前言

什么是数据库

数据库：高效的存储的处理数据的介质（介质主要有两种：磁盘的内存）

数据库的分类

数据库给予存储介质的不同：进行了分类，分位两类：关系型数据库（SQL）和非关系型数据库（NOSQL），不是关系型的数据库都叫做非关系型数据库。

两种数据库的区别

1.关系型数据库：安全（保存磁盘基本不可能丢失），容易理解，但比较浪费空间，相对非关系型数据库效率低。

2.非关系型数据库：效率高，但不安全。因为数据是保存在内存里的，所以断电数据就丢失了

不同数据库阵营的产品

关系型数据库

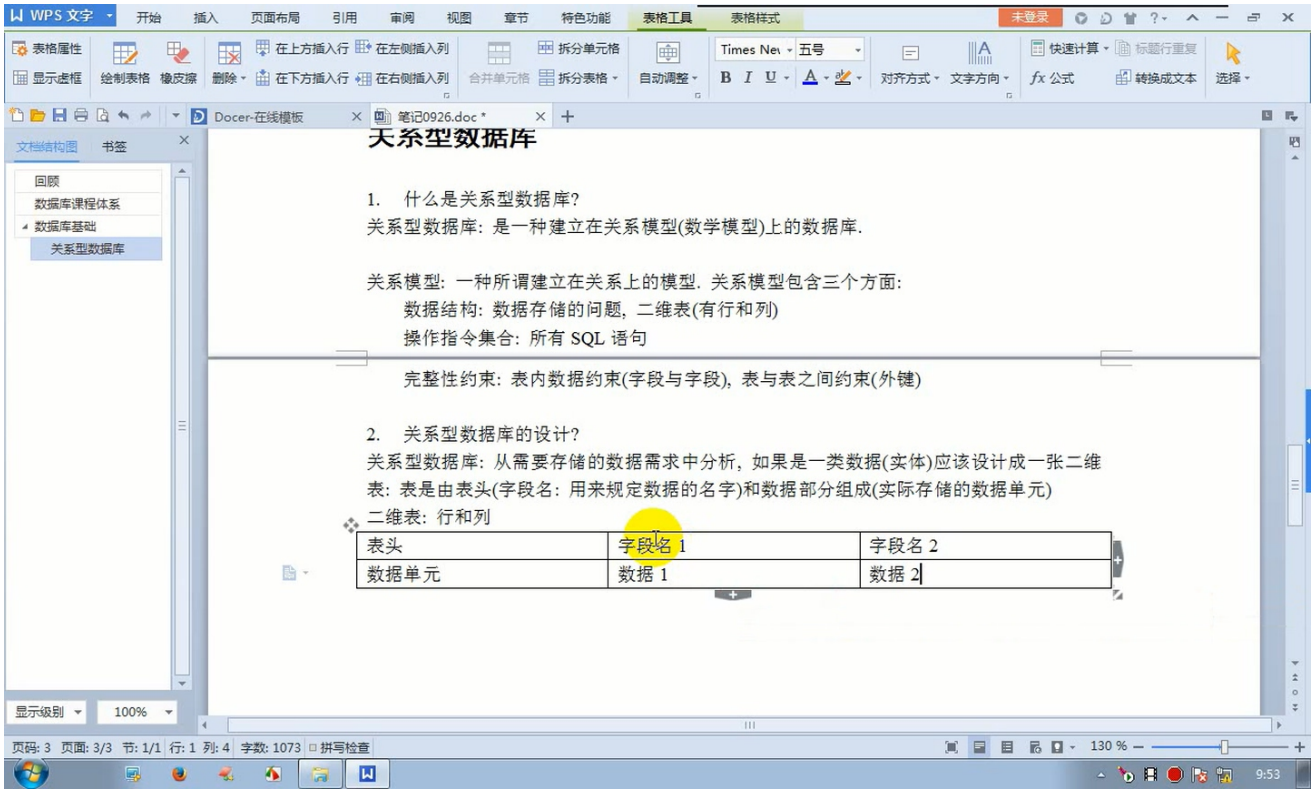
大型：Oracle ， DB2

中型：SQL-Server ,Mysql...

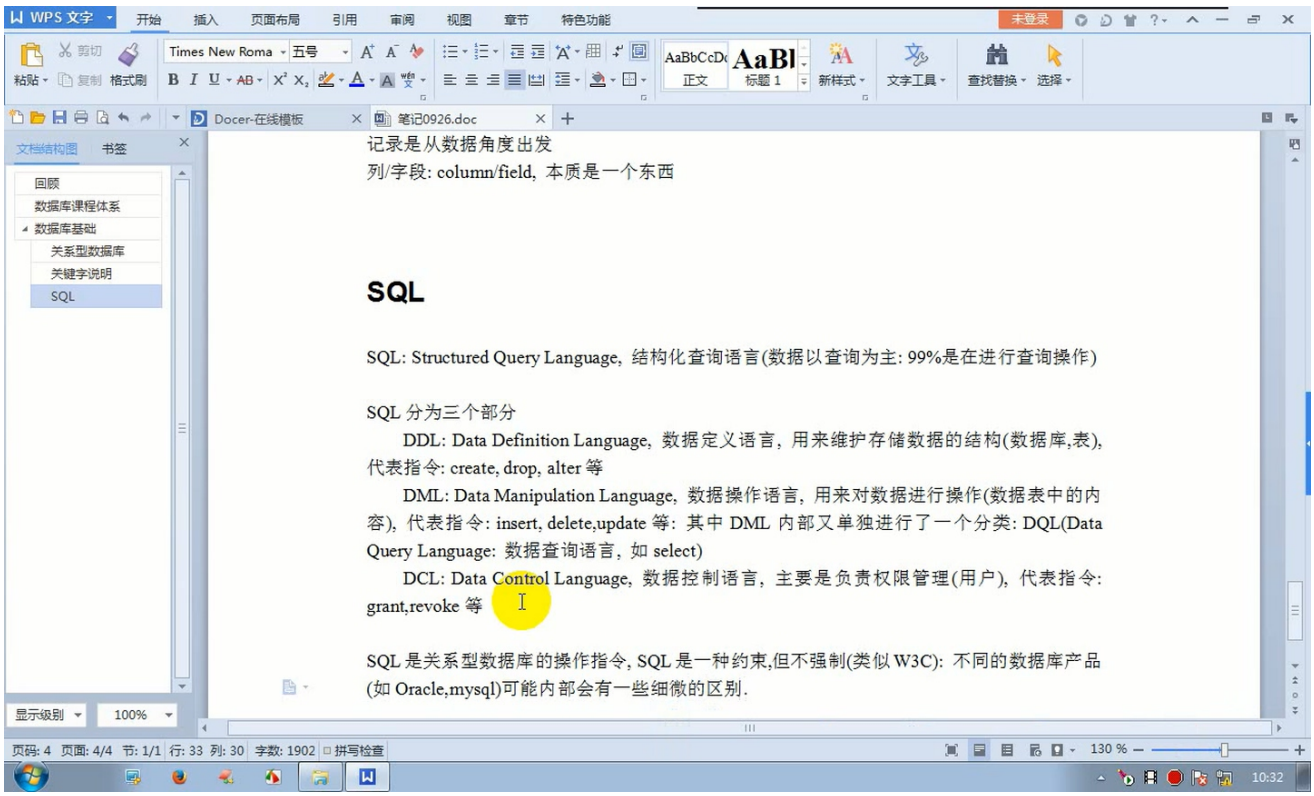
小型：access...

非关系型数据库：memcached，mongodb，redis（可以同步到磁盘，相对安全）

什么是关系型数据库


关系型数据库
1. 什么是关系型数据库?
关系型数据库：是一种建立在关系模型(数学模型)上的数据库.
关系模型：一种所谓建立在关系上的模型. 关系模型包含三个方面:
数据结构：数据存储的问题，二维表(有行和列)
操作指令集合：所有 SQL 语句
完整性约束：表内数据约束(字段与字段)，表与表之间约束(外键)
2. 关系型数据库的设计?
关系型数据库：从需要存储的数据需求中分析，如果是一类数据(实体)应该设计成一张二维表：表是由表头(字段名：用来规定数据的名字)和数据部分组成(实际存储的数据单元)
二维表：行和列
表头
字段名 1
字段名 2
数据单元
数据 1
数据 2

SQL的定义


记录是从数据角度出发
列/字段: column/field，本质是一个东西
SQL
SQL: Structured Query Language，结构化查询语言(数据以查询为主: 99%是在进行查询操作)
SQL 分为三个部分
DDL: Data Definition Language，数据定义语言，用来维护存储数据的结构(数据库,表)，代表指令: create, drop, alter 等
DML: Data Manipulation Language，数据操作语言，用来对数据进行操作(数据表中的内容)，代表指令: insert, delete,update 等：其中 DML 内部又单独进行了一个分类: DQL(Data Query Language: 数据查询语言，如 select)
DCL: Data Control Language，数据控制语言，主要是负责权限管理(用户)，代表指令: grant,revoke 等
SQL 是关系型数据库的操作指令, SQL 是一种约束,但不强制(类似 W3C)：不同的数据库产品(如 Oracle,mysql)可能内部会有一些细微的区别.

SQL指令

注意！！！

如果命名使用到系统关键字或者保留字时，需要用反引号括住。

如创建数据库使用到系统关键字database，那么该如何创建以database名的数据库

sql命令：

```
create database `database` charset utf;
```

库操作命令

- **连接数据库**：mysql.exe -h主机名 -P端口号 -u用户名 -p回车（后输入密码

```
Microsoft Windows [版本 10.0.16299.15]
(c) 2017 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>mysql.exe -hlocalhost -P3306 -uroot -p
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.21 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

- **断开连接**： exit | quit | \q
- **创建数据库**：Create database 数据库名字（字母_数字）库选项

  库选线：用来约束数据库，分为两个选项

  1.字符集设定：charset/character set 具体字符集（数据库存储数据的编码格式）中文常见 GBK，utf8

  2.校对集设定:collate 具体校对集（数据比较规则）。校对集依赖字符集

- **数据库查询：**

  1.show databases; 查询所有数据库

  2.show databases like '查询模式'；//模糊查询

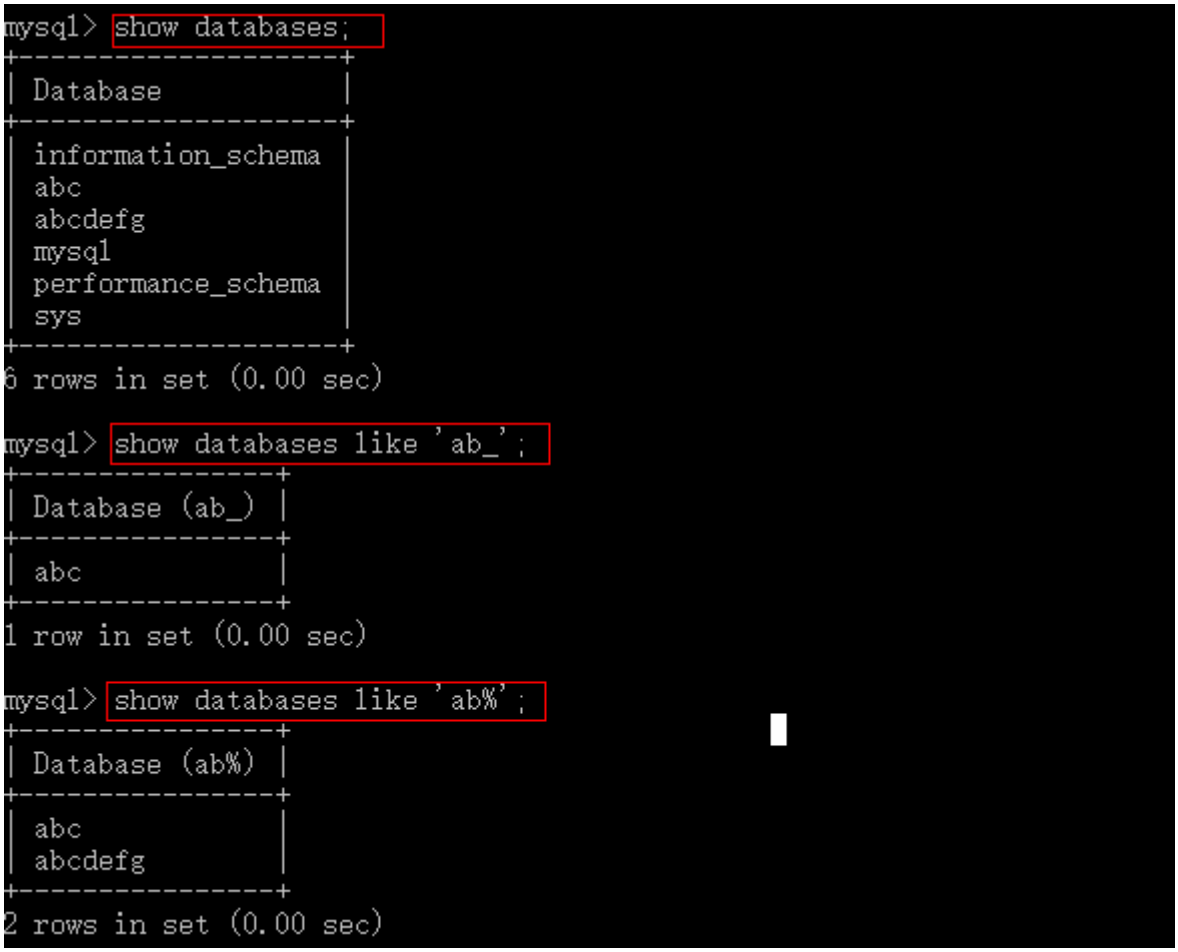
  查询模式：% _

```
%:匹配多个字符

_ :匹配单个字符
```

注意！！！

如果数据库名与出现_或者%。在匹配查询模式时需要加转义符 ""



- **更改数据据库：**Alter 数据库名 【库选项】

  注意更改数据库只能修改库选项：字符集和校对集
- **删除数据库：**drop database 数据库名
- **查询数据库创建语句：**show create database 数据库名

表结构操作命令

- **创建表：**

  create table [if no exists] **表名**(

  字段名 数据类型，

  字段名 数据类型（最后一个字段不需要加逗号）

) [表选项]；

  if no exists： 如果不存在，进行检查。如果存在就不执行以后的操作

表选项:

```
1.字符集 ： charset/character set   字符集

2.校对集 ： collate  校对集

3.存储引擎 ：  engine 具体的存储引擎（innodb和myisam）
```

但此时并不能创建表，因为并未指定数据库

所以在创建表之前需要预先指定操作数据库

1.显示指定数据库：在创建表明前声明数据库。如↓

create table [if no exists] **数据库.表名**(

字段名 数据类型，

字段名 数据类型（最后一个字段不需要加逗号）

) [表选项]；

2.隐式指定数据库：在创建表指令之前先声明使用那个数据库

输入use 数据库名；

- **查询表创建语句：**show create table 表名

- **修改表本身**

  修改表本省可分为修改表和修改字段

  修改表结构

  1.修改表名

  rename table 原表明 to 新表名

  2.修改表选项

  alter table 表名 表选项

  修改字段

  1. 新增(添加新字段)

  ```
  alter table 表名

  add column
  ```

  字段名 数据属性

  【列属性】【位置】

  如:

```
alter table student add column id int first;

alter table student add column id int after age;
```

位置：

first： 该字段排在首位

after： 指定排在哪个字段后头

2. 修改 (可修改数据类型，位置，列属性)

```
alter table 表名

modify 字段名  更改数据类型

【列属性】【位置】
```

3. 重名(不单可以修改字段名，而且可以修改数据类型，列属性和位置)

```
alter table 表名

change 原字段名 新字段名 数据类型

【列属性】【位置】
```

4. 删除

```
alter table 表名 drop 字段名
```

表数据操作命令

- **添加表数据：**

  1.给全表字段插入数据，不需要指定字段列表，要求数据的值出现的顺序必须与表中设计的字段出现的顺序一样：凡是非数值数据，都需要使用单引号包裹

  insert into 表名 values (值列表)【,(值列表)】； --可以一次性插入多条记录

  如：

  insert into student values

  (1,'man','小明'),

  (2,'woman','小花');

  2.给部分字段插入数据，需要选定字段列表：字段列表出现的顺序与字段的顺序无关；但是值列表的顺序必须与选定的字段顺序一致。

  insert into 表名(字段列表，字段列表) values(值列表)【,(值列表)】；

如：

```
insert into student(id,name)
values
(1,'小明'),
(2,'小花');
```

- **查询数据：**
  - 查询所有数据

    ```
    select  *  from  表名  【where条件语句】;
    ```

  - 查询指定字段数据

    ```
    select  字段名[,字段名]  from 表名 【where条件语句】;
    ```

- **更新数据：**

