mysql 登录命令：mysql -h ip -P 端口 -u 用户名 -p，如果是登录本金 ip 和端口可以省略，-P 大写的 P 后面跟上端口

mysql --version 或者mysql -V用于在未登录情况下，查看本机 mysql 版本：

select version();：登录情况下，查看链接的库版本：

### 显示所有数据库：****show databases;****

### 进入指定的库：****use 库名;****

### 显示当前库中所有的表：****show tables;****

**停止命令：net stop mysql**

**启动命令：net start mysql**

### 查看表的创建语句：****show create table 表名;****

### 查看表结构：****desc 表名;****

### 查看当前所在库：****select database();****

### 查看当前 mysql 支持的存储引擎：SHOW ENGINES;

### 查看系统变量及其值：SHOW VARIABLES;

### 查看某个系统变量：SHOW VARIABLES like '变量名';

**SQL 的语言分类**

* **DQL（Data Query Language）**：数据查询语言 select 相关语句
* **DML（Data Manipulate Language）**：数据操作语言 insert 、update、delete 语句
* **DDL（Data Define Languge）**：数据定义语言 create、drop、alter 语句
* **TCL（Transaction Control Language）**：事务控制语言 set autocommit=0、start transaction、savepoint、commit、rollback

**MySQL 的数据类型**

**主要包括以下五大类**

* **整数类型**：bit、bool、tinyint、smallint、mediumint、int、bigint
* **浮点数类型**：float、double、decimal
* **字符串类型**：char、varchar、tinyblob、blob、mediumblob、longblob、tinytext、text、mediumtext、longtext
* **日期类型**：Date、DateTime、TimeStamp、Time、Year
* 其他数据类型：暂不介绍，用的比较少。



上面[ ]包含的内容是可选的，默认是有符号类型的，无符号的需要在类型后面跟上unsigned

**类型(n)说明**

在开发中，我们会碰到有些定义整型的写法是 int(11)，这种写法个人感觉在开发过程中没有什么用途，不过还是来说一下，int(N)我们只需要记住两点：

* 无论 N 等于多少，int 永远占 4 个字节
* **N 表示的是显示宽度，不足的用 0 补足，超过的无视长度而直接显示整个数字，但这要整型设置了 unsigned zerofill 才有效**

int(n)中的 n 省略的时候，**宽度为对应类型无符号最大值的十进制的长度**，如 bigint 无符号最大值为 -1 = 18,446,744,073,709,551,615‬；长度是 20 位



float 数值类型用于表示单精度浮点数值，而 double 数值类型用于表示双精度浮点数值，float 和 double 都是浮点型，而 decimal 是定点型。

浮点型和定点型可以用类型名称后加（M，D）来表示，M 表示该值的总共长度，D 表示小数点后面的长度，M 和 D 又称为精度和标度。

float 和 double 在不指定精度时，默认会按照实际的精度来显示，而 DECIMAL 在不指定精度时，默认整数为 10，小数为 0。

|  |
| --- |
| mysql> create table **test5**(a **float**(5,2),b **double**(5,2),c **decimal**(5,2)); Query OK, 0 rows **affected** (0.01 sec)  mysql> insert into test5 **values** (1,1,1),(2.1,2.1,2.1),(3.123,3.123,3.123),(4.125,4.125,4.125),(5.115,5.115,5.115),(6.126,6.126,6.126),(7.116,7.116,7.116),(8.1151,8.1151,8.1151),(9.1251,9.1251,9.1251),(10.11501,10.11501,10.11501),(11.12501,11.12501,11.12501); Query OK, 7 rows affected, 5 warnings (0.01 sec) Records: 7  Duplicates: 0  Warnings: 5  mysql> select \* from test5; +-------+-------+-------+ | a     | b     | c     | +-------+-------+-------+ |  1.00 |  1.00 |  1.00 | |  2.10 |  2.10 |  2.10 | |  3.12 |  3.12 |  3.12 | |  4.12 |  4.12 |  4.13 | |  5.12 |  5.12 |  5.12 | |  6.13 |  6.13 |  6.13 | |  7.12 |  7.12 |  7.12 | |  8.12 |  8.12 |  8.12 | |  9.13 |  9.13 |  9.13 | | 10.12 | 10.12 | 10.12 | | 11.13 | 11.13 | 11.13 | +-------+-------+-------+ 11 rows in **set** (0.00 sec) |

**结果说明（注意看）：**

c 是 decimal 类型，认真看一下输入和输出，发现**decimal 采用的是四舍五入**

认真看一下a和b的输入和输出，尽然不是四舍五入，一脸闷逼，float 和 double 采用的是**四舍六入五成双**

decimal 插入的数据超过精度之后会触发警告。

**什么是四舍六入五成双？**

就是 5 以下舍弃 5 以上进位，如果需要处理数字为 5 的时候，需要看 5 后面是否还有不为 0 的任何数字，如果有，则直接进位，如果没有，需要看 5 前面的数字，若是奇数则进位，若是偶数则将 5 舍掉

我们将浮点类型的（M,D）精度和标度都去掉，看看效果：

|  |
| --- |
| mysql> create table **test6**(a **float**,b **double**,c decimal); Query OK, 0 rows **affected** (0.02 sec)  mysql> insert into test6 **values** (1,1,1),(1.234,1.234,1.4),(1.234,0.01,1.5); Query OK, 3 rows affected, 2 warnings (0.00 sec) Records: 3  Duplicates: 0  Warnings: 2  mysql> select \* from test6; +-------+-------+------+ | a     | b     | c    | +-------+-------+------+ |      1 |      1 |     1 | | 1.234 | 1.234 |    1 | | 1.234 |  0.01 |    2 | +-------+-------+------+ 3 rows in **set** (0.00 sec) |

a 和 b 的数据正确插入，而 c 被截断了

浮点数 float、double 如果不写精度和标度，则会按照实际显示

decimal 不写精度和标度，小数点后面的会进行四舍五入，并且插入时会有警告!

