问题总结

1. 获取系统时间

https://blog.csdn.net/u012326462/article/details/82081756

2. INSERT INTO指令中赋值字符串用单引号

3. Windows安装

> - https://www.runoob.com/mysql/mysql-install.html
>
>   **MySQL版本为8+不需要配置my.ini中的datadir**
>
> - root密码为空时，设置root密码
>
>   https://blog.csdn.net/fengzhantian/article/details/90213785
> - 配置环境变量
>
>   https://blog.csdn.net/wd2011063437/article/details/78793917
> - 修改root密码

4. 不是只有主键可以自增，给非主键设置自增时需要设成唯一主键

https://www.cnblogs.com/mm20/p/8060425.html

PRIMARY KEY (主键的列不能为NULL)

5.批量插入大量数据

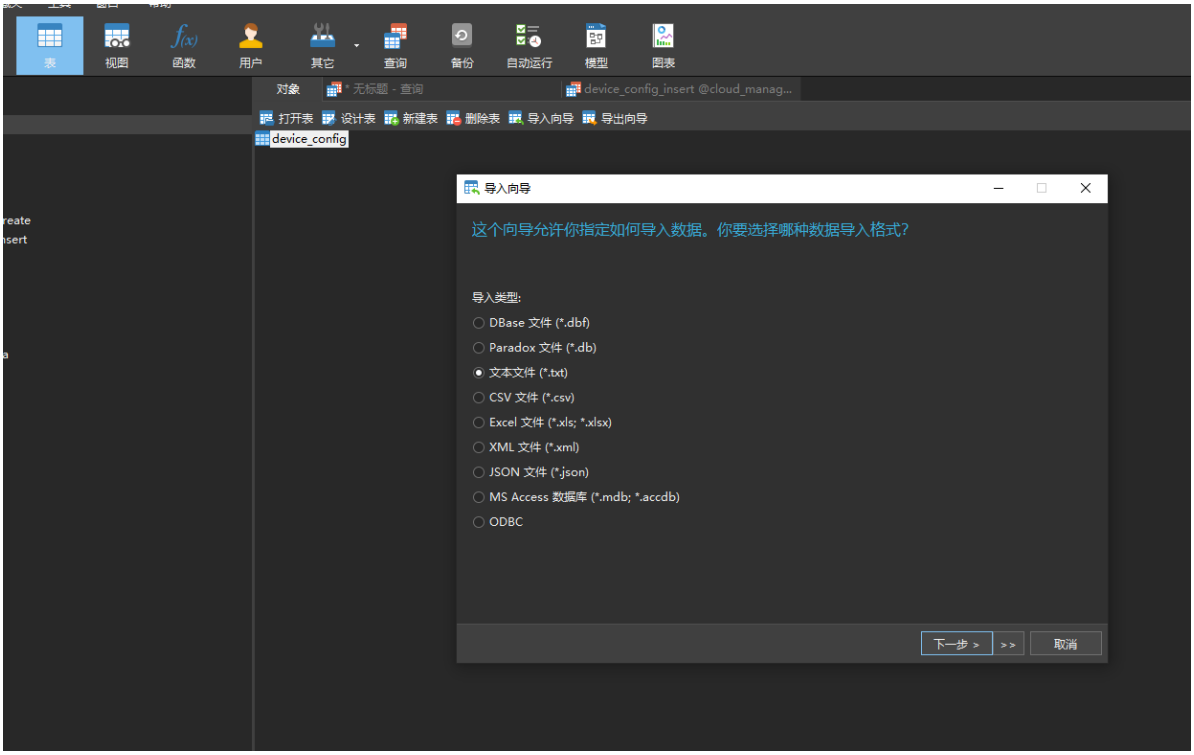
通过INSERT INTO插入

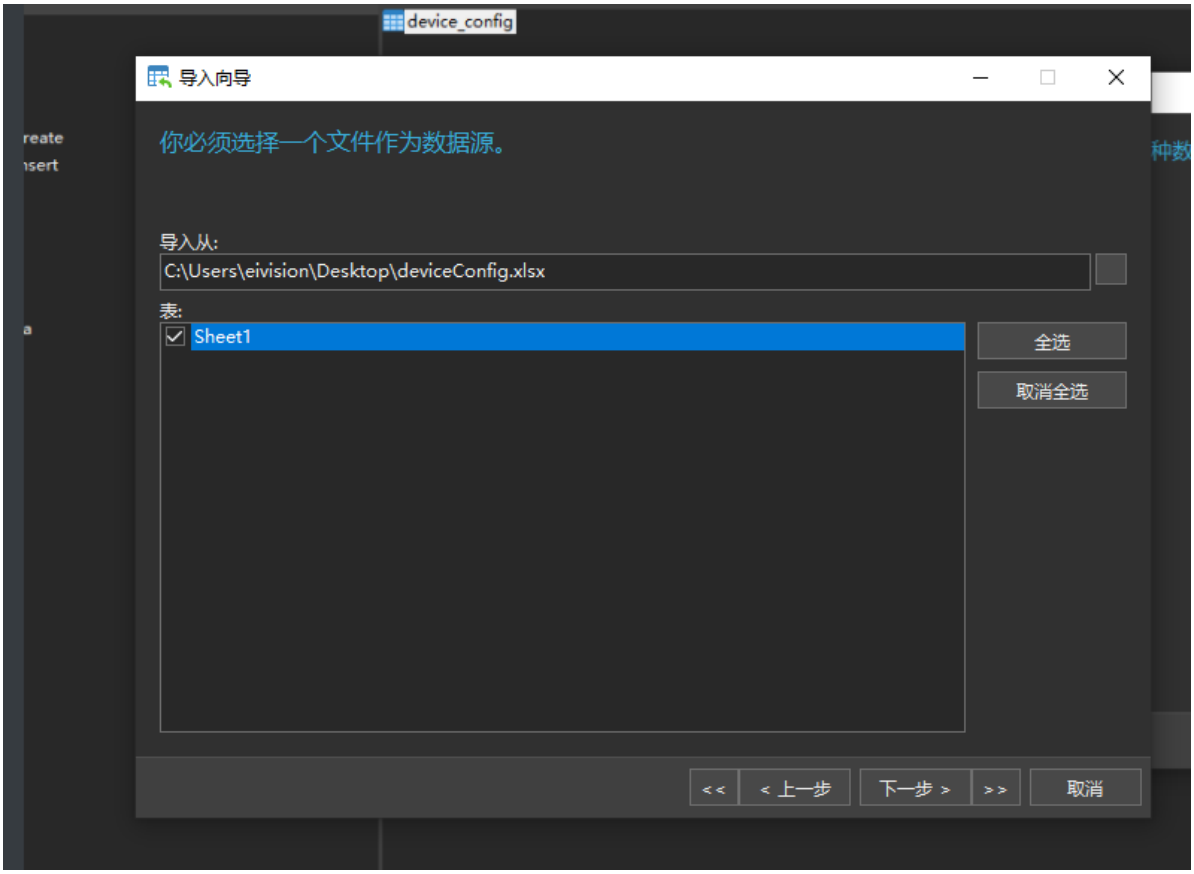
```
INSERT INTO device_config(
`LineNO`,
`DeviceStatus_001`,
`DeviceStatus_002`,
`DeviceStatus_003`,
`DeviceStatus_004`,
`DeviceStatus_005`,
`DeviceStatus_006`,
`DeviceStatus_007`,
`DeviceStatus_008`,
`DeviceStatus_009`,
`DeviceStatus_010`,
`DeviceStatus_011`,
`DeviceStatus_012`,
`DeviceStatus_013`,
`DeviceStatus_101`,
`DeviceStatus_102`,
`DeviceStatus_103`,
`DeviceStatus_104`,
`DeviceStatus_105`
```

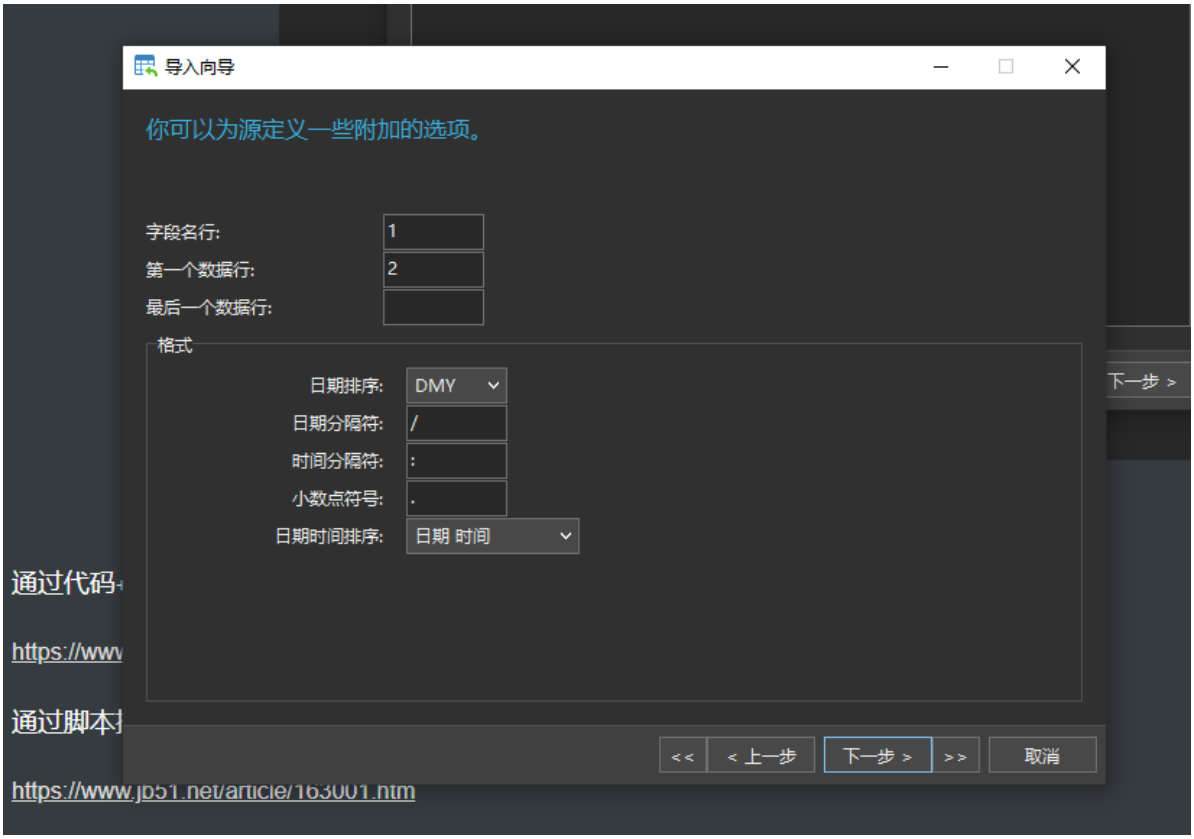
```
)
VALUES
('001','1','1','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'),
('002','0','0','0','1','1','1','0','0','0','0','0','0','0','0','0','0','0','0'),
('003','0','0','0','0','0','0','1','1','1','0','0','0','0','0','0','0','0','0'),
('004','0','0','0','0','0','0','0','0','0','1','1','1','0','0','0','0','0','0'),
('005','0','0','0','0','0','0','0','0','0','0','0','0','1','1','1','0','0','0'),
('006','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','1','1','1'),
('007','1','1','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'),
('008','0','0','0','1','1','1','0','0','0','0','0','0','0','0','0','0','0','0'),
('009','0','0','0','0','0','0','1','1','1','0','0','0','0','0','0','0','0','0'),
('010','0','0','0','0','0','0','0','0','0','1','1','1','0','0','0','0','0','0'),
('011','0','0','0','0','0','0','0','0','0','0','0','0','1','1','1','0','0','0'),
('012','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','1','1','1'),
('013','1','1','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'),
('101','0','0','0','1','1','1','0','0','0','0','0','0','0','0','0','0','0','0'),
('102','0','0','0','0','0','0','1','1','1','0','0','0','0','0','0','0','0','0'),
('103','0','0','0','0','0','0','0','0','0','1','1','1','0','0','0','0','0','0'),
('104','0','0','0','0','0','0','0','0','0','0','0','0','1','1','1','0','0','0'),
('105','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','1','1','1');
```

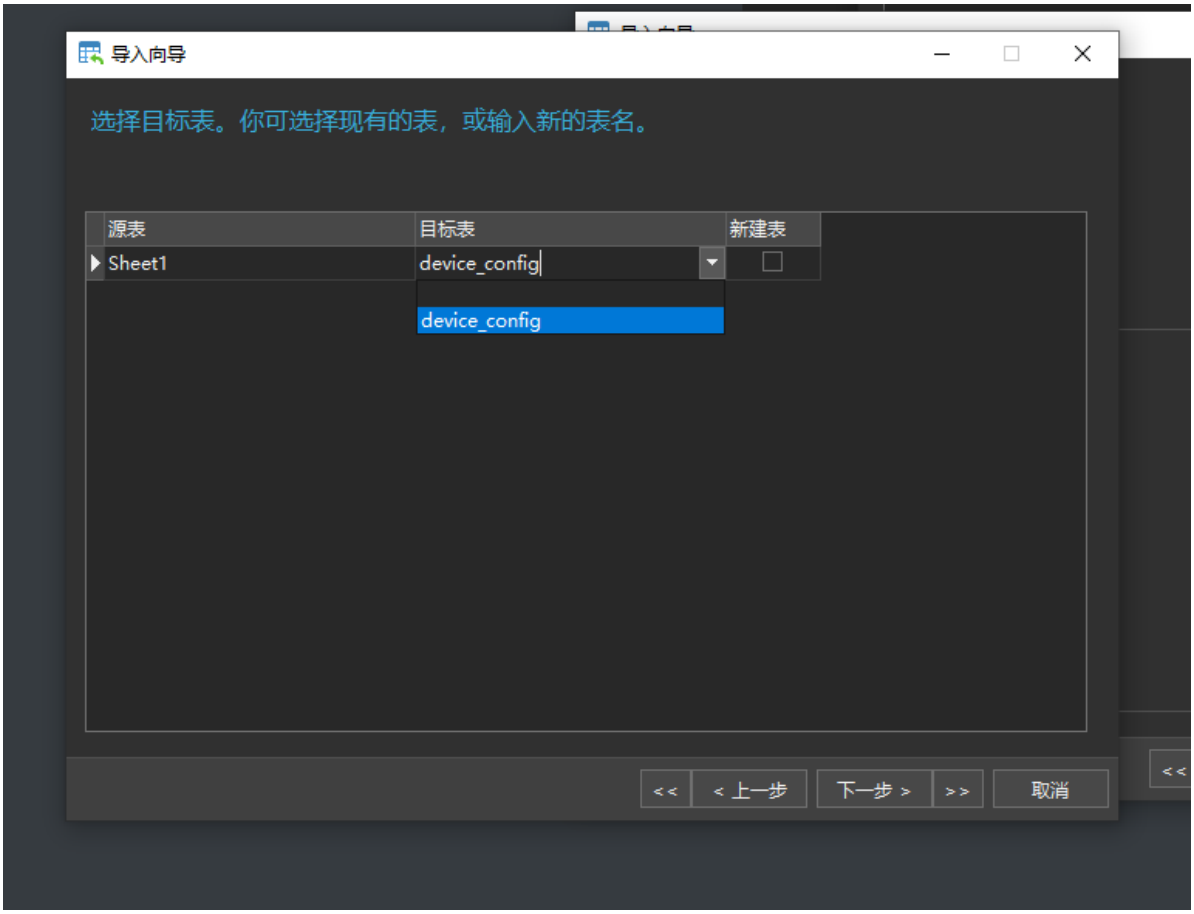
通过Excel导入

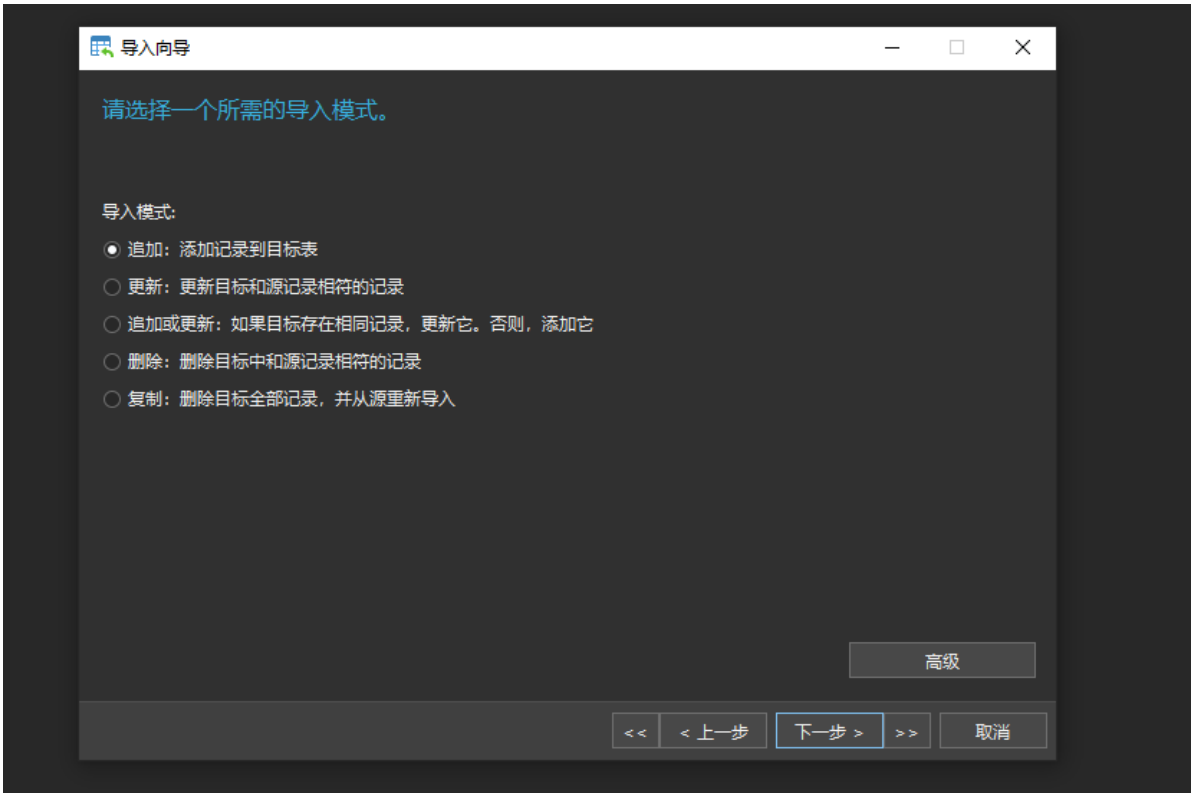
https://www.cnblogs.com/vuciao/p/10586766.html




device_config
导入向导
你必须选择一个文件作为数据源。
导入从:
C:\Users\eivision\Desktop\deviceConfig.xlsx
表:
Sheet1
全选
取消全选
<< < 上一步 下一步 > >> 取消


导入向导
你可以为源定义一些附加的选项。
字段名行: 1
第一个数据行: 2
最后一个数据行:
格式
日期排序: DMY
日期分隔符: /
时间分隔符: :
小数点符号: .
日期时间排序: 日期 时间
<< < 上一步 下一步 > >> 取消
下一步 >
通过代码
https://www
通过脚本
https://www.jb51.net/article/163001.htm





通过代码+事务插入

https://www.jianshu.com/p/63bd412cb030

通过脚本插入

https://www.jb51.net/article/163001.htm

6. 关于反引号

反引号一般用于把字段名、表名括起来，但一般来说字段名、表名若不会与关键字混淆，就不需要加。

7. MySQL默认不区分大小写

MySQL默认是不区分大小写的，但是在很多情况下需要大小敏感

修改MySql Server安装目录下的 my.ini 文件，在mysqld节下加入下面一行 set-variable=lower_case_table_names=0 （0：大小写敏感；1：大小写不敏感）最后重启一下MySql服务即可。

8. MySQL配置局域网可访问

- 两台电脑通过网线连接，配置同一网段
- https://www.cnblogs.com/-ccj/p/12031243.html

  配置用户名、密码

9. 插入/删除记录后将表的主键ID重排

https://www.cnblogs.com/devcjq/articles/6020391.html

```
truncate table table_name;  -- 清空表重新插入
```

MySQL基本知识

1.基本概念

- MySQL是一个C/S型DBMS，user通过客户端访问服务器mysqld.exe，服务器软件副本可在本机也可在服务器上。

  常用客户端有GUI的也有CLI的，如cmd中的mysql、Navicat等。在cmd中输入mysql调用CLI形式的Client软件，直接输入mysql可能报错，因为该CLI客户端软件没有用户名和密码来连接服务器。

  ```
  mysql -hlocalhost -p3036 -uroot -p  //输入用户名和密码，调用CLI，连接服务器
  ```

- 注释

  单行：--空格，或#

  多行/**/

2. 数据结构

3.SHOW命令

```
show databases; //显示所有数据库
show tables;    //显示所有表
desc t_name;    //显示表结构
show create database db_name;   //显示数据库db_name创建时的信息
show create table t_name;   //显示表t_name创建时的信息
```

4.检索数据

- 检索出来的原始数据需要应用程序规定显示的格式
- 单列

  ```
  SELECT filed_name FROM t_name;
  ```

- 多列

  ```
  SELECT filed_name1,field_name2 FROM t_name;
  ```

- 所有列

  ```
  //通配符
  //一般在不知道字段名称时用于显示
  SELECT * FROM t_name;
  ```

- 非主键的字段显示存在值重复的情况

  字段若不是主键，则表有多少行就会显示多少行，存在重复问题。

  使用DISTINCT，只显示该字段的不同值。

  ```
  SELECT DISTINCT filed_name FROM t_name;
  ```

- 限制显示的行数

  ```
  SELECT filed_name FROM t_name LIMIT n;  //从0行开始显示n行
  SELECT filed_name FROM t_name LIMIT n,m;    //从n行开始显示m行,行编号从0开始
  ```

- 完全限定名

  ```
  SELECT t_name.filed_name FROM db_name.t_name;
  //使用.标识字段所在表和表所在数据库
  ```

- 排序数据

  (1) ORDER BY

  可以按照查询的字段进行排序，也可按照未查询的字段进行排序。

  按多个字段排序，当第一个字段相同时，才会在第一个字段相同的行内部按照第二个字段进行排序。若所有行第一个字段都相同，不会按照第二个字段排序。

```
SELECT field_name1,field_name2,field_name3 FROM t_name ORDER BY
field_name1,field_name2;
```

(2) 默认升序，DESC设置降序

```
SELECT field_name1,field_name2,field_name3 FROM t_name ORDER BY field_name1
DESC;
```

(3) DESC只针对关键字前紧邻的字段有效。其他未用DESC修饰的字段仍然以升序排序。

```
SELECT field_name1,field_name2,field_name3 FROM t_name ORDER BY field_name1
DESC,field_name2;
//field_name1按照降序排序，field_name1相同的按照field_name2升序排序
```

(4) 若想对多个字段降序排序，每个字段都需要DESC修饰

- 语句顺序

  SELECT => FROM => ORDER BY => LIMIT

5. 过滤数据

- 通过WHERE子句指定过滤条件

  为什么要在数据库中指定过滤，而不通过应用层程序过滤：处理数据库的工作会极大影响应用程序的性能，且服务器需要发送大量无用数据占用带宽。

  注意关键字where，where后面跟上一个或者多个条件，条件是对前面数据的过滤，只有满足where后面条件的数据才会被返回

- WHERE子句在ORDEY BY之前，LIMIT在最后。
- 过滤条件

```
=
!=
<
<=
>
>=
BETWEEN val1 AND val2   //范围：[val1,val2]
IN (val1,val2,val3,val4)    //列举
```

- 创建表时，可指定某一个字段是否可以不包含值。当一个列不含值时，为包含空值NULL

```
//空值检查，与字段包含0、空字符串、空格不同。
WHERE field_name IS NULL;
```