```
update  表名  set  字段名  =  值  【where条件语句;】  --建议都有where:要不就更新全部
```

- **删除数据：**

```
delete  from  表名  【where条件语句】--建议都有where：要不全部数据都会被删除
```

表数据操作总结

增：insert into 表名【(字段名)】 values（字段值，字段值）；

删：delete from 表名 【where条件语句】；

改：update 表名 set 字段名=字段值 【where条件语句】；

查：selete *【指定查询字段】 from 表名 【where条件语句】；

删除注意！！！

表结构操作

drop：表示删除表，删除字段。同样也可以删除数据库。drop database 数据库名;

字段数据操作

delete：表示删除数据

-

变量

系统变量简介

所有系统的实现(如字符集,校对集),都是系统内部变量进行控制的

所以可通过修改系统变量进行更改系统设置.

如:修改mysql系统编码,实现对服务端对客户端收发数据的编码控制,能够解决乱码问题.

查看变量

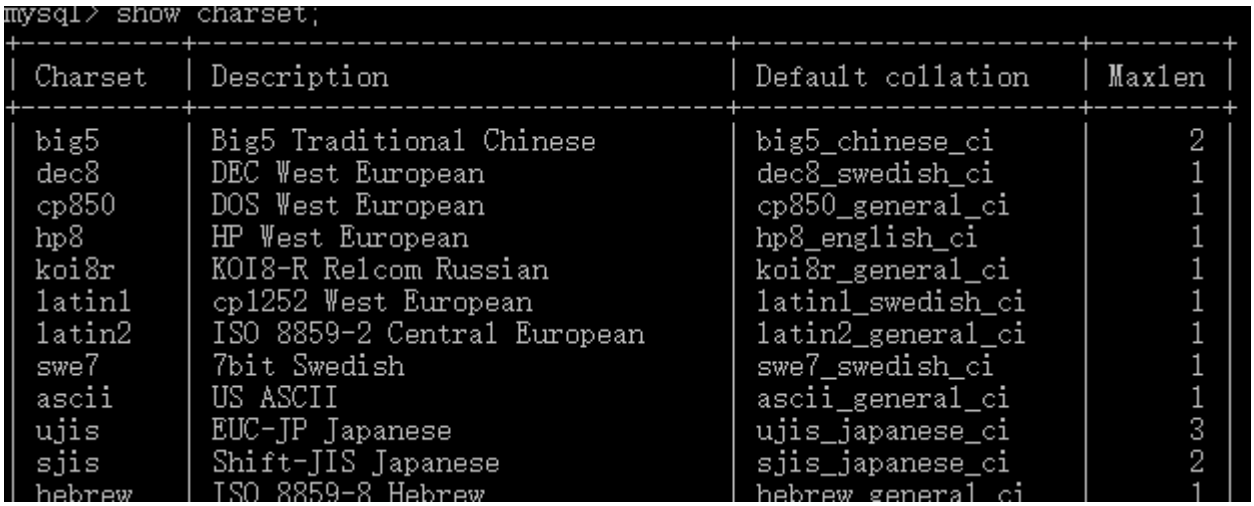
语法

show variables like '模糊变量名';

select @@变量名;

- 查看系统支持所有编码集

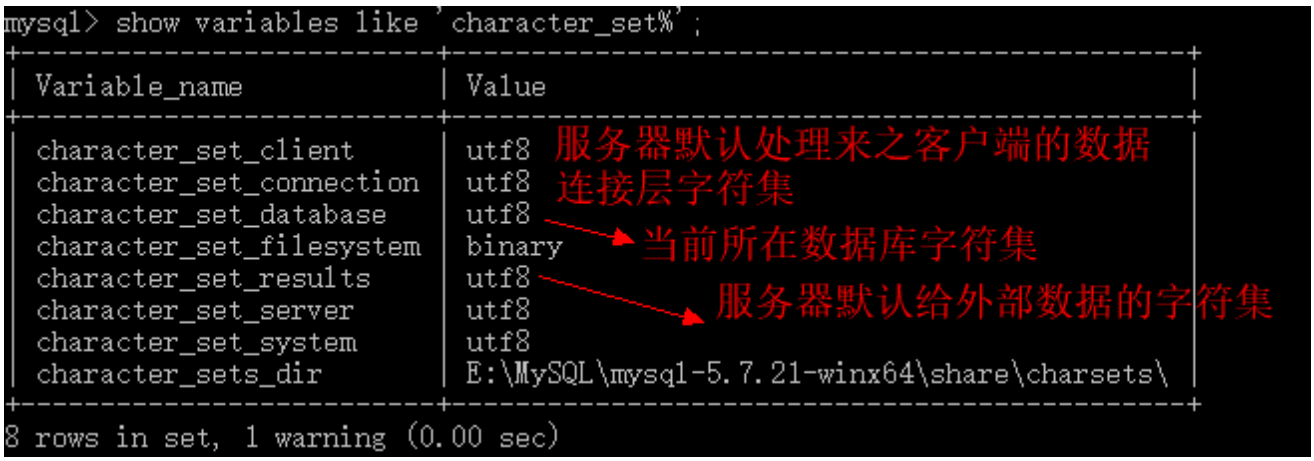
show charset;



```
mysql> show charset;
+----------+---------------------------------+---------------------+--------+
| Charset  | Description                     | Default collation   | Maxlen |
+----------+---------------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese        | big5_chinese_ci     |      2 |
| dec8     | DEC West European               | dec8_swedish_ci     |      1 |
| cp850    | DOS West European               | cp850_general_ci    |      1 |
| hp8      | HP West European                | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian           | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European            | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European     | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                    | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                        | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese                 | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese              | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew               | hebrew_general_ci   |      1 |
```

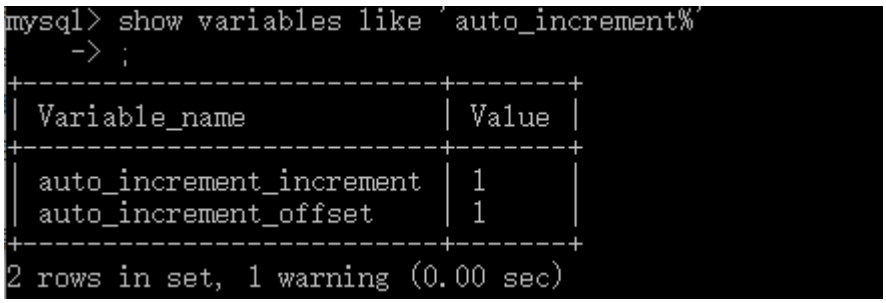
- 查看当前系统编码

```
show variables like 'character_set%';
select @@character_set_client;
```



- 查看自动增长变量值

```
show variables like='auto_increment%';
```



修改会话期间系统变量

当前修改值只限与会话期间，其他客户端并不会影响。

由于MySQL中'='更作用于进行比较，那么赋值符号有另外一种：':='；

set 变量名 := 值;

set @@变量名 :=值;

如:

```
set auto_increment_increment:=2;
set @@auto_increment_increment:=2;
```

修改全局系统变量

一次修改，永久有效。其他客户端都会有效。

set global 系统名:=值;

set global @@系统名:=值;

```
set global auto_increment_increment:=2;
set global @@auto_increment_increment:=2;
```

自定义变量

为了区分系统变量和自定义变量。

@@：系统变量

@：自定义变量

语法

- 声明和定义全局变量

set @变量名='值';

```
set @name='小文';
```

- 声明和定义局部变量

  全局变量可以在任何地方使用，局部变量只能在函数内部或触发器内部使用。

```
declare 变量名 数据类型;
```

从数据表中取值并赋值给变量

select 赋值字段名[，赋值字段名] from 表名 into @变量名;

```
select name from class into @name;
```

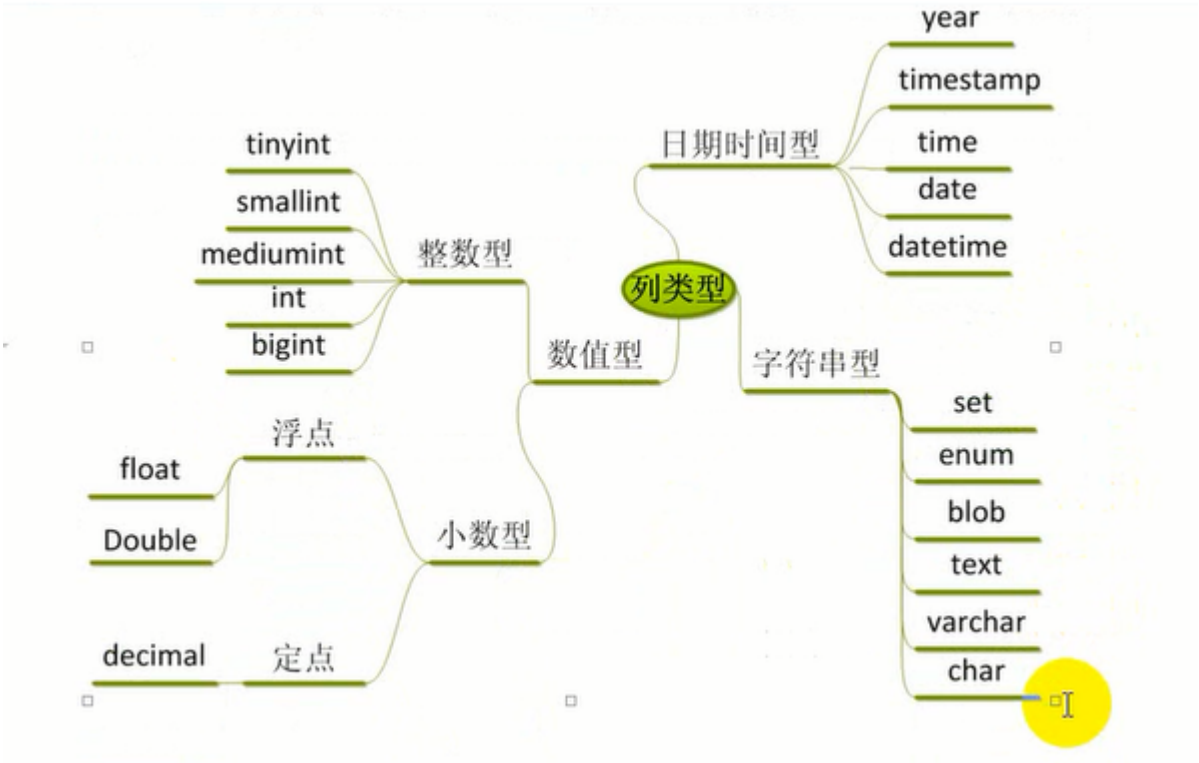
数据类型

为何要有众多数据类型?

因为数据库数据是以二维表架构存储数据。

那么二维表架构的特点就是，即使在该字段列中暂无该数据，但这个字段列仍是占据着空间资源。

所以有了众多数据类型后，就可以根据需求定义数据类型进而减少空间资源的浪费。



整数型还可分为有符号和无符号

默认是有符号型

有符号：代表数值有正负数区别。

无符号：则无正负数特性。

要想设定无符号型需加unsigned关键字。如：craete table 表名（age int unsigned）；

因为有符号有正负数的特性，那么它的数值有效范围是+-（2^类型所占字节）/2。

相对有符号，则无符号的有效范围值是从0开始——（2^类型所占字节）。

如：

类型	字节	最小值（有符号/无符号）	最大值（有符号/无符号）
TINYINT	1	-128/0	127/255
SMALLINT	2	-32768/0	32767/65535
MEDIUMINT	3	-8388608/0	8388607/16777215
INT/INTEGE	4	-2147483648/0	2147483647 /4294967295
BIGINT	8	-9223372036854775808/0	9223372036854775807 /18446744073709551615

显示宽度

显示宽度

显示宽度：没什么特别的含义，只是默认告诉用户可以显示的形式而已。世界上的用户是可以控制的，这种控制不会改变数据本身的大小。默认数据类型的显示长度是改类型最大显示长度。

更改显示长度: create table teacher(age **int(2)**); //加（显示长度）

显示长度的意义： 在当数据不够显示宽度的时候，会自动让数据变成对应显示宽度。通常需要搭配一个前导0来增加宽度，不改变值得大小：zerofill（零填充）。

设置zerofill（零填充）： alter table 表名 add column 字段名 数据类型（显示长度） zerofill;

```
alter table student add column id int(3) zerofill;
```

添加了zerofill后，数据类型固定是unsigned。

小数型

小数型：带有小数点或者范围超出整数得数值类型

SQL中；将小数型细分为：浮点型和定点型

浮点型：小数点浮动，精度有限，而且回丢失精度

定点型：小数点固定，精度固定，不会丢失精度

浮点型

浮点型数据是一种精度型数据：因为超出范围后，会丢失精度（自动四舍五入）

浮点型：理论上分为两种精度

float：单精度，占4字节。精度范围大概在7位左右。

double：双精度，占用8字节。精度范围大概在15位左右。

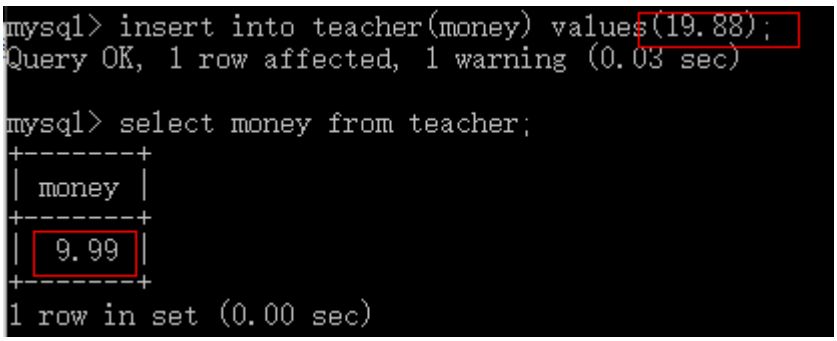
创建浮点数表:

默认直接声明float表示没有小数部分；

float（M,D）：M代表总长度。D代表小数部分长度。整数部分长度=M-D；

如 craete table 表名（money float（10，2））；

```
alter  table teacher add column money **float(10,2)**;
```



注意！！！如果精度型数据的整数部分可以超出长度，但超出部分会被舍弃。而小数部分超出则被四舍五入

定点型

定点型：绝对的保证整数部分不会四舍五入（不会丢失精度），小数部分可能会丢失（理论不上不会丢失，因为小数部分数值太大，基本上使用不到）。

decimal：动态去顶数据类型的大小。大致是每9个数字，采用4个字节存储。整数和小数分开计算。

M：最大是65。D：最大是30默认是decimal（10，0）。

精度型和定点型的区别

精度型因为四舍五入导致超出精度长度。那么系统默认是允许的。而定点型是不允许的。

精度型整数部分可以超出整数长度（但会造成数据丢失）。而定点型不允许。

时间日期类型

datetime：时间日期，格式是YYYY-mm-dd HH:ii:ss，表示的范围是1000到9999年，有0值：0000-00-00 00:00:00

date:日期，就是datetime中的date部分 YYYY-mm-dd 0000-00-00

time:时间【段】指定的某个区间之间。时间到+时间

timestamp：时间戳，并不是时间戳，只是从1970年开始的YYYY-mm-dd HH:ii:ss格式与datetime一致

year：年份，两种形式，year（2）和year（4）：1901-2156；

类型	存储空间（字节）	最小值（理论）	最大值（理论）
FLOAT	4	-3.402823466E+38	3.402823466E+38
DOUBLE	8	-1.7976931348623157E+308	1.7976931348623157E+308
DECIMAL	变长，大致是每9个数字，采用4个字节存储。整数和小数分开计算	M，最大是65 D，最大是30 默认是10，2 -（65个9）	（+65个9）

timestamp特点：只要当前所在的记录被更新，那么该数据段timestamp字段一定会自动更新当前时间

在许多编程语言中都有许多函数处理时间。可以将时间戳转化成各种时间格式。

字符串类型

在SQL中，将字符串类型分为6类：char，varchar，text，blob，enum，set

定长字符串

定长字符串：char,磁盘（二维表）在定义结果的时候，就已经确定了最终数据和存储长度。

char（L）：L代表length，可以存储的长度，单位为字符，最大长度值可以为255

char（4）在utf8环境下所需要的空间大小为4*3=12个字节。

变长字符串

变长字符串：varchar在分配空间的时候，按照最大的空间分配：但是实际最终用了多少，是根据具体的数据了确定。

varchar（L）:L表示字符长度，理论长度最大是65536的字符，但是会多出1到2个字节来存储确定的实际长度。

varchart（10）：存储了10个汉字，那么所占的空间大小是：10*3+1=31字节

为什么有时候确定字节的大小是1个或者2个呢?

因为如果字符长度小于256的话，一个字节就已经够表示所占的字节长度（2^8=256），但是如果超过256的话就需要最多2个字节来表示了。2^16=65536。理想最大字符长度。

那么如何选择定长或者变长字符串呢?

定长字符串类型比较浪费空间，但效率高。如果数据基本确定长度的话，就可以使用定长字符串。

如：身份证，电话号码。。。。。。

变长字符串类型比较节省空间，但效率低（因为每次取出数据都要拿到字符长度在去拿指定字符长度的数据）：如果数据不能确定长度的话，就可以使用定长字符串。

如：地址。。。。。。

文本字符串

如果数据了非常大，通常说超过255个字符就会使用文本字符串

文本字符串根据存储的数据的格式进行分类：text和blob

- Text：存储文字（二进制数据实际上都是存储路径）
- Blob：存储二进制数据（图片。。。但通常不用）

枚举字符串

枚举（enum），事先将所有可能出现的结果设计好，实际上存储的数据必须是规定好的数据中的一个。

枚举的使用方式

定义：enum（可能出现的元素列表）；如 create table student（sex enum（'男','女','保密'））chartset utf8;

使用：存储数据只能存储上面定义好的数据

有效数据: insert into student values ('男');

```
insert into student values (1);
```

特点： 枚举可以插入数值，之后系统会自动进行转换。按照从1开始。即1：男；2：女；3：保密。

无效数据：insert into student values ('人妖');

枚举的作用： 节省存储的空间,枚举实际存储的是数值而不是字符串本身。

枚举的原理： 枚举在进行数据规范的时候（定义的时候，即创建表时），系统会自动建立一个数字与枚举元素的对应关系（关系放在日志中，一般查看不到）；然后在进行数据插入的时候和，系统自动将字符串转换成对应的数字存储，然后在进行数据提取的时候，系统自动将数值转换成对应的字符串显示。

但由于在中间多了层转换，导致效率较低。

在 mysql 中,系统也是自动转换数据格式的：而且基本与 PHP 一样(尤其是字符串转数字)

证明字段存储的数据是数值：将数据取出来 +0 就可以判断出原来的数据存的到底是字符串还是数值：如果是字符串最终结果永远为 0，否则就是其他值.

