

1 SELECT

1.1 检索不同行

DISTINCT

demo

```
1 | SELECT DISTINCT x FROM table
```

注意

不可部分使用distinct，若使用则会作用于所有的select指出来的列。

1.2 限制结果

LIMIT num。限制结果条数。

LIMIT start,num。从start行开始查询num条。从0开始编号。

mysql5使用替代：LIMIT num OFFSET start。

IFNULL(a,b)用于判断若a表达式值为NULL，则返回b表达式的值。

```
1 | 查询第二大的工资，若不存在，则返回NULL。  
2 | SELECT IFNULL(  
3 |   (SELECT DISTINCT salary  
4 |     FROM Employee  
5 |     ORDER BY salary DESC  
6 |     LIMIT 1 OFFSET 1),NULL  
7 | ) as SecondHighestSalary;
```

demo

```
1 | SELECT * FROM table LIMIT 5
```

1.3 排序数据

ORDER BY col1,col2,col3.....。依次按照col1, col2进行排序。默认升序
DESC降序。

1.4 WHERE

操作符

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

BETWEEN a AND b 在[a,b]之间。

注意

若比较类型为字符串 则需要加: '实际值'。

检测空值

IS NULL

AND OR

AND 优先级高于OR 但可以通过加()改变复杂语句运行顺序。

IN (a,b)

也可使用嵌套select语句。

NOT

1.5 通配符过滤

LIKE (匹配整个串)

1. %通配符表示任何字符出现任意次数: LIKE 'jet%'则可查找以jet开头的行。通配符可以多处使用'%abc%', %可以匹配[0,+oo]个字符。
2. _通配符匹配单个字符，必须是一个字符。

1.6 正则表达式 (查找子串)

REGEXP 正则表达式不区分大小写，除非加BINARY关键字，

1 | 例: WHERE prod_name REGEXP BINARY 'JetPack .000'

1. .匹配一个任意字符
2. | 相当于or
3. 匹配几个字符之一: [123]匹配1或2或3
4. [^1] 表示除了1
5. 匹配范围 [0-9]

6. 特殊字符的匹配，需要转义，加\\。例如查找'!'，则需要查找'\\!'

9.2.6 匹配字符类

存在找出你自己经常使用的数字、所有字母字符或所有数字字母字符等的匹配。为更方便工作，可以使用预定义的字符集，称为字符类（character class）。表9-2列出字符类以及它们的含义。

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]）

9.2.7 匹配多个实例

目前为止使用的所有正则表达式都试图匹配单次出现。如果存在一个匹配，该行被检索出来，如果不存在，检索不出任何行。但有时需要对匹配的数目进行更强的控制。例如，你可能需要寻找所有的数，不管数中包含多少数字，或者你可能想寻找一个单词并且还能够适应一个尾随的s（如果存在），等等。

这可以用表9-3列出的正则表达式重复元字符来完成。

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）

9.2.8 定位符

目前为止的所有例子都是匹配一个串中任意位置的文本。为了匹配特定位置的文本，需要使用表9-4列出的定位符。

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

1.7 计算字段

即在查询时将多个列合并为一列得到新的数据段。

拼接字段

可使用Concat(a,b,c,d) 完成拼接，可看做是字符串拼接。

```
1 | 例: SELECT Concat(name, '(', age, ')')  
2 |   FROM user;
```

删除数据右侧多余空格：RTrim(cal) 消除cal列的。 LTrim()左侧， Trim()两侧。

执行算术运算

支持加减乘除。

1.8 函数

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
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

表11-1中的SOUNDEX需要做进一步的解释。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。SOUNDEX考虑了类似的发音字符和音节，使得能对串进行发音比较而不是字母比较。虽然SOUNDEX不是SQL概念，但MySQL（就像多数DBMS一样）都提供对SOUNDEX的支持。

输入

```
SELECT cust_name, cust_contact
FROM customers
WHERE Soundex(cust_contact) = Soundex('Y Lie');
```

输出

cust_name	cust_contact
Coyote Inc.	Y Lee

分析

在这个例子中，WHERE子句使用Soundex()函数来转换cust_contact列值和搜索串为它们的SOUNDEX值。因为Y.Lee和Y.Lie发音相似，所以它们的SOUNDEX值匹配，因此WHERE子句正确地过滤出了所需的数据。

