进行汇总。换句话说，在建立分组时，指定的所有列都一起计算 （所以不能从个别的列取回数据）。**
> - **`GROUP BY`子句中列出的每个列都必须是检索列或有效的表达式 （但不能是聚集函数）。如果在`SELECT`中使用表达式，则必须在`GROUP BY`子句中指定相同的表达式。不能使用别名。**
> - **除聚集计算语句外，`SELECT`语句中的每个列都必须在`GROUP BY`子 句中给出。**
> - **如果分组列中具有NULL值，则`NULL`将作为一个分组返回。如果列中有多行`NULL`值，它们将分为一组。**
> - **`GROUP BY`子句必须出现在`WHERE`子句之后，ORDER BY子句之前。**

使用`WITH ROLLUP`关键字，可以得到每个分组以及每个分组汇总级别（针对每个分组）的值

```
mysql> SELECT vend_id, COUNT(*) AS num_prods FROM products GROUP BY vend_id WITH ROLLUP;
+---------+-----------+
| vend_id | num_prods |
+---------+-----------+
|    1001 |         3 |
|    1002 |         2 |
|    1003 |         7 |
|    1005 |         2 |
|    NULL |        14 |
+---------+-----------+
5 rows in set (0.00 sec)
```

**过滤分组**

MySQL为此目的提供了另外的子句，那就是`HAVING`子句。`HAVING`非常类似于`WHERE`。

事实上，目前为止所学过的所有类型的`WHERE`子句都可以用`HAVING`来替代。

唯一的差别是`WHERE`过滤行，而`HAVING`过滤分组。

所学过的有关`WHERE`的所有这些技术和选项都适用于`HAVING`。

```
SELECT cust_id, COUNT(*) AS orders FROM orders GROUP BY cust_id HAVING COUNT(*)
>= 2;
```

过滤`COUNT(*) >=2`（两个以上的订单）的那些分组

> `WHERE`和`HAVING`的差别：**这里有另一种理解方法，`WHERE`在数据分组前进行过滤，`HAVING`在数据分组后进行过滤。这是一个重要的区别，`WHERE`排除的行不包括在分组中。这可能会改变计算值，从而影响`HAVING`子句中基于这些值过滤掉的分组**

```
SELECT vend_id, COUNT(*) AS num_prods FROM products WHERE prod_price >= 10 GROUP
BY vend_id HAVING COUNT(*) >= 2;
```

它列出具有2个（含）以上、价格为10（含）以上的产品的供应商

**分组和排序**

表13-1 ORDER BY与GROUP BY

| ORDER BY | GROUP BY |
| --- | --- |
| 排序产生的输出 | 分组行。但输出可能不是分组的顺序 |
| 任意列都可以使用（甚至非选择的列也可以使用） | 只可能使用选择列或表达式列，而且必须使用每个选择列表达式 |
| 不一定需要 | 如果与聚集函数一起使用列（或表达式），则必须使用 |

**你以某种方式分组数据（获得特定的分组聚集值），并不表示你需要以相同的方式排序输出。应该提供明确的`ORDER BY`子句，即使其效果等 同于`GROUP BY`子句也是如此**

> **一般在使用`GROUP BY`子句时，应该也给出`ORDER BY`子句。这是保证数据正确排序的唯一方法。千万不要仅依赖`GROUP BY`排序数据。**

它检索总计订单价格大于等于50的订 单的订单号和总计订单价格

```
SELECT order_num, SUM(quantity * item_price) AS ordertotal
FROM orderitems
GROUP BY order_num
HAVING SUM(quantity * item_price) >= 50;
```

```
+-----------+------------+
| order_num | ordertotal |
+-----------+------------+
|     20005 |     149.87 |
|     20006 |      55.00 |
|     20007 |    1000.00 |
|     20008 |     125.00 |
+-----------+------------+
```

为按总计订单价格排序输出，需要添加`ORDER BY`子句

```
SELECT order_num, SUM(quantity * item_price) AS ordertotal
FROM orderitems
GROUP BY order_num
HAVING SUM(quantity * item_price) >= 50
ORDER BY ordertotal;
```

```
+-----------+------------+
| order_num | ordertotal |
+-----------+------------+
|     20006 |      55.00 |
|     20008 |     125.00 |
|     20005 |     149.87 |
|     20007 |    1000.00 |
+-----------+------------+
```

**SELECT 子句顺序**

表13-2　SELECT子句及其顺序

| 子　　句 | 说　　明 | 是否必须使用 |
| --- | --- | --- |
| SELECT | 要返回的列或表达式 | 是 |
| FROM | 从中检索数据的表 | 仅在从表选择数据时使用 |
| WHERE | 行级过滤 | 否 |

| 子　　句 | 说　　明 | 是否必须使用 |
| --- | --- | --- |
| GROUP BY | 分组说明 | 仅在按组计算聚集时使用 |
| HAVING | 组级过滤 | 否 |
| ORDER BY | 输出排序顺序 | 否 |
| LIMIT | 要检索的行数 | 否 |

## 子查询

```
SELECT cust_id
FROM orders
WHERE order_num IN (SELECT order_num
                    FROM orderitems
                    WHERE prod_id = 'TNT2');
```

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN(SELECT order_num
                                     FROM orderitems
                                     WHERE prod_id = 'TNT2'));
```

对于能嵌套的子查询的数目没有限制，不过在实际使用时由于性能的限制，不能嵌套太多的子查询。

> **在`WHERE`子句中使用子查询（如这里所示），应该保证`SELECT`语句具有与`WHERE`子句中相同数目的列。通常， 子查询将返回单个列并且与单个列匹配，但如果需要也可以使用多个列**

虽然子查询一般与`IN`操作符结合使用，但也可以用于测试等于（=）、 不等于（<>）等

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
       FROM orders
       WHERE orders.cust_id = customers.cust_id) AS orders
FROM customers
ORDER BY cust_name;
```

显示customers 表中每个客户的订单总数

```
WHERE orders.cust_id = customers.cust_id
```

比较`orders`表中的`cust_id`与当前正从`customers`表中检索的`cust_id`

如果不限制列限定名

```
SELECT COUNT(*) FROM orders WHERE cust_id = cust_id;
```

总是返回orders表中的订单总数

> 用子查询测试和调试查询很有技巧性，特别是在这些语句的复杂性不断增加的情况下更是如此。用子查询建立（和测试）查询的最可靠的方法是逐渐进行， 这与MySQL处理它们的方法非常相同。首先，建立和测试最内层的查询。然后，用硬编码数据建立和测试外层查询，并且仅在确认它正常后才嵌入子查询。这时，再次测试它。对于要增加的每个查询，重复这些步骤。这样做仅给构造查询增加了一点点时间，但节省了以后（找出查询为什么不正常）的大量时间，并且极大地提高了查询一开始就正常工作的可能性。

## 联结表

### 联结

联结是一种机制，用来在一条`SELECT`语句中关联表，因此称之为联结。使用特殊的语法，可以联结多个表返回一组输出，联结在运行时关联表中正确的行

> - 关系表的设计就是要保证把信息分解成多个表，一类数据一个表。各表通过某些常用的值（即关系设计中的关系（relational））互相关联
> - 外键为某个表中的一列，它包含另一个表的主键值，定义了两个表之间的关系
> - 能够适应不断增加的工作量而不失败。设 计良好的数据库或应用程序称之为可伸缩性好（scale well）。

### 创建联结

```
SELECT vend_name, prod_name, prod_price
FROM vendors, products
WHERE vendors.vend_id = products.vend_id
ORDER BY vend_name, prod_name;
```

**WHERE子句的重要性**

在一条`SELECT`语句中联结几个表时，相应的关系是在运行中构造的。在数据库表的定义中不存在能指示MySQL如何对表进行联结的东西。你必须自己做这件事情。在联结两个表时，你实际上做的是将第一个表中的每一行与第二个表中的每一行配对。`WHERE`子句作为过滤条件，它只包含那些匹配给定条件（这里是联结条件）的行。没有WHERE子句，第一个表中的每个行将与第二个表中的每个行配对，而不管它们逻辑上是否可以配在一起

> 笛卡尔积：由没有联结条件的表关系返回的结果为笛卡儿积。检索出的行的数目将是第一个表中的行数乘以第二个表中的行数

```
SELECT vend_name, prod_name, prod_price FROM vendors, products ORDER BY
vend_name, prod_name;
```

> **应该保证所有联结都有`WHERE`子句，否则MySQL将返回比想要的数据多得多的数据。同理，应该保证`WHERE`子句的正确性。不正确的过滤条件将导致MySQL返回 不正确的数据**

有时我们会听到返回称为叉联结（cross join）的笛卡儿积的联结类型。

**内部联结**

目前为止所用的联结称为等值联结（equijoin），它基于两个表之间的 相等测试。这种联结也称为内部联结。

```
SELECT vend_name, prod_name, prod_price FROM vendors INNER JOIN products ON
vendors.vend_id = products.vend_id;
```

> ANSI SQL规范首选`INNER JOIN`语法。此外， 尽管使用`WHERE`子句定义联结的确比较简单，但是使用明确的联结语法能够确保不会忘记联结条件，有时候这样做也能影响性能

**联结多个表**

```
SELECT prod_name, vend_name, prod_price, quantity FROM orderitems, products,
vendors
WHERE products.vend_id = vendors.vend_id
AND orderitems.prod_id = products.prod_id
AND order_num = 20005;
```

显示编号为20005的订单中的物品。

> MySQL在运行时关联指定的每个表以处理联结。 这种处理可能是非常耗费资源的，因此应该仔细，不要联结 不必要的表。联结的表越多，性能下降越厉害

之前

```
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN(SELECT order_num
                                     FROM orderitems
                                     WHERE prod_id = 'TNT2'));
```

可以转换成

```
SELECT cust_name, cust_contact FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
AND orderitems.order_num = orders.order_num
AND prod_id = 'TNT2';
```

前两个关联联结中的表，后一个过滤产品TNT2的数据

## 高级联结

### 表别名

```
SELECT cust_name, cust_contact FROM customers AS c, orders AS o, orderitems AS
oi
WHERE c.cust_id = o.cust_id
AND oi.order_num = o.order_num
AND prod_id = 'TNT2';
```

`customers AS c` 建立c作为customers的别名

- 表别名不仅能用于WHERE子句，它还可以用于SELECT的列表、ORDER BY子句 以及语句的其他部分
- 表别名只在查询执行中使用。与列别名不一样，表别名不返回到客户机。

### 不同的联结

#### 自联结

```
SELECT prod_id, prod_name FROM products
WHERE vend_id = (SELECT vend_id FROM products WHERE prod_id = 'DTNTR');
```