float、double会存在精度问题，decimal精度正常，比如银行对统计结果要求比较精准的建议使用decimal

日期类型

| **类型** | **字节大小** | **范围** | **格式** | **用途** |
| --- | --- | --- | --- | --- |
| DATE | 3 | 1000-01-01/9999-12-31 | YYYY-MM-DD | 日期值 |
| TIME | 3 | '-838:59:59'/'838:59:59' | HH:MM:SS | 时间值或持续时间 |
| YEAR | 1 | 1901/2155 | YYYY | 年份值 |
| DATETIME | 8 | 1000-01-01 00:00:00/9999-12-31 23:59:59 | YYYY-MM-DD HH:MM:SS | 混合日期和时间值 |
| TIMESTAMP | 4 | 1970-01-01 00:00:00/2038 结束时间是第 **2147483647** 秒，北京时间 **2038-1-19 11:14:07**，格林尼治时间 2038 年 1 月 19 日 凌晨 03:14:07 | YYYYMMDD HHMMSS | 混合日期和时间值，时间戳 |

**字符串类型**

| **类型** | **范围** | **存储所需字节** | **说明** |
| --- | --- | --- | --- |
| char(M) | [0,m]，m 的范围[0,2^8-1] | m | 定产字符串 |
| varchar(M) | [0,m]，m 的范围[0,2^16-1] | m | 0-65535 字节 |
| tinyblob | 0-255(2^8-1)字节 | L+1 | 不超过 255 个字符的二进制字符串 |
| blob | 0-65535(2^16-1)字节 | L+2 | 二进制形式的长文本数据 |
| mediumblob | 0-16777215(2^24-1)字节 | L+3 | 二进制形式的中等长度文本数据 |
| longblob | 0-4294967295(2^32-1)字节 | L+4 | 二进制形式的极大文本数据 |
| tinytext | 0-255(2^8-1)字节 | L+1 | 短文本字符串 |
| text | 0-65535(2^16-1)字节 | L+2 | 长文本数据 |
| mediumtext | 0-16777215(2^24-1)字节 | L+3 | 中等长度文本数据 |
| longtext | 0-4294967295(2^32-1)字节 | L+4 | 极大文本数据 |

char 类型占用固定长度，如果存放的数据为固定长度的建议使用 char 类型，如：手机号码、身份证等固定长度的信息。

表格中的 L 表示存储的数据本身占用的字节，L 以外所需的额外字节为存放该值的长度所需的字节数。

MySQL 通过存储值的内容及其长度来处理可变长度的值，这些额外的字节是无符号整数。

请注意，可变长类型的最大长度、此类型所需的额外字节数以及占用相同字节数的无符号整数之间的对应关系：

例如，MEDIUMBLOB 值可能最多  - 1 字节长并需要 3 个字节记录其长度，3 个字节的整数类型 MEDIUMINT 的最大无符号值为  - 1。

**mysql 类型和 java 类型对应关系**

| **MySQL Type Name** | **Return value ofGetColumnClassName** | **Returned as Java Class** |
| --- | --- | --- |
| BIT(1) (new in MySQL-5.0) | BIT | java.lang.Boolean |
| BIT( > 1) (new in MySQL-5.0) | BIT | byte[] |
| TINYINT | TINYINT | java.lang.Boolean if the configuration property tinyInt1isBit is set to true(the default) and the storage size is 1, orjava.lang.Integer if not. |
| BOOL, BOOLEAN | TINYINT | See TINYINT, above as these are aliases forTINYINT(1), currently. |
| SMALLINT[(M)][unsigned] | SMALLINT [UNSIGNED] | java.lang.Integer (regardless if UNSIGNED or not) |
| MEDIUMINT[(M)][unsigned] | MEDIUMINT [UNSIGNED] | java.lang.Integer, if UNSIGNEDjava.lang.Long |
| INT,INTEGER[(M)][unsigned] | INTEGER [UNSIGNED] | java.lang.Integer, if UNSIGNEDjava.lang.Long |
| BIGINT[(M)][unsigned] | BIGINT [UNSIGNED] | java.lang.Long, if UNSIGNEDjava.math.BigInteger |
| FLOAT[(M,D)] | FLOAT | java.lang.Float |
| DOUBLE[(M,B)] | DOUBLE | java.lang.Double |
| DECIMAL[(M[,D])] | DECIMAL | java.math.BigDecimal |
| DATE | DATE | java.sql.Date |
| DATETIME | DATETIME | java.sql.Timestamp |
| TIMESTAMP[(M)] | TIMESTAMP | java.sql.Timestamp |
| TIME | TIME | java.sql.Time |
| YEAR[(2|4)] | YEAR | If yearIsDateType configuration property is set to false, then the returned object type isjava.sql.Short. If set to true (the default) then an object of type java.sql.Date (with the date set to January 1st, at midnight). |
| CHAR(M) | CHAR | java.lang.String (unless the character set for the column is BINARY, then byte[]is returned. |
| VARCHAR(M) [BINARY] | VARCHAR | java.lang.String (unless the character set for the column is BINARY, then byte[]is returned. |
| BINARY(M) | BINARY | byte[] |
| VARBINARY(M) | VARBINARY | byte[] |
| TINYBLOB | TINYBLOB | byte[] |
| TINYTEXT | VARCHAR | java.lang.String |
| BLOB | BLOB | byte[] |
| TEXT | VARCHAR | java.lang.String |
| MEDIUMBLOB | MEDIUMBLOB | byte[] |
| MEDIUMTEXT | VARCHAR | java.lang.String |
| LONGBLOB | LONGBLOB | byte[] |
| LONGTEXT | VARCHAR | java.lang.String |
| ENUM('value1','value2',...) | CHAR | java.lang.String |
| SET('value1','value2',...) | CHAR | java.lang.String |

**数据类型选择的一些建议**

* **选小不选大**：一般情况下选择可以正确存储数据的最小数据类型，越小的数据类型通常更快，占用磁盘，内存和 CPU 缓存更小。
* **简单就好**：简单的数据类型的操作通常需要更少的 CPU 周期，例如：整型比字符操作代价要小得多，因为字符集和校对规则(排序规则)使字符比整型比较更加复杂。
* **尽量避免 NULL**：尽量制定列为 NOT NULL，除非真的需要 NULL 类型的值，有 NULL 的列值会使得索引、索引统计和值比较更加复杂。
* **浮点类型的建议统一选择 decimal**
* **记录时间的建议使用 int 或者 bigint 类型，将时间转换为时间戳格式，如将时间转换为秒、毫秒，进行存储，方便走索引**