```
mysql> -- 将字段结果取出来进行+0运算
mysql> select gender + 0, gender from my_enum;
+------------+--------+
| gender + 0 | gender |
+------------+--------+
|          1 | 男     |
|          3 | 保密   |
+------------+--------+
2 rows in set (0.00 sec)
```

集合字符串

与枚举字符串相似。

好比的区别就是枚举时单选框，而集合字符串时多选框

都是能够降低磁盘空间的利用率，但同时都会降低效率。

集合字符串的原理

```
create table my_set(
hobby set('篮球','足球','乒乓球','羽毛球','排球','台球','网球','棒球')
--                  足球                          台球    网球
-- 集合中：每一个元素都是对应一个二进制位，被选中为1，没有则为0：最后反过来
--          0     1      0          0      0      1       1      0
-- 反过来    01100010 = 98

)charset utf8;

-- 插入数据
insert into my_set values('足球,台球,网球');
insert into my_set values(3);

-- 查看集合数据
select hobby + 0, hobby from my_set;

-- 98转成二进制 = 64 + 32 + 2 = 01100010
```

定义：create table student(exercise set('足球','篮球','排球','乒乓球','网球','台球'));

插入：insert into student (exercise) value('足球，篮球');

```
        insert into student (exercise) value('3');
```

与枚举类似，都可以插入数值，之后系统会自动进行进制的转换，将10进制转换成2进制，在进行位运算。查看哪位为高位即被选中。

MySql的记录长度

Mysql中规定：任何一条记录最长不能超过65535个字节。（这也就时说varchar永远也不能达到理论值65535个找字符）

Varchar的实际长度能达到多少呢？看字符集编码

utf8下1个字符3个字节

gbk下1个字符2个字节

由于varchar需要额外有1到2个字节用来存储字符长度。

那么varchar在不同编码下的最大字符大小是

```
mysql>
mysql> -- 求出varchar在utf8和GBK下的实际最大值
mysql> create table my_utf8(
    -> name varchar(21844) -- 21844 * 3 + 2 = 65532 + 2 = 65534
    -> )charset utf8;
Query OK, 0 rows affected (0.12 sec)

mysql>
mysql> create table my_gbk(
    -> name varchar(32766) -- 32766 * 2 + 2 = 65532 + 2 = 65534
    -> )charset gbk;
Query OK, 0 rows affected (0.07 sec)

mysql>
```

那么在特殊情况下。如小说作文该怎样存储到数据库呢?

mysql中text文本字符串是不会占用记录长度(额外存储),但是text也是属于记录的一部分。仍是占据着一定的记录长度的一部分（最大不过10字节）。主要用于保存text文本字符串的内存地址和字符长度（不然上哪找哦！）

注意！！！如果varchar的字节长度超出记录长度的话，那么定义的varchar类型会自动转换成text类型

列属性

列属性简介

真正约束字段的是数据类型，但是数据类型的约束很单一，需要有一些额外的约束，用来更加保证数据的合法性。

定义

create table student （age int 列属性）；

null： 可以为空（默认）

```
create table student(age int null);
```

not null：不可为空

```
create table student(age int not null);
```

comment: 描述该字段。

```
create table student(age int comment "年龄");
```

defaul： 设置默认值

```
create table student(age int defalut 17);
```

primary key : 主键

主键：，主要的键，一张表只能有一个字段使用对应的键，用来唯一的约束该字段里面的数据（不能重复）。主要用来标识一个数据段。一张表只能最多一个主键。

增加主键主要有三种

> 在创建表的时候，直接在字段之后跟primary key关键字。

如:

```
create table student (age int primary key) ;
```

但只能使用一个字段作为主键。

> 在创建表的时候，在所有字段子后，使用primary key(主键字段列表)来创建主键。

如:

```
create table student (
            number int ,
            course char(10) ,
            score tinyint unsigned,
            primary key(number,course)
            ) charset utf8;
```

使用两个字段作为主键（**这叫做复合主键**），两个字段数据组合成的数据作为唯一性。用来表示某个字段数据。

> 当表已经创建好之后，额外追加主键，可以通过修改字段属性，也可以直接追加。

```
1.alter table 表名 modify 字段名 数据类型 primary key;
2.alter table 表名 add primary key(字段列表);
```

删除主键： alter table 表名 drop primary key;

注意

如果字段加了主键,那么该字段会自动添加not null(不能为空)属性.

unique key :唯一键

简介

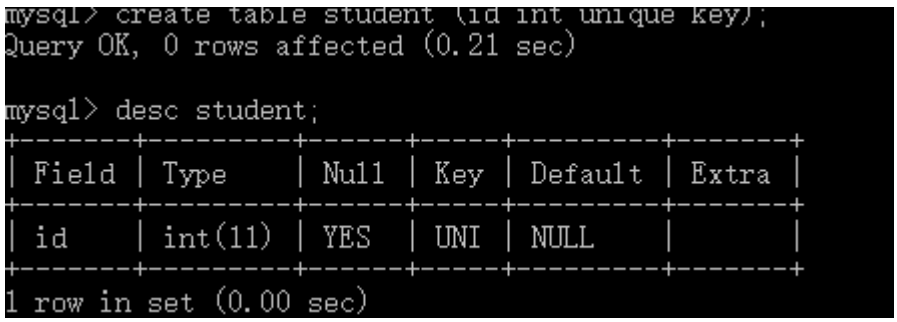
一张表往往有很多字段需要具有唯一性(数据不能重复),但是一张表中只能有一个主键

那么唯一键(unique key) 就可以解决表中很多字段需要唯一性约束的问题.

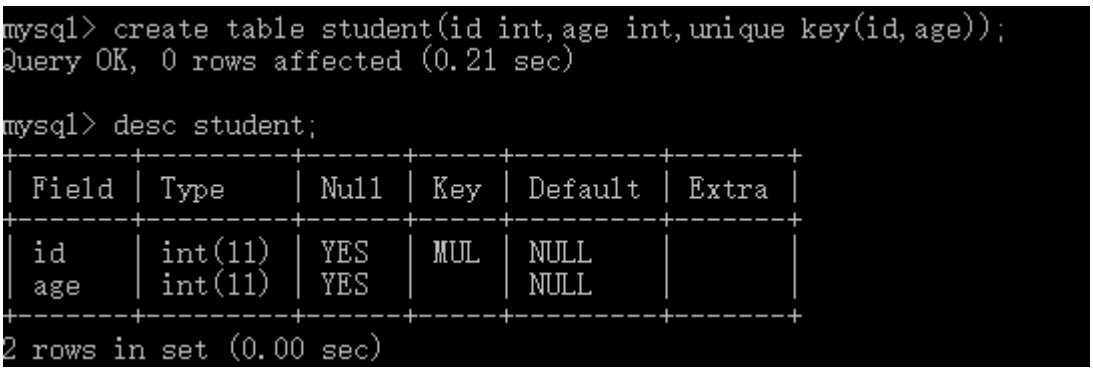
> 唯一键的本质和主键差不多.但唯一键允许字段为null(空),而且可以多个为空(空字段部参与唯一性的比较)

设置唯一键的三种方法大致和设置主键类似.

> 在创建表时设置:create table student (id int unique key);



> 在创建表的时候，在所有字段子后，使用unique key(唯一字段列表)来创建多个字段组合成的唯一性。



> 当表已经创建好之后，额外追加唯一键，可以通过修改字段属性，也可以直接追加。

alter table student add unique key(字段名);

删除唯一键

alter table 表名 drop index 唯一键字段名

注意

如果表内没有主键,而一个字段既有not null和unique属性. 那么该字段就可以看成时主键.(但并不是主键,只是跟主键并无差别).

auto_increment : 自增长

自增长：当对应的字段，不给值，或者说不给默认值，或者给null的时候，会自动被系统触发。系统会从当前字段中已有的最大值在进行+1操作， 赋值给最新的字段.

自增长通常需要跟主键搭配

- 定义:

  1.任何一个字段要做自增长必须前提是本身是一个索引(key一栏有值,也就是主键)

2.自增长字段必须是数字.

3.一张表最多只能有一个自增长.

索引

简介

系统根据某种算法,将已有的数据(未来可能新增的数据),单独建立一个文件;文件能够实现快速的匹配数据,并且能够快速的找到对应表中的记录.

意义

1.提升查询数据的效率

2.约束数据的有效性(唯一性等)

增加索引的前提条件

索引本身会产生索引文件(有可能比数据文件还大),会耗费磁盘空间.

如果某个字段需要作为查询的条件经常使用,那么可以使用索引.

如果某个字段需要进行数据的有效性约束,也可以使用索引(主键,唯一键)

添加索引方式

1.主键索引:primary key

2.唯一索引:unique key

3.全局索引:fulltext index

4.普通索引:index

数据库表关系

一对一

简介

一张表的一条记录一定只能与另外一张表的一条记录对应.

举例

学生表:姓名.性别,年龄,体重,婚姻状况,籍贯.家庭住址,紧急联系人

Id(P)	姓名	性别	年龄	体重	身高	婚姻	籍贯	住址	联系人

表设计成以上形式:符合要求,其中姓名,性别,年龄,身高,体重属于常用数据,但婚姻,籍贯,住址,和联系人属于不常用数据.如果每次查询都是查询所有数据,不常用数据就会影响效率(因为有时候查询是无用的).

解决方案:将常用的和不常用的信息分离存储,分成两张表

常用信息表

Id(P)	姓名	性别	年龄	体重	身高
1					

不常用信息表: 保证不常用信息与常用信息一定能够对应的上.找到一个具有唯一性(确定记录)的字段共同连接两种表(一般使用到常用表的索引).

Id(P)	婚姻	籍贯	住址	联系人
2				
1				

一对多

简介

一张表中有一条记录可以对应另外一张表中的多条记录,但是返回过来,另外张表的一条记录只能对应第一张表的一条记录.这种关系就是一对多或者多对一.

举例

母亲与孩子的关系:母亲和两个孩子

妈妈表

ID(P)	名字	年龄	性别

孩子表

ID(P)	名字	年龄	性别	妈妈 ID
				妈妈表主键

多对多

简介

一张表(A)中的一条记录能够对应另外一张表(B)中的多条记录,同时B表中的一条记录也能对应A表中的多条记录.这也就是多对多的关系

举例

老师教学:老师和学生

老师表

T_ID(P)	姓名	性别
1	A	男
2	B	女

学生表

S_ID(P)	姓名	性别
1	张三	男
2	小芳	女

中间表

ID	T_ID(老师)	S_ID(学生)
1	1	1
2	1	2
3	2	1
4		

分析

增加中间表之后:中间表与老师表形成了一对多的关系.而且中间表是多表.维护了能够唯一找到一表的关系.

同样的,学生表与中间表也是一对多的关系,一对多的关系可以匹配到关联表之间的数据.

学生找老师:找到学生id→中间表寻找匹配记录(多条)→老师表匹配(一条)

老师找学生:找到老师id→中间表寻找匹配记录(多条)→学生表匹配(一条)

数据库设计

简介

Nonmal Format.是一种离散数学中的只是，是为了解决一种数据的存储与优化的问题。保存数据的存储之后，凡是能够通过关系寻找出来的数据，坚决不再重复存储，终极目标是为了减少数据的冗余。

范式是一种分层结构的规范，分为六层,每一层都比上一层更加严格，若要满足下一层范式，前提满足上一层范式。

六层范式：1NF，2NF，3NF，4NF，5NF，6NF.1NF是最底层，要求最低。6 NF最高层，最严格。

为什么要用范式来设计数据库

MySql属于关系型数据库，有空间浪费，也是致力于节省存储空间：与范式所有解决问题不谋而合。在设计数据库的时候，会利用范式来知道设计。

但是数据库不单是要解决空间问题，同时也要保证效率问题。范式只是解决空间问题，所以数据库的设计又不可能完全按照范式的要求实现。一般情况下满足前三中范式就可以了。

范式在数据库的设计中具有指导意义，但并不是一个强制规范。只是按照范式设计的数据库是一个比较合理的设计。

但如果不考虑磁盘空间,而是更估计效率问题,那么可以不考虑范式来规定设计数据库.而是增加数据冗余提高效率.效率与冗余是一个博弈的问题.要提升效率必然要增加冗余.

高级操作

主键冲突

主键冲突(Duplicate key)

如何解决

> 直接更新数据

insert into 表名[字段列表] values (值列表) on duplicate key update 字段=新值;

操作:先进行插入,但插入不成功,又进行了更新.两步操作

> 先删除数据在进行更新

replace into 表名[字段列表] values (值列表) ;

操作:先判断是否发生主键冲突,如果没有就直接插入(一步为止).如果有就删除旧数据后插入新数据(两步操作).

蠕虫复制

简介

从已有的数据中获取数据,然后将数据有进行新增操作,数据成倍的增加.

操作

> 复制数据表结构

create table 表名 like 数据表名;

```
mysql> create table student01 like student;
Query OK, 0 rows affected (0.23 sec)

mysql> desc student;
+-------+-----------------+------+-----+---------+-------+
| Field | Type            | Null | Key | Default | Extra |
+-------+-----------------+------+-----+---------+-------+
| id    | int(10) unsigned | NO   | PRI | NULL    |       |
| name  | char(1)         | YES  |     | NULL    |       |
+-------+-----------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> desc student01;
+-------+-----------------+------+-----+---------+-------+
| Field | Type            | Null | Key | Default | Extra |
+-------+-----------------+------+-----+---------+-------+
| id    | int(10) unsigned | NO   | PRI | NULL    |       |
| name  | char(1)         | YES  |     | NULL    |       |
+-------+-----------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

> 拷贝数据表数据

insert into 新表名[指定字段名] select *[指定字段名] from 拷贝数据表名 [where条件];

```
mysql> insert into student01 select * from student;
Query OK, 6 rows affected (0.04 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> select * from student;
+----+------+
| id | name |
+----+------+
|  1 | a    |
|  2 | b    |
|  3 | b    |
|  4 | d    |
|  5 | e    |
|  6 | c    |
+----+------+
6 rows in set (0.00 sec)

mysql> select * from student01;
+----+------+
| id | name |
+----+------+
|  1 | a    |
|  2 | b    |
|  3 | b    |
|  4 | d    |
|  5 | e    |
|  6 | c    |
+----+------+
6 rows in set (0.00 sec)
```

意义

1.从已有表拷贝数据到新表中(方便需要已在线项目测试);

2.可以迅速的让表中的数据膨胀到一定的数据级,可测试表的压力以及效率(查询.....);

模糊查询

使用场景:

在查询某一数据时，如果不能确定某一条件，只能模糊确定条件。如在学生表中，我只知道某一学生他的名字姓张。

那么我就可以根据模糊查询把姓张的学生筛选出来。

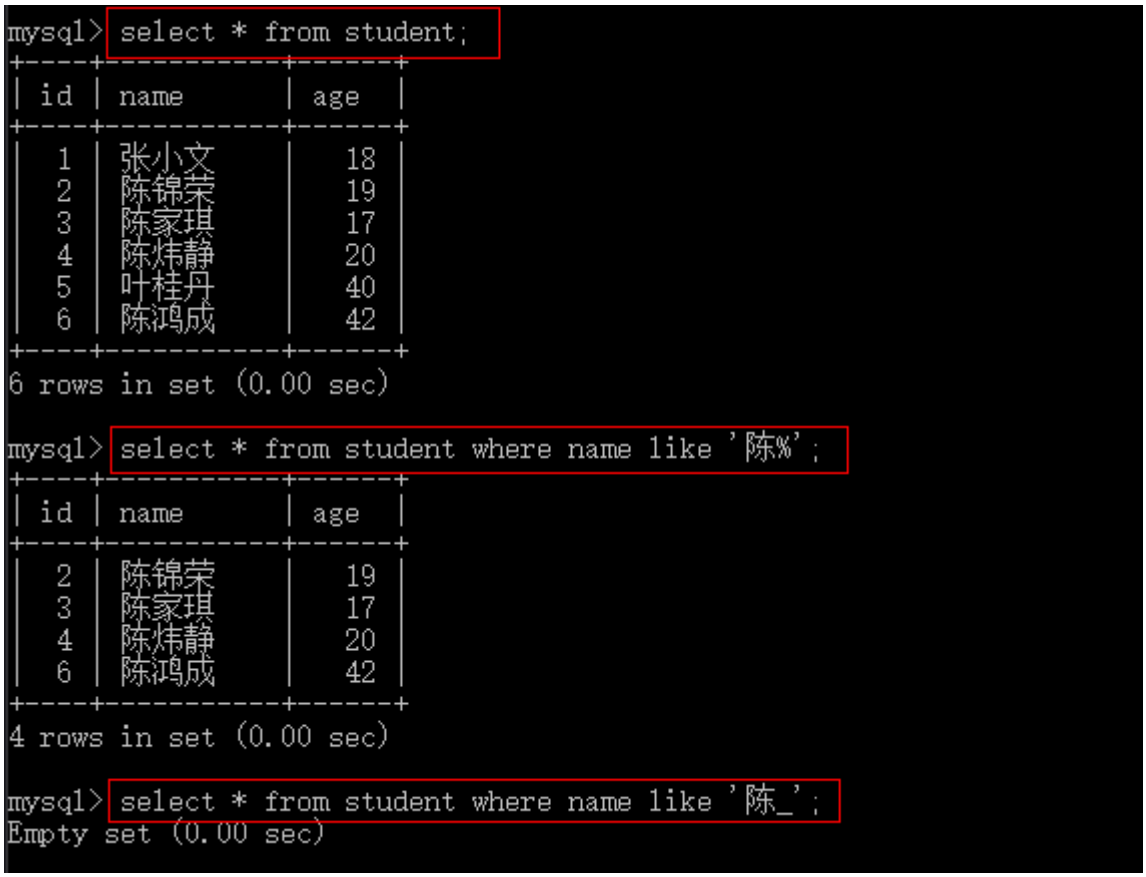
语法

where 条件字段 like '模糊值';

% : 匹配所有字符

_ :匹配一个字符

```
select * from student where name like '陈%';
select * from student where name like '陈_';
```



连接查询

简介

将两张表中的字段记录通过外键或者关联字段一并查询出来

语法

select 表一别名.字段1 , 表二别名.字段1

from

表一as 表一别名

left join

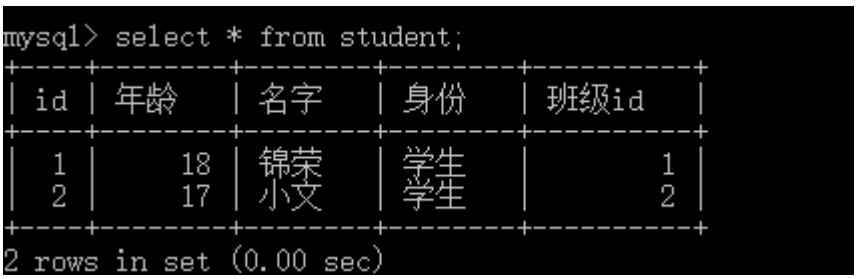
表二 as 表二别名

on

表一外键=表二主键; -- 通过外键关联两表

举例

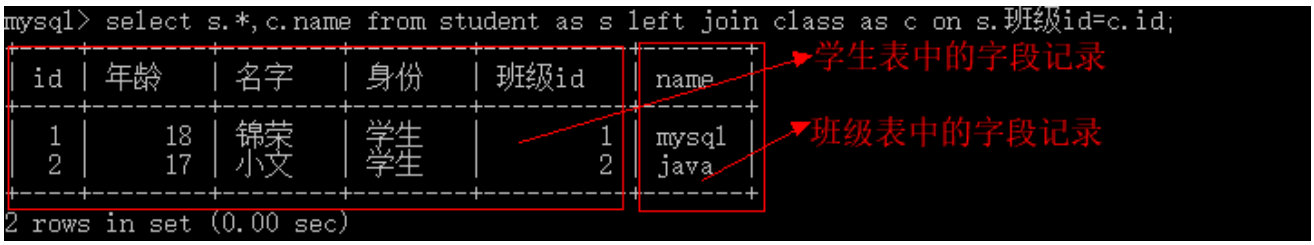
学生表



班级表

连接查询结果

```
select s.*,c.name from student as s left join class as c on s.班级id=c.id;
```



限制记录

limit: 限制查询,修改删除数目;

使用如:select * from 表名 [where条件] limit 限制数目;

如:

只搜索所有只查询三条记录

```
select * from student limit 3;
```

只所有记录中的删除三条记录

```
delete from student limit 3;
```

删除表

高级删除表数据简介

普通删除表数据操作`delete from 表名;`

但有个问题是,如果表中某个字段是自动增长的,那么删除数据仅仅只是删除了数据而已.而不会自动增长那字段的增长变量并不会改变.也就是下次插入自动增长不会从0开始,而是从删除表之前状态的最大值开始.

```
mysql> show create table student;
+---------+-----------------------------------------------
---------------------+
| Table   | Create Table
                     |
+---------+-----------------------------------------------
---------------------+
| student | CREATE TABLE `student` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 |
+---------+-----------------------------------------------
---------------------+
1 row in set (0.00 sec)

mysql> delete from student;
Query OK, 3 rows affected (0.04 sec)

mysql> show create table student;
+---------+-----------------------------------------------
---------------------+
| Table   | Create Table
                     |
+---------+-----------------------------------------------
---------------------+
| student | CREATE TABLE `student` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 |
+---------+-----------------------------------------------
---------------------+
1 row in set (0.00 sec)
```

解决办法

先删除表在创建新表

```
mysql> truncate class;
Query OK, 0 rows affected (0.18 sec)

mysql> show create table class;
+-------+------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------+
| Table | Create Table

      |
+-------+------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------+
| class | CREATE TABLE `class` (
  `id` int(11) NOT NULL,
  `name` varchar(10) DEFAULT NULL,
  `age` int(11) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id` (`id`),
  UNIQUE KEY `age` (`age`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+-------+------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------+
1 row in set (0.00 sec)
```

Select选项

基本语法

```
select  字段列表/* from   表名  [where条件];
```

完整语法

```
select  [select选项]  字段列表[字段别名]  from    数据源(表)  where条件子句   group by 子句
having 子句       order by 子句 limit 子句;
```

字段别名

简介

当数据进行查询出来的时候,有时候名字并不一定就满足需求(多表查询的时候可能会出现同名字段),需要对字段名进行重名名(别名);

```
mysql> select * from student;
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)

mysql> select id as 身份证 from student;
+-----------+
| 身份证    |
+-----------+
|         1 |
+-----------+
1 row in set (0.00 sec)
```

数据源

一条数据源

select * from 表名;

多条数据源

select * from 表名,表名;

从一张表中取出一条记录,去另外一张表中匹配所有记录,而且全部保留(记录数和字段数),将这种结果称为:笛卡儿积(交叉连接)

```
mysql> select * from student;
+--------+--------+--------+
| 年龄   | 名字   | 身份   |
+--------+--------+--------+
|     18 | 小红   | 学生   |
|     18 | 小黄   | 学生   |
|     18 | 小蓝   | 学生   |
|     18 | 小绿   | 学生   |
|     18 | 小青   | 学生   |
|     18 | 小紫   | 学生   |
+--------+--------+--------+
6 rows in set (0.00 sec)

mysql> select * from teacher;
+--------+---------+--------+
| 年龄   | 名字    | 身份   |
+--------+---------+--------+
|     48 | 老师A   | 老师   |
|     48 | 老师B   | 老师   |
|     48 | 老师C   | 老师   |
|     48 | 老师D   | 老师   |
|     48 | 老师E   | 老师   |
|     48 | 老师F   | 老师   |
+--------+---------+--------+
6 rows in set (0.00 sec)
```

```
mysql> select * from student,teacher;
+--------+--------+--------+--------+---------+--------+
| 年龄   | 名字   | 身份   | 年龄   | 名字    | 身份   |
+--------+--------+--------+--------+---------+--------+
|     18 | 小红   | 学生   |     48 | 老师A   | 老师   |
|     18 | 小黄   | 学生   |     48 | 老师A   | 老师   |
|     18 | 小蓝   | 学生   |     48 | 老师A   | 老师   |
|     18 | 小绿   | 学生   |     48 | 老师A   | 老师   |
|     18 | 小青   | 学生   |     48 | 老师A   | 老师   |
|     18 | 小紫   | 学生   |     48 | 老师A   | 老师   |
|     18 | 小红   | 学生   |     48 | 老师B   | 老师   |
|     18 | 小黄   | 学生   |     48 | 老师B   | 老师   |
|     18 | 小蓝   | 学生   |     48 | 老师B   | 老师   |
|     18 | 小绿   | 学生   |     48 | 老师B   | 老师   |
|     18 | 小青   | 学生   |     48 | 老师B   | 老师   |
|     18 | 小紫   | 学生   |     48 | 老师B   | 老师   |
|     18 | 小红   | 学生   |     48 | 老师C   | 老师   |
|     18 | 小黄   | 学生   |     48 | 老师C   | 老师   |
|     18 | 小蓝   | 学生   |     48 | 老师C   | 老师   |
|     18 | 小绿   | 学生   |     48 | 老师C   | 老师   |
|     18 | 小青   | 学生   |     48 | 老师C   | 老师   |
|     18 | 小紫   | 学生   |     48 | 老师C   | 老师   |
|     18 | 小红   | 学生   |     48 | 老师D   | 老师   |
|     18 | 小黄   | 学生   |     48 | 老师D   | 老师   |
|     18 | 小蓝   | 学生   |     48 | 老师D   | 老师   |
|     18 | 小绿   | 学生   |     48 | 老师D   | 老师   |
|     18 | 小青   | 学生   |     48 | 老师D   | 老师   |
|     18 | 小紫   | 学生   |     48 | 老师D   | 老师   |
|     18 | 小红   | 学生   |     48 | 老师E   | 老师   |
|     18 | 小黄   | 学生   |     48 | 老师E   | 老师   |
|     18 | 小蓝   | 学生   |     48 | 老师E   | 老师   |
|     18 | 小绿   | 学生   |     48 | 老师E   | 老师   |
|     18 | 小青   | 学生   |     48 | 老师E   | 老师   |
|     18 | 小紫   | 学生   |     48 | 老师E   | 老师   |
|     18 | 小红   | 学生   |     48 | 老师F   | 老师   |
|     18 | 小黄   | 学生   |     48 | 老师F   | 老师   |
|     18 | 小蓝   | 学生   |     48 | 老师F   | 老师   |
|     18 | 小绿   | 学生   |     48 | 老师F   | 老师   |
|     18 | 小青   | 学生   |     48 | 老师F   | 老师   |
|     18 | 小紫   | 学生   |     48 | 老师F   | 老师   |
+--------+--------+--------+--------+---------+--------+
36 rows in set (0.00 sec)
```

子查询:数据源是一条查询语句(查询语句的结构时二维表);

selec * from (select语句) as 别名;

注意!!! 必须在数据源加别名

不然会:

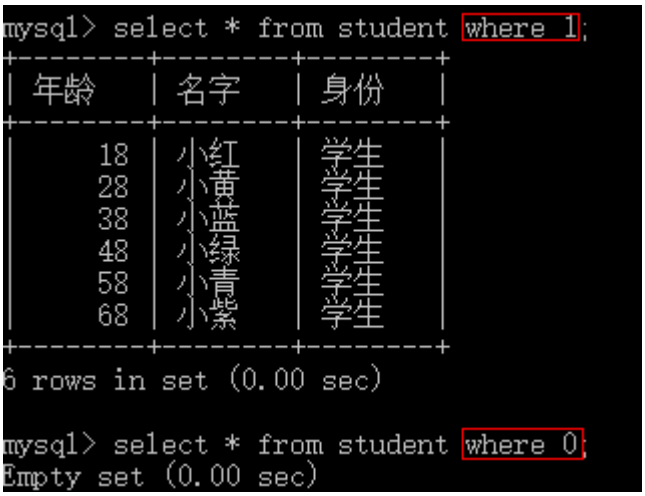
```
mysql> select 年龄 from (select * from teacher);
ERROR 1248 (42000): Every derived table must have its own alias
```

where子句

简介

用来判断数据,筛选数据

where子句返回结果:0或者1.0代表false;1代表true(在C中非0即为真);



判断条件

> 比较运算符:> ; < ; >= ; <= ; !=(<>) ; like ; between 区间1 and 区间2 ; in/not in

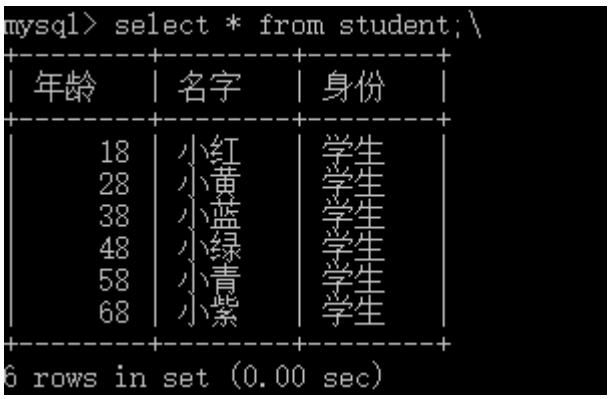
> 逻辑运算符:&&(and) ; ||(or) ; !(not);

原理

where是唯一一个直接从磁盘获取数据的时候就开始判断条件,从磁盘取出一条记录.开始进行where判断.判断的结果如果成立就保存到内存,失败直接放弃.