等于

```
SELECT p1.prod_id, p1.prod_name FROM products AS p1, products AS p2
WHERE p1.vend_id = p2.vend_id
AND p2.prod_id = 'DTNTR';
```

> 自联结通常作为外部语句用来替代从相同表中检索数据时使用的子查询语句。虽然最终的结果是相同的，但有时候处理联结远比处理子查询快得多。应该试一 下两种方法，以确定哪一种的性能更好

#### 自然联结

自然联结排除多次出现，使每个列只返回一次

```
SELECT c.*, o.order_num, o.order_date, oi.prod_id, oi.quantity, oi.item_price
FROM customers AS c, orders AS o, orderitems AS oi
WHERE c.cust_id = o.cust_id
AND oi.order_num = o.order_num
AND prod_id = 'FB';
```

通配符只对第一个表使用。所有其他列明确列出，所以没有重复的列被检索出来

迄今为止我们建立的每个内部联结都是自然联结

### 外部联结

```
SELECT customers.cust_id, orders.order_num
FROM customers
LEFT OUTER JOIN orders
ON customers.cust_id = orders.cust_id;
```

**与内部联结关联两个表中的行不同的是，外部联结还包括没有关联行的行。在使用`OUTER JOIN`语法时，必须使用RIGHT或LEFT关键字指定包括其所有行的表（`RIGHT`指出的是`OUTER JOIN`右边的表，而`LEFT`指出的是`OUTER JOIN`左边的表）**

> MySQL不支持简化字符=和=的使用，这两种操作符在其他DBMS中是很流行的
>
> 存在两种基本的外部联结形式：**左外部联结和右外部联结。**它们之间的唯一差别是所关联的表的顺序不同。换句话说，**左外部联结**可通过颠倒`FROM`或`WHERE`子句中表的顺序转换为**右外部联结。**因此，两种类型的外部联结可互换使用，而究竟使用哪一种纯粹是根据方便而定。

```
SELECT customers.cust_name, customers.cust_id, COUNT(orders.order_num) AS
num_ord
FROM customers INNER JOIN orders
ON customers.cust_id = orders.cust_id
GROUP BY customers.cust_id;
```

检索所有客户及每个客户所 下的订单数

还可以与其他联结一起使用

```
SELECT customers.cust_name, customers.cust_id, COUNT(orders.order_num) AS
num_ord
FROM customers LEFT OUTER JOIN orders
ON customers.cust_id = orders.cust_id
GROUP BY customers.cust_id;
```

使用左外部联结来包含所有客户，甚至包含那些没有任何下订单的客户

### 联结要点

> - **注意所使用的联结类型。一般我们使用内部联结，但使用外部联结也是有效的。**
> - **保证使用正确的联结条件，否则将返回不正确的数据。**
> - **应该总是提供联结条件，否则会得出笛卡儿积。**
> - **在一个联结中可以包含多个表，甚至对于每个联结可以采用不同的联结类型。虽然这样做是合法的，一般也很有用，但应该在一 起测试它们前，分别测试每个联结。这将使故障排除更为简单。**

## 组合查询

多数SQL查询都只包含从一个或多个表中返回数据的单条SELECT语句。MySQL也允许执行多个查询（多条SELECT语句），并将结果作为单个查询结果集返回。这些组合查询通常称为并（union）或复合查询（compound query）

- 在单个查询中从不同的表返回类似结构的数据；
- 对单个表执行多个查询，按单个查询返回数据。

### 创建组合查询

可用`UNION`操作符来组合数条SQL查询。利用`UNION`，可给出多条`SELECT`语句，将它们的结果组合成单个结果集

```
SELECT vend_id, prod_id, prod_price FROM products WHERE prod_price <= 5
UNION
SELECT vend_id, prod_id, prod_price FROM products WHERE vend_id IN (1001, 1002);
```

`UNION`指示MySQL执行两条SELECT语句，并把输出组合成单个查询结果集

使用多条WHERE子句而不是使用UNION的相同查询

```
SELECT vend_id, prod_id, prod_price FROM products WHERE prod_price <= 5 OR
vend_id IN (1001, 1002);
```

### UNION规则

> - **`UNION`必须由两条或两条以上的`SELECT`语句组成，语句之间用关键字`UNION`分隔（因此，如果组合4条`SELECT`语句，将要使用3个UNION关键字）。**
> - **UNION中的每个查询必须包含相同的列、表达式或聚集函数（不过各个列不需要以相同的次序列出）。**
> - **列数据类型必须兼容：类型不必完全相同，但必须是DBMS可以隐含地转换的类型（例如，不同的数值类型或不同的日期类型）**

`UNION`从查询结果集中自动去除了重复的行（换句话说，它的行为与单条`SELECT`语句中使用多个`WHERE`子句条件一样）。

这是`UNION`的默认行为，但是如果需要，可以改变它。事实上，如果想返回所有匹配行，可使用`UNION ALL`而不是`UNION`

```
SELECT vend_id, prod_id, prod_price FROM products WHERE prod_price <= 5
UNION ALL
SELECT vend_id, prod_id, prod_price FROM products WHERE vend_id IN (1001, 1002);
```

> **UNION和WHERE** `UNION`几乎总是完成与多个`WHERE`条件相同的工作。`UNION ALL`为`UNION`的一种形式，它完成`WHERE`子句完成不了的工作。如果确实需要每个条件的匹配行全 部出现（包括重复行），则必须使用`UNION ALL`而不是`WHERE`

#### 对组合结果进行排序

在用`UNION`组合查询时，**只能使用一条`ORDER BY`子句，它必须出现在最后一条SELECT语句之后。**对于结果集，不存在用一种方式排序一部分，而又用另一种方式排序另一部分的情况，因此不允许使用多条`ORDER BY`子句

```
SELECT vend_id, prod_id, prod_price FROM products WHERE prod_price <= 5
UNION
SELECT vend_id, prod_id, prod_price FROM products WHERE vend_id IN (1001, 1002)
ORDER BY vend_id, prod_price;
```

#### 使用UNION的组合查询可以应用不同的表

## 全文本搜索

两个最常使用的引擎为MyISAM和InnoDB， 前者支持全文本搜索，而后者不支持。

MySQL创建指定列中各词的一个索引，搜索可以针对这些词进行。

#### 使用全文本搜索

为了进行全文本搜索，必须索引被搜索的列，而且要随着数据的改变不断地重新索引。在对表列进行适当设计后，MySQL会自动进行所有的索引和重新索引。 在索引之后，`SELECT`可与`Match()`和`Against()`一起使用以实际执行搜索

##### 启用全文本搜索支持

一般在创建表时启用全文本搜索。`CREATE TABLE`语句接受`FULLTEXT`子句，它给出被索引列的一个逗号分隔的列表

```
CREATE TABLE productnotes(
node_id int          NOT NULL    AUTO_INCREMENT,
prod_id char(10)      NOT NULL,
note_date datetime    NOT NULL,
note_text text       NULL,
PRIMARY KEY(node_id),
FULLTEXT(note_text)
)ENGINE = MyISAM;
```

为了进行全文本搜索， MySQL根据子句`FULLTEXT(note_text)`的指示对它进行索引。在定义之后，MySQL自动维护该索引。在增加、更新或删除行时， 索引随之自动更新。

可以在创建表时指定`FULLTEXT`，或者在稍后指定（在这种情况下所 有已有数据必须立即索引）

> **更新索引要花时间，虽然不是很多，但毕竟要花时间。如果正在导入数据到一个新表， 此时不应该启用`FULLTEXT`索引。应该首先导入所有数据，然后再修改表，定义`FULLTEXT`。这样有助于更快地导入数据（而且使索引数据的总时间小于在导入每行时分别进行索引所需 的总时间）**

##### 进行全文搜索

使用两个函数`Match()`和`Against()`执行全文本搜索， 其中`Match()`指定被搜索的列，`Against()`指定要使用的搜索表达式

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('rabbit');
```

> **传递给 `Match()` 的值必须与 `FULLTEXT()` 定义中的相同。如果指定多个列，则必须列出它们（而且次序正确）。**
>
> **除非使用`BINARY`方式（本章中没有介绍），否则全文本搜索不区分大小写**

上面的搜索可以简单地用`LIKE`子句完成

```
SELECT note_text FROM productnotes WHERE note_text LIKE '%rabbit%';
```

> 确实包含词rabbit的两个行每行都有一个等级值，文本中词靠前的行的等级值比词靠后的行的等级值高
>
> 如果指定多个搜索项，则包含多数匹配词的那些行将具有比包含较少词（或仅有一个匹配）的那些行高的等级值

### 使用查询拓展

查询扩展用来设法放宽所返回的全文本搜索结果的范围

> 在使用查询扩展时，MySQL对数据和索引进行**两遍扫描**来完成搜索：
>
> - 首先，进行一个基本的全文本搜索，找出与搜索条件匹配的所有行；
> - 其次，MySQL检查这些匹配行并选择所有有用的词（我们将会简要地解释MySQL如何断定什么有用，什么无用）。
> - 再其次，MySQL再次进行全文本搜索，这次不仅使用原来的条件，而且还使用所有有用的词。

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('anvils');
```

```
+-----------------------------------------------------------------------------------------------------------------------------------------------+
| note_text                                                                                                                                     |
+-----------------------------------------------------------------------------------------------------------------------------------------------+
| Multiple customer returns, anvils failing to drop fast enough or falling backwards on purchaser. Recommend that customer considers using heavier anvils. |
+-----------------------------------------------------------------------------------------------------------------------------------------------+
```

只有一行包含词anvils，因此只返回一行。

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('anvils' WITH
QUERY EXPANSION);
```

```
+-----------------------------------------------------------------------------------------------------------------------------------------------+
| note_text                                                                                                                                     |
+-----------------------------------------------------------------------------------------------------------------------------------------------+
| Multiple customer returns, anvils failing to drop fast enough or falling backwards on purchaser. Recommend that customer considers using heavier anvils. |
| Customer complaint:
Sticks not individually wrapped, too easy to mistakenly detonate all at once.
Recommend individual wrapping.                        |
| Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping, recommend ANV02 or ANV03 instead.  |
| Please note that no returns will be accepted if safe opened using explosives.                                                                 |
| Customer complaint: rabbit has been able to detect trap, food apparently less effective now.                                                  |
| Customer complaint:
Circular hole in safe floor can apparently be easily cut with handsaw.                                                                        |
| Matches not included, recommend purchase of matches or detonator (item DTNTR).                                                               |
+-----------------------------------------------------------------------------------------------------------------------------------------------+
```

第一行包含词anvils，因此等级最高。第二行与anvils无关，但因为它包含第一行中的两个词（customer 和recommend），所以也被检索出来。第3行也包含这两个相同的词，但它 们在文本中的位置更靠后且分开得更远，因此也包含这一行，但等级为第三。第三行确实也没有涉及anvils（按它们的产品名）

> **表中的行越多（这些行中的文本就越多），使用查询扩展返回的结果越好**

#### 布尔运算

MySQL支持全文本搜索的另外一种形式，称为布尔方式：

> 可以提供关于如下内容的细节：
>
> - 要匹配的词；
> - 要排斥的词（如果某行包含这个词，则不返回该行，即使它包含 其他指定的词也是如此）；
> - 排列提示（指定某些词比其他词更重要，更重要的词等级更高）；
> - 表达式分组；
> - 另外一些内容

> **布尔方式不同于迄今为止使用的全文本搜索语法的地方在于，即使没有定义`FULLTEXT`索引，也可以使用它。但这是一种非常缓慢的操作 （其性能将随着数据量的增加而降低）。**

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('heavy' IN
BOOLEAN MODE);
```