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

首先需要注意的是MySQL使用的日期格式。无论你什么时候指定一个日期，不管是插入或更新表值还是用WHERE子句进行过滤，日期必须为格式yyyy-mm-dd。

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

1.9 汇总数据

聚集函数

对数据进行汇总，而不需要将参与的所有数据都实际检索出来，比如要获取表中的行数，而不需要将整张表展示出来。

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

注意：

1. AVG, MAX, MIN, SUM函数忽略列值为NULL的行
 - 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。
2. 值。
 - 使用COUNT(column)对特定列中具有值的行进行计数，忽略NULL值。
3. MAX, MIN函数均允许返回任意列中的最小值，不管是数值类型的，在对文本数据时，若文本数据按相应列排序，则最小值是最前面的行，最大值是最后面的。
4. 所有函数均可结合WHERE子句使用。

聚集不同值

以上5个聚集函数都可以如下使用：

- 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。



ALL为默认 ALL参数不需要指定，因为它是默认行为。如果不指定DISTINCT，则假定为ALL。

下面的例子使用AVG()函数返回特定供应商提供的产品的平均价格。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：

输入

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

组合聚集函数

多个聚集函数同时使用

```
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;
```

1.10 分组数据

创建分组

GROUP BY

在具体使用GROUP BY子句前，需要知道一些重要的规定。

- 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 WITH ROLLUP;
```

过滤分组

HAVING过滤分组，WHERE过滤行

 **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。

那么，有没有在一条语句中同时使用WHERE和HAVING子句的需要呢？事实上，确实有。假如想进一步过滤上面的语句，使它返回过去12个月内具有两个以上订单的顾客。为达到这一点，可增加一条WHERE子句，过滤出过去12个月内下过的订单。然后再增加HAVING子句过滤出具两个以上订单的分组。

为更好地理解，请看下面的例子，它列出具有2个（含）以上、价格为10（含）以上的产品的供应商：

输入

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

排序：order by

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
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

1.11 子查询

in() 当然也可用于测试等于 (=) , 不等于 (<>) 等。

输入

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

输出

cust_id
10001
10004

作为计算字段使用子查询

输入

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

输出

cust_name	cust_state	orders
Coyote Inc.	MI	2
E Fudd	IL	1
Mouse House	OH	0
Wascals	IN	1
Yosemite Place	AZ	1

1.12 联结表

外键 表中某一列是另一个表的主键。

维护引用完整性：对于外键必须插入该外键所在表中已存在的主键。

等值联结（内部联结）

where a.id=b.id

笛卡尔积 由没有联结条件的表关系返回的结果为笛卡尔积 检索行数为两表行数之积。

a INNER JOIN b ON a.id=b.id;

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

多表联结：AND 连接即可。

```
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;
```

注意：联结表越多，性能下降越厉害。

1.13 高级联结

自联结

在单条select语句中不止一次引用相同的表

子查询：

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。下面是解决此问题的一种方法：

输入

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

自联结：

对一张表起两次别名，消除二义性，让mysql可以区分出所指表。

```
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';
```

自然联结

应用场景：某个列不止出现在一个表中，自然联结可以消除重复列。

使用通配符，对其他表的列使用明确子集完成。

```
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';
```

外部联结

将一个表中的行与另一个表关联，但是同时需要将没有关联到的行也显示出来。

- 对每个客户下了多少订单进行计数，包括那些至今尚未下订单的客户；
- 列出所有产品以及订购数量，包括没有人订购的产品；
- 计算平均销售规模，包括那些至今尚未下订单的客户。

在上述例子中，联结包含了那些在相关表中没有关联行的行。这种类型的联结称为外部联结。

OUTER JOIN 结合左右联结使用。

外部联结语法类似。为了检索所有客户，包括那些没有订单的客户，可如下进行：

输入

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

输出

cust_id	order_num
10001	20005
10001	20009
10002	NULL
10003	20006
10004	20007

left则会显示所有左边的行，right则是右边表。

带聚集函数的联结

使用自然联结，外部联结均可。

使用联结的条件

- 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。

1.14 组合查询

UNION将多个SELECT语句组合成一个结果集。

在多个SELECT语句中间使用UNION连接即可，例：

分析

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。为了组合这两条语句，按如下进行：

输入