- 用AND/OR连接多条WHERE语句。用()明确分组过滤条件，避免歧义。

```
SELECT field_name1,field_name2,field_name3 FROM t_name
WHERE field_name1=val1 AND field_name2<=val2;
```

- IN(val1,val2,val3,...)指定WHERE语句中字段的所需取值列表。需要取表中某些指定的字段的行。

  IN子句中可以包含SELECT子句。

- NOT

  可以容易的过滤出不符合给定条件的行。

```
NOT IN
NOT BETWEEN AND
```

- BETWEEN AND

  操作符 BETWEEN ... AND 会选取介于两个值之间的数据范围，这些值**可以是数值、文本或者日期，属于一个闭区间查询。**

- LIKE模式，使用通配符进行过滤

  LIKE谓词：在子句中使用LIKE表明子句中使用的搜索模式，是通配符匹配而不是相等匹配。

```
SELECT field_name1,field_name2,field_name3 FROM t_name
WHERE field_name1 LIKE 'jet%';
```

  %可在搜索模式中任意位置，区分大小写，可有多个。可匹配0/1/多个字符。

  注意：待匹配字符串末尾有空格，需要用函数去掉首尾空格。

  _只能匹配一个字符

  通配符速度慢，不要滥用，尽量不把通配符放在字符串开头。

```
-- 24、查询id、货品名称、分类编号，零售价大于等于80并且货品名称匹配'%罗技M1_ _':
SELECT id,productName,dir_id,salePrice FROM product WHERE salePrice>=80 AND
productName LIKE '%罗技M1__'
```

6.正则表达式

- REGEXP模式表示后面跟着的是正则表达式
- 表示匹配任意一个字符：.
- 匹配或关系的若干个串： |
- 匹配集合中的单一字符，集合：[c1c2c3...]，表示匹配c1或c2或c3。

```
[^c1c2c3]   //匹配除c1c2c3之外的所有字符
[0-9]
[a-z]   //表示范围
```

- 转义：

  \

```
\\.   表示匹配.
\\\   表示匹配\
```

- 字符含义

  https://www.cnblogs.com/kuqs/p/5727409.html

```
\bbuckets\b  表示两侧都是词界，只匹配buckets单词
^buckets$    同上，只匹配字符串buckets
```

正则练习网页：https://alf.nu/RegexGolf

- REGEXP '1000'和LIKE '1000'的区别

  对于'a1000a'，后者不会匹配，前者会匹配。前者相当于LIKE'%1000%'。后者相当于REGEXP '%1000$'。

- MySQL的正则不支持\w,\s,\d
- 练习

  https://blog.csdn.net/qq_28633249/article/details/77686976

7.默认值约束

- DEFAULT用于约束字段，当字段未输入值时，自动添加一个预设的值。

  一般用于NOT NULL，防止未输入值导致出错。

  http://c.biancheng.net/view/2447.html

```
mysql> CREATE TABLE tb_dept3
    -> (
    -> id INT(11) PRIMARY KEY,
    -> name VARCHAR(22),
    -> location VARCHAR(50) DEFAULT 'Beijing'
    -> );
    mysql>INSERT INTO tb_name
    ->(id,name)
    ->VALUES
    ->(1,'wang');
    //某个字段是默认值时，插入字段中默认字段不写
```

- 修改表时添加默认约束

```
ALTER TABLE <数据表名>
CHANGE COLUMN <字段名> <数据类型> DEFAULT <默认值>;
```

8.非空约束

http://c.biancheng.net/view/2448.html

- 字段不指定时，默认为可为NULL
- 删除NOT NULL约束

```
ALTER TABLE <数据表名>
CHANGE COLUMN <字段名> <字段名> <数据类型> NULL;
```

- NULL相等不用=，用IS

```
SELECT id,productName,dir_id FROM product WHERE dir_id IS NULL
```

```
SELECT id,productName,dir_id FROM product WHERE dir_id IS NOT NULL
```

9.常用指令

库操作

```
CREATE DATABASE db_name;      //增
DROP DATABASE db_name;       //删
ALTER DATABASE db_name;      //改
SHOW DATABASES;              //查
USE db_name;     //使用库
SELECT DATABASE();  //当前使用的库
```

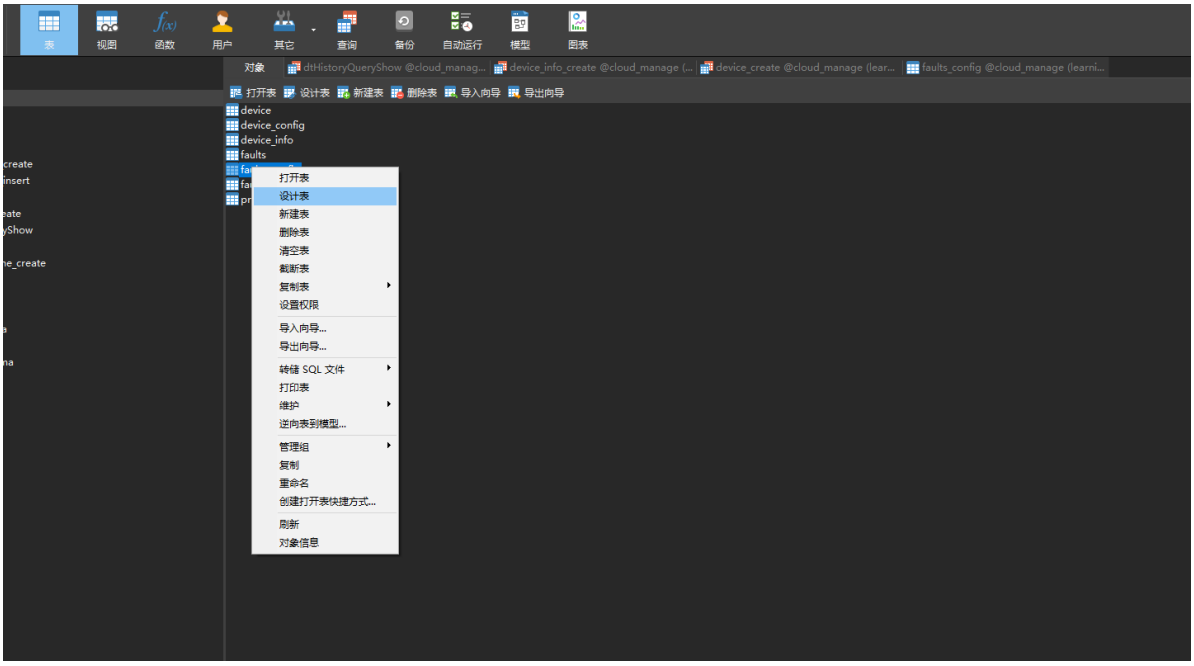
表操作

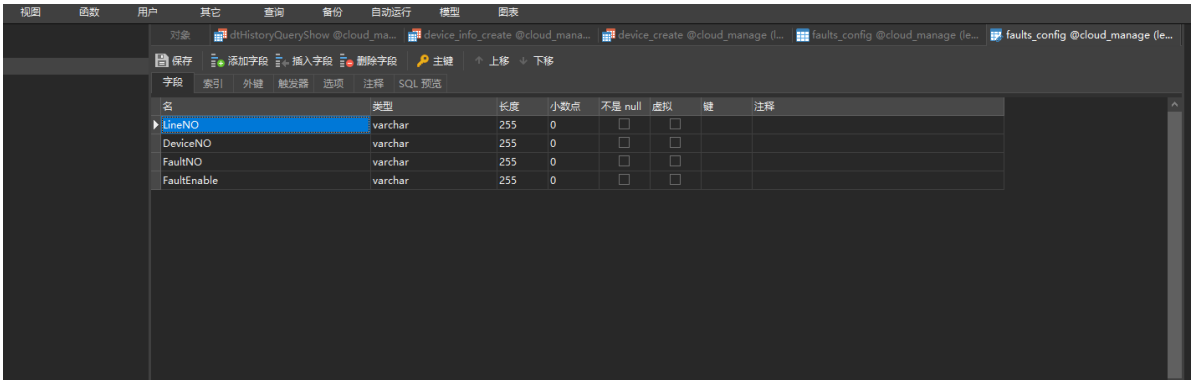
```
//增
CREATE TABLE IF NOT EXISTS runoob_tbl(
   `runoob_id` INT UNSIGNED NOT NULL AUTO_INCREMENT,     --默认非NULL，自增
   `runoob_title` VARCHAR(100) NOT NULL,                 --默认非NULL
   `runoob_author` VARCHAR(40) DEFAULT NULL,             --默认值NULL
   `submission_date` DATE   DEFAULT NULL,
   PRIMARY KEY ( `runoob_id` )
)ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;
```

```
//删
DROP TABLE runoob_tbl;  //表被删除
TRUNCATE TABLE runoob_tb1;  //表还在，但是表中记录都被清空
```

```
//改
ALTER TABLE tb_name ADD COLUMN `newField` typename(n) NOT NULL DEFAULT '...'
COMMENT '...' AFTER field_;
ALTER TABLE tb_name DROP COLUMN field_;
ALTER TABLE tb_name MODIFY COLUMN field_ typename(n) NOT NULL DEFAULT '' COMMENT
AFTER field_pre;
ALTER TABLE tb_name CHANGE COLUMN field_old field_new typename(n) NOT NULL
DEFAULT '' COMMENT '' AFTER field_;
```

用Navicat修改表结构：





主键字段的值不能重复

```
//查
SHOW TABLES;     //显示当前库中所有的表
```

字段操作

```
//插入记录
INSERT INTO tb_name (field1,field2) VALUES (val11,val12),(val21,val22),
(val31,val32);        //若每个字段都按顺序赋值则可省略字段名，若有字段被省略，则需要将赋值的
字段写出来
//删除记录
DELETE FROM tb_name [WHERE clause];
//修改某条记录的某个字段的值
UPDATE tb_name SET `field_name` = newVal;
//显示某个字段
SELECT `field_name1` FROM tb_name;
//复制表中记录（其中，字段1的值不是表中的，其他字段是从表中查询得到的）
insert into tablename (字段1,字段2,字段3,...) select 自定义的值 as 字段1,字段2,字段
3,... from tablename where....;
```

- MODIFY和CHANGE的区别：

  https://www.cnblogs.com/mentalidade/p/6293200.html

  MODIFY无法修改字段的名称，CHANGE可以。

- COMMENT

给字段添加注释

- AFTER

  标记当前新增/修改的字段位于某一字段后。

- 小结：

  库/表：CREATE/DROP/ALTER/SHOW，且后要加上DATABASE/TABLE表明是库还是表

  字段：INSERT INTO/DELETE FROM/UPDATE SET/SELECT FROM，且不要加上TABLE关键字。

获取表的所有字段名

```
select COLUMN_NAME from information_schema.COLUMNS where table_name =
'your_table_name' and table_schema = 'your_db_name';
```

将从表A中查询到数据插入到表B

https://blog.csdn.net/ZZQHELLO2018/article/details/103784869?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0.no_search_link&spm=1001.2101.3001.4242.1

10.主键

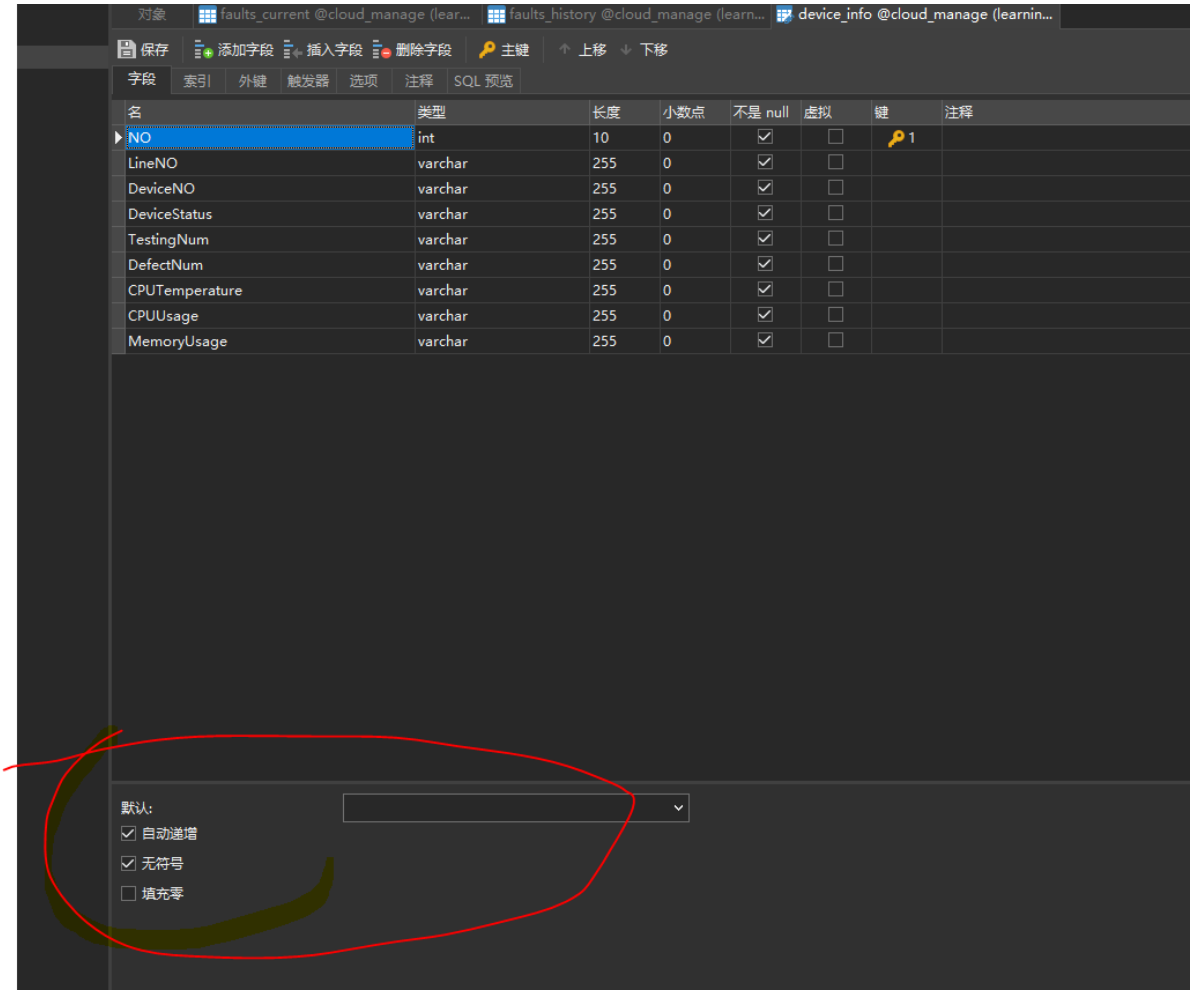
（1）概念

主键

唯一主键

（2）Navicat中给表添加主键

默认自增1



11.自带函数

（1）拼接计算字段

concat(field1, '(', field2 ,')')

' '中的字符将直接拼接在字段上

```
SELECT t1.DeviceNO,t2.DeviceName,
(CASE WHEN t1.DeviceStatus=1 THEN '正常'
WHEN t1.DeviceStatus=0 THEN '异常'
WHEN t1.DeviceStatus=-1 THEN '无效'
END) AS DeviceStatus,
t1.TestingNum,t1.DefectNum,
CONCAT(t1.CPUTemperature,'℃') AS CPUTemperature,CONCAT(t1.CPUUsage,'%') AS
CPUUsage,CONCAT(t1.MemoryUsage,'%') AS MemoryUsage
FROM device_info AS t1 INNER JOIN device AS t2
ON t1.DeviceNO=t2.DeviceNO
WHERE t1.LineNO='001'
ORDER BY t1.`NO`;
```

（2）去掉空格

Trim(field) //去掉field左右两侧的空格

LTrim(field) //去掉左侧的空格

RTrim(field) //去掉右侧的空格

(3) 别名

AS 后跟字符串单引号

```
SELECT CONCAT(`id`,'(',`name`,')') AS 'result'
FROM `student`
WHERE `department`='中文系'
ORDER BY `id`;
```

(4) 计算字段

不仅已有字段可以作为字段由SELECT显示，已有字段的计算表达式也可，还可用AS取一个别名。

```
SELECT prod_id,prod_quantity,prod_price,
prod_quantity*prod_price AS 'total price'
FROM orderitems
WHERE order_num=20005;
```

```
SELECT LineNO,
(
DeviceStatus_001 +
DeviceStatus_002 +
DeviceStatus_003 +
DeviceStatus_004 +
DeviceStatus_005 +
DeviceStatus_006 +
DeviceStatus_007 +
DeviceStatus_008 +
DeviceStatus_009 +
DeviceStatus_010 +
DeviceStatus_011 +
DeviceStatus_012 +
DeviceStatus_013 +
DeviceStatus_101 +
DeviceStatus_102 +
DeviceStatus_103 +
DeviceStatus_104 +
DeviceStatus_105
)AS DeviceTotalNum
FROM device_config;
```

(5) 日期时间

养成对应含有日期时间的字段，若需要时间则加上TIME()，若需要日期则加上DATE()的习惯。

- Now()

  当前年月日时分秒，YYYY-MM-DD格式

- Date()

  日期时间中的日期

- Time()

  日期时间中的时间

- Year()

  日期时间中的年份，年份一定要使用4位YYYY避免歧义
- Month()

  日期时间中的月份
- curdate()

  CURRENT_DATE()

  当前日期
- curtime()

  CURRENT_TIME()

  当前时分秒
- CURRENT_TIMESTAMP()

  当前日期时间
- Year(Now())

  当前年份
- Year(Now())-age

  已知年龄age，获得出生年份
- Year(Now())-birth

  已知出生年份birth，获得年龄

```
//查询2020年9月所有订单
SELECT `id`,`name`
FROM orders
WHERE Date(order_date) BETWEEN '2020-9-01' AND '2020-9-30'; //需要知道月中有多少天
```

```
SELECT `id`,`name`
FROM orders
WHERE Year(order_date)=2020 AND Month(order_date)=9;
```

(6) 聚合函数

处理一组数据，返回一个值的函数

AVG()

显示某一个字段/列的均值，跳过值为NULL的对象。

可用WHERE子句过滤出符合条件的所有对象的某一列的均值。

求均值的列的字段作为函数参数。

```
SELECT AVG(prod_price) AS 'result'
FROM products
WHERE `prod_id` IN(1001,1002,1003);
```

MIN()/MAX()

求**某列**的最小值/最大值

COUNT()

COUNT(*) 求整张表的列数，不忽略NULL

COUNT() 对某一列计数时，**忽略NULL的记录。**

SUM()

求某列记录之和，忽略NULL

对一行多列求和

```
SELECT LineNO,
(
DeviceStatus_001 +
DeviceStatus_002 +
DeviceStatus_003 +
DeviceStatus_004 +
DeviceStatus_005 +
DeviceStatus_006 +
DeviceStatus_007 +
DeviceStatus_008 +
DeviceStatus_009 +
DeviceStatus_010 +
DeviceStatus_011 +
DeviceStatus_012 +
DeviceStatus_013 +
DeviceStatus_101 +
DeviceStatus_102 +
DeviceStatus_103 +
DeviceStatus_104 +
DeviceStatus_105
)AS DeviceTotalNum
FROM device_config;
```

统计一行中

- 搭配DISTINCT使用，忽略列中的重复记录

```
SELECT AVG(DISTINCT prod_price) AS 'result'
FROM products;  //求prod_price的记录的均值
```

- 显示多个函数值

```
SELECT COUNT(*) AS `total_num`,
AVG(prod_price) AS `avg_price`,
SUM(prod_price) AS `total_price`
FROM products;
```

（7）查询函数

isnull

isnull函数，判断参数是否为空，若为空返回1，否则返回0

ifnull

ifnull函数，2个参数，判断第一个参数是否为空，如果为空返回第2个参数的值，否则返回第1个参数的值

（8）ROW_COUNT()——返回被操作影响的行数

mysql中的ROW_COUNT()可以返回前一个SQL进行UPDATE，DELETE，INSERT操作所影响的行数

```
create table t(
id int,
name varchar(50),
address varchar(100),
primary key(id,name)
)engine =InnoDB;
```

```
insert into t
(id,name,address)
values
(1,'yubowei','weifang'),
(2,'sam','qingdao');
```

```
update t set address = 'weifang'
where id = 1 and name = 'yubowei';
-- 此时查看影响的行数：
select row_count(); -- 执行结果为0；
```

```
update t set address = 'beijing'
where id = 1 and name = 'yubowei';
-- 此时查看影响的行数：
select row_count(); -- 执行结果为1；
```

从上面的测试可以得出在MySQL中只有**真正对记录进行修改了的情况下，*row_count*才会去记录影响的行数**，否则如果记录存在但是没有实际修改则不会将该次更新记录到row_count中。

（9）其他函数

函数名称	作用
version	数据库版本号
database	当前的数据库
user	当前连接用户
password	返回字符串密码形式
md5	返回字符串的md5数据

```
mysql> SELECT md5('123456');
+----------------------------------+
| md5('123456') |
+----------------------------------+
| e10adc3949ba59abbe56e057f20f883e |
+----------------------------------+
```

12. ORDER BY

用GROUP BY/HAVING过滤后，显示，注意用ORDER BY排序。

```
SELECT `prod_id`,COUNT(*) AS `prod_num` //显示prod_id和prod_num
FROM products                          //数据源
GROUP BY `prod_id`
HAVING COUNT(*)>=2;
ORDER BY `prod_num`;                     //按照prod_num排序
```

13. LIMIT

- LIMIT后跟的：不能是负数、只能是明确的数字不能是表达式
- **跳过M条数据**，显示N条数据

```
LIMIT M, N
```

- 默认跳过0条数据

```
LIMIT N     -- 跳过0条数据，显示N条数据
```

分页

开发过程中，分页我们经常使用，分页一般有2个参数：

```
SELECT * FROM t_name LIMIT (page-1)*pageSize,pageSize;
```

> page：表示第几页，从1开始，范围[1,+∞)
> pageSize：每页显示多少条记录，范围[1,+∞)
> 如：page = 2，pageSize = 10，表示获取第2页数据共十条

```
SELECT * FROM t_name ORDER BY `age` desc LIMIT 0,2;     -- 第一页
SELECT * FROM t_name ORDER BY `age` desc LIMIT 2,2;     -- 第二页
SELECT * FROM t_name ORDER BY `age` desc LIMIT 4,2;     -- 第三页
```

14.分组——GROUP BY（重点）

（1）注意事项

- GROUP BY指定分组依据的字段
- GROUP BY搭配聚合函数使用。
- SELECT后面跟着的只能是：出现在GROUP BY后的字段、聚合函数

> oracle、sqlserver、db2中也是按照这种规范来的。
> 文中使用的是5.7版本，默认是按照这种规范来的。
> mysql早期的一些版本，没有上面这些要求，select后面可以跟任何合法的列

- **特别的，若*SELECT*后不是聚合函数，也非*GROUP BY*后的字段，则显示原始表格分组后分组中第一条记录的该字段值，不受前面几个*SELECT*字段的影响。**

```
-- 需求：获取每个用户下单的最大金额及下单的年份，输出：用户id，最大金额，年份
-- 错误写法：
mysql> select
user_id 用户id, max(price) 最大金额, the_year 年份
FROM t_order t
GROUP BY t.user_id;
+----------+--------------+--------+
| 用户id | 最大金额 | 年份 |
+----------+--------------+--------+
| 1001 | 88.88 | 2017 |
| 1002 | 44.44 | 2018 |
| 1003 | 66.66 | 2018 |
+----------+--------------+--------+
3 rows in set (0.03 sec)
mysql> select * from t_order;
+----+---------+---------------+-------+----------+
| id | user_id | user_name | price | the_year |
+----+---------+---------------+-------+----------+
| 1 | 1001 | 路人甲Java | 11.11 | 2017 |
| 2 | 1001 | 路人甲Java | 22.22 | 2018 |
| 3 | 1001 | 路人甲Java | 88.88 | 2018 |
| 4 | 1002 | 刘德华 | 33.33 | 2018 |
| 5 | 1002 | 刘德华 | 12.22 | 2018 |
| 6 | 1002 | 刘德华 | 16.66 | 2018 |
| 7 | 1002 | 刘德华 | 44.44 | 2019 |
| 8 | 1003 | 张学友 | 55.55 | 2018 |
| 9 | 1003 | 张学友 | 66.66 | 2019 |
+----+---------+---------------+-------+----------+
9 rows in set (0.00 sec)
```

