# 《MySQL必知必会》读书笔记

# 1、导读

**本文是阿秀个人学习SQL过程中，学习《MySQL必知必会》所记录的读书笔记，100%原创。**

书名虽然是《MySQL必知必会》，但是其实并不是讲MySQL的，而是讲SQL的，SQL的学习和练习方法已经分享在阿秀的学习圈中了，感兴趣的可以去看看。

建议各位在学习本书的时候，也在本地建一个数据库，挨个建表，然后自己按照书中例子敲一遍过来，记得绝对清楚，因为好记性是不如烂笔头的。

MySQL纯净版PDF文件分享

MySQL必知必会（文字版）.pdf

# 2、建表命令

SQL | 复制代码

```sql
########################################
# MySQL Crash Course
# http://www.forta.com/books/0672327120/
# Example table creation scripts
########################################


########################
# Create customers table
########################
CREATE TABLE 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(10)  NULL ,
  cust_country char(50)  NULL ,
  cust_contact char(50)  NULL ,
  cust_email   char(255) NULL ,
  PRIMARY KEY (cust_id)
) ENGINE=InnoDB;

########################
# Create orderitems table
########################
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;


####################
# Create orders table
####################
CREATE TABLE orders
(
  order_num  int      NOT NULL AUTO_INCREMENT,
  order_date datetime NOT NULL ,
```

```
  cust_id    int      NOT NULL ,
  PRIMARY KEY (order_num)
) ENGINE=InnoDB;

#######################
# Create products table
#######################
CREATE TABLE products
(
  prod_id    char(10)      NOT NULL,
  vend_id    int           NOT NULL ,
  prod_name  char(255)     NOT NULL ,
  prod_price decimal(8,2)  NOT NULL ,
  prod_desc  text          NULL ,
  PRIMARY KEY(prod_id)
) ENGINE=InnoDB;

#######################
# Create vendors table
#######################
CREATE TABLE vendors
(
  vend_id      int      NOT NULL AUTO_INCREMENT,
  vend_name    char(50) NOT NULL ,
  vend_address char(50) NULL ,
  vend_city    char(50) NULL ,
  vend_state   char(5)  NULL ,
  vend_zip     char(10) NULL ,
  vend_country char(50) NULL ,
  PRIMARY KEY (vend_id)
) ENGINE=InnoDB;

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


####################
# Define foreign keys
```

```
94  ####################
95  ALTER TABLE orderitems ADD CONSTRAINT fk_orderitems_orders FOREIGN KEY
    (order_num) REFERENCES orders (order_num);
96  ALTER TABLE orderitems ADD CONSTRAINT fk_orderitems_products FOREIGN KEY
    (prod_id) REFERENCES products (prod_id);
97  ALTER TABLE orders ADD CONSTRAINT fk_orders_customers FOREIGN KEY
    (cust_id) REFERENCES customers (cust_id);
98  ALTER TABLE products ADD CONSTRAINT fk_products_vendors FOREIGN KEY
    (vend_id) REFERENCES vendors (vend_id);
```