此全文本搜索检索包含词heavy的所有行

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('heavy -rope*'
IN BOOLEAN MODE);
```

这一次仍然匹配词heavy，但-rope* 明确地指示MySQL排除包含rope*（任何以rope开始的词，包括ropes）的行

表18-1　全文本布尔操作符

| 布尔操作符 | 说　明 |
| --- | --- |
| + | 包含，词必须存在 |
| - | 排除，词必须不出现 |
| > | 包含，而且增加等级值 |
| < | 包含，且减少等级值 |
| ( ) | 把词组成子表达式（允许这些子表达式作为一个组被包含、排除、排列等） |
| ~ | 取消一个词的排序值 |
| * | 词尾的通配符 |
| " " | 定义一个短语（与单个词的列表不一样，它匹配整个短语以便包含或排除这个短语） |

例子：

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('+safe +
(<combination)' IN BOOLEAN MODE);
```

这个搜索匹配词 safe 和 combination，降低后者的等级

```
SELECT note_text FROM productnotes WHERE Match(note_text) Against('"rabbit
bait"' IN BOOLEAN MODE);
```

这个搜索匹配短语rabbit bait而不是匹配两个词 rabbit 和 bait。

**全文本搜索的说明**

> - 在索引全文本数据时，**短词被忽略且从索引中排除。**短词定义为那些具有3个或3个以下字符的词（如果需要，这个数目可以更改）。
> - MySQL带有一个内建的**非用词（stopword）列表，这些词在索引全文本数据时总是被忽略。**如果需要，可以覆盖这个列表（请参 阅MySQL文档以了解如何完成此工作）。
> - 许多词出现的频率很高，搜索它们没有用处（返回太多的结果）。 因此，MySQL规定了一条50%规则，**如果一个词出现在50%以上的行中，则将它作为一个非用词忽略。**50%规则不用于`IN BOOLEAN MODE`。
> - 如果表中的行数**少于3行**，则全文本搜索**不返回结果**（因为每个词或者不出现，或者至少出现在50%的行中）。
> - **忽略词中的单引号。**例如，don't索引为dont。
> - **不具有词分隔符（包括日语和汉语）的语言不能恰当地返回全文本搜索结果。**
> - 如前所述，**仅在MyISAM数据库引擎中支持全文本搜索。**

> **邻近搜索**是许多全文本搜索支持的一个特性，它能搜索相邻的词（在相同的句子中、相同的段落中或者 在特定数目的词的部分中，等等）。**MySQL全文本搜索现在还不支持邻近操作符**，不过未来的版本有支持这种操作符的计划

## 插入数据

`INSERT`是用来插入（或添加）行到数据库表的。

> 可针对每个表或每个用户，利用MySQL的安全机制禁止使用`INSERT`语句

**插入完整的行**

要求指定表名和被插入到新行中的值

```
INSERT INTO customers
Value(NULL,
      'Pep E. LaPew',
      '100 Main Street',
      'Los Angeles',
      'CA',
      '90046',
      'USA',
      NULL,
      NULL);
```

存储到每个表列中的数据在`VALUES`子句中给出，对每个列必须提供一个值。如果某个列没有值（如上面的cust_contact和cust_email列），应该使用`NULL`值（假定表允许对该列指定空值）。**各个列必须以它们在表定义中出现的次序填充。**第一列cust_id也为`NULL`。这是因为每次插入一个新行时，该列由MySQL**自动增量。**你不想给出一个值（这是MySQL的工作），又**不能省略此列**（如前所述，必须给出每个列），所以指定一个`NULL`值（它被 MySQL忽略，MySQL在这里插入下一个可用的cust_id值）

但是不安全

更安全的是：

```
INSERT INTO customers(cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
```

```
                    cust_country,
                    cust_contact,
                    cust_email)
              VALUES('Pep E. LaPew',
                    '100 Main Street',
                    'Los Angeles',
                    'CA',
                    '90046',
                    'USA',
                    NULL,
                    NULL);
```

在插入行时，MySQL将用`VALUES`列表中的相应值填入列表中的对应项。因为提供了列名，`VALUES`必须以其指定的次序匹配指定的列名，不 一定按各个列出现在实际表中的次序。其优点是，即使表的结构改变， 此`INSERT`语句仍然能正确工作。你会发现cust_id的`NULL`值是不必要的， cust_id列并没有出现在列表中，所以不需要任何值。

**转换顺序也行**

> **一般不要使用没有明确给出列的列表的`INSERT`语句。使用列的列表能使SQL代码继续发挥作用，即使表结构发生了变化**
>
> **不管使用哪种`INSERT`语法，都必须给出`VALUES`的正确数目。如果不提供列名，则必须给每个表列提供一个值。如果提供列名，则必须对每个列出的列给出一个值。 如果不这样，将产生一条错误消息，相应的行插入不成功。**

> 如果表的定义允许，则可以在`INSERT`操作中省略某些列。省略的列必须满足以下某个条件。
>
> - **该列定义为允许NULL值（无值或空值）。**
> - **在表定义中给出默认值。这表示如果不给出值，将使用默认值。 如果对表中不允许`NULL`值且没有默认值的列不给出值，则 MySQL将产生一条错误消息，并且相应的行插入不成功。**

`INSERT`操作可能很耗时（特别是有很多索引需要更新时），而且它可能降低等待处理的`SELECT`语句的性能。

如果数据检索是最重要的（通常是这样），则你可以通过在`INSERT`和`INTO`之间添加关键字`LOW_PRIORITY`，指示MySQL 降低`INSERT`语句的优先级，如下所示：

```
INSERT LOW_PRIORITY INTO
```

同样也适用`UPDATE`和`DELETE`语句

**插入多个行**

```
INSERT INTO customers(cust_name, cust_address, cust_city, cust_state, cust_zip,
cust_country)
VALUES('Pep E. LaPew', '100 Main Street', 'Los Angeles', 'CA', '90046', 'USA');
INSERT INTO customers(cust_name, cust_address, cust_city, cust_state, cust_zip,
cust_country)
VALUES('M. Martian', '42 Galaxy Way', 'New York', 'NY', '11213', 'USA');
```

或者，只要每条`INSERT`语句中的列名（和次序）相同，可以如下组合各语句

```
INSERT INTO customers(cust_name, cust_address, cust_city, cust_state, cust_zip,
cust_country)
VALUES('Pep E. LaPew', '100 Main Street', 'Los Angeles', 'CA', '90046', 'USA'),
('M. Martian', '42 Galaxy Way', 'New York', 'NY', '11213', 'USA');
```

> **此技术可以提高数据库处理的性能，因为MySQL用单条`INSERT`语句处理多个插入比使用多条`INSERT`语句快**

#### 插入检索出来的数据

`INSERT`一般用来给表插入一个指定列值的行。但是，`INSERT`还存在另一种形式，可以利用它将一条`SELECT`语句的结果插入表中。这就是所谓的`INSERT SELECT`

```
INSERT INTO
custnew(cust_id, cust_contact, cust_email, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country)
SELECT cust_id, cust_contact, cust_email, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country
FROM customers;
```

这个例子使用`INSERT SELECT`从customers中将所有数据导入custnew

这条语句将插入多少行有赖于custnew表中有多少行。 如果这个表为空，则没有行被插入（也不产生错误，因为操作仍然是合法的）。

> MySQL甚至不关心`SELECT`返回的**列名**。它使用的是**列的位置**，因此`SELECT`中的第一列（不管其列名）将用来填充表列中指定的第一个列，第二列将用来填充表列中指定的第二个列，如此等等。这对于从使用不同列名的表中导入数据是非常有用的

## 更新和删除数据

#### 更新数据

使用`UPDATE`更新（修改）数据，两种方式使用`UPDATE`

- 更新表中特定行
- 更新表中所有行

> **不要省略`WHERE`子句** 在使用`UPDATE`时一定要注意细心。因为稍不注意，就会更新表中**所有行**。

> 基本的 UPDATE语句由3部分组成，分别是：
>
> - 要更新的表；
> - 列名和它们的新值
> - 确定要更新行的过滤条件

```
UPDATE customers SET cust_email = 'elmer@fudd.com' WHERE cust_id = 10005;
```

`SET`命令用来将新值赋给被更新的列。

更新多个列

```
UPDATE customers SET cust_name = 'The Fudds', cust_email = 'elmer@fudd.com'
WHERE cust_id = 10005;
```

在更新多个列时，只需要使用单个`SET`命令，每个"列=值"对之间用逗号分隔（最后一列之后不用逗号）

> `UPDATE`语句中可以使用子查 询，使得能用`SELECT`语句检索出的数据更新列数据

> 如果用`UPDATE`语句更新多行，并且在更新这些行中的一行或多行时出一个现错误，则整个`UPDATE`操作被取消 （错误发生前更新的所有行被恢复到它们原来的值）。为即使是发生错误，也继续进行更新，可使用`IGNORE`关键字
>
> ```
> UPDATE IGNORE customers...
> ```

为了删除某个列的值，可设置它为`NULL`（假如表定义允许`NULL`值）

```
UPDATE customers SET cust_email = NULL WHERE cust_id = 10005;
```

其中`NULL`用来去除cust_email列中的值。

#### 删除数据

为了从一个表中删除（去掉）数据，使用`DELETE`语句，两种方式使用`DELETE`：

- 从表中删除特定的行；
- 从表中删除所有行

> **不要省略WHERE子句** 在使用`DELETE`时一定要注意细心。因为稍不注意，就会错误地删除表中**所有行。**

```
DELETE FROM customers WHERE cust_id = 10006;
```

**`DELETE`不需要列名或通配符。`DELETE`删除整行而不是删除列。为了删除指定的列，请使用`UPDATE`语句。**

> **`DELETE`语句从表中删除行，甚至是删除表中所有行。但是，`DELETE`不删除表本身。**

> 如果想从**表中删除所有行**，不要使用`DELETE`。 可使用`TRUNCATE TABLE`语句，它完成相同的工作，但速度更快（`TRUNCATE`实际是删除原来的表并重新创建一个表，而不是逐行删除表中的数据）