```
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);
```

注意点

1. UNION须由两条及以上SELECT语句组成
2. 每个查询必须包含相同列，表达式或聚集函数（顺序可不一致，但查询内容须一致）
3. 列数据类型必须兼容
4. **UNION自带去重**，若不带去重则将关键字改为UNION ALL即可。
5. 对数据进行排序，ORDER BY需要出现在最后一条SELECT中。

1.15 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。

5.6.24版本后InnoDB也支持了。

使用LIKE或正则表达式的限制

1. 两者都是在整个表中通过匹配所有行进行检索，故效率随表规模增大而下降
2. 明确控制，很难明确控制匹配什么与不匹配什么
3. 一个特殊词的搜索将会返回包含该词的所有行，而不区分包含单个匹配的行和包含多个匹配的行
(按照可能是更好的匹配来排列它们)。类似，一个特殊词的搜索将不会找出不包含该词但包含其他相关词的行。

添加索引

在建表时，使用FULLTEXT (a,b,c) 子句，例：

输入

```
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;
```

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）。

全文本 搜索

使用Match()和Against()函数，Match函数指定被搜索列，Against指定使用的搜索表达式。

输入

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

输出

note_text
Customer complaint: rabbit has been able to detect trap, food apparently less effective now. Quantity varies, sold by the sack load. All guaranteed to be bright and orange, and suitable for use as rabbit bait.

使用完整的**Match函数**，传递值必须与FULLTEXT定义相同，若指定了多个列，则需全部列出，且次序正确。

全文本搜索默认不区分大小写，使用BINARY区分。

全文本搜索对结果排序，具有较高等级的行先返回。

扩展查询

查询扩展用来设法放宽所返回的全文本搜索结果的范围。考虑下面的情况。你想找出所有提到anvils的注释。只有一个注释包含词anvils，但你还想找出可能与你的搜索有关的所有其他行，即使它们不包含词anvils。

这也是查询扩展的一项任务。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。

输入

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

布尔文本搜索

1. 要匹配的词
2. 要排斥的词
3. 排列提示
4. 表达式分组
5. 其余
6. 即使没有FULLTEXT索引也可以使用，但很慢

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）

输入

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

分析

这个搜索匹配包含词rabbit和bait的行。

若不限定操作符，则包含两单词任一的即可被搜索到。

总结

1. 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）。
2. 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）。
3. 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。
4. 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）。
5. 忽略词中的单引号。例如，don't索引为dont。
6. 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。

2 INSERT

降低语句优先级：

INSERT LOW_PRIORITY INTO

顺便说一下，这也适用于下一章介绍的UPDATE和DELETE语句。

2.1 插入多行数据

```
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'
);
```

单条插入语句插入多条数据的效率比多条INSERT语句高。

2.2 将检索数据插入表中

应用场景：数据迁移。例：将custnew表中数据插入customers中（注意自增主键使用新表自身，SELECT查出来的信息按你列出来的列顺序填到新表的对应列上）。