## 3、插入数据命令

SQL | 复制代码

```sql
#########################################
# MySQL Crash Course
# http://www.forta.com/books/0672327120/
# Example table population scripts
#########################################


############################
# Populate customers table
############################
INSERT INTO customers(cust_id, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country, cust_contact, cust_email)
VALUES(10001, 'Coyote Inc.', '200 Maple Lane', 'Detroit', 'MI', '44444',
'USA', 'Y Lee', 'ylee@coyote.com');
INSERT INTO customers(cust_id, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country, cust_contact)
VALUES(10002, 'Mouse House', '333 Fromage Lane', 'Columbus', 'OH',
'43333', 'USA', 'Jerry Mouse');
INSERT INTO customers(cust_id, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country, cust_contact, cust_email)
VALUES(10003, 'Wascals', '1 Sunny Place', 'Muncie', 'IN', '42222',
'USA', 'Jim Jones', 'rabbit@wascally.com');
INSERT INTO customers(cust_id, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country, cust_contact, cust_email)
VALUES(10004, 'Yosemite Place', '829 Riverside Drive', 'Phoenix', 'AZ',
'88888', 'USA', 'Y Sam', 'sam@yosemite.com');
INSERT INTO customers(cust_id, cust_name, cust_address, cust_city,
cust_state, cust_zip, cust_country, cust_contact)
VALUES(10005, 'E Fudd', '4545 53rd Street', 'Chicago', 'IL', '54545',
'USA', 'E Fudd');


########################
# Populate vendors table
########################
INSERT INTO vendors(vend_id, vend_name, vend_address, vend_city,
vend_state, vend_zip, vend_country)
VALUES(1001,'Anvils R Us','123 Main Street','Southfield','MI','48075',
'USA');
INSERT INTO vendors(vend_id, vend_name, vend_address, vend_city,
vend_state, vend_zip, vend_country)
VALUES(1002,'LT Supplies','500 Park Street','Anytown','OH','44333',
'USA');
INSERT INTO vendors(vend_id, vend_name, vend_address, vend_city,
vend_state, vend_zip, vend_country)
```

```
31  VALUES(1003,'ACME','555 High Street','Los Angeles','CA','90046', 'USA');
32  INSERT INTO vendors(vend_id, vend_name, vend_address, vend_city,
    vend_state, vend_zip, vend_country)
33  VALUES(1004,'Furball Inc.','1000 5th Avenue','New York','NY','11111',
    'USA');
34  INSERT INTO vendors(vend_id, vend_name, vend_address, vend_city,
    vend_state, vend_zip, vend_country)
35  VALUES(1005,'Jet Set','42 Galaxy Road','London', NULL,'N16 6PS',
    'England');
36  INSERT INTO vendors(vend_id, vend_name, vend_address, vend_city,
    vend_state, vend_zip, vend_country)
37  VALUES(1006,'Jouets Et Ours','1 Rue Amusement','Paris', NULL,'45678',
    'France');
38
39
40  #########################
41  # Populate products table
42  #########################
43  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
44  VALUES('ANV01', 1001, '.5 ton anvil', 5.99, '.5 ton anvil, black,
    complete with handy hook');
45  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
46  VALUES('ANV02', 1001, '1 ton anvil', 9.99, '1 ton anvil, black, complete
    with handy hook and carrying case');
47  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
48  VALUES('ANV03', 1001, '2 ton anvil', 14.99, '2 ton anvil, black,
    complete with handy hook and carrying case');
49  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
50  VALUES('OL1', 1002, 'Oil can', 8.99, 'Oil can, red');
51  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
52  VALUES('FU1', 1002, 'Fuses', 3.42, '1 dozen, extra long');
53  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
54  VALUES('SLING', 1003, 'Sling', 4.49, 'Sling, one size fits all');
55  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
56  VALUES('TNT1', 1003, 'TNT (1 stick)', 2.50, 'TNT, red, single stick');
57  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
58  VALUES('TNT2', 1003, 'TNT (5 sticks)', 10, 'TNT, red, pack of 10
    sticks');
59  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
60  VALUES('FB', 1003, 'Bird seed', 10, 'Large bag (suitable for road
    runners)');
61  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
62  VALUES('FC', 1003, 'Carrots', 2.50, 'Carrots (rabbit hunting season
    only)');
63  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
64  VALUES('SAFE', 1003, 'Safe', 50, 'Safe with combination lock');
65  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
```

```
66  VALUES('DTNTR', 1003, 'Detonator', 13, 'Detonator (plunger powered),
    fuses not included');
67  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
68  VALUES('JP1000', 1005, 'JetPack 1000', 35, 'JetPack 1000, intended for
    single use');
69  INSERT INTO products(prod_id, vend_id, prod_name, prod_price, prod_desc)
70  VALUES('JP2000', 1005, 'JetPack 2000', 55, 'JetPack 2000, multi-use');
71
72
73
74  ########################
75  # Populate orders table
76  ########################
77  INSERT INTO orders(order_num, order_date, cust_id)
78  VALUES(20005, '2005-09-01', 10001);
79  INSERT INTO orders(order_num, order_date, cust_id)
80  VALUES(20006, '2005-09-12', 10003);
81  INSERT INTO orders(order_num, order_date, cust_id)
82  VALUES(20007, '2005-09-30', 10004);
83  INSERT INTO orders(order_num, order_date, cust_id)
84  VALUES(20008, '2005-10-03', 10005);
85  INSERT INTO orders(order_num, order_date, cust_id)
86  VALUES(20009, '2005-10-08', 10001);
87
88
89  ############################
90  # Populate orderitems table
91  ############################
92  INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
93  VALUES(20005, 1, 'ANV01', 10, 5.99);
94  INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
95  VALUES(20005, 2, 'ANV02', 3, 9.99);
96  INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
97  VALUES(20005, 3, 'TNT2', 5, 10);
98  INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
99  VALUES(20005, 4, 'FB', 1, 10);
100 INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
101 VALUES(20006, 1, 'JP2000', 1, 55);
102 INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
103 VALUES(20007, 1, 'TNT2', 100, 10);
104 INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
    item_price)
```

```
VALUES(20008, 1, 'FC', 50, 2.50);
INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
item_price)
VALUES(20009, 1, 'FB', 1, 10);
INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
item_price)
VALUES(20009, 2, 'OL1', 1, 8.99);
INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
item_price)
VALUES(20009, 3, 'SLING', 1, 4.49);
INSERT INTO orderitems(order_num, order_item, prod_id, quantity,
item_price)
VALUES(20009, 4, 'ANV03', 1, 14.99);

##############################
# Populate productnotes table
##############################
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(101, 'TNT2', '2005-08-17',
'Customer complaint:
Sticks not individually wrapped, too easy to mistakenly detonate all at
once.
Recommend individual wrapping.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(102, 'OL1', '2005-08-18',
'Can shipped full, refills not available.
Need to order new can if refill needed.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(103, 'SAFE', '2005-08-18',
'Safe is combination locked, combination not provided with safe.
This is rarely a problem as safes are typically blown up or dropped by
customers.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(104, 'FC', '2005-08-19',
'Quantity varies, sold by the sack load.
All guaranteed to be bright and orange, and suitable for use as rabbit
bait.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(105, 'TNT2', '2005-08-20',
'Included fuses are short and have been known to detonate too quickly
for some customers.
Longer fuses are available (item FU1) and should be recommended.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
```

```
VALUES(106, 'TNT2', '2005-08-22',
'Matches not included, recommend purchase of matches or detonator (item DTNTR).'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(107, 'SAFE', '2005-08-23',
'Please note that no returns will be accepted if safe opened using explosives.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(108, 'ANV01', '2005-08-25',
'Multiple customer returns, anvils failing to drop fast enough or falling backwards on purchaser. Recommend that customer considers using heavier anvils.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(109, 'ANV03', '2005-09-01',
'Item is extremely heavy. Designed for dropping, not recommended for use with slings, ropes, pulleys, or tightropes.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(110, 'FC', '2005-09-01',
'Customer complaint: rabbit has been able to detect trap, food apparently less effective now.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(111, 'SLING', '2005-09-02',
'Shipped unassembled, requires common tools (including oversized hammer).'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(112, 'SAFE', '2005-09-02',
'Customer complaint:
Circular hole in safe floor can apparently be easily cut with handsaw.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(113, 'ANV01', '2005-09-05',
'Customer complaint:
Not heavy enough to generate flying stars around head of victim. If being purchased for dropping, recommend ANV02 or ANV03 instead.'
);
INSERT INTO productnotes(note_id, prod_id, note_date, note_text)
VALUES(114, 'SAFE', '2005-09-07',
'Call from individual trapped in safe plummeting to the ground, suggests an escape hatch be added.
Comment forwarded to vendor.'
);