#### 更新或删除数据原则

- 除非确实打算更新和删除每一行，否则绝对不要使用不带`WHERE`子句的`UPDATE`或`DELETE`语句。
- 保证每个表都有主键，尽可能像`WHERE`子句那样使用它（可以指定各主键、多个值或值的范围）。
- 在对`UPDATE`或`DELETE`语句使用`WHERE`子句前，应该先用`SELECT`进行测试，保证它过滤的是正确的记录，以防编写的`WHERE`子句不正确。
- 使用强制实施引用完整性的数据库，这样MySQL将不允许删除具有与其他表相关联的数据的行。

## 创建和操纵表

### 创建表

为了用程序创建表，可使用SQL的`CREATE TABLE`语句

为利用`CREATE TABLE`创建表，必须给出下列信息：

- 新表的名字，在关键字`CREATE TABLE`之后给出；
- 表列的名字和定义，用逗号分隔

```
CREATE TABLE IF NOT EXISTS customers(
    cust_id             int      NOT NULL AUTO_INCREMENT,
    cust_name           char(50) NOT NULL,
    cust_address        char(50) NULL,
    cust_city           char(50) NULL,
    cust_state          char(5)  NULL,
    cust_zip            char(50) NULL,
    cust_country        char(50) NULL,
    cust_contact        char(50) NULL,
    cust_email          char(255) NULL,
    PRIMARY KEY(cust_id)
)ENGINE=InnoDB;
```

表的主键可以在创建表时用 `PRIMARY KEY` 关键字指定

> 在创建新表时，指定的表名必须不存在，否则将出错。如果要防止意外覆盖已有的表，SQL要求首先手工删除该表，然后再重建它，而不是简单地用创建表语句覆盖它。
>
> 如果你仅想在一个表不存在时创建它，应该在表名后给出`IF NOT EXISTS`。这样做不检查已有表的模式是否与你打算创建 的表模式相匹配。它只是查看表名是否存在，并且仅在表名不 存在时创建它
>
> ```
> CREATE TABLE IF NOT EXISTS customers(...);
> ```

### NULL值

`NULL`值就是没有值或缺值。允许`NULL`值的列也允许在插入行时不给出该列的值。不允许`NULL`值的列不接受该列没有值的行， 换句话说，在插入或更新行时，该列必须有值。

`NULL`为**默认设置**，如果不指定`NOT NULL`，则认为指定的是`NULL`

> 不要把`NULL`**值**与**空串**相混淆。`NULL`值是没有值， 它**不是空串**。如果指定`''`（两个单引号，其间没有字符），这在`NOT NULL`列中是允许的。**空串是一个有效的值**，它不是无值。NULL值用关键字NULL而不是空串指定

### 主键

**主键值必须唯一。**即，表中的每个行必须具有唯一的主键值。如果主键使用单个列，则它的值必须唯一。如果使用多个列，则这些列的组合值必须唯一。

`PRIMARY KEY(vend_id)`

为创建由多个列组成的主键，应该以**逗号**分隔的列表给出各列名，

```
CREATE TABLE orderitems(
    order_num    int           NOT NULL,
    order_item   int           NOT NULL,
    prod_id      char(10)      NOT NULL,
    quantity     int           NOT NULL,
    item_price   decimal(8,2)  NOT NULL,
    PRIMARY KEY(order_num, order_item)
)ENGINE = InnoDB;
```

主键可以在创建表时定义，或者在创建表之后定义

> 主键为其值唯一标识表中每个行的列。主键中只能使用不允许`NULL`值的列。允许`NULL`值的列**不能作为唯一标识**

#### 使用AUTO_INCREMENT

```
cust_id     int     NOT NULL AUTO_INCREMENT,
```

`AUTO_INCREMENT`告诉MySQL，本列每当增加一行时自动增量。每次 执行一个`INSERT`操作时，MySQL自动对该列增量（从而才有这个关键字`AUTO_INCREMENT`），给该列赋予下一个可用的值。这样给每个行分配一个 唯一的cust_id，从而可以用作主键值。

每个表**只允许一个**`AUTO_INCREMENT`列，而且它**必须被索引**（如，通过使它成为主键）。

> **覆盖AUTO_INCREMENT** 你可以简单地在`INSERT`语句中**指定一个值**，只要它是**唯一的（至今尚未使用过）**即可，该值将被用来**替代自动生成的值。后续的增量将开始使用该手工插入的值。**

```
SELECT last_insert_id()
```

此语句返回最后一个`AUTO_INCREMENT`值，然后可以将它用于后续的MySQL语句。

#### 指定默认值

如果在插入行时没有给出值，MySQL允许指定此时使用的默认值。

```
CREATE TABLE orderitems(
    order_num    int           NOT NULL,
    order_item   int           NOT NULL,
    prod_id      char(10)      NOT NULL,
    quantity     int           NOT NULL DEFAULT 1,
    item_price   decimal(8,2)  NOT NULL,
    PRIMARY KEY(order_num, order_item)
)ENGINE = InnoDB;
```

> 与大多数DBMS不一样，MySQL**不允许使用函数作为默认值**，它**只支持常量。**

> 许多数据库开发人员使用默认值而不是`NULL`列，特别是对用于计算或数据分组的列更是如此

#### 引擎类型

与其他DBMS一样，MySQL有一个具体管理和处理数据的内部引擎。 在你使用`CREATE TABLE`语句时，该引擎具体创建表，而在你使用`SELECT`语句或进行其他数据库处理时，该引擎在内部处理你的请求。多数时候， 此引擎都隐藏在DBMS内，不需要过多关注它。

但MySQL与其他DBMS不一样，它具有多种引擎。它打包多个引擎， 这些引擎都隐藏在MySQL服务器内，全都能执行`CREATE TABLE`和`SELECT`等命令

如果省略ENGINE=语句，则使用默认引擎（很可能是MyISAM），多数SQL语句都会默认使用它。

- InnoDB是一个可靠的事务处理引擎，它**不支持全文本搜索**；
- MEMORY在功能等同于MyISAM，但由于**数据存储在内存**（不是磁盘） 中，速度很快（特别适合于临时表）；
- MyISAM是一个性能极高的引擎，它支持全文本搜索， 但**不支持事务处理**

#### 更新表

为更新表定义，可使用`ALTER TABLE`语句。

`ALTER TABLE`需要的信息

- 在`ALTER TABLE`之后给出要更改的表名（该表必须存在，否则将 出错）；
- 所做更改的列表

```
ALTER TABLE vendors ADD vend_phone CHAR(20);
```

给表添加一个列

```
ALTER TABLE vendors DROP COLUMN vend_phone;
```

删除刚刚添加的列

`ALTER TABLE`的一种常见用途是定义外键

```
ALTER TABLE orderitems ADD CONSTRAINT fk_orderitems_orders FOREIGN
KEY(order_num) REFERENCES orders(order_num);
```

为了对单个表进行多个更改，可以使用单条ALTER TABLE语句，每个更改用逗号分隔。

复杂的表结构更改一般需要手动删除过程，它涉及以下步骤：

- 用新的列布局创建一个新表；
- 使用`INSERT SELECT`语句从旧表复制数据到新表。如果有必要，可使用转换函数和计算字段
- 检验包含所需数据的新表；
- 重命名旧表（如果确定，可以删除它）；
- 用旧表原来的名字重命名新表；
- 根据需要，重新创建触发器、存储过程、索引和外键。

> 使用`ALTER TABLE`要极为小心，应该在进行改动前做一个完整的备份（模式和数据的备份）。
>
> 数据库表的更改不能撤销，如果增加了不需要的列，可能不能删除它们。类似地，如果删除了不应该删除的列，可能会丢失该列中的所有数据。

#### 删除表

```
DROP TABLE customer;
```

**删除表没有确认， 也不能撤销，执行这条语句将永久删除该表。**

#### 重命名表

```
RENAME TABLE customers2 TO customers;
```

对多个表进行重命名

```
RENAME TABLE backup_customers TO customers,
             backup_vendors   TO vendors,
             backup_products  TO products;
```

## 视图

试图的应用：

- 重用SQL语句。
- 简化复杂的SQL操作。在编写查询后，可以方便地重用它而不必知道它的基本查询细节。
- 使用表的组成部分而不是整个表。
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限。
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据

在视图创建之后，可以用与表基本相同的方式利用它们。可以对视图执行`SELECT`操作，过滤和排序数据，将视图联结到其他视图或表，甚至能添加和更新数据

重要的是知道视图仅仅是用来查看存储在别处的数据的一种设施。 **视图本身不包含数据，因此它们返回的数据是从其他表中检索出来的。 在添加或更改这些表中的数据时，视图将返回改变过的数据。**

> 性能：因为视图不包含数据，所以每次使用视图时，都必须处理查询执行时所需的任一个检索。如果你用多个联结和过滤创建了复杂的视图或者嵌套了视图，可能会发现性能 下降得很厉害。

#### 规则

- 与表一样，视图必须唯一命名（不能给视图取与别的视图或表相 同的名字）。
- 对于可以创建的视图数目没有限制。
- 为了创建视图，必须具有足够的访问权限。这些限制通常由数据库管理人员授予。
- 视图可以嵌套，即可以利用从其他视图中检索数据的查询来构造一个视图。
- `ORDER BY`可以用在视图中，但如果从该视图检索数据`SELECT`中也 含有`ORDER BY`，那么该视图中的`ORDER BY`将被覆盖。
- 视图不能索引，也不能有关联的触发器或默认值。
- 视图可以和表一起使用。

**使用视图**

- 视图用`CREATE VIEW`语句来创建。
- 使用`SHOW CREATE VIEW viewname`；来查看创建视图的语句。
- 用DROP删除视图，其语法为`DROP VIEW viewname;`。
- 更新视图时，可以先用`DROP`再用`CREATE`，也可以直接用`CREATE OR REPLACE VIEW`。如果要更新的视图不存在，则第2条更新语句会创建一个视图；如果要更新的视图存在，则第2条更新语句会替换原有视图。

#### 利用视图简化复杂的联结

```
CREATE VIEW productcustomers AS
SELECT cust_name, cust_contact, prod_id
FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
AND orderitems.order_num = orders.order_num;
```

这条语句创建一个名为productcustomers的视图，它联结三个 表，以返回已订购了任意产品的所有客户的列表。

检索订购了产品TNT2的客户：

```
SELECT cust_name, cust_contact FROM productcustomers WHERE prod_id = 'TNT2';
```