**Mysql 权限验证分为 2 个阶段：**

1. 阶段 1：连接数据库，此时 mysql 会根据你的用户名及你的来源（ip 或者主机名称）判断是否有权限连接
2. 阶段 2：对 mysql 服务器发起请求操作，如 create table、select、delete、update、create index 等操作，此时 mysql 会判断你是否有权限操作这些指令

**权限生效时间**

用户及权限信息放在库名为 mysql 的库中，mysql 启动时，这些内容被读进内存并且从此时生效，所以如果通过直接操作这些表来修改用户及权限信息的，需要重启mysql或者执行flush privileges;才可以生效。

用户登录之后，mysql 会和当前用户之间创建一个连接，此时用户相关的权限信息都保存在这个连接中，存放在内存中，此时如果有其他地方修改了当前用户的权限，这些变更的权限会在下一次登录时才会生效。

用户信息在mysql.user表中，如下：

|  |
| --- |
| mysql> use mysql; Database changed mysql> select user,host from user; +---------------+--------------+ | user          | host         | +---------------+--------------+ | test4         | 127.0.0.%    | | test4         | 192.168.11.% | | mysql.session | localhost    | | mysql.sys     | localhost    | | root          | localhost    | | test2         | localhost    | +---------------+--------------+ 6 rows in **set** (0.00 sec) |

**创建用户**

**语法：**

create user 用户名[@主机名] [identified by '密码'];

1. 主机名默认值为%，表示这个用户可以从任何主机连接 mysql 服务器
2. 密码可以省略，表示无密码登录

**示例 1：不指定主机名**

不指定主机名时，表示这个用户可以从任何主机连接 mysql 服务器

mysql> create user test1;

用户创建之后可以在mysql库中通过 select user,host from user;查看到。

**其他示例**

create user 'test2'@'localhost' identified by '123';

说明：test2 的主机为 localhost 表示本机，此用户只能登陆本机的 mysql

create user 'test3'@% identified by '123';

说明：test3 可以从任何机器连接到 mysql 服务器

create user 'test4'@'192.168.11.%' identified by '123';

说明：test4 可以从 192.168.11 段的机器连接 mysql

**修改密码【3 种方式】**

**方式 1：通过管理员修改密码**

SET PASSWORD FOR '用户名'@'主机' = PASSWORD('密码');

**方式 2：create user 用户名[@主机名][identified by '密码'];**

set password = password('密码');

**方式 3：通过修改 mysql.user 表修改密码**

use mysql;  
update user set authentication\_string = password('321') where user = 'test1' and host = '%';  
flush privileges;

**通过表的方式修改之后，需要执行flush privileges;才能对用户生效。**

**5.7 中 user 表中的 authentication\_string 字段表示密码，老的一些版本中密码字段是 password。**

**给用户授权**

创建用户之后，需要给用户授权，才有意义。

**语法：**

grant privileges ON database.table TO 'username'[@'host'] [with grant option]

**grant 命令说明：**

* priveleges (权限列表)，可以是all，表示所有权限，也可以是select、update等权限，多个权限之间用逗号分开。
* ON 用来指定权限针对哪些库和表，格式为数据库.表名 ，点号前面用来指定数据库名，点号后面用来指定表名，\*.\* 表示所有数据库所有表。
* TO 表示将权限赋予某个用户, 格式为username@host，@前面为用户名，@后面接限制的主机，可以是 IP、IP 段、域名以及%，%表示任何地方。
* WITH GRANT OPTION 这个选项表示该用户可以将自己拥有的权限授权给别人。注意：经常有人在创建操作用户的时候不指定 WITH GRANT OPTION 选项导致后来该用户不能使用 GRANT 命令创建用户或者给其它用户授权。*备注：可以使用 GRANT 重复给用户添加权限，权限叠加，比如你先给用户添加一个 select 权限，然后又给用户添加一个 insert 权限，那么该用户就同时拥有了 select 和 insert 权限。*

**示例：**

grant all on \*.\* to 'test1'@‘%’;

说明：给 test1 授权可以操作所有库所有权限，相当于 dba

grant select on seata.\* to 'test1'@'%';

说明：test1 可以对 seata 库中所有的表执行 select

grant select,update on seata.\* to 'test1'@'%';

说明：test1 可以对 seata 库中所有的表执行 select、update

grant **select**(user,host) on mysql.user to 'test1'@'localhost';

说明：test1 用户只能查询 mysql.user 表的 user,host 字段

**查看用户有哪些权限**

**show grants for '用户名'[@'主机']**

主机可以省略，默认值为%，示例：

|  |
| --- |
| mysql> show grants **for** 'test1'@'localhost'; +--------------------------------------------------------------------+ | Grants **for** test1@localhost                                         | +--------------------------------------------------------------------+ | GRANT USAGE ON \*.\* TO 'test1'@'localhost'                          | | GRANT **SELECT** (host, user) ON `mysql`.`user` TO 'test1'@'localhost' | +--------------------------------------------------------------------+ 2 rows in **set** (0.00 sec) |

**show grants;**

查看当前用户的权限，如：

|  |
| --- |
| mysql> show grants; +---------------------------------------------------------------------+ | Grants **for** root@localhost                                           | +---------------------------------------------------------------------+ | GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'localhost' WITH GRANT OPTION | | GRANT ALL PRIVILEGES ON `test`.\* TO 'root'@'localhost'              | | GRANT DELETE ON `seata`.\* TO 'root'@'localhost'                     | | GRANT PROXY ON ''@'' TO 'root'@'localhost' WITH GRANT OPTION        | +---------------------------------------------------------------------+ 4 rows in **set** (0.00 sec) |

**撤销用户的权限**

**语法**

revoke privileges ON database.table FROM '用户名'[@'主机'];

