**MySQL学习笔记**

官网：http://dev.mysql.com

# 1 MySQL介绍与安装

库与表的概念

表：多行多列的数据。

库：多个表形成一个库。

多个库存放在数据库服务器上。

SQL概念

SQL是结构化查询语言。

DML：数据操作语言，数据的增删改查，使用者角度。

DDL：数据定义语言，建表、建库、建视图等，建设者的角度。

DCL：数据控制语言，数据管理库权限，管理者角度。

## 1.1 数据库的连接

**mysql -h -u -p**

-h host主机

-u 用户名

-p 密码

例如：mysql -h192.168.8.102 -uroot -p123456

## 1.2 库级知识

显示数据库: show databases;

选择数据库: use dbname;

创建数据库: create database dbname charset utf8;

删除数据库: drop database dbname;

## 1.3 安装mysql

安装在Centos7系统上

|  |  |
| --- | --- |
| 安装前创建用户和数据文件存储文件夹 | mkdir -p /data/mysql  groupadd mysql  useradd -g mysql mysql  chown mysql.mysql -R /data/mysql  安装依赖包：yum -y install ncurses-devel |
| 下载mysql源码 | wget http://dev.mysql.com/get/Downloads/MySQL-5.7/mysql-boost-5.7.15.tar.gz |
| 解压 | tar -zxvf mysql-boost-5.7.15.tar.gz |
| 进入目录 | cd mysql-5.7.15 |
| 检查编译环境 | cmake \  -DCMAKE\_INSTALL\_PREFIX=/usr/local/mysql \  -DMYSQL\_DATADIR=/data/mysql \  -DSYSCONFDIR=/usr/local/mysql \  -DMYSQL\_USER=mysql \  -DWITH\_MYISAM\_STORAGE\_ENGINE=1 \  -DWITH\_INNOBASE\_STORAGE\_ENGINE=1 \  -DWITH\_ARCHIVE\_STORAGE\_ENGINE=1 \  -DWITH\_MEMORY\_STORAGE\_ENGINE=1 \  -DWITH\_READLINE=1 \  -DMYSQL\_UNIX\_ADDR=/var/run/mysql/mysql.sock \  -DMYSQL\_TCP\_PORT=3306 \  -DENABLED\_LOCAL\_INFILE=1 \  -DENABLE\_DOWNLOADS=1 \  -DWITH\_PARTITION\_STORAGE\_ENGINE=1 \  -DEXTRA\_CHARSETS=all \  -DDEFAULT\_CHARSET=utf8 \  -DDEFAULT\_COLLATION=utf8\_general\_ci \  -DWITH\_DEBUG=0 \  -DMYSQL\_MAINTAINER\_MODE=0 \  -DWITH\_SSL:STRING=bundled \  -DWITH\_ZLIB:STRING=bundled \  -DDOWNLOAD\_BOOST=1 \  -DWITH\_BOOST=/home/vison/mysql-5.7.15/boost  **问题**：编译MySQL5.7以及更高的版本时，都需要下载并引用或者直接安装boost库，否则在执行 cmake 命令时会报如下错误，在下载mysql源码时最好下载带有boost库的版本。  **解决办法**：在cmake命令后面添加参数  -DDOWNLOAD\_BOOST=1 -DWITH\_BOOST=Boost库路径 |
| 编译安装 | make && make install  注：编译非常耗时间和cpu，内存建议2G以上 |
| 添加mysql的环境变量 | export PATH=/usr/local/mysql/bin:$PATH && source /etc/profile |
| 初始化MySQL自身的数据库 | /usr/local/mysql/bin/mysqld --initialize-insecure --user=mysql --basedir=/usr/local/mysql --datadir=/data/mysql  注：指定安装目录和数据存放目录 |
| 创建MySQL日志存放目录 | #创建目录  mkdir -p /var/run/mysql && mkdir -p /var/log/mysql  #修改用户  chown -R mysql:mysql /var/log/mysql && chown -R mysql:mysql /var/run/mysql  注：MySQL日志存放目录以及权限都是根据前面my.cnf文件写的，也就是两者需要保持一致，配置后检测是否一致 |
| 设置mysqld的开机启动 | cp /usr/local/mysql/support-files/mysql.server /etc/init.d/mysqld  chmod +x /etc/init.d/mysqld  chkconfig --add mysqld  chkconfig mysqld on |

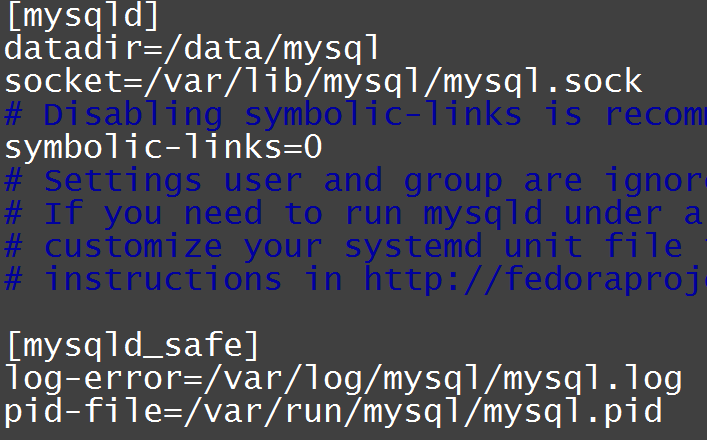
修改配置文件：

**vim /usr/local/mysql/my.cnf**

在[msqld]字段下指定datadir数据文件存放路径和socket连接文件路径。

在[mysqld\_safe]字段下指定log-error错误日志路劲和pid-file进程pid文件路径。

修改如下图所示：



启动mysql服务：**systemctl start mysql**

重启或停止mysql服务是把start改为restart或stop

查看是否启动成功：**ps -ef | grep mysqld**

登陆mysql：**mysql -uroot -p**

# 2 表的增删改查操作

## 2.1 insert 插入数据操作

规则： 1）往哪张表添加行？ **INSERT INTO 表名**

2）给那几个列添加值？ **（列名称）可省略**

3）分别是什么值？ **VALUES （列对应的值）**

插入操作时注意事项：列和值必须一一对应，并且符合类型要求。

|  |
| --- |
| **INSERT INTO user** (**uid**, **name**, **age**) **VALUES** (1, **'张三'**, 25); **INSERT INTO user** (**name**, **age**) **VALUES** (**'李四'**, 26); **INSERT INTO user** (**name**) **VALUES** (**'王五'**); **INSERT INTO user VALUES** (5, **'刘备'**, 52); *# 忽略列时需要写完所有对应列的值 #INSERT INTO user VALUES ('关羽',45);# 列没有严格对应，执行错误*  **INSERT INTO user** (**name**, **age**) **VALUES** (**'关羽'**, 45),(**'张飞'**, 46);*# 一次插入多行数据* |

## 2.2 update 修改数据操作

规则： 1）改哪一张表？ **UPDATE 表名**

2）改哪几列的值？ **SET 列名=值1，列名=值2......**

3）在哪些行生效？ **WHERE 表达式**

注意事项：一定要有where约束条件，即在哪些生效。

|  |
| --- |
| **UPDATE user SET age**=27 **WHERE name**=**'王五'**; **UPDATE user SET name**=**'赵六'**,**age**=28 **WHERE uid**=4; *# 修改多列用逗号隔开* |

## 2.3 delete 删除数据操作

表数据最小单位是行，所以是按行删除，不必指定列。

规则： 1）删除一张表的数据？ **DELETE FROM 表名**

2）删掉表中的哪些行？ **WHERE 表达式**

注意事项：删除必须写where约束条件，也不能写常量，如where 1，否则会删除整张表数据。

|  |
| --- |
| **DELETE FROM user WHERE age**=26;  **DELETE FROM user WHERE uid**>2; |

## 2.4 select 查询数据操作

规则： 1）查询哪些列数据？ **SELECT列名1 列名2......**

2）从哪张表查询？ **FROM 表名**

3）选择哪些行生效？ **WHERE** **表达式**

|  |
| --- |
| **SELECT** *\** **FROM user**; *# 实际开发中很少使用* **SELECT** *\** **FROM user WHERE name**=**'关羽'**; **SELECT name**,**age FROM user WHERE age**<30; *# 查询符合条件的指定列* |

# 3 综合查询操作

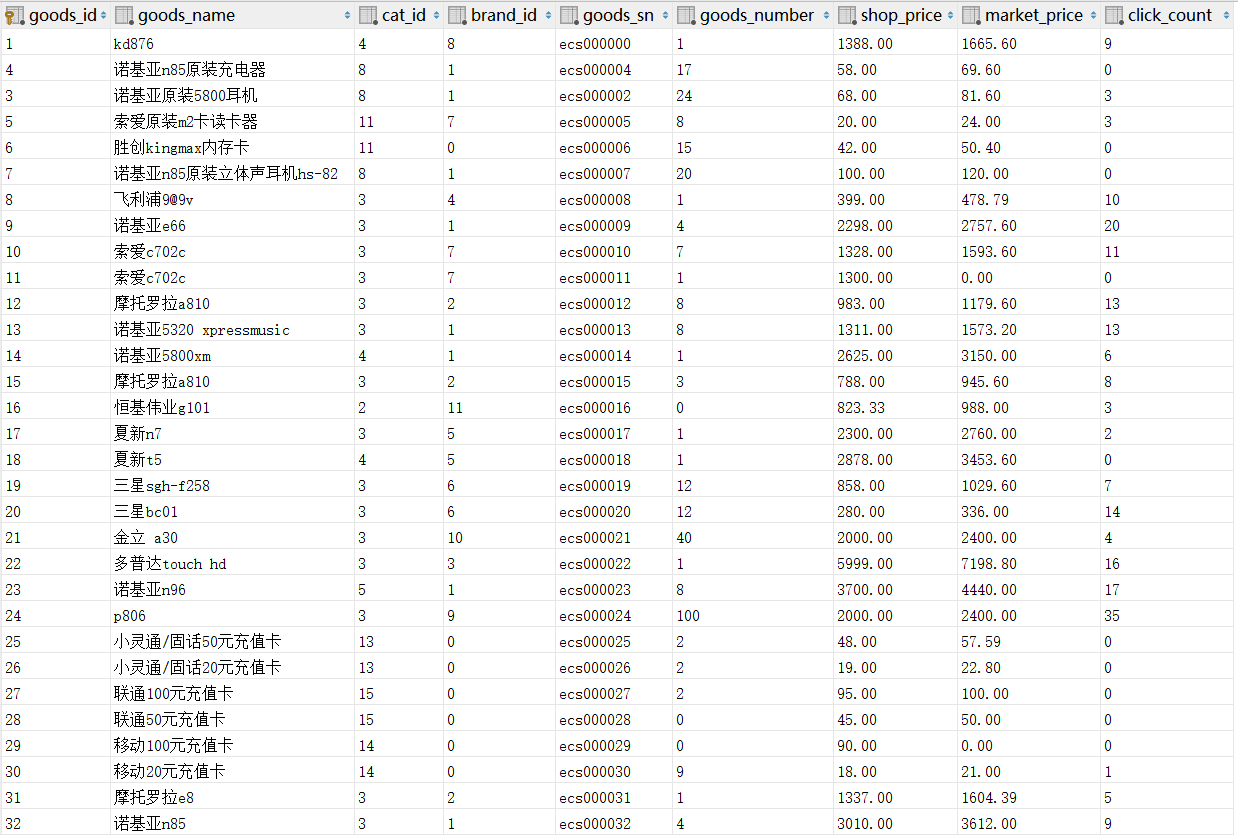
MySQL的比较运算和逻辑运算符如下图所示，注意and比or的优先级高，当表达式比较复杂时，习惯使用括号，避免出现歧义。

in用法：in的后面跟着枚举，例如in (3,11)

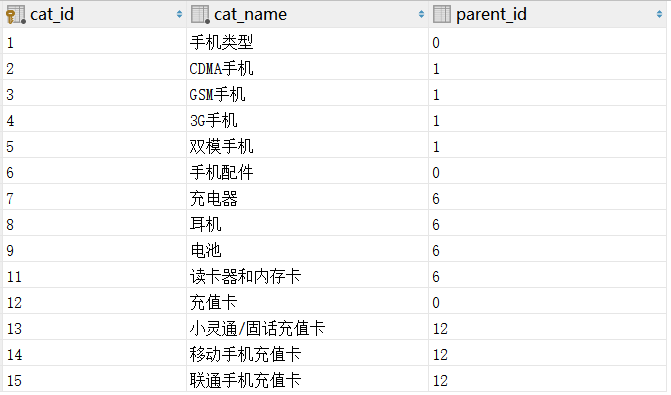
between通常和 and连用，例如 between 100 and 500



练习使用的商品表如下图所示：



练习使用的栏目表如下图所示：



## 3.1 where 条件约束

where是针对磁盘数据文件，后面一般都是列变量的表达式。

模糊查询时，百分号%表示匹配所有字符，下划线\_表示匹配一个字符。

|  |
| --- |
| **⑴** 主键为32的商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE goods\_id**=32;  **⑵** 不属第3栏目的所有商品  **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE cat\_id**!=3;  **⑶** 本店价格高于3000元的商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE shop\_price**>3000;  **⑷** 本店价格低于或等于100元的商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE shop\_price**<=100;  **⑸** 取出第4栏目或第11栏目的商品(不用or)  **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE cat\_id IN** (4,11);  **⑹** 取出100<=价格<=500的商品(不许用and)  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE shop\_price BETWEEN** 100 **AND** 500;  **⑺** 不属于第3栏目且不属于第11栏目的商品(and,或not in分别实现)  ① **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE** (**cat\_id**!=3) && (**cat\_id**!=11);  ② **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE cat\_id NOT IN** (3,11);  **⑻** 取出价格大于100且小于300,或者大于3000小于4000商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE** (**shop\_price**>100 && **shop\_price**<300) || (**shop\_price**>3000 && **shop\_price**<4000);  **⑼** 取出第3个栏目下面价格<1000或>3000,并且点击量>5的系列商品  **SELECT cat\_id**,**goods\_name**,**shop\_price**,**click\_count FROM** goods **WHERE** (**cat\_id**=3) && (**shop\_price**<1000 || **shop\_price**>3000) && (**click\_count**>=5);  **⑽** 取出第1个栏目下面的商品(注意:第1栏目下面没商品,但其子栏目下有)  **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE cat\_id IN** (2,3,4,5);*# 手动查看第1栏目的子栏目有2 3 4 5*  **⑾** 取出名字以"诺基亚"开头的商品  **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE goods\_name LIKE '诺基亚%'**;  **⑿** 取出诺基亚N系列的手机  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE goods\_name LIKE '诺基亚N\_\_'**;  **⒀** 取出名字不以"诺基亚"开头的商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE goods\_name NOT LIKE '诺基亚%'**;  **⒁** 取出第3个栏目下面价格在1000到3000之间,并且点击量>5 的"诺基亚"开头的商品  **SELECT cat\_id**,**goods\_name**,**shop\_price**,**click\_count FROM** goods **WHERE** (**cat\_id**=3) && (**shop\_price**>1000 && **shop\_price**<3000) && (**click\_count**>5) && (**goods\_name LIKE '诺基亚%'**);  **⒂** 把num值处于[20,29]之间,改为20，num值处于[30,39]之间的,改为30  **UPDATE** mian **SET num**=*floor*(**num**/10)\*10 **WHERE num BETWEEN** 20 **AND** 39;  **⒃** 把good表中商品名为'诺基亚xxxx'的商品,改为'HTCxxxx'  **UPDATE** goods **SET goods\_name**=*INSERT*(**goods\_name**,1,3,**'HTC'**) **WHERE goods\_name LIKE '诺基亚%'**; *# 注：字符串位置从1开始* |

## 3.2 group 数据分组

使用group时首先会对该列重新排序，再做统计，所以比较耗资源，尽量避免使用。

|  |
| --- |
| **⑴** 查出最贵和最便宜的商品的价格  **SELECT** *max*(**shop\_price**),*min*(**shop\_price**) **FROM** goods;  **⑵** 查出最新和最旧的商品编号  **SELECT** *max*(**goods\_id**),*min*(**goods\_id**) **FROM** goods;  **⑶** 查询该店所有商品的库存总量  **SELECT** *sum*(**goods\_number**) **FROM** goods;  **⑷** 查询所有商品的平均价格  **SELECT** *avg*(**shop\_price**) **FROM** goods;  **⑸** 查询该店一共有多少种商品  **SELECT** *count*(*\**) **FROM** goods;  **⑹** 查询每个栏目下面最贵商品价格、最便宜商品价格、商品平均价格、商品库存量、商品库存总价格、商品种类  **SELECT** *max*(**shop\_price**),*min*(**shop\_price**),*avg*(**shop\_price**),*sum*(**goods\_number**),*count*(*\**) **FROM** goods **GROUP BY cat\_id**; |

## 3.3 having 筛选数据

having是针对where条件结果进行筛选，是对内存数据操作，所以having必须用在where之后。

|  |
| --- |
| **⑴** 查询该店的商品比市场价所节省的价格  **SELECT goods\_name**,**market\_price**-**shop\_price FROM** goods;  **⑵** 查询每个商品所积压的货款  **SELECT goods\_name**,**shop\_price**\***goods\_number FROM** goods;  **⑶** 查询该店积压的总货款  **SELECT** *sum*(**shop\_price**\*goods.**goods\_number**) **FROM** goods;  **⑷** 查询该店每个栏目下面积压的货款  **SELECT cat\_id**,*sum*(**shop\_price**\***goods\_number**) **FROM** goods **GROUP BY cat\_id**;  **⑸** 查询比市场价省钱200元以上的商品及该商品所省的钱  **SELECT goods\_id**,**goods\_name**,(**market\_price**-goods.**shop\_price**) **AS** save **FROM** goods **HAVING** save>200;  **⑹** 查询积压货款超过2W元的栏目,以及该栏目积压的货款  **SELECT cat\_id**,*sum*(**shop\_price**\*goods.**goods\_number**) **AS** catPrice **FROM** goods **GROUP BY cat\_id HAVING** catPrice>20000;  **⑺** 查询出2门及2门以上不及格者的平均成绩    **SELECT name**,*avg*(**score**),*sum*(**score**<60) **AS** fail **FROM** result **GROUP BY name HAVING** fail>=2; |

## 3.4 order by 排序和limit限制取出条目

升序：asc（默认）

降序：desc

当按某列排序无法满足要求时，可以在列的内部再继续排序。

实际使用中oder by经常和limit配合一起使用。

limit有两个参数，分别是limit 偏移量，取出条目。可以使用在分页上。

|  |
| --- |
| (1)按价格由高到低排序  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **ORDER BY shop\_price ASC**;  (2)按发布时间由早到晚排序  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **ORDER BY goods\_id DESC**;  (3)接栏目由低到高排序,栏目内部按价格由高到低排序  **SELECT cat\_id**,**goods\_name**,**shop\_price FROM** goods **ORDER BY cat\_id ASC**, **shop\_price DESC**;  (4)取出价格最高的前三名商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **ORDER BY shop\_price DESC LIMIT** 0,3;  (5)取出点击量前三名到前5名的商品  **SELECT goods\_name**,**shop\_price**,**click\_count FROM** goods **ORDER BY click\_count DESC LIMIT** 0,5; |

## 3.5 子查询

### 3.5.1 where子查询

内层select查询的结果充当外层select的where条件的查询。

|  |
| --- |
| 取出每个栏目下最新的商品  **SELECT cat\_id**,**goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE goods\_id IN** (**SELECT** *max*(**goods\_id**) **FROM** goods **GROUP BY cat\_id**); |