#### 用视图重新格式化检索出的数据

```
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country),')') AS vend_title
FROM vendors ORDER BY vend_name;
```

假如经常需要这个格式的结果。不必在每次需要时执行联结， 创建一个视图，每次需要时使用它即可。为把此语句转换为视图

```
CREATE VIEW vendorlocations AS
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country),')') AS vend_title
FROM vendors ORDER BY vend_name;
```

#### 用视图过滤不想要的数据

可以定义 customeremaillist视图，它过滤没有电子邮件地址的客户

```
CREATE VIEW customeremaillist AS
SELECT cust_id, cust_name, cust_email
FROM customers
WHERE cust_email IS NOT NULL;
```

如果从视图检索数据时使用了一条`WHERE`子句，则两组子句（一组在视图中，另一组是传递给视 图的）将**自动组合**

#### 使用视图与计算字段

```
CREATE VIEW orderitemsexpanded AS
SELECT order_num, prod_id, quantity, item_price, quantity * item_price AS
expanded_price
FROM orderitems;
```

#### 更新视图

通常，视图是可更新的（即，可以对它们使用`INSERT`、`UPDATE`和`DELETE`）。更新一个视图将更新其基表（可以回忆一下，视图本身没有数 据）。如果你对视图增加或删除行，实际上是对其基表增加或删除行。

基本上可以说，如果MySQL不能正确地确定被更新的基数据，则不允许更新（包括插入和删除）。这实际上意味着，如果视图定义中有以下操作，则不能进行视图的更新：

- 分组（使用`GROUP BY`和`HAVING`）；
- 联结；
- 子查询；
- 并；
- 聚集函数（Min()、Count()、Sum()等）；
- DISTINCT；
- 导出（计算）列

一般，应该将视图用于检索（`SELECT`语句） 而不用于更新（`INSERT`、`UPDATE`和`DELETE`）。

## 存储过程

存储过程简单来说，就是为以后的使用而保存的一条或多条MySQL语句的集合。可将其视为批文件，虽然它们的作用不仅限于批处理

#### 使用存储过程的优点和缺陷

优点：

- 通过把处理封装在容易使用的单元中，简化复杂的操作
- 由于不要求反复建立一系列处理步骤，这保证了数据的完整性。 如果所有开发人员和应用程序都使用同一（试验和测试）存储过 程，则所使用的代码都是相同的。 这一点的延伸就是防止错误。需要执行的步骤越多，出错的可能性就越大。防止错误保证了数据的一致性。
- 简化对变动的管理。如果表名、列名或业务逻辑（或别的内容） 有变化，只需要更改存储过程的代码。使用它的人员甚至不需要知道这些变化。 这一点的延伸就是安全性。通过存储过程限制对基础数据的访问减 少了数据讹误（无意识的或别的原因所导致的数据讹误）的机会。
- 提高性能。因为使用存储过程比使用单独的SQL语句要快。
- 存在一些只能用在单个请求中的MySQL元素和特性，存储过程可以使用它们来编写功能更强更灵活的代码

缺陷：

- 一般来说，存储过程的编写比基本SQL语句复杂，编写存储过程需要更高的技能，更丰富的经验。
- 你可能没有创建存储过程的安全访问权限。许多数据库管理员限制存储过程的创建权限，允许用户使用存储过程，但不允许他们创建存储过程。

### 使用存储过程

#### 执行存储过程

MySQL称存储过程的执行为调用，因此MySQL执行存储过程的语句为`CALL`。`CALL`接受存储过程的名字以及需要传递给它的任意参数。

```
call productpricing(@pricelow, @pricehigh, @priceaverage);
```

#### 创建存储过程

```
CREATE PROCEDURE productpricing()
BEGIN
    SELECT Avg(prod_price) AS proceverage
    FROM products;
END;
```

> **mysql命令行客户机的分隔符** 默认的MySQL语句分隔符为`;`（正如你已经在迄今为止所使用的MySQL语句中所看到的那样）。mysql命令行实用程序也使用`;`作为语句分隔符。如果命令行实用程序要解释存储过程自身内的`;`字符，则它们最终不会成为存储过程的成分，这会使存储过程中的SQL出现句法错误。 解决办法是临时更改命令行实用程序的语句分隔符，如下所示：
>
> ```
> DELIMITER //
>
> CREATE PROCEDURE productpricing()
> BEGIN
>     SELECT Avg(prod_price) AS proceverage
>     FROM products;
> END //
>
> DELIMITER ;
> ```
>
> 其中，`DELIMITER //`告诉命令行实用程序使用`//`作为新的语句结束分隔符，可以看到标志存储过程结束的`END`定义为`END //`而不是`END;`。这样，存储过程体内的`;`仍然保持不动，并且正确地传递给数据库引擎。最后，为恢复为原来的语句分隔符，可使用`DELIMITER ;`。 除`\`符号外，任何字符都可以用作语句分隔符。

执行

```
CALL productpricing();
```

因为存储过程实际上是一种函数，所以存储过程名后需要有`()`符号（即使不传递参数也需要）。

#### 删除存储过程

```
DROP PROCEDURE productpricing;
```

没有`()`

> 如果指定的过程不存在，则`DROP PROCEDURE`将产生一个错误。当过程存在想删除它时（如果过程不存在也不产生错误）可使用`DROP PROCEDURE IF EXISTS`。

### 使用参数

一般，存储过程并不显示结果，而是把结果返回给你指定的变量（**内存**中一个特定的位置，用来临时存储数据）。

```
CREATE PROCEDURE productpricing(OUT pl DECIMAL(8,2), OUT ph DECIMAL(8,2), OUT pa
DECIMAL(8,2))
BEGIN
    SELECT Min(prod_price) INTO pl FROM products;
    SELECT Max(prod_price) INTO ph FROM products;
    SELECT Avg(prod_price) INTO pa FROM products;
END;
```

关键字`OUT`指出相应的参数用来从存储过程传出一个值（返回给调用者）。MySQL支持`IN`（传递给存储过程）、`OUT`（从存储过程传出，如这里所用）和`INOUT`（对存储过程传入和传出）类型的参数。

> 参数的类型： 存储过程的参数允许的数据类型与表中使用 的数据类型相同。
>
> 注意，记录集不是允许的类型，因此，不能通过一个参数返回多个行和列。这就是前面的例子为什么要使用3个参数（和3 条SELECT语句）的原因。

> **变量名：所有MySQL变量都必须以@开始。**

为了获得3个值:

```
SELECT @pricelow, @pricehigh, @priceaverage;
```

### 使用`IN`和`OUT`参数

```
CREATE PROCEDURE ordertotal(IN onumber INT, OUT ototal DECIMAL(8,2))
BEGIN
    SELECT Sum(item_price * quantity)
    FROM orderitems
    WHERE order_num = onumber
    INTO ototal;
END;
```

### 建立智能存储过程

考虑这个场景。你需要获得与以前一样的订单合计，但需要对合计增加营业税，不过只针对某些顾客（或许是你所在州中那些顾客）。那么， 你需要做下面几件事情:

- 获得合计（与以前一样）；
- 把营业税有条件地添加到合计；
- 返回合计（带或不带税）

```
-- Name: ordertotal
-- Parameters: onumber = order number
--             taxable = 0 if not taxable
--             ototal  = order total variable

CREATE PROCEDURE ordertotal(
    IN onumber INT,
```

```
    IN taxable BOOLEAN,
    OUT ototal DECIMAL(8,2)
)COMMENT 'Obtain order total, optionally adding tax'
BEGIN
    -- Declare variable for total
    DECLARE total DECIMAL(8,2);
    -- Declare tax percentage
    DECLARE taxrate INT DEFAULT 6;

    --Get the order total
    SELECT Sum(item_price * quantity)
    FROM orderitems
    WHERE order_num = onumber
    INTO total;

    --Is this taxable?
    IF taxable THEN
        -- Yes, so add taxrate to the total
        SELECT total + (total / 100 * taxrate) INTO total;
    END IF;

    -- And Finally, save to out variable
    SELECT total INTO ototal;

END;
```

首先，增加了注释（前面放置--）。 在存储过程复杂性增加时，这样做特别重要。添加了另外一个参数taxable，它是一个布尔值（如果要增加税则为真，否则为假）。在存储过程体中，用`DECLARE`语句定义了两个局部变量。`DECLARE`要求指定变量名和数据类型，它也支持可选的默认值（这个例子中的taxrate的默认被设置为6%）。

> 本例子中的存储过程在`CREATE PROCEDURE`语 句中包含了一个`COMMENT`值。它不是必需的，但如果给出，将在`SHOW PROCEDURE STATUS`的结果中显示。

调用

```
CALL ordertotal(20005, 0, @total);
SELECT @total;
```

BOOLEAN值指定为1表示真，指定为0表示假（实际上，非零值 都考虑为真，只有0被视为假）。`TRUE`和`FALSE`也可以

#### 检查存储过程

```
SHOW CREATE PROCEDURE ordertotal;
```

为了获得包括何时、由谁创建等详细信息的存储过程列表，使用`SHOW PROCEDURE STATUS`。

> `SHOW PROCEDURE STATUS`列出所有存储过程。为限制其输出，可使用`LIKE`指定一个过滤模式
>
> ```
> SHOW PROCEDURE STATUS LIKE 'ordertotal';
> ```

## 游标

游标（cursor）是一个存储在MySQL服务器上的数据库查询，它不是一条`SELECT`语句，而是被该语句检索出来的结果集。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据

> **不像多数DBMS，MySQL游标只能用于存储过程（和函数）。**

**使用游标**

- 在能够使用游标前，必须声明（定义）它。这个过程实际上没有检索数据，它只是定义要使用的`SELECT`语句。
- 一旦声明后，必须打开游标以供使用。这个过程用前面定义的 SELECT语句把数据实际检索出来。
- 对于填有数据的游标，根据需要取出（检索）各行。
- 在结束游标使用时，必须关闭游标。

在声明游标后，可根据需要频繁地打开和关闭游标。在游标打开后， 可根据需要频繁地执行取操作。

#### 创建游标

游标用`DECLARE`语句创建。`DECLARE`命名游标，并定义相应的`SELECT`语句，根据需要带`WHERE`和其他子句。

```
CREATE PROCEDURE processorders()
BEGIN
    DECLARE ordernumbers CURSOR
    FOR
    SELECT order_num FROM orders;
END;
```

存储过程处理完成后，游标就消失（因为它局限于存储过程）。

在定义游标之后，可以打开它。

#### 打开和关闭游标

打开游标:

游标用`OPEN CURSOR`语句来打开

```
OPEN ordernumbers;
```

在处理`OPEN`语句时执行查询，存储检索出的数据以供浏览和滚动。

关闭游标:

```
CLOSE ordernumbers;
```

`CLOSE`释放游标使用的所有内部内存和资源，因此在每个游标不再需要时都应该关闭

在一个游标关闭后，如果没有重新打开，则不能使用它。但是，使用声明过的游标不需要再次声明，用`OPEN`语句打开它就可以了。

> 如果你不明确关闭游标，MySQL将会在到达`END`语句时自动关闭它。

```
CREATE PROCEDURE processorders()
BEGIN
    -- Declare the cursor
    DECLARE ordernumbers CURSOR
    FOR
    SELECT order_num FROM orders;

    -- Open the cursor
    OPEN ordernumbers;

    -- Close the cursor
    CLOSE ordernumbers;
END;
```

这个存储过程声明、打开和关闭一个游标。但对检索出的数据什么也没做。

#### 使用游标数据

在一个游标被打开后，可以使用`FETCH`语句分别访问它的每一行。 `FETCH`指定检索什么数据（所需的列），检索出来的数据存储在什么地方。 它还向前移动游标中的内部行指针，使下一条`FETCH`语句检索下一行（不重复读取同一行）

```
CREATE PROCEDURE processorders()
BEGIN
    -- Declare local variables
    DECLARE o INT;

    -- Declare the cursor
    DECLARE ordernumbers CURSOR
    FOR
    SELECT order_num FROM orders;

    -- Open the cursor
    OPEN ordernumbers;

    -- Get order number
    FETCH ordernumbers INTO o;

    -- Close the cursor
    CLOSE ordernumbers;
END;
```

`FETCH`用来检索当前行的order_num列（将自动从第一行开始）到一个名为o的局部声明的变量中。

```
CREATE PROCEDURE processorders()
BEGIN
-- Declare local variables
DECLARE done BOOLEAN DEFAULT 0;
DECLARE o INT;

-- Declare the cursor
DECLARE ordernumbers CURSOR
FOR
SELECT order_num FROM orders;
```

```
-- Declare continue handler
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

-- Open the cursor
OPEN ordernumbers;

-- Loop through all rows
REPEAT

-- Get order number
FETCH ordernumbers INTO o;

-- End of loop
UNTIL done END REPEAT;

-- Close the cursor
CLOSE ordernumbers;

END;
```

这个例子中的`FETCH`是在`REPEAT`内，因此它反复执行直到done为真（由`UNTIL done END REPEAT;`规定）。为使它起作用，用一个`DEFAULT 0`（假，不结束）定义变量done。

done被设置为真：`DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;`

这条语句定义了一个`CONTINUE HANDLER`，它是在条件出现时被执行的代码。这里，它指出当`SQLSTATE '02000'`出现时，`SET done=1`。`SQLSTATE '02000'`是一个未找到条件，当`REPEAT`由于没有更多的行供循环而不能继续时，出现这个条件

> `DECLARE`语句的发布存在特定的次序。 用`DECLARE`语句定义的局部变量必须在**定义任意游标或句柄之前定义**，而**句柄必须在游标之后定义**。不遵守此顺序将产 生错误消息

完整版：

```
CREATE PROCEDURE processorders()
BEGIN
    -- Declare local variables
    DECLARE done BOOLEAN DEFAULT 0;
    DECLARE o INT;
    DECLARE t DECIMAL(8,2);

    -- Declare the cursor
    DECLARE ordernumbers CURSOR
    FOR
    SELECT order_num FROM orders;

    -- Declare continue handler
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;

    -- Create a table to store the results
    CREATE TABLE IF NOT EXISTS ordertotals(order_num INT, total DECIMAL(8,2));

    -- Open the cursor
    OPEN ordernumbers;

    -- Loop through all rows
```

```
    REPEAT

        -- Get order number
        FETCH ordernumbers INTO o;

        -- Get the total for this order
        CALL ordertotal(o, 1, t);

        -- Insert order and total into ordertotals
        INSERT INTO ordertotals(order_num, total)
        VALUES(o, t);

    -- End of loop
    UNTIL done END REPEAT;

    -- Close the cursor
    CLOSE ordernumbers;
END;
```

我们增加了另一个名为t的变量（存储每个订单的合计）。此存储过程还在运行中创建了一个新表（如果它不 存在的话），名为ordertotals。这个表将保存存储过程生成的结果。`FETCH`像以前一样取每个order_num，然后用`CALL`执行另一个存储过程来计算每个订单的带税的合计（结果存储到t）。最后，用`INSERT`保存每个订单的订单号和合计。

## 触发器

在某个表发生更改时自动处理。这确切地说就是触发器。触发器是MySQL响应以下任意语句而自动执行的一条MySQL语句（或位于`BEGIN`和`END`语句之间的一组语句）：

- DELETE;
- INSERT;
- UPDATE。

### 创建触发器

- 唯一的触发器名;
- 触发器关联的表;
- 触发器应该响应的活动（`DELETE`、`INSERT`或`UPDATE`）;
- 触发器何时执行（处理之前或之后）

> **在MySQL 5中，触发器名必须在每个表中唯一，但不是在每个数据库中唯一。**这表示同一 数据库中的两个表可具有相同名字的触发器。这在其他每个数据库触发器名必须唯一的DBMS中是不允许的，而且以后的MySQL版本很可能会使命名规则更为严格。因此，现在最好是在数据库范围内使用唯一的触发器名。

```
CREATE TRIGGER newproduct AFTER INSERT ON products
FOR EACH ROW SELECT 'Product added' INTO @asd;
```

MYSQL5以后，不允许触发器返回任何结果，因此使用 into @变量名，将结果赋值到变量中，用select调用即可

> **只有表才支持触发器，视图不支持（临时表也不支持）**

**触发器按每个表每个事件每次地定义，每个表每个事件每次只允许一个触发器。**因此，每个表最多支持6个触发器（每条`INSERT`、`UPDATE`和`DELETE`的之前和之后）。单一触发器不能与多个事件或多个表关联，所以，如果你需要一个对`INSERT`和`UPDATE`操作执行的触发器，则应该定义两个触发器。

> 如果`BEFORE`触发器失败，则MySQL将不执行请求的操作。此外，如果`BEFORE`触发器或语句本身失败，MySQL 将不执行`AFTER`触发器（如果有的话）。

#### 删除触发器

```
DROP TRIGGER newproduct;
```

触发器不能更新或覆盖。为了修改一个触发器，必须先删除它， 然后再重新创建。

#### 使用触发器

##### INSERT触发器

`INSERT`触发器在`INSERT`语句执行之前或之后执行。

- 在`INSERT`触发器代码内，可引用一个名为NEW的虚拟表，访问被插入的行；
- 在`BEFORE INSERT`触发器中，NEW中的值也可以被更新（允许更改被插入的值）；
- 对于`AUTO_INCREMENT`列，NEW在`INSERT`执行之前包含0，在`INSERT`执行之后包含新的自动生成值。

```
CREATE TRIGGER neworder AFTER INSERT ON orders
FOR EACH ROW SELECT NEW.order_num INTO @porder_num;
```

由于 order_num 是`AUTO_INCREMENT`，所以`BEFORE INSERT`触发器是0

如果是 order_date（不是`AUTO_INCREMENT`），那么返回的就是插入的值

在插入一个新订单到orders表时，MySQL生成一个新订单号并保存到order_num中。触发器从`NEW.order_num`取得这个值并返回它。此触发器必须按照`AFTER INSERT`执行，因为在`BEFORE INSERT`语句执行之前，新order_num还没有生成。对于orders的每次插入使用这个触发器将总是返回新的订单号

> 通常，将`BEFORE`用于数据验证和净化（目 的是保证插入表中的数据确实是需要的数据）。本提示也适用于`UPDATE`触发器

#### DELETE触发器

`DELETE`触发器在`DELETE`语句执行之前或之后执行。

- 在`DELETE`触发器代码内，你可以引用一个名为OLD的虚拟表，访问被删除的行；
- OLD中的值全都是**只读的**，不能更新。

```
CREATE TRIGGER deleteorder BEFORE DELETE ON orders
FOR EACH ROW
BEGIN
INSERT INTO archive_orders(order_num, order_date, cust_id)
VALUES(OLD.order_num, OLD.order_date, OLD.cust_id);
END;
```

它使用一条INSERT语句 将OLD中的值（要被删除的订单）保存到一个名为archive_ orders的存档表中（为实际使用这个例子，你需要用与orders相同的列创建一个名为archive_orders的表）

> 触发器deleteorder使用`BEGIN`和`END`语句标记触发器体。这在此例子中并不是必需的，不过也没有害处。使用**BEGIN END块**的好处是触发器**能容纳多条SQL语句**（在**BEGIN END块**中一条挨着一条）。

#### UPDATE触发器

`UPDATE`触发器在`UPDATE`语句执行之前或之后执行。

- 在`UPDATE`触发器代码中，你可以引用一个名为OLD的虚拟表访问以前（UPDATE语句前）的值，引用一个名为NEW的虚拟表访问新更新的值；
- 在`BEFORE UPDATE`触发器中，NEW中的值可能也被更新（允许更改将要用于`UPDATE`语句中的值）；
- OLD中的值全都是只读的，不能更新

```
CREATE TRIGGER updatevendor BEFORE UPDATE ON vendors
FOR EACH ROW SET NEW.vend_state = Upper(NEW.vend_state);
```

每次更新一个行时，`NEW.vend_state`中的值（将用来更新表行的值）都用`Upper(NEW.vend_state)`替换

#### 关于触发器的进一步介绍

- 创建触发器可能需要特殊的安全访问权限，但是，触发器的执行是自动的。如果`INSERT`、`UPDATE`或`DELETE`语句能够执行，则相关的触发器也能执行。
- 应该用触发器来保证数据的一致性（大小写、格式等）。在触发器 中执行这种类型的处理的优点是它总是进行这种处理，而且是透明地进行，与客户机应用无关。
- 触发器的一种非常有意义的使用是创建审计跟踪。使用触发器， 把更改（如果需要，甚至还有之前和之后的状态）记录到另一个表非常容易。
- 遗憾的是，MySQL触发器中**不支持**`CALL`语句。这表示不能从触发器内调用存储过程。所需的存储过程代码需要复制到触发器内。

## 事务处理

事务处理（transaction processing）可以用来维护数据库的完整性，它 保证成批的MySQL操作要么完全执行，要么完全不执行。

- 事务（transaction）指一组SQL语句；
- 回退（rollback）指撤销指定SQL语句的过程；
- 提交（commit）指将未存储的SQL语句结果写入数据库表；
- 保留点（savepoint）指事务处理中设置的临时占位符（place-holder），你可以对它发布回退（与回退整个事务处理不同）。