> the_year不是分组依据字段，也不是聚合函数。
>
> 1001|88.88|2017中显示的2017不是88.88的年份，**而与前两个字段无关**，显示的是分组中第一条记录1001|11.11|2017的值2017。

```
-- 正确写法：
SELECT
user_id 用户id,
price 最大金额,
the_year 年份
FROM
t_order t1,(SELECT
t.user_id uid, MAX(t.price) pc
FROM
t_order t
GROUP BY t.user_id) t2
WHERE
t1.user_id = t2.uid
AND t1.price = t2.pc;
```

- **建议：在写分组查询的时候，最好按照标准的规范来写，select后面出现的列必须在group by中或者必须使用聚合函数。**
- **Having后只能跟聚合函数和分组依据的字段（GROUP BY后的字段）**
- GROUP BY后跟ORDER BY，排序的字段若不是分组依据的字段，**排序是按照组中该组中第一条记录的该字段进行排序的。**
- 若不用ORDER BY指定排序的依据，则会按照GROUP BY的字段排序

```
SELECT `prod_id`,COUNT(*) AS `prod_num` //显示prod_id和prod_num
FROM products                          //数据源
GROUP BY `prod_id`;                     //从products中按照prod_id分组
```

(2) 单字段分组

(3) 多字段分组

```
mysql> SELECT
user_id 用户id, the_year 年份, COUNT(id) 下单数量
FROM
t_order
GROUP BY user_id , the_year;
+----------+--------+--------------+
| 用户id | 年份 | 下单数量 |
+----------+--------+--------------+
| 1001 | 2017 | 1 |
| 1001 | 2018 | 2 |
| 1002 | 2018 | 3 |
| 1002 | 2019 | 1 |
| 1003 | 2018 | 1 |
| 1003 | 2019 | 1 |
+----------+--------+--------------+
6 rows in set (0.00 sec)
```

(4) 分组前筛选数据——WHERE

WHERE在分组前对行进行过滤

```
SELECT
user_id 用户id, COUNT(id) 下单数量
FROM
t_order t
WHERE
t.the_year = 2018
GROUP BY user_id;
```

> 语句逻辑顺序：
>
> FROM——哪张表
>
> WHERE——分组前过滤数据
>
> GROUP BY——分组
>
> SELECT——显示哪些字段

（5）分组后筛选数据——HAVING

必须搭配GROUP BY使用，以组为单位进行过滤。

HAVING后只可跟**聚合函数、分组依据字段**进行筛选。

WHERE控制那些行进入分组。

```
SELECT
user_id 用户id, COUNT(id) 下单数量
FROM
t_order t
WHERE
t.the_year = 2018
GROUP BY user_id
HAVING count(id)>=2;
```

（6）子句的顺序

SELECT->FROM->WHERE->GROUP BY->HAVING->ORDER BY->LIMIT

```
select 列 from
表名
where [查询条件]
group by [分组表达式]
having [分组过滤条件]
order by [排序条件]
limit [offset,] count;
```

子句	说明	是否必须使用
select	要返回的列或表示式	是
form	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

15. 日期和时间

函数名称	作 用
curdate 和 current_date	两个函数作用相同，返回当前系统的日期值
curtime 和 current_time	两个函数作用相同，返回当前系统的时间值
now 和 sysdate	两个函数作用相同，返回当前系统的日期和时间值
unix_timestamp	获取UNIX时间戳函数，返回一个以 UNIX 时间戳为基础的无符号整数
from_unixtime	将 UNIX 时间戳转换为时间格式，与UNIX_TIMESTAMP互为反函数
month	获取指定日期中的月份
monthname	获取指定日期中的月份英文名称
dayname	获取指定日期对应的星期几的英文名称
dayofweek	获取指定日期是一周中是第几天，返回值范围是1~7,1=周日
week	获取指定日期是一年中的第几周，返回值的范围是否为 0∽52 或 1∽53
dayofyear	获取指定日期是一年中的第几天，返回值范围是1~366
dayofmonth	获取指定日期是一个月中是第几天，返回值范围是1~31
year	获取年份，返回值范围是 1970∽2069
time_to_sec	将时间参数转换为秒数
sec_to_time	将秒数转换为时间，与TIME_TO_SEC 互为反函数
date_add 和 adddate	两个函数功能相同，都是向日期添加指定的时间间隔
date_sub 和 subdate	两个函数功能相同，都是向日期减去指定的时间间隔
addtime	时间加法运算，在原始时间上添加指定的时间
subtime	时间减法运算，在原始时间上减去指定的时间
datediff	获取两个日期之间间隔，返回参数 1 减去参数 2 的值
date_format	格式化指定的日期，根据参数返回指定格式的值
weekday	获取指定日期在一周内的对应的工作日索引

（1）unix_timestamp——获取UNIX时间戳

unix时间戳

> UNIX_TIMESTAMP(date) 若无参数调用，返回一个32bit无符号整数类型的 UNIX 时间戳（'1970-01-01
> 00:00:00'GMT之后的秒数）。

（2）from_unixtime——时间戳转日期

FROM_UNIXTIME(unix_timestamp[,format]) 函数把 UNIX 时间戳转换为普通格式的日期时间值，与 UNIX_TIMESTAMP () 函数互为反函数。
有2个参数：
unix_timestamp：时间戳（秒）
format：要转化的格式 比如“”%Y-%m-%d“” 这样格式化之后的时间就是 2017-11-30
可以有的形式：

格式	说明
%M	月名字(January ~ December)
%W	星期名字(Sunday ~ Saturday)
%D	有英语前缀的月份的日期(1st, 2nd, 3rd, 等等)
%Y	年, 数字, 4 位
%y	年, 数字, 2 位
%a	缩写的星期名字(Sun ~ Sat)
%d	月份中的天数, 数字(00 ~ 31)
%e	月份中的天数, 数字(0 ~ 31)
%m	月, 数字(01 ~ 12)
%c	月, 数字(1 ~ 12)
%b	缩写的月份名字(Jan ~ Dec)
%j	一年中的天数(001 ~ 366)
%H	小时(00 ~ 23)
%k	小时(0 ~ 23)
%h	小时(01 ~ 12) a
%I（i的大写）	小时(01 ~ 12) Ja
%l（L的小写）	小时(1 ~ 12)
%i	分钟, 数字(00 ~ 59)
%r	时间,12 小时(hh:mm:ss [AP]M) 号
%T	时间,24 小时(hh:mm:ss)
%S	秒(00 ~ 59)
%s	秒(00 ~ 59)
%p	AM或PM
%W	一个星期中的天数英文名称(Sunday~Saturday)
%w	一个星期中的天数(0=Sunday ~ 6=Saturday)
%U	星期(0 ~ 52), 这里星期天是星期的第一天
%u	星期(0 ~ 52), 这里星期一是星期的第一天
%%	输出%

（3）month——获取指定日期的月份

MONTH(date) 函数返回指定 date 对应的月份，范围为 1 ~ 12

```
mysql> select month('2017-12-15'),month(now());
+---------------------+--------------+
| month('2017-12-15') | month(now()) |
+---------------------+--------------+
| 12 | 9 |
+---------------------+--------------+
```

（4）monthname——获取指定日期月份的英文名称

MONTHNAME(date) 函数返回日期 date 对应月份的英文全名

```
mysql> select monthname('2017-12-15'),monthname(now());
+-------------------------+------------------+
| monthname('2017-12-15') | monthname(now()) |
+-------------------------+------------------+
| December | September |
+-------------------------+------------------+
```

（5）dayname——获取指定日期的星期名称

```
mysql> select now(),dayname(now());
+---------------------+----------------+
| now() | dayname(now()) |
+---------------------+----------------+
| 2019-09-17 17:13:08 | Tuesday |
+---------------------+----------------+
```

（6）time_to_sec——将时间转换为秒值

TIME_TO_SEC(time) 函数返回将参数 time 转换为秒数的时间值，转换公式为**“小时 ×3600+ 分钟 ×60+ 秒”**。

```
select time_to_sec('15:15:15'),now(),time_to_sec(now());
+--------------------------+---------------------+--------------------+
| time_to_sec('15:15:15') | now() | time_to_sec(now()) |
+--------------------------+---------------------+--------------------+
| 54915 | 2019-09-17 17:30:44 | 63044 |
+--------------------------+---------------------+--------------------+
```

（7）sec_to_time——将秒值转换为时间格式

```
mysql> select sec_to_time(100),sec_to_time(10000);
+------------------+--------------------+
| sec_to_time(100) | sec_to_time(10000) |
+------------------+--------------------+
| 00:01:40 | 02:46:40 |
+------------------+--------------------+
```

（8）date_format——格式化指定的日期

DATE_FORMAT(date，format) 函数是根据 format 指定的格式显示 date 值。
DATE_FORMAT() 函数接受两个参数：
date：是要格式化的有效日期值format：是由预定义的说明符组成的格式字符串，每个说明符前面都有一个百分比字符(%)。
format：格式和上面的函数 from_unixtime 中的format一样，可以参考上面的。

```
mysql> select date_format('2017-11-30','%Y%m%d') as col0,now() as col1,
date_format(now(),'%Y%m%d%H%i%s') as col2;
+----------+---------------------+----------------+
| col0 | col1 | col2 |
+----------+---------------------+----------------+
| 20171130 | 2019-09-17 17:56:12 | 20190917175612 |
+----------+---------------------+----------------+
```

16.并集——UNION

https://blog.csdn.net/mine_song/article/details/70184072

https://www.jb51.net/article/65696.htm

MySql只支持Union(并集)集合运算，好像也是4.0以后才有的；但是对于交集Intersect、差集Except，就没有实现了。

一般的解决方案用in和not in来解决，小量数据还可以，但数据量大了效率就很低了

- UNION

  连接多条SELECT语句，将查询出来的多条结果放在一张表上

  若有重复的记录，则去重

- UNION ALL

  不去重

- 连接的语句最好用括号括起来，括号内部的ORDER BY / LIMIT作用于括号内的SELECT语句。否则ORDER BY/LIMIT语句放在最后一条SELECT后，作用的是整个表，而不是最后一条SELECT查询到的表。见T2-8。

17.约束

18.子查询（重点）

https://www.cnblogs.com/xiaoxi/p/6734025.html

https://blog.csdn.net/weixin_35782148/article/details/113089077

https://www.cnblogs.com/zhuiluoyu/p/5822481.html

一个查询的结果作为另一个查询的条件

查询可以嵌套，子句中的查询叫子查询，相对的是外查询/主查询。

嵌套查询，先执行子查询，再执行主查询，从内到外。

子查询用括号。

一般在子查询中，程序先运行在嵌套在最内层的语句，再运行外层。因此在写子查询语句时，可以先测试下内层的子查询语句是否输出了想要的内容，再一层层往外测试，增加子查询正确率。否则多层的嵌套使语句可读性很低。

- 分类

  标量子查询：返回单一值的标量，最简单的形式。

  列子查询：返回的结果集是 N 行一列。

  行子查询：返回的结果集是一行 N 列。

  表子查询：返回的结果集是 N 行 N 列。

- 操作符

  可以使用的操作符：= > < >= <= <> ANY IN SOME ALL EXISTS

（1）子查询作为过滤条件

子查询的结果是一个值

SELECT 查询字段 FROM 表 **WHERE 字段=（子查询）；**

子查询结果是多行单列的时候

SELECT 查询字段 FROM 表 **WHERE 字段 IN （子查询）**

子查询的结果是多行多列

子查询结果是多行多列的临时表，肯定接在FROM后

SELECT 查询字段 FROM （子查询） **表别名 WHERE 条件**

子查询作为表必须取别名，否则这张表没有名称则无法访问表中的字段 。注意：用临时表的别名引用字段时，反引号括住字段名，临时表名不要括住。

```
 SELECT `goods_id`,`goods_name`,`cat_id`,`shop_price`,tmp.`total` FROM
(SELECT `goods_id`,`goods_name`,`cat_id`,`shop_price`,COUNT(`goods_id`) AS
'total' FROM goods ORDER BY `cat_id` ASC,`goods_id` DESC) AS tmp
GROUP BY `cat_id`;
```

```
mysql> #子查询结果
mysql>
mysql> SELECT goods_id,goods_name,cat_id,shop_price FROM goods ORDER BY cat_id ASC,goods_id DESC;
+----------+-------------------------------+--------+------------+
| goods_id | goods_name                    | cat_id | shop_price |
+----------+-------------------------------+--------+------------+
|       16 | 恒基伟业G101                  |      2 |     823.33 |
|       32 | 诺基亚N85                     |      3 |    3010.00 |
|       31 | 摩托罗拉E8                    |      3 |    1337.00 |
|       24 | P806                          |      3 |    2000.00 |
|       22 | 多普达Touch HD                |      3 |    5999.00 |
|       21 | 金立 A30                      |      3 |    2000.00 |
|       20 | 三星BC01                      |      3 |     280.00 |
|       19 | 三星SGH-F258                  |      3 |     858.00 |
|       17 | 夏新N7                        |      3 |    2300.00 |
|       15 | 摩托罗拉A810                  |      3 |     788.00 |
|       13 | 诺基亚5320 XpressMusic        |      3 |    1311.00 |
|       12 | 摩托罗拉A810                  |      3 |     983.00 |
|       11 | 索爱C702c                     |      3 |    1300.00 |
|       10 | 索爱C702c                     |      3 |    1328.00 |
|        9 | 诺基亚E66                     |      3 |    2298.00 |
|        8 | 飞利浦9@9v                    |      3 |     399.00 |
|       18 | 夏新T5                        |      4 |    2878.00 |
|       14 | 诺基亚5800XM                  |      4 |    2625.00 |
|        1 | KD876                         |      4 |    1388.00 |
|       23 | 诺基亚N96                     |      5 |    3700.00 |
|        7 | 诺基亚N85原装立体声耳机HS-82  |      8 |     100.00 |
|        4 | 诺基亚N85原装充电器           |      8 |      58.00 |
|        3 | 诺基亚原装5800耳机            |      8 |      68.00 |
|        6 | 胜创KINGMAX内存卡             |     11 |      42.00 |
|        5 | 索爱原装M2卡读卡器            |     11 |      20.00 |
|       26 | 小灵通/固话20元充值卡         |     13 |      19.00 |
|       25 | 小灵通/固话50元充值卡         |     13 |      48.00 |
|       30 | 移动20元充值卡                |     14 |      18.00 |
|       29 | 移动100元充值卡               |     14 |      90.00 |
|       28 | 联通50元充值卡                |     15 |      45.00 |
|       27 | 联通100元充值卡               |     15 |      95.00 |
+----------+-------------------------------+--------+------------+
31 rows in set (0.00 sec)
```

```
mysql> #使用from子查询，子查询结果集相当于一张临时表
mysql>
mysql> SELECT goods_id,goods_name,cat_id,shop_price FROM
    ->     (SELECT goods_id,goods_name,cat_id,shop_price FROM goods ORDER BY cat_id ASC,goods_id DESC) AS tmp
    -> GROUP BY cat_id;
+----------+-------------------------------+--------+------------+
| goods_id | goods_name                    | cat_id | shop_price |
+----------+-------------------------------+--------+------------+
|       16 | 恒基伟业G101                  |      2 |     823.33 |
|       32 | 诺基亚N85                     |      3 |    3010.00 |
|       18 | 夏新T5                        |      4 |    2878.00 |
|       23 | 诺基亚N96                     |      5 |    3700.00 |
|        7 | 诺基亚N85原装立体声耳机HS-82  |      8 |     100.00 |
|        6 | 胜创KINGMAX内存卡             |     11 |      42.00 |
|       26 | 小灵通/固话20元充值卡         |     13 |      19.00 |
|       30 | 移动20元充值卡                |     14 |      18.00 |
|       28 | 联通50元充值卡                |     15 |      45.00 |
+----------+-------------------------------+--------+------------+
9 rows in set (0.00 sec)
```

- Q：主查询中未使用聚合函数，却使用GROUP BY。按照cat_id分组，查询goods_id,goods_name,cat_id,shop_price等字段，为什么每个cat_id只显示一条记录（cat_id分组下的第一条记录）？

  不规范的用法？？？

with as将临时表作为变量再操作

MySQL8.0以上版本的特性，版本低于8.0使用该语法将会显示错误

```sql
SELECT a.name FROM
(
SELECT name, gender FROM employee
)AS t1
WHERE t1.gender='man';

-- 使用with as可写成:
WITH t1 AS(
SELECT name,gender FROM emplyee
)
SELECT t1.gender='man';
```

```sql
WITH t1 AS
(
SELECT
LineNO,
DeviceStatus_001 +
DeviceStatus_002 +
DeviceStatus_003 +
DeviceStatus_004 +
DeviceStatus_005 +
DeviceStatus_006 +
DeviceStatus_007 +
DeviceStatus_008 +
DeviceStatus_009 +
DeviceStatus_010 +
DeviceStatus_011 +
DeviceStatus_012 +
DeviceStatus_013 +
DeviceStatus_101 +
DeviceStatus_102 +
DeviceStatus_103 +
DeviceStatus_104 +
DeviceStatus_105 AS `DeviceTotalNum`
FROM device_config
)
SELECT t1.LineNO,t2.LineName,t1.DeviceTotalNum
FROM t1 INNER JOIN device AS t2
ON t1.LineNO=t2.LineNO;
```

（2）子查询作为计算字段

子查询得到的结果还可以作为SELECT显示的字段。

T1-3

（3）多行操作符

ANY（SOME）, ALL, IN, EXISTS

> 子查询放在小括号内。
>
> 标量子查询，一般搭配着单行单列操作符使用 >、<、>=、<=、=、<>、!=
>
> 列子查询，一般搭配着多行操作符使用

> **in(not in)**——列表中的“任意一个”
> **any或者some**——和子查询返回的“某一个值”比较，比如**a>some(10,20,30)**，a大于子查询中任意一个即可，a大于子查询中最小值即可，等同于a>min(10,20,30)。
> **all**——和子查询返回的“所有值”比较，比如**a>all(10,20,30)**，a大于子查询中所有值，换句话说，a大于子查询中最大值即可满足查询条件，等同于a>max(10,20,30);

```
/*返回location_id是1400或1700的部门中的所有员工姓名*/
/*方式1*/
/*①查询location_id是1400或1700的部门编号*/
SELECT DISTINCT department_id
FROM departments
WHERE location_id IN (1400, 1700);
/*②查询员工姓名，要求部门是①列表中的某一个*/
SELECT a.last_name
FROM employees a
WHERE a.department_id IN (SELECT DISTINCT department_id
FROM departments
WHERE location_id IN (1400, 1700));
/*方式2：使用any实现*/
SELECT a.last_name
FROM employees a
WHERE a.department_id = ANY (SELECT DISTINCT department_id
FROM departments
WHERE location_id IN (1400, 1700));
/*拓展，下面与not in等价*/
SELECT a.last_name
FROM employees a
WHERE a.department_id <> ALL (SELECT DISTINCT department_id
FROM departments
WHERE location_id IN (1400, 1700));
```

(4) NULL的坑

=、<>、<、>、>=、<=、IN、NOT IN过滤NULL时无法查询出值为NULL的记录！

判断NULL只能用IS NULL、IS NOT NULL

```
mysql> select 1>=NULL;
+---------+
| 1>=NULL |
+---------+
| NULL |
+---------+
1 row in set (0.00 sec)
-- ////////////////////////////////////////////////////////
mysql> select 1 in (null),1 not in (null),null in (null),null not in (null);
+-------------+-----------------+----------------+--------------------+
| 1 in (null) | 1 not in (null) | null in (null) | null not in (null) |
+-------------+-----------------+----------------+--------------------+
| NULL | NULL | NULL | NULL |
+-------------+-----------------+----------------+--------------------+
1 row in set (0.00 sec)
-- ////////////////////////////////////////////////////////
mysql> select 1=any(select null),null=any(select null);
+--------------------+----------------------+
| 1=any(select null) | null=any(select null) |
```

```
+------------------+----------------------+
| NULL | NULL |
+------------------+----------------------+
```

IN后跟着NULL

```
mysql> select * from test1;
+------+------+
| a | b |
+------+------+
| 1 | 1 |
| 1 | NULL |
| NULL | NULL |
+------+------+
3 rows in set (0.00 sec)
-- IN后跟着NULL，查出的都是空集
mysql> select * from test1 where a in (null);
Empty set (0.00 sec)
-- IN后除了跟着NULL还有别的值，NULL查不出，其他的值可以查询出来
mysql> select * from test1 where a in (null,1);
+------+------+
| a | b |
+------+------+
| 1 | 1 |
| 1 | NULL |
+------+------+
2 rows in set (0.00 sec)
```

NOT IN后跟着NULL

```
mysql> select * from test1 where a not in (null);
Empty set (0.00 sec)
-- ////////////////////////////////////////////////////
mysql> select * from test1 where a not in (2);
+------+------+
| a | b |
+------+------+
| 1 | 1 |
| 1 | NULL |
+------+------+
2 rows in set (0.00 sec)
-- NOT IN后跟着的除了NULL还有别的值，则都查不出来
mysql> select * from test1 where a not in (null,2);
Empty set (0.00 sec)
```

COUNT后跟着NULL

COUNT(col)——统计不包含字段col中值为NULL的行

COUNT(*)——统计包含字段col中值为NULL的行

主键的字段会自动设为NOT NULL，故添加记录时主键的值不能是NULL

```
CREATE TABLE `test3` (
`a` int(11) NOT NULL,
`b` int(11) DEFAULT NULL,
PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
-- 插入数据时主键不能是NULL
mysql> insert into test3 values (null,1);
ERROR 1048 (23000): Column 'a' cannot be null
```

因为NULL的情况确实比较难以处理，容易出错，最有效的方法就是避免使用NULL。所以，强烈建议创建字段的时候字段不允许为NULL，设置一个默认值。

> 任何值和NULL使用运算符（>、<、>=、<=、!=、<>）或者（in、not in、any/some、all），返回值都为NULL
> 当IN和NULL比较时，无法查询出为NULL的记录
> 当NOT IN 后面有NULL值时，不论什么情况下，整个sql的查询结果都为空
> 判断是否为空只能用IS NULL、IS NOT NULL
> count(字段)无法统计字段为NULL的值，count(*)可以统计值为null的行
> 当字段为主键的时候，字段会自动设置为not null
> NULL导致的坑让人防不胜防，强烈建议创建字段的时候字段不允许为NULL，给个默认值