### 3.5.2 from子查询

内层select作为一张表(AS table)，外层select从内层表取出数据。

|  |
| --- |
| 取出每个栏目下最新的商品  **SELECT cat\_id**,**goods\_id**,**goods\_name**,**shop\_price FROM** (**SELECT** *\** **FROM** goods **ORDER BY cat\_id ASC**,**goods\_id DESC**) **AS** tmp **GROUP BY cat\_id**; |

### 3.5.3 exists子查询

exists指定一个子查询，检测行的存在。该子查询实际上并不返回任何数据，而是返回值True或False。

exists子查询能完成的where子查询 in也能完成。

|  |
| --- |
| 取出没有商品的栏目  **SELECT** *\** **FROM** category **WHERE NOT** *exists*(**SELECT** *\** **FROM** goods **WHERE** category.**cat\_id**=goods.**cat\_id**); |

### 3.5.4 子查询练习

|  |
| --- |
| (1)查询出最新一行商品  **SELECT goods\_id**,**goods\_name**,**shop\_price FROM** goods **WHERE goods\_id**=(**SELECT** *max*(**goods\_id**) **FROM** goods);  (2)查询出编号为19的商品的栏目名称(where和连接查询)  where子查询：  **SELECT cat\_id**,**cat\_name FROM** category **WHERE cat\_id**=(**SELECT cat\_id FROM** goods **WHERE goods\_id**=19);  连接查询：  **SELECT** category.**cat\_id**,category.**cat\_name FROM** category **INNER JOIN** goods **ON** category.**cat\_id**=goods.**cat\_id  WHERE** goods.**goods\_id**=19;  (3)用where和from型子查询方式把每个栏目下面最新的商品取出来  where子查询：  **SELECT goods\_id**,**cat\_id**,**goods\_name FROM** goods **WHERE goods\_id IN** (**SELECT** *max*(**goods\_id**) **FROM** goods **GROUP BY cat\_id**);  from子查询：  **SELECT goods\_id**,**cat\_id**,**goods\_name FROM** (**SELECT** *\** **FROM** goods **ORDER BY cat\_id ASC**,**goods\_id DESC**) **AS** tmp **GROUP BY** tmp.**cat\_id**;  (4)用exists型子查询,查出所有有商品的栏目  **SELECT** *\** **FROM** category **WHERE** *exists*(**SELECT** *\** **FROM** goods **WHERE** category.**cat\_id**=goods.**cat\_id**); |

## 3.6 连接查询

### 3.6.1 INNER JOIN内连接查询

把两张独立代表拼接成一张大表，拼接时指定匹配条件。

格式：表1 INNER JOIN 表2 ON 匹配条件

|  |
| --- |
| 把boy表和girl表同一组的人取出来  **SELECT** *\** **FROM** boy **INNER JOIN** girl **ON** boy.**hid**=girl.**hid**; |

### 3.6.2 LEFT JOIN左连接查询

以左边的表为标准，有匹配条件取出值来，没有则填充默认值。

|  |
| --- |
| 以boy表为标准，把boy表和girl表进行同组匹配  **SELECT** *\** **FROM** boy **LEFT JOIN** girl **ON** boy.**hid**=girl.**hid**; |

### 3.6.3 RIGHT JOIN右连接查询

以右边的表为标准，有匹配条件取出值来，没有则填充默认值。

|  |
| --- |
| 以girl表为标准，把boy表和girl表进行同组匹配  **SELECT** *\** **FROM** boy **RIGHT JOIN** girl **ON** boy.**hid**=girl.**hid**; |

|  |
| --- |
| (1)取出所有商品的商品名,栏目名,价格  **SELECT** goods.**goods\_id**,goods.**goods\_name**,category.**cat\_name**,goods.**shop\_price FROM** goods **LEFT JOIN** category **ON** goods.**cat\_id**=category.**cat\_id**;  (2)取出第4个栏目下的商品的商品名,栏目名,价格  **SELECT** goods.**goods\_id**,goods.**goods\_name**,category.**cat\_name**,goods.**shop\_price FROM** goods **LEFT JOIN** category **ON** goods.**cat\_id**=category.**cat\_id WHERE** goods.**cat\_id**=4;  (3) 查出 2006-6-1 到2006-7-1之间举行的所有比赛，并且用以下形式列出：主队名 比分 客队 比赛时间  表名：m  +-----+------+------+------+------------+  | mid | hid | gid | mres | matime |  +-----+------+------+------+------------+  | 1 | 1 | 2 | 2:0 | 2006-05-21 |  | 2 | 2 | 3 | 1:2 | 2006-06-21 |  | 3 | 3 | 1 | 2:5 | 2006-06-25 |  | 4 | 2 | 1 | 3:2 | 2006-07-21 |  +-----+------+------+------+------------+  队名：t  +------+----------+  | tid | tname |  +------+----------+  | 1 | 国安 |  | 2 | 申花 |  | 3 | 公益联队 |  +------+----------+  写法一：  **SELECT** ttb.**tname**,ttb.**mres**,t.**tname**,ttb.**matime FROM**  (**SELECT** *\** **FROM** m **LEFT JOIN** t **ON** m.**hid**=t.**tid**)**AS** ttb  **LEFT JOIN** t **ON** ttb.**gid**=t.**tid**  **WHERE** ttb.**matime BETWEEN '2006-06-01' AND '2006-07-01'**;  写法二：  **SELECT** t1.**tname**,**mres**,t2.**tname**,**matime FROM** m **LEFT JOIN** t **AS** t1 **ON** m.**hid** = t1.**tid  LEFT JOIN** t **AS** t2 **ON** m.**gid** = t2.**tid  WHERE matime BETWEEN '2006-06-01' AND '2006-07-01'**; |

## 3.7 union查询

把两条或多条sql查询的结果合并成一个结果集。

主要用途：两张不同的表中有相同的列时，把结果集合并。或者简化两个复杂的where条件。

使用条件：两个表的列必须相同，但是例外的是列名称可以不相同。

注意：

①完全相等的行将会合并，合并是耗时的操作，一般不让union合并，使用union all则不合并。

②union子句中一般不用order by，在union合并后可以再order by。

|  |
| --- |
| (1)合并表a和表b，不合并相同的行  **SELECT** *\** **FROM** a **UNION ALL SELECT** *\** **FROM** b;  (2)对两个表相同id的行进行num列求和。  **SELECT** tmp.**id**,*sum*(tmp.**num**) **FROM**  (**SELECT** *\** **FROM** a **UNION ALL SELECT** *\** **FROM** b) **AS** tmp  **GROUP BY** tmp.**id**; |

# 4 表的列操作

## 查看表字段信息

DESC表名

例如：DESC student;

## 查看表的所有信息

SHOW CREATE TABLE 表名;

|  |
| --- |
| **SHOW CREATE TABLE** student; |

## 查看当前库下所有表信息

**SHOW TABLE status**;

或列形式输出：**SHOW TABLE status \G**;

或过滤表输出：**SHOW TABLE status WHERE name**=**'goods' \G**;

## 新建表

CREATE TABLE 表名(列名 列类型 列属性，……);

|  |
| --- |
| **CREATE TABLE** student (  **sid TINYINT UNSIGNED**,  **name VARCHAR**(20),   **age INT** ); |

## 添加列

ALTER TABLE 表名 ADD COLUMN 列名 列类型 列属性，……

|  |
| --- |
| 在最后添加列：  **ALTER TABLE** student **ADD COLUMN sn TINYINT**(6) **ZEROFILL**;  在指定位置添加列：  **ALTER TABLE user ADD height TINYINT** **AFTER weight**; |

## 删除表

DROP TABLE 表名;

|  |
| --- |
| **DROP TABLE** t1; |

## 清空表内容

TRUNCATE 表名;

TRUNCATE相当与drop和creat表两个操作。

## 删除列

ALTER TABLE 表名 DROP 列名;

|  |
| --- |
| **ALTER TABLE** student **DROP COLUMN** sn; |

## 修改表名

ALTER TABLE 旧表名 RENAME TO 新表名;

|  |
| --- |
| **ALTER TABLE** student **RENAME TO** stu; |

## 修改列属性

ALTER TABLE表名MODIFY sid 列名 列类型 列属性;

|  |
| --- |
| **ALTER TABLE** student **MODIFY sid TINYINT UNSIGNED**; |

## 修改列名称和属性

ALTER TABLE 表名 CHANGE 旧列名 新列名 列类型 列属性;

|  |
| --- |
| **ALTER TABLE** student **CHANGE sid** **id INT UNSIGNED**; |

## 添加主键约束

ALTER TABLE 表名 ADD CONSTRAINT PRIMARY KEY 表名(列名);

|  |
| --- |
| **ALTER TABLE** student **ADD CONSTRAINT PRIMARY KEY** student(**sid**); |

## 删除主键约束

ALTER TABLE 表名 DROP PRIMARY KEY;

|  |
| --- |
| **ALTER TABLE** student **DROP PRIMARY KEY**; |

添加外键约束

alter table 从表 add constraint 外键（形如：FK\_从表\_主表） foreign key 从表(外键字段) references 主表(主键字段);

删除外键约束

alter table 表名 drop foreign key 外键（区分大小写）;

## 删除查询缓存

|  |
| --- |
| **RESET QUERY CACHE;** |

# 5 类型

建表的过程就是声明列的过程。

## 整型

|  |
| --- |
| **整数类型         字节数       有符号范围（默认）       无符号范围**  TINYINT         1字节         (-128，127)           (0，255)  SMALLINT       2字节     (-32768，32767)       (0，65 535)  MEDIUMINT     3字节     (-，-1) (0，-1)  INT   4字节   (-，-1) (0，-1)  BIGINT         8字节  (-，-1) (0，-1) |

可选参数：整型[(M)] [UNSIGNED] [ZEROFILL]

UNSIGNED：无符号数，例如：age TINYINT UNSIGNED;

ZEROFILL：其中参数M表示字节宽度，自动会转为无符号整型，一般适用于学号、编码等宽度无符号数字，可以用0填充，例如：sn TINYINT(6) ZEROFILL;

## 浮点型

注意：浮点数有精度损失的。

|  |  |  |  |
| --- | --- | --- | --- |
| **浮点类型** | **字节数** | **有符号范围（默认）** | **无符号范围** |
| FLOAT | 4 | -3.402823466E+38到-1.175494351E-38 | 0和1.175494351E-38到3.402823466E+38 |
| DOUBLE | 8 | -1.7976931348623157E+308到-2.2250738585072014E-308 | 0和2.2250738585072014E-308到 1.7976931348623157E+308 |

可选参数：浮点型[(M,D)] [UNSIGNED] [ZEROFILL]

其中M表示显示的值（包括小数位）最大位数，D表示小数位数。

例如：sara FLOAT(7,3) 规定显示的值不会超过7位数字，小数点后面带有 3位数字。

对于小数点后面的位数超过允许范围的值，MySQL 会自动将它四舍五入为最接近它的值。

## 定点型

把整数和小数部分分开存储，比较精确。

可选参数：DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]

例如：sara DECIMAL(6,2);

## 字符串

|  |
| --- |
| **字符串类型     字节数        描述及存储需求**      CHAR         0～255           定长字符串      VARCHAR      0～255          变长字符串      TINYBLOB     0～255         不超过 255 个字符的二进制字符串      TINYTEXT     0～255         短文本字符串      BLOB         0～65535       二进制形式的长文本数据      TEXT         0～65535       长文本数据      MEDIUMBLOB   0～-1 二进制形式的中等长度文本数据      MEDIUMTEXT   0～-1 中等长度文本数据      LOGNGBLOB     0～-1 二进制形式的极大文本数据      LONGTEXT     0～-1 极大文本数据 |

注意：

char型，如果不够M个字符，内部用空格补齐，取出时会把右侧空格删除，意味着本身有的空格会被丢弃。varchar类型则不会。

速度上定长比不定长的更快。

blob是二进制类型，用来储存图像、音频等类型。可以防止字符集问题导致信息丢失。

## 枚举和集合

枚举：ENUM('value1','value2',...)，插入值时，只能在枚举范围中的一个。

集合：SET('value1','value2',...)，插入值时，值是在集合的一个或多个。

## 日期时间类型

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **字节数** | **范围** | **格式** |
| DATE | 4 | 1000-01-01～9999-12-31 | YYYY-MM-DD |
| TIME | 3 | '00:00:00'～'23:59:59' | HH:MM:SS |
| YEAR | 1 | 1901～2155 | YYYY |
| DATETIME | 8 | 1000-01-01 00:00:00～9999-12-31 23:59:59 | YYYY-MM-DD HH:MM:SS |
| TIMESTAMP | 4 | 1970-01-01 00:00:00～2037年某时 | YYYY-MM-DD HH:MM:SS |

注意：

TIMESTAMP是系统自动填充的时间戳，效率不高，不建议使用，通常用int unsigned来储存时间戳。

|  |
| --- |
| 获取时间戳  **SELECT** *UNIX\_TIMESTAMP*() ; *#当前时间戳1451588688* **SELECT** *UNIX\_TIMESTAMP*(**'2016-01-01 00:00:00'**);  把日期时间转为时间戳  **SELECT** *FROM\_UNIXTIME*(1451588888, **'%Y-%m-%d %h:%i:%s'**); *# 2016-01-01 03:08:08* |

# 6 列的属性

## 6.1 列的默认值

实际使用中避免列的默认值为null，创建表时给列一个初始值。

在列的类型后面添加NOT NULL DEFAULT 值。

例如：**id INT UNSIGNED NOT NULL DEFAULT** 0

## 6.2 主键与自增

主键：primary key，能够区分每一行，一般和auto\_increment一起使用。

有两种方式声明该列为主键：

方式一：在列的类型之后声明primary key

例如：**id INT UNSIGNED PRIMARY KEY**

方式二：声明完列之后，在最后声明那一列为主键primary key(列名)

例如：**PRIMARY KEY**(**id**)

主键与自增搭配使用：

例如：**id INT PRIMARY KEY AUTO\_INCREMENT**

提高效率建表原则：定长与变长分离，常用和不常用分离。

# 7 视图

视图又称为虚拟表，是通过sql查询的结果集。

## 视图用途

(1)权限控制：有些不想给用户看的列，比如进货价格，通过视图可以控制该列不被用户看到。

(2)简化复杂的查询，通过视图把查询结果保存下来，视图作为查询条件。

创建视图：CREATE VIEW 视图名称 AS 查询语句

例如：**CREATE VIEW** vuser **AS SELECT username**,**gender FROM user**;

注意：当视图和物理表的列关系是一一对应（类似可逆函数），即视图相当于物理表的子表时，无论修改视图还是修改物理表，都会自动对应修改。如果不是一一对应的不可以修改。

## 视图的algorithm

视图有两种算法：merge(合并)和temptable(临时表)

merge算法：把建视图语句和查询视图语句合并后查询物理表的语句。

temptable算法：先从物理表查询得到临时表，然后再查询临时表得到结果。

创建视图时可以指定算法，不指定算法则让系统自适应选择算法。

CREATE [ALGORITHM = {MERGE | TEMPTABLE}] VIEW AS 查询语句

例如：**CREATE ALGORITHM** = **MERGE VIEW** vuser **AS SELECT username**,**gender FROM user**;

## 视图操作

和表的操作类似，把table改为view则可。

查看视图：**DESC** 视图名

查看视图所有信息：**SHOW CREATE VIEW**视图名;

删除视图：**DROP VIEW** 视图名

查看哪些表为视图：**SHOW TABLE status WHERE comment**=**'view'**;

# 8 表引擎

|  |
| --- |
| **CREATE TABLE** account(  **id INT UNSIGNED PRIMARY KEY**,  **name CHAR**(10) **NOT NULL DEFAULT ''**,  **balance INT NOT NULL DEFAULT** 0 )**ENGINE** innodb **CHARSET** *utf8*; |

引擎不同，组织数据方式也不同，常用的Myisam和InnoDB两种引擎，如下图。



Myisam和InnoDB的6个区别：

|  |  |  |
| --- | --- | --- |
|  | **MyISAM** | **InnoDB** |
| **构成上的区别** | 每个MyISAM在磁盘上存储成三个文件。   .frm文件存储表定义。  .MYD 数据文件。  .MYI索引文件。 | 基于磁盘的资源是InnoDB表空间数据文件和它的日志文件，InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB |
| **事务处理上方面** | MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快，但是不提供事务支持 | InnoDB提供事务支持事务，外部键等高级数据库功能 |
| **SELECT**  **UPDATE,**  **INSERT，Delete操作** | 如果执行大量的SELECT，MyISAM是更好的选择 | **①** 如果你的数据执行大量的**INSERT或UPDATE**，出于性能方面的考虑，应该使用InnoDB表    **② DELETE   FROM table**时，InnoDB不会重新建立表，而是一行一行的删除。    **③LOAD  TABLE FROM MASTER**操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性（例如外键）的表不适用。 |
| **对AUTO\_INCREMENT的操作** | 每表一个AUTO\_INCREMEN列的内部处理。    **MyISAM为INSERT和UPDATE操作自动更新这一列**。这使得AUTO\_INCREMENT列更快（至少10%）。在序列顶的值被删除之后就不能再利用。(当AUTO\_INCREMENT列被定义为多列索引的最后一列，可以出现重使用从序列顶部删除的值的情况）。    AUTO\_INCREMENT值可用ALTER TABLE或myisamch来重置    对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引    更好和更快的auto\_increment处理 | 如果你为一个表指定AUTO\_INCREMENT列，在数据词典里的InnoDB表句柄包含一个名为自动增长计数器的计数器，它被用在为该列赋新值。    自动增长计数器仅被存储在主内存中，而不是存在磁盘上    关于该计算器的算法实现，请参考    **AUTO\_INCREMENT列在InnoDB里如何工作** |
| **表的具体行数** | select count(\*) from table,MyISAM只要简单的读出保存好的行数，注意的是，当count(\*)语句包含   where条件时，两种表的操作是一样的 | InnoDB 中不保存表的具体行数，也就是说，执行select count(\*) from table时，InnoDB要扫描一遍整个表来计算有多少行 |
| **锁** | 表锁 | 提供行锁(locking on row level)，提供与 Oracle 类型一致的不加锁读取(non-locking read in    SELECTs)，另外，InnoDB表的行锁也不是绝对的，如果在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表，例如update table set num=1 where name like “%aaa%” |

# 9 字符集和校对集

乱码原因：文字本来的字符集和展示的字符集不一致造成的。

解决乱码方法：

查看数据库的字符集是否一致：**SHOW VARIABLES LIKE '%charact%'**

把character\_set\_client，character\_set\_connection，character\_set\_results三个设置为一致的字符集即可，

简写为set names utf8;

校对集：就是排序规则。

# 10 索引

索引是数据的目录，能快速定位需要取数据的位置。

索引提高列查询的速度，降低了增、删、改的速度。

一般在查询频繁的列或值重复度低的列添加索引。

## 单列索引

key：普通索引

例如：key cat\_id(cat\_id)

unique key：唯一索引

例如：unique key email(email)

primary key：主键索引，一张表只能存在一个

例如：primary key (id)

fulltext：全文索引，在中文环境下几乎无效。一般用第三方解决方案sphinx

索引长度：建索引时，可以取列的部分字符作为索引，节省空间。

例如：unique key email(email(10)); 取email列的前10个字符作为索引。

## 多列索引

把两列或多列的值作为整体后再建索引，左前缀规则。

例如：key xm(xing, ming)

|  |
| --- |
| **CREATE TABLE name** (  **xing CHAR**(2),  **ming CHAR**(10),  **KEY** xm(**xing**,**ming**) );  在select前面添加关explain键字，得到结果的possible\_key可以查看是否使用到索引查询。  **EXPLAIN SELECT** *\** **FROM name WHERE xing**=**'刘' AND ming**=**'备'**;*# 使用索引xm查询* **EXPLAIN SELECT** *\** **FROM name WHERE xing**=**'刘'**;*# 使用索引xm查询* **EXPLAIN SELECT** *\** **FROM name WHERE ming**=**'备'**;*# 没有使用索引xm查询* |