可以先通过show grants命令查询一下用户对于的权限，然后使用revoke命令撤销用户对应的权限，示例：

|  |
| --- |
| mysql> show grants **for** 'test1'@'localhost'; +--------------------------------------------------------------------+ | Grants **for** test1@localhost                                         | +--------------------------------------------------------------------+ | GRANT USAGE ON \*.\* TO 'test1'@'localhost'                          | | GRANT **SELECT** (host, user) ON `mysql`.`user` TO 'test1'@'localhost' | +--------------------------------------------------------------------+ 2 rows in **set** (0.00 sec)  mysql> revoke **select**(host) on mysql.user from test1@localhost; Query OK, 0 rows **affected** (0.00 sec)  mysql> show grants **for** 'test1'@'localhost'; +--------------------------------------------------------------+ | Grants **for** test1@localhost                                   | +--------------------------------------------------------------+ | GRANT USAGE ON \*.\* TO 'test1'@'localhost'                    | | GRANT **SELECT** (user) ON `mysql`.`user` TO 'test1'@'localhost' | +--------------------------------------------------------------+ 2 rows in **set** (0.00 sec) |

上面我们先通过grants命令查看 test1 的权限，然后调用 revoke 命令撤销对mysql.user表host字段的查询权限，最后又通过 grants 命令查看了 test1 的权限，和预期结果一致。  
**删除用户【2 种方式】**

**方式 1：**

**drop user '用户名'[@‘主机’]**，示例：

mysql> drop user test1@localhost;  
Query OK, 0 rows **affected** (0.00 sec)

drop 的方式删除用户之后，用户下次登录就会起效。

**方式 2：**

通过删除 mysql.user 表数据的方式删除，如下：

delete from user where user='用户名' and host='主机';  
flush privileges;

注意通过表的方式删除的，需要调用flush privileges;刷新权限信息（权限启动的时候在内存中保存着，通过表的方式修改之后需要刷新一下）。

**授权原则说明**

* 只授予能满足需要的最小权限，防止用户干坏事，比如用户只是需要查询，那就只给 select 权限就可以了，不要给用户赋予 update、insert 或者 delete 权限
* 创建用户的时候限制用户的登录主机，一般是限制成指定 IP 或者内网 IP 段
* 初始化数据库的时候删除没有密码的用户，安装完数据库的时候会自动创建一些用户，这些用户默认没有密码
* 为每个用户设置满足密码复杂度的密码
* 定期清理不需要的用户，回收权限或者删除用户

**总结**

1. 通过命令的方式操作用户和权限不需要刷新，下次登录自动生效
2. 通过操作 mysql 库中表的方式修改、用户信息，需要调用flush privileges;刷新一下，下次登录自动生效
3. mysql 识别用户身份的方式是：用户名+主机
4. 本文中讲到的一些指令中带主机的，主机都可以省略，默认值为%，表示所有机器
5. mysql 中用户和权限的信息在库名为 mysql 的库中



**DDL 介绍**

DDL：Data Define Language 数据定义语言，主要用来对数据库、表进行一些管理操作。

如：建库、删库、建表、修改表、删除表、对列的增删改等等。

文中涉及到的语法用[]包含的内容属于可选项，下面做详细说明。

**default value**：为该字段设置默认值，默认值为 value

|  |
| --- |
| mysql> create table **test2**(     ->   a **int** not **null** comment '字段a',     ->   b **int** not **null** **default** 0 comment '字段b'     -> ); |

**primary key**：标识该字段为该表的主键，可以唯一的标识记录，插入重复的会报错

两种写法，如下：

方式 1：跟在列后，如下：

|  |
| --- |
| mysql> create table **test3**(      ->   a **int** not **null** comment '字段a' primary key     -> ); |

方式 2：在所有列定义之后定义，如下：

|  |
| --- |
| mysql> create table **test4**(     ->   a **int** not **null** comment '字段a',     ->   b **int** not **null** **default** 0 comment '字段b',     ->   primary key(a)     -> ); |

方式 2 支持多字段作为主键，多个之间用逗号隔开，语法：primary key(字段 1,字段 2,字段 n)，

|  |
| --- |
| mysql> create table **test7**(     ->    a **int** not **null** comment '字段a',     ->    b **int** not **null** comment '字段b',     ->   PRIMARY KEY (a,b)     ->  ); |

**foreign key**：为表中的字段设置外键

**语法：foreign key(当前表的列名) references 引用的外键表(外键表中字段名称)**

**unique key(uq)**：标识该字段的值是唯一的

支持一个到多个字段，插入重复的值会报违反唯一约束，会插入失败。

定义有 2 种方式。

方式 1：跟在字段后，如下：

mysql> create table **test8**(  
    ->    a **int** not **null** comment '字段a' unique key  
    ->  );

方式 2：所有列定义之后定义，如下：

mysql> create table **test9**(  
    ->    a **int** not **null** comment '字段a',  
    ->   unique key(a)  
    ->  );

方式 2 支持多字段，多个之间用逗号隔开，语法：primary key(字段 1,字段 2,字段 n)，示例：

mysql> create table **test10**(  
    ->   a **int** not **null** comment '字段a',  
    ->   b **int** not **null** comment '字段b',  
    ->   unique key(a,b)  
    -> );

**auto\_increment**：标识该字段的值自动增长（整数类型，而且为主键）

mysql> create table **test11**(  
    ->   a **int** not **null** AUTO\_INCREMENT PRIMARY KEY comment '字段a',  
  ->   b **int** not **null** comment '字段b'  
    -> );

**删除表**

drop table [**if** exists] 表名;

**修改表名**

alter table 表名 rename [to] 新表名;

**表设置备注**

alter table 表名 comment '备注信息';

**复制表**

**只复制表结构**

create table 表名 like 被复制的表名;

**复制表结构+数据**

create table 表名 [as] select 字段,... from 被复制的表 [where 条件];

**表中列的管理**

**添加列**

alter table 表名 add column 列名 类型 [列约束];

alter table test14 add column b **int** not **null** **default** 0 comment '字段b';