(5)练习

https://blog.csdn.net/weixin_45664854/article/details/105498760

T1

```
drop database if EXISTS mydb;
create database mydb;
use mydb;
-- 子查询
CREATE TABLE emp(
    empno       INT,
    ename       VARCHAR(50),
    job     VARCHAR(50),
    mgr     INT,
    hiredate    DATE,
    sal     DECIMAL(7,2),       -- 2表示小数的位数为2,7表示整数位数+小数位数不能超过7
    comm        DECIMAL(7,2),
    deptno      INT
) ;
INSERT INTO emp VALUES(7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL,20);
INSERT INTO emp VALUES(7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300,30);
INSERT INTO emp VALUES(7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500,30);
INSERT INTO emp VALUES(7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL,20);
INSERT INTO emp VALUES(7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400,30);
INSERT INTO emp VALUES(7698,'BLAKE','MANAGER',7839,'1981-05-01',2850,NULL,30);
INSERT INTO emp VALUES(7782,'CLARK','MANAGER',7839,'1981-06-09',2450,NULL,10);
INSERT INTO emp VALUES(7788,'SCOTT','ANALYST',7566,'1987-04-19',3000,NULL,20);
INSERT INTO emp VALUES(7839,'KING','PRESIDENT',NULL,'1981-11-17',5000,NULL,10);
INSERT INTO emp VALUES(7844,'TURNER','SALESMAN',7698,'1981-09-08',1500,0,30);
INSERT INTO emp VALUES(7876,'ADAMS','CLERK',7788,'1987-05-23',1100,NULL,20);
INSERT INTO emp VALUES(7900,'JAMES','CLERK',7698,'1981-12-03',950,NULL,30);
```

```
INSERT INTO emp VALUES(7902,'FORD','ANALYST',7566,'1981-12-03',3000,NULL,20);
INSERT INTO emp VALUES(7934,'MILLER','CLERK',7782,'1982-01-23',1300,NULL,10);
select * from emp;

CREATE TABLE dept(
    deptno      INT,
    dname       VARCHAR(14),
    loc         VARCHAR(13)
);
INSERT INTO dept VALUES(10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO dept VALUES(20, 'RESEARCH', 'DALLAS');
INSERT INTO dept VALUES(30, 'SALES', 'CHICAGO');
INSERT INTO dept VALUES(40, 'OPERATIONS', 'BOSTON');
select * from dept;

select * from emp;
```

```
-- 1.查询出高于10号部门的平均工资 的员工信息
SELECT * FROM
emp
WHERE `sal`>
(SELECT AVG(`sal`) FROM emp WHERE `deptno`=10);
```

```
-- 2.查询出比10号部门任何员工薪资高的员工信息
SELECT * FROM
emp
WHERE `sal`>(SELECT MAX(`sal`) FROM emp WHERE `deptno`=10);
```

```
-- 3.获取员工的名字和部门的名字
SELECT ename,(SELECT dname FROM dept WHERE dept.deptno=emp.deptno) AS
`dept_name`
FROM emp
ORDER BY ename;
```

```
-- 4.查询与SCOTT同一个部门的员工
SELECT `ename` FROM
emp
WHERE `deptno`=
(SELECT `deptno` FROM emp WHERE `ename`='SCOTT');
```

```
-- 5.查询工作和工资与MARTIN完全相同的员工信息
SELECT *
FROM emp
WHERE job=(SELECT job FROM emp WHERE ename='MARTIN')
AND
sal=(SELECT sal FROM emp WHERE ename='MARTIN')
AND ename!='MARTIN';
```

T2

https://zhuanlan.zhihu.com/p/345457364

https://zhuanlan.zhihu.com/p/345459843

多表，记得把各个表的字段列出，标明表之间相同的字段。

```
-- 创建库
CREATE DATABASE SQL50;
-- 创建4张表
CREATE TABLE SC(Sid VARCHAR(10),Cid VARCHAR(10),score DECIMAL(18,1));
CREATE TABLE Student(Sid VARCHAR(10),Sname VARCHAR(10),Sage DATE,Ssex
VARCHAR(10));
CREATE TABLE Course (Cid VARCHAR(10),Cname VARCHAR(10),Tid VARCHAR(10));
CREATE TABLE Teacher(Tid VARCHAR(10),Tname VARCHAR(18));
-- 注意：这里有个DECIMAL(18,1)数据类型，意思就是整数位最大18位，小数位1位。

-- 插入数据
-- 向[学生表]插入数据
INSERT INTO Student VALUES('0001' , '李小龙' , '1995/1/2' , '男');
INSERT INTO Student VALUES('0002' , '成龙' ,   '2006/12/13' , '男');
INSERT INTO Student VALUES('0003' , '周星驰' , '1996/4/28' , '男');
INSERT INTO Student VALUES('0004' , '周润发' , '2001/7/16' , '男');
INSERT INTO Student VALUES('0005' , '李连杰' , '2018/5/26' , '男');
INSERT INTO Student VALUES('0006' , '梁朝伟' , '2020/4/16' , '男');
INSERT INTO Student VALUES('0007' , '张国荣' , '2018/7/21' , '男');
INSERT INTO Student VALUES('0008' , '梁家辉' , '1995/9/13' , '男');
INSERT INTO Student VALUES('0009' , '刘德华' , '1990/10/7' , '男');
INSERT INTO Student VALUES('0010' , '刘青云' , '2006/6/2' , '男');
INSERT INTO Student VALUES('0011' , '邱淑贞' , '1990/1/8' , '女');
INSERT INTO Student VALUES('0012' , '张敏' ,   '2017/12/7' , '女');
INSERT INTO Student VALUES('0013' , '朱茵' ,   '2015/11/6' , '女');
INSERT INTO Student VALUES('0014' , '关之琳' , '2002/11/18' , '女');
INSERT INTO Student VALUES('0015' , '林青霞' , '2014/2/14' , '女');
INSERT INTO Student VALUES('0016' , '王祖贤' , '2011/5/25' , '女');
INSERT INTO Student VALUES('0017' , '钟楚红' , '2006/12/10' , '女');
INSERT INTO Student VALUES('0018' , '黎姿' ,   '2000/12/25' , '女');
INSERT INTO Student VALUES('0019' , '李嘉欣' , '1992/2/19' , '女');
INSERT INTO Student VALUES('0020' , '周慧敏' , '2010/8/15' , '女');

-- 向[课程表]插入数据
INSERT INTO Course VALUES('s1' , '语文' , '0001');
INSERT INTO Course VALUES('s2' , '数学' , '0002');
INSERT INTO Course VALUES('s3' , '英语' , '0003');

-- 向[教师表]插入数据
INSERT INTO Teacher VALUES('0001' , '数据分析王子');
INSERT INTO Teacher VALUES('0002' , '刘亦菲');
INSERT INTO Teacher VALUES('0003' , '彭于晏');

-- 向[成绩表]插入数据
INSERT INTO SC VALUES('0001' , 's1' ,81);
INSERT INTO SC VALUES('0001' , 's2' ,67);
INSERT INTO SC VALUES('0001' , 's3' ,88);
INSERT INTO SC VALUES('0002' , 's1' ,77);
INSERT INTO SC VALUES('0002' , 's3' ,86);
INSERT INTO SC VALUES('0003' , 's1' ,81);
```

```
INSERT INTO SC VALUES('0003' , 's2' ,70);
INSERT INTO SC VALUES('0003' , 's3' ,91);
INSERT INTO SC VALUES('0004' , 's1' ,62);
INSERT INTO SC VALUES('0004' , 's2' ,93);
INSERT INTO SC VALUES('0004' , 's3' ,95);
INSERT INTO SC VALUES('0005' , 's1' ,72);
INSERT INTO SC VALUES('0005' , 's2' ,52);
INSERT INTO SC VALUES('0005' , 's3' ,61);
INSERT INTO SC VALUES('0006' , 's1' ,80);
INSERT INTO SC VALUES('0006' , 's2' ,86);
INSERT INTO SC VALUES('0007' , 's1' ,100);
INSERT INTO SC VALUES('0007' , 's2' ,62);
INSERT INTO SC VALUES('0007' , 's3' ,72);
INSERT INTO SC VALUES('0008' , 's1' ,88);
INSERT INTO SC VALUES('0008' , 's2' ,72);
INSERT INTO SC VALUES('0008' , 's3' ,51);
INSERT INTO SC VALUES('0009' , 's2' ,56);
INSERT INTO SC VALUES('0009' , 's3' ,52);
INSERT INTO SC VALUES('0010' , 's1' ,84);
INSERT INTO SC VALUES('0010' , 's2' ,61);
INSERT INTO SC VALUES('0010' , 's3' ,75);
INSERT INTO SC VALUES('0011' , 's2' ,82);
INSERT INTO SC VALUES('0011' , 's3' ,92);
INSERT INTO SC VALUES('0012' , 's1' ,97);
INSERT INTO SC VALUES('0012' , 's2' ,88);
INSERT INTO SC VALUES('0012' , 's3' ,64);
INSERT INTO SC VALUES('0013' , 's1' ,75);
INSERT INTO SC VALUES('0013' , 's2' ,97);
INSERT INTO SC VALUES('0013' , 's3' ,67);
INSERT INTO SC VALUES('0014' , 's1' ,83);
INSERT INTO SC VALUES('0014' , 's2' ,86);
INSERT INTO SC VALUES('0014' , 's3' ,96);
INSERT INTO SC VALUES('0015' , 's2' ,86);
INSERT INTO SC VALUES('0015' , 's3' ,57);
INSERT INTO SC VALUES('0016' , 's1' ,75);
INSERT INTO SC VALUES('0016' , 's3' ,99);
INSERT INTO SC VALUES('0017' , 's1' ,56);
INSERT INTO SC VALUES('0017' , 's2' ,98);
INSERT INTO SC VALUES('0017' , 's3' ,80);
INSERT INTO SC VALUES('0018' , 's1' ,53);
INSERT INTO SC VALUES('0018' , 's2' ,51);
INSERT INTO SC VALUES('0018' , 's3' ,69);
INSERT INTO SC VALUES('0019' , 's1' ,97);
INSERT INTO SC VALUES('0019' , 's2' ,99);
INSERT INTO SC VALUES('0019' , 's3' ,64);
INSERT INTO SC VALUES('0020' , 's1' ,81);
INSERT INTO SC VALUES('0020' , 's2' ,52);
INSERT INTO SC VALUES('0020' , 's3' ,62);
```

```
-- 1.查询名字中含有「龙」字的学生信息
SELECT *
FROM student
WHERE `Sname` LIKE '%龙%';
```

```sql
-- 2.查询「数」姓的所有老师的信息
SELECT *
FROM teacher
WHERE `Tname` LIKE '数%';
```

```sql
-- 3.查询男生、女生人数
SELECT `Ssex` AS '性别',COUNT(*)
FROM student
GROUP BY `Ssex`
ORDER BY `Ssex`;
```

```sql
-- 4.查询2000年(不含2000)后出生的学生名单
SELECT `Sname`
FROM student
WHERE YEAR(`Sage`)>2000;
```

```sql
-- 5.查询同名同姓学生名单，并统计同名人数
SELECT `Sname`,tmp.`total`
FROM (SELECT *,COUNT(`Sname`) AS 'total' FROM student GROUP BY `Sname`) tmp
WHERE tmp.`total`>1;
-- 解法2：
SELECT `Sname`,COUNT(*)
FROM student
GROUP BY `Sname`
HAVING COUNT(*)>1;
```

```sql
-- 6.查询每门课程选修人数
SELECT (SELECT `Cname` FROM course WHERE course.Cid=sc.Cid) AS '课程名
称',COUNT(`Sid`)
FROM sc
GROUP BY `Cid`;
-- 解法2：
SELECT course.Cname,COUNT(*)
FROM course INNER JOIN sc ON sc.Cid=course.Cid
GROUP BY course.Cname
ORDER BY course.Cname;
```

```sql
-- 7.查询每门课程的平均成绩
SELECT AVG(score),(SELECT `Cname` FROM course WHERE course.`Cid`=sc.`Cid`)
FROM sc
GROUP BY `Cid`;

-- 解法2：
SELECT course.Cname,AVG(sc.score)
FROM course INNER JOIN sc ON course.Cid=sc.Cid
GROUP BY course.Cname
ORDER BY course.Cid;
```

```sql
-- 8.查询每门成绩最好的前3名
(SELECT *
FROM sc
WHERE `Cid`=(SELECT `Cid` FROM course WHERE `Cname`='语文')
ORDER BY `score` DESC
```

```
LIMIT 3)
UNION
(SELECT *
FROM sc
WHERE `Cid`=(SELECT `Cid` FROM course WHERE `Cname`='数学')
ORDER BY `score` DESC
LIMIT 3)
UNION
(SELECT *
FROM sc
WHERE `Cid`=(SELECT `Cid` FROM course WHERE `Cname`='英语')
ORDER BY `score` DESC
LIMIT 3);

-- UNION组合的几个SELECT语句最好用括号括起来，若不括起来，LIMIT、ORDER BY语句在最后一个SELECT语句后，会影响整个查询结果，而不仅仅是最后一个SELECT。

-- 下面的代码中，最后的ORDER BY将覆盖第一个SELECT中按照Cid升序排序，整个表降序排序；LIMIT 3将整个表只显示3行
(SELECT *
FROM sc
WHERE `Cid`=(SELECT `Cid` FROM course WHERE `Cname`='语文')
ORDER BY `Cid`
LIMIT 3)
UNION
(SELECT *
FROM sc
WHERE `Cid`=(SELECT `Cid` FROM course WHERE `Cname`='数学')
LIMIT 3)
UNION
SELECT *
FROM sc
WHERE `Cid`=(SELECT `Cid` FROM course WHERE `Cname`='英语')
ORDER BY `Cid` DESC
LIMIT 3;
```

```
-- 9.查询选修过任一课程的学生的学生信息
```

```
-- 10.查询至少选修两门课程的学生学号
SELECT `Sid`
FROM sc
GROUP BY `Sid`
HAVING COUNT(`Cid`)>=2;
```

```
-- 11.查询出只选修两门课程的学生学号和姓名
SELECT `Sid`,`Sname`
FROM student
WHERE `Sid`IN
(SELECT `Sid`
FROM sc
GROUP BY `Sid`
HAVING COUNT(`Cid`)=2);

-- 解法2：
```

```
SELECT tmp.Sid AS `学生学号`,student.Sname AS `学生姓名`,course.Cname AS `课程名称`
FROM student INNER JOIN
(SELECT sc.Sid,sc.Cid,COUNT(sc.Cid) FROM sc GROUP BY sc.Sid HAVING
COUNT(sc.Cid)=2)AS tmp
ON student.Sid=tmp.Sid INNER JOIN course ON course.Cid= tmp.Cid;
```

```
-- 12.查询出选修了全部课程的学生信息
SELECT *
FROM student
WHERE `Sid` IN (SELECT `Sid` FROM sc GROUP BY `Sid` HAVING COUNT(`Cid`)=(SELECT
COUNT(`Cid`) FROM course));

-- 解法2：
SELECT * FROM student INNER JOIN
(SELECT sc.Sid FROM sc GROUP BY sc.Sid HAVING COUNT(sc.Cid)=(SELECT
COUNT(course.Cid) FROM course)) AS tmp
ON student.Sid=tmp.Sid
```

```
-- 13.查询没有选修全部课程的同学的信息
-- 解法1：
SELECT *
FROM student
WHERE `Sid` IN (SELECT `Sid` FROM sc GROUP BY `Sid` HAVING COUNT(`Cid`)<(SELECT
COUNT(`Cid`) FROM course));
-- 解法2：
SELECT *
FROM student
WHERE `Sid` NOT IN (SELECT `Sid` FROM sc GROUP BY `Sid` HAVING COUNT(`Cid`)=
(SELECT COUNT(`Cid`) FROM course));
```

```
-- 14.查询各学生的年龄，只按年份来算
SELECT `Sname` AS '姓名',YEAR(NOW())-YEAR(`Sage`) AS '年龄'
FROM student;
```

```
-- 15.查询所有学生的姓名、课程名称和分数
```

19.联结查询（重点）

https://zhuanlan.zhihu.com/p/145679471

https://zhuanlan.zhihu.com/p/58108883

https://blog.csdn.net/wheredata/article/details/87191983

https://www.cnblogs.com/guokaifeng/p/11192266.html

https://blog.csdn.net/qq_36078992/article/details/106005655

> 1. 先确定数据要用到哪些表。
> 2. 将多个表先通过笛卡尔积变成一个表。
> 3. 然后去除不符合逻辑的数据（根据两个表的关系去掉），内联/左联/自联。
> 4. 最后当做是一个虚拟表一样来加上条件即可。

（1）笛卡尔积——叉联结+WHERE过滤

叉联结形成笛卡尔积，有很多无意义行，通过WHERE过滤出指定条件的行

是全相乘效率低，全相乘会在内存中生成一个非常大的数据(临时表)，因为有很多不必要的数据。

全相乘不能好好的利用索引，因为全相乘生成一张临时表，临时表里是没有索引的，大大降低了查询效率。

联结要指定条件，否则会返回笛卡尔积，新表列数=表1列数*表2列数。

```
drop table if exists test1;
create table test1(
a int
);
drop table if exists test2;
create table test2(
b int
);
insert into test1 values (1),(2),(3);
insert into test2 values (3),(4),(5)
```

```
mysql> select * from test1;
+------+
| a |
+------+
| 1 |
| 2 |
| 3 |
+------+
3 rows in set (0.00 sec)
mysql> select * from test2;
+------+
| b |
+------+
| 3 |
| 4 |
| 5 |
+------+
3 rows in set (0.00 sec)
```

```
mysql> select * from test1 t1,test2 t2;
+------+------+
| a | b |
+------+------+
| 1 | 3 |
| 2 | 3 |
| 3 | 3 |
| 1 | 4 |
| 2 | 4 |
| 3 | 4 |
| 1 | 5 |
| 2 | 5 |
| 3 | 5 |
+------+------+
9 rows in set (0.00 sec)
```

```
mysql> select * from test1 t1,test2 t2 where t1.a = t2.b;
+------+------+
| a | b |
+------+------+
| 3 | 3 |
+------+------+
1 row in set (0.00 sec
```

（2）内联结

内连接相当于在笛卡尔积的基础上加上了连接的条件进行过滤

```
for(Object eleA : A){
    for(Object eleB : B){
        if(连接条件是否为true){
            System.out.print(eleA+","+eleB);
        }
    }
}
```

```
mysql> select t1.emp_name,t2.team_name from t_employee t1 inner join t_team t2
on t1.team_id = t2.id;
+---------------+-----------+
| emp_name | team_name |
+---------------+-----------+
| 路人甲Java | 架构组 |
| 张三 | 测试组 |
| 李四 | java组 |
+---------------+-----------+
```

当没有连接条件ON的时候，内连接上升为笛卡尔积，结果和笛卡尔积相同。

```
mysql> select t1.emp_name,t2.team_name from t_employee t1 inner join t_team t2;
+---------------+-----------+
| emp_name | team_name |
+---------------+-----------+
| 路人甲Java | 架构组 |
| 路人甲Java | 测试组 |
| 路人甲Java | java组 |
| 路人甲Java | 前端组 |
| 张三 | 架构组 |
| 张三 | 测试组 |
| 张三 | java组 |
| 张三 | 前端组 |
| 李四 | 架构组 |
| 李四 | 测试组 |
| 李四 | java组 |
| 李四 | 前端组 |
| 王五 | 架构组 |
| 王五 | 测试组 |
| 王五 | java组 |
| 王五 | 前端组 |
| 赵六 | 架构组 |
| 赵六 | 测试组 |
```

```
| 赵六 | java组 |
| 赵六 | 前端组 |
+---------------+-----------+
```

内连接——INNER JOIN和先笛卡尔积再WHERE两种写法

```
-- 查询架构组的员工
select t1.emp_name,t2.team_name from t_employee t1 inner join t_team t2
on t1.team_id = t2.id and t2.team_name = '架构组';
```

```
select t1.emp_name,t2.team_name from t_employee t1 inner join t_team t2
on t1.team_id = t2.id where t2.team_name = '架构组';
```

```
select t1.emp_name,t2.team_name from t_employee t1, t_team t2 where
t1.team_id = t2.id and t2.team_name = '架构组';
```

建议使用笛卡尔积+WHERE语法，简洁：

```
select 字段 from 表1, 表2, 表3 [where 关联条件];
```

内连接

内连接 (inner join) 使用 `inner join` 关键字连接两张表，并使用 `on` 子句来设置连接条件

语法格式如下：

```
select <字段名> from <tab_name> <tab_name> inner join <tab_name> [on子句]
```

多个表连接时，再 from 后边连续使用 `inner join` 或 `join` 即可

案例操作：

```
# 在 学生表 和 课程表中，查询 学生对应的课程
select s.name c.course_name from student as s inner join course as c on s.course_id = c.id
```

> 注意：当对多个表进行查询时，要在 select 语句后面指定字段是来源自哪一张表
> 语法为 `表名.列名` ，如果表名较长，可以给表设置别名，这样就可以直接在 select 后写 `表的别名.列名`

```
SELECT t1.DeviceNO,t2.DeviceName,
(CASE WHEN t1.DeviceStatus=1 THEN '正常'
WHEN t1.DeviceStatus=0 THEN '异常'
WHEN t1.DeviceStatus=-1 THEN '无效'
END) AS DeviceStatus,
t1.TestingNum,t1.DefectNum,t1.CPUTemperature,t1.CPUUsage,t1.MemoryUsage
FROM device_info AS t1 INNER JOIN device AS t2
ON t1.DeviceNO=t2.DeviceNO
WHERE t1.LineNO='001'
ORDER BY t1.`NO`;
```

多表内联结——需多个INNER JOIN

mysql 在运行是关联指定的每个表以处理联结，这种处理是非常消耗资源的，所以不要联结过多的表，表越多性能下降越厉害

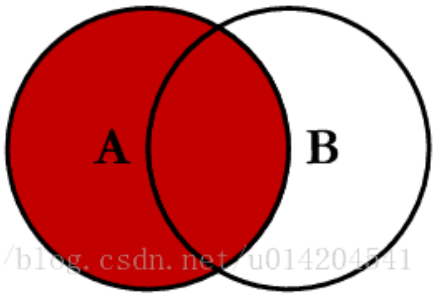
```
select [column_list]
            FROM
            t1 INNER JOIN t2 ON [连接条件1]
            INNER JOIN t3 ON [连接条件2]
            ...
            WHERE where_conditions;
```

```
SELECT t1.`NO`, t2.LineName, t3.DeviceName, t4.FaultName,(CASE WHEN
FaultEnable=1 THEN '使能' WHEN FaultEnable=0 THEN '禁止' END) AS FaultEnable
FROM faults_config AS t1
INNER JOIN productionline AS t2 ON t1.LineNO=t2.LineNO
INNER JOIN device AS t3 ON t1.DeviceNO=t3.DeviceNO
INNER JOIN faults AS t4 ON t1.FaultNO=t4.FaultNO AND t1.DeviceNO=t4.DeviceNO
ORDER BY t1.`NO`;
```

（3）左联结/右联结

https://blog.csdn.net/weixin_39608063/article/details/113187275

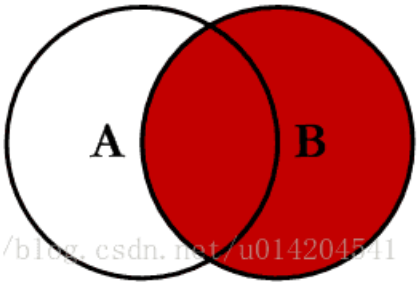
左连接查询达到了同样的效果，但是不会有其它冗余数据，查询速度快，消耗内存小，而且使用了索引。左连接查询效率相比于全相乘的查询效率快了10+倍以上。



实现代码：

```
SELECT  A.PK AS A_PK,A.Value AS A_Value,B.PK AS B_PK,B.Value AS B_Value
FROM table_a A
LEFT JOIN  table_b B
ON A.PK = B.PK;
```

右连接查询跟左连接查询类似，只是**右连接是以右表为主表**，会将右表所有数据查询出来，而左表则根据条件去匹配，如果左表没有满足条件的行，则左边默认显示NULL。左右连接是可以互换的。



实现代码：

```sql
SELECT  A.PK AS A_PK,A.Value AS A_Value,B.PK AS B_PK,B.Value AS B_Value
FROM table_a A
RIGHT JOIN   table_b B
ON A.PK = B.PK;
```

```sql
SELECT LineName,
(CASE WHEN COUNT(FaultTime)>0 THEN 1
WHEN COUNT(FaultTime)=0 THEN 0
END) AS LineStatus
FROM productionline AS t1 LEFT JOIN faults_time AS t2
ON t1.LineNO=t2.LineNO
GROUP BY LineName
ORDER BY t1.`NO`;
```