```
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;
```

select语句也可以包含where等过滤条件。

3 UPDATA DELETE

upd

updata table SET col = val WHERE id = x;若不指定where语句则更新全表。

更新多列则添加逗号即可。

语句中可以使用子查询。

IGNORE 关键字, 若upd语句更新多行, 更新某行时出错, 也继续。UPDATA IGNORE table即可。

del

删除整行。

删表更快的方式：TRUNCATE TABLE，删掉原来的表，建一个新表。

4 CREATE

处理现有表，在创建表时若已存在，若任要建表，则需手动删除已有表，再建表，**而不是直接覆盖**。

若仅想在一个表不存在时创建他，则应在表名后跟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。

DEFAULT 默认值

PRIMARY KEY 主键

引擎类型

外键不能跨引擎。

SHOW ENGINES可查看所有支持引擎。

MyISAM与InnoDB的区别

1. 存储结构，每个MyISAM在磁盘上存储成三个文件。InnoDB所有表都存在同一个数据文件中。
2. 存储空间，MyISAM可被压缩，存储空间较小，支持静态，动态，压缩表；InnoDB需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。
3. 可移植性、备份及恢复，MyISAM数据以文件存储，跨平台转移方便，备份恢复可单独对某个表操作；
4. 事务支持，InnoDB支持事务。
5. AUTO_INCREMENT。
6. 表锁，MyISAM表锁；InnoDB支持事务与行级锁，若非主键的where则会锁全表。
7. 全文索引，都支持。
8. 表主键，均允许无索引与主键，但InnoDB无主键时会自动生成一个主键。
9. 表行数，MyISAM保存有行数，InnoDB计算行数需遍历所有行。
10. CRUD，MyISAM执行大量SELECT语句；InnoDB执行大量增改操作。
11. 外键，InnoDB支持。

更新表

ALTER TABLE语句。

1. 添加列 ALTER TABLE table ADD 列。
2. 删除列 DROP COLUMN col;
3. 一般修改表前建议备份。需要建新表，将数据复制，在此基础上进行操作。

删除表

DROP TABLE table **永久删除表。**

重命名

RENAME TABLE table col1 TO col2;

5 视图

5.1 规则

1. 视图名唯一
2. 视图数目无限制
3. 创建视图须有足够的访问权限
4. 视图可以嵌套
5. 视图不能索引
6. 视图可与表一起使用，联结表与视图。

5.2 创建

CREATE VIEW创建

SHOW CREATE VIEW viewname;查看创建视图

DROP VIEW view删除

CREATE OR REPLACE VIEW更新

5.3 简化复杂联结

视图的最常见的应用之一是隐藏复杂的SQL，这通常都会涉及联结。请看下面的例子：

输入

```
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;
```

```
1 | SELECT cust_name
2 | FROM productcustomers
3 | WHERE customers.cust_id=1;
```

5.4 格式化检索出的数据

比如要格式化一个数据，将name, city以一定格式输出，要用Concat函数做一些格式化，这时可以将这个格式化数据包装成一个视图，就可以直接SELECT *这个视图了，例：

1. 将格式化数据包装成视图

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

2. 就可以直接查视图了

```
SELECT *
FROM vendorlocations;
```

5.5 过滤不需要的数据

过滤掉无邮件地址的客户：

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

5.6 使用视图与计算字段

一样的，包装成视图就行。

视图更新

视图不存储数据，只是一种中间工具，所以修改视图数据，实际上是对原表进行修改，所以对视图修改没有意义。

若视图中使用了分组、联结、子查询、并、聚集函数、DISTINCT、计算列则不可更新。

6 使用存储过程

保存一条或多条MySQL语句集合，可简单视作批文件。

简便，安全，高性能

注意

MySQL默认使用';'作为一条语句的结束，而存储过程类似函数，故在创建存储过程时若多条语句会发生书写某条语句后便结束书写，那肯定是不对的。

所以这里需要先将MySQL语句结束符进行修改，书写完毕后再修改回来，例：

```
DELIMITER //
CREATE PROCEDURE productpricing()
BEGIN
    SELECT Avg(prod_price) AS priceaverage
    FROM products;
END //
DELIMITER ;
```

先使用DELIMITER 将结束标记修改，写完存储过程后，再修改回来即可。

存储过程可以传递参数，类比函数的形参。

使用：CALL abc();若有参数，则括号内书写参数。

删除

```
DROP PROCEDURE abc; 不需要写括号，只写名称
```

变量

输入变量需用IN修饰，返回变量需用OUT修饰：

```
CREATE PROCEDURE productpricing(
    OUT p1 DECIMAL(8,2),
    OUT ph DECIMAL(8,2),
    OUT pa DECIMAL(8,2)
)
BEGIN
    SELECT Min(prod_price)
    INTO p1
    FROM products;
    SELECT Max(prod_price)
    INTO ph
    FROM products;
    SELECT Avg(prod_price)
    INTO pa
    FROM products;
END;
```

使用变量需用加@标识：

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

存储过程本身不输出结果，而是将结果存储在变量中，之后需再书写SELECT语句查询变量：