使用

数据表



找到学生年龄为18,28,38的学生;

使用子句: =加|| 或者 in

```
mysql> select * from student where 年龄=18||年龄=28||年龄=38;
+--------+--------+--------+
| 年龄   | 名字   | 身份   |
+--------+--------+--------+
|     18 | 小红   | 学生   |
|     28 | 小黄   | 学生   |
|     38 | 小蓝   | 学生   |
+--------+--------+--------+
3 rows in set (0.02 sec)

mysql> select * from student where 年龄 in(18,28,38);
+--------+--------+--------+
| 年龄   | 名字   | 身份   |
+--------+--------+--------+
|     18 | 小红   | 学生   |
|     28 | 小黄   | 学生   |
|     38 | 小蓝   | 学生   |
+--------+--------+--------+
3 rows in set (0.02 sec)
```

找年龄在18-38的学生

使用子句: >=加<=&& 或者 beteen 区间1 and 区间2

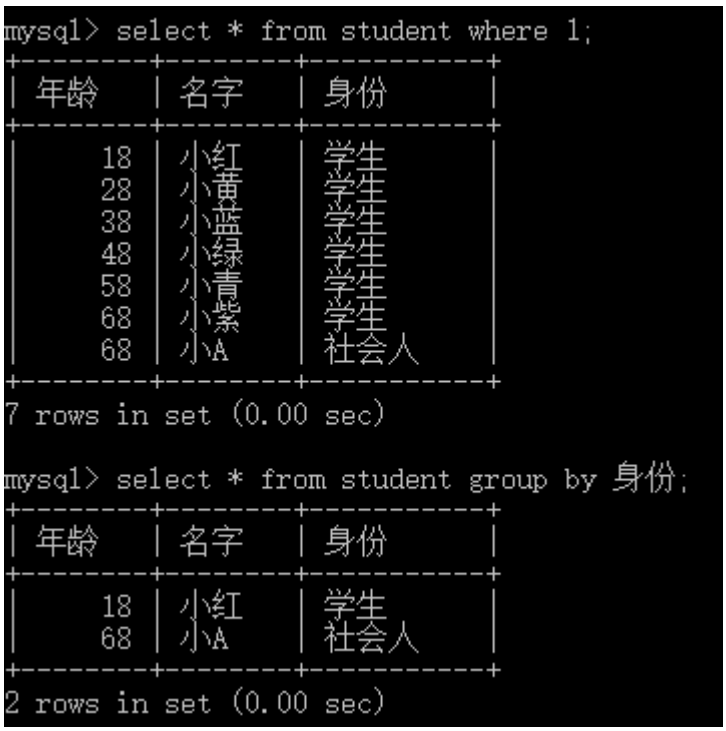
```
mysql> select * from student where 年龄>=18&&年龄<=38;
+--------+--------+--------+
| 年龄   | 名字   | 身份   |
+--------+--------+--------+
|     18 | 小红   | 学生   |
|     28 | 小黄   | 学生   |
|     38 | 小蓝   | 学生   |
+--------+--------+--------+
3 rows in set (0.00 sec)

mysql> select * from student where 年龄 between 18 and 38;
+--------+--------+--------+
| 年龄   | 名字   | 身份   |
+--------+--------+--------+
|     18 | 小红   | 学生   |
|     28 | 小黄   | 学生   |
|     38 | 小蓝   | 学生   |
+--------+--------+--------+
3 rows in set (0.00 sec)
```

group by 子句

简介

分组的意思,根据某个字段进行分组(相同的放一组,不同的放到一组)



使用

一般如果单单使用group by没什么卵用

group by 是进行分组统计的使用的,一般需要配合sql统计函数配合使用;

sql统计函数

- count(统计字段):统计分组后的记录数,每一组有多少记录.如果是count(*)则是统计用多少字段数.
- max(统计字段):统计每组中的最大值.
- min(统计字段):统计每组中的最小值.
- avg(统计字段):统计每组中的平均值.
- sum(统计字段):统计每组中的总和.

基本语法

..... 数据源 group by 分组字段[,分组字段];

单字段分组

举例:根据性别分组后,统计不同性别的字段

```
mysql> select * from teacher;
+----+--------+--------+------+------+--------+
| id | number | name   | sex  | age  | height |
+----+--------+--------+------+------+--------+
|  1 |      1 | 张三   | 男   |   35 |    185 |
|  2 |      2 | 李四   | 男   |   35 |    172 |
|  3 |      3 | 赵六   | 女   |   25 |    188 |
|  4 |      4 | 二五   | 女   |   15 |    168 |
|  5 |      5 | 小文   | 女   |   20 |    168 |
+----+--------+--------+------+------+--------+
5 rows in set (0.00 sec)

mysql> select sex,count(*),max(height),min(height),sum(age),avg(age) from teacher group by sex;
+------+----------+-------------+-------------+----------+----------+
| sex  | count(*) | max(height) | min(height) | sum(age) | avg(age) |
+------+----------+-------------+-------------+----------+----------+
| 女   |        3 |         188 |         168 |       60 |  20.0000 |
| 男   |        2 |         185 |         172 |       70 |  35.0000 |
+------+----------+-------------+-------------+----------+----------+
2 rows in set (0.00 sec)
```

多字段分组

先根据一个字段进行分组,然后对分组后的结果再次按照其他字段进行分组

```
mysql> select * from teacher;
+----+--------+--------+------+------+--------+----------+
| id | number | name   | sex  | age  | height | position |
+----+--------+--------+------+------+--------+----------+
|  1 |      1 | 张三   | 男   |   35 |    185 | 码农     |
|  2 |      2 | 李四   | 男   |   35 |    172 | 经理     |
|  3 |      3 | 赵六   | 女   |   25 |    188 | 码农     |
|  4 |      4 | 二五   | 女   |   15 |    168 | 经理     |
|  5 |      5 | 小文   | 女   |   20 |    168 | 码农     |
+----+--------+--------+------+------+--------+----------+
5 rows in set (0.00 sec)

mysql> select sex,position,count(*) from teacher group by sex,position;
+------+----------+----------+
| sex  | position | count(*) |
+------+----------+----------+
| 女   | 码农     |        2 |
| 女   | 经理     |        1 |
| 男   | 码农     |        1 |
| 男   | 经理     |        1 |
+------+----------+----------+
4 rows in set (0.00 sec)
```

先按照性别分组后，在分组后的基础上在按职位分。最后统计字段数

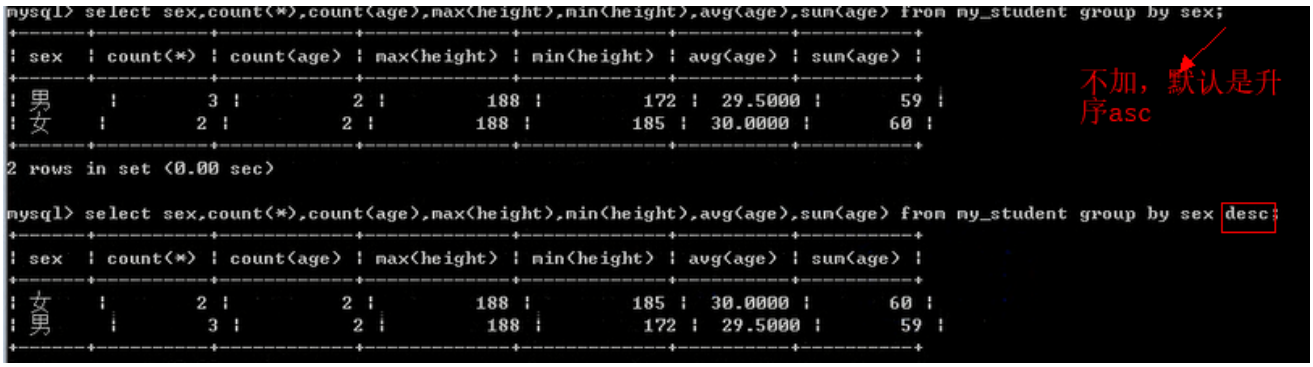
分组后排序

group by(分组)会自动排序,根据分组字段,默认是升序(校对集不同,排序规则不同,结果不同);

group by 字段 [asc|desc] ;对分组的结果合并之后的整个结果排序

asc :升序(默认)

desc:降序.



having子句

简介

Having 子句用于对group by 分组设置约束条件，用法与where子句基本相同，不同是where子句作用于基表视图，以便从中选择满足条件的记录；having子句则作用于分组，用于选择满足条件的组，其条件表达式中通常会使用聚合函数。

与where子句一样,进行条件判断

where是针对磁盘数据进行判断.而having是针对内存数据进行处理.

having 的优势和劣势

having 能做where能做的几乎所有事情,但是where却不能做having的很多事情.

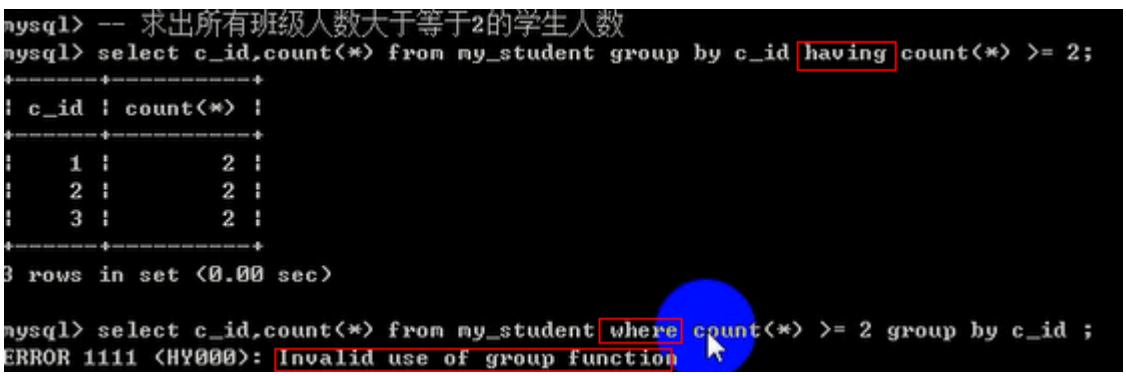
但是。因为having是针对内存数据处理的,也就是说所有数据进入了内存后在进行筛选判.所以也就是导致内存巨大的消耗.

where的优势

where是针对磁盘数据进行处理的,在筛选处理之后这些数据在进入内存.那么where相对having不耗费内存资源.

使用场景

- 数据进入到内存之后,会进行分组操作.分组操作后需要筛选就只能用having来处理了.



- 使用字段别名进行条件处理

  having能够使用字段别名,where不能.因为where是从磁盘取出的数据进行处理,而给字段进行取别名是数据进入到内存之后才有的.

```
mysql> select c_id,count(*) as total from my_student group by c_id having total >= 2;
+------+-------+
| c_id | total |
+------+-------+
|    1 |     2 |
|    2 |     2 |
|    3 |     2 |
+------+-------+
3 rows in set (0.00 sec)

mysql> select c_id,count(*) as total from my_student where total >= 2 group by c_id ;
ERROR 1054 (42S22): Unknown column 'total' in 'where clause'
```

order by 子句

简介

排序,根据某个字段进行升序或者降序排序,以来校对集

使用基本语法

order by 字段名 [asc|desc]

asc : 升序(默认)

desc : 降序

单字段排序

order by 字段名 [asc|desc] ;

```
mysql> select * from my_student group by c_id;
+----+-------------+------+------+------+--------+------+
| id | number      | name | sex  | age  | height | c_id |
+----+-------------+------+------+------+--------+------+
|  1 | itcast0001  | 张三 | 女   |   35 |    185 |    1 |
|  4 | itcast0004  | 赵六 | 男   | NULL |    188 |    2 |
|  2 | itcast0002  | 李四 | 男   |   29 |    172 |    3 |
+----+-------------+------+------+------+--------+------+
```

多字段排序

order by 字段名1 字段名2 [asc|desc] ;

先根据某个字段进行排序,然后排序后的内部,在按照某个数据在进行再次排序

```
mysql> select * from my_student order by c_id;
+----+-------------+------+------+------+--------+------+
| id | number      | name | sex  | age  | height | c_id |
+----+-------------+------+------+------+--------+------+
|  1 | itcast0001  | 张三 | 女   |   35 |    185 |    1 |
|  3 | itcast0003  | 王五 | 女   |   25 |    188 |    1 |
|  4 | itcast0004  | 赵六 | 男   | NULL |    188 |    2 |
|  6 | itcast0006  | 小芳 | 女   |   18 |    160 |    2 |
|  2 | itcast0002  | 李四 | 男   |   29 |    172 |    3 |
|  5 | itcast0005  | 小明 | 男   |   30 |    185 |    3 |
+----+-------------+------+------+------+--------+------+
6 rows in set (0.00 sec)
```

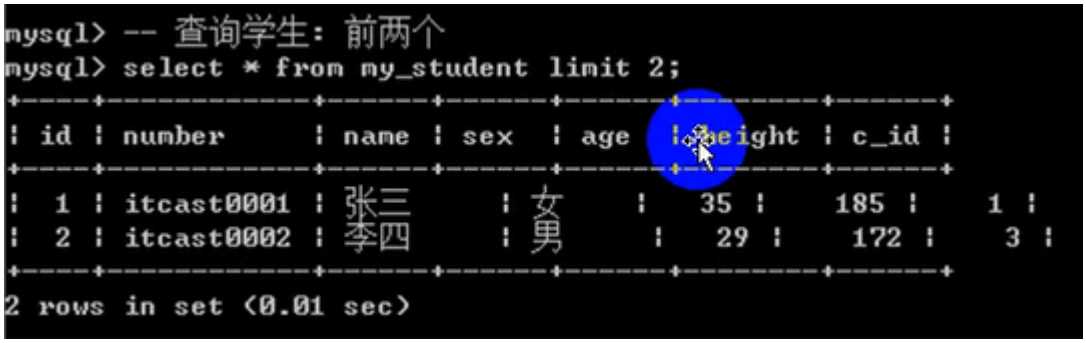
limit子句

简介

是一种限制结果的语句:限制数量

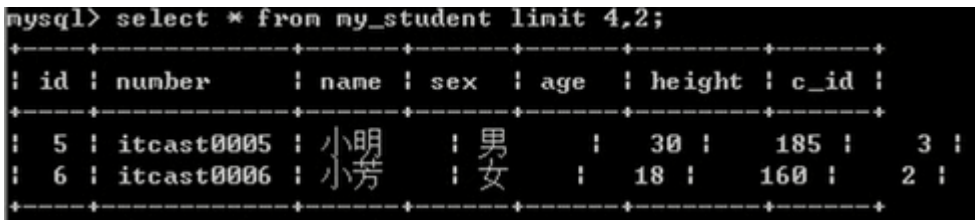
使用

- 只用来限制长度(数据量) : limit 数据量



- 限制起始位置,限制数量 : limit 起始位置,长度

意义:可用来实现数据分页展示效果



五大子句的执行顺序

where：从磁盘中处理数据

group by： 在内存中处理数据

having： 在内存中处理数据

order by： 在内存中处理数据

limit： 在内存中处理数据

外键

简介

foreign key(外键),外面的键(键不在自己表中),如果一张表中有一个字段(非主键)指向另外一张表的主键,那么该字段称之为外键.

场景

在表关系中,一对多或者多对一的时就是使用到外键的情况.

外键约束

外键默认的作用有两点:一个对父表,一个对子表(外键字段所在的表)

- 对子表约束:对字表数据进行写操作(增和改)的时候,如果对应的外键字段在父表找不到对应的匹配.那么操作会失败(约束子表数据的操作).
- 对父表约束:父表数据进行写操作(删和改,涉及到主键数据本身),如果对应的主键在子表中已经被引用,那么就不允许操作.

约束条件

简介

所谓外键约束,就是值对外键的左右,上面所讲的外键作用是默认的,其实可以通过对外键的需求,进行定制操作

约束模式

- **district:** 严格模式(默认的),父表不能删除或者更新一个已经被子表数据引用的记录
- **cascade:**级联模式,父表的操作,对应子表关联的数据(外键字段)也跟着变化
- **set null :**置空模式:父表的操作之后,子表对应的数据(外键字段)被置空

指定外键约束模式语法

foreign key(外键字段) references 父表(主键字段) on delete 约束模式 on update 约束模式;

```
alter table student add foreign key(班级id) references class(id) on update cascade on
delete set null;
```

添加外键条件

- 外键要存在,首先必须保证表的存储引擎是innodb(默认的存储引擎),如果不是innodb存储引擎,那么外键可以创建成功,但是没有约束效果.
- 外键字段的字段类型必须和父表的主键类型一致
- 外键表的外键名不能冲突
- 增加外键的字段(已经存在数据),必须保证外键字段数据与父表主键对应.

增加外键

- 在创建表的时候或者创建表之后增加

语法:

```
create table 表名(
                字段名             数据类型,

       字段名             数据类型,

       [constraint    外键名]    foreign key(外键字段)  references  外部表(外部
表主键字段)

       );
```

```
  create table  student (

  id int(11),

  年龄 int(11) ,

  名字 varchar(10) ,

  身份 varchar(10) ,

  班级id  int ,

  PRIMARY KEY (id),

  foreign key(班级id) references class(id)

  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- 在创建表结构之后,添加外键

语法: alter table 表名 add [constraint 外键名] foreign key(外键字段名) references 外部表名(外部表主键字段名);

```
alter table student add constraint 班级id外键 foreign key(班级id) references class(id);
```

注意

添加外键要求字段本身必须是一个索引(普通索引),如果字段本身没有索引,添加外键时会自动创建一个索引,然后才会创建外键本身.

这也就是为什么外键字段key属性时mul的原因,因为他有两个索引(普通索引和外键索引).