（4）在联结中使用聚集函数

（5）自联结

一张表看成两张表，要取2个别名。否则字段指定存在歧义。

（6）INNER JOIN和LEFT JOIN的代码模拟

> 上面java代码中两个表的连接查询使用了嵌套循环，外循环每执行一次，内循环的表都会全部遍历一次，如果放到mysql中，就相当于内表（被驱动表）全部扫描了一次（一次全表io读取操作），主表（外循环）如果有n条数据，那么从表就需要全表扫描n次，表的数据是存储在磁盘中，每次全表扫描都需要做io操作，**io操作是最耗时间的**，如果mysql按照上面的java方式实现，那效率肯定很低。
> 那mysql是如何优化的呢?
> msql内部使用了一个**内存缓存空间**，就叫他 join_buffer 吧，先把外循环的数据放到 join_buffer 中，然后对从表进行遍历，从表中取一条数据和 join_buffer 的数据进行比较，然后从表中再取第2条和 join_buffer 数据进行比较，直到从表遍历完成，使用这方方式来减少从表的io扫描次数，当 join_buffer 足够大的时候，大到可以存放主表所有数据，那么从表只需要全表扫描一次（即只需要一次全表io读取操作）。
> mysql中这种方式叫做 **Block Nested Loop**

20. 流程控制语句

（1）条件语句

IF函数

```
IF(expr1, val1, val2)   -- 如果第一个条件为True,则返回第二个参数，否则返回第三个
```

```
select if(author='Felix', 'yes', 'no') as Author from books;
```

IF-ELSE结构——只能使用在BEGIN END之间

```
IF 条件语句1 THEN 语句1;
ELSEIF 条件语句2 THEN 语句2;
...
ELSE 语句n;
END IF;
```

```
/*删除id=7的记录*/
DELETE FROM t_user WHERE id=7;
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc2;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc2(v_id int,v_sex varchar(8),v_name varchar(16),OUT result
TINYINT)
BEGIN
    DECLARE v_count TINYINT DEFAULT 0;/*用来保存user记录的数量*/
    /*根据v_id查询数据放入v_count中*/
    select count(id) into v_count from t_user where id = v_id;
    /*v_count>0表示数据存在，则修改，否则新增*/
    if v_count>0 THEN
        BEGIN
            DECLARE lsex TINYINT;
            select if(v_sex='男',1,2) into lsex;
            update t_user set sex = lsex,name = v_name where id = v_id;
            /*获取update影响行数*/
            select ROW_COUNT() INTO result;
        END;
    else
        BEGIN
            DECLARE lsex TINYINT;
            select if(lsex='男',1,2) into lsex;
            insert into t_user VALUES (v_id,lsex,v_name);
            select 0 into result;
        END;
    END IF;
END $
/*结束符置为;*/
DELIMITER ;
```

```
mysql> SELECT * FROM t_user;
+----+-----+---------------+
| id | sex | name |
+----+-----+---------------+
```

```
| 1 | 1 | 路人甲Java |
| 2 | 1 | 张学友 |
| 3 | 2 | 王祖贤 |
| 4 | 1 | 郭富城 |
| 5 | 2 | 李嘉欣 |
| 6 | 1 | 郭富城 |
+----+-----+---------------+
6 rows in set (0.00 sec)
mysql> CALL proc2(7,'男','黎明',@result);
Query OK, 1 row affected (0.00 sec)
mysql> SELECT @result;
+---------+
| @result |
+---------+
| 0 |
+---------+
1 row in set (0.00 sec)
mysql> SELECT * FROM t_user;
+----+-----+---------------+
| id | sex | name |
+----+-----+---------------+
| 1 | 1 | 路人甲Java |
| 2 | 1 | 张学友 |
| 3 | 2 | 王祖贤 |
| 4 | 1 | 郭富城 |
| 5 | 2 | 李嘉欣 |
| 6 | 1 | 郭富城 |
| 7 | 2 | 黎明 |
+----+-----+---------------+
7 rows in set (0.00 sec)
mysql> CALL proc2(7,'男','梁朝伟',@result);
Query OK, 1 row affected (0.00 sec)
mysql> SELECT @result;
+---------+
| @result |
+---------+
| 1 |
+---------+
1 row in set (0.00 sec)
mysql> SELECT * FROM t_user;
+----+-----+---------------+
| id | sex | name |
+----+-----+---------------+
| 1 | 1 | 路人甲Java |
| 2 | 1 | 张学友 |
| 3 | 2 | 王祖贤 |
| 4 | 1 | 郭富城 |
| 5 | 2 | 李嘉欣 |
| 6 | 1 | 郭富城 |
| 7 | 2 | 梁朝伟 |
+----+-----+---------------+
7 rows in set (0.00 sec)
```

CASE-WHEN-THEN

数据SQL CASE 表达式是一种通用的条件表达式，类似于其它语言中的 if/else 语句。

可以表达多种条件。

- **如果是语句需要加分号**
- ***CASE WHEN*语句中必须要有*ELSE*语句，不能有所有*WHEN*情况都不匹配的情况，否则会报错`1339 Case not found for CASE statement`。若*ELSE*中没有可以可写的，加个*BEGIN END*即可**

> https://stackoverflow.com/questions/7881211/mysql-case-not-found-for-case-statement-on-a-stored-procedure

```
-- 该段SQL报错：1339 Case not found for CASE statement。正确的见22.事务-(8)
CREATE DEFINER=`root`@`localhost` PROCEDURE `p_deleteDevice`(IN ln
VARCHAR(20), IN dn VARCHAR(20))
BEGIN
DECLARE ifAffectedRow TINYINT(1) DEFAULT 1;
DECLARE SQL_FOR_UPDATE_device_config VARCHAR(100);

START TRANSACTION;

SET SQL_FOR_UPDATE_device_config=CONCAT('UPDATE device_config SET
`DeviceStatus_', dn, '`=0 WHERE LineNO=', ln, ';');
SET @sql=SQL_FOR_UPDATE_device_config;
PREPARE stmt FROM @sql;
EXECUTE stmt;
-- 使用PREPARE，获取ROW_COUNT必须要在DEALLOCATE释放sql语句之前
CASE ROW_COUNT()
WHEN 0 THEN SET ifAffectedRow=0;
END CASE;
DEALLOCATE PREPARE stmt;

DELETE FROM device_info WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN SET ifAffectedRow=0;
ELSE ALTER TABLE device_info AUTO_INCREMENT=1;
END CASE;

DELETE FROM device_info_threshold WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN SET ifAffectedRow=0;
ELSE ALTER TABLE device_info_threshold AUTO_INCREMENT=1;
END CASE;

DELETE FROM device_info_paranameandsuffix WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN SET ifAffectedRow=0;
ELSE ALTER TABLE device_info_paranameandsuffix AUTO_INCREMENT=1;
END CASE;

DELETE FROM faults_config WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN SET ifAffectedRow=0;
ELSE ALTER TABLE faults_config AUTO_INCREMENT=1;
END CASE;

IF(ifAffectedRow=1) THEN COMMIT;
```

```
ELSE ROLLBACK;
END IF;

END
```

```
-- 方式1：
CASE 表达式
WHEN 值1 THEN 结果1或者语句1
WHEN 值2 THEN 结果2或者语句2
...
ELSE 结果n或者语句n
END CASE; （如果是放在BEGIN END之间需要加CASE，如果在SELECT后则不需要）

-- 方式2：
CASE
WHEN <条件1> THEN <命令>
WHEN <条件2> THEN <命令>
...
ELSE commands
END CASE;
```

```
SELECT
t.name 姓名,
(CASE t.sex
WHEN 1 THEN '男'
WHEN 2 THEN '女'
ELSE '未知'
END) 性别
FROM t_stu t;
```

```
SELECT
t.name 姓名,
(CASE
WHEN t.sex = 1 THEN '男'
WHEN t.sex = 2 THEN '女'
ELSE '未知'
END) 性别
FROM t_stu t;
```

```
SELECT t1.DeviceNO,t2.DeviceName,
(CASE WHEN t1.DeviceStatus=1 THEN '正常'
WHEN t1.DeviceStatus=0 THEN '异常'
WHEN t1.DeviceStatus=-1 THEN '无效'
END) AS DeviceStatus,
t1.TestingNum,t1.DefectNum,
CONCAT(t1.CPUTemperature,'℃') AS CPUTemperature,CONCAT(t1.CPUUsage,'%') AS
CPUUsage,CONCAT(t1.MemoryUsage,'%') AS MemoryUsage
FROM device_info AS t1 INNER JOIN device AS t2
ON t1.DeviceNO=t2.DeviceNO
WHERE t1.LineNO='001'
ORDER BY t1.`NO`;
```

（2）循环语句

https://www.cnblogs.com/chenliyang/p/6553068.html

WHILE循环

```
[标签:]while 循环条件 do
循环体
end while [标签];
```

> - 标签——是给while取个名字
> - iterate ——结束本次循环
> - leave ——跳出循环

```
/*删除test1表记录*/
DELETE FROM test1;
/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc5;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc5(v_count int)
BEGIN
    DECLARE i int DEFAULT 0;
    a:WHILE i<=v_count DO
        SET i=i+1;
        /*如果i不为偶数，跳过本次循环*/
        IF i%2!=0 THEN
            ITERATE a;
        END IF;
        /*插入数据*/
        INSERT into test1 values (i);
    END WHILE;
END $
/*结束符置为;*/
DELIMITER ;
```

```
mysql> DELETE FROM test1;
Query OK, 5 rows affected (0.00 sec)
mysql> CALL proc5(10);
Query OK, 1 row affected (0.01 sec)
mysql> SELECT * from test1;
+----+
| a |
+----+
| 2 |
| 4 |
| 6 |
| 8 |
| 10 |
+----+
5 rows in set (0.00 sec)
```

```
-- test2表有2个字段（a,b），写一个存储过程（2个参数：v_a_count，v_b_count)，使用双重循环插入数据，数据条件：a的范围[1,v_a_count]、b的范围[1,v_b_count]所有偶数的组合。

/*删除存储过程*/
DROP PROCEDURE IF EXISTS proc8;
/*声明结束符为$*/
DELIMITER $
/*创建存储过程*/
CREATE PROCEDURE proc8(v_a_count int,v_b_count int)
BEGIN
    DECLARE v_a int DEFAULT 0;
    DECLARE v_b int DEFAULT 0;
    a:WHILE v_a<=v_a_count DO
        SET v_a=v_a+1;
        SET v_b=0;
        b:WHILE v_b<=v_b_count DO
            SET v_b=v_b+1;
            IF v_a%2!=0 THEN
                ITERATE a;
            END IF;
            IF v_b%2!=0 THEN
                ITERATE b;
            END IF;
            INSERT INTO test2 VALUES (v_a,v_b);
        END WHILE b;
    END WHILE a;
END $
/*结束符置为;*/
DELIMITER ;
```

```
mysql> CALL proc8(4,6);
Query OK, 1 row affected (0.01 sec)
mysql> SELECT * from test2;
+---+---+
| a | b |
+---+---+
| 2 | 2 |
| 2 | 4 |
| 2 | 6 |
| 4 | 2 |
| 4 | 4 |
| 4 | 6 |
+---+---+
6 rows in set (0.00 sec)
```

21.索引

https://blog.csdn.net/qq_41174684/article/details/91350623

（1）什么情况应不建索引或少建索引

表记录太少

如果一个表只有5条记录，采用索引去访问记录的话，那首先需访问索引表，再通过索引表访问数据表，一般索引表与数据表不在同一个数据块，这种情况下ORACLE至少要往返读取数据块两次。而不用索引的情况下ORACLE会将所有的数据一次读出，处理速度显然会比用索引快。

经常插入、删除、修改的表

（2）索引区分度

索引区分度

我们看2个有序数组

[1,2,3,4,5,6,7,8,8,9,10]

[1,1,1,1,1,8,8,8,8,8]

上面2个数组是有序的，都是10条记录，如果我需要检索值为8的所有记录，那个更快一些?

咱们使用二分法查找包含8的所有记录过程如下：先使用二分法找到最后一个小于8的记录，然后沿着这条记录向后获取下一个记录，和8对比，知道遇到第一个大于8的数字结束，或者到达数组末尾结束。

采用上面这种方法找到8的记录，第一个数组中更快的一些。因为第二个数组中含有8的比例更多的，需要访问以及匹配的次数更多一些。

这里就涉及到数据的区分度问题：

索引区分度 = count(distint 记录) / count(记录)。

当索引区分度高的时候，检索数据更快一些，索引区分度太低，说明重复的数据比较多，检索的时候需要访问更多的记录才能够找到所有目标数据。

当索引区分度非常小的时候，基本上接近于全索引数据的扫描了，此时查询速度是比较慢的。

第一个数组索引区分度为1，第二个区分度为0.2，所以第一个检索更快的一些。

所以我们创建索引的时候，尽量选择区分度高的列作为索引。

（3）普通索引——单列索引

表已建立后创建索引

```
CREATE INDEX indexName ON t_name (`fieldName`);
```

修改表结构——添加索引

```
ALTER TABLE t_name ADD INDEX indexName(`fieldName`);

ALTER TABLE dept ADD INDEX index1(`deptno`);
```

建表的时候创建索引

```
-- [...]中表示可选内容
CREATE TABLE t_name(
`ID` INT NOT NULL,
`username` VARCHAR(16) NOT NULL,
INDEX [indexName] (username(length))
);
```

删除索引

```
DROP INDEX [indexName] ON t_name;
```

显示已建立的索引

```
SHOW INDEX FROM t_name\G
```

(3) 复合索引

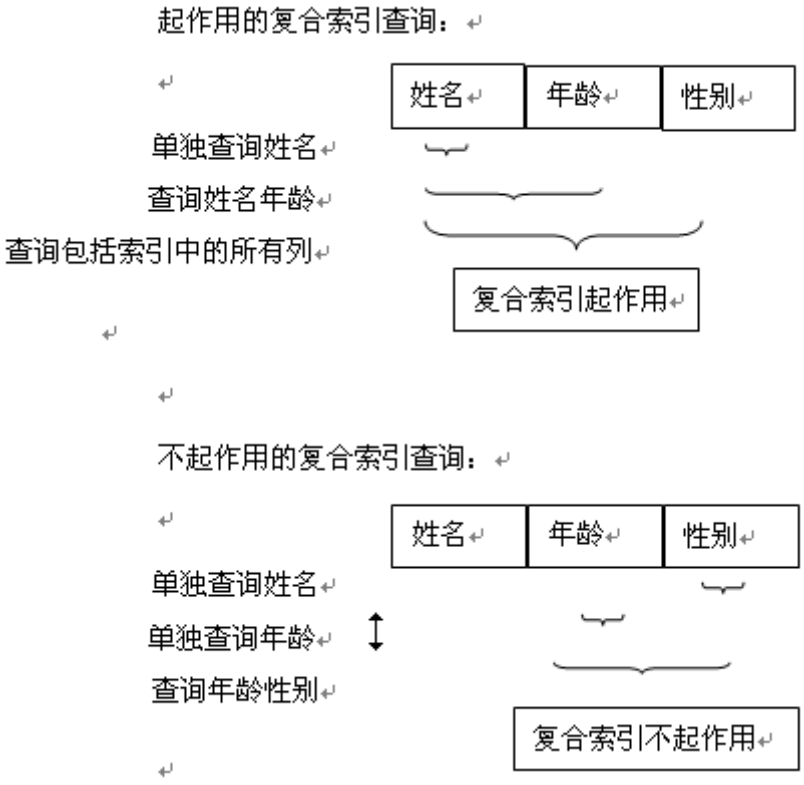
```
CREATE INDEX indexName ON t_name (colName1(len1), colName2(len2));
```

建立复合索引时的注意事项:

> 对一张表来说,如果有一个复合索引 on (col1,col2),就没有必要同时建立一个单索引 on col1;
>
> 如果查询条件需要,可以在已有单索引 on col1的情况下,添加复合索引on (col1,col2),对于效率有一定的提高;
>
> 同时建立多字段(包含5、6个字段)的复合索引没有特别多的好处,相对而言,建立多个窄字段(仅包含一个,或顶多2个字段)的索引可以达到更好的效率和灵活性。

使用复合索引的注意事项:

> - 对于复合索引,在查询使用时,最好将**条件顺序按找索引的顺序**,这样效率最高;
>
>   ```
>   select * from table1 where col1=A AND col2=B AND col3=D
>   ```
>
>   如果使用
>
>   ```
>   where col2=B AND col1=A
>   ```
>
>   或者
>
>   ```
>   where col2=B
>   ```
>
>   将不会使用索引。



- 如果where条件中是**OR关系，必须所有的or条件都必须是独立索引**，否则加索引不起作用。见：mysql关于or的索引问题

（4）唯一索引

创建唯一索引必须要指定关键字unique，唯一索引和单列索引类似，主要的区别是：**唯一索引限制列的值必须唯一，允许出现一个空值NULL。**对于多个字段，对于多个字段而言，列值的组合必须是唯一的，创建唯一索引也有3种方式：

类似创建单列索引，只是在INDEX前加个UNIQUE。

```
create unique index index_name on
tbl_name(index_col_name1,index_col_name2,index_col_name3);
```

与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值

```
CREATE UNIQUE INDEX indexName ON mytable(username(length))

ALTER table mytable ADD UNIQUE [indexName] (username(length))

CREATE TABLE mytable(
ID INT NOT NULL,
username VARCHAR(16) NOT NULL,
UNIQUE [indexName] (username(length))
);
```

（5）主键索引

主键索引也就是丛生索引，**是一种特殊的唯一索引，不允许有空值。**创建主键索引的语法是：

（6）索引的使用

最左匹配原则

少用联结表

平时我们做项目的时候，建议少用表连接，比如电商中需要查询订单的信息和订单中商品的名称，可以先查询查询订单表，然后订单表中取出商品的id列表，采用in的方式到商品表检索商品信息，由于商品id是商品表的主键，所以检索速度还是比较快的。

（7）索引的原理

详细内容、B+树原理、MySQL页见《Mysql笔记.PDF》

22. 事务

（1）使用条件

- 在 MySQL 中只有使用了 **Innodb数据库引擎的数据库或表才支持事务。**
- 事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。
- **事务只能用来管理 insert,update,delete 语句**，不能控制SELECT和DROP。

（2）几个概念

- 事务——TRANSACTION
- 回滚——ROLLBACK TO，回退到某个保留点
- 提交——COMMIT，将未保存的动作写入数据库表
- 保留点——SAVAPOINT，一系列语句中的**占位符**，当回退时，可以标记退回到什么位置。每个保留点都取**唯一的名字。**

（3）隐式事务——更改MySQL的自动提交行为

MySQL默认情况下，只要表发生更改，就会立即提交。

```
SET GLOBAL AUTOCOMMIT=0;     -- 关闭自动提交，直到AUTOCOMMIT置1时才提交
SET GLOBAL AUTOCOMMIT=1;     -- 开启自动提交
```

```
mysql> show variables like 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit | ON |
+---------------+-------+
1 row in set, 1 warning (0.00 sec)
-- autocommit 为ON表示开启了自动提交
```

（4）显式事务——手动提交、回滚

https://blog.csdn.net/weixin_39641173/article/details/113945257?utm_medium=distribute.pc_relevant_bbs_down.none-task-blog-baidujs-1.nonecase&depth_1-utm_source=distribute.pc_relevant_bbs_down.none-task-blog-baidujs-1.nonecase

- 开启事务

```
START TRANSACTION  -- 设置是否开启自动提交事务
```

```
set autocommit=0;
...
set autocommit=1;
```

- COMMIT

  将COMMIT与START TRANSACTION之间的动作提交，动作就提交到表，无法撤销了。**提交后事务被显式结束。**

- ROLLBACK

  将ROLLBACK与START TRANSACTION之间的部分撤销，回到START TRANSACTION事务开启时，**回滚后事务被显式结束。**

- 使用保留点

  SAVAPOINT P1

  ROLLBACK TO P1

  使用savepoint回滚难免有些性能消耗，一般可以用IF改写

  savepoint的良好使用的场景之一是“嵌套事务”，你可能希望程序执行一个小的事务，但是不希望回滚外面更大的事务

- 保留点ROLLBACK或者COMMIT后就自动释放。也可用RELEASE P1手动释放。

```
/**
(1)在执行sql语句之前，我们要开启事务 start transaction;

(2)正常执行我们的sql语句

(3)当sql语句执行完毕，存在两种情况：

1，全都成功，我们要将sql语句对数据库造成的影响提交到数据库中，committ

2，某些sql语句失败，我们执行rollback(回滚)，将对数据库操作赶紧撤销
*/

SET AUTOCOMMIT=0
START TRANSACTION;
UPDATE table1 SET field1='aaa' WHERE type=1;
UPDATE table2 SET field2='bbb' WHERE type=1;
COMMIT;//或ROLLBACK;
END;
```

（5）优化查询——只读事务

```
START TRANSACTION READ ONLY;
```

表示在事务中执行的是一些只读操作，如查询，但是不会做insert、update、delete操作，数据库内部对只读事务可能会有一些性能上的优化

若在事务中使用插入、删除等，会导致事务报错。

（6）并行事务——隔离级别与脏读

并行事务存在的问题

> **脏读**——一个事务读到另一个事务还没有提交的数据（没有提交的数据一旦回滚到原数据，那读到的数据就是脏数据）
>
> **可重复读**——一个事务操作中对于一个读取操作不管多少次，读取到的结果都是一样的。
>
> **幻读**——幻读在可重复读的模式下才会出现，其他隔离级别中不会出现
>
> > 事务A操作如下： 1、打开事务 2、查询号码为X的记录，不存在 3、插入号码为X的数据，插入报错（为
> > 什么会报错，先向下看） 4、查询号码为X的记录，发现还是不存在（由于是可重复读，所以读取记录X
> > 还是不存在的）
> >
> > 事物B操作：在事务A第2步操作时插入了一条X的记录，所以会导致A中第3步插入报错（违反了唯一约
> > 束）

隔离级别

https://blog.csdn.net/weixin_39924486/article/details/113088777

事务隔离级别主要是解决了多个事务之间数据可见性及数据正确性的问题。
隔离级别分为4种：

> **读未提交**——READ-UNCOMMITTED
>
> **读已提交**——READ-COMMITTED
>
> **可重复读**——REPEATABLE-READ
>
> **串行**——SERIALIZABLE

上面*4*种隔离级别越来越强，会导致数据库的并发性也越来越低。 比如最高级别 *SERIALIZABLE* 会让事物串行执行，并发操作变成串行
了，会导致系统性能直接降低

读已提交（*READ-COMMITTED*）通常用的比较多

隔离级别	脏读	不可重复读	幻读
READ-UNCOMMITTED	有	有	无
READ-COMMITTED	无	有	无
REPEATABLE-READ	无	无	有
SERIALIZABLE	无	无	无

隔离级别修改

```
transaction-isolation=READ-UNCOMMITTED

-- 以管理员身份打开cmd窗口，重启mysql
C:\Windows\system32>net stop mysql
mysql 服务正在停止..
mysql 服务已成功停止。
C:\Windows\system32>net start mysql
mysql 服务正在启动 .
mysql 服务已经启动成功。

-- 查看隔离级别
mysql> show variables like 'transaction_isolation';
+-----------------------+----------------+
| Variable_name | Value |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
+-----------------------+----------------+
1 row in set, 1 warning (0.00 sec)
```

READ-UNCOMMITTED——读未提交

当前事务进行中时，存在脏读，可以读到未commit的其他事务操作的数据。

```
transaction-isolation=READ-UNCOMMITTED

net stop mysql

net start mysql

show variables like 'transaction_isolation';

delete from test1;
select * from test1;
```