```
SELECT @priceaverage;
```

输出

@priceaverage
16.133571428

一个有in, 也有out的例子:

```
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;
```

使用:

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

复杂情景

```

-- Name: ordertotal
-- Parameters: onumber = order number
--           taxable = 0 if not taxable, 1 if 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;

```

可以添加注释--

上述场景为：自己定义两个变量，一个是临时存储最终金额的浮点数total，一个表示利率的taxrate。

1. 首先查询合计金额
2. 若需要税收，则会通过if语句
3. 最后将临时变量的值存储到返回变量中

if , elseif需要跟then, else不需要

检查存储过程

SHOW CREATE PROCEDURE ordertotal;

显示创建一个存储过程的语句

若要显示何时，创建者，需要使用SHOW PROCEDURE STATUS会列出所有存储过程

限制输出，可使用LIKE关键字指定过滤。

7 游标 (MySQL5以后)

可以理解为迭代器。

使用DECLARE创建游标

MySQL游标只可以应用于存储过程与函数。

如下在名为process...的存储过程中声明一个名为ordernumbers的游标

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

使用OPEN name打开游标， CLOSE name关闭游标。

看一个例子，将表中数据利用游标存到新表中：

先将order_num存到ordernumbers游标中，然后利用循环遍历游标每一行，将数据存到变量o中，向新表插入数据

```
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;

```

关于循环终止

```
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。

8 触发器 (MySQL5以后)

用于响应upd, del, insert而自动执行的一条MySQL语句

创建

需要声明的信息

1. 唯一触发器名
2. 关联的表
3. 触发器该响应的活动 (DELETE、INSERT或UPDATE)

4. 触发器何时执行 (处理前或之后)

CREATE TRIGGER创建

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

注意：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

```
DROP TRIGGER x;
```

insert 触发器

1. 在INSERT触发器代码中，可以引用一个名为NEW的虚拟表，访问被插入的行；
2. 在BEFORE INSERT触发器中，NEW中的值也可以被更新（允许改变被插入的值）；
3. 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。

MySQL5之后不允许触发器返回结果集，故可将结果放入变量中：

```
1 CREATE TRIGGER hello AFTER INSERT ON class
2 FOR EACH ROW SELECT NEW.c_id INTO @ee;
```

插入后将该次插入时生成的AUTO_INCREMENT的值放入ee变量中。

然后就可以在ee获取到该值了：

```
mysql> select @ee;
+-----+
| @ee  |
+-----+
|    2 |
+-----+
```

通常before用于数据验证与净化，比如可以将某个字段大写：

```
1 CREATE TRIGGER hello BEFORE INSERT ON class
2 FOR EACH ROW SET NEW.c_name=Upper(NEW.c_name);
3
4 insert into class(c_name,teacher_id) values('asdf',2);
```

这样做保证插入数据之后，name字段是大写的。

delete 触发器

1. 触发器代码内部可以引用OLD虚拟表，访问删除的行。
2. OLD内值只读。

比如可以使用一个before触发器，将要删除的行从OLD表备份一份到其他表中：

```
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;
```

若由于某种原因，订单不能存档，DELETE本身将被放弃。

upd 触发器

1. OLD访问旧信息，NEW访问新信息
2. NEW, OLD中关于值的访问与之前一致

例：将名字在插入前改为大写：

```
1 CREATE TRIGGER upd BEFORE UPDATE ON class
2 FOR EACH ROW SET NEW.c_name=Upper(New.c_name);
3
4 update class SET c_name='hello' where c_id=2;
```