**控制事物处理**

管理事务处理的关键在于将SQL语句组分解为逻辑块，并明确规定数据何时应该回退，何时不应该回退

MySQL使用下面的语句来标识事务的开始：

```
START TRANSACTION;
```

**使用ROLLBACK**

MySQL的`ROLLBACK`命令用来回退（撤销）MySQL语句

```
SELECT * FROM ordertotals;
START TRANSACTION;
DELETE FROM ordertotals;
SELECT * FROM ordertotals;
ROLLBACK;
SELECT * FROM ordertotals;
```

显然，`ROLLBACK`只能在一个事务处理内使用（在执行一条`START TRANSACTION`命令之后）

> 事务处理用来管理`INSERT`、`UPDATE`和`DELETE`语句。你不能回退`SELECT`语句。（这样做也没有什么义。）你不能回退`CREATE`或`DROP`操作。事务处理块中可以使用 这两条语句，但如果你执行回退，它们不会被撤销

**使用COMMIT**

一般的MySQL语句都是直接针对数据库表执行和编写的。这就是所谓的隐含提交（implicit commit），即提交（写或保存）操作是自动进行的。

但是，在事务处理块中，提交不会隐含地进行。为进行明确的提交， 使用`COMMIT`语句

```
START TRANSACTION;
DELETE FROM orderitems WHERE order_num = 20010;
DELETE FROM orders WHERE order_num = 20010;
COMMIT;
```

最后的COMMIT语句仅在不出错时写出更改。如果第一条`DELETE`起作用，但第二条失败，则`DELETE`不会提交（实际上， 它是被自动撤销的）。

当`COMMIT`或`ROLLBACK`语句执行后，事务会自动关闭（将来的更改会隐含提交）。

**使用保留点**

为了支持回退部分事务处理，必须能在事务处理块中合适的位置放置占位符。这样，如果需要回退，可以回退到某个占位符。 这些占位符称为保留点。

为了创建占位符，可如下使用`SAVEPOINT`语句

```
SAVEPOINT delete1;
```

每个保留点都取标识它的唯一名字，以便在回退时，MySQL知道要回退到何处。为了回退到本例给出的保留点

```
ROLLBACK delete1;
```

> 可以在MySQL代码中设置任意多的保留点，越多越好。为什么呢？因为保留点越多，你就越能按自己的意愿灵活地进行回退

> 保留点在事务处理完成（执行一条`ROLLBACK`或`COMMIT`）后自动释放。自MySQL5以来，也可以用`RELEASE SAVEPOINT`明确地释放保留点

#### 更改默认的提交行为

正如所述，默认的MySQL行为是自动提交所有更改。换句话说，任何时候你执行一条MySQL语句，该语句实际上都是针对表执行的，而且所做 的更改立即生效。为指示MySQL不自动提交更改

```
SET autocommit = 0;
```

`autocommit`标志决定是否自动提交更改，不管有没有`COMMIT`语句。设置`autocommit`为 **0**（假）指示MySQL不自动提交更改 （直到`autocommit`被设置为真为止）

> `autocommit`标志是针对**每个连接**而不是服务器的。

## 全球化和本地化

#### 字符集和校对顺序

数据库表被用来存储和检索数据。不同的语言和字符集需要以不同的方式存储和检索。因此，MySQL需要适应不同的字符集（不同的字母 和字符），适应不同的排序和检索数据的方法。

在讨论多种语言和字符集时，将会遇到以下重要术语：

- 字符集为字母和符号的集合；
- 编码为某个字符集成员的内部表示；
- 校对为规定字符如何比较的指令。

在MySQL的正常数据库活动（`SELECT`、`INSERT`等）中，不需要操心太多的东西。使用何种字符集和校对的决定在服务器、数据库和表级进行

#### 使用字符集和校对顺序

MySQL支持众多的字符集。为查看所支持的字符集完整列表

```
SHOW CHARACTER SET;
```

这条语句显示所有可用的字符集以及每个字符集的描述和默认校对

为了查看所支持校对的完整列表

```
SHOW COLLATION;
```

此语句显示所有可用的校对，以及它们适用的字符集

通常系统管理在安装时定义一个默认的字符集和校对。此外，也可以在创建数据库时，指定默认的字符集和校对。为了确定所用的字符集 和校对，可以使用以下语句

```
SHOW VARIABLES LIKE 'character%';
SHOW VARIABLES LIKE 'collation%';
```

实际上，字符集很少是服务器范围（甚至数据库范围）的 设置。不同的表，甚至不同的列都可能需要不同的字符集，而且两者都可以在创建表时指定。 为了给表指定字符集和校对，可使用带子句的`CREATE TABLE`

```
CREATE TABLE mytable(
    column1 INT,
    column2 VARCHAR(10)
)DEFAULT CHARACTER SET hebrew
COLLATE hebrew_general_ci;
```

指定一个字符集（`CHARACTER SET`）和一个校对顺序（`COLLATE`）。

一般，MySQL如 下确定使用什么样的字符集和校对。

- 如果指定`CHARACTER SET`和`COLLATE`两者，则使用这些值。
- 如果只指定`CHARACTER SET`，则使用此字符集及其默认的校对（如`SHOW CHARACTER SET`的结果中所示）。
- 如果既不指定`CHARACTER SET`，也不指定`COLLATE`，则使用数据库默认。

除了能指定字符集和校对的表范围外，MySQL还允许对每个列设置它们

```
CREATE TABLE mytable(
column1 INT,
column2 VARCHAR(10),
column3 VARCHAR(10) CHARACTOR SET latin1 COLLATE latin1_general_ci
)DEFAULT CHARACTER SET hebrew
 COLLATE hebrew_general_ci;
```

对整个表以及一个特定的列指定了`CHARACTER SET`和`COLLATE`。

如果你需要用与创建表时不同的校对顺序排序特定的`SELECT`语 句，可以在`SELECT`语句自身中进行

```
SELECT * FROM customers
ORDER BY cust_name COLLAET latin1_general_cs;
```

除了这里看到的在`ORDER BY`子句中使用以外，`COLLATE`还可以用于`GROUP BY`、`HAVING`、**聚集函数**、**别名**等

最后，值得注意的是，如果绝对需要，串可以在字符集之间进行转换。为此，使用`Cast()`或`Convert()`函数

## 安全管理

### 访问控制

MySQL服务器的安全基础是：用户应该对他们需要的数据具有适当的访问权，既不能多也不能少。

> 重要的是注意到，访问控制的目的不仅仅 是防止用户的恶意企图。数据梦魇更为常见的是无意识错误 的结果，如错打MySQL语句，在不合适的数据库中操作或其 他一些用户错误。通过保证用户不能执行他们不应该执行的 语句，访问控制有助于避免这些情况的发生。

> 应该严肃对待root登录的使用。仅在绝对需 要时使用它（或许在你不能登录其他管理账号时使用）。不应该在日常的MySQL操作中使用root

### 管理用户

MySQL用户账号和信息存储在名为mysql的MySQL数据库中。需要直接访问它的时机之一是在需要获得所有用户账号列表时。

```
use mysql;
SELECT user FROM user;
```

mysql数据库有一个名为user的表，它包含所有用户账号。user表有一个名为user的列，它存储用户登录名。

> 试验对用户账号和权限进行更改的 最好办法是打开多个数据库客户机（如mysql命令行实用程序的多个副本），一个作为管理登录，其他作为被测试的用户登录。

#### 创建用户账号

为了创建一个新用户账号，使用`CREATE USER`语句

```
CREATE USER ben IDENTIFIED BY 'p@$$wOrd';
```

> `IDENTIFIED BY`指定的口令为纯文本，MySQL将在保存到`user`表之前对其进行**加密**。为了作为散列值指定口令，使用`IDENTIFIED BY PASSWORD`。

> `GRANT`语句也可以创建用户账号，但一般来说`CREATE USER`是最清楚和最简单的句子。 此外，也可以通过直接插入行到`user`表来增加用户，不过为安全起见，一般不建议这样做。MySQL用来存储用户账号信息的表（以及表模式等）极为重要，对它们的任何毁坏都可能严重地伤害到MySQL服务器。因此，相对于直接处理来说，最好是用标记和函数来处理这些表。

重新命名一个用户账号，使用`RENAME USER`语句

```
RENAME USER ben TO bforta;
```

> 仅MySQL 5或之后的版本支持`RENAME USER`。 为了在以前的MySQL中重命名一个用户，可使用`UPDATE`直接更新`user`表

#### 删除用户账户

删除一个用户账号（以及相关的权限），使用`DROP USER`语句

```
DROP USER bforta;
```

> 自MySQL 5以来，`DROP USER`**删除用户账号和所有相关的账号权限**。在MySQL 5以前，`DROP USER`只能用来**删除用户账号**，**不能删除相关的权限**。因此，如果使用旧版本的MySQL，需要先用`REVOKE`删除与账号相关的权限，然后再用`DROP USER`删除账号

### 设置访问权限

在创建用户账号后，必须接着分配访问权限。新创建的用户账号没有访问权限。它们能登录MySQL，但不能看到数据，不能执行任何数据库操作。

为看到赋予用户账号的权限，使用`SHOW GRANTS FOR`

```
SHOW GRANTS FOR bforta;
```

```
+-------------------------------------------------+
| Grants for bforta@%                             |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'bforta'@'%'              |
+-------------------------------------------------+
```

输出结果显示用户bforta有一个权限`USAGE ON \*.\*`。`USAGE`表示根本没有权限（我知道，这不很直观），所以，此结果表示在 任意数据库和任意表上对任何东西没有权限。

> **用户定义为user@host**　MySQL的权限用**用户名**和**主机名**结合定义。如果不指定主机名，则使用默认的主机名**%**（授予用户访问权限而不管主机名）

######

#### 添加权限

为设置权限，使用`GRANT`语句。`GRANT`要求你至少给出以下信息

- 要授予的权限；
- 被授予访问权限的数据库或表；
- 用户名。

`GRANT`的用法

```
GRANT SELECT ON crashcourse.* TO bforta;
```

此`GRANT`允许用户在`crashcourse.*`（`crashcourse`数据库的所有表）上使用`SELECT`。通过只授予`SELECT`访问权限，用户`bforta`对`crashcourse`数据库中的所有数据具有只读访问权限

```
SHOW GRANTS FOR bforta;
```

每个`GRANT`添加（或更新）用户的一个权限。MySQL读取所有授权，并根据它们确定权限。

#### 撤销权限

`GRANT`的反操作为`REVOKE`，用它来撤销特定的权限。

```
REVOKE SELECT ON crashcourse.* FROM bforta;
```

这条`REVOKE`语句取消刚赋予用户`bforta`的`SELECT`访问权限。被撤销的访问权限必须存在，否则会出错。

`GRANT`和`REVOKE`可在几个层次上控制访问权限：

- **整个服务器**，使用`GRANT ALL`和`REVOKE ALL`；
- **整个数据库**，使用`ON database.*`；
- **特定的表**，使用`ON database.table`；