按时间顺序在2个窗口中执行下面操作：

时间	窗口A	窗口B
T1	start transaction;	
T2	select * from test1;	
T3		start transaction;
T4		insert into test1 values (1);
T5		select * from test1;
T6	select * from test1;	
T7		commit;
T8	commit;	


```
-- A窗口如下：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from test1;
Empty set (0.00 sec)
mysql> select * from test1;
+------+
| a |
+------+
| 1 |
+------+
1 row in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

-- B窗口如下：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)
mysql> select * from test1;
+------+
| a |
+------+
| 1 |
+------+
1 row in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

> T5-B：有数据，T6-A窗口：无数据，A看不到B的数据，说明没有脏读。
> T6-A窗口：无数据，T8-A：看到了B插入的数据，此时B已经提交了，A看到了B已提交的数据，说明可
> 以读取到已提交的数据。
> T2-A、T6-A：无数据，T8-A：有数据，多次读取结果不一样，说明不可重复读。
> 结论：读已提交情况下，无法读取到其他事务还未提交的数据，可以读取到其他事务已经提交的数

> 据，
> 多次读取结果不一样，未出现脏读，出现了读已提交、不可重复读。

REPEATABLE-READ——可重复读

当前事务未commit前，无论其他事务对某些数据的操作怎样，读取这些数据的结果不变，即可重复读。

按时间顺序在2个窗口中执行下面操作：

时间	窗口A	窗口B
T1	start transaction;	
T2	select * from test1;	
T3		start transaction;
T4		insert into test1 values (1);
T5		select * from test1;
T6	select * from test1;	
T7		commit;
T8	select * from test1;	
T9	commit;	
T10	select * from test1;	

```
-- A窗口如下：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from test1;
Empty set (0.00 sec)
mysql> select * from test1;
Empty set (0.00 sec)
mysql> select * from test1;
Empty set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from test1;
+------+
| a |
+------+
| 1 |
| 1 |
+------+
2 rows in set (0.00 sec)
-- B窗口如下：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
mysql> insert into test1 values (1);
Query OK, 1 row affected (0.00 sec)
mysql> select * from test1;
+------+
| a |
+------+
```

```
| 1 |
| 1 |
+------+
2 rows in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

> T2-A、T6-A窗口：无数据，T5-B：有数据，A看不到B的数据，说明没有脏读。
> T8-A：无数据，此时B已经提交了，A看不到B已提交的数据，A中3次读的结果一样都是没有数据的，说
> 明可重复读。
> 结论：可重复读情况下，未出现脏读，未读取到其他事务已提交的数据，多次读取结果一致，即可重复
> 读。

幻读

幻读只会在 REPEATABLE-READ （可重复读）级别下出现，需要先把隔离级别改为可重复读。

```
transaction-isolation=REPEATABLE-READ

net stop mysql

net start mysql

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

```
-- 上面我们创建t_user表，name添加了唯一约束，表示name不能重复，否则报错。
mysql> create table t_user(id int primary key,name varchar(16) unique key);
Query OK, 0 rows affected (0.01 sec)
mysql> insert into t_user values (1,'路人甲Java'),(2,'路人甲Java');
ERROR 1062 (23000): Duplicate entry '路人甲Java' for key 'name'
mysql> select * from t_user;
Empty set (0.00 sec)
```

按时间顺序在2个窗口中执行下面操作：

时间	窗口A	窗口B
T1	start transaction;	
T2		start transaction;
T3		-- 插入 路人甲Java insert into t_user values (1,'路人甲Java');
T4		select * from t_user;
T5	-- 查看 路人甲Java 是否存在 select * from t_user where name='路人甲Java';	
T6		commit;
T7	-- 插入 路人甲Java insert into t_user values (2,'路人甲Java');	
T8	-- 查看 路人甲Java 是否存在 select * from t_user where name='路人甲Java';	
T9	commit;	


```
-- A窗口如下：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from t_user where name='路人甲Java';
Empty set (0.00 sec)
mysql> insert into t_user values (2,'路人甲Java');
ERROR 1062 (23000): Duplicate entry '路人甲Java' for key 'name'
mysql> select * from t_user where name='路人甲Java';
Empty set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
-- B窗口如下：
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)
mysql> insert into t_user values (1,'路人甲Java');
Query OK, 1 row affected (0.00 sec)
mysql> select * from t_user;
+----+---------------+
| id | name |
+----+---------------+
| 1 | 路人甲Java |
+----+---------------+
1 row in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

> A想插入数据 路人甲Java ，插入之前先查询了一下（T5时刻）该用户是否存在，发现不存在，然后在T7
> 时刻执行插入，报错了，报数据已经存在了，因为T6时刻 B 已经插入了 路人甲Java 。
> 然后A有点郁闷，刚才查的时候不存在的，然后A不相信自己的眼睛，又去查一次（T8时刻），发现 路
> 人甲Java 还是不存在的。
> 此时A心里想：数据明明不存在啊，为什么无法插入呢？这不是懵逼了么，A觉得如同发生了幻觉一样。

SERIALIZABLE——串行

SERIALIZABLE会**让并发的事务串行执行**（多个事务之间读写、写读、写写会产生互斥，效果就是串行执行，多个事务之间的**读读不会产生互斥**）。

> 读写互斥：事务A中先读取操作，事务B发起写入操作，事务A中的读取会导致事务B中的写入处于等待状态，直到A事务完成为止。
> 表示我开启一个事务，为了保证事务中不会出现上面说的问题（脏读、不可重复读、读已提交、幻读），那么我读取的时候，其他事务有修改数据的操作需要排队等待，等待我读取完成之后，他们才可以继续。
> 写读、写写也是互斥的，读写互斥类似。

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

按时间顺序在2个窗口中执行下面操作：

时间	窗口A	窗口B
T1	start transaction;	
T2	select * from test1;	
T3		start transaction;
T4		insert into test1 values (1);
T5	commit;	
T6		commit;

> 按时间顺序运行上面的命令，会发现T4-B这样会被阻塞，直到T5-A执行完毕。
> 可以看出来，事务只能串行执行了。串行情况下不存在脏读、不可重复读、幻读的问题了。

(7) 事务对于Insert数据速度的影响

- InnoDB引擎，事务中insert的数据未commit之前，在内存的buffer_pool中

  https://blog.csdn.net/weixin_36230541/article/details/113188052

  MySQL不管事务有没有提交，DML语言所操作的数据就是innodb_buffer_pool缓存中的数据。只有check point检查到需要将脏页刷新到硬盘时，才会将数据写入磁盘

- 对于Insert性能影响

https://blog.csdn.net/zdw19861127/article/details/78597523

> 对于一些数据量较大的系统，数据库面临的问题除了查询效率低下，还有就是数据入库时间长。总会有一些手段可以提高insert效率：
>
> **1. 多条Insert语句合并成一条语句，VALUES后跟多条数据，以逗号隔开**
>
> > https://www.zhihu.com/question/274242137/answer/1621320644
> >
> > 这里第二种SQL执行效率高的主要原因是合并后日志量(MySQL的binlog和innodb的事务让日志)减少了，降低日志刷盘的数据量和频率，从而提高效率。通过合并SQL语句，同时也能减少SQL语句解析的次数，减少网络传输的IO。
> >
> > 数据库的一个插入动作，包含了连接，传输，执行，提交/回滚 等等的动作，在 执行的时候可能还会遇到锁表，等待等等，所以，批量插比逐个插效率高，是大部分情况，而不是绝对情况
> >
> > 大部分情况下，批量插和逐个插，在执行层面，耗时接近；而不用多次连接数据库，在数据传输层面，也是一次性传输效率高（网络传输和这个模型类似，也有很多前置后置过程），而提交，也是只发起了一次，因而显得效率高
>
> **2. 使用事务**
>
> > 多条Insert语句放在事务中，以commit执行

（8）事务处理处理多次UPDATE、DELETE、INSERT

用事务将多次UPDATE、DELETE、INSERT操作原子化，要么一次执行成功，要么回滚。

有多个操作时，UPDATE、DELETE、INSERT等操作会返回影响的行数，通过ROW_COUNT()获取。将其记录在变量count中，通过判断count数值，决定是提交或回滚。

> **使用PREPARE，获取ROW_COUNT必须要在DEALLOCATE释放sql语句s之前**

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `deleteDevice`(IN ln VARCHAR(20), IN
dn VARCHAR(20))
BEGIN
DECLARE ifAffectedRow TINYINT(1) DEFAULT 1;      -- TINYINT括号内加数字，表示位数，如
INT(4):0001
DECLARE SQL_FOR_UPDATE_device_config VARCHAR(100);  -- 保存最后执行的动态SQL语句
START TRANSACTION;

-- 查询字段是动态，所以用动态SQL
SET SQL_FOR_UPDATE_device_config=CONCAT('UPDATE device_config SET
DeviceStatus_', dn, '=0 WHERE LineNO=', ln, ';');
SET @sql=SQL_FOR_SELECT;
PREPARE stmt FROM @sql;
EXECUTE stmt;
-- 使用PREPARE，获取ROW_COUNT必须要在DEALLOCATE释放sql语句s之前
CASE ROW_COUNT()     -- 判断语句执行影响表device_config的行数
    WHEN 0 THEN
        SET ifAffectedRow=0;
    ELSE
        BEGIN END;
END CASE;
DEALLOCATE PREPARE stmt;

DELETE FROM device_info WHERE LineNO=ln AND DeviceNO=dn;
CASE
    WHEN ROW_COUNT()=0 THEN
```

```
        SET ifAffectedRow=0;     -- THEN中若是语句需要加分号
    ELSE
        ALTER TABLE device_info AUTO_INCREMENT=1;
END CASE;

DELETE FROM device_info_threshold WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN
    SET ifAffectedRow=0;
ELSE
    ALTER TABLE device_info_threshold AUTO_INCREMENT=1;
END CASE;

DELETE FROM device_info_paranameandsuffix WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN
    SET ifAffectedRow=0;
ELSE
    ALTER TABLE device_info_paranameandsuffix AUTO_INCREMENT=1;
END CASE;

DELETE FROM faults_config WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN
    SET ifAffectedRow=0;
ELSE
    ALTER TABLE faults_config AUTO_INCREMENT=1;
END CASE;

IF(ifAffectedRow=1) THEN COMMIT;     -- 若5次操作同时成功则commit，否则rollback
ELSE ROLLBACK;
END IF;

END
```

23. 变量

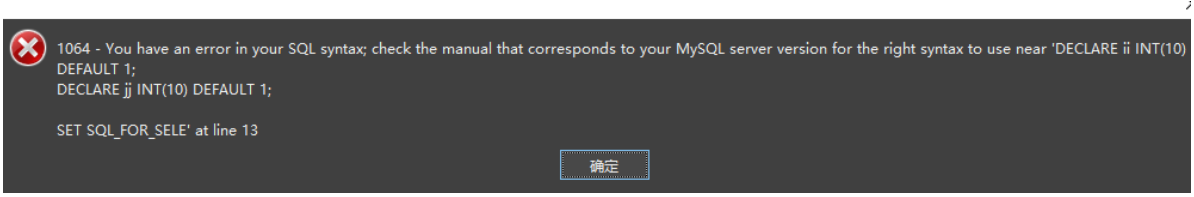
变量分为系统变量、自定义变量

自定义变量分为：局部变量、用户变量

（1）局部变量

declare用于定义局部变量变量，**在存储过程和函数中通过declare定义变量在begin...end中，且在语句之前。**并且可以通过重复定义多个变量。declare变量的作用范围同编程里面类似，在这里一般是在对应的begin和end之间。在end之后这个变量就没有作用了，不能使用了。这个同编程一样。

局部变量的定义DECLARE只能写在开头？否则报错?



赋值

```
/*方式1*/
set 局部变量名=值;
set 局部变量名:=值;
select 局部变量名:=值;
/*方式2*/
select 字段 into 局部变量名 from 表;
```

（2）用户变量

用户变量可以在任何地方使用也就是**既可以在begin end里面使用，也可以在他外面使用。**

定义

```
/*方式1*/
set @变量名=值;
/*方式2*/
set @变量名:=值;
/*方式3*/
select @变量名:=值;
```

赋值

```
/*方式1：这块和变量的声明一样*/
set @变量名=值;
set @变量名:=值;
select @变量名:=值;
/*方式2*/
select 字段 into @变量名 from 表;
```

使用

```
insert into employees (first_name,email) values (@first_name,@email);
```

---

24. 存储过程

存储过程类似批处理脚本，但脚本未经编译。存储过程经过预编译，执行时不需要再次编译。

（1）概念

MySQL 5.0 版本开始支持存储过程。

存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。类似于函数。

存储过程是为了完成特定功能的SQL语句集，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数(需要时)来调用执行。

存储过程思想上很简单，就是数据库 SQL 语言层面的代码封装与重用。

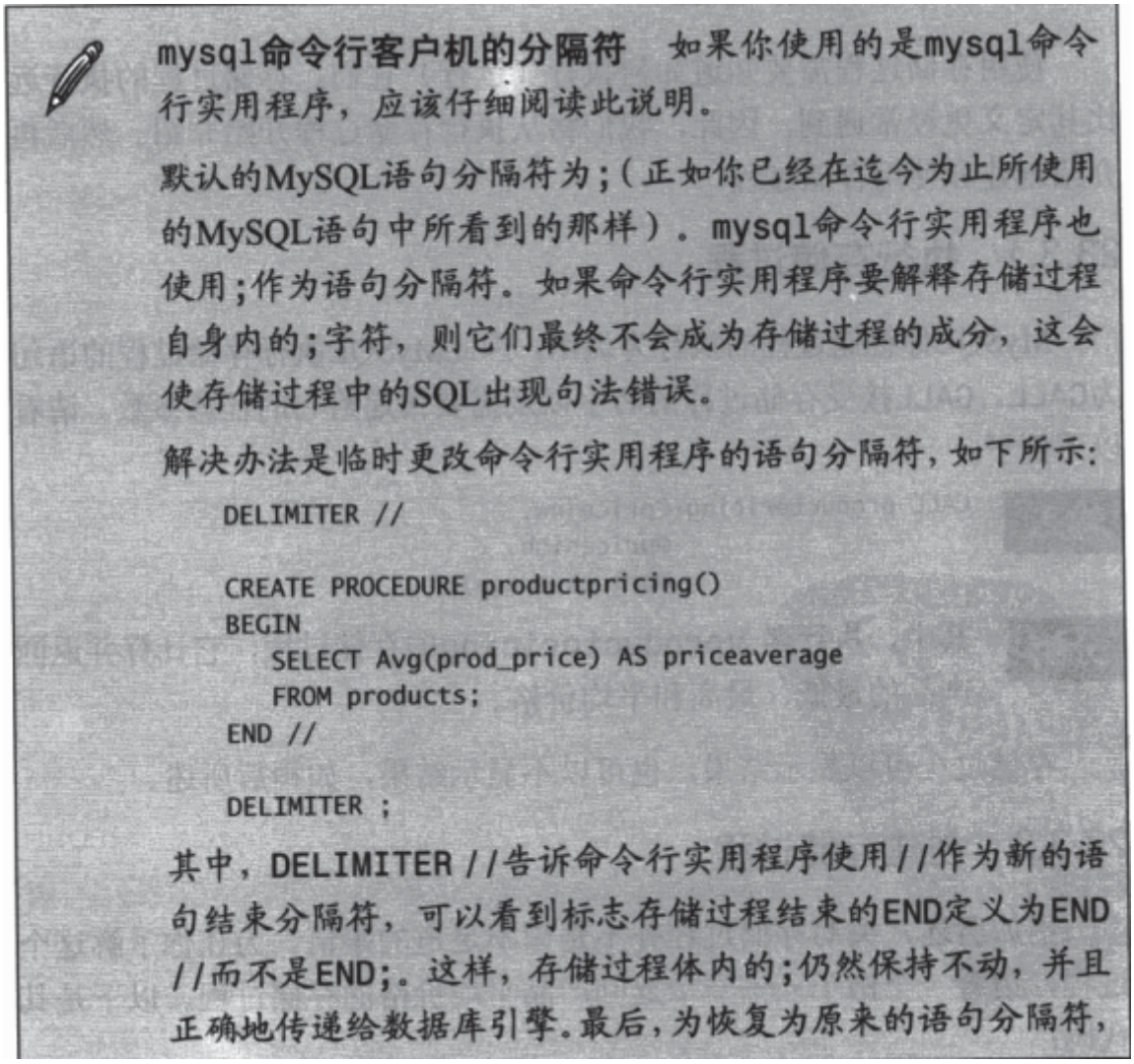
- 存储过程就是具有名字的一段代码，用来完成一个特定的功能。
- 创建的存储过程保存在数据库的数据字典中。

使用存储过程比多条语句效率更高。且存储过程将语句封装，便于调用的人使用时保持代码的一致性。

- 好处

> 减少编译次数
>
> 减少了变异次数减少了和数据库的链接次数，提高效率

(2) 修改结束符



mysql命令行客户机的分隔符 如果你使用的是mysql命令行实用程序，应该仔细阅读此说明。

默认的MySQL语句分隔符为;（正如你已经在迄今为止所使用的MySQL语句中所看到的那样）。mysql命令行实用程序也使用;作为语句分隔符。如果命令行实用程序要解释存储过程自身内的;字符，则它们最终不会成为存储过程的成分，这会使存储过程中的SQL出现句法错误。

解决办法是临时更改命令行实用程序的语句分隔符，如下所示：

```
DELIMITER //

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

DELIMITER ;
```

其中，DELIMITER //告诉命令行实用程序使用//作为新的语句结束分隔符，可以看到标志存储过程结束的END定义为END //而不是END;。这样，存储过程体内的;仍然保持不动，并且正确地传递给数据库引擎。最后，为恢复为原来的语句分隔符，

(3) 参数列表

- in：该参数可以作为输入，也就是该参数需要调用方传入值。
- out：该参数可以作为输出，也就是说该参数可以作为返回值。

```
SELECT COUNT(*),max(id) into user_count,max_id from t_user;
```

- inout：该参数既可以作为输入也可以作为输出，也就是说该参数需要在调用的时候传入值，又可以作为返回值。
- 参数模式默认为IN。
- 一个存储过程可以有多个输入、多个输出、多个输入输出参数

（4）创建存储过程

存储过程一旦创建就被保存在服务器上供调用，直至删除。

```
DELIMITER $
CREATE PROCEDURE proc()
BEGIN
    SELECT * FROM t_name;
END//

DELIMITER ;
```

```
DELIMITER $
CREATE PROCEDURE func(
    OUT para1 VARCHAR(255),
    OUT para2 INT(10)
)
BEGIN
    SELECT name FROM t1_name INTO para1;
    SELECT age FROM t2_name INTO para2;
END $

DELIMITER ;
```

> - **OUT声明参数是传出存储过程的**
> - **IN声明参数是传入存储过程的**
> - **INTO语句表示将语句结果保存到变量para中**

```
-- 显示创建该存储过程的语句
SHOW CREATE PROCEDURE proc;
```

```
delete a from t_user a where a.id = 4;
/*如果存储过程存在则删除*/
DROP PROCEDURE IF EXISTS proc3;
/*设置结束符为$*/
DELIMITER $
/*创建存储过程proc3*/
CREATE PROCEDURE proc3(id int,age int,in name varchar(16),out user_count int,out
max_id INT)
BEGIN
INSERT INTO t_user VALUES (id,age,name);
/*查询出t_user表的记录，放入user_count中,max_id用来存储t_user中最小的id*/
SELECT COUNT(*),max(id) into user_count,max_id from t_user;
END $
/*将结束符置为;*/
DELIMITER ;

/*创建了3个自定义变量*/
SELECT @id:=4,@age:=55,@name:='郭富城';
/*调用存储过程*/
CALL proc3(@id,@age,@name,@user_count,@max_id);
```

```
/*删除函数*/
DROP FUNCTION IF EXISTS get_user_id;
/*设置结束符为$*/
DELIMITER $
/*创建函数*/
CREATE FUNCTION get_user_id(v_name VARCHAR(16))
returns INT
BEGIN
DECLARE r_id int;
SELECT id INTO r_id FROM t_user WHERE name = v_name;
return r_id;
END $
/*设置结束符为;*/
DELIMITER ;
```

(5) 调用存储过程

```
CALL func();
```

```
CALL func(@paraName, @paraAge); //传递两个变量给存储过程
```

- MySQL中变量都必须以@开始
- 显示变量

  ```
  SELECT @paraName, @paraAge;
  ```

```
-- 例子
DELIMITER $
CREATE PROCEDURE totalStudentEAchClass(IN classNO VARCHAR(255), OUT totalStudent
INT(10))
BEGIN
    SELECT COUNT(studentID) FROM student WHERE classID=classNO INTO
totalStudent;
END $
DELIMITER ;

CALL totalStudentEAchClass("A班", @totalS);
SELECT @totalS;
```

(6) 删除存储过程

```
DROP PROCEDURE IF EXISTS proc;  //注意只写存储过程名，不加括号
```

(7) 显示所有存储过程

```
SHOW PROCEDURE STATUS;
```

（8）智能存储——脚本

迄今为止使用的所有存储过程基本上都是封装MySQL简单的 SELECT语句。虽然它们全都是有效的存储过程例子,但它们所能完成的工作你直接用这些被封装的语句就能完成(如果说它们还能带来更多的东西,那就是使事情更复杂)。**只有在存储过程内包含业务规则和智能处理时,它们的威力才真正显现出来,来使我们的语句执行更加可靠和智能，比如我们可以声明局部变量、添加内部注释、使用循环或判断语句等等。**

```
DELIMITER $
CREATE PROCEDURE ordertotal(
    IN onumber INT,
    IN taxable BOOLEAN,
    OUT ototal DECIMAL(8,2)
) COMMENT 'Obatin 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 $

DELIMITER ;
```

（9）字段名、表名作为变量传入存储过程进行查询，必须用动态SQL

https://www.cnblogs.com/fenxiangheiye/archive/2013/02/18/Mysql.html

```
-- 动态SQL
-- SQL_FOR_SELECT局部变量，保存最终执行的SQL语句
CREATE PROCEDURE `proc1`(IN colname varchar(20))
BEGIN
DECLARE SQL_FOR_SELECT varchar(255);
SET SQL_FOR_SELECT=CONCAT('SELECT `',colname,'` FROM device');
SET @sql=SQL_FOR_SELECT;
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;
END
```

```
CALL proc1('DeviceNO');
```

示例:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `initDtSideTileBar`()
BEGIN

DECLARE different_device_num INT DEFAULT 0;
DECLARE colname VARCHAR(20);
DECLARE SQL_FOR_SELECT varchar(1000);
DECLARE ii INT(10) DEFAULT 2;

SELECT COUNT(*) INTO different_device_num FROM device;

-- SELECT 打印变量调试
-- SELECT different_device_num;

SET SQL_FOR_SELECT='SELECT device_config.LineNO ,DeviceStatus_001';

a:WHILE ii<=different_device_num DO
    SELECT deviceNO INTO colname FROM device WHERE NO=ii;
    SET SQL_FOR_SELECT=CONCAT(SQL_FOR_SELECT, '+DeviceStatus_', colname);
    SET ii=ii+1;
END WHILE;

SET SQL_FOR_SELECT=CONCAT(SQL_FOR_SELECT, ' AS DeviceTotalNum FROM
device_config');

SET SQL_FOR_SELECT=CONCAT('SELECT t1.LineNO,t2.LineName,t1.DeviceTotalNum FROM
(', SQL_FOR_SELECT, ') AS t1 INNER JOIN productionline AS t2 ON
t1.LineNO=t2.LineNO;');

-- SELECT SQL_FOR_SELECT;

SET @sql=SQL_FOR_SELECT;
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;

END
```