```

## 4、全书30小节读书笔记汇总

**select、insert、delete、update 基本语句**

insert into customers(cust_name,cust_address,cust_city,cust_state,cust_zip,cust_contry) values('pep','100 main','los angels','CA','90046','usa'),('m.martian','42 galaxy','new york','NY','11213','USA')

update customers set cust_name = 'the fudds',cust_email = 'elmer@fudd.com' where cust_id = 1005;

delete from customers where cust_id = 10006;

**第1–7章**

select prod_id,prod_price,prod_name from products order by prod_price DESC；降序，升序是ASC，当然了，默认是升序排列的

select prod_price from products order by prod_price desc limit 1；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。

select prod_name, prod_price from products where prod_name='fuses';

select prod_name,prod_price from products where prod_price<10;

select prod_name,prod_price from products where prod_price<=10;

select vend_id,prod_name,prod_price from products where vend_id!=1003; 不匹配1003

> 其中！= 号也可以用 <> 来代替，效果是一样的
>
> 比如  select vend_id,prod_name,prod_price from products where vend_id <> 1003;

select prod_name,prod_price from products where prod_price between  5 and 10;

> between a and b 范围值检查，这里前后都是闭区间  即 **[5,10]**

select cust_id from customers where cust_email IS NULL；空值检查

基本操作符 where XXX and XXX 或者是or

select prod_name,prod_price from products where vend_id=1002 or vend_id=1003 and prod_price>=10；查询价格>=10且由1002或者1003制造的产品信息，但是**and优先级更高**，所以后面两列先结合了，返回的结果跟预想的不一样，应该用下面的这样

**select prod_name,prod_price from products where (vend_id=1002 or vend_id=1003) and prod_price>=10;**

select prod_name,prod_price from products where vend_id IN(1002,1003) ORDER BY prod_name; 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。

其实in 和 or基本是一样的，但是in一般比 or执行速度要快一点以及计算次序更易理解，比如

select prod_name,prod_price from products where vend_id =1002 or vend_id =1003  order by prod_name;

select prod_name,prod_price from products where vend_id NOT IN (1002,1003) ORDER BY prod_name; WHERE子句中的NOT操作符有且只有一个功能，那就是否定它之后所跟的任何条件。 NOT是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，这与多数其他DBMS允许使用NOT对各种条件取反有很大的差别。

**第8章 使用通配符进行过滤**

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。 %告诉MySQL接受jet之后的任意字符，不管它有多少字符。

在搜索串中 %代表**任意字符出现任意次数**

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，不论它之前或之后出现什么字符。有点全局搜索含有anvil文本的意思了

select prod_name from products where prod_name LIKE 's%e'; 匹配出以s开头以e结尾的所有产品

select prod_id,prod_name from products where prod_name LIKE '_ ton anvil';

另一个有用的通配符是下划线（_） 。下划线的用途与%一样，但**下划线只匹配单个字符而不是多个字符**。它是只能匹配一个字符的， % 是任意字符任意次数。

这句和下一句的返回结果就不一样，

select prod_id,prod_name from products where prod_name LIKE '% ton anvil'; 这句明显多返回一个匹配结果

不要过度使用通配符。如果其他操作符能达到相同的目的，应该使用其他操作符。

在确实需要使用通配符时，除非绝对有必要，即使在有必要的情况下也不要把它们用在搜索模式的开始处。

把通配符置于搜索模式的开始处，搜索起来是最慢的，因为这样就是全文检索了。

**第9章 正则表达式**

select prod_name from products where prod_name REGEXP '1000' ORDER BY prod_name;除关键字LIKE被REGEXP替代外，这条语句看上去非常像使用LIKE的语句（第8章）。它告诉MySQL：REGEXP后所跟的东西作为正则表达式（与文字正文1000匹配的一个正则表达式）处理

> select prod_name from products where prod_name LIKE '%1000' ORDER BY prod_name;
>
> select prod_name from products where prod_name REGEXP '1000' ORDER BY prod_name;
>
> 这两句话其实是一样的效果

select prod_name from products where prod_name REGEXP '1000|2000' ORDER BY prod_name; | 是指选择符，选择其中一个

select prod_name from products where prod_name REGEXP '[123] Ton' ORDER BY prod_name;

这里，使用了正则表达式[123] Ton。[123]定义一组字符，它的意思是匹配1或2或3，因此， 1 ton和2 ton都匹配且返回（没有3 ton）。 正如所见，[]是另一种形式的OR语句。 事实上，正则表达式‘’[123]Ton‘’ 为‘’[1|2|3]Ton‘’的缩写，也可以使用后者

其实 [0123456789] 与[0-9]是一样的效果，还有类似的[a-z]或者[A-Z]，这里的 - 有一种直到的意思在里面

select prod_name from products where prod_name REGEXP '[1-5] Ton' ORDER BY prod_name; 这里使用正则表达式[1-5] Ton。[1-5]定义了一个范围，这个表达式意思是匹配1到5，因此返回3个匹配行。由于5 ton匹配，所以返回.5 ton

select vend_name from vendors where vend_name REGEXP '\.' ORDER BY vend_name; 为了匹配特殊字符，必须用\为前导。 \-表示查找-， \.表示查找. ,类似于python中的转义字符 \ 符号

select prod_name from products where prod_name REGEXP '\([0-9] sticks?\)' ORDER BY prod_name; 正则表达式\([0-9] sticks?\)需要解说一下。 \(匹配(，[0-9]匹配任意数字（这个例子中为1和5）， sticks?匹配stick和sticks（ s后的?使s可选，因为?匹配它前面的任何字符的0次或1次出现），\)匹配)。没有?，匹配stick和sticks会非常困难。

select prod_name  from products where prod_name REGEXP '[1]' order by prod_name;

select prod_name  from products where prod_name REGEXP '[0-9\.]' order by prod_name;