## 冗余索引

在某个列上可能存在多个索引

比如某个表：key xm(xing, ming)，key ming(ming)，对于列ming来说，索引xm 和 ming两个索引覆盖，叫做冗余索引。

## 索引操作

查看索引：**SHOW INDEX FROM** 表名;

删除索引：**ALTER TABLE** 表名 **DROP INDEX** 索引名;

添加索引

添加普通索引：**ALTER TABLE** 表名 **ADD INDEX** 索引名(列名...);

添加唯一索引：**ALTER TABLE** 表名 **ADD UNIQUE** 索引名(列名);

添加主键索引：**ALTER TABLE** 表名 **ADD PRIMARY KEY**(列名);

# 11 事务

事务的4个特性:ACID

原子性(Atomicity）：是指某几句sql的影响，要么都发生，要么都不发生。

一致性(Consistency)：事务前后的数据，保持业务上的合理一致。

隔离性(Isolation)：在事务进行过程中，其他事务看不到此事务的任何效果。

持久性(Durability)：事务一旦发生，不能取消，只能通过补偿性事务来抵消效果。

事务与引擎：myisam引擎不支持事务，innodb和BDB引擎支持。

一个完整的事务过程：

启动事务：**START TRANSACTION**;

sql执行：增删改查

结束事务：**COMMIT(提交)或ROLLBACK(取消)**;

注意commit一旦发生之后，rollback无法回滚。

实例：

|  |
| --- |
| **START TRANSACTION**;*# 启动事务* **UPDATE** account **SET balance**=**balance**+1000 **WHERE name**=**'曹操'**; **UPDATE** account **SET balance**=**balance**-1000 **WHERE name**=**'刘备'**; **COMMIT**;*# 提交事务* |

可以设置事务的级别：

set session transaction isolation level [read uncommitted | read committed | repeatable read | serializable]

read uncommitted：可以读未提交的事务内容，称为”脏读”，破坏了事务的隔离性，一般不用。

read commited：在一个事务进行过程中，读不到另一个进行事务的操作，但是可以读到另一个结束事务的操作影响，一般不用。

repeatable read：在一个事务过程中，所有信息都来自事务开始那一瞬间的信息，不受其他已提交事务的影响，大多数的系统，用此隔离级别，建议使用。

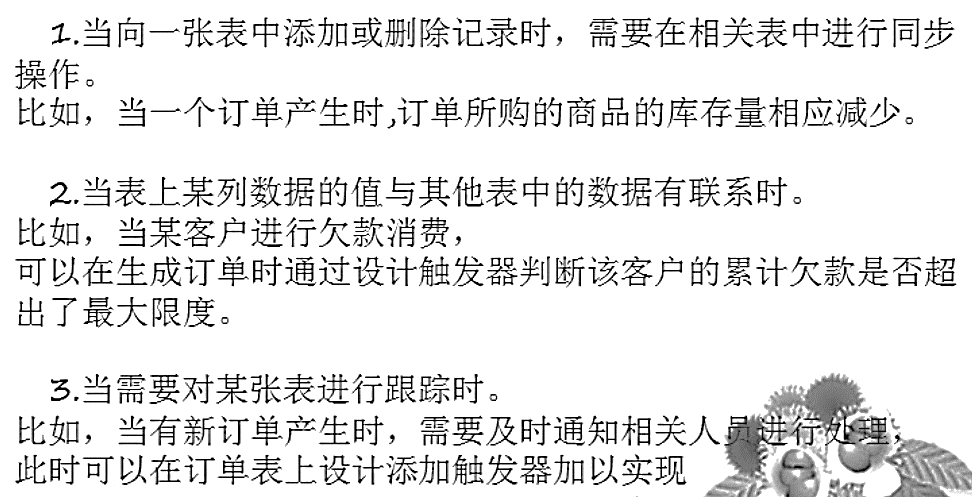
serializeable：串行化，把所有事务进行编号，按顺序一个一个来执行，也就取消了事务冲突的可能，隔离级别最高，但事务相互等待的时间会比较长，在实际中使用也不是很多。

# 12 触发器

触发器是一类特殊的事务，可以监视某种数据操作(insert|update|delete)，并触发相关操作(insert|update|delete)。

## 使用场合

有时碰到表中某些数据改变，希望同时引起改变其他数据改变的需求，利用触发器可以满足这样的需求。例如商城中的有客户下订单后，库存量、购买人数等这些数据需要跟着改变。



作用：在表中某些特定数据变化时自动完成某些查询，运用触发器不仅可以简化程序，并且可以增加程序的灵活性。

## 触发器操作

查看触发器：**SHOW TRIGGERS**;

删除触发器：**DROP TRIGGER** 触发器名;

## 创建触发器语法的四要素

监视地点(table表)

监视事件(insert | update | delete)

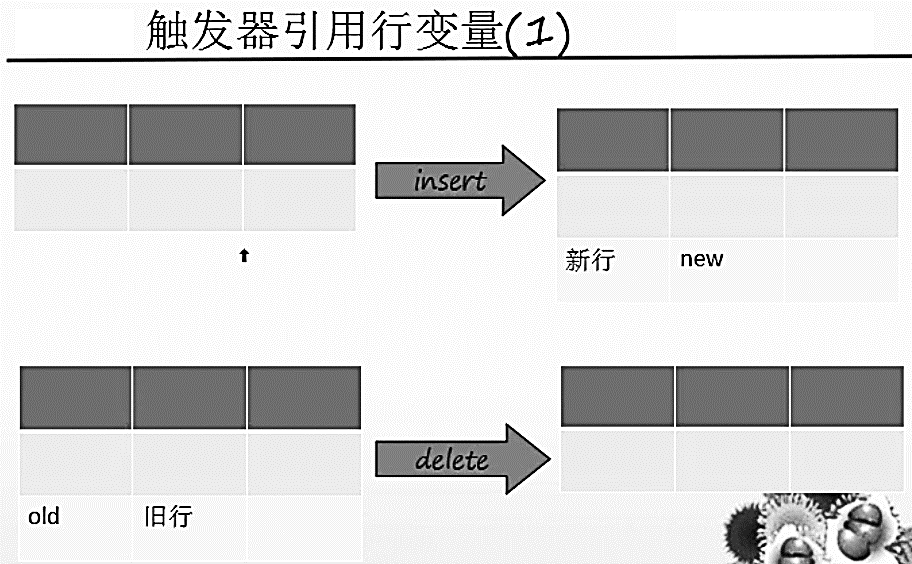
触发时间(before | after)

触发事件(insert | update | delete)

## 创建触发器语法

|  |
| --- |
| **CREATE TRIGGER** 触发器名  **BEFORE 或 AFTER** *# 触发时间*  **INSERT 或UPDATE 或 DELETE** *# 监视事件*  **ON 表名** *# 监视地点*  **FOR EACH ROW** *#在mysql中必须写，行级触发器，在oracle可以不写，表示语句级触发器*  **BEGIN** *# 开始触发* sql语句1 sql语句2 ……  **END** *# 结束触发* |

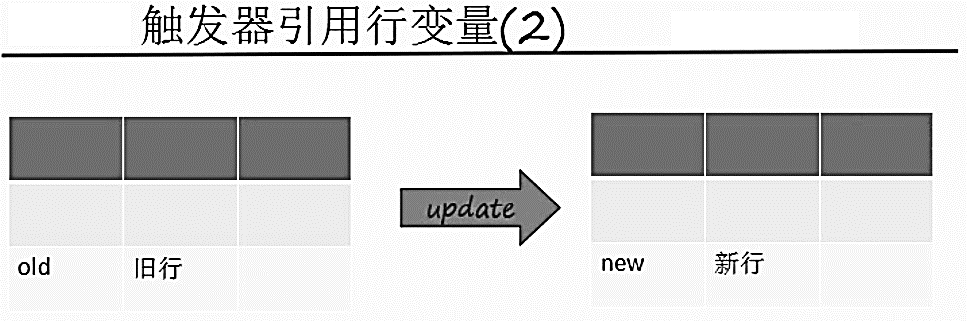
## 触发器变量



**监控insert行为时，引用变量用new**

**监控delete行为时，引用变量用old**

|  |
| --- |
| (1)订单表插入数据时，触发商品表对应的数据修改的触发器。  **CREATE TRIGGER** tr1 **AFTER INSERT ON** orders **FOR EACH ROW BEGIN  UPDATE** goods **SET num**=**num**-*new*.**much WHERE gid**=*new*.**gid**; **END**;  改进版，当购买数量大于库存数量是，默认为库存数量,防止爆仓。  **CREATE TRIGGER** tr4 **BEFORE INSERT ON** orders **FOR EACH ROW BEGIN  DECLARE** rnum **SMALLINT UNSIGNED DEFAULT** 0;  **SELECT num INTO** rnum **FROM** goods **WHERE gid**=*new*.**gid**;  **IF** *new*.**much**>rnum **THEN  SET** *new*.**much** = rnum;  **END IF**;   **UPDATE** goods **SET num**=**num**-*new*.**much WHERE gid**=*new*.**gid**; **END**;  (2)订单表删除数据时，触发商品表对应的数据修改的触发器（实际中订单只能失效，不能删除）。  **CREATE TRIGGER** tr2 **AFTER DELETE ON** orders **FOR EACH ROW BEGIN  UPDATE** goods **SET num**=**num**+*old*.**much WHERE gid**=*old*.**gid**; **END**; |



**监控update行为时，引用变量update前用old，update后用new。**

|  |
| --- |
| (3)修改订单表数据时，触发商品表对应的数据修改的触发器。  **CREATE TRIGGER** tr3 **BEFORE UPDATE ON** orders **FOR EACH ROW BEGIN  UPDATE** goods **SET num**=**num**+*old*.**much**-*new*.**much WHERE gid**=*new*.**gid**; **END**; |

# 13 存储过程

把若干条sql语句封装起来并起个名字，在过程中把数据存储到数据库中。

## 创建存储过程语法

定义语法：

|  |
| --- |
| **CREATE PROCEDURE** *名称*()  **BEGIN** *# sql语句* **END**; |

调用存储过程：**CALL** *存储过程名字*();

查看存储过程：**SHOW PROCEDURE STATUS**;

删除存储过程：**DROP PROCEDURE***存储过程名字*;

存储过程是可以编程的，意味着可以使用变量、表达式、控制结构来完成复杂的功能。

## 声明变量

**DECLARE** *变量名 变量类型 [default 默认值]*

注意：声明变量必须在begin和end之间声明。

变量可以参与sql语句的运算。

**SET** 变量名 := 表达式

|  |
| --- |
| **CREATE PROCEDURE** *test1*()  **BEGIN  DECLARE** leng **INT DEFAULT** 0;  **DECLARE** widch **INT DEFAULT** 0;  **SET** leng := 5;  **SET** widch := 6;  **SELECT** leng\*widch;  **END**; |

## 参数

参数分为in、 out、 inout类型。in表示输入类型，out表示输出类型，inout表示输入输出类型。

|  |
| --- |
| (1)in和out类型  **CREATE PROCEDURE** *cuArea*(**in** r **INT**, **OUT** area **INT**)  **BEGIN  SET** area:=0;*# 如果输出area参与运算时必须设置area的初始值，因为null参与运算的值都为null* **SET** area := 3.14\*r\*r;  **END**;  调用： **CALL** *cuArea*(10,*@area*); **SELECT** @area;*# 结果：314*  (2)inout类型  **CREATE PROCEDURE** *add\_1*(**INOUT** v **INT**)  **BEGIN  SET** v := v + 1;  **END**;  *# 定义变量和调用* **SET** *@v* := 1; **CALL** *add\_1*(@v); **SELECT** @v; *# 结果：2* |

## if条件控制结构

**IF** 条件1 **THEN**  
**ELSEIF**条件2 **THEN**

**......**  
**ELSE**  
**END IF**;

其中ELSEIF和ELSE可以没有。

|  |
| --- |
| **CREATE PROCEDURE** *compare*(v1 **INT**,v2 **INT**)  **BEGIN  IF** v1>v2 **THEN  SELECT** *concat*(v1,**'大于'**,v2);  **ELSEIF** v1<v2 **THEN  SELECT** *concat*(v1,**'小于'**,v2);  **ELSE  SELECT** *concat*(v1,**'等于'**,v2);  **END IF**;  **END**; |

## case选择控制结构

**CASE** 变量  
 **WHEN** 值 **THEN 表达式**;  
 ......  
 **ELSE 不满足条件最后的默认结果**;  
 **END CASE**;

注意：else 可以省略。

|  |
| --- |
| **CREATE PROCEDURE** *cs*()  **BEGIN  DECLARE** v **INT**;  **SET** v := *floor*(*rand*()\*10);  **CASE** v  **WHEN** 0 **THEN SELECT '星期日'**;  **WHEN** 1 **THEN SELECT '星期一'**;  **WHEN** 2 **THEN SELECT '星期二'**;  **WHEN** 3 **THEN SELECT '星期三'**;  **WHEN** 4 **THEN SELECT '星期四'**;  **WHEN** 5 **THEN SELECT '星期五'**;  **WHEN** 6 **THEN SELECT '星期六'**;  **ELSE SELECT 'unknown day'**;  **END CASE**;  **END**; |

## while循环结构

**WHILE** 条件 **DO**  执行语句  
**END WHILE**;  
注意：避免死循环

|  |
| --- |
| **CREATE PROCEDURE** *cusum* (v **INT**)  **BEGIN  DECLARE** s **INT DEFAULT** 0;  **WHILE** v>0 **DO  SET** s := s + v;  **SET** v := v-1;  **END WHILE**;  **SELECT** s;  **END**;  调用： **CALL** *cusum*(100); *# 结果：5050* |

## repeat循环结构

**REPEAT**  执行语句......  
 **UNTIL** 条件 **END REPEAT**;

|  |
| --- |
| **CREATE PROCEDURE** *cuSum2*(v **INT**)  **BEGIN  DECLARE** sum **INT DEFAULT** 0;  **REPEAT  SET** sum := sum+v;  **SET** v:=v-1;  **UNTIL** v<=0 **END REPEAT**;  **SELECT** sum;  **END**;  *# 调用* **CALL** *cuSum2*(100); *# 结果：5050* |

# 14 游标

一条sql的select语句取出对应的n条资源，取出资源的接口(句柄)就是游标，沿着游标，每次只取出一行，取出的行可以任意的逻辑控制了，而select没有这种功能。

游标语法:

|  |
| --- |
| **DECLARE** 游标名 **CURSOR FOR select语句**; *# (1)声明游标*  **DECLARE EXIT HANDLER FOR NOT FOUND 表达式** ;*# 设置触发边界标志*  **OPEN**游标名; *# (2)打开游标* **FETCH**游标名**INTO** 变量1, 变量2......; *# (3)取值* **CLOSE**游标名; *# (4)关闭游标* |

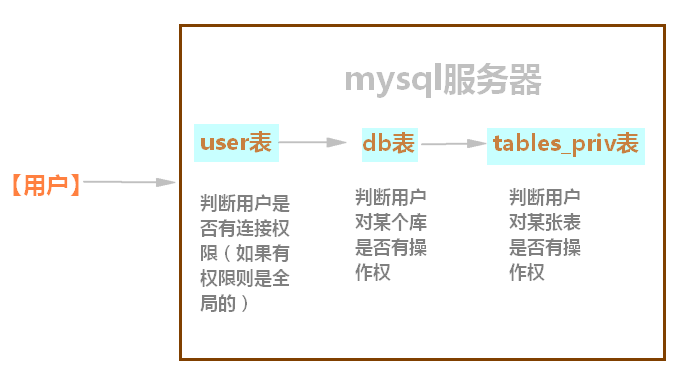
用循环读取游标数据，结束条件是判断是否去到最后一条数据（事先计算出来的总数）。

|  |
| --- |
| **CREATE PROCEDURE** *cursor1*()  **BEGIN  DECLARE** tmp\_name **VARCHAR**(20);  **DECLARE** tme\_num **INT**;  **DECLARE** cnt **INT**;  **DECLARE** i **INT DEFAULT** 0;   **DECLARE** get\_goods **CURSOR FOR SELECT name**,**num FROM** goods;  **OPEN** get\_goods;  **SELECT** *count*(*\**) **INTO** cnt **FROM** goods;  **WHILE** i<cnt **DO  SET** i:=i+1;  **FETCH** get\_goods **INTO** tmp\_name,tme\_num;*# 取一条数据* **SELECT** tmp\_name,tme\_num;  **END WHILE**;   **CLOSE** get\_goods;  **END**;  *# 调用* **CALL** *cursor1*(); |

在mysql的cursor中可以用declare exit或continue handler for not fond来操作越界标志。类似于js中的事件，当读取游标完毕则触发该事件。其中exit和continue的区别是是否执行后面的sql语句。

|  |
| --- |
| (1)触发越界后执行exit，不执行后面的sql语句  **CREATE PROCEDURE** *cursor2*()  **BEGIN  DECLARE** tmp\_name **VARCHAR**(20);  **DECLARE** tme\_num **INT**;  **DECLARE** isEnd **BOOL DEFAULT FALSE**;   **DECLARE** get\_goods **CURSOR FOR SELECT name**,**num FROM** goods;  **DECLARE EXIT HANDLER FOR NOT FOUND SET** isEnd:=**TRUE** ;*# 设置触发边界标志* **OPEN** get\_goods;   **WHILE** !isEnd **DO  FETCH** get\_goods **INTO** tmp\_name,tme\_num;*# 取一条数据* **SELECT** tmp\_name,tme\_num;  **END WHILE**;   **CLOSE** get\_goods;  **END**;  *# 调用* **CALL** *cursor2*();  (2)触发越界后执行continue，继续后面的sql语句。  **CREATE PROCEDURE** *cursor3*()  **BEGIN  DECLARE** tmp\_name **VARCHAR**(20);  **DECLARE** tme\_num **INT**;  **DECLARE** isEnd **BOOL DEFAULT FALSE**;   **DECLARE** get\_goods **CURSOR FOR SELECT name**,**num FROM** goods;  **DECLARE CONTINUE HANDLER FOR NOT FOUND SET** isEnd:=**TRUE** ;*# 设置触发边界标志* **OPEN** get\_goods;   **FETCH** get\_goods **INTO** tmp\_name,tme\_num;*# 进入循环前先取一条数据* **WHILE** !isEnd **DO  SELECT** tmp\_name,tme\_num;  **FETCH** get\_goods **INTO** tmp\_name,tme\_num;*# 取一条数据* **END WHILE**;   **CLOSE** get\_goods;  **END**;  *# 调用* **CALL** *cursor3*(); |

# 15 mysql用户与权限管理



## 15.1用户连接mysql服务器

【用户】<----------->【mysql服务器】认证用户依据有3个参数：

你从哪里来？host

你是谁？user

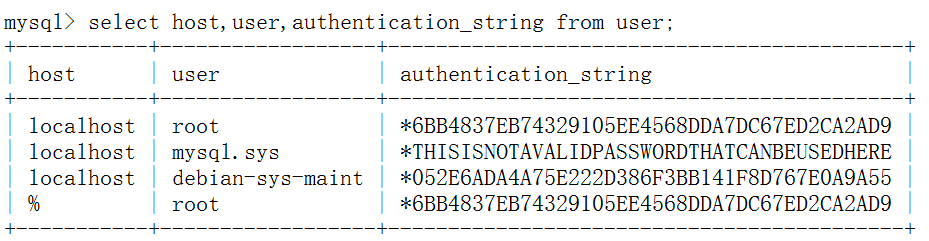
你的密码是多少？password

连接数据库：mysql -h192.168.8.102 -uroot -p123456;

查看当前登录用户：**SELECT** *user*();

可以通过库名mysql里的user表来查看有哪些用户可以登录。

|  |
| --- |
| **USE** mysql; **DESC user**;*# 查看表结构*  **SELECT host**,**user**,**password FROM user**; |



### 修改host域，使用IP能连接起来

|  |
| --- |
| **UPDATE user SET host**=**'192.168.8.101' WHERE host**=**'::1'**; **FLUSH PRIVILEGES**; *# 冲刷权限* |

### 修改用户密码

|  |
| --- |
| **UPDATE user SET password**=*password*(**'123456'**) **WHERE host**=**'192.168.8.101'**; **FLUSH PRIVILEGES**; *# 冲刷权限* |

## 15.2授权给用户

用户有没有权限执行操作(例如：insert、update、delete等)

### 新建一个用户并授权

**GRANT [***权限1, 权限2......***] ON** *数据库名.该库下的表名* **TO** *用户名@主机名* **IDENTIFIED BY** *密码*;

例如：授权给主机为192.168.8.n局域网内的用户root所有数据库权限

**GRANT ALL ON** *\**.*\** **TO** *root*@*'192.168.8.%'* **IDENTIFIED BY '123456'**;

常用权限

all，creat，drop，insert，update，delete，select......