（10） 在存储过程中使用UNION

https://segmentfault.com/q/1010000019889121

（11）存储过程中使用临时表

存储过程中使用临时表，简化过程

（12）用存储过程封装动态SQL语句

https://blog.csdn.net/qq_38216661/article/details/98871552

> - **可以在存储过程中动态的拼接表名，字段名，来达到动态查询的效果**
> - sql语句中还可以用?来代表参数，这样可以有效的防止sql注入

```
-- 通过输入确定查询条件
DROP PROCEDURE IF EXISTS judgeIfExistInFaultHistory;
DELIMITER $
CREATE PROCEDURE judgeIfExistInFaultHistory(valLineNO VARCHAR(20), valDeviceNO
VARCHAR(20), valFaultNO VARCHAR(20), valFaultTime VARCHAR(20))
BEGIN

SELECT COUNT(t1.`NO`)
FROM faults_history AS t1
WHERE t1.LineNO=valLineNO AND t1.DeviceNO=valDeviceNO AND t1.FaultNO=valFaultNO
AND t1.FaultTime=valFaultTime;

END$
DELIMITER ;
```

```
string cmdGetDeviceNO = "SELECT `DeviceNO` FROM device;";
            DataTable dtDeviceNOTemp = new DataTable();
            _initDtMySQL(ref dtDeviceNOTemp, cmdGetDeviceNO);
            string strT1 = "SELECT LineNo, DeviceStatus_" +
dtDeviceNOTemp.Rows[0]["DeviceNO"].ToString();
            int rowNumDtdeviceNOTemp = dtDeviceNOTemp.Rows.Count;
            for (int i = 1; i < rowNumDtdeviceNOTemp; i++)
            {
                strT1 += "+DeviceStatus_" + dtDeviceNOTemp.Rows[i]
["DeviceNO"].ToString();
            }
            strT1 += " AS DeviceTotalNum FROM device_config";
            string cmdInitDtSideTileBar = "SELECT
t1.LineNO,t2.LineName,t1.DeviceTotalNum " +
                                          "FROM (" + strT1 + ")AS t1 " +
                                          "INNER JOIN productionline AS t2 " +
                                          "ON t1.LineNO=t2.LineNO;";
//19ms
```

```
-- 封装存储过程
DELIMITER $
CREATE PROCEDURE initDtSideTileBar()
BEGIN

DECLARE different_device_num INT DEFAULT 0;
DECLARE colname VARCHAR(20);
DECLARE SQL_FOR_SELECT varchar(1000);
DECLARE ii INT(10) DEFAULT 2;

SELECT COUNT(*) INTO different_device_num FROM device;

-- SELECT 打印变量调试
-- SELECT different_device_num;

SET SQL_FOR_SELECT='SELECT device_config.LineNO ,DeviceStatus_001';

a:WHILE ii<=different_device_num DO
    SELECT deviceNO INTO colname FROM device WHERE NO=ii;
    SET SQL_FOR_SELECT=CONCAT(SQL_FOR_SELECT, '+DeviceStatus_', colname);
    SET ii=ii+1;
END WHILE;

SET SQL_FOR_SELECT=CONCAT(SQL_FOR_SELECT, ' AS DeviceTotalNum FROM
device_config');

SET SQL_FOR_SELECT=CONCAT('SELECT t1.LineNO,t2.LineName,t1.DeviceTotalNum FROM
(', SQL_FOR_SELECT, ') AS t1 INNER JOIN productionline AS t2 ON
t1.LineNO=t2.LineNO;');

-- SELECT SQL_FOR_SELECT;

SET @sql=SQL_FOR_SELECT;    -- 将指令字符串SQL_FOR_SELECT作为sql语句
PREPARE stmt FROM @sql;
EXECUTE stmt;               -- 通过EXECUTE执行sql语句
DEALLOCATE PREPARE stmt;    -- 通过DEALLOCATE PREPARE释放sql语句

END$
DELIMITER ;
```

```
/*建库javacode2018*/
DROP DATABASE IF EXISTS javacode2018;
CREATE DATABASE javacode2018;
USE javacode2018;
/*建表test1*/
DROP TABLE IF EXISTS test1;
CREATE TABLE test1 (
id INT NOT NULL COMMENT '编号',
name VARCHAR(20) NOT NULL COMMENT '姓名',
sex TINYINT NOT NULL COMMENT '性别,1：男，2：女',
email VARCHAR(50)
);
/*准备数据*/
DROP PROCEDURE IF EXISTS proc1;
DELIMITER $
CREATE PROCEDURE proc1()
BEGIN
```

```
DECLARE i INT DEFAULT 1;
START TRANSACTION;
WHILE i <= 2000000 DO
INSERT INTO test1 (id, name, sex, email) VALUES
(i,concat('javacode',i),if(mod(i,2),1,2),concat('javacode',i,'@163.com'));
SET i = i + 1;
if i%10000=0 THEN
COMMIT;
START TRANSACTION;
END IF;
END WHILE;
COMMIT;
END $
DELIMITER ;
```

（13）带有输出参数的存储过程

参数列表用OUT修饰

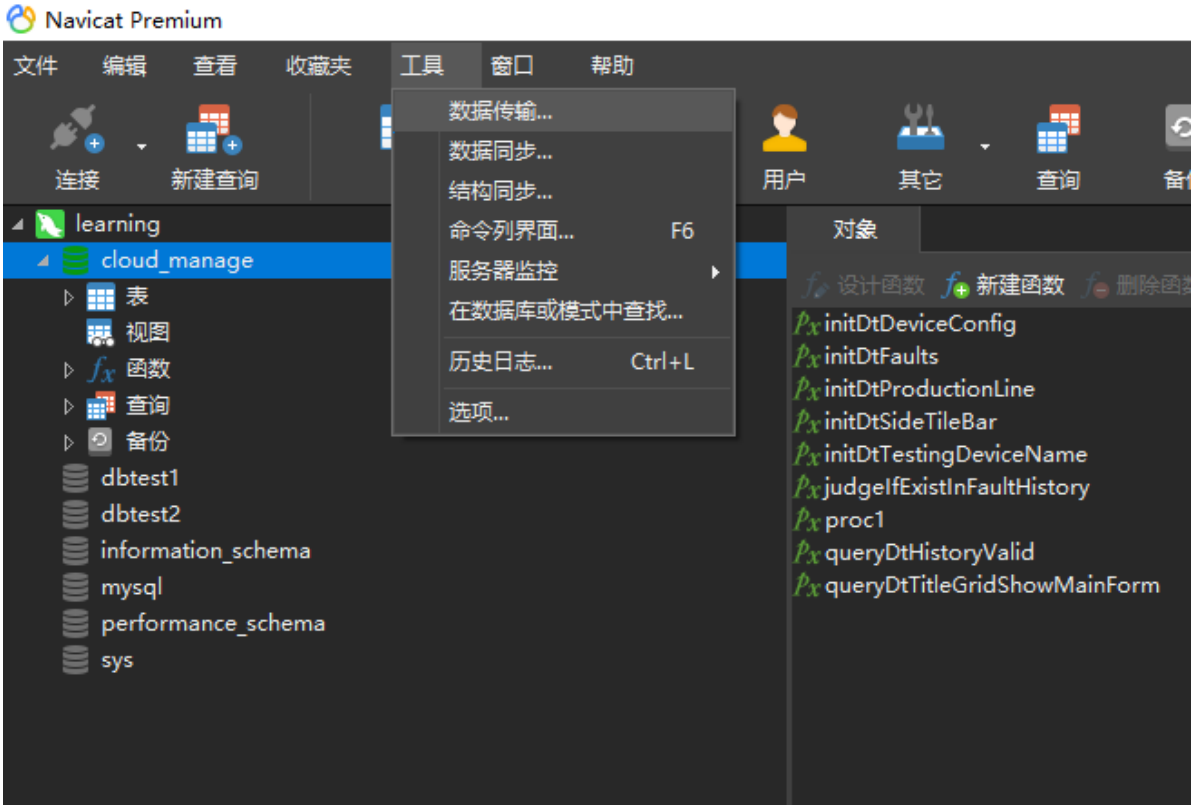
将存储过程中的参数赋值给最终返回的输出参数，用`SELECT val INTO output`

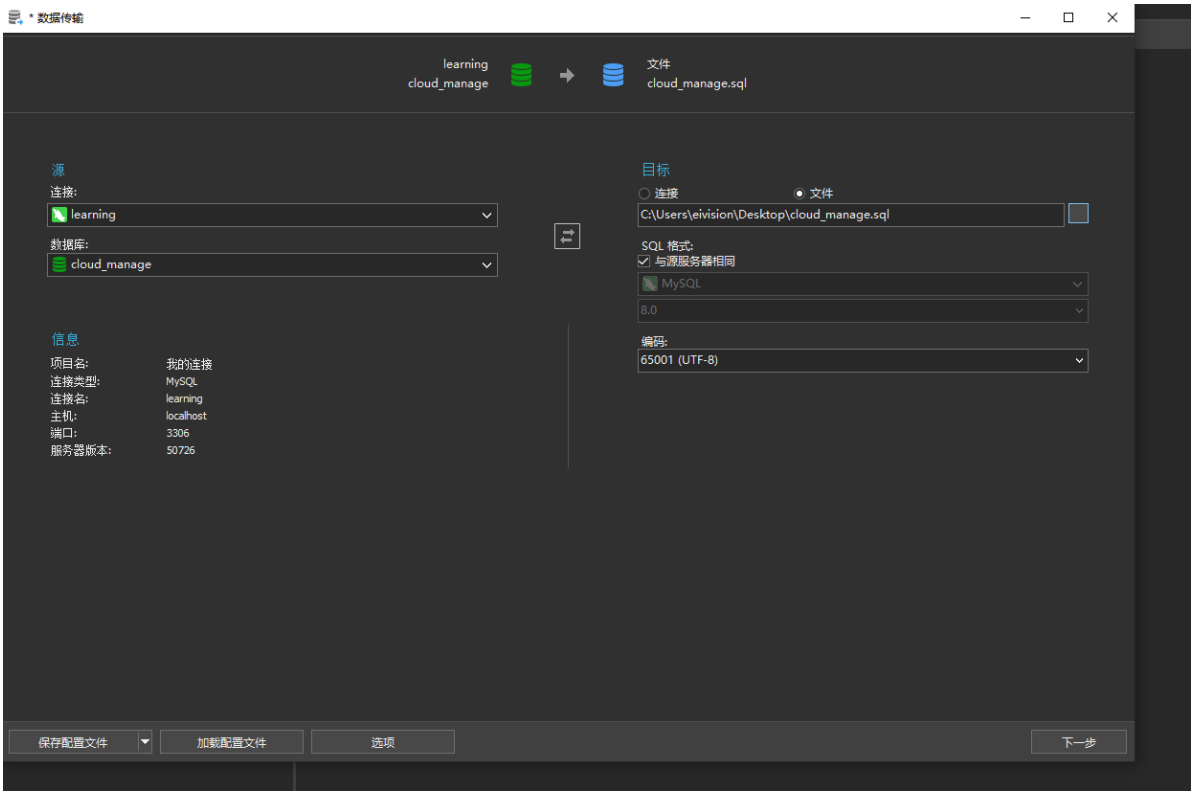
```
CREATE DEFINER=`root`@`localhost` PROCEDURE `p_deleteDevice`(IN ln VARCHAR(20),
IN dn VARCHAR(20), OUT ifRowAffected INT(1))
BEGIN
DECLARE ifAffectedRow TINYINT(1) DEFAULT 1;
DECLARE SQL_FOR_UPDATE_device_config VARCHAR(100);

START TRANSACTION;

SET SQL_FOR_UPDATE_device_config=CONCAT('UPDATE device_config SET
`DeviceStatus_', dn, '`=0 WHERE LineNO=', ln, ';');
SET @sql=SQL_FOR_UPDATE_device_config;
PREPARE stmt FROM @sql;
EXECUTE stmt;
-- 使用PREPARE，获取ROW_COUNT必须要在DEALLOCATE释放sql语句之前
CASE ROW_COUNT()
    WHEN 0 THEN
        SET ifAffectedRow=0;
    ELSE
        BEGIN END;
END CASE;
DEALLOCATE PREPARE stmt;

DELETE FROM device_info WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
    WHEN 0 THEN
        SET ifAffectedRow=0;
    ELSE
        ALTER TABLE device_info AUTO_INCREMENT=1;
END CASE;

DELETE FROM device_info_threshold WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN
    SET ifAffectedRow=0;
ELSE
    ALTER TABLE device_info_threshold AUTO_INCREMENT=1;
```

```
END CASE;

DELETE FROM device_info_paranameandsuffix WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN
    SET ifAffectedRow=0;
ELSE
    ALTER TABLE device_info_paranameandsuffix AUTO_INCREMENT=1;
END CASE;

DELETE FROM faults_config WHERE LineNO=ln AND DeviceNO=dn;
CASE ROW_COUNT()
WHEN 0 THEN
    SET ifAffectedRow=0;
ELSE
    ALTER TABLE faults_config AUTO_INCREMENT=1;
END CASE;

IF(ifAffectedRow=1) THEN
    COMMIT;
    SELECT ifAffectedRow INTO ifRowAffected;
ELSE
    ROLLBACK;
END IF;

END
```

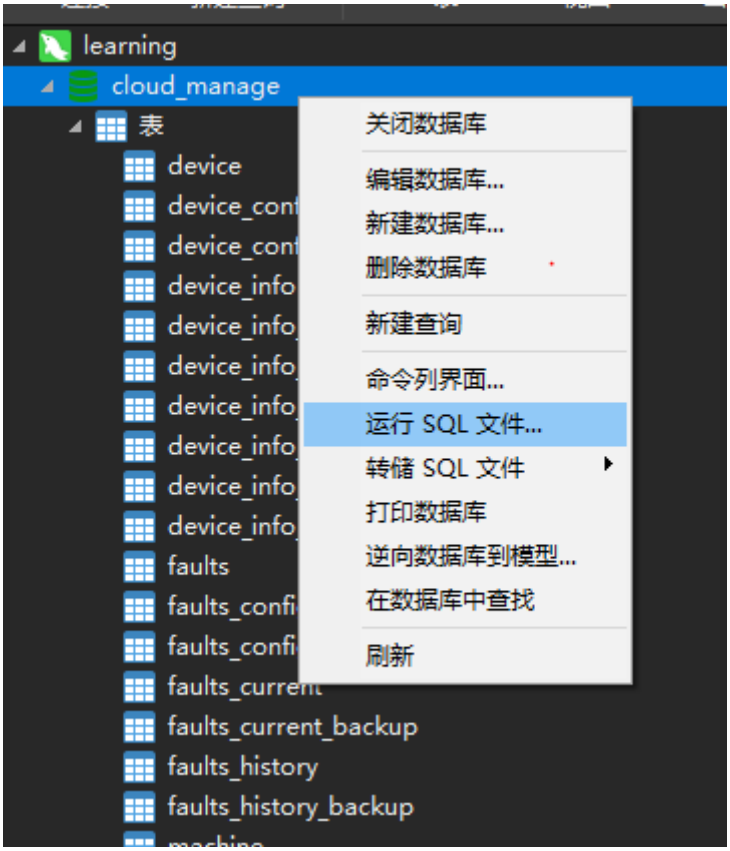
（14）存储过程的导出导入

https://jingyan.baidu.com/article/b7001fe1b162d80e7282ddcc.html


Navicat Premium
文件
编辑
查看
收藏夹
工具
窗口
帮助
连接
新建查询
用户
其它
查询
数据传输...
数据同步...
结构同步...
命令列界面... F6
服务器监控
在数据库或模式中查找...
历史日志... Ctrl+L
选项...
learning
cloud_manage
表
视图
函数
查询
备份
dbtest1
dbtest2
information_schema
mysql
performance_schema
sys
对象
设计函数
新建函数
initDtDeviceConfig
initDtFaults
initDtProductionLine
initDtSideTileBar
initDtTestingDeviceName
judgeIfExistInFaultHistory
proc1
queryDtHistoryValid
queryDtTitleGridShowMainForm


* 数据传输
learning
cloud_manage
文件
cloud_manage.sql
源
连接:
learning
数据库:
cloud_manage
信息
项目名: 我的连接
连接类型: MySQL
连接名: learning
主机: localhost
端口: 3306
服务器版本: 50726
目标
连接
文件
C:\Users\eivision\Desktop\cloud_manage.sql
SQL 格式:
与源服务器相同
MySQL
8.0
编码:
65001 (UTF-8)
保存配置文件
加载配置文件
选项
下一步


* 数据传输
learning
cloud_manage
文件
cloud_manage.sql
数据库对象
从"数据库对象"选择要传输的项目。若要自定义单个表的传输模式，请高亮显示"数据库对象"上的表。
表 (0/13)
运行期间的全部表 (*)
自定义 (0/13)
device
device_config
device_info
device_info_paranameandsuffix
device_info_threshold
faults
faults_config
faults_config_backup
faults_current
faults_current_backup
faults_history
faults_history_backup
productionline
视图 (0/0)
运行期间的全部视图 (*)
自定义 (0/0)
函数 (0/9)
运行期间的全部函数 (*)
自定义 (0/9)
initDtDeviceConfig
initDtFaults
initDtProductionLine
initDtSideTileBar
initDtTestingDeviceName
judgeIfExistInFaultHistory
proc1
queryDtHistoryValid
queryDtTitleGridShowMainForm
事件 (0/0)
运行期间的全部事件 (*)
自定义 (0/0)


learning
cloud_manage
表
device
device_cont
device_cont
device_info
device_info
device_info
device_info
device_info
device_info
device_info
faults
faults_confi
faults_confi
faults_current
faults_current_backup
faults_history
faults_history_backup
关闭数据库
编辑数据库...
新建数据库...
删除数据库
新建查询
命令列界面...
运行 SQL 文件...
转储 SQL 文件
打印数据库
逆向数据库到模型...
在数据库中查找
刷新

25. 函数

26.异常

27. 触发器

https://www.cnblogs.com/geaozhang/p/6819648.html

https://www.cnblogs.com/phpper/p/7587031.html

（1）概念

触发器是与表有关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据的完整性。

举个例子，比如你现在有两个表【用户表】和【日志表】，当一个用户被创建的时候，就需要在日志表中插入创建的log日志，如果在不使用触发器的情况下，你需要编写程序语言逻辑才能实现，但是如果你定义了一个触发器，触发器的作用就是当你在用户表中插入一条数据的之后帮你在日志表中插入一条日志信息。当然触发器并不是只能进行插入操作，还能执行修改，删除。

（2）使用

- **存储过程中不能创建触发器**
- **只有表支持触发器，视图和临时时表不支持**
- **！！尽量少使用触发器，不建议使用。**

假设触发器触发每次执行1s，insert table 500条数据，那么就需要触发500次触发器，光是触发器执行的时间就花费了500s，而insert 500条数据一共是1s，那么这个insert的效率就非常低了。因此我们特别需要注意的一点是触发器的begin end;之间的语句的执行效率一定要高，资源消耗要小。

触发器尽量少的使用，因为不管如何，它还是很消耗资源，如果使用的话要谨慎的使用，确定它是非常高效的：触发器是针对每一行的；对增删改非常频繁的表上切记不要使用触发器，因为它会非常消耗资源。

如果不需要某个触发器时一定要将这个触发器删除，以免造成意外操作，这很关键。

（3）语句

创建

- 单条语句

```
CREATE TRIGGER trigger_name trigger_time trigger_event ON tb_name FOR EACH ROW
trigger_stmt
```

> trigger_name：触发器的名称
> **tirgger_time**：触发时机，为BEFORE或者AFTER
>
> > after是先完成数据的增删改，再触发，触发器中的语句晚于监视的增删改，无法影响前面的增删该动作。就类似于先吃饭，再付钱。　　before是先完成触发，再增删改，触发的语句先于监视的增删改发生，我们有机会判断修改即将发生的操作。就类似于先付钱，再吃饭

> **trigger_event**：触发事件，为INSERT、DELETE或者UPDATE
> tb_name：表示建立触发器的表明，就是在哪张表上建立触发器
> **trigger_stmt**：触发器的程序体，可以是一条SQL语句或者是用BEGIN和END包含的多条语句
> 所以可以说MySQL创建以下六种触发器：
> BEFORE INSERT,BEFORE DELETE,BEFORE UPDATE
> AFTER INSERT,AFTER DELETE,AFTER UPDATE

```
mysql> CREATE TRIGGER trig1 AFTER INSERT
    -> ON work FOR EACH ROW
    -> INSERT INTO time VALUES(NOW());
```

- 多条语句

```
DELIMITER $
CREATE TRIGGER 触发器名 BEFORE|AFTER 触发事件
ON 表名 FOR EACH ROW
BEGIN
    执行语句列表
END$
DELIMITER ;
```

> BEGIN与END之间的执行语句列表参数表示需要执行的多个语句，不同语句用分号隔开

```
mysql> DELIMITER $
mysql> CREATE TRIGGER trig2 BEFORE DELETE
    -> ON work FOR EACH ROW
    -> BEGIN
    ->     INSERT INTO time VALUES(NOW());
    ->     INSERT INTO time VALUES(NOW());
    -> END$
mysql> DELIMITER ;
```

触发器类型

触发器类型	激活触发器的语句
INSERT型触发器	INSERT,LOAD DATA,REPLACE
UPDATE型触发器	UPDATE
DELETE型触发器	DELETE,REPLACE

NEW和OLD

MySQL 中定义了 NEW 和 OLD，用来表示触发器的所在表中，触发了触发器的那一行数据，来引用触发器中发生变化的记录内容，具体地：

①在INSERT型触发器中，NEW用来表示将要（BEFORE）或已经（AFTER）插入的新数据；

②在UPDATE型触发器中，OLD用来表示将要或已经被修改的原数据，NEW用来表示将要或已经修改为的新数据；

③在DELETE型触发器中，OLD用来表示将要或已经被删除的原数据；

使用方法：

NEW.columnName （columnName为相应数据表某一列名）

另外，OLD是只读的，而NEW则可以在触发器中使用 SET 赋值，这样不会再次触发触发器，造成循环调用（如每插入一个学生前，都在其学号前加“2013”）。

如：

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);

mysql> delimiter $$
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
    -> FOR EACH ROW
    -> BEGIN
    ->     IF NEW.amount < 0 THEN
    ->         SET NEW.amount = 0;
    ->     ELSEIF NEW.amount > 100 THEN
    ->         SET NEW.amount = 100;
    ->     END IF;
    -> END$$
mysql> delimiter ;

mysql> update account set amount=-10 where acct_num=137;

mysql> select * from account;
+----------+---------+
| acct_num | amount  |
+----------+---------+
|      137 |    0.00 |
|      141 | 1937.50 |
|       97 | -100.00 |
+----------+---------+

mysql> update account set amount=200 where acct_num=137;

mysql> select * from account;
+----------+---------+
| acct_num | amount  |
+----------+---------+
|      137 |  100.00 |
|      141 | 1937.50 |
|       97 | -100.00 |
+----------+---------+
```

查看触发器

```
SHOW TRIGGERS\G;
```

所有触发器信息都存储在information_schema数据库下的triggers表中，可以使用SELECT语句查询，如果触发器信息过多，最好通过TRIGGER_NAME字段指定查询

```
SELECT * FROM information_schema.triggers\G
```

该方法可以根据触发器名查询制定触发器的详细信息：

```
mysql> select * from information_schema.triggers
    -> where trigger_name='upd_check'\G;
```

删除触发器