	c_id	c_name	teacher_id
▶	1	cy's	1
	2	HELLO	2

9 事务处理

InnoDB支持事务处理。

事务处理用来维护数据库的完整性，它保证成批的MySQL要么全部执行，要么全部不执行。

使用 START TRANSACTION 开启事务。

使用 ROLLBACK回退事务，但无法撤销CREATE与DROP。

使用COMMIT提交事务。

支持部分回退（使用保留点）

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，可如下进行：

输入

```
ROLLBACK TO delete1;
```

该回滚点之后的事务全部失效，之前的不失效。

保留点在事务处理完成后自动释放。

修改默认的提交行为

```
SET autocommit=0;
```



标志为连接专用 autocommit标志是针对每个连接而不是服务器的。

事务的数据读取问题

1. 脏读，事务a读取了事务b未提交的数据，事务b回滚了事务，该数据无意义。
2. 不可重复读，事务a第一次读取了一条记录，事务b修改了记录，a第二次读取该记录发现值改变，数据不重复了，系统没有读取到重复的数据。
3. 幻读，事务该次查询出来的数据不足以支撑后续的业务操作，比如事务a查询了一次数据，发现某数据不存在，于是想要插入数据，结果显示该数据已存在（主键唯一）无法插入（另一个事务b在某时刻插入了该数据单）。

事务特性ACID

1. 原子性，一个事务是不可分割单位，事务中的操作要么都做要么都不做
2. 一致性，事务必须使数据库从一个一致性状态变到另一个一致性状态。数据表中的数据修改要么是所有操作一次性修改，要么根本不动
3. 隔离性，一个事务的执行不能被其他事务干扰，并发时一个客户端使用事务操作一个数据时，另一个客户端不可以对该数据进行操作，若操作则会等到事务完成时才会操作
4. 持久性，事务一旦提交，它对数据库中数据的改变是永久性的，接下来的其他操作或故障不应该对其有任何影响。事务提交后则不会再回滚（回滚无效）

事务隔离级别

使用select @@transaction_isolation;可查看默认的事务隔离级别。

使用 set session transaction isolation level 隔离级别。

1. 读未提交 (read uncommitted) , 所有事务可以读取其他事务未提交的数据，会引发数据的三种读取问题。
2. 读已提交 (read committed) , 只可以读取到其他事务提交的数据，只可以避免脏读。
3. 可重复读 (repeatable read) , 确保事务可以多次从一个字段中读取相同值，事务持续期间禁止其他事务对该字段进行修改，避免了脏读与不可重复读。
4. 串行化 (serializable) , 事务持续期间，禁止其他事务对该表进行任何插入，修改，删除操作（相当于表锁），性能低下。

10 外键

1. 建表时增加外键：

```
[constraint `外键索引名称`] foreign key(column) references 其他表(要关联的其他表的字段名);
```

[]部分可以不写，系统会自动分配一个索引名称，但是后续若要删除外键需要知晓外键名称，故最好自己定义一个。

2. 修改表：

```
alter table tablename add constraint `外键索引名称` foreign key(column) references 其他表(要  
关联的其他表的字段名);
```

注意：修改表时增加外键需要满足外键本身就具有的特性（从表中该字段的所有值主表中必须存
在）。

外键不允许修改只能删除：

```
alter table tablename foreign key `外键索引名称`;外键创建会自动增加一个索引，外键删除索引仍  
存在
```

主表：被外键引用的表。

从表：引用外键的表。

约束模式

1. **strict** 严格模式，默认，不允许操作
2. **cascade** 级联模式，主表变化，从表跟着变化
3. **set null** 置空模式，主表变化（删除），从表对应记录设置为空（从表该字段允许为空才可以）

设置约束：针对upd与del

```
on update cascade on delete set null 更新级联，删除置空。
```

```
alter table my_student add foreign key(class_id)  
references my_class(class_id)  
  
-- 约束  
on update cascade  
on delete set null;
```

11 锁

show open tables; 可查看表的锁定情况。

lock table tablename1 read(write),tablename2 read(write); 可以同时对多张表进行加锁。

1. 读锁是共享锁，不同客户端可读取加了读锁的表，但不可以对其进行修改等工作。**但是只能读取该
锁定的表，不能读别的表。别的客户端对其他表进行操作时，会阻塞。**
2. 写锁不共享，**自身不可以查看其他表，只可以对锁表操作**，其他用户若要查看该表则阻塞。

MyISAM执行select前会自动给涉及到的表加读锁，执行修改前会自动加写锁。

读锁会阻塞写，但不阻塞读，写锁都堵塞。

InnoDB支持行锁：

事务a对某行进行了修改操作，但未提交，事务b若要修改此行，则阻塞。