针对某个库做授权

**GRANT ALL ON** *test*.*\** **TO** *root*@*'192.168.8.%'*;

**FLUSH PRIVILEGES**; *# 冲刷权限*

针对某个表授权

**GRANT select,update,insert ON** *test*.*goods* **TO** *root*@*'192.168.8.%'*;

**FLUSH PRIVILEGES**; *# 冲刷权限*

### 收回用户权限

**REVOKE [***权限1, 权限2......***] ON** *数据库名.该库下的表名* **FROM** *用户名@主机名*;

例如：收回某用户的所有权限

**REVOKE ALL ON** *\**.*\** **FROM** *root*@*'192.168.8.%'*;

## 15.3 手工跳过mysql认证登录方法

有时候忘记mysql帐号密码或修改了主机ip使得无法登录mysql，

解决办法：

(1) 关闭mysql服务

**/etc/init.d/mysql start**

(2) 启动mysql服务时添加跳过权限检测

**/usr/local/mysql/bin/mysqld --skip-grant-tables**

(3) 在新的终端登录mysql修改数据库mysql下user表的host、user、name列

登录mysql：/usr/local/mysql/bin/mysql

>use mysql;

>select host,user,password from user;

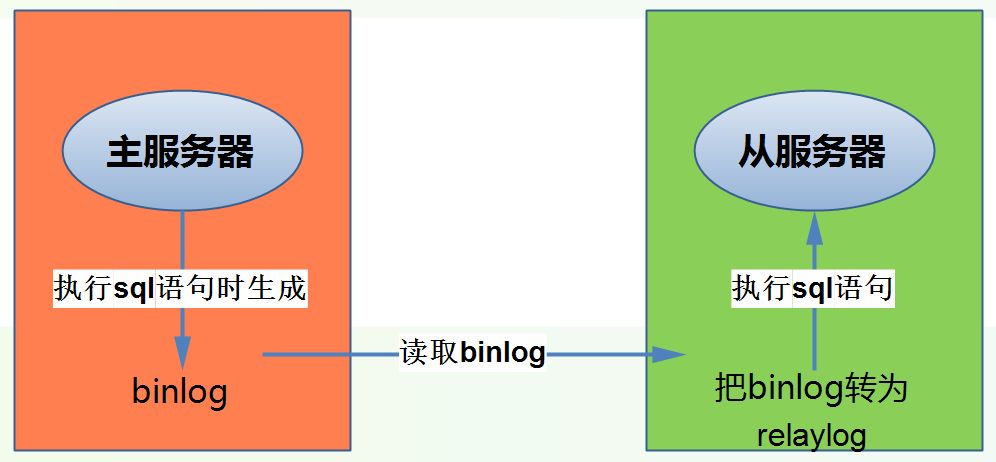
>update user set host='ip地址',user='用户名',password=password('你的密码') where 定位那一行;

最后重启mysql服务即可。

# 16 mysql集群

## 16.1 mysql主从复制

原理



配置：

◎主mysql配置binlog

◎从mysql器配置relaylog

◎主mysql授权给从mysql器一个帐号

◎从mysql连接主mysql

### 16.1.1修改主mysql配置文件

(1) 编辑配置文件：sudo vim /etc/mysql/my.cnf

在[mysqld]字段下

① 给主mysql起一个名字，在局域网内一般为ip最后一个数字

**server-id = 101**

② 开启logbin二进制日志，并给logbin起个名字

**log-bin = mysql\_bin**

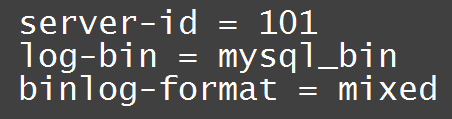
③ 指定日志格式，分别有mixed（由系统决定row还是statement）、row（记录磁盘变化）、statement（记录执行语句），各有各的应用场景。

**binlog-format = mixed**

④ 重启mysql服务器

**sudo service mysql restart**

添加配置内容如下图所示：



(2) 给从mysql器添加一个授权帐号

**GRANT REPLICATION CLIENT**,**REPLICATION SLAVE ON** *\**.*\** **TO** *'repl'*@*'192.168.8.%'* **IDENTIFIED BY '123456'**;

**FLUSH PRIVILEGES**; *# 冲刷权限*

(3) 查看mysql是否具备充当主mysql条件

**SHOW MASTER STATUS**;

### 16.1.2 配置从mysql器relaylog

(1) 从mysql器也开启binlog，防止数据丢了可以恢复。

编辑配置文件：**vim /usr/local/mysql/my.cnf**

在[mysqld]字段下

① 给主mysql起一个名字，防止多个主从混乱

**server-id = 102**

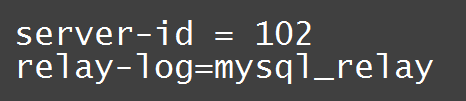
② 开启relaylog日志，并给relaylog起个名字

**relay-log=mysql\_relay**

③ 重启mysql服务器：

**sudo service mysql restart**

添加配置内容如下图所示：



(2) 在从mysql器通过语句指定要复制的主mysql (可以一主多从，不能多主)

|  |
| --- |
| **CHANGE MASTER TO  MASTER\_HOST**=**'192.168.8.101'**,  **MASTER\_USER** = **'repl'**,  **MASTER\_PASSWORD** = **'123456'**,  **MASTER\_LOG\_FILE** = **'mysql\_bin.000003'**, *# 用语句SHOW MASTER STATUS;查看*  **MASTER\_LOG\_POS** = 120; *# 用语句SHOW MASTER STATUS;查看* |

(3) 配置连接信息后启动从mysql器功能

**START SLAVE;**

(4) 查看从mysql是否连接到主mysql

**SHOW SLAVE STATUS**;

如果有连接错误，

ping下能否有网络连接，

telnet 192.168.8.101:3306判断是否连接上，

如果连接不上，试试在终端停止防火墙service iptables stop，

如果出错使用命令**STOP SLAVE**停止slave

注意：主从复制的时间间隔一般为毫秒级别，达到秒级别的使用时风险比较大。

常用语句：

**SHOW MASTER STATUS**;*# 查看主mysql状态*

**START SLAVE;** *# 配置好连接主mysql信息后开始监听master变化*

**SHOW SLAVE STATUS**; *# 显示从mysql状态*

**STOP SLAVE** *# 停止slave*

## 16.2 主主复制

思路：

◎两台mysql都设置二进制和relay日志

◎两台mysql都设置replication帐号

◎两台mysql都设置对方为自己的master

修改两台主mysql配置文件

(1) 编辑配置文件：**sudo vim /etc/mysql/my.cnf**

在[mysqld]字段下

① 给主mysql起一个名字，在局域网内一般为ip最后一个数字，注意两台主mysql名字不能一样。

**server-id = 101**

② 开启logbin二进制日志，并给logbin起个名字

**log-bin = mysql\_bin**

③ 指定日志格式，分别有mixed（由系统决定row还是statement）、row（记录磁盘变化）、statement（记录执行语句），各有各的应用场景。

**binlog-format = mixed**

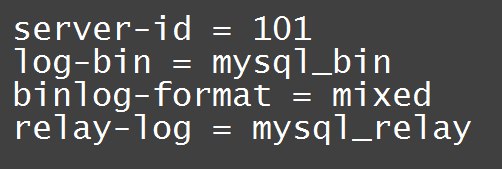
④ 开启relaylog日志，并给relaylog起个名字

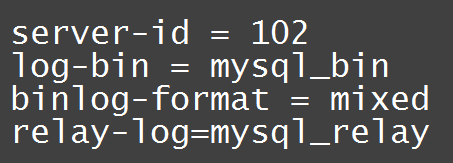
**relay-log=mysql\_relay**

⑤ 重启mysql服务器

**sudo service mysql restart**

两台主mysql添加配置内容如下图所示，可以看出只是server-id不同。





(2) 给从mysql添加一个授权帐号，两个主mysql授权帐号可以相同。

**GRANT REPLICATION CLIENT**,**REPLICATION SLAVE ON** *\**.*\** **TO 'repl'**@*'192.168.8.%'* **IDENTIFIED BY '123456'**;

**FLUSH PRIVILEGES**; *# 冲刷权限*

(3) 查看两个mysql是否具备充当主mysql条件

**SHOW MASTER STATUS**;

(4) 互相连接对方的mysql

|  |
| --- |
| **CHANGE MASTER TO  MASTER\_HOST**=**'192.168.8.101'**,  **MASTER\_USER** = **'repl'**,  **MASTER\_PASSWORD** = **'123456'**,  **MASTER\_LOG\_FILE** = **'mysql\_bin.000002'**, *# 从SHOW MASTER STATUS查看对方的状态得到*  **MASTER\_LOG\_POS** = 560; *# 从SHOW MASTER STATUS查看对方的状态得到* |
| **CHANGE MASTER TO  MASTER\_HOST**=**'192.168.8.102'**,  **MASTER\_USER** = **'repl'**,  **MASTER\_PASSWORD** = **'123456'**,  **MASTER\_LOG\_FILE** = **'mysql\_bin.000005'**, *# 从SHOW MASTER STATUS查看对方的状态得到*  **MASTER\_LOG\_POS** = 1212; *# 从SHOW MASTER STATUS查看对方的状态得到* |

(5) 配置连接信息后都启动从mysql器功能

**START SLAVE;**

(6) 查看能否互相连接到mysql

**SHOW SLAVE STATUS**;

如果出错使用命令**STOP SLAVE**停止slave

这样设置后会有个主键冲突问题，即(表里有主键，并且时自动增长的，当分别同时插入两台主mysql时，同步的时候就出现主键冲突问题)

解决办法：

(1) 在mysql表设置自动增长步长为2，一个mysql从1开始自动增长1 3 5......，另一个从2开始自动增长2 4 6......

一台mysql：

set global auto\_increment\_increment = 2;

set global auto\_increment\_offset = 1;

set session auto\_increment\_increment = 2;

set session auto\_increment\_offset = 1;

另一台mysql：

set global auto\_increment\_increment = 2;

set global auto\_increment\_offset = 2;

set session auto\_increment\_increment=2;

set session auto\_increment\_offset = 2;

注：auto-increment-increment和auto-increment-offset要写到配置文件中，防止下次重启后失效．

这种方式也有缺陷，当新增一台主机作为主mysql是，又要从新修改自动增长步长。

(2) 从业务上统一主键id

比如使用redis的incr命令(自动加1操作)，每插入一行数据前先执行redis的incr命令获取主机id值，然后在插入数据到mysql。

## 16.3 被动模式下主主复制

被动模式下主主复制和普通主主复制区别就是设置一台主服务器为只读状态，也就是说把只读的主mysql作为另一个可读写主mysql的备份，当可读写的主mysql主机坏了，只需设置只读的主mysql配置为可读写就可以了，实现mysql无缝切换。

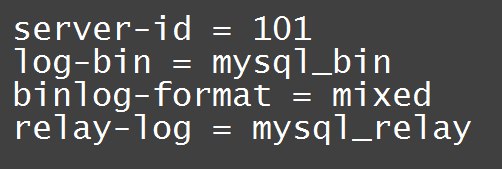
在主主配置的基础上，只需配置一台主mysql作为只读mysql即可。

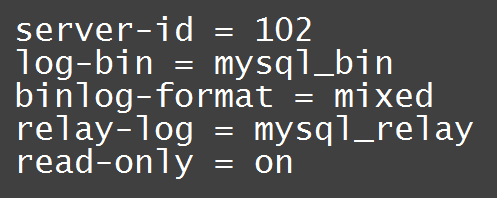
(1) 编辑配置文件：**sudo vim /etc/mysql/my.cnf**

在[mysqld]字段下添加字段

**read-only=on**

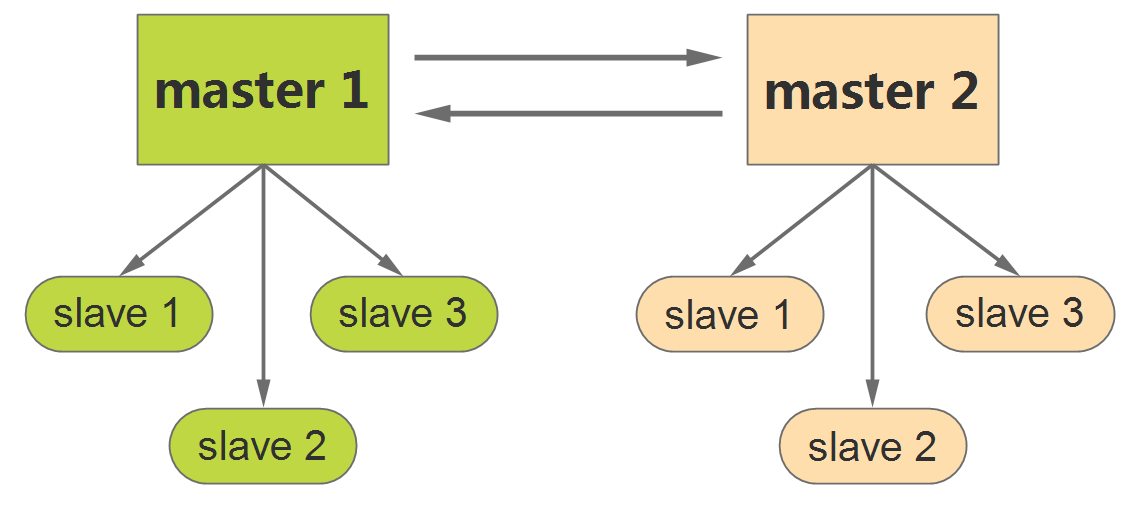
被动模式下两台主mysql添加配置内容如下图所示：





(2) 重启mysql：**sudo service mysql restart**

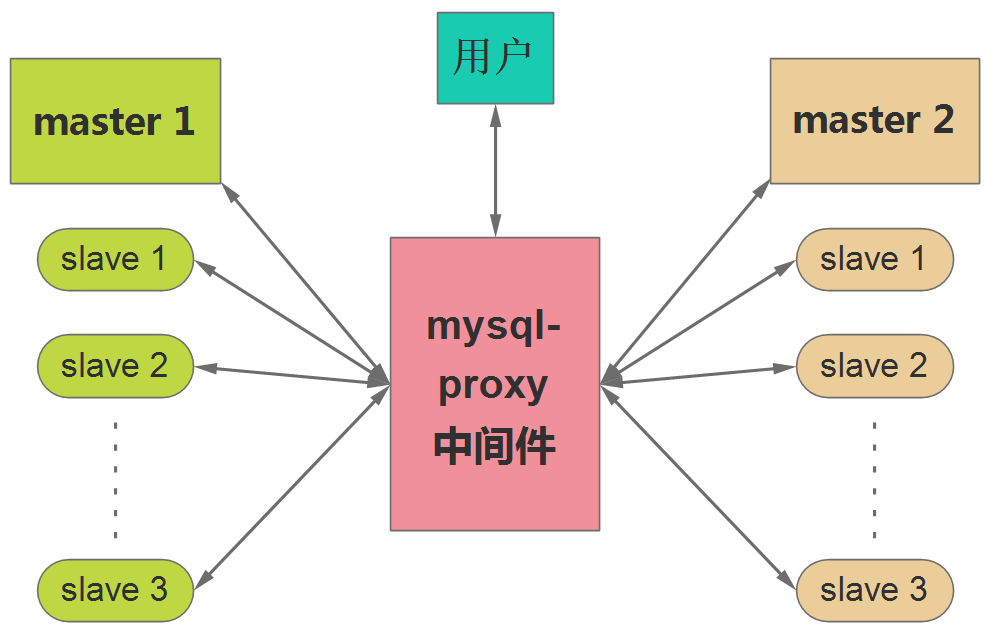
实际场景中通常是两个主mysql下又有从mysql如下图所示，也可以作为读写分离。



## 16.4 mysql集群负载均衡

原理：

通过mysql-proxy中间件来管理mysql集群，用户不需要知道集群中有多少个mysql服务器和地址，只需要连接mysql-proxy中间件，用户增删改查操作都是通过mysql-proxy中间件完成，mysql-proxy既可以mysql进行读写分离，也可以负载均衡，如下图所示：



安装mysql-proxy中间件

|  |  |
| --- | --- |
| 下载源码 | wget http://ftp.ntu.edu.tw/pub/MySQL/Downloads/MySQL-Proxy/mysql-proxy-0.8.4-linux-glibc2.3-x86-64bit.tar.gz |
| 解压 | tar zxvf mysql-proxy-0.8.4-linux-glibc2.3-x86-64bit.tar.gz |
| 进入目录 | cd mysql-proxy-0.8.4-linux-glibc2.3-x86-64bit |
| 把解压文件夹移动到常用的安装软件目录下并修改名字，安装完成 | mv mysql-proxy-0.8.4-linux-glibc2.3-x86-64bit /usr/local/mysql-proxy |

**(1) mysql-proxy说明**

mysql-proxy启动参数思路：

代理了哪个端口?

代理了哪写mysql服务?