**修改列**

alter table 表名 modify column 列名 新类型 [约束];  
或者  
alter table 表名 change column 列名 新列名 新类型 [约束];

alter table test14 change column c d **varchar**(10) not **null** **default** '' comment '字段d';

**2 种方式区别：modify 不能修改列名，change 可以修改列名**

**删除列**

alter table 表名 drop column 列名;

alter table test14 drop column d;

**DML**

DML(Data Manipulation Language)数据操作语言，以 INSERT、UPDATE、DELETE 三种指令为核心，分别代表插入、更新与删除，**是必须要掌握的指令**，DML 和 SQL 中的 select 熟称 CRUD（增删改查）。

文中涉及到的语法用[]包含的内容属于可选项，下面做详细说明。

**插入操作**

**插入单行 2 种方式**

**方式 1**

insert into 表名[(字段,字段)] values (值,值);

**说明：**

值和字段需要一一对应

如果是字符型或日期类型，值需要用单引号引起来；如果是数值类型，不需要用单引号

字段和值的个数必须一致，位置对应

字段如果不能为空，则必须插入值

可以为空的字段可以不用插入值，但需要注意：字段和值都不写；或字段写上，值用 null 代替

表名后面的字段可以省略不写，此时表示所有字段，顺序和表中字段顺序一致。

**方式 2**

insert into 表名 set 字段 = 值,字段 = 值;

方式 2 不常见，建议使用方式 1

**批量插入 2 种方式**

**方式 1**

insert into 表名 [(字段,字段)] values (值,值),(值,值),(值,值);

**方式 2**

insert into 表 [(字段,字段)]  
数据来源select语句;

**说明：**

数据来源 select 语句可以有很多种写法，需要注意：select 返回的结果和插入数据的字段数量、顺序、类型需要一致。

关于 select 的写法后面文章会详细介绍。

**使用 delete 删除**

**delete 单表删除**

delete [别名] from 表名 [[as] 别名] [where条件];

注意：

如果无别名的时候，表名就是别名

如果有别名，delete 后面必须写别名

如果没有别名，delete 后面的别名可以省略不写。

delete t2,t1 from test1 t1,test2 t2 where t1.a=t2.c2;

同时对 2 个表进行删除，条件是 test.a=test.c2 的记录

**使用 truncate 删除**

**语法**

truncate 表名;

**drop，truncate，delete 区别**

* drop (删除表)：删除内容和定义，释放空间，简单来说就是**把整个表去掉**，以后要新增数据是不可能的，除非新增一个表。

drop 语句将删除表的结构被依赖的约束（constrain），触发器（trigger）索引（index），依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

如果要删除表定义及其数据，请使用 drop table 语句。

* truncate (清空表中的数据)：删除内容、释放空间但不删除定义(**保留表的数据结构**)，与 drop 不同的是，只是清空表数据而已。

注意：truncate 不能删除具体行数据，要删就要把整个表清空了。

* delete (删除表中的数据)：delete 语句用于**删除表中的行**。delete 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存，以便进行进行回滚操作。

**truncate 与不带 where 的 delete ：只删除数据，而不删除表的结构（定义）**

**truncate table 删除表中的所有行，但表结构及其列、约束、索引等保持不变。**

对于有 foreign key 约束引用的表，不能使用 truncate table ，而应使用不带 where 子句的 delete 语句。由于 truncate table 记录在日志中，所以它不能激活触发器。

delete 语句是数据库操作语言(dml)，这个操作会放到 rollback segement 中，**事务提交之后才生效**；如果有相应的 trigger，执行的时候将被触发。

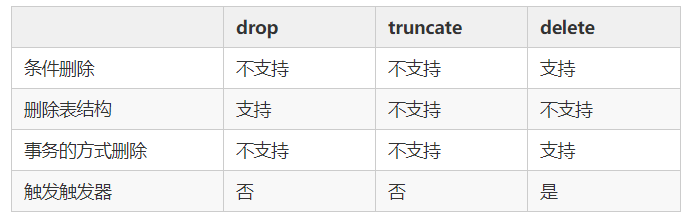
**truncate、drop 是数据库定义语言(ddl)，操作立即生效**，原数据不放到 rollback segment 中，不能回滚，操作不触发 trigger。

**如果有自增列，truncate 方式删除之后，自增列的值会被初始化，delete 方式要分情况（如果数据库被重启了，自增列值也会被初始化，数据库未被重启，则不变）**

**如果要删除表定义及其数据，请使用 drop table 语句**

**安全性：小心使用 drop 和 truncate，尤其没有备份的时候，否则哭都来不及**

**删除速度，一般来说: drop> truncate > delete**



**DQL**

DQL(Data QueryLanguage)：数据查询语言，通俗点讲就是从数据库获取数据的，按照 DQL 的语法给数据库发送一条指令，数据库将按需求返回数据。

DQL 分多篇来说，本文属于第 1 篇。

**查询函数**

select 函数;

如：

|  |
| --- |
| mysql> select **mod**(10,4),**isnull**(**null**),**ifnull**(1,'第一个参数为空返回这个值，否则返回第一个参数'); +-----------+--------------+-----------------------------------------------------+--------------------------------------------------------------------------------+ | mod(10,4) | isnull(**null**) | ifnull(1,'第一个参数为空返回这个值，否则返回第一个参数')                       | +-----------+--------------+-----------------------------------------------------+--------------------------------------------------------------------------------+ |         2 |            1     | 1                                                                              | +-----------+--------------+-----------------------------------------------------+--------------------------------------------------------------------------------+ 1 row in **set** (0.00 sec) |

说明一下：

mod 函数，对两个参数取模运算。

isnull 函数，判断参数是否为空，若为空返回 1，否则返回 0。

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。

**列别名**

在创建数据表时，一般都会使用英文单词或英文单词缩写来设置字段名，在查询时列名都会以英文的形式显示，这样会给用户查看数据带来不便，这种情况可以使用别名来代替英文列名，增强阅读性。

**语法：**

select 列 [as] 别名 from 表;

mysql> select a as 列1,b as 列2 from test1;

mysql> select a '列1',b '列2' from test1;