注意看这里的两个区别 ^ 是文本开始的意思，^匹配串的开始。因此， [0-9\.]只在.或任意数字为串中第一个字符时才匹配它们。是需要以这些为开头的，如果没有， 则还要多检索出4个别的行（那些中间有数字的行）

**第10章 创建计算字段**

select CONCAT(vend_name,'(',vend_country,')') from vendors  ORDER BY vend_name;

Concat()拼接串，即把多个串连接起来形成一个较长的串。

Concat()需要一个或多个指定的串，各个串之间用逗号分隔。上面的SELECT语句连接以下4个元素,就是将数据库中查询到的内容一自己想要的形式展现出来，增增减减一些字符来使得查询到的结果更好看一些。

select CONCAT(RTRIM(vend_name),'(',RTRIM(vend_country),')') from vendors  ORDER BY vend_name; RTrim()函数去掉值右边的所有空格。通过使用RTrim()，各个列都进行了去除右边空格的操作整理。同样的 LTRIM就是去除左边的空格，TRIM就是去除左右两边的空格了。

select CONCAT(RTRIM(vend_name),'(',RTRIM(vend_country),')') as vend_title from vendors  ORDER BY vend_name; SQL支持列别名。 别名（ alias） 是一个字段或值的替换名。别名用AS关键字赋予。计算字段之后跟了文本AS vend_title。它指示SQL创建一个包含指定计算的名为vend_title的计算字段。从输出中可以看到，结果与以前的相同，但现在列名为vend_title，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。

select prod_id,quantity,item_price from orderitems WHERE order_num = 20005;

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，此计算结果为quantity*item_price，并且使用expanded_price 作为输出结果列名

select NOW(); 可返回当前日期与时间

**第11章 使用函数处理字符串**

select vend_name,UPPER(vend_name) as vend_name_upcase from vendors order by vend_name;

upper函数可以将小写字母 转化为大写的字母形式

select cust_name,cust_contact from customers where Soundex(cust_contact) = SOUNDEX('Y Lie');
使用soundex()函数，匹配所有发音类似于Y.Lie的用户名，返回结果为 Y Lee

select cust_id,order_num from orders where order_date = '2005-09-01'；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，更加可靠的select语句是下面这样的。

select cust_id,order_num from orders where DATE(order_date) = '2005-09-01';

如果你想检索出2005年9月下的所有订单，怎么办？简单的相等测试不行，因为它也要匹配月份中的天数。有以下几种解决办法，

- 使用DATE函数提取出日期，判断是否在9.1-9.30之间，因为between and 是双向闭合区间
  select cust_id,order_num,order_date from orders where Date(order_date) BETWEEN '2005-09-01' AND '2005-09-30';
- 还有另外一种办法（一种不需要记住每个月中有多少天或不需要操心闰年2月的办法）： Year()是一个从日期（或日期时间）中返回年份的函数。类似，Month()从日期中返回月份。
  select cust_id,order_num from orders where YEAR(order_date) = 2005 and MONTH(order_date) = 9;

**第12章 汇总函数**

select AVG(prod_price) as avg_price from products；使用AVG函数返回所有平均值

select AVG(prod_price) AS AVG_PRICE from products where vend_id = 1003; 返回特定列的平均值

select COUNT(*) as num_cust from customers; 返回客户的总数，不管有没有值或者是为NULL

select COUNT(cust_email) AS num_cust from customers；可以返回特定列的汇总数量

select MAX(prod_price) as max_price from products; 返回最大值

select MIN(prod_price) as min_price from products; 返回最小值

select SUM(prod_price) as sum_price from products;  返回总数

select SUM(item_price*quantity) AS total_price from orderitems where order_num = 20005; SUM函数并且取个别名

select prod_price  from products where vend_id = 1003;

select DISTINCT prod_price  from products where vend_id = 1003;

select SUM(DISTINCT prod_price) as avg_distinct_price  from products where vend_id = 1003;
 DISTINCT 可以用来去重，对于那些重复的只统计一次。

select COUNT(*) AS num_items, MIN(prod_price) AS min_price, MAX(prod_price) as max_price, AVG(prod_price) as avg_price from products; 实际上SELECT语句可根据需要包含多个聚集函数.这里用单条SELECT语句执行了4个聚集计算，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）

**第13章 HAVING语句**

select vend_id,COUNT(*) as num_prods from products GROUP BY vend_id;  上面的SELECT语句指定了两个列， vend_id包含产品供应商的ID，num_prods为计算字段（用COUNT(*)函数建立）。 GROUP BY子句指示MySQL按vend_id排序并分组数据。这导致对每个vend_id而不是整个表计算num_prods一次。从输出中可以看到，供应商1001有3个产品，供应商1002有2个产品，供应商1003有7个产品，而供应商1005有2个产品

select vend_id,COUNT(*) AS num_prods from products GROUP BY vend_id WITH ROLLUP; 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，如下所示

select vend_id,COUNT(*) AS num_prods from products GROUP BY vend_id;

select cust_id,COUNT(*) AS orders from orders GROUP BY cust_id HAVING COUNT(*) >= 2;

> 即使这里已经有orderers这个别名了，在having子句中依然要使用count函数而不是使用orders别名

这条SELECT语句的前3行类似于上面的语句。最后一行增加了HAVING子句，它过滤COUNT(*) >=2（两个以上的订单）的那些分组。 正如所见，这里WHERE子句不起作用，因为过滤是基于分组聚集值而不是特定行值的。**HAVING和WHERE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： 这条语句中，第一行是使用了聚集函数的基本SELECT，它与前面的例子很相像。 WHERE子句过滤所有prod_price至少为10的行。然后按vend_id分组数据， HAVING子句过滤计数为2或2以上的分组。如果没有WHERE子句，将会多检索出两行（供应商1002，销售的所有产品价格都在10以下；供应商1001，销售3个产品，但只有一个产品的价格大于等于10）： HAVING在数据分组后再进行过滤

select order_num,SUM(quantity*item_price) as total_price from orderitems GROUP BY order_num HAVING SUM(quantity*item_price)>=50 ORDER BY total_price;

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：为按总计订单价格排序输出，需要添加ORDER BY子句,在这个例子中， GROUP BY子句用来按订单号（order_num列）分组数据，以便SUM(*)函数能够返回总计订单价格。 HAVING子句过滤数据，使得只返回总计订单价格大于等于50的订单。最后， 用ORDERBY子句排序输出.