对mysql是否进行读写分离?

mysql-proxy是自动负载均衡的，这里的均衡并不是sql语句上的均衡，而是mysql-proxy和用户连接上的均衡，例如当前有20台mysql服务器，有1000个用户连接过来，此时mysql-proxy会把1000个连接数平均分给20台mysql服务器，一旦用户连接上了一台mysql服务器，在用户没有断开连接之前，用户的增删改查或事务操作都是该mysql服务器操作的。

**(2) 启动mysql-proxy**

查看帮助: **/usr/local/mysql-proxy/bin/mysql-proxy --help-all**

在普通启动：

**/usr/local/mysql-proxy/bin/mysql-proxy -P 192.168.8.102:4040 -b 192.168.8.101:3306 -b 192.168.8.102:3306**

-P参数指定mysql-proxy运行的ip地址和端口

-b参数指定mysql服务器的ip地址和端口

--daemon表示在后台启动

**(3) 设置读写分离mysql启动方式**

mysql-proxy通过一个脚本文件判断用户输入的sql是写入还是读取，脚本位置在/usr/local/mysql-proxy/share/doc/mysql-proxy/rw-splitting.lua

读写分离启动mysql-proxy

**/usr/local/mysql-proxy/bin/mysql-proxy -P 192.168.8.102:4040 -b 192.168.8.101:3306 -r 192.168.8.102:3306 -s /usr/local/mysql-proxy/share/doc/mysql-proxy/rw-splitting.lua**

-P参数指定mysql-proxy运行的ip地址和端口

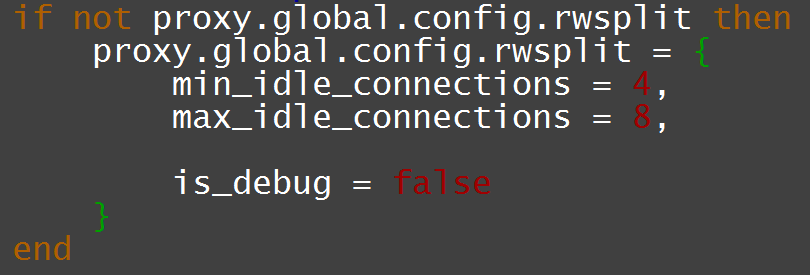
-b参数指定mysql服务器的ip地址和端口

-r参数指定只读mysql服务器的ip地址和端口

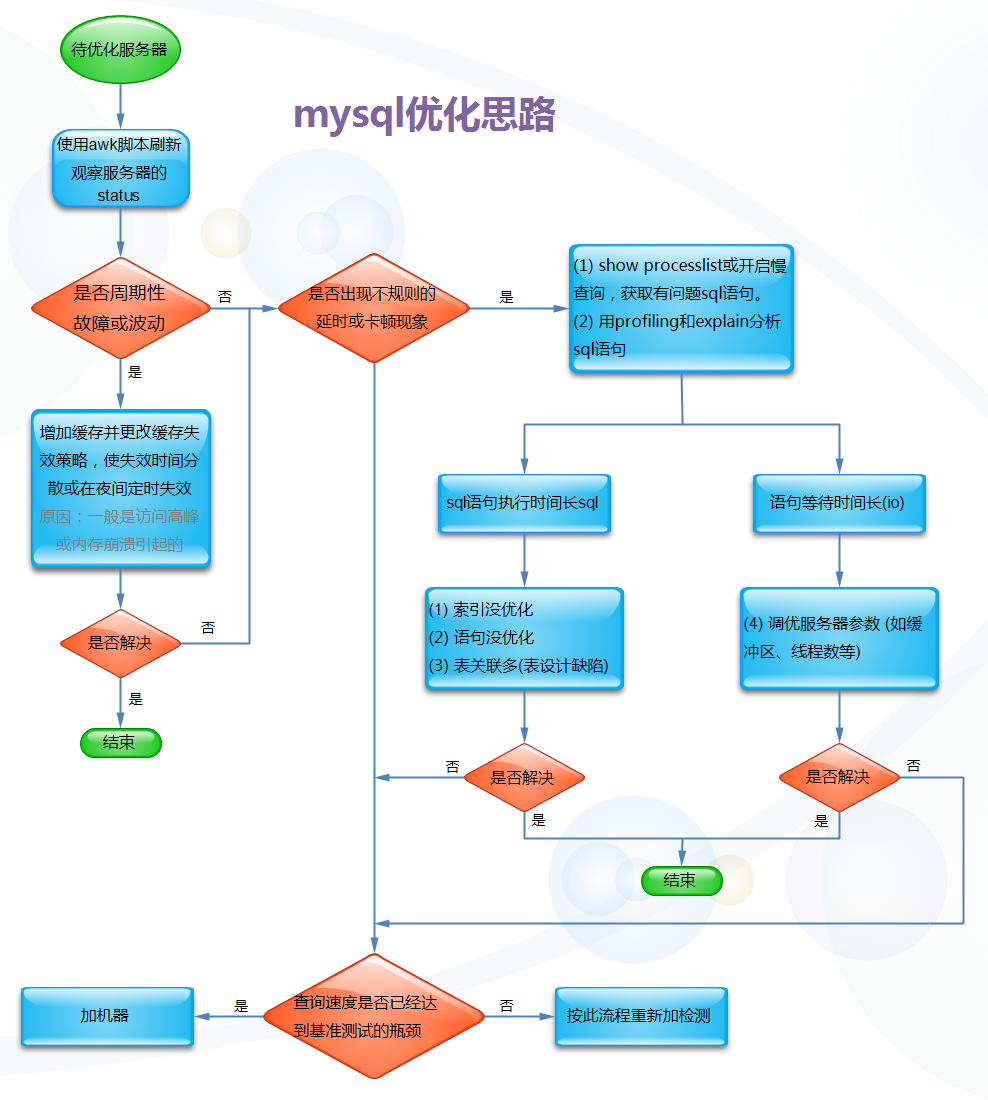
-s参数指定脚本文件的路径

--daemon表示在后台启动

注意：因为-s指定了脚本，脚本有默认最小和最大连接空闲连接数，当连接数超过默认值时，mysql-proxy才会做负载均衡。



# 17 mysql优化分析利器



## 17.1 记录mysql查询数量和连接情况

在mysql命令行查看服务器状态：**SHOW STATUS**;

在linux命令行查看服务器状态：**mysqladmin -uroot -p123456 ext**

(如果查看不了，请检查mysql的用户名和密码是否正确)

得到很多行服务器状态数据，主要看3个参数：

Queries: 查询次数

Threads\_connected: 线程连接数

Threads\_running: 线程运行数

在linux命令行获取上面三个参数值：mysqladmin -uroot -p123456 ext | awk **'/Queries/{q=$4} /Threads\_connected/{c=$4} /Threads\_running/{r=$4} END{printf "%d %d %d\n", q, c, r}'**

一般使用脚本循环获取mysql状态，脚本内容如下：

|  |
| --- |
| **#!/bin/bash  while** true **do** mysqladmin -uroot ext | awk **'/Queries/{q=$4} /Threads\_connected/{c=$4} /Threads\_running/{r=$4} END{printf "%d %d %d\n", q,c,r}'** *>> status.txt* sleep 1 **done** |

## 17.2 记录mysql的进程状态分布

在mysql命令行查看服务器状态：**SHOW PROCESSLIST**;

在linux命令行查看服务器状态：**mysql -uroot -p123456 -e 'show processlist \G'**

(如果查看不了，请检查mysql的用户名和密码是否正确)

我们只关心State状态这行，在linux下执行命令获取State：

**mysql -uroot -p123456 -e 'show processlist \G' | grep State**

一般使用脚本循环获取mysql进程状态，脚本内容如下：

|  |
| --- |
| **#!/bin/bash  while** true **do** mysql -uroot -e **'show processlist \G'** | grep State *>> process.txt* usleep 100000 **done** |

## 17.3 使用profile分析记录各个sql执行时间

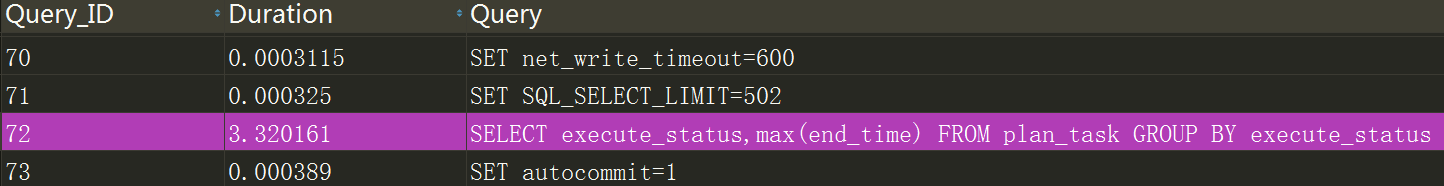
(1) 打开profile功能：

**SET PROFILING =1;**

打开profile功能后，执行的每一条sql语句都会被记录下来。

(2) 查看执行所有sql耗时时间列表(在phpstorm可以按使用时间排序)，可以找到有问题的语句(耗时比较大的)

**SHOW PROFILES;**



(3) 查看sql语句详细执行过程，可以分析该语句耗时都用在哪个状态下

**SHOW PROFILE for query 72;**

其中数字72是在SHOW PROFILES命令显示的查询id。

(4) 关闭profile功能：

**SET PROFILING =0;**

## 17.4 explain分析sql语句执行效果

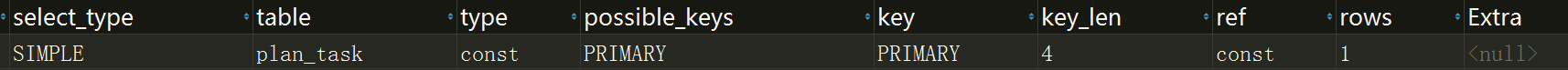
explain分析结果

|  |  |
| --- | --- |
| **属性名** | **说明** |
| **id** | 查询的编号，简单查询通常为1，如果有子查询会增加子查询行编号 |
| **select\_type** | 1. **simple**:不含子查询的简单的查询 2. **primary**:含有子查询或派生查询，(subquery表示非from子查询)、(derived表示from型子查询)、(union)、(union result) |
| **table** | 查询针对的表名，如果原表名加了别名，就显示表的别名，也有可能是null |
| **type** | 查询的方式，是分析查询过程的重要依据。   1. **All**: 全表扫描，在磁盘全表扫描是非常慢的，尽量避免出现全表扫描这样的查询。   例如：EXPLAIN SELECT \* FROM goods;   1. **index**: 扫描索引的所有节点，因为在内存扫描，性能比All要好，虽然说尽量使用索引，但不希望从头到尾扫描一遍。   例如：EXPLAIN SELECT \* FROM goods WHERE goods\_name LIKE '%诺基亚%';   1. **range**: 根据索引做范围扫描，比index性能好。   例如：EXPLAIN SELECT \* FROM goods WHERE click\_count>20;   1. **ref**: 通过索引列直接引用到某些数据行，比range性能好。   例如：EXPLAIN SELECT \* FROM goods WHERE cat\_id=3;   1. **eq\_ref**: 从表中读取一行，在联表查询使用到索引时经常出现，比ref性能好。   例如：EXPLAIN SELECT goods\_name,cat\_name FROM goods INNER JOIN category ON goods.cat\_id=category.cat\_id;   1. **const | null**: 查询是最快的，   例如：EXPLAIN SELECT \* FROM goods WHERE goods\_id=3;  例如：EXPLAIN SELECT count(\*) FROM goods; |
| **possible\_keys** | 可能用到的索引 |
| **key** | 最终使用的索引 |
| **key\_len** | 索引的长度，长度越短越好。 |
| **ref** | 引用了哪索引或列 |
| **rows** | 估计扫描的行 |
| **Extra** | 1. using index：使用索引覆盖 2. using where：使用了条件查询 3. using temporary：使用临时表，尽量不出现 4. using filesort：使用了排序，尽量不出现 5. range checked for each record：使用了在范围检查每个记录，不要出现 |

explain命令+耗时比较多sql查询语句可以查看该语句是否使用了索引，使用索引说明已经优化过，没有使用索引则对表进行优化。

实例1：

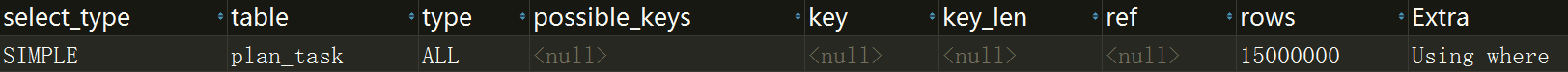
**EXPLAIN SELECT \* FROM plan\_task WHERE id=100000;**



从结果可以看出已经使用了索引。

实例2：

**EXPLAIN SELECT \* FROM plan\_task WHERE background\_color=100000;**



从结果看出没有使用到索引，直接使用where条件。

# 18 sysbench工具

## 18.1 sysbench安装

|  |  |
| --- | --- |
| 下载地址 | http://113.107.112.152/file3.data.weipan.cn/2655721/cbb900e63a99e8f955f672bdbe82dd7901486717?ip=1475943707,119.131.170.105&ssig=0m0CW7jRCI&Expires=1475945497&KID=sae,l30zoo1wmz&fn=sysbench-0.4.8.tar.gz&skiprd=2&se\_ip\_debug=119.131.170.105&corp=2&from=1221134&wsiphost=local |
| 解压 | tar zxvf sysbench-0.4.8.tar.gz |
| 进入目录 | cd sysbench-0.4.8 |
| 生成configure文件 | ./autogen.sh |
| 修改源码目录下aclocal.m4文件 | vim aclocal.m4  把LIBTOOL='$(SHELL) $(top\_builddir)/libtool'改成LIBTOOL='$(SHELL) /usr/bin/libtool' |
| 检查编译环境 | ./configure --prefix=/usr/local/sysbench --with-mysql-includes=/usr/local/mysql/include --with-mysql-libs=/usr/local/mysql/lib |
| 编译和安装 | make && make install |

## 18.2 sysbench测试

**(1) 测试CPU性能**

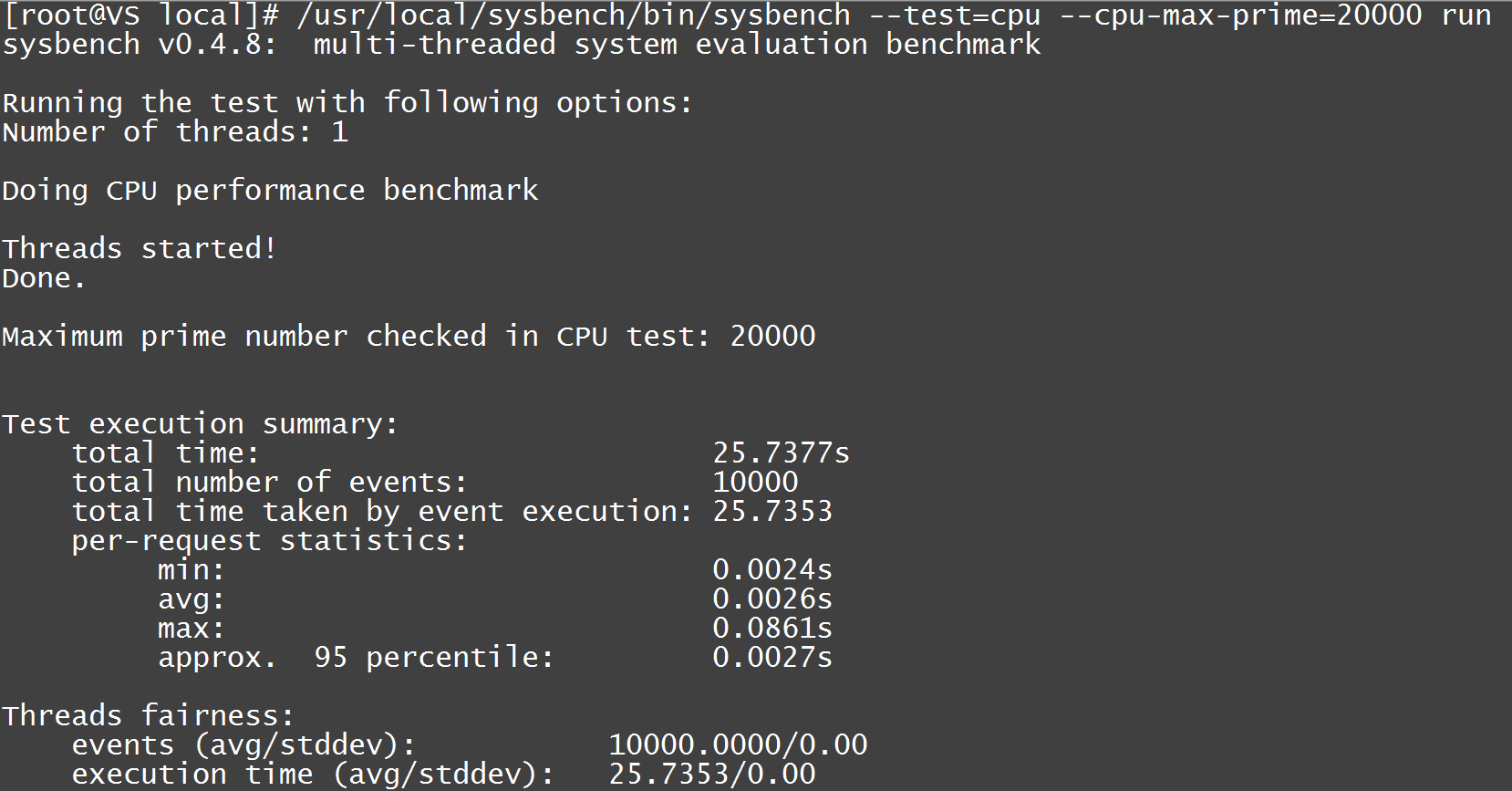
**/usr/local/sysbench/bin/sysbench --test=cpu --cpu-max-prime=20000 run**

cpu测试主要是查找素数，指定了最大的素数为20000，自己可以根据机器cpu的性能来适当调整数值。

注意: 服务器类型，有偏运算型的，有偏存储，所需要的指标不一样。

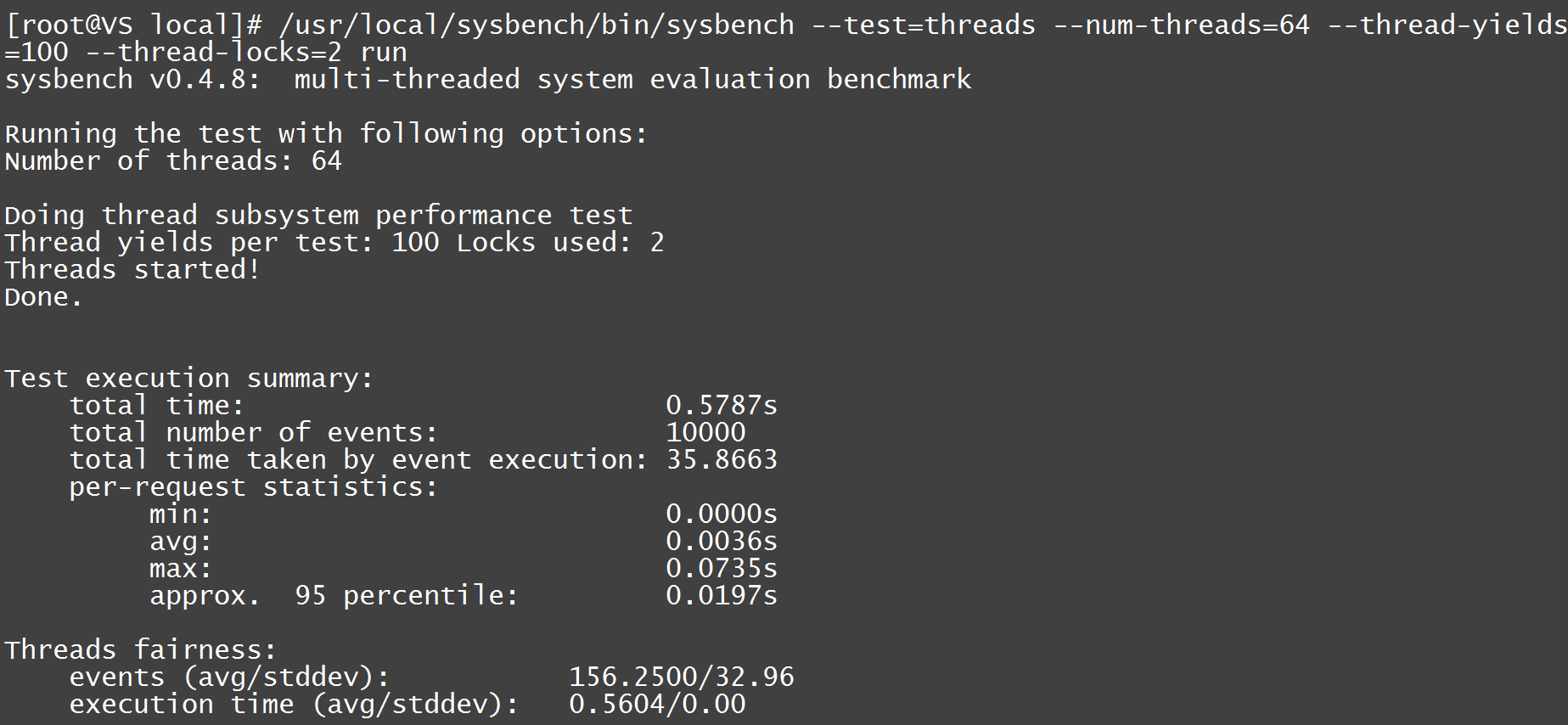
偏运算的(如视频转码服务器)要求CPU强，而存储则优先选择大容量和快速存储备.