```
mysql> desc student;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| id       | int(11)     | NO   | PRI | NULL    |       |
| 年龄     | int(11)     | YES  |     | NULL    |       |
| 名字     | varchar(10) | YES  |     | NULL    |       |
| 身份     | varchar(10) | YES  |     | NULL    |       |
| 班级id   | int(11)     | YES  | MUL | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

删除外键

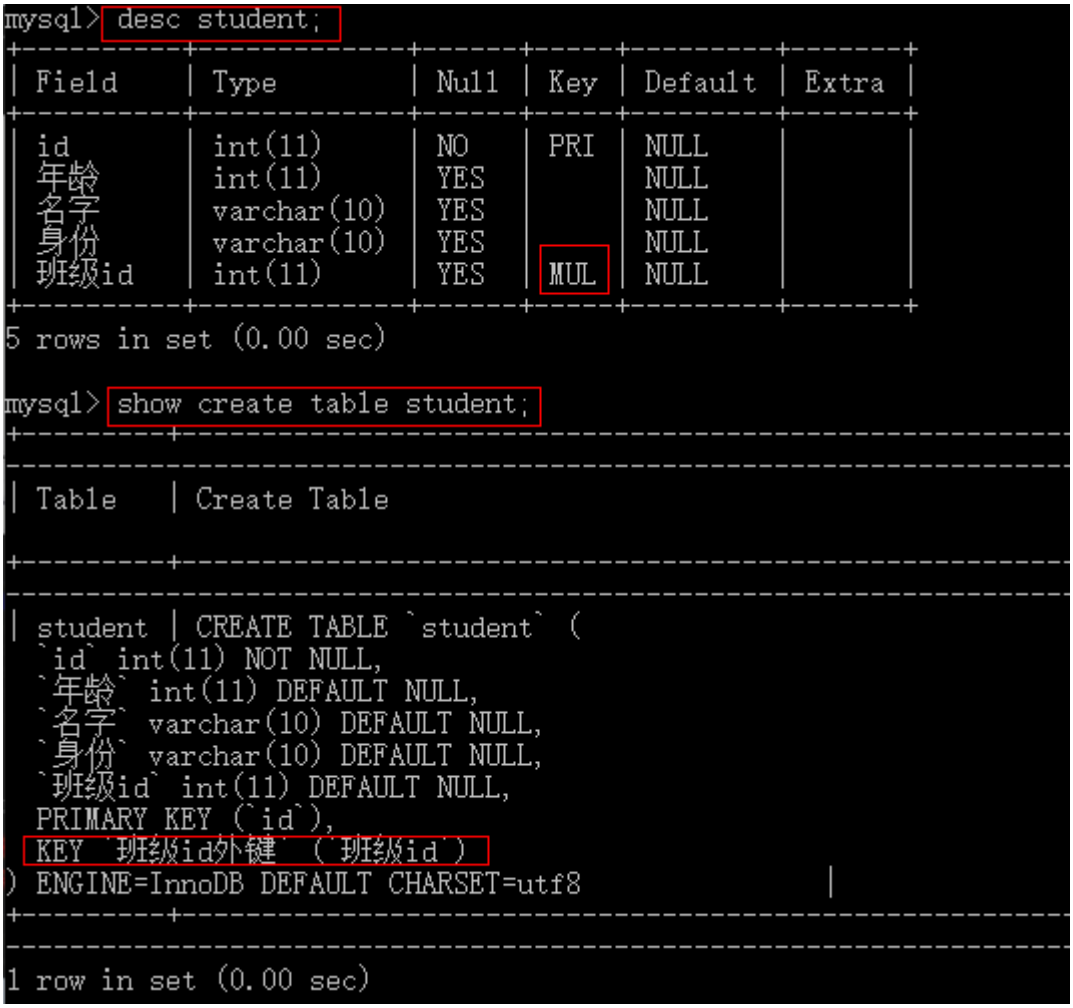
语法

alter table 表名 drop foreign key 外键名;

注意

删除外键不能在查看表结构上看出区别

只能在查看表创建语句看出区别



联合查询

简介

将多次查询(多条select查询语句),在记录上进行拼接(字段数不会增加).

意义

- 查询同一张表,但是需求不同,如查询学生信息,男的按照身高升序,女的升高降序
- 多表查询,多张表的结构完全一样,保存的数据(结构)也是一样.

如:多表统计,在数据量非常大的数据表中,如在QQ表中,每次登陆都要检索匹配QQ表的数据,那么效率就会非常慢.这时就需要进行分表.如可以按照QQ号尾号的单双数,分两张表.那么在查询的时候就将数据检索量减少了一半,那么效率也进而提高了一倍.

最后在进行数据统计时就可用到联合查询.

语法

select 语句1

union [union选项]

select 语句2

union选项:与select选项一样

- all:保留所有(不管重复)
- distinct:去重(整个重复),默认;

```
    (select * from teacher where sex='男' order by height desc limit 99999)
    union [union选项]
    (select * from teacher where sex='女' order by height  asc limit 99999);
```

注意

联和查询如果使用到order by排序的话,select 语句需要用'()'括起来,并在加上limit 限制(一般超过数据量就行).

子查询

简介

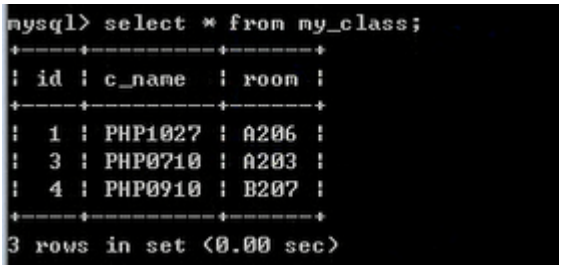
查询是在某个查询结果之上进行的。一条select语句内部包含另外一条select语句。

举例表结构

一张学生表（my_student）

```
mysql> select * from my_student;
+----+-------------+------+------+------+--------+------+
| id | number      | name | sex  | age  | height | c_id |
+----+-------------+------+------+------+--------+------+
|  1 | itcast0001 | 张三 | 女   |   35 |    185 |    1 |
|  2 | itcast0002 | 李四 | 男   |   32 |    172 |    3 |
|  3 | itcast0003 | 王五 | 女   |   34 |    188 |    1 |
|  4 | itcast0004 | 赵六 | 男   |   35 |    188 |    2 |
|  5 | itcast0005 | 小明 | 男   |   34 |    185 | NULL |
|  6 | itcast0006 | 小芳 | 女   |   27 |    160 |    2 |
|  7 | itcast0007 | 周杨 | 男   |   33 |    180 |    1 |
+----+-------------+------+------+------+--------+------+
7 rows in set (0.00 sec)
```

一张班级表（my_class）



分类

标量子查询

子查询得到的结果是一行一列

使用场景：

where语句的一个条件值是一个子查询语句得到的一行一列(一个值)记录数据。

举例：

知道班级名为PHP0710，想获取改班级的所有学生。

查询步骤：

1. 确定数据源，获取所有的学生

   `select * from my_student c_id=?; -- 其中c_id的值就是根据班级名从班级表子查询到班级得到值。`

2. 子查询获取班级ID，可通过班级名字查询得到

   `select id from my_class where c_name_'PHP0710'; -- 一定是只有一个值（一行一列）。`

完整语句：

```
select * from my_student where c_id=(select c_id from my_class where c_name='PHP0710');
```

列子查询

子查询得到的结果是一列多行

使用场景：

where语句的一个条件值是一个子查询语句得到的一行多列记录数据。也就是能匹配多值（类似in）。

举例：

查询所在在读班级的学生（班级表中存在的班级的学生）

查询步骤：

1. 确定数据源（学生）

   `select * from my_student where c_id in (?); -- 其中? 就是子查询得到的数据结果（多列一行），也就是说该学生表班级id能在班级表中查询得到班级id匹配到`

2. 确定有效班级id,所有班级id

   `select id from my_class;`

完整语句:

```
select * from student where c_id in (select c_id from c_class);
```

行子查询

子查询得到的结果是多列一行

使用场景:

标量字查询和列子查询，其中where条件只有一个，也就是说数据记录列只有一列。那么 where条件有一个以上的话，就需要用到行子查询.

举例:

查询在整个学生表中，年龄最大且身高最高的学生.

步骤:

1. 确定数据源（学生）

   `select * from my_student where age=? and height=? -- 其中的? 就是子查询的得到的条件值（多列一行）;`

2. 确定最大的年龄和最大的身高

   `select max(age),max(height) from student ;`

完整语句:

```
select * from my_student
where
(age,height)
=
(select max(age),max(height) from student );
```

表子查询

子查询得到的结果是多行多列

使用场景:

对子查询出来的完整数据重新进行查询分析统计

举例:

找出每个班最高的学生

1. 确定数据源（学生）

`select * from ? group by c_id; -- ? 就是子查询的得到的完整数据`

2. 对每个班的学生按照身高进行降序排序.

   `select * from my_student order by height desc;`

3. 对排序后的每个班的学生进行按照班级分类;

完整语句:

```
select * from (select * from my_student order by height desc) as student group by c_id;
```

注意

子查询语句需要‘（）’括起来

行子查询条件字段也要用‘（）’括起来

表子查询中查询的得到的数据源需要取区别名,因为 from跟着的是表名

视图

简介

是一种有结构（有行有列）但是结构中不存放真实数据的虚拟表，虚拟表的结构源不是自己定义的，而是从对应的基表（普通表）产生（视图的数据来源）

意义

1.视图可以节省SQL语句，可将一条复杂的查询语句使用视图进行保存，以后就可以直接对视图进行操作。

2.数据安全，视图操作主要是针对查询，如果对视图结构进行处理（删除），不会影响基表数据。

3.视图往往在大项目使用，根据业务需求，对外提供本项目有用的数据，隐藏无关的数据。保证数据安全。

如:在现如今的微信二次开发中,微信平台不会直接提供数据表让第三方进行操作,而是创建一个视图,提供第三方需要的数据.然后第三方开发者在该视图上进行操作(读取数据).

创建视图

语法

create view 视图名 as select 语句;

select 语句可以是普通查询,可以是连接查询;可以是联合 查询,可以是子查询.

- 单表视图(数据源只有一张表)

```
create view student_view as select id,年龄,名字,班级id from student ;
```

```
mysql> desc student;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| id       | int(11)     | NO   | PRI | NULL    |       |
| 年龄     | int(11)     | YES  |     | NULL    |       |
| 名字     | varchar(10) | YES  |     | NULL    |       |
| 身份     | varchar(10) | YES  |     | NULL    |       |
| 班级id   | int(11)     | YES  | MUL | NULL    |       |
+----------+-------------+------+-----+---------+-------+
5 rows in set (0.07 sec)

mysql> create view student_view as select id,年龄,名字,班级id from student ;
Query OK, 0 rows affected (0.04 sec)

mysql> desc student_view;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| id       | int(11)     | NO   |     | NULL    |       |
| 年龄     | int(11)     | YES  |     | NULL    |       |
| 名字     | varchar(10) | YES  |     | NULL    |       |
| 班级id   | int(11)     | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```


- 多表视图(数据源有一张表以上)

如通过连接查询,同时获取两张表数据

create view 视图名

as

select 表一别名.字段名,表二别名.字段名

from 表一 as 表一别名

--连接表二

left join

表二 as 表二别名

on

表一别名.外键=表二别名.主键; ** --通过外键连接两表**

```
create view 多表视图 as
select 学生表别名.*,班级表别名.name from student as 学生表别名
left  join
class as 班级表别名
on 学生表别名.班级id=班级表别名.id;
```

```
mysql> create view 多表视图 as
    -> select 学生表别名.*,班级表别名.name from student as 学生表别名
    -> left  join
    -> class as 班级表别名
    -> on 学生表别名.班级id=班级表别名.id;
Query OK, 0 rows affected (0.04 sec)

mysql> desc 多表视图;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| id       | int(11)     | NO   |     | NULL    |       |
| 年龄     | int(11)     | YES  |     | NULL    |       |
| 名字     | varchar(10) | YES  |     | NULL    |       |
| 身份     | varchar(10) | YES  |     | NULL    |       |
| 班级id   | int(11)     | YES  |     | NULL    |       |
| name     | varchar(10) | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

查看视图

查看视图与查看表结构并无区别

- desc 视图名
- show create table 视图名
- show create view 视图名

删除视图

删除视图与删除表并无区别

drop view 视图名;

删除视图并不会影响基表结构,这也是视图意义所在.

修改视图

修改视图显示字段,即是重新确定数据源

alter view 修改视图名 as select语句;

视图操作

注意

对视图的操作都会作用到基表,所以往往对视图的权限仅仅只限与查询

添加数据

往视图添加数据,该数据会被添加到基表中

- 往多表视图添加数据:**不允许**
- 往单表视图添加数据:insert普通表一样

删除数据

往视图删除数据,该数据会从基表中删除

- 往多表视图删除数据:**不允许**
- 往单表视图删除数据:delete普通表一样

更新数据

往多表视图更新数据:**不允许**

往视图更新数据,基本中的该字段的数据也会被更新

更新视图与跟新普通表一样

但更新视图只能更新视图拥有的数据记录

update 视图名 set 字段名=值;

查询数据

查询视图与查询表并无区别,按照普通查询语法查询即可.

视图算法

简介

系统对视图外部查询视图的select语句的一种解析方式

分类

undefined:未定义(默认),这不是一种实际的使用算法,而是让系统自己选择一下算法.

temptable:临时表算法:系统先执行视图的select语句,后执行外部查询的语句.

merge:合并算法(一般如果视图未定义算法的话,系统会优先使用该算法,因为该算法效率高):系统将视图的select语句和外部的查询视图的select语句行合并,然后在执行.

使用场景

如果视图的select语句中出现的五子句(where,group by,having,order by,limit),比外部的查询语句要靠后,建议使用temptable(临时表算法),其他情况默认使用merge(合并算法)

举例

查询每个班级中最高的学生

学生表（my_student）

步骤

1.创建视图将每个班级进行降序排序

```
create view order_student as select * from my_student order by height desc;
```

2.对排序后的视图进行分组

```
select * from order_student group by c_id;
```

但由于系统优先使用算法时merge,导致查询出来的实际sql语句是

```
select * from student group by c_id order by height desc;
```

先进行分组,在进行排序。

总结

步骤一正确语句

```
create algorithm=temptable view order_student as select * from my_student order by height
desc;
```

分类

语法

在创建视图时指定算法

create **algorithm=算法名([temptable,merge])** view 视图名 as select 语句;

```
 create algorithm=temptable  view sv as select * from student where id <3;
```

备份与还原

简介

备份：将当前已有的数据或者记录保存

还原：将已经保留的数据恢复到对应的表中

为什么要做备份还原

保存数据丢失，被盗和误操作

备份方式

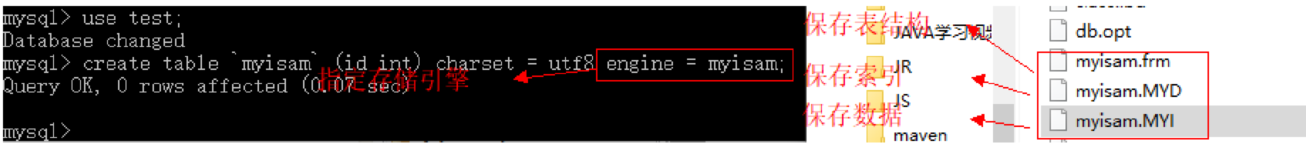
数据表备份还原

备份：

不需要通过任何命令，直接保存数据文件。

但数据表保存条件前提是：数据的存储引擎是myisam

myisam存储引擎存储数据的特点是:它会将数据表结构，数据，索引单独分开保存成不同文件。



frm： 保存结构

MYD： 保存数据

MYI： 保存索引

还原：

直接将三个数据文件放到创建数据库文件目录下即可

单表备份还原

只能保存数据，不保存表结构

备份

select */字段列表 into outfile '文件保存路径' from 数据源

还原

load data infile '文件保存路径' into table 表名[(字段列表)]

SQL备份还原(一般用的比较多)

可备份表结构和表数据,可整个数据库都备份或单独备份表

备份:

通过mysqldump.exe客户端进行备份

mysqldump -h主机名 -P端口号 -u用户名 -p密码 数据库名 [备份表名1，备份表名2] > '备份文件路径';

```
mysqldump  -hlocalhost  -P3306 -uroot -p123456 test student >
C:/Users/Administrator/Desktop/student.sql
```

如不指定表名，即是备份整个数据库 **还原:**

```
1.mysql客户端还原
```

```
mysql -h主机名 -P端口号 -u用户名 -p密码 数据库名 < '备份路径';
```

```
 2.sql指令还原
```

```
source '备份路径';
```

增量备份还原

一般用于大项目上，因为数据量变化大，为了数据安全，所以备份周期比较短。

不是针对数据或者SQL指令进行备份，是针对mysql服务器的日志文件（对mysql系统进行sql指令操作都会被记录到日志文件内）进行备份；

增量备份：指定时间段开始进行备份，备份数据不会重复，而且所有的操作都会备份。

事务

简介

在一系列连续的操作中要保证全部操作同时都被实现的机制。

使用条件

目前只有innodb存储引擎和BDB存储引擎才支持事务机制。

而BDB是收费的，使用比较多的是innodb。

使用场景：

有一张银行帐户表，有A用户给B用户转账，A账户先减少，B账户在增加，但是A操作完后由于特殊原因导致下一步操作终止了。但是A账户已经减少了，而B账户上又没有增加。那么这个时候就出现了事物机制。