**第14章 使用子查询**

select order_num from orderitems where prod_id='TNT2';

select  cust_id from orders where order_num IN (20005,20007)

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 (10001,10004);

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')); 一般来说，子查询最好不要嵌套太多层

**第15章 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;

其实就是数据库的连接操作

select vend_name,prod_name,prod_price from vendors,products ORDER BY vend_name,prod_name;

需要指定where条件，要不然就会返回两个表的笛卡儿积了

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; 联结的表越多，性能下降的越厉害，如果不是特别需要，尽可能少联结表

## 第16章 创建高级联结

select concat(RTrim(vend_name),'(',RTrim(vend_country),')') AS vend_title from vendors ORDER BY vend_name;

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';

select prod_id,prod_name from products where vend_id = (select vend_id from products where prod_id = 'DTNTR') 这是子查询的方法，下面的是自联结，假如你发现某物品（其ID为DTNTR）存在问题，因此想知道生产该物品的供应商生产的其他物品是否也存在这些问题。此查询要求首先找到生产ID为DTNTR的物品的供应商，然后找出这个供应商生产的其他物品。

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'; 为同样的表起了两个别名，这样是允许的。

## 第17章 组合查询

select vend_id,prod_id,prod_price from products where prod_price <= 5;

select vend_id,prod_id,prod_price from products where vend_id IN (1001,1002);第一条SELECT检索价格不高于5的所有物品。第二条SELECT使用IN找出供应商1001和1002生产的所有物品。

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) ; 这条语句由前面的两条SELECT语句组成，语句中用UNION关键字分隔。 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。

**用union的前提是select所要查询的项必须一样才行**，比如这里都是vend_id，prod_id和prod_price，这是必须的。这和下面的查询其实是一样的.

> select vend_id,prod_id,prod_price from products where prod_price <= 5 OR vend_id IN(1001,1002);

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) ;

使用union all来返回所有的结果，即使两个select中有相同的结果也不做去重处理。

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）ORDER BY vend_id,prod_price;SELECT语句的输出用ORDER BY子句排序。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。

**第18章 理解全文本搜索**

并非所有引擎都支持全文本搜索 正如第21章所述， MySQL支持几种基本的数据库引擎。并非所有的引擎都支持本书所描述的全文本搜索。

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。这就是为什么虽然本书中创建 的多数样例表使用InnoDB，而有一个样例表（ productnotes表）却使用MyISAM的原因。如果你的应用中需要全文本搜索功能，应该记住这一点。

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

下面的CREATE 语句演示了FULLTEXT子句的使用

```sql
  CREATE TABLE productnotes

  (

note_id    int           NOT NULL AUTO_INCREMENT,

prod_id    char(10)      NOT NULL,

note_date datetime       NOT NULL,

note_text  text          NULL ,

PRIMARY KEY(note_id),

FULLTEXT(note_text)

 ) ENGINE=MyISAM;
```

这些列中有一个名为note_text的列，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。这里的 FULLTEXT索引单个列，如果需要也可以指定多个列。

在索引之后使用match函数指定被搜索的列，against指定要使用的搜索表达式

select note_id,note_text from productnotes where match(note_text) against('rabbit');

此SELECT语句检索单个列note_text。由于WHERE子句，一个全文本搜索被执行。 Match(note_text)指示MySQL针对指定的列进行搜索， Against('rabbit')指定词rabbit作为搜索文本。由于有两行包含词rabbit，这两个行被返回

使用完整的Match()说明传递给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）

其实刚才的搜索也可以用LIKE子句来完成：

select note_id,note_text from productnotes WHERE note_text LIKE '%rabbit%'

前者（使用全文本搜索）返回以文本匹配的良好程度排序的数据。两个行都包含词rabbit，但包含词rabbit作为第3个词的行的等级比作为第20个词的行高。这很重要。全文本搜索的一个重要部分就是对结果排序。具有较高等级的行先返回（因为这些行很可能是你真正想要的行），这点事like实现不了的地方。

select note_id,note_text,MATCH(note_text) Against('rabbit') AS rank from productnotes;

这里， 在SELECT而不是WHERE子句中使用Match()和Against()。这使所有行都被返回（因为没有WHERE子句）。 Match()和Against()用来建立一个计算列（别名为rank），**此列包含全文本搜索计算出的等级值**。等级由MySQL根据行中词的数目、唯一词的数目、整个索引中词的总数以及包含该词的行的数目计算出来。正如所见，不包含词rabbit的行等级为0（因此不被前一例子中的WHERE子句选择）。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

排序多个搜索项 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,全文本搜索提供了简单的LIKE搜索所不能提供的功能，而且数据是索引的，全文本搜索速度较快。

使用查询拓展功能，利用查询拓展能找出可能相关的结果，即使他们并不精确包含所要查找的词

select note_id,note_text from productnotes where MATCH(note_text) against ('anvils')；这里并没有使用查询拓展，只返回一行

SELECT note_id,note_text FROM productnotes where match(note_text) against ('anvils' WITH QUERY EXPANSION) ;这次使用查询拓展，这次返回了7行。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）。正如所见，查询扩展极大地增加了返回的行数，但这样做也增加了你实际上并不想要的行的数目。

布尔方式搜索即使没有FULLTEXT索引也可以使用，但比较慢，会随着数据量的增加而降低。布尔方式支持要匹配的词，要排斥的词以及排列提示和表达式分组等.排列而不排序 在布尔方式中，不按等级值降序排序返回的行