测试的数据，孤立起来看，是没有意义的，数据要有比较才有意义, 比如多台服务器的测试数据，比较CPU性能。



**(2) 线程测试**

**/usr/local/sysbench/bin/sysbench --test=threads --num-threads=64 --thread-yields=100 --thread-locks=2 run**



**(3) 磁盘IO性能测试**

参数--file-test-mode说明

seqwr：顺序写入

seqrewq：顺序重写

seqrd：顺序读取

rndrd：随机读取

rndwr：随机写入

rndrw：混合随机读写

准备文件：

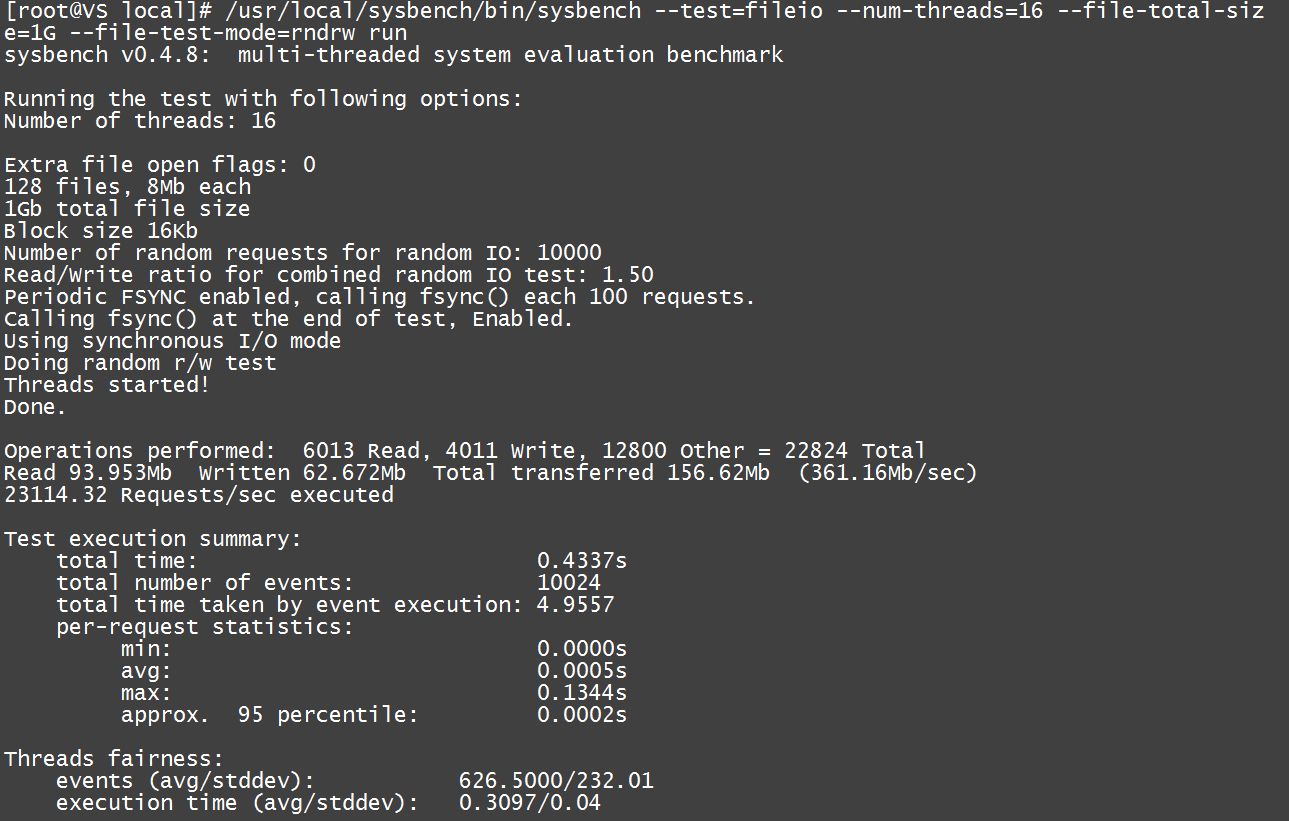
**/usr/local/sysbench/bin/sysbench --test=fileio --num-threads=16 --file-total-size=1G --file-test-mode=rndrw prepare**

压力测试：

**/usr/local/sysbench/bin/sysbench --test=fileio --num-threads=16 --file-total-size=1G --file-test-mode=rndrw run**

删除文件：

**/usr/local/sysbench/bin/sysbench --test=fileio --num-threads=16 --file-total-size=1G --file-test-mode=rndrw cleanup**



(4) 内存测试

**/usr/local/sysbench/bin/sysbench --test=memory --memory-block-size=8k --memory-total-size=1G run**

(5) OLTP测试

准备数据：过程是在test库下创建sbtest表，并随机插入3000条数据

**/usr/local/sysbench/bin/sysbench --test=oltp --mysql-table-engine=innodb --db-driver=mysql --mysql-db=test --oltp-table-name=sbtest --oltp-table-size=3000 --mysql-host=192.168.8.102 --mysql-user=root --mysql-password=123456 prepare**

运行测试：

**/usr/local/sysbench/bin/sysbench --test=oltp --mysql-table-engine=innodb --db-driver=mysql --mysql-db=test --oltp-table-name=sbtest --oltp-table-size=3000 --mysql-host=192.168.8.102 --mysql-user=root --mysql-password=123456 run**

删除测试表：

**/usr/local/sysbench/bin/sysbench --test=oltp --mysql-table-engine=innodb --db-driver=mysql --mysql-db=test --oltp-table-name=sbtest --oltp-table-size=3000 --mysql-host=192.168.8.102 --mysql-user=root --mysql-password=123456 cleanup**

测试 OLTP 时，准备表过程可以设置参数--mysql-table-engine为 myisam或Innodb等

sysbench也支持lua脚本自定义测试类型

例如参数--test=/home/vison/oltp.lua

# 19 模拟服务器周期性变化实验

在网站访问高峰期时，缓存数据也刚好失效，会造成大量请求查询mysql，这时为mysql请求高峰期，一段时间后，据大多客户请求的数据已经缓存，客户大多从缓存服务器读取数据，这时为mysql请求低谷期，当缓存数据过期失效后，又出现大量请求mysql，一直重复这个过程。

解决服务器周期性变化方法，把缓存失效期集中访问量一天最少的时候，例如凌晨2点到4点期间。

实验核心是不断查看mysql状态的3个参数，使用脚本执行命令每秒读取mysql状态一次。

读取mysql状态命令：

**mysqladmin -uroot -p123456 ext | awk '/Queries/{q=$4} /Threads\_connected/{c=$4} /Threads\_running/{r=$4} END{printf "%d %d %d\n", q,c,r}'**

文件说明：

|  |  |
| --- | --- |
| mem.php | 判断memcached是否有数据，没有则从mysql读取数据，再把数据写入memcached |
| schedule.sql | mysql创建库、创建表、随机填充数据语句 |
| status.sh | 每秒读取mysql状态脚本 |

实验前准备：

把php整合到nginx，把mem.php复制到网站根目录下

打开schedule.sql，按步骤执行sql语句，生成一张有内容的表。

操作步骤：

(1) 启动nginx服务器

**/usr/local/nginx/sbin/nginx**

(2) 启动php进程

**/usr/local/php/sbin/php-fpm**

(3) 启动memcached服务

**memcached -u nobody -m 256 -d**

(4) 启动mysql服务(一般开机默认启动)

**mysql/bin/mysqld**

(5) 执行status.sh脚本，获取mysql查询数量、线程连接数、线程运行数，把每秒读取的数据写入status.txt

**sh status.sh**

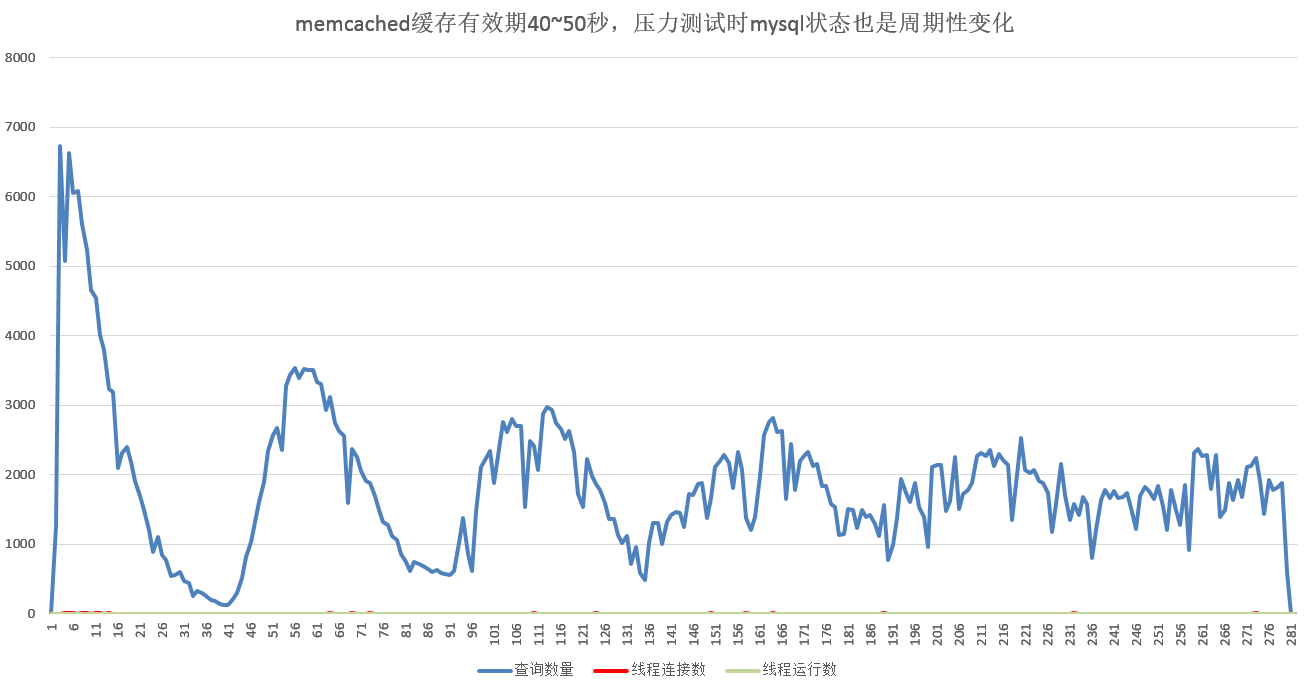
(6) 使用ab工具测试

**./ab -c 20 -n 1000000 http://192.168.8.102/mem.php**

(7) 把获取mysql状态结果统计出每一秒查询的数量，统计结果保存到result.txt

**awk '{q=$1-last;last=$1} {printf("%d %d %d\n", q, $2, $3)}' bench.txt >> result.txt**

(8) 用notepad文本编辑器打开result.txt，用制表分隔符'\t'替换原来数据之间的空格分隔符，然后直接复制数据到exel表格，插入图标就可以绘制出图形来，如下图所示：



mem.php文件内容：

|  |
| --- |
| **<?php  function** miConn(){ *// 初始化数据库* **static** $conn=**null**;  **if** ($conn===**null**){  $conn=**new** mysqli(**"192.168.8.102"**,**"root"**,**"123456"**,**"schedule"**); *// 连接数据库* **if** ($conn->**connect\_errno**) {  }   $conn->query(**'set names utf8'**); *// 设置字符编码* }  **return** $conn; }   $memcache = **new** Memcache; *// 实例化对象* **if** ($memcache->connect(**'192.168.8.102'**, 11211)==**false**){ *// 连接缓存服务器* **echo "连接缓存服务器失败"**;  **return**; }  $mkey=*rand*(1,30000)+13000000;  *// 从缓存服务器中获取key的值，并判断值key是否存在，不存在则添加数据* **if** ($memcache->get($mkey)==**false**){*// 获取数据并判断是否存在* $conn=miConn();  **if** ($conn===**null**){  **return**;  }   $sql = **"select id,execute\_type,execute\_status,FROM\_UNIXTIME(start\_time, '%Y-%m-%d %h:%i:%s'),FROM\_UNIXTIME(end\_time, '%Y-%m-%d %h:%i:%s') from plan\_task where id="** . $mkey;  **if** ($result=$conn->query($sql)){  $memcache->add($mkey, $result, **false**, *mt\_rand*(40,50)); *//缓存数据* } }  $conn->close();*// 关闭mysql连接* $memcache->close(); *// 关闭memcached连接* |

schedule.sql文件内容：

|  |
| --- |
| *# (1)创建新的库* **CREATE DATABASE schedule**;  *# (2)进入schedule数据库* **USE schedule**;  *# (3)在schedule数据库下创建表* **CREATE TABLE** plan\_task (  **id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT NOT NULL**, *# 主键* **creat\_at TIMESTAMP NOT NULL DEFAULT** *CURRENT\_TIMESTAMP*, *# 创建时间* **task\_no INT UNSIGNED NOT NULL DEFAULT** 0, *# 任务编号* **execute\_type TINYINT NOT NULL DEFAULT** 0, *# 执行任务类型* **execute\_status TINYINT NOT NULL DEFAULT** 0, *# 当前执行状态* **background\_color MEDIUMINT UNSIGNED NOT NULL DEFAULT** 0, *# 背景颜色* **font\_color MEDIUMINT UNSIGNED NOT NULL DEFAULT** 0, *# 字体颜色* **start\_time INT UNSIGNED NOT NULL DEFAULT** 0, *# 任务开始时间* **end\_time INT UNSIGNED NOT NULL DEFAULT** 0 *# 任务结束时间* )**ENGINE** myisam **DEFAULT CHARSET** *utf8*;   *# (4)模拟数据生成器函数（在规定范围随机生成），用来填充表数据 # 注意，这个函数不能在终端执行，因为分号冲突，需要使用phpstorm或phpMyadmin执行* **CREATE PROCEDURE** *generate\_data*(cnt **INT**)  **BEGIN  DECLARE** st **INT UNSIGNED DEFAULT** 0;  **DECLARE** et **INT UNSIGNED DEFAULT** 0;  **WHILE** cnt > 0 **DO  SET** cnt := cnt - 1;  **SET** st := *floor*(*rand*() \* 31622401) + 1451577600;  **SET** et := st + *floor*(*rand*() \* 1740) + 60;  **INSERT INTO** plan\_task (**task\_no**, **execute\_type**, **execute\_status**, **background\_color**, **font\_color**, **start\_time**, **end\_time**)  **VALUES** (*floor*(*rand*() \* 100), *floor*(*rand*() \* 2), *floor*(*rand*() \* 5), *floor*(*rand*() \* 16777216), *floor*(*rand*() \* 16777216), st, et);  **END WHILE**;  **END**;  *# (5)添加150000000行数据，随机生成* **CALL** *generate\_data*(5000000); **CALL** *generate\_data*(5000000); **CALL** *generate\_data*(5000000); |

status.sh文件内容：

|  |
| --- |
| **#!/bin/bash  while** true **do** mysqladmin -uroot ext | awk **'/Queries/{q=$4} /Threads\_connected/{c=$4} /Threads\_running/{r=$4} END{printf "%d %d %d\n", q,c,r}'** *>> status.txt* sleep 1 **done** |

# 20 观察服务器进程状态实验

实验目的：

观察mysql是否出现比较耗时的状态，比较耗时状态包括：

|  |  |
| --- | --- |
| **状态名称** | **产生原因** |
| converting HEAP to MyISAM  查询结果太大时，把结果放在磁盘 | 一次性查询出来结果太大，内存装不下。例如select \* from big\_table; |
| create tmp table  创建临时表 | 一般合理利用索引就比较少出现创建临时表  (1) group by的列和order by的列不相同时，  (2) distinct和order by一起使用  (3) 开启了SQL\_SMALL\_RESULT选项 |
| copying to tmp table on disk  把内存临时表复制到磁盘 | (1) 取出的列含有text/blob类型时(内存表不支持)  (2) 在group by或distinct的列中存在>512字节的string类型的列  (3) select 中含有>512字节的string类型的列同时又使用了union或union all语句 |
| locked  被其他查询锁住 |  |
| logging slow query  记录慢查询 |  |
| sending data  发送数据 | 当查询结果比较多时，大部分时间都浪费在发送数据上，应该尽量减少 |

通过mysql内置强大的profile工具可以查看执行每条语句内部执行过程和耗时。

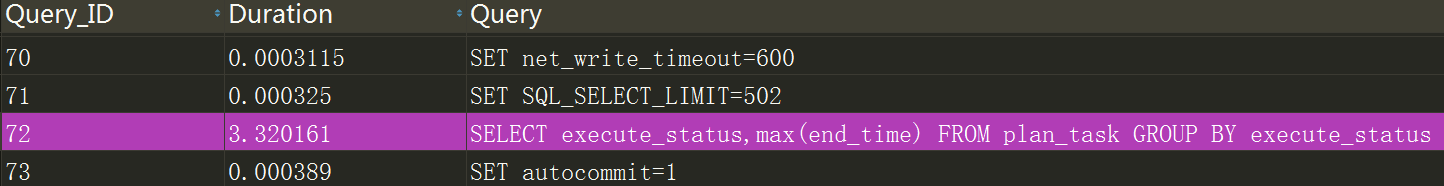
(1) 打开profile功能：

**SET PROFILING =1;**

打开profile功能后，执行的每一条sql语句都会被记录下来。

(2) 查看执行所有sql耗时时间列表，可以找到有问题的语句(耗时比较大的)