不考虑事务的隔离性会出现的问题

- 脏读

  脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

  当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

  ```
  update account set money=money+100 where name='B';  (此时A通知B)

  update account set money=money - 100 where name='A';
  ```

  当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

- 不可重复读

  不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

  例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

  不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

  在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据A和B依次查询就可能不同，A和B就可能打起来了……

- 虚度（幻读）

  幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

  幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

事务特性：ACID

- A（atomic）原子性

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

- C（consistency）一致性

  一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

- I（Isolation）隔离性

  隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

- D（durability）持久性

  持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

四种隔离级别

- Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。
- Repeatable read (可重复读)：可避免脏读、不可重复读的发生。
- Read committed (读已提交)：可避免脏读的发生。
- Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

  以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。

在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别

> 查看数据库当前事务的隔离级别

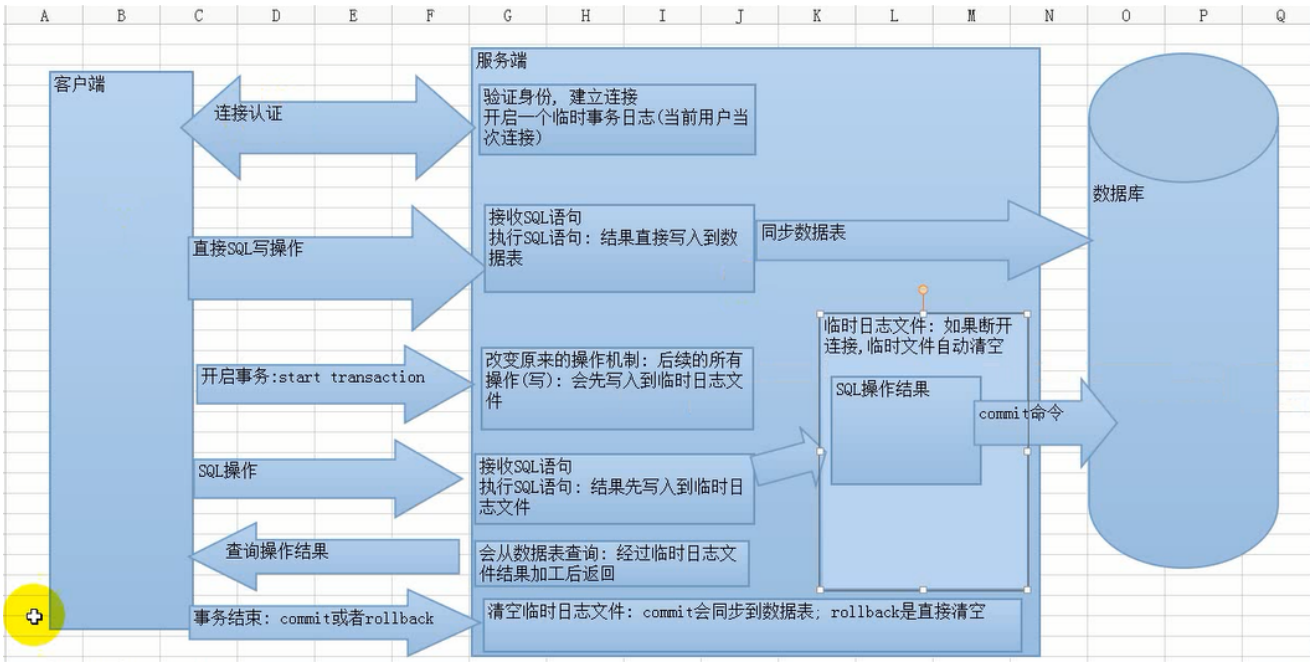
```
select @@tx_isolation;
```

> 设置事务隔离级别

```
set  [glogal | session]  transaction isolation level 隔离级别名称;

set tx_isolation='隔离级别名称;'
```

原理



事务操作

自动事务（默认的）

在正常情况下，如果对一张表进行操作（写）的话，一般MySql系统会自动帮我们提交事务同步到表中。所以在一般情况下，在进行写操作之后都用进行事务的提交（commit），因为系统会自动帮我们提交。

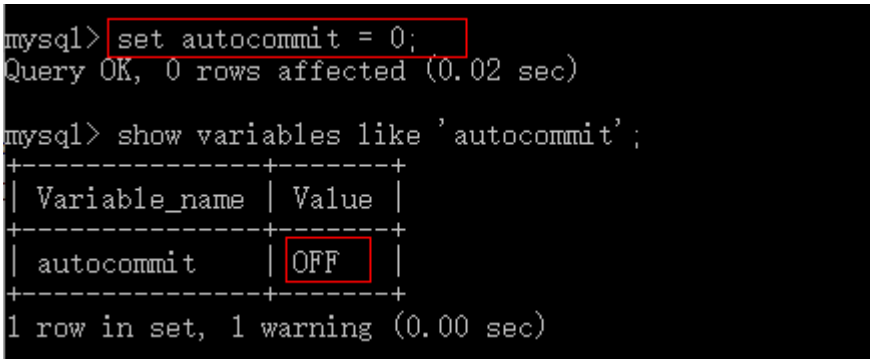
查看是否开启自动提交事务

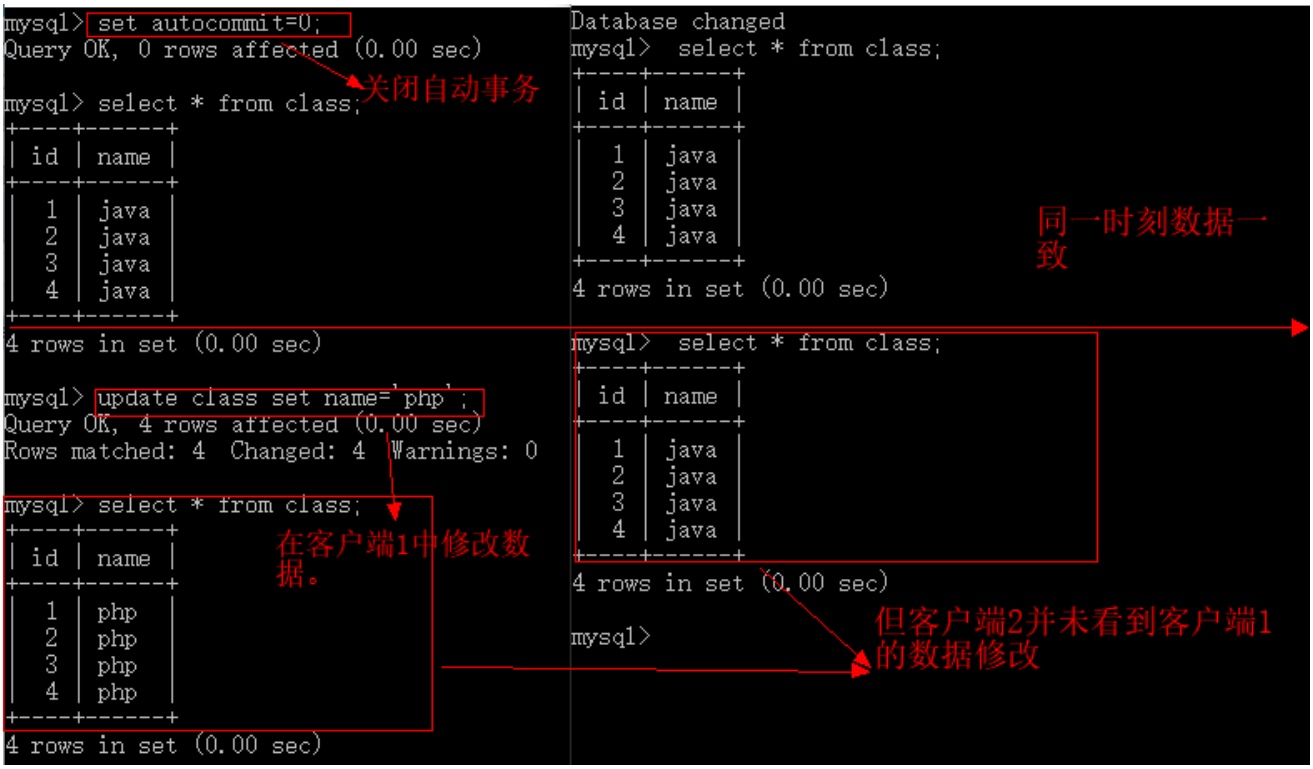
```
show variables like 'autocommit';
```

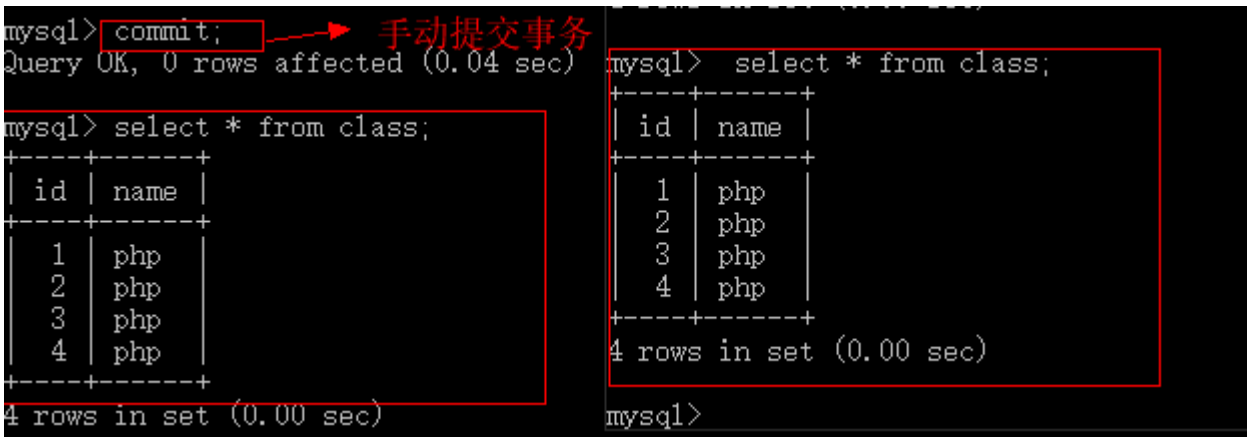
```
mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set, 1 warning (0.00 sec)
```

更改自动提交，变为手动。那么在每次进行写操作时，最后都要提交同步到表中。不然所有操作都无效。

```
set autocommit = 0;
```







手动事务

步骤

- 开启事务：告诉系统一下所有操作（写）不要直接写入到数据表，先存放到事务日志；

```
start transcation;
```

- 中间事务操作（写）：更改数据表数据

```
update student set age=18 where id=1;
update student set age=16 where id=1;
```

- 关闭事务：选择性的将日志文件中的操作的结构保存到数据表（同步）,并清空事务日志(原来的操作全部清空)。

  两种方式：

  1.提交事务：同步数据表（操作有效）

```
commit;
```

  2.回滚事务：直接情况日志表（操作失效）

```
rollback;
```

回滚点

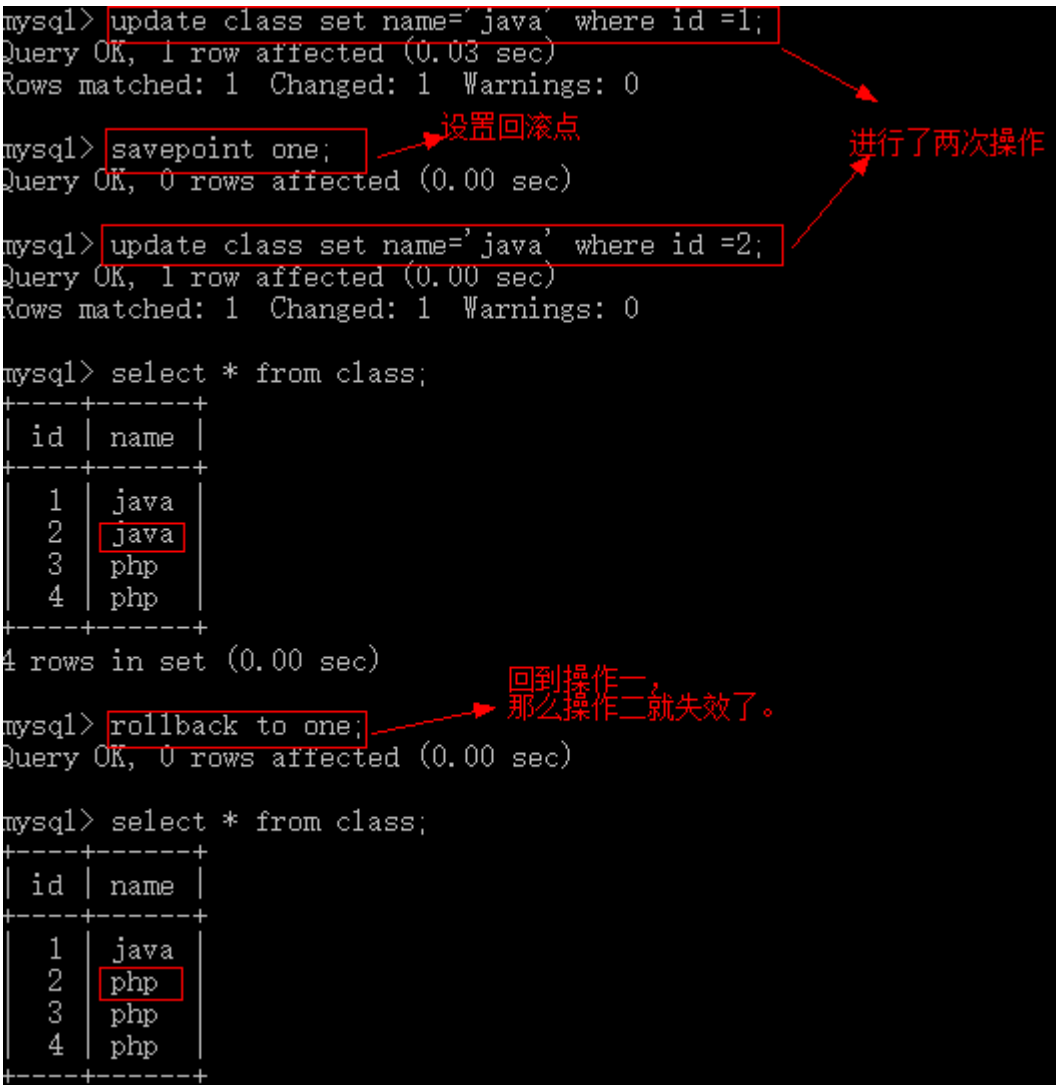
简介

在某个成功的操作完成之后，后续的操作有可能成功，但是不管成功还是失败，前面的操作都已经成功。可以在当前成功的位置设置一个回滚点。可以供后续失败的操作返回到该位置，而不是返回所有操作。

语法

savepoint 回滚点名;

rollback to 回滚点名;



锁机制

简介

各个客户端之间如果对相同表进行操作的话，Mysql会有一个锁机制。从而保证数据操作不混乱（类似多线程中实现卖票）。

innodb默认是行锁，但是如果在事务操作的过程中，没有使用到索引（主键或唯一键。。。），那么系统会自动全部检索数据，自动升级为表锁。

- 行锁：当前行数据段只有当前用户才能进行操作，其他用户只能等待该用户操作完之后才能操作。
- 表锁：当前表中的所有数据段只有当前用户才能进行操作，其他用户只能等待该用户操作完之后才能操作。

触发器

简介

trigger(触发器)，事先为某张表绑定好一段代码，当表中的某些数据记录发生变化的时候（增删改）的时候，系统会自动触发代码(执行增删改)并执行。

触发器包含

- 事件类型：增：insert；删：drop；改：update
- 触发时间：前：before；后：after
- 触发对象：表中的每一条记录

一张表中只能拥有一种触发时间的一种事件类型。

也就是说在insert前触发或者insert后触发。

最多有6个触发器。

命令

创建

注意

由于触发器内部的每行代码结束符和sql语句的结束冲突。

所以在创建触发器之前需要更改sql结束符。

并在最在创建之后恢复sql结束符。

修改sql语句结束符

```
delimiter 结束符
delimiter $
```

```
delimiter $

create trigger 下单 after insert on my_order for each row

begin

update my_goods set inv=inv-1 where id=1;

end

$

delimiter ;
```

查看触发器

- 查看所有触发器

```
show triggers;
```

- 查看构造

show create trigger 下单\G;

```
mysql> show create trigger 下单
    -> ;
| Trigger | sql_mode                                   | SQL Original Statement
                                                       | character_set_client | collation_connection | Database Collation | Created
| 下单    | NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION | CREATE DEFINER=`skip-grants user`@`skip-grants host` trigger 下单 after insert on my_orde
 begin update my_goods set inv=inv-1 where id=1;end    | utf8                 | utf8_general_ci      | utf8_general_ci    | 2018-04-18 13:59:52.69
1 row in set (0.00 sec)

mysql> show create trigger 下单\G;
*************************** 1. row ***************************
               Trigger: 下单
              sql_mode: NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
SQL Original Statement: CREATE DEFINER=`skip-grants user`@`skip-grants host` trigger 下单 after insert on my_order for each row begin update my_g
v-1 where id=1;end
  character_set_client: utf8
  collation_connection: utf8_general_ci
    Database Collation: utf8_general_ci
               Created: 2018-04-18 13:59:52.69
1 row in set (0.00 sec)

ERROR:
No query specified
```

删除构造器

drop trigger 触发器名;

使用

触发器不用去调用，而是当构造触发器时当触发到事件时，系统会自动触发调用。

记录

简介

不管触发器是否触发了，只要当某种操作准备执行，系统就会将当前要操作的状态和操作之后的状态记录起来。供触发器使用，其中，当前状态保存到old中，操作之后的状态保存到new中。

保存的该状态其实也就是一张表，不同时刻状态操作的表。

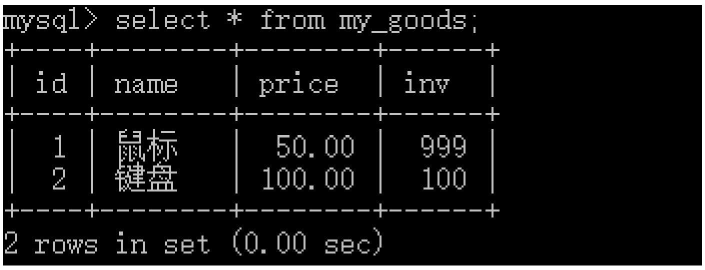
old代表旧记录。

new代表新纪录。

意义

拿到了记录之后就可以动态的进行数据更新.