**总结**

* **建议别名前面跟上 as 关键字**
* **查询数据的时候，避免使用 select \*，建议需要什么字段写什么字段**

**like（模糊查询）**

select 列名 from 表名 where 列 like pattern;

pattern 中可以包含通配符，有以下通配符：

%：表示匹配任意一个或多个字符

\_：表示匹配任意一个字符。

**BETWEEN AND(区间查询)**

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。

selec 列名 from 表名 where 列名 between 值1 and 值2;

返回对应的列的值在[值 1,值 2]区间中的记录

使用 between and 可以提高语句的简洁度

两个临界值不要调换位置，只能是大于等于左边的值，并且小于等于右边的值。

**IN 查询**

IN 操作符允许我们在 WHERE 子句中规定多个值。

select 列名 from 表名 where 字段 in (值1,值2,值3,值4);

**NOT IN 查询**

not in 和 in 刚好相反，in 是列表中被匹配的都会被返回，NOT IN 是和列表中都不匹配的会被返回。

select 列名 from 表名 where 字段 not **in** (值1,值2,值3,值4);

**NULL 存在的坑**

**查询运算符(>,<,=)、like、between and、in、not in 对 NULL 值查询不起效。**

**那 NULL 如何查询呢？继续向下看**

**IS NULL/IS NOT NULL（NULL 值专用查询）**

上面介绍的各种运算符对 NULL 值均不起效，mysql 为我们提供了查询空值的语法：IS NULL、IS NOT NULL。

**IS NULL（返回值为空的记录）**

select 列名 from 表名 where 列 is **null**;

查询指定的列的值为 NULL 的记录。

**IS NOT NULL（返回值不为空的记录）**

select 列名 from 表名 where 列 is not **null**;

查询指定的列的值不为 NULL 的记录。

**<=>（安全等于）**

<=>：既可以判断 NULL 值，又可以判断普通的数值，可读性较低，用得较少

mysql> select \* from test8 t where t.a<=>**null**;  
+------+------+  
| a     | b    |  
+------+------+  
| NULL | b    |  
| NULL | NULL |  
+------+------+  
2 rows in **set** (0.00 sec)

**经典面试题**

下面的 2 个 sql 查询结果一样么？

select \* from students;  
select \* from students where name like '%';

结果分 2 种情况：

当 name 没有 NULL 值时，返回的结果一样。

当 name 有 NULL 值时，第 2 个 sql 查询不出 name 为 NULL 的记录。

**总结**

* **like 中的%可以匹配一个到多个任意的字符，\_可以匹配任意一个字符**
* **空值查询需要使用 IS NULL 或者 IS NOT NULL，其他查询运算符对 NULL 值无效**
* **建议创建表的时候，尽量设置表的字段不能为空，给字段设置一个默认值**
* **<=>（安全等于）玩玩可以，建议少使用**

**排序查询（order by）**

电商中：我们想查看今天所有成交的订单，按照交易额从高到低排序，此时我们可以使用数据库中的排序功能来完成。

**排序语法：**

select 字段名 from 表名 order by 字段1 [asc|desc],字段2 [asc|desc];

需要排序的字段跟在order by之后；

asc|desc 表示排序的规则，asc：升序，desc：降序，默认为 asc；

支持多个字段进行排序，多字段排序之间用逗号隔开。

**多字段排序**

比如学生表，先按学生年龄降序，年龄相同时，再按学号升序，如下：

|  |
| --- |
| mysql> select \* from stu order by age desc,id asc; +------+-----+---------------+ | id   | age | name          | +------+-----+---------------+ | 1004 |  20 | 张国荣        | | 1005 |  20 | 刘德华        | | 1010 |  19 | 梁朝伟        | | 1001 |  18 | 路人甲Java    | | 1003 |  18 | 张学友        | +------+-----+---------------+ 5 rows in **set** (0.00 sec) |

**where 之后进行排序**

需求：查询订单金额>=100 的，按照订单金额降序排序，显示 2 列数据，列头：订单编号、订单金额，如下：

|  |
| --- |
| mysql> select a.id 订单编号,a.price 订单金额 from t\_order a where a.price>=100 order by a.price desc; +--------------+--------------+ | 订单编号     | 订单金额     | +--------------+--------------+ |            3 |       500.00 | |            4 |       300.00 | |            6 |       200.50 | |            2 |       100.68 | +--------------+--------------+ 4 rows in **set** (0.00 sec) |

**limit 介绍**

limit 用来限制 select 查询返回的行数，常用于分页等操作。

**语法：**

select 列 from 表 limit [offset,] count;

说明：

offset：表示偏移量，通俗点讲就是跳过多少行，offset 可以省略，默认为 0，表示跳过 0 行；范围：[0,+∞)。

count：跳过 offset 行之后开始取数据，取 count 行记录；范围：[0,+∞)。

limit 中 offset 和 count 的值不能用表达式。

下面我们列一些常用的示例来加深理解。

**获取前 n 行记录**

select 列 from 表 limit 0,n;  
或者  
select 列 from 表 limit n;

**获取排名第 n 到 m 的记录**

我们需要先跳过 n-1 条记录，然后取 m-n+1 条记录，如下：

select 列 from 表 limit n-1,m-n+1;

**limit 中不能使用表达式，limit 后面只能够跟明确的数字。**

**当排序过程中存在相同的值时，没有其他排序规则时，mysql 懵逼了，不知道怎么排序了。排序中存在相同的值时，需要再指定一个排序规则，通过这种排序规则不存在二义性**

mysql> select \* from test1 order by b asc,a desc limit 0,2;

**总结**

* order by ... [asc|desc]用于对查询结果排序，asc：升序，desc：降序，asc|desc 可以省略，默认为 asc
* limit 用来限制查询结果返回的行数，有 2 个参数（offset，count），offset：表示跳过多少行，count：表示跳过 offset 行之后取 count 行
* limit 中 offset 可以省略，默认值为 0
* limit 中 offset 和 count 都必须大于等于 0
* limit 中 offset 和 count 的值不能用表达式
* 分页排序时，排序不要有二义性，二义性情况下可能会导致分页结果乱序，可以在后面追加一个主键排序