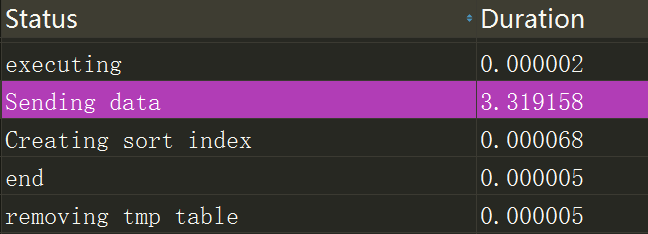
**SHOW PROFILES;**



(3) 查看sql语句详细执行过程，可以分析该语句耗时都用在哪个状态下

**SHOW PROFILE for query 72;**

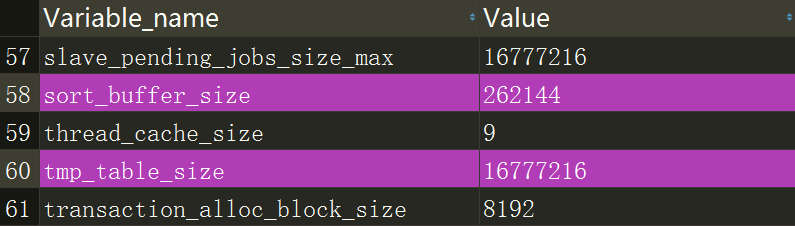
其中数字72是在SHOW PROFILES命令显示的查询id。



查看mysql默认变量值：

在mysql客户端通过命令查看mysql默认临时表、排序缓存大小等

**SHOW VARIABLES LIKE '%size%';**



可以通过设置变量修改其大小

**SET GLOBAL TMP\_TABLE\_SIZE = 33554432;  
SET SESSION TMP\_TABLE\_SIZE = 33554432;**

实验：

文件说明

|  |  |
| --- | --- |
| process.sh | 记录服务器状态脚本 |

实验步骤：

(1) 启动mysql服务(一般开机默认启动)

**/usr/local/mysql/bin/mysqld**

(2) 准备数据：过程是在test库下创建sbtest表，并随机插入3000条数据

**/usr/local/sysbench/bin/sysbench --test=oltp --mysql-table-engine=innodb --db-driver=mysql --mysql-db=test --oltp-table-name=sbtest --oltp-table-size=3000 --mysql-host=192.168.8.102 --mysql-user=root --mysql-password=123456 prepare**

(3) 打开profile功能

**SET PROFILING =1;**

(4) 执行process.sh脚本

**sh process.sh**

(5) mysql压力测试

**/usr/local/sysbench/bin/sysbench --test=oltp --mysql-table-engine=innodb --db-driver=mysql --mysql-db=test --oltp-table-name=sbtest --oltp-table-size=3000 --mysql-host=192.168.8.102 --mysql-user=root --mysql-password=123456 run**

(6) 在压力测试过程中手动执行比较耗时的sql语句，比如创建临时表、排序、连接查询等

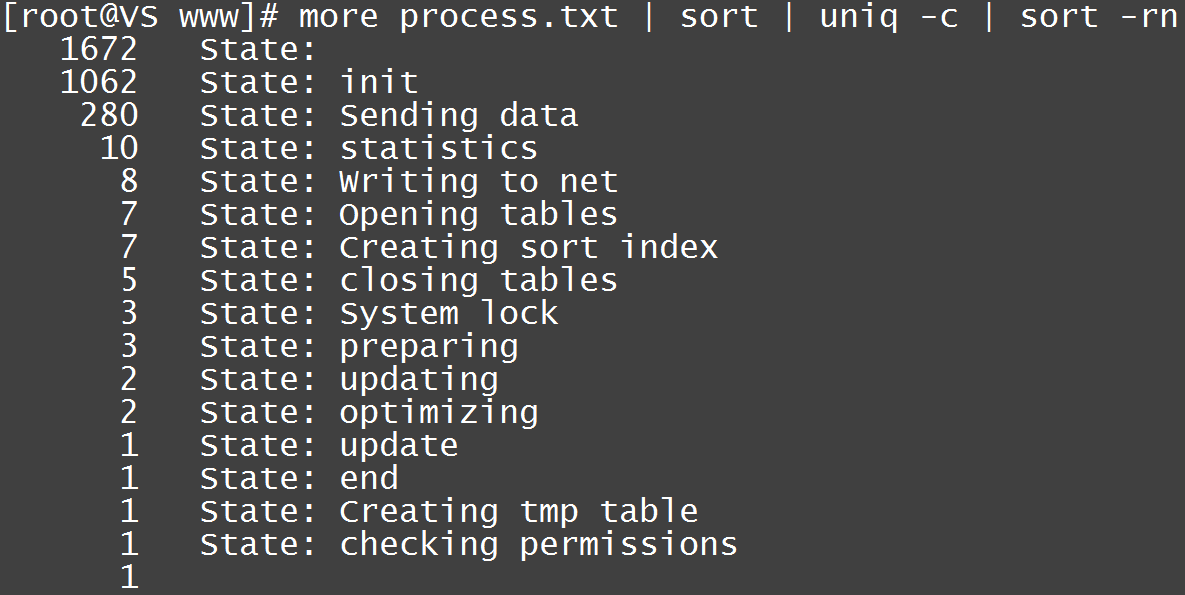
**SELECT execute\_type,max(start\_time) FROM plan\_task GROUP BY execute\_type;**

**SELECT execute\_status,max(end\_time) FROM plan\_task GROUP BY execute\_status;**

**SELECT \* FROM plan\_task GROUP BY execute\_type ORDER BY start\_time;**

(7) 压力测试结束后关闭执行process.sh脚本，mysql状态保存在process.txt中，然后对process.txt进行统计

**more process.txt | sort | uniq -c | sort -rn**



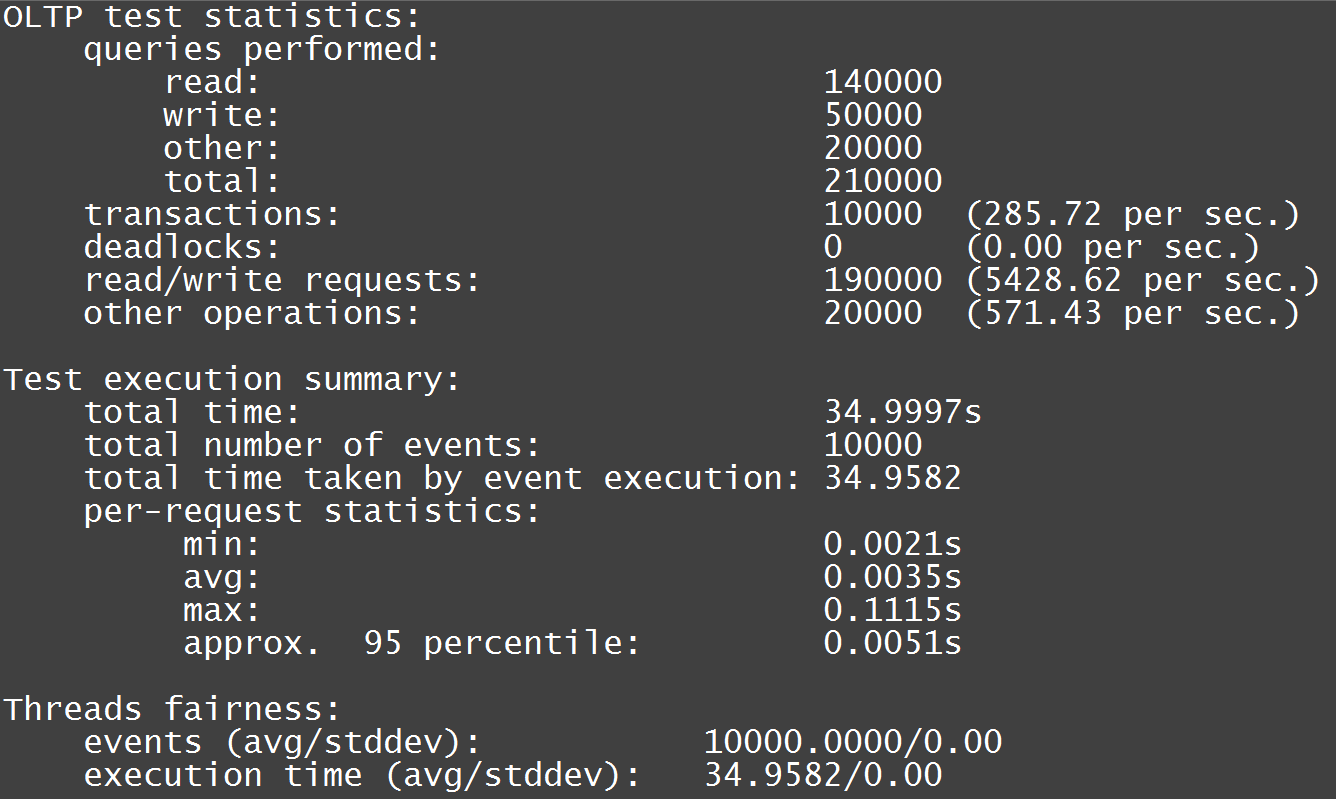
(8) 通过profile查看哪些sql语句比较耗时，在phpstorm可以按时间顺序排列

**SHOW PROFILES;**

(9) 通过profile查看sql语句耗时在哪个执行状态

**SHOW PROFILE for query 查询id;**

sysbench压力测试mysql结果：



process.sh文件内容：

|  |
| --- |
| **#!/bin/bash  while** true **do** mysql -uroot -e **'show processlist \G'** | grep State *>> process.txt* usleep 100000 **done** |

# 21 表的优化

列选择原则：

(1) 字段类型优先选择顺序

整型 > date、time > char 、varchar > blob

因为整形、time类型运算快又节省空间；char、varchar要考虑字符集的转换与排序时的校对集(a B排序哪个优先)、速度慢；blob无法使用内存临时表，设计到排序必须在硬盘完成，速度慢。

(2) 字段长度够用就行(tinyint、 varchar(N))

因为大的字段浪费内存，影响速度，例如某字段varchar(30)能存下的内容，如果用varchar(100)的话，在联表查询时，varchar(100)要花跟多内存。

(3) 避免使用null

因为null不利于索引，要用多一个字节来特殊标注该字段值是否为null，在磁盘上占据多一个字节。一般声明字段时都带有not null define属性

(4) 对于字段属性分类数量确定而且数量比较少时，优先选择枚举类型enum，例如性别、学历等

因为enum列在内部是用整型来存储的

优点：当一个表enum列与另一个表的enum相关联时速度最快，当enum成员是char类型，并且字节比较多时，enum依然时整型存储，可以节省IO。

缺点：在碰到与char关联时，需要转化消耗点时间，速度要比enum<->enum和char<->char要慢。

# 22索引优化

满足理想索引

1. 查询频繁
2. 区分度高
3. 长度小
4. 尽量能覆盖常用查询字段

## B-tree索引

B-tree（多路搜索树，并不是二叉的）是一种常见的数据结构。使用B-tree结构可以显著减少定位记录时所经历的中间过程，从而加快存取速度。按照翻译，B 通常认为是Balance的简称。这个数据结构一般用于数据库的索引，综合效率较高，常被用于对检索时间要求苛刻的场合。

B-tree可以理解为已经**排好序**的快速查找数据结构。

btree的左前缀规则：

1. 按f1、f2......建立的复合索引，在where条件中，按f1、f2......由左到右的顺序（当and时不用按顺序），索引才会发挥作用。
2. 如果中间某列没有条件或like条件，导致后面的列，索引用不上。
3. 索引也能用于排序和分组，因为分组要先排序后在计算。所以我们的order by或group如果能针对有顺序的表进行，可以避免临时表和文件排序。也就是说我们的order by或group按顺序使用索引的列，则可发挥索引的作用。

B-tree索引的误区：在表中的常用的列都独立加上索引就以为常用的查询都完全用到索引了。

例如user表中的gender和age列都独立加上索引，

查询出表中大于60岁的女性的语句：

select \* from user where gender='女' and age>60;

注意：上面语句只能用上gender或age索引，因为是独立索引，同时只能用上一个，上面sql查询要完全使用索引，需要建立多列索引，例如：index(gender,age)

## hash索引

哈希索引包含以数组形式组织的 Bucket 集合。哈希函数将索引键映射到哈希索引中对应的 Bucket。使用哈希索引必须要使用哈希集群。哈希索引可能是访问数据库中数据的最快方法(时间复杂度为O(1))，但它也有自身的缺点，只支持等值计算，不支持范围搜索或排序

问：hash索引查找速度最快，为什么不用hash索引呢？

答：(1) 哈希函数计算的结果是随机的，如果在磁盘上随机放置数据，hash找到数据在磁盘位置很快，但是磁盘随机读取数据却很慢。

(2) hash无法对范围查询进行优化

(3) hash无法利用前缀索引，不能前缀优化。例如在mysql查找列值为zhangsan，B-tree索引支持'zhang'或'zhangsan'关键字索引，而hash必须为'zhangsan'全部搜索。

(4) 排序也无法优化

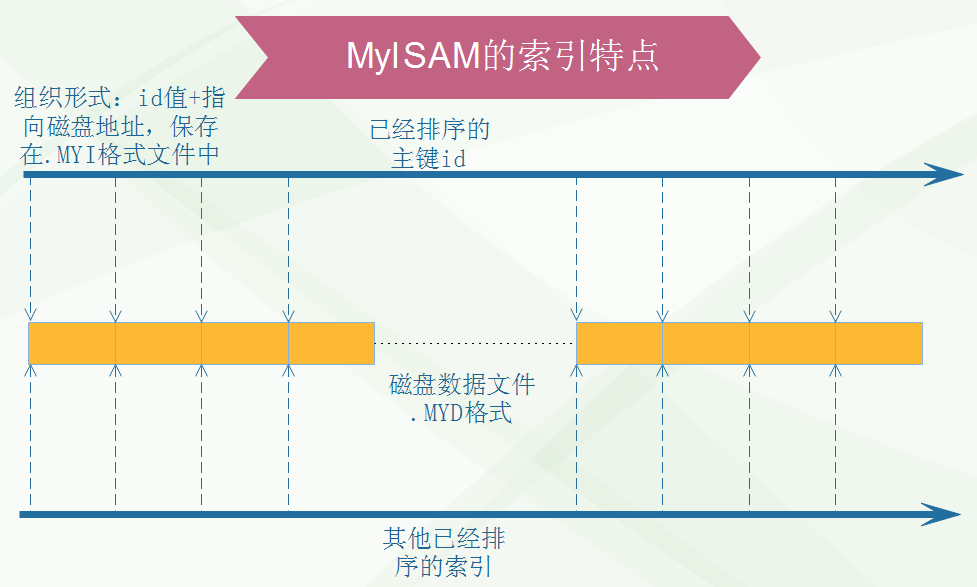
B-tree索引和hash索引区别

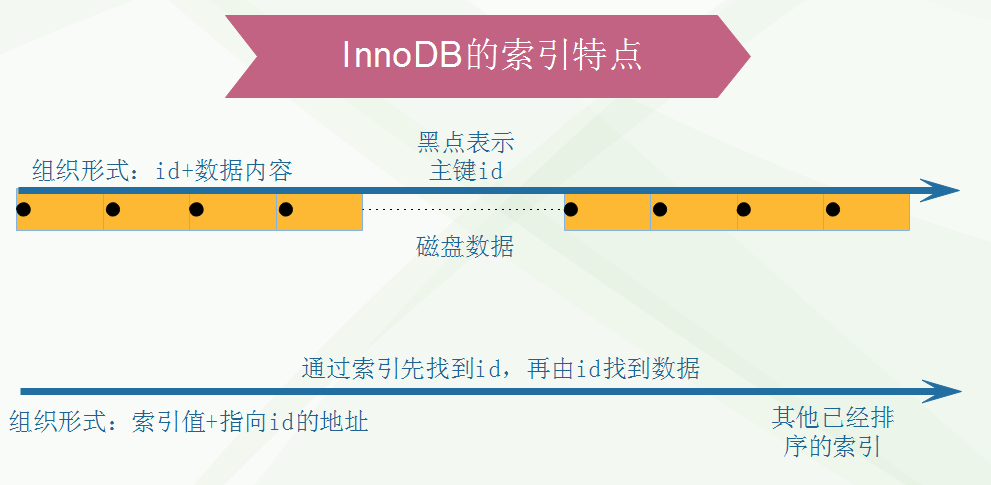
1. hash索引查找数据基本上能一次定位数据，当然有大量碰撞的话性能也会下降。而btree索引就得在节点上挨着查找了，很明显在数据精确查找方面hash索引的效率是要高于btree的。
2. 那么不精确查找呢，也很明显，因为hash算法是基于等值计算的，所以对于“like”等范围查找hash索引无效，不支持。
3. 对于btree支持的联合索引的最优前缀，hash也是无法支持的，联合索引中的字段要么全用要么全不用。
4. hash不支持索引排序，索引值和计算出来的hash值大小并不一定一致。

## 聚簇索引

MyISAM和InnoDB的索引特点

MyISAM和InnoDB的索引特点，各自指向方式不同。





**MyISAM的索引特点**：指向的是数据在磁盘上的位置。

**InnoDB的索引特点**：

1. 主键作为索引，既储存索引值有储存行数据。
2. 次级索引是先指向主键id，然后再从id获取行数据，也就是说InnoDB的索引是间接获取数据，中间多了从id获取数据的过程。
3. 如果没有主键系统把unique key作为主键，如果也没有unique key，系统内部自动生成row id作为主键。

**聚簇索引**：索引和数据混合一起，如InnoDB的id和数据是混合在一起的。有个缺点，当数据文件比较大时，查询数据时不断翻越磁盘扇区的页，使用时间会比较长。

**回行**：从索引定位到获取磁盘数据的过程，查询时回行时比较耗时，索引查找是快的，回行去数据是慢的。如果能直接从索引中获取数据，将会省去回行过程，提高查询速度，因为索引在内存运行的。

**索引覆盖**：查询的列恰好时索引的一部分，只在索引就能获取想要的数据，不需要回行到磁盘取数据。如果查询的数据能用到索引覆盖，速度是最快的。用explain 查询语句，看extra项目是否有Using index，如果有则使用到索引覆盖。

注意：主键值为随机的插入行数据时，主键节点会分裂，对于MyISAM类型的表，影响不是很大，因为节点的包括的内容比较小(只有指向磁盘地址)，在内存里完成，转移数据时耗时小；对于InnoDB类型的表影响比较大，因为表使用的聚簇索引，每个节点都包括行内容，节点分裂时需要转移的数据比较多，耗时也比较多。

所以高性能索引是使用自动递增的整型，例如定义主键类型**id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT**;

插入数据时不用指定主键值，让其有规律的自动增加，减少节点分裂的耗时。

**聚簇索引排序慢原因分析**

例如有一个表结构和引擎如下，有10000行数据。

|  |
| --- |
| **CREATE TABLE** ts(  **id CHAR(30) PRIMARY KEY AUTO\_INCREMENT**,  **val INT NOT NULL DEFAULT** 0,  **str1 VARCHAR**(2000),  **str2 VARCHAR**(2000),  **KEY** idval(**id**,**val**) )**ENGINE** innodb **CHARSET** *utf8*; |

下面有两条排序语句速度相差很大

|  |
| --- |
| *#速度慢*  **SELECT id FROM** ts **ORDER BY id**;  *#速度快*  **SELECT id FROM** ts **ORDER BY id**,**val**; |

分析原因：表驱动为InnoDB，为聚簇索引。

对主键id进行排序慢的原因：因为主键索引id属于聚簇索引，每个节点包含一行数据内容，而且字段内容比较大，储存时超越磁盘最小块，对主键id排序时会跨很多磁盘的数据块，导致速度很慢。

对联合索引排序快的原因：因为联合索引不是聚簇索引，节点内容很小(只有指向主机id地址)，联合排序时使用了索引覆盖，不需要回行取数据，而且是在内存中完成，索引速度比较快。