- **特定的列**；
- **特定的存储过程。**

表28-1　权限

| 权　　限 | 说　　明 |
| --- | --- |
| ALL | 除GRANT OPTION外的所有权限 |
| ALTER | 使用ALTER TABLE |
| ALTER ROUTINE | 使用ALTER PROCEDURE和DROP PROCEDURE |
| CREATE | 使用CREATE TABLE |
| CREATE ROUTINE | 使用CREATE PROCEDURE |
| CREATE TEMPORARY TABLES | 使用CREATE TEMPORARY TABLE |
| CREATE USER | 使用CREATE USER、DROP USER、RENAME USER和REVOKE ALL PRIVILEGES |
| CREATE VIEW | 使用CREATE VIEW |
| DELETE | 使用DELETE |
| DROP | 使用DROP TABLE |
| EXECUTE | 使用CALL和存储过程 |
| FILE | 使用SELECT INTO OUTFILE和LOAD DATA INFILE |
| GRANT OPTION | 使用GRANT和REVOKE |
| INDEX | 使用CREATE INDEX和DROP INDEX |
| INSERT | 使用INSERT |
| LOCK TABLES | 使用LOCK TABLES |
| PROCESS | 使用SHOW FULL PROCESSLIST |
| RELOAD | 使用FLUSH |
| REPLICATION CLIENT | 服务器位置的访问 |
| REPLICATION SLAVE | 由复制从属使用 |

| 权　　限 | 说　　明 |
| --- | --- |
| SELECT | 使用SELECT |
| SHOW DATABASES | 使用SHOW DATABASES |
| SHOW VIEW | 使用SHOW CREATE VIEW |
| SHUTDOWN | 使用mysqladmin shutdown（用来关闭MySQL） |
| SUPER | 使用CHANGE MASTER、KILL、LOGS、PURGE、MASTER和SET GLOBAL。还允许mysqladmin调试登录 |
| UPDATE | 使用UPDATE |
| USAGE | 无访问权限 |

使用`GRANT`和`REVOKE`，再结合上面表中列出的权限，你能对用户可以 就你的宝贵数据做什么事情和不能做什么事情具有完全的控制。

> 在使用`GRANT`和`REVOKE`时，**用户账号必须存在**， 但**对所涉及的对象没有这个要求。**这允许管理员在创建数据库和表之前设计和实现安全措施。 这样做的副作用是，当某个数据库或表被删除时（用`DROP`语句），相关的访问权限仍然存在。而且，如果将来重新创建该数据库或表，这些权限仍然起作用。

> 可通过列出各权限并用逗号分隔，将多条 GRANT语句串在一起，如下所示：
>
> ```
> GRANT SELECT, INSERT ON crashcourse.* TO bforta;
> ```

#### 更改密码

**MYSQL8.0之前**

为了更改用户口令（密码），可使用`SET PASSWORD`语句。

```
SET PASSWORD FOR bforta = Password('n3w p@$$wOrd');
```

`SET PASSWORD`更新用户口令。新口令必须传递到`Password()`函 数进行加密。

`SET PASSWORD`还可以用来设置你自己的口令

```
SET PASSWORD = Password('n3w p@$$wOrd');
```

在不指定用户名时，`SET PASSWORD`更新当前登录用户的口令。

**MYSQL 8.0之后**

```
ALTER USER 'bforta' IDENTIFIED WITH mysql_native_password BY "n3w p@$$wOrd";
```

或

```
ALTER user 'bforta' IDENTIFIED BY 'n3w p@$$wOrd';
```

更改之后要`FLUSH privileges;`

## 数据库维护

**备份数据**

由于MySQL数据库是基于磁盘的文件，普通的备份系统和例程就能备份MySQL的数据。但是，由于这些文件总是处于打开和使用状态，普通的文件副本备份不一定总是有效

- 使用命令行实用程序mysqldump转储所有数据库内容到某个外部文件。在进行常规备份前这个实用程序应该正常运行，以便能正确地备份转储文件。
- 可用命令行实用程序mysqlhotcopy从一个数据库复制所有数据 （并非所有数据库引擎都支持这个实用程序）。
- 可以使用MySQL的`BACKUP TABLE`或`SELECT INTO OUTFILE`转储所 有数据到某个外部文件。这两条语句都接受将要创建的系统文件名，此系统文件必须**不存在**，否则会出错。数据可以用`RESTORE TABLE`来复原。

> 为了保证所有数据被写到磁盘（包括索引数据），可能需要在进行备份前使用`FLUSH TABLES`语句。

**进行数据库维护**

- `ANALYZE TABLE`，用来检查表键是否正确。

  ```
  ANALYZE TABLE orders;
  ```

- `CHECK TABLE`用来针对许多问题对表进行检查。在`MyISAM`表上还对**索引**进行检查。`CHECK TABLE`支持一系列的用于`MyISAM`表的方式。

  `CHANGED`检查自最后一次检查以来改动过的表。`EXTENDED`执行最彻底的检查，`FAST`只检查未正常关闭的表，`MEDIUM`检查所有被删 除的链接并进行键检验，`QUICK`只进行快速扫描。

  **发现和修复问题**

  ```
  CHECK TABLE orders, orderitems;
  ```

  ```
  +------------------------+-------+----------+-----------------------+
  | Table                  | Op    | Msg_type | Msg_text              |
  +------------------------+-------+----------+-----------------------+
  | crashcourse.orders     | check | status   | OK                    |
  | crashcourse.orderitems | check | warning  | Table is marked as    |
  |                        |       |          | crashed               |
  | crashcourse.orderitems | check | status   | OK                    |
  +------------------------+-------+----------+-----------------------+
  ```

- 如果MyISAM表访问产生不正确和不一致的结果，可能需要用`REPAIR TABLE`来修复相应的表。这条语句不应该经常使用，如果需要经常使用，可能会有更大的问题要解决。
- 如果从一个表中删除大量数据，应该使用`OPTIMIZE TABLE`来收回所用的空间，从而优化表的性能。

### 诊断启动问题

服务器启动问题通常在对MySQL配置或服务器本身进行更改时出现。MySQL在这个问题发生时报告错误，但由于多数MySQL服务器是作 为系统进程或服务自动启动的，这些消息可能看不到。 在排除系统启动问题时，首先应该尽量用手动启动服务器。MySQL服务器自身通过在命令行上执行`mysqld`启动。下面是几个重要的`mysqld`命令行选项：

- --help显示帮助——一个选项列表；
- --safe-mode装载减去某些最佳配置的服务器；
- --verbose显示全文本消息（为获得更详细的帮助消息与--help 联合使用）；
- --version显示版本信息然后退出。

### 查看日志文件

- **错误日志**。它包含启动和关闭问题以及任意关键错误的细节。此日志通常名为`hostname.err`，位于`data`目录中。此日志名可用 `--log-error`命令行选项更改。
- **查询日志**。它记录所有MySQL活动，在诊断问题时非常有用。此日志文件可能会很快地变得非常大，因此不应该长期使用它。此 日志通常名为`hostname.log`，位于`data`目录中。此名字可以用`--log`命令行选项更改。
- **二进制日志**。它记录更新过数据（或者可能更新过数据）的所有语句。此日志通常名为`hostname-bin`，位于`data`目录内。此名字 可以用`--log-bin`命令行选项更改。注意，这个日志文件是MySQL5中添加的，以前的MySQL版本中使用的是更新日志。
- **缓慢查询日志**。顾名思义，此日志记录执行缓慢的任何查询。**这个日志在确定数据库何处需要优化很有用**。此日志通常名为`hostname-slow.log`，位于`data`目录中。此名字可以用`--log-slow-queries`命令行选项更改。

在使用日志时，可用`FLUSH LOGS`语句来刷新和重新开始所有日志文件

## 改善性能

- 首先，MySQL（与所有DBMS一样）具有特定的硬件建议。在学习和研究MySQL时，使用任何旧的计算机作为服务器都可以。但对用于生产的服务器来说，应该坚持遵循这些硬件建议。
- 一般来说，关键的生产DBMS应该运行在自己的专用服务器上。
- MySQL是用一系列的默认设置预先配置的，从这些设置开始通常是很好的。但过一段时间后你可能需要**调整内存分配、缓冲区大小**等。（为查看当前设置，可使用`SHOW VARIABLES;`和`SHOW STATUS;`。）
- MySQL一个多用户多线程的DBMS，换言之，它经常同时执行多个任务。如果这些任务中的某一个执行缓慢，则所有请求都会执行缓慢。如果你遇到显著的性能不良，可使用`SHOW PROCESSLIST`显示所有活动进程（以及它们的线程ID和执行时间）。你还可以用`KILL`命令终结某个特定的进程（使用这个命令需要作为管理员登录）。
- 总是有不止一种方法编写同一条`SELECT`语句。应该试验联结、并、 子查询等，找出最佳的方法。
- 使用`EXPLAIN`语句让MySQL解释它将如何执行一条`SELECT`语句。
- 一般来说，**存储过程**执行得比一条一条地执行其中的各条MySQL语句快。
- 应该总是使用正确的数据类型。
- 决不要检索比需求还要多的数据。换言之，不要用**SELECT ***（除非你真正需要每个列）。
- 有的操作（包括`INSERT`）支持一个可选的`DELAYED`关键字，如果使用它，将把控制立即返回给调用程序，并且一旦有可能就实际执行该操作。
- 在导入数据时，应该关闭自动提交。你可能还想删除索引（包括`FULLTEXT`索引），然后在导入完成后再重建它们。
- 必须索引数据库表以改善数据检索的性能。确定索引什么不是一件微不足道的任务，需要分析使用的`SELECT`语句以找出重复的`WHERE`和`ORDER BY`子句。如果一个简单的`WHERE`子句返回结果所花的时间太长，则可以断定其中使用的列（或几个列）就是需要索 引的对象。
- 你的`SELECT`语句中有一系列复杂的`OR`条件吗？通过使用多条`SELECT`语句和连接它们的`UNION`语句，你能看到极大的性能改进。
- 索引改善数据检索的性能，但损害数据插入、删除和更新的性能。 如果你有一些表，它们收集数据且不经常被搜索，则在有必要之 前不要索引它们。（索引可根据需要添加和删除。）
- `LIKE`很慢。一般来说，最好是使用`FULLTEXT`而不是`LIKE`。
- 数据库是不断变化的实体。一组优化良好的表一会儿后可能就面目全非了。由于表的使用和内容的更改，理想的优化和配置也会改变。
- 最重要的规则就是，每条规则在某些条件下都会被打破。