SELECT note_id,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（有两行）。其中使用了关键字IN BOOLEAN MODE，但实际上没有指定布尔操作符，因此，其结果与没有指定布尔方式的结果相同。

SELECT  note_id,note_text FROM productnotes where MATCH(note_text) against ('heavy –rope*' IN BOOLEAN MODE); 匹配包含heavy但不包含任意以rope开始的词的行， 可使用这个查询，这次只返回一行。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，这就是为什么上一个例子中的第一行被排除的原因。 全文本布尔操作符还有很多不止 – 和 * 号.

select note_id,note_text from productnotes where match(note_text) against ('+rabbit + bait ' IN BOOLEAN MODE)； 匹配包含词rabbit和bait 的行

select note_id,note_text from productnotes where match(note_text) against ('rabbit bait ' IN BOOLEAN MODE); 没有指定操作符，这个搜索匹配包含rabbit和bait中的至少一个词的行

select note_id,note_text from productnotes where match(note_text) against ('"rabbit bait"' IN BOOLEAN MODE); 但是这个匹配短语 “rabbit bait”而不是匹配两个词语rabbit和bait

select note_id,note_text from productnotes where match(note_text) against ('>rabbit <carrot' IN BOOLEAN MODE);匹配rabbit和carrot，增加前者的等级，降低后者的等级

select  note_id,note_text from productnotes where match(note_text) against ('+safe + (<combination)' IN BOOLEAN MODE); 匹配词为safe和combination，降低后者的等级

**第19章 插入数据**

insert INTO customers VALUES(NULL,'pep E.Lapew','100 Main','los'，XXXX) 这样的简单插入语句，必须要注意的是values的顺序必须跟数据库中列的顺序相同才行，第一个值就要放在第一个值中去

insert into customers(cust_name,cust_contact,cust_email,cust_address,cust_city,cust_state,cust_zip,cust_country) values('pep E.lapew',NULL,NULL,'100 main street','los angeles','CA','90046','USA');

这里insert语句填充了所有的列名，这样即使列顺序和数据库中的不一样，这样指定了列名也能正确的插入进去。一般不要使用没有明确给出列的列表的INSERT语句,或者再三确认要插入的值得顺序，以保证不会出现顺序错误的情况。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。

**一次插入多条数据**：insert into customers(cust_name,cust_address,cust_city,cust_state,cust_zip,cust_contry) values('pep','100 main','los angels','CA','90046','usa'),('m.martian','42 galaxy','new york','NY','11213','USA') 其中单条insert语句有多组值，每对值用一对圆括号括起来，中间用逗号分开，以此来插入多条语句。毫无疑问只使用一次insert就插入多条语句，肯定比分别插入这些数据要快。

**插入检索出的数据,即insert select 组合起来使用**，真的是6 ，第一次看到这种说法

insert into customers(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 custnew; 这个例子使用INSERT SELECT从custnew中将所有数据导入customers。 SELECT语句从custnew检索出要插入的值，而不是列出它们。 SELECT中列出的每个列对应于customers表名后所跟的列表中的每个列。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）。如果这个表确实含有数据，则所有数据将被插入到customers。这个例子导入了cust_id（假设你能够确保cust_id的值不重复）。你也可以简单地省略这列（从INSERT和SELECT中），这样MySQL就会生成新值。

INSERT SELECT中的列名 为简单起见，这个例子在INSERT和SELECT语句中使用了相同的列名。但是，不一定要求列名匹配。事实上， 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。当然了insert select中的select语句是可以包含where语句来过滤插入的数据的。

**第20章 更新和删除数据**

更新数据，其中update语句是可以更新整个表的数据或者特定行的数据的，使用时千万注意。其中update语句比较简单，比如

update customers set cust_email = 'elmer@fudd.com' where cust_id = 1005;UPDATE语句总是以要更新的表的名字开始。在此例子中，要更新的表的名字为customers。 SET命令用来将新值赋给被更新的列。如这里所示， SET子句设置cust_email列为指定的值：UPDATE语句以WHERE子句结束，它告诉MySQL更新哪一行。没有WHERE子句， MySQL将会用这个电子邮件地址更新customers表中所有行，这不是我们所希望的。

而更新多个列的语句稍有不同：update customers set cust_name = 'the fudds',cust_email = 'elmer@fudd.com' where cust_id = 1005;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）。在此例子中，更新客户10005的cust_name和cust_email列。

uodate customers set cust_email = NULL where cust_id = 1005;为了删除某个列的值，可设置它为NULL（假如表定义允许NULL值）。其中NULL是用来去除cust_email列的值

删除数据

delete语句删除特定的行或者所有行，也是需要注意稍不注意就会删错了

delete from customers where cust_id = 10006;这条语句很容易理解。 DELETE FROM要求指定从中删除数据的表名。WHERE子句过滤要删除的行。在这个例子中，只删除客户10006。如果省略WHERE子句，它将删除表中每个客户。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。删除表的命令是drop ，比如drop customers。

可以使用truncate table 来删除表中所有行。更快的删除 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）。使用truncate可以使得自增id再次从0开始增加。

前一节中使用的UPDATE和DELETE语句全都具有WHERE子句，这样做的理由很充分。如果省略了WHERE子句，则UPDATE或DELETE将被应用到表中所有的行。换句话说，如果执行UPDATE而不带WHERE子句，则表中每个行都将用新值更新。类似地，如果执行DELETE语句而不带WHERE子句，表的所有数据都将被删除。

下面是许多SQL程序员使用UPDATE或DELETE时所遵循的习惯。

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（如果忘记这个内容，请参阅第15章），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（关于这个内容，请参阅第15章），这样MySQL将不允许删除具有与其他表相关联的数据的行

小心使用 MySQL，因为MySQL没有撤销（ undo）按钮。应该非常小心地使用UPDATE和DELETE，否则你会发现自己更新或删除了错误的数据

## 第21章 创建和操纵表

使用客户端建表或者MySQL语句建表，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。

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。

SQL 复制代码

```sql
create TABLE orders(

order_num int NOT NULL,

order_date datetime NOT NULL,

cust_id int NOT NULL,

PRIMARY KEY(order_num)

) ENGINE = InnoDB;
```

这条语句创建本书中所用的orders表。 orders包含3个列，分别是订单号、订单日期和客户ID。所有3个列都需要，因此每个列的定义都含有关键字NOT NULL。这将会阻止插入没有值的列。如果试图插入没有值的列，将返回错误，且插入失败。NULL为默认设置，如果不指定NOT NULL，则认为指定的是NULL.