举例1



```
mysql> select * from my_goods;
+----+--------+--------+------+
| id | name   | price  | inv  |
+----+--------+--------+------+
|  1 | 鼠标   |  50.00 |  999 |
|  2 | 键盘   | 100.00 |  100 |
+----+--------+--------+------+
2 rows in set (0.00 sec)
```

创建触发器

更新数据调用触发器

```
mysql> update my_goods set inv:=100 where id=1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

输出保存全局变量

```
mysql> select * from my_goods;
+----+--------+--------+------+
| id | name   | price  | inv  |
+----+--------+--------+------+
|  1 | 鼠标   |  50.00 |  999 |
|  2 | 键盘   | 100.00 |  100 |
+----+--------+--------+------+
2 rows in set (0.00 sec)

mysql> update my_goods set inv:=100 where id=1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select @old_inv,@new_inv;
+----------+----------+
| @old_inv | @new_inv |
+----------+----------+
|      999 |      100 |
+----------+----------+
1 row in set (0.00 sec)
```

举例2

有问题的触发器,注意触发代码。

当中的id值是写死的，那么每次触发改变的数据只是一条记录.

而且业务也是不合理的。本因是订单商品数量下单多少，库存也因相应减少多少.。

```
delimiter $

create trigger 下单 after insert on my_order for each row

begin

update my_goods set inv=inv-1 where id=1;

end$

delimiter ;
```

正确触发器

```
delimiter $

create trigger 下单 after insert on my_order for each row

begin

update my_goods set inv=inv-g_number where id=new.g_id;

end$

delimiter ;
```

注意

- 因为记录数据存在于内存，而from是获取磁盘数据，所以不能作为数据源输出.
- 触发器不能够select查询数据，只能增删改.
- insert时，old记录为null，delete时，new记录为null

代码执行结构

if分支

语法

```
if 条件判断
then --满足条件执行下面代码
---------
else --不满足条件执行下面代码
---------
ene if
```

then和end if就像是{}，用来包裹代码

while分支

语法

- 完整执行while代码

```
while 条件判断

do

满足条件执行代码;

end while;
```

- 不完整执行while代码（出现iterate，leave）

```
循环名称:while 条件判断
do
满足条件执行代码;
if 条件判断
then
leave/iterate 循环名称;
end if;
end while;
```

iterate:类似continue

leave:类似break

函数

简介

将一段代码封装到一个结构中，在需要执行的时候，调用函数执行结构代码。

函数的调用

```
select 函数名();
```

注意

- 与创建触发器一样，函数的内部每段代码结束符跟sql语句的结束符冲突，所以首先需要对sql的结束符进行替换.
- 函数与触发器一样，不能进行查询操作.

常见系统函数

myslq中数据的处理最小单位是字符，不是字节。

- substring()：字符串截取
- char_length：字符长度
- length：字节长度
- instr：判断某一字符串是否在某个具体字符串存在
- lpad：左填充，将字符串按照某个指定的填充方式（字符串）,填充到指定长度(字符单位)
- insert：找到目标位置，指定长度的字符串，替换成目标字符串

- strcmp：字符串的比较

自定义函数

语法

创建

[^create function 函数名([形参])]:

查看

- 查看所有函数，包括系统函数

```
show function status\G;
```

- 查看指定函数

```
show create function 函数名\G;
```

删除

```
drop function 函数名;
```

存储过程

简介与函数的区别

存储过程是一种处理数据的方式，跟函数对比起来，区别就是存储过程没有返回值而已。

函数只能对数据的操作只有**增删改**，而存储过程**增删改查**都行。

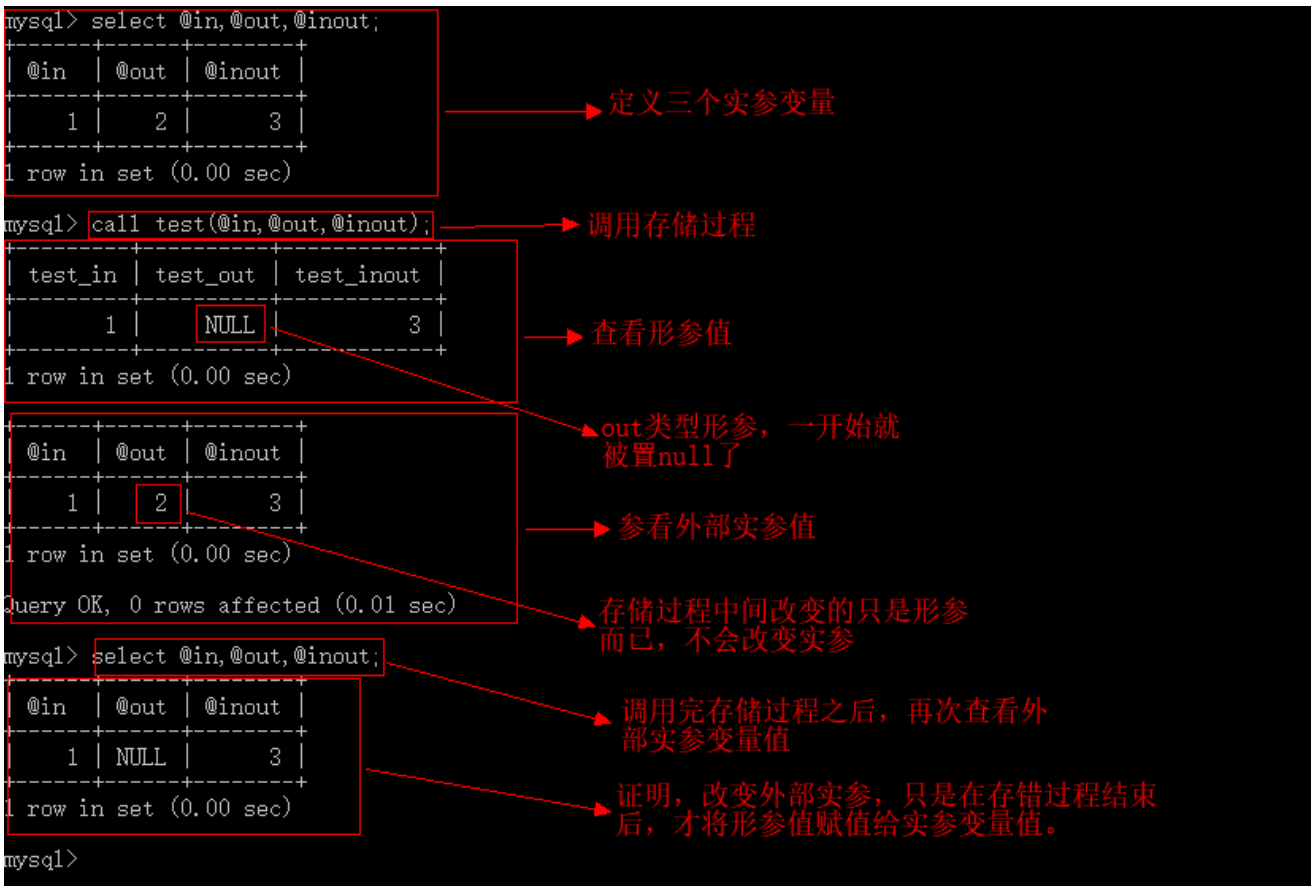
三种形式参数

由于存储过程不能返回值，那么存储过程的参数除了传入参数到存储过程外，就是将处理数据赋值给实参的i形式返回出去。

证明

存储过程构造

```
delimiter $$
create procedure test(in test_in int , out test_out int , inout test_inout int )
begin
select test_in,test_out,test_inout; -- 查看三种类型形参
select @in,@out,@inout;             -- 查看三种类型外部实参
end$$
delimiter ;
```



存储过程三种形式参数的意义

那么存储过程没有返回值的话，怎么对数据进行返回呢?

所以存储过程三种形式的参数除了**能够从获取外部参数值外，也能够将存储过程处理之后的数据赋值给外部实参。**进而以另外种形式将数据返回出去。

语法

创建存储过程

```
delimiter $$
create procedure 存储过程名 (形参类型 形参名 形参数据类型,...)
begin
存储过程结构逻辑代码
......
end$$
delimiter ;
```

如：

```
delimiter $$
create procedure test(in test_in int , out test_out int , inout test_inout int )
begin
select test_in,test_out,test_inout;
select @in,@out,@inout;
end$$
delimiter ;
```

查看存储过程构造

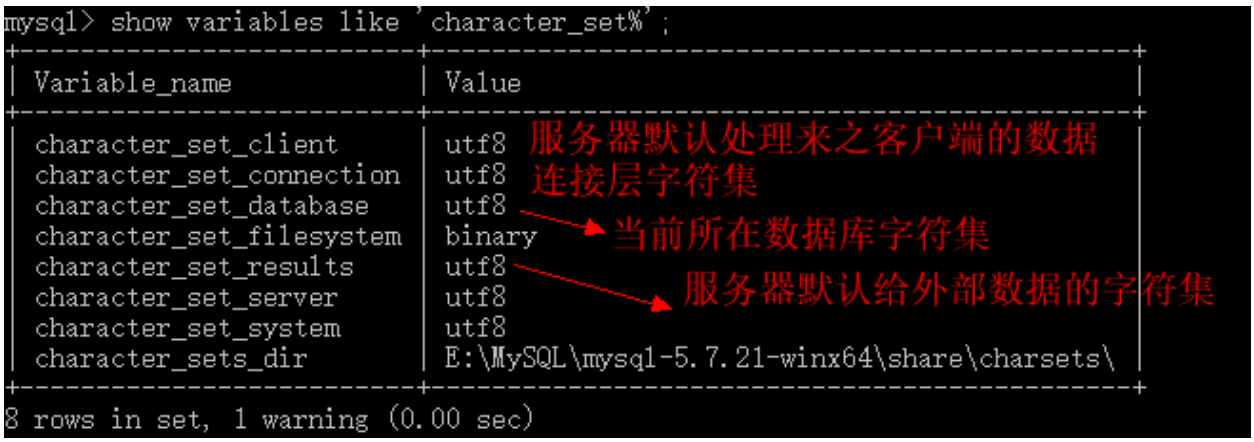
```
show create procedure 存储过程名;
```

删除存储过程

```
drop procedure 存储过程名;
```

处理中文字符乱码问题

- **查看数据库系统支持所有字符编码：** show chaacter set;
- **查看数据库系统对外默认支持编码：** show variables like 'character_set%';



连接层字符集有什么用?

它时字符集转变的中间者，如果统一了效率更高，不统一也没问题。

如果客户端时GBK，而数据库系统时utf-8，那么需要改变数据库系统的字符集。

那么改变的是什么字符集呢?

1。服务器对外接受来之客户端的数据字符集

2。服务器对外发送给客户端的数据字符集

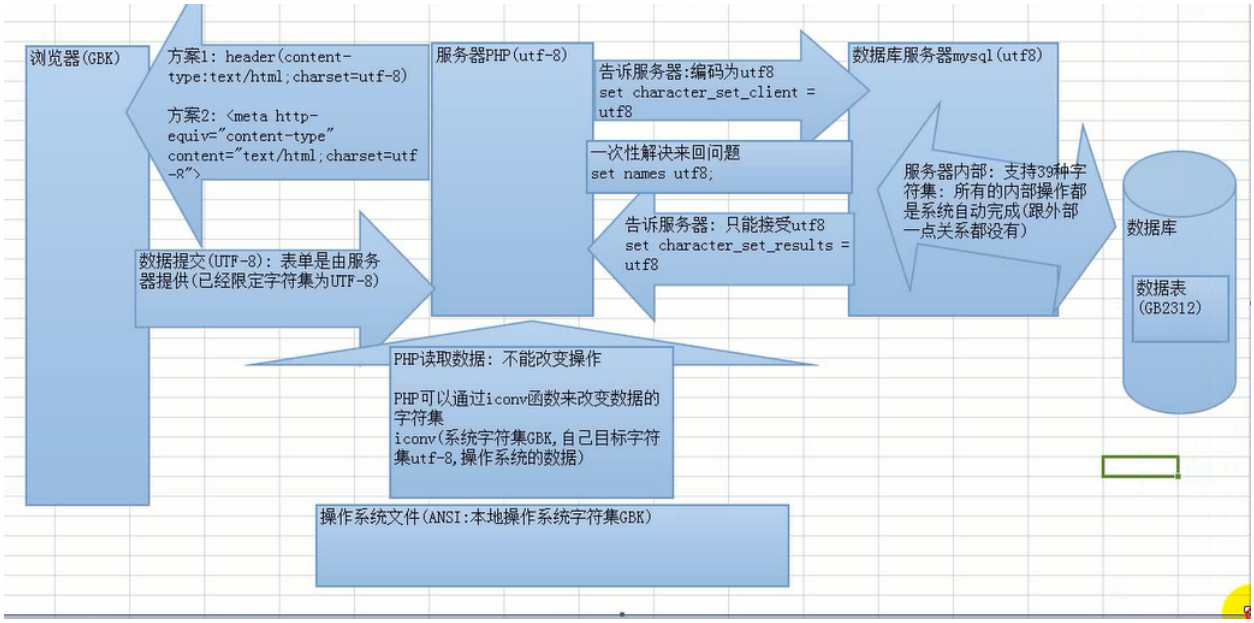
3。连接层字符集（提高效率）

- **修改数据库系统指定字符集：** set 修改字符集常量名 = 字符集;

  如：set character_set_client= gbk;

- **修改数据库系统三种处理数据字符集：**set names 字符集；

  三种：对外接受，对外发送，连接层字符集



注意！！！修改数据库系统字符集只限于会话期间

Mysql的JDBC驱动连接问题

SSL连接

没有服务器的身份验证建立SSL连接，不推荐。根据MySQL 5.5.45 +，+，+ 5.6.26 5.7.6要求SSL连接必须建立明确的选项默认情况下如果不设置。符合现有的应用程序不使用SSL的verifyservercertificate属性设置为“false”。你需要显式禁用SSL设置usessl = false，或设置usessl =真实提供服务器证书验证信任库。

解决办法

再连接url添加useSSL=false参数

```
spring.datasource.url=jdbc:mysql://localhost:3306/demo?
useUnicode=true&characterEncoding=UTF-8&useSSL=false
```

添加到数据库乱码问题

添加中文数据时可能出现插入时乱码

解决办法

再连接url添加useUnicode=true&characterEncoding=utf-8参数

```
spring.datasource.url=jdbc:mysql://localhost:3306/demo?
useUnicode=true&characterEncoding=utf-8
```

The server time zone value '�й����□ħ��' is unrecognized

mysql-connector-java这个jar包是最新的时候 在配置datasource.url时不能简单的这样配：
spring.datasource.url=jdbc:mysql://localhost:3306/chat

解决办法

需要加上一些必要的后缀信息（改成下面的配置就可以了）：

```
spring.datasource.url=jdbc:mysql://localhost:3306/chat?
useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTi
mezone=UTC
```