**分组查询**

**语法：**

|  |
| --- |
| SELECT column, group\_function,... FROM table [WHERE condition] GROUP BY group\_by\_expression [HAVING group\_condition]; |

说明：

group\_function：聚合函数。

group\_by\_expression：分组表达式，多个之间用逗号隔开。

group\_condition：分组之后对数据进行过滤。

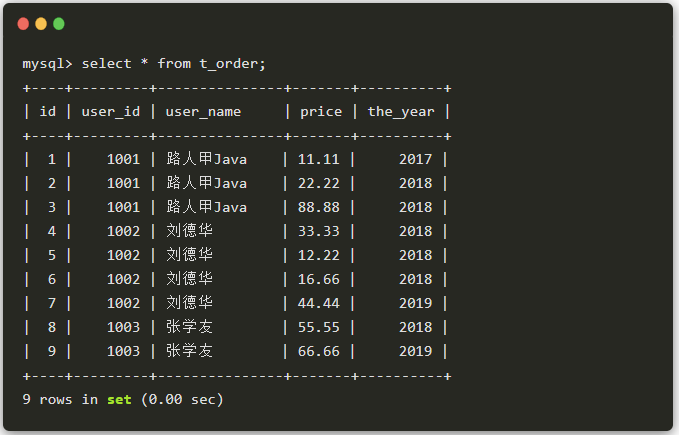
分组中，select 后面只能有两种类型的列：

1. 出现在 group by 后的列
2. 或者使用聚合函数的列

**聚合函数**

| **函数名称** | **作用** |
| --- | --- |
| max | 查询指定列的最大值 |
| min | 查询指定列的最小值 |
| count | 统计查询结果的行数 |
| sum | 求和，返回指定列的总和 |
| avg | 求平均值，返回指定列数据的平均值 |

分组时，可以使用使用上面的聚合函数。



**单字段分组**

\*\*需求：\*\*查询每个用户下单数量，输出：用户 id、下单数量，如下：

mysql> SELECT  
            user\_id 用户id, COUNT(id) 下单数量  
        FROM  
            t\_order  
        GROUP BY user\_id;

**多字段分组**

\*\*需求：\*\*查询每个用户每年下单数量，输出字段：用户 id、年份、下单数量，如下：

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;

**分组前筛选数据**

分组前对数据进行筛选，使用 where 关键字

\*\*需求：\*\*需要查询 2018 年每个用户下单数量，输出：用户 id、下单数量，如下：

mysql> 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;

**分组后筛选数据**

分组后对数据筛选，使用 having 关键字

\*\*需求：\*\*查询 2018 年订单数量大于 1 的用户，输出：用户 id，下单数量，如下：

**方式 1：**

mysql> 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;

**方式 2：**

mysql> 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 下单数量>=2;

**where 和 having 的区别**

where 是在分组（聚合）前对记录进行筛选，而 having 是在分组结束后的结果里筛选，最后返回整个 sql 的查询结果。

可以把 having 理解为两级查询，即含 having 的查询操作先获得不含 having 子句时的 sql 查询结果表，然后在这个结果表上使用 having 条件筛选出符合的记录，最后返回这些记录，因此，having 后是可以跟聚合函数的，并且这个聚集函数不必与 select 后面的聚集函数相同。

**分组后排序**

**需求**：获取每个用户最大金额，然后按照最大金额倒序，输出：用户 id，最大金额，如下：

mysql> SELECT  
          user\_id 用户id, max(price) 最大金额  
        FROM  
          t\_order t  
        GROUP BY user\_id  
        ORDER BY 最大金额 desc;

**where & group by & having & order by & limit 一起协作**

where、group by、having、order by、limit 这些关键字一起使用时，先后顺序有明确的限制，语法如下：

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;

**注意：**

**写法上面必须按照上面的顺序来写。**

**示例：**

\*\*需求：\*\*查询出 2018 年，下单数量大于等于 2 的，按照下单数量降序排序，最后只输出第 1 条记录，显示：用户 id，下单数量，如下：

mysql> 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  
        ORDER BY 下单数量 DESC  
        LIMIT 1;

**mysql 分组中的坑**

本文开头有介绍，分组中 select 后面的列只能有 2 种：

1. 出现在 group by 后面的列
2. 使用聚合函数的列

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。

**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。**

**总结**

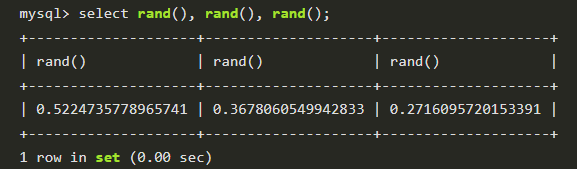
1. 在写分组查询的时候，最好按照标准的规范来写，**select 后面出现的列必须在 group by 中或者必须使用聚合函数**。
2. select 语法顺序：select、from、where、group by、having、order by、limit，顺序不能搞错了，否则报错。

**MySQL 数值型函数**

| **函数名称** | **作 用** |
| --- | --- |
| abs | 求绝对值 |
| sqrt | 求二次方根 |
| mod | 求余数 |
| ceil 和 ceiling | 两个函数功能相同，都是返回不小于参数的最小整数，即向上取整 |
| floor | 向下取整，返回值转化为一个 BIGINT |
| rand | 生成一个 0~1 之间的随机数，传入整数参数是，用来产生重复序列 |
| round | 对所传参数进行四舍五入 |
| sign | 返回参数的符号 |
| pow 和 power | 两个函数的功能相同，都是所传参数的次方的结果值 |
| sin | 求正弦值 |
| asin | 求反正弦值，与函数 SIN 互为反函数 |
| cos | 求余弦值 |
| acos | 求反余弦值，与函数 COS 互为反函数 |
| tan | 求正切值 |
| atan | 求反正切值，与函数 TAN 互为反函数 |
| cot | 求余切值 |