正如所述，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,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.

```sql
create TABLE orders(

order_num int NOT NULL,

order_item int  NOT NULL,

prod_id char(10) NOT NULL,

item_price decimal(8,2) NOT NULL,

PRIMARY KEY(order_num,order_item)

)ENGINE = InnoDB;
```

orderitems表包含orders表中每个订单的细节。每个订单有多项物品，但每个订单任何时候都只有1个第一项物品， 1个第二项物品，如此等等。因此，订单号（order_num列）和订单物品（order_item列）的组合是唯一的，从而适合作为主键，其定义为：PRIMARY KEY(order_num,order_item).第1章介绍过，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.

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）。

如何在使用AUTO_INCREMENT时获得自增的列值呢？可使用**last_insert_id()函数**获得这个值，如下所示：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。如下：

select last_insert_id(order_num) from orders

> 经过测试得知：SELECT LAST_INSERT_ID(id) from guomei_product ORDER BY id DESC limit 1;
>
> last_insert_id函数默认返回的是递增顺序，也就是从1-N，如果想要返回最后一个 应该要使用order by XXX DESC 并且加上 limit 1来进行约束。

C++ 复制代码

```
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;
```

这条语句创建包含组成订单的各物品的orderitems表（订单本身存储在orders表中）。 quantity列包含订单中每项物品的数量。在此例子中，给该列的描述添加文本DEFAULT 1指示MySQL，在未给出数量的情况下使用数量1

使用默认值而不是NULL值，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。

以下是几个需要知道的引擎：

InnoDB是一个可靠的事务处理引擎（参见第26章），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（参见第18章），但不支持事务处理

引擎类型可以混用。除productnotes表使用MyISAM外，本书中的样例表都使用InnoDB。 原因是作者希望支持事务处理（因此，使用InnoDB），但也需要在productnotes中支持全文本搜索（因此，使用MyISAM）。

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：

1. 用新的列布局创建一个新表；
2. 使用INSERT SELECT语句（关于这条语句的详细介绍，请参阅第19章）从旧表复制数据到新表。如果有必要，可使用转换函数和
3. 计算字段；
4. 检验包含所需数据的新表；
5. 重命名旧表（如果确定，可以删除它）；
6. 用旧表原来的名字重命名新表；
7. 根据需要，重新创建触发器、存储过程、索引和外键。

RENAME TABLE customers TO customers;重命名表

RENAME TABLE back_customers TO customers，back_venders TO vendors,back_products TO products;

重命名多个表，彼此之间用逗号隔开

CREATE TABLE用来创建新表， ALTER TABLE用来更改表列（或其他诸如约束或索引等对象），而DROP TABLE用来完整地删除一个表。这些语句必须小心使用，并且应在做了备份后使用。

## 第22章 使用视图

视图用CREATE VIEW AS XXX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。

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。如果执行SELECT * FROM productcustomers，将列出订购了任意产品的客户

select cust_name,cust_contact from productcustomers where prod_id = 'TNT2'; 检索订购了产品TNT2的客户

这条语句通过WHERE子句从视图中检索特定数据。在MySQL处理此查询时，它将指定的WHERE子句添加到视图查询中的已有WHERE子句中，以便正确过滤数据。可以看出，视图极大地简化了复杂SQL语句的使用。利用视图，可一次性编写基础的SQL，然后根据需要多次使用

比如需要某些格式的结果，并且也会多次使用这类型的，可以创建一个视图，更方便的使用自己想要的例子

create VIEW vendorlocations AS SELECT Concat(RTRIM(vend_name),'(',RTRIM(vend_country),')') AS vend_title from vendors ORDER BY vend_name;

select * from vendorlocations;这条语句使用与以前的SELECT语句相同的查询创建视图。为了检索出以创建所有邮件标签的数据，可如下进行

以及通过视图过滤出不想要的数据，然后更好的进行查询也是可以的

应该将视图用于检索（SELECT语句）而不用于更新（INSERT、UPDATE和DELETE）。视图为虚拟的表。它们包含的不是数据而是根据需要检索数据的查询

**第23章 使用存储过程,这章用的是不是很多**

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。使用存储过程比使用单独的SQL语句要快，简单来说，就是三个好处：简单安全和高性能。存储过程的执行远比其定义更经常遇到，我们将从执行存储过程开始介绍。然后再介绍创建和使用存储过程。

创建存储过程，一个返回产品平均价格的存储过程。

SQL 复制代码

```sql
CREATE PROCEDURE productpricing()

BEGIN

select AVG(prod_price) AS priceaverage from products;

END;
```

此存储过程名为productpricing，用CREATE PROCEDURE productpricing()语句定义。如果存储过程接受参数，它们将在()中列举出来。此存储过程没有参数，但后跟的()仍然需要。 BEGIN和END语句用来限定存储过程体，过程体本身仅是一个简单的SELECT语句

如何使用这个存储过程呢

CALL productpricing();CALL productpricing();执行刚创建的存储过程并显示返回的结果。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

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。

存储过程在创建之后，被保存在服务器上以供使用，直至被删除。比如删除刚才创建的存储过程，可以使用如下语句：

DROP PROCEDURE productpricing;

这条语句删除刚创建的存储过程。请注意没有使用后面的()，只给出存储过程名.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。

**第24章 使用游标**

游标不是一条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。游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。用的不多