```
drop trigger if exists trigger_name
```

（4）Navicat查看、修改触发器

https://blog.csdn.net/weixin_40486955/article/details/103871901

（5）触发器中调用存储过程

28.视图

https://www.jianshu.com/p/814d8aee700a

概念

从数据库系统内部来看，视图是由一张或多张表中的数据组成的

从数据库系统外部来看，视图就如同一张表一样，对表能够进行的一般操作都可以应用于视图，例如查询，插入，修改，删除操作等

视图不存储数据。

通俗理解就是将一张表中经常要查询的列和记录创建成一张虚拟的表，其实viewer视图中存放的是select语句 . 视图中看到的数据会随着原始表格的更新而动态更新

视图是由从数据库的基本表中选出来的数据组成的逻辑窗口，它与基本表不同的是，视图是一个虚表。数据库中只存放视图的定义，而不存放视图包含的数据，这些数据仍存放在原来的基表中。所以基表中的数据如果发生改变，从视图中查询出的数据也随之改变。

视图可以是建立在一个或多个表上，也可以建立在视图上，但是不管怎么样对视图数据的操作最终都会转换为对基本表的操作，因为视图是一个虚表，数据实际上保存在基本表中

用途

https://www.zhihu.com/question/21675233?from=profile_question_card

重用SQL语句；简化复杂的SQL操作；使用表的组成部分；保护数据；更改数据格式和表示。实现SQL语句的重用，简化复杂SQL语句（联结查询）的使用，给与用户一部分表（经过查询得出的表是原表的一部分）。

- 简化查询

  比如说要查询公司里面每个部门的员工数，并且按照人数由多到少进行排序，如果写SQL语句的话每次都要这么写：

```
select `dept_emp`.`dept_no` AS `dept_no`,count(1) AS `emp_sum` from
`dept_emp` group by `dept_emp`.`dept_no` order by `emp_sum` desc;
```

如果创建成视图的话，每次只要查询视图就行了

```
create VIEW `v_test` AS select `dept_emp`.`dept_no` AS `dept_no`,count(1) AS
`emp_sum` from `dept_emp` group by `dept_emp`.`dept_no` order by `emp_sum`
desc
```

充当临时表

有一个查询语句非常复杂，大概有100行这么多，有时还想把这个巨大无比的select语句和其他表关联起来得到结果，写太多很麻烦，可以用一个视图来代替这100行的select语句，充当一个变量角色

- 权限控制

  比如说公司A有一张用户表，公司B现在需要公司A用户表的用户信息，但是公司A不想让公司B获取用户的一些敏感信息（比如说家庭住址之类的），这时就可以公司A创建一个不含用户敏感信息的视图，然后再把这个视图的查询权限赋予公司B就可以了。

语句

建议在视图名前加上v_区分视图和表

`AS`表明用SELECT出来的表填充视图`productscustomers`

```
-- 创建
CREATE VIEW view_name
-- 删除
DROP VIEW view_name
-- 显示创建
SHOW CREATE VIEW view_name

-- 例子
CREATE VIEW productscustomers AS
SELECT cust_name,cust_contact,prod_id
FROM customers INNER JOIN orders ON customers.cust_id=orders.cust_id
INNER JOIN orderitems ON orders.order_num=orderitems.order_num;
-- 上面语句查询列出订购任意产品的客户
-- 现在想查询订购了产品'TN2'的客户
SELECT cust_name
FROM productscustomers
WHERE prod_id='TN2';
```

和临时表的区别

---

29. 临时表

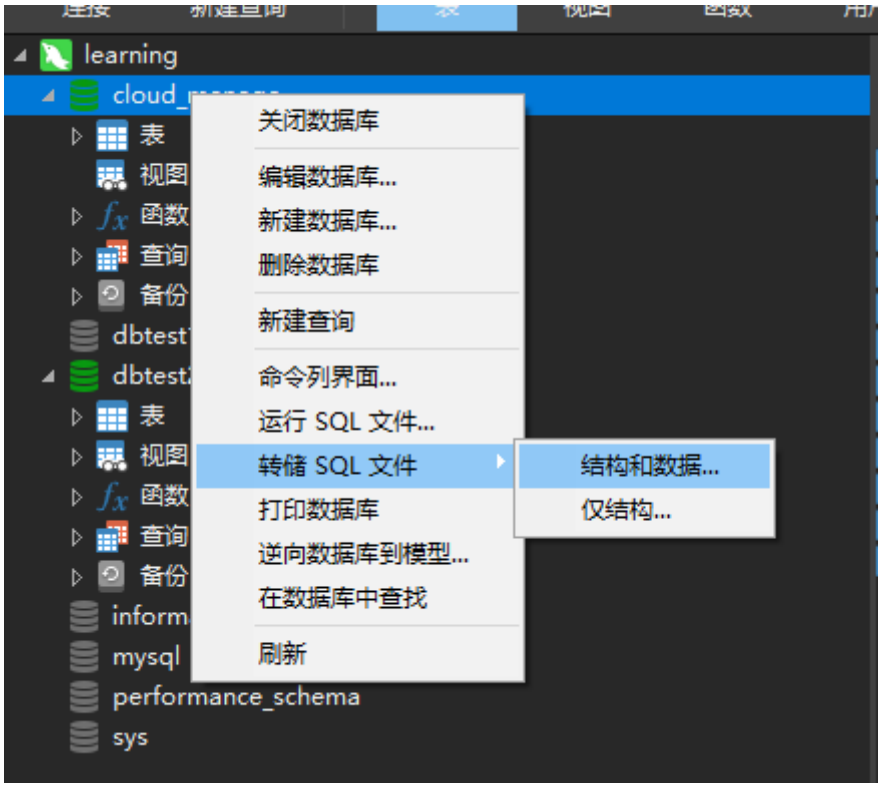
---

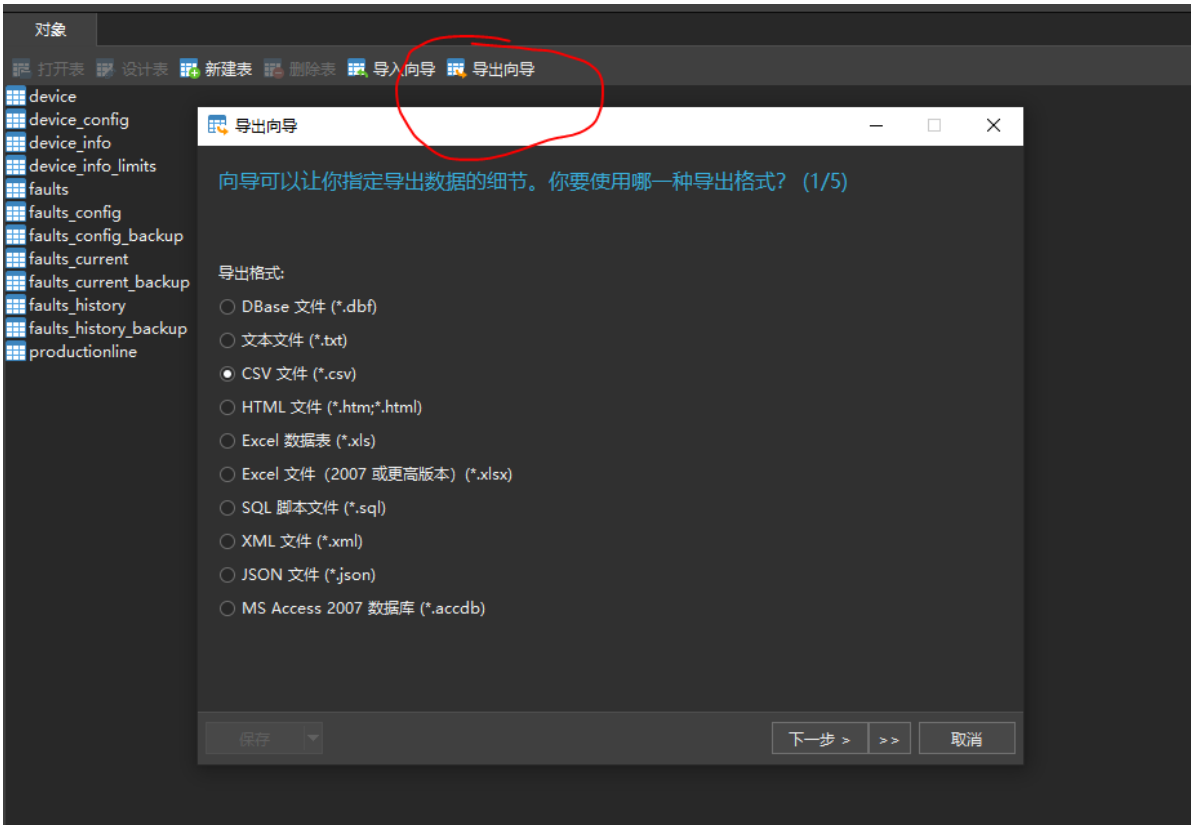
30. 导出导入

https://www.cnblogs.com/chenbin93/p/14697451.html

https://www.cnblogs.com/FengGeBlog/p/9974207.html

navicat右键，转储、运行SQL文件，导出导入.sql文件。导出/导入向导选择文件格式.csv。





(1) 导出.sql

- 导出数据库

```
-- 导出一个库
mysqldump  -u user_name -p pwd  dbname > Backup.sql;
-- 导出多个库，库名用空格隔开
mysqldump  -u user_name -p pwd  --databases dbname1  dbname2  dbname3  ....
> Backup.sql;
-- 导出所有库
mysqldump  -u user_name -p pwd  --all-databases > Backup.sql;
```

  Navicat——https://jingyan.baidu.com/article/ca00d56cba9926a99eebcfd6.html

- 导入数据库

  https://jingyan.baidu.com/article/a24b33cd2de7e219ff002b6b.html

- 导出表

```
-- 指定库名，表名用用空格隔开
mysqldump  -u user_name  -p pwd dbname table1 table2  table3... >
Backup.sql;
```

- 创建一个空库，use该库，source指令导入。

```
CREATE DATABASE db;
USE db;
SOURCE route.sql;
```

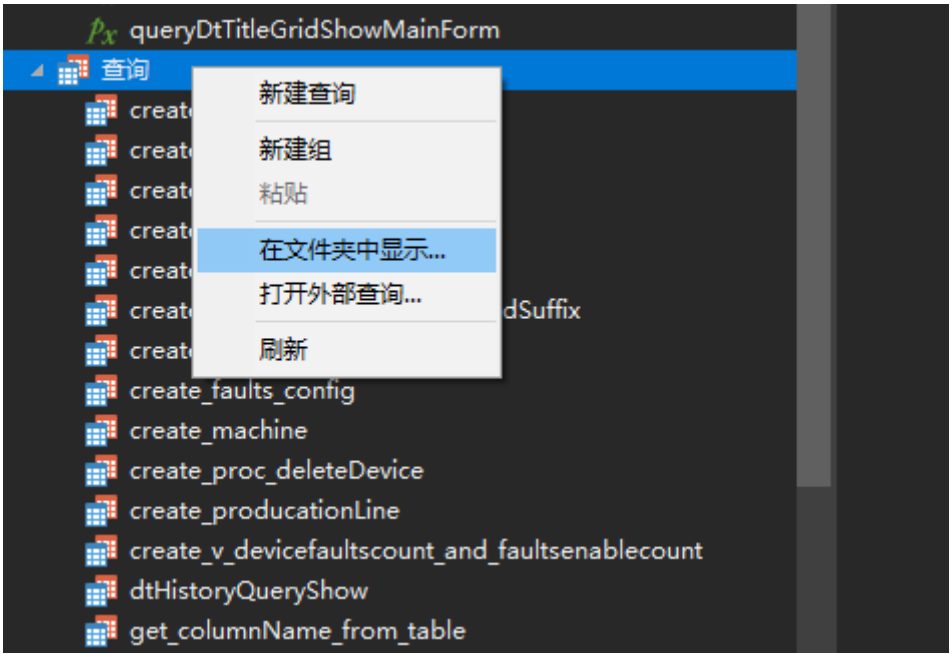
(2) 导出.csv

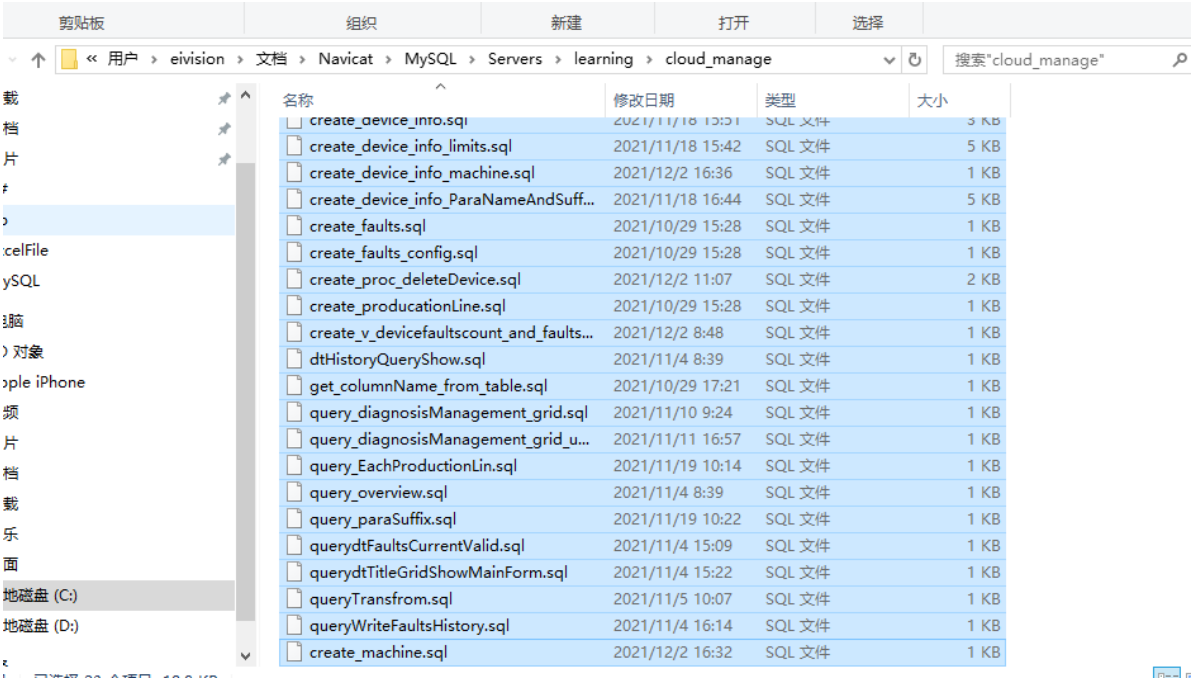
.csv文件可被Excel打开。

- 导出

- 导入

(3) 导出导入查询





一条查询对应一个*.sql*文件，导入时将文件放入文件夹即可，在查询页面*F5*刷新即可看见保存的查询

（4）表、存储过程、视图一起导出

见24.存储过程—(14)

---

31. 并发控制——分布式锁

https://blog.csdn.net/sunwenhao_2017/article/details/81565783

https://blog.csdn.net/u013474436/article/details/104924782/

（1）为什么要有并发控制（分布式锁）？

- 在单机时代，虽然不需要分布式锁，但也面临过类似的问题，只不过在单机的情况下，如果有多个线程要同时访问某个共享资源的时候，我们可以采用线程间加锁的机制，即当某个线程获取到这个资源后，就立即对这个资源进行加锁，当使用完资源之后，再解锁，其它线程就可以接着使用了。例如，在JAVA中，甚至专门提供了一些处理锁机制的一些API（synchronize/Lock等）。

但是到了分布式系统的时代，这种线程之间的锁机制，就没作用了，**系统可能会有多份并且部署在不同的机器上**，这些资源已经不是在线程之间共享了，而是属于**进程之间共享的资源。**

因此，为了解决这个问题，我们就必须引入分布式锁。**分布式锁是指在分布式的部署环境下，通过锁机制来让多客户端互斥的对共享资源进行访问。**

- 当程序中可能出现并发的情况时，就需要保证在并发情况下数据的准确性，以此确保当前用户和其他用户一起操作时，所得到的结果和他单独操作时的结果是一样的。这就叫做并发控制。并发控制的目的是保证一个用户的工作不会对另一个用户的工作产生不合理的影响。

  没有做好并发控制，就可能导致脏读、幻读和不可重复读等问题。

- 在多用户环境中，在同一时间可能会有多个用户更新相同的记录，这会产生冲突。这就是著名的并发性问题。
  典型的冲突有：
  1.**丢失更新**：一个事务的更新覆盖了其它事务的更新结果，就是所谓的更新丢失。例如：用户A把值从6改为2，用户B把值从2改为6，则用户A丢失了他的更新。
  2.**脏读**：当一个事务读取其它完成一半事务的记录时，就会发生脏读取。例如：用户A,B看到的值都是6，用户B把值改为2，用户A读到的值仍为6。 为了解决这些并发带来的问题。 我们需要引入并发控制机制。

- 常说的并发控制，一般都和数据库管理系统(DBMS)有关。在 DBMS 中并发控制的任务，是确保多个事务同时增删改查同一数据时，不破坏事务的隔离性、一致性和数据库的统一性。

  实现并发控制的主要手段分为**乐观并发控制和悲观并发控制两种。**
  无论是悲观锁还是乐观锁，都是人们定义出来的概念，可以认为是一种思想。其实不仅仅是关系型数据库系统中有乐观锁和悲观锁的概念，像 hibernate、tair、memcache 等都有类似的概念。所以，不应该拿乐观锁、悲观锁和其他的数据库锁等进行对比。**乐观锁比较适用于读多写少的情况(多读场景)，悲观锁比较适用于写多读少的情况(多写场景)。**

（2）分布式锁概念

> - 分布式锁使用者位于不同的机器中，锁获取成功之后，才可以对共享资源进行操作
> - 锁具有重入的功能：即一个使用者可以多次获取某个锁
> - 获取锁有超时的功能：即在指定的时间内去尝试获取锁，超过了超时时间，如果还未获取成功，则
>   返回获取失败
> - 能够自动容错，比如：A机器获取锁lock1之后，在释放锁lock1之前，A机器挂了，导致锁lock1未
>   释放，结果会lock1一直被A机器占有着，遇到这种情况时，分布式锁要能够自动解决，可以这么
>   做：持有锁的时候可以加个持有超时时间，超过了这个时间还未释放的，其他机器将有机会获取锁

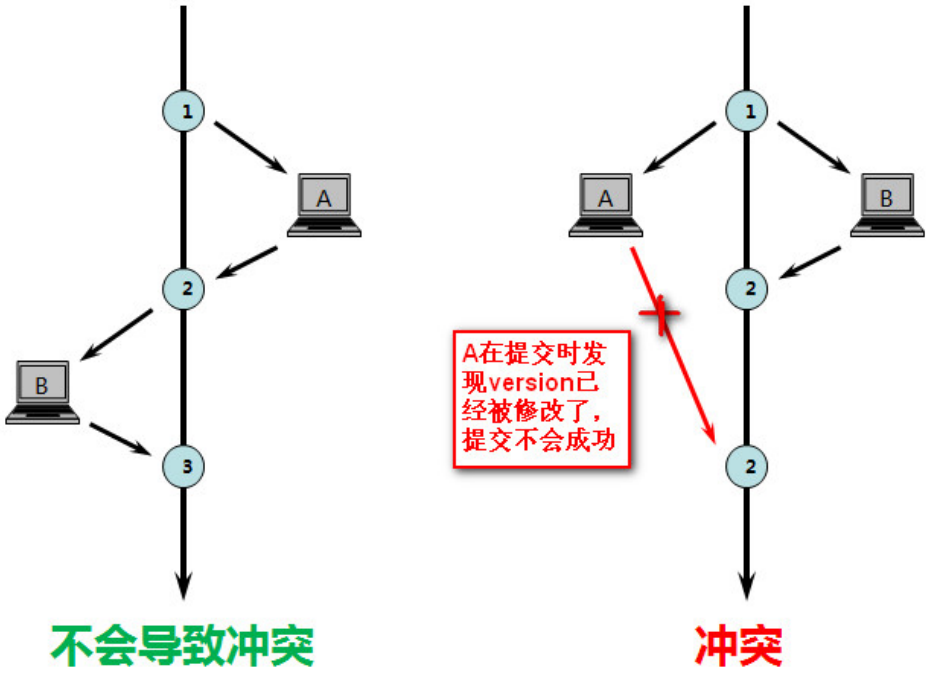
（3）分布式锁的实现方法

数据库锁现在使用较多的就上面说的3种方式，**排他锁（悲观锁）**，**版本号(乐观锁)**，**记录锁**，各有优缺点

乐观锁（optimistic_lock）

乐观锁（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。那么我们如何实现乐观锁呢，一般来说有以下2种方式：

1. 使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。用下面的一张图来说明：



如上图所示，如果更新操作顺序执行，则数据的版本（version）依次递增，不会产生冲突。但是如果发生有不同的业务操作对同一版本的数据进行修改，那么，先提交的操作（图中B）会把数据version更新为2，当A在B之后提交更新时发现数据的version已经被修改了，那么A的更新操作会失败。

2. 乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp），和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

32. MySQL中间件

https://www.cnblogs.com/armyfai/p/13595055.html

https://blog.csdn.net/hanguofei/article/details/103465363

https://www.cnblogs.com/zhou2019/p/10918131.html

33. 游标

34. 账号管理

- 严肃对待root账号的使用，仅在绝对需要的时候才使用。
- 账号、密码存在mysql数据库中，user表的user列存储用户名。

  不要直接操作mysql数据库中数据。

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

- 创建用户账号

```
CREATE USER user_name IDENTIFIED BY 'password';
```

  创建好的账号没设置权限，可登录但不能看任何数据，也不能进行任何操作。

- 查看权限

```
SHOW GRANTS FOR user_name;
```

- 授予权限

```
-- 给账号授予在库上查询所有数据的权限
GRANT SELECT ON database_name.* TO user_name;
-- 授予在表上查询的权限
GRANT SELECT ON database_name.t_name TO user_name;
-- 整个服务器
GRANT ALL
-- 库
ON database_name.*
-- 表
ON database_name.t_name
-- 列
```

- 撤销权限

```
REVOKE SELECT ON database_name.* FROM user_name;
```

- 重命名用户账号

```
RENAME USER user_name_old TO user_name_new;
```

- 删除用户账号

```
DROP USER user_name;
```

- 修改密码

```
-- 为当前user修改密码
SET PASSWORD = PASSWORD('password');
-- 修改指定用户密码
SET PASSWORD FOR user_name = PASSWORD('password');
```

35. 日志

https://blog.csdn.net/defonds/article/details/46858949

需要配置启用日志。

36.存储图片

图片/视频不直接存在数据库中（要以二进制数据存），而是存在文件系统中，将图片的路径存在数据库中。

37.MySQL与NoSQL

http://www.cppblog.com/sunicdavy/archive/2015/07/20/210992.html

用了NoSQL系列的数据库, 才意识到: 游戏服务器的数据存储和游戏服务器的存盘两个概念差异其实蛮大的.

MySQL中, 背包其实跟角色完全没有关系, 只是通过1个角色id映射过去, 人为的割裂了数据的关联性. 还硬生生的整出个概念叫结构化查询让你学

NoSQL中, 只是把数据库当成是存储点, 每个角色的数据是完整的一块. 里面怎么存随你便. 每个角色通过id来查询, 其他都没有了

于是乎, 游戏开发变得异常简单. MySQL角色进门查询4~5次才能搞定要的数据.而NoSQL一口气全查出来, 存盘也无需增量, 直接存盘就可以了

所以现在觉得, NoSQL的思路对于游戏服务器存储来说简直是完美的!

NoSQL下实现方案很多, 游戏常用的就这么3家: mongo, redis, memcached

下面说下优缺点

mongo

磁盘映射内存数据库

value为document类型, 基于BSON的value序列化

应用场景:

适合多写少读, 例如日志和备份

转载请注明: 战魂小筑http://www.cppblog.com/sunicdavy

redis

内存数据库

单核

value限制512M

多种value类型, 游戏用途使用私有的序列化协议(例如protobuf)

支持落地(bgsave)

用户: 新浪, 淘宝, Flickr, Github

应用场景: 适合读写都很高, 数据处理复杂等

转载请注明: 战魂小筑http://www.cppblog.com/sunicdavy

memcached

内存数据库

多核

value限制1M

不支持落地(持久化)

用户: LiveJournal、hatena、Facebook、Vox

应用场景: 动态系统中的缓冲, 适合多读少写

---

38. MySQL规范及性能优化

（1）MySQL三表禁止join

很有启发的一篇blog

（2）表联结的字段必须加索引