当把str1和str2内容比较大的列删除后，两条语句执行速度差别不大。

如果表类驱动型为MyISAM两条语句执行速度差别不大。

## 多列索引

在建立多列索引后，必须满足左前缀要求(从左到右按顺序，中间不能断开)，索引才能发挥作用。

例如多列索引index(a,b,c)

|  |  |
| --- | --- |
| sql的where语句 | 索引是否发挥作用 |
| where a=1 | **√ 使用a列** |
| where a=1 and b=2 | **√ 使用a、b列** |
| where a=1 and b=2 and c=3 | **√ a、b、c列都使用** |
| where b=2  where c=3 | **×** |
| where a=1 and c=3  where b=2 and c=3 | **×** |
| where a=1 and b>2 and c=3  where a=1 and b like '%2' and c=3 | **√** **使用a、b列** |

使用多列索引误区：查询哪个列索引都会发挥作用。

注意：多列索引一定要结合业务逻辑进行优化，例如查100~200元的男装商品，这条查询涉及到价格和商品栏目两列，可以把这两列作为一个多列索引。

## 多列索引优化实验

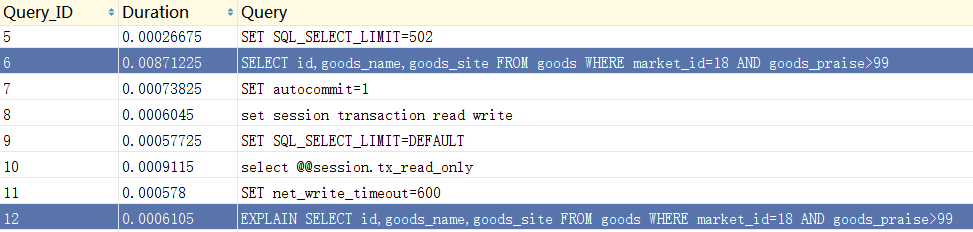
使用的表时商品表，大概有8000行数据左右

索引优化前使用的索引时列的独立索引，当有几个列同时作为查找条件时，只能有一个列作为索引。

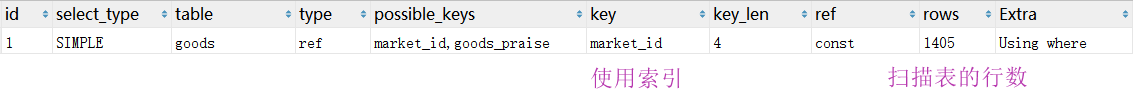
### 优化前

|  |
| --- |
| *# 打开语句分析命令* **SET** PROFILING = 1;  *# 查询市场分类为18并且评分99分以上的商品*  **SELECT id**,**goods\_name**,**goods\_site FROM** goods **WHERE market\_id**=18 **AND goods\_praise**>=99;  *# 查看是否使用索引*  **EXPLAIN SELECT id**,**goods\_name**,**goods\_site FROM** goods **WHERE market\_id**=18 **AND goods\_praise**>=99;  *# 打开语句分析结果*  **SHOW PROFILES** ; |

profile分析结果如下图：



explain分析结果如下图：



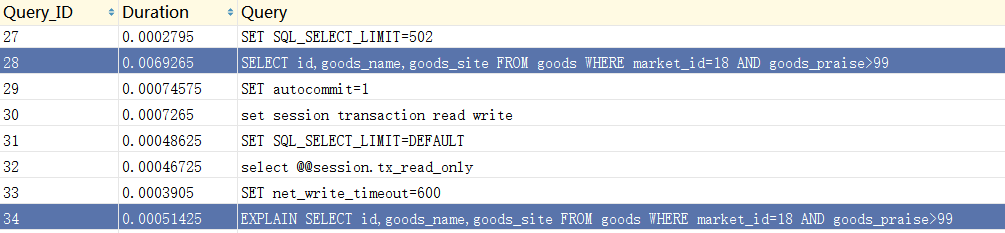
用profile，explain命令分析结果：这次的select语句查询使用了8.7ms，使用的索引是market\_id，扫描表的行数一共1405行，额外使用了where条件。

### 优化后

添加market\_id和goods\_praise联合的多列索引，然后explain同样的sql语句

|  |
| --- |
| *# 添加多列索引*  **ALTER TABLE** goods **ADD INDEX** markey\_praise(**market\_id**,**goods\_praise**);  *# 打开语句分析命令* **SET** PROFILING = 1;  *# 查询市场分类为18并且评分99分以上的商品*  **SELECT id**,**goods\_name**,**goods\_site FROM** goods **WHERE market\_id**=18 **AND goods\_praise**>=99;  *# 查看是否使用索引*  **EXPLAIN SELECT id**,**goods\_name**,**goods\_site FROM** goods **WHERE market\_id**=18 **AND goods\_praise**>=99;  *# 打开语句分析结果*  **SHOW PROFILES** ; |

profile分析结果如下图：



explain分析结果如下图：



用profile，explain命令分析结果：这次的select语句查询使用了6.9ms，使用的索引是market\_praise，扫描表的行数一共2369行，没有额外使用了where条件，使用索引索引覆盖条件。

### 结果

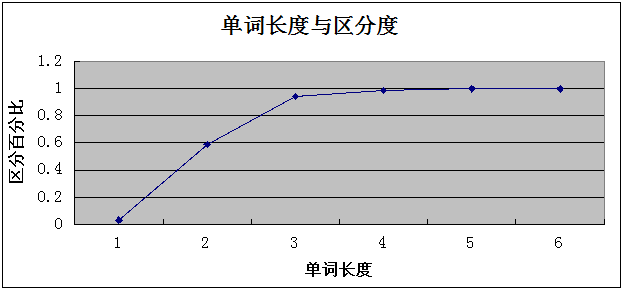
优化后比优化前查询速度提升了1.1ms，虽然扫描行数量多了，但是都是在索引扫描的，比在原表扫描快很多。

## 索引的长度和区分度平衡实验

索引的长度和区分度是矛盾的，长度大区分度高，反之区分度小。到底选多少个字节作为索引长度比较合理呢，需要根据表内容的实际试验。

一般使用方法，针对表截取长度和区分度测试，例如一个中文单词表，共一万多行，有一个字到14个字的行都有。通过测试从1个词开始测试，使用sql语句**SELECT** *count*(**distinct** *left*(word,6))/*count*(*\**) **FROM** dict;

测试从1到6，最后得出结果



对于一般系统，区别度能达到10%时，索引性能就可以接受。

针对列中的值，从左到右截取来建立索引

1. 截得越短，重复度越高，区分度越低，索引效果越不好。
2. 截得越长，重复度越低，区分度越高，索引效果越好，但是带来负面影响越大，占有空间越大，也会减慢查询速度，增删改操作变慢。

## 伪哈希索引

在实际中的表的列有可能比较长，例如网址url的列，网址前面的几个字节<http://www都是相同的，如果把url列作为索引，前面10>个字节都是重复的，要区分网址至少截取16个字节以上，使得索引长度很长。不利于增删改查。

为了解决这个问题，在表中多添加一个url对应hash值的列(例如列名urlhash)，即url值对应唯一hash值，然后使用列urlhash作为索引(整型)，从而大大减少索引的长度，提高查询速度。

伪哈希索引伪哈希索引

实验：

|  |
| --- |
| *# (1)创建一个表* **CREATE TABLE** company(  **id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT**,  **url VARCHAR**(50) **NOT NULL DEFAULT ''**,  **INDEX** url(**url**(15)) )**ENGINE** myisam **CHARSET** *utf8*;  *# (2)插入3行数据* **INSERT INTO** company (**url**) **VALUES** (**'http://www.baidu.com'**),(**'http://www.qq.com'**),(**'http://www.taobao.com'**);  *# (3)查看表数据* **SELECT** *\** **FROM** company;  *# (4)使用explain分析查询语句是否用到索引* **EXPLAIN SELECT** *\** **FROM** company **WHERE url**=**'http://www.qq.com'**;  *explain分析索引结果如下图，从图得到索引长度达到47个字节。*    *# (5) 添加hash列* **ALTER TABLE** company **ADD COLUMN url\_hash INT UNSIGNED NOT NULL DEFAULT** 0;  *# (6) 删除原来url索引，添加url\_hash索引* **ALTER TABLE** company **DROP INDEX** url; **ALTER TABLE** company **ADD INDEX** url\_hash(**url\_hash**);  *# (7) 把网站映射出hash值存放到对应的url\_hash列上* **UPDATE** company **SET url\_hash**=*crc32*(**url**);  *# (8) 使用explain分析查询语句是否用到索引* **EXPLAIN SELECT** *\** **FROM** company **WHERE url\_hash**=*crc32*(**'http://www.qq.com'**);  *explain分析索引结果如下图，从图得到索引长度达到4个字节。* |

实验结果：使用了伪hash索引解决长索引的问题，提高查询速度。

## 延时索引技巧

用大量数据分页优化说明延时索引技巧。

例如显示搜索结果有100000条，分页显示每页20条，sql语句为

**SELECT** filed **FROM** table **LIMIT** (N-1)\*20,20;其中N表示第几页。

(1) 优先从业务逻辑优化，条件是限制分页数量，也就是说搜到100000条结果，给用户显示最多是40页就以及满足客户需求了，查询时间是ms级别。

为什么要限制N的大小呢？因为limit的偏移量(offset)很大时，效率非常低，limit offset,num的工作机制是先查询，然后再跳过offset获取num条数据，当offset很大时，说明查询结果的数据很大，通过开启profile分析sql语句，发现大部分时间都用在Sending data(即回行传输的数据)上了，限制N的大小非常有必要，百度或谷歌都是现在查询结果的页数，一般都是看前几页搜索结果，极少人回去翻页到几十页去看搜索结果。

(2) 如果不允许从业务逻辑优化，还有个办法就是利用主键id作为查找条件再筛选结果，大量数据查询时间是ms级别。

例如sql语句：**SELECT** filed **FROM** baidu **WHERE** id>100000 **LIMIT** 0,20;

这种优化查询用到索引所以速度也很快，但是条件是要求数据完整性，不允许删除数据。否则会造成每次查询结果不一致，解决办法时不进行物理删除，用逻辑标记删除，最终在页面上显示时，逻辑删除的条目不显示结果即可。

(3) 延时索引优化，这种方法没有条件限制，大量数据查询时间为秒级别。

例如语句**SELECT** filed **FROM table WHERE** id **IN** (**SELECT** id **FROM table WHERE** id>200000 **LIMIT** (N-1)\*20,20);

首先查找取出主键id，然后再通过id来取数据，这种通过索引中间过程再从表中取数据过程叫延时索引，延时索引的好处时，节省了大量的回行时间(sending data时间)，提高查询速度。

上面语句的工作过程是通过分页去取主键id，取主键id过程用到了索引覆盖，速度比较快，省去回行过程，获取id之后在回行取表中的内容，因为回行数量很少，sending data时间也少，索引查询速度也快，当数据量很大时，上面语句的据大多数时间都在取id过程上了。

当数据量很大时，利用延时索引比直接取数据速度要快4倍左右。

## 索引与排序

对于带有排序的sql语句，例如下面语句

**SELECT** *\** **FROM** goods **WHERE cat\_id**=3 **ORDER BY shop\_price DESC**;

可能发生2种情况：

(1) 跳过排序过程，直接取出最终结果，查询速度快。要达到这个结果，必须事先建立索引，利用索引本身有序的特点，取出来就是有顺序的，所以省略了排序过程，达到查询速度快的效果。对sql进行explain分析结果如下图：



上图中的extra字段并没有出现filesort提示，说明没有经过排序过程，而且使用了cat\_price联合索引。

(2) 先从表中取出数据作为临时表，然后再对临时做排序(在内存或磁盘排序)，排序是比较耗时的，因尽量避免，所以查询速度会比较慢。

现在把cat\_price联合索引删除，再查看explain分析sql语句结果

|  |
| --- |
| **ALTER TABLE** goods **DROP INDEX** cat\_price;  **EXPLAIN SELECT** *\** **FROM** goods **WHERE cat\_id**=3 **ORDER BY shop\_price**; |

上图中的extra字段出现了filesort提示，说明有经过排序过程。

所以在碰到有排序的查询时，排序的列尽量作为索引(或联合索引)，目的是取出来的数据本身就是有顺序的，避免排序这个耗时的过程，从而提高查询速度。

## 冗余索引

在某个列上可能存在多个索引，例如下面这个表

|  |
| --- |
| **CREATE TABLE user**(  **id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT**,  **first\_name CHAR**(5) **NOT NULL DEFAULT ''**,  **last\_name CHAR**(10) **NOT NULL DEFAULT ''**,  **KEY** first\_name(**first\_name**),  **KEY** last\_name(**last\_name**),  **KEY** full\_name(**first\_name**,**last\_name**) )**ENGINE** myisam **CHARSET** *utf8*; |

表中的列first\_name既有独立作为索引，也有联合索引full\_name，这叫冗余索引，实际中也会用到，单列查询时可以用到索引，多列查询时也可以用到索引。

注意index AB(A,B)和index BA(B,A)是不同的索引，对于列A或B来说，AB和BA属于冗余索引。

## 索引碎片和维护

索引碎片的形成原因：在长期的数据更改中，索引文件和数据文件都将会产生空洞，形成碎片。

索引碎片会对查询速度会有影响

维护索引碎片有两种方法：

(1) 修改表的nop操作，这个操作不影响表数据

**ALTER TABLE** 表名 **ENGINE** 驱动名;

(2) **OPTIMIZE TABLE** 表名

注意：当数据比较大时，维护是很耗时间的，通常在访问量比较少的夜里维护，当数据修改不频繁可以按年来做维护，如果数据修改比较多可以按月来修复。

# 23 子查询、连接查询的效率比较

## 23.1 子查询实验

下面通过实验来验证子查询有索引和没有索引的执行的过程。

实验前：

有两张表，表名分别是plan\_task和num，plan\_task表有15000000行数据，而num表很简单，只有一列id，且只有8行数据。

实验过程：

|  |
| --- |
| *# (1) 测试没有索引的子查询(plan\_task的font\_color没有建索引)* **SELECT** *\** **FROM** plan\_task **WHERE** font\_color **IN** (**SELECT id FROM number**); *# 查询时间：9 rows retrieved starting from 1 in 5s 826ms (execution: 5s 808ms, fetching: 18ms)*  *# 使用explain分查询语句* **EXPLAIN SELECT** *\** **FROM** plan\_task **WHERE** font\_color **IN** (**SELECT id FROM number**);    *# (2) 测试有索引的子查询(plan\_task的主键索引id)* **SELECT** *\** **FROM** plan\_task **WHERE** id **IN** (**SELECT id FROM number**); *# 查询时间：8 rows retrieved starting from 1 in 21ms (execution: 5ms, fetching: 16ms)*  *# 使用explain分析查询语句* **EXPLAIN SELECT** *\** **FROM** plan\_task **WHERE** id **IN** (**SELECT id FROM number**); |

实验结果：

(1) 没有索引的子查询，执行过程是先执行主语句，再执行子语句，然后主语句逐个去匹配子语句结果。从explain显示结果顺序可以看到，主语句进行了在磁盘全表扫描，速度非常的慢，最终查找9行数据使用了将近6s时间。

(2) 有索引的子查询，执行过程则反过来，先执行从语句得到结果，在执行主语句从结果中匹配，从explain显示结果顺序可以看到，因此没有全表扫描，只需从结果中匹配一行就找到数据了，速度非常快，获取8行数据用了43ms时间。

## 23.2 连接查询实验

实验中使用的表和数据和子查询实验都一样，只是把子查询改为连接查询。

实验过程：

|  |
| --- |
| *# (1) 测试没有索引的连接查询(plan\_task的font\_color没有建索引)* **SELECT** *\** **FROM** plan\_task **INNER JOIN number ON** plan\_task.font\_color=**number**.**id**; *# 查询时间：9 rows retrieved starting from 1 in 5s 824ms (execution: 5s 796ms, fetching: 28ms)*  *# 使用explain分查询语句* **EXPLAIN SELECT** *\** **FROM** plan\_task **INNER JOIN number ON** plan\_task.font\_color=**number**.**id**;    *# (2) 测试有索引的连接查询(plan\_task的主键索引id)* **SELECT** *\** **FROM** plan\_task **INNER JOIN number ON** plan\_task.id=**number**.**id**; *# 查询时间：8 rows retrieved starting from 1 in 22ms (execution: 5ms, fetching: 17ms)*  *# 使用explain分析查询语句* **EXPLAIN SELECT** *\** **FROM** plan\_task **INNER JOIN number ON** plan\_task.id=**number**.**id**; |

实验结果：

没有索引和有索引两种情况的连接查询和子查询的效率是差不多的，没有哪个更好，实际场景中最好两种查询都用explain检查一下，确定使用哪一个查询。

# 24 表分区

表分区是把原来一张表按一定方式拆分为多个相同类型的表，同时原来一张表的数据文件也会拆分为多个数据文件，拆分方式可以按范围、散列点方式来分区。

表分区的好处：拆分后可以打开的线程数更多了，因为数据在不同的文件上，文件被锁的可能性会降低，一定程度上提高了读写速度。

例如单个.myd文件都达到了10G，读取数据时效率降低，通过表分区把10拆分为1G，可以提高效率。

对用户使用来说，表有没有分区是无影响的，增删改查操作还是一样。

**(1) 按范围来分区**

例如有一张用户表，按主键id的范围对用户表分区

|  |
| --- |
| **CREATE TABLE user**(  **id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT**, *# 主键* **name VARCHAR**(20) **NOT NULL DEFAULT ''**, *# 名称* **aid TINYINT UNSIGNED NOT NULL DEFAULT** 0 *# 地区代号* )**ENGINE** myisam **CHARSET** *utf8* **PARTITION BY RANGE** (**id**)( *# 按主键id范围对表进行分区* **PARTITION** u0 **VALUES LESS THAN** (10), *# 第1个分区范围* **PARTITION** u1 **VALUES LESS THAN** (20), *# 第2个分区范围* **PARTITION** u2 **VALUES LESS THAN** (**MAXVALUE**) *# 最后一个分区范围* ); |

也可以按时间按范围来分表，例如年、月

**(2) 按散列点对表进行分区**

例如有一张地区表和一张用户表，按地区代号对用户表进行分区

地区表：

|  |
| --- |
| **CREATE TABLE** area(  **aid INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT**, *# 主键* **addr VARCHAR**(10) **NOT NULL DEFAULT ''** *# 地区名称，不要为null* )**ENGINE** myisam **CHARSET** *utf8*; |

用户表：

|  |
| --- |
| **CREATE TABLE user**(  **id INT UNSIGNED PRIMARY KEY AUTO\_INCREMENT**, *# 主键* **name VARCHAR**(20) **NOT NULL DEFAULT ''**, *# 名称* **aid TINYINT UNSIGNED NOT NULL DEFAULT** 0 *# 地区代号* )**ENGINE** myisam **CHARSET** *utf8* **PARTITION BY LIST** (**aid**)( *# 按散列点(地区代号)对表进行分区* **PARTITION** a1 **VALUES IN** (1), *# 按地区代号分区* **PARTITION** a2 **VALUES IN** (2), *# 按地区代号分区* **PARTITION** a3 **VALUES IN** (3), *# 按地区代号分区* **PARTITION** a4 **VALUES IN** (4) *# 按地区代号分区* ); |

向数据库中插入大量数据的技巧

首先是建表时所有的列都不要声明索引(包括主键和unique)，原因时防止每插入一行数据都会使得索引排序，造成插入数据比较慢，特别是InnoDB引擎的表。插入完数据后，在声明主键和索引。

sql文件导入

|  |
| --- |
| use o2o;  set names utf8;  source D:/o2o.sql |

mysql周期性波动或故障解决方案