**第25章 触发器**

MySQL语句在需要时被执行，存储过程也是如此。但是，如果你想要某条语句（或某些语句）在事件发生时自动执行，怎么办呢？例

如： 每当增加一个顾客到某个数据库表时，都检查其电话号码格式是否正确，州的缩写是否为大写； 每当订购一个产品时，都从库存数量中减去订购的数量； 无论何时删除一行，都在某个存档表中保留一个副本。所有这些例子的共同之处是它们都需要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

注意，只有DELETE,INSERT,UPDATE.才能触发触发器，其他MySQL语句不支持触发器。

在创建触发器时，需要给出4条信息： 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）。

触发器用CREATE TRIGGER语句创建。，比如：

CREATE TRIGGER newproduct AFTER INSERT ON products FOR EACH ROW SELECT 'Product added';

CREATE TRIGGER用来创建名为newproduct的新触发器。触发器可在一个操作发生之前或之后执行，这里给出了AFTER INSERT，所以此触发器将在INSERT语句成功执行后执行。这个触发器还指定FOR EACH ROW，因此代码对每个插入行执行。在这个例子中，文本Product added将对每个插入的行显示一次.使用INSERT语句添加一行或多行到products中，你将看到对每个成功的插入，显示Product added消息。

只有表才支持触发器,视图是不支持的,而且每个表每个事件每次只允许一个触发器,因此需要注意的是每个表最多支持3*2 = 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.

触发器失败 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）。

删除触发器使用DROP TRIGGER XXX语句,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

三种触发器:INSERT,DELETE,UPDATE,每种又分before和after,一共六种

**INSERT 触发器**

INSERT触发器在INSERT语句执行之前或之后执行。需要知道以下几点： 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。一个实际有用的例子:

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

此代码创建一个名为neworder的触发器，它按照AFTER INSERTON orders执行。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BEFOREINSERT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。

因为从MySQL5以后不支持触发器返回结果集，需要用一个变量接收结果集输出,所以在插入语句后面加上这条SELECT @p,即可输出之前定义的触发器@p变量的结果集。如下:

insert INTO orders(order_date, cust_id) VALUES(NOW(), 10001);

select @p;

BEFORE或AFTER? 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。需要知道以下两

点： 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。

下面的例子演示使用OLD保存将要被删除的行到一个存储表中.

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;

DELETE from orders where YEAR(order_date) = 2019;

在任意订单被删除前将执行此触发器。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）

**UPDATE 触发器**

UPDATE触发器在UPDATE语句执行之前或之后执行。需要知道以下几点： 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。

下面的例子保证州名缩写总是大写（不管UPDATE语句中给出的是大写还是小写）

CREATE TRIGGER updatevendorBEFORE UPDATE on vendors FOR EACH ROW SET NEW.vend_state = UPPER(NEW.vend_state);

任何数据净化都需要在UPDATE前进行，就像这个例子中一样。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。

## 第26章 管理事务处理

并非所有引擎都支持事务处理 正如第21章所述， MySQL支持几种基本的数据库引擎。正如本章所述，并非所有引擎都支持明确的事务处理管理。 MyISAM和InnoDB是两种最常使用的引擎。前者不支持明确的事务处理管理，而后者支持。这就是为什么本书中使用的样例表被创建来使用InnoDB而不是更经常使用的MyISAM的原因。如果你的应用中需要事务处理功能，则一定要使用正确的引擎类型。

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。事务处理是一种机制，用来管理必须成批执行的MySQL操作，以保证数据库不包含不完整的操作结果。利用事务处理，可以保证一组操作不会中途停止，它们或者作为整体执行，或者完全不执行（除非明确指示）。如果没有错误发生，整组语句提交给（写到）数据库表。如果发生错误，则进行回退（撤销）以恢复数据库到某个已知且安全的状态。

下面是关于事务处理需要知道的几个术语： 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（ placeholder），你可以对它发布回退（与回退整个事务处理不同）。

MySQL使用下面的语句来标识事务的开始：START TRANSACTION

MySQL的ROLLBACK命令用来回退（撤销） MySQL语句，

```sql
select * from archive_orders;

START TRANSACTION;

DELETE FROM archive_orders;

select * from archive_orders;

ROLLBACK;

select * from archive_orders;
```

这个例子从显示archive_orders表（此表在第24章中填充）的内容开始。首先执行一条SELECT以显示该表不为空。然后开始一个事务处理，用一条DELETE语句删除archive_orders中的所有行。另一条SELECT语句验证archive_orders确实为空。这时用一条ROLLBACK语句回退START TRANSACTION之后的所有语句，最后一条SELECT语句显示该表不为空。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）。

哪些语句可以回退?

事务处理用来管理INSERT、 UPDATE和DELETE语句。你不能回退SELECT语句。（这样做也没有什么意义。）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。

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

```sql
START TRANSACTION;

DELETE FROM orderitems WHERE ORDER_NUM = 20010;

DELETE FROM orders WHERE ORDER_NUM = 20010;

COMMIT;
```

在这个例子中，从系统中完全删除订单20010。因为涉及更新两个数据库表orders和orderItems，所以使用事务处理块来保证订单不被部分删除。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）.

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，可以使用SAVEPOINT 语句

比如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，可如下进行：ROLLBACK TO delete1;

保留点越多越好 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。释放保留点 保留点在事务处理完成（执行一条ROLLBACK或COMMIT）后自动释放。自MySQL 5以来，也可以用RELEASESAVEPOINT明确地释放保留点。

MySQL用户账号和信息存储在名为mysql的MySQL数据库中,具体用户信息在user表中

ANALYZE TABLE，用来检查表键是否正确。比如：ANALYZE TABLE orders;

CHECK TABLE orders,orderitems; 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。如下所示， CHECKTABLE发现和修复问题：

## 第30章 总结，改善性能

回顾一下前面各章的重点，提供进行性能优化探讨和分析的一个出发点。

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SHOWSTATUS;。）

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

---

1. 0–9\. ↩