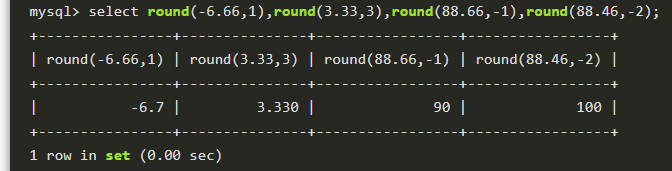
**rand:生成一个随机数**

生成一个 0~1 之间的随机数，传入整数参数是，用来产生重复序列



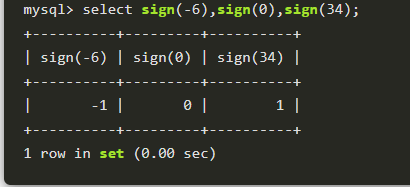
**round:四舍五入函数**

返回最接近于参数 x 的整数；ROUND(x,y) 函数对参数 x 进行四舍五入的操作，返回值保留小数点后面指定的 y 位。



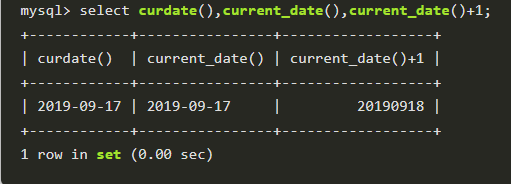
**sign:返回参数的符号**

返回参数的符号，x 的值为负、零和正时返回结果依次为 -1、0 和 1。



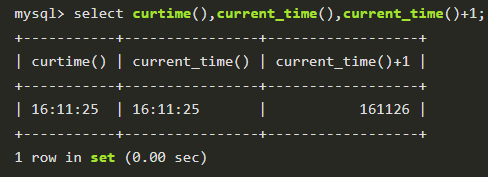
**curdate 和 current\_date:两个函数作用相同，返回当前系统的日期值**

CURDATE() 和 CURRENT\_DATE() 函数的作用相同，将当前日期按照“YYYY-MM-DD”或“YYYYMMDD”格式的值返回，具体格式根据函数用在字符串或数字语境中而定，返回的date类型。



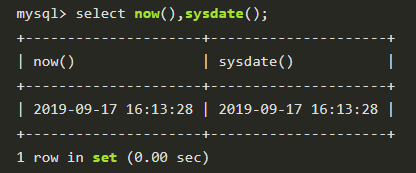
**curtime 和 current\_time:获取系统当前时间**

CURTIME() 和 CURRENT\_TIME() 函数的作用相同，将当前时间以“HH：MM：SS”或“HHMMSS”格式返回，具体格式根据函数用在字符串或数字语境中而定，返回time类型。



**now 和 sysdate:获取当前时间日期**

NOW() 和 SYSDATE() 函数的作用相同，都是返回当前日期和时间值，格式为“YYYY-MM-DD HH：MM：SS”或“YYYYMMDDHHMMSS”，具体格式根据函数用在字符串或数字语境中而定，返回datetime类型。



**unix\_timestamp:获取 UNIX 时间戳**

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

|  |
| --- |
| mysql> select **unix\_timestamp**(),**unix\_timestamp**(now()),**now**(),**unix\_timestamp**('2019-09-17 12:00:00'); +------------------+-----------------------+---------------------+---------------------------------------+ | unix\_timestamp() | unix\_timestamp(now()) | now()               | unix\_timestamp('2019-09-17 12:00:00') | +------------------+-----------------------+---------------------+---------------------------------------+ |       1568710893 |            1568710893 | 2019-09-17 17:01:33 |                            1568692800 | +------------------+-----------------------+---------------------+---------------------------------------+ 1 row in **set** (0.00 sec) |

**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

|  |
| --- |
| mysql> select **from\_unixtime**(1568710866),**from\_unixtime**(1568710866,'%Y-%m-%d %H:%h:%s'); +---------------------------+-----------------------------------------------+ | from\_unixtime(1568710866) | from\_unixtime(1568710866,'%Y-%m-%d %H:%h:%s') | +---------------------------+-----------------------------------------------+ | 2019-09-17 17:01:06       | 2019-09-17 17:05:06                           | +---------------------------+-----------------------------------------------+ 1 row in **set** (0.00 sec) |

**MySQL，深入了解连接查询及原理**

**外连接**

外连接涉及到 2 个表，分为：主表和从表，要查询的信息主要来自于哪个表，谁就是主表。

外连接查询结果为主表中所有记录。如果从表中有和它匹配的，则显示匹配的值，这部分相当于内连接查询出来的结果；如果从表中没有和它匹配的，则显示 null。

**内连接**

**语法：**

select 字段 from 表1 inner join 表2 on 连接条件;  
或  
select 字段 from 表1 join 表2 on 连接条件;  
或  
select 字段 from 表1, 表2 [where 关联条件];

内连接相当于在笛卡尔积的基础上加上了连接的条件。

当没有连接条件的时候，内连接上升为笛卡尔积。

**左外链接：使用 left join 关键字，left join 左边的是主表。**

**右外连接：使用 right join 关键字，right join 右边的是主表。**

**子查询**

出现在 select 语句中的 select 语句，称为子查询或内查询。

外部的 select 查询语句，称为主查询或外查询。

**子查询分类**

**按结果集的行列数不同分为 4 种**

* 标量子查询（结果集只有一行一列）
* 列子查询（结果集只有一列多行）
* 行子查询（结果集有一行多列）
* 表子查询（结果集一般为多行多列）

**按子查询出现在主查询中的不同位置分**

* **select 后面**：仅仅支持标量子查询。
* **from 后面**：支持表子查询。
* **where 或 having 后面**：支持标量子查询（单列单行）、列子查询（单列多行）、行子查询（多列多行）
* **exists 后面（即相关子查询）**：表子查询（多行、多列）

**where 或 having 后面，可以使用**

1. 标量子查询（单行单列行子查询）
2. 列子查询（单列多行子查询）
3. 行子查询（一行多列）

**特点**

1. 子查询放在小括号内。
2. 子查询一般放在条件的右侧。
3. 标量子查询，一般搭配着单行单列操作符使用 **>、<、>=、<=、=、<>、!=**
4. 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);

1. 子查询的执行优先于主查询执行，因为主查询的条件用到了子查询的结果。

**mysql 中的 in、any、some、all**

in，any，some，all 分别是子查询关键词之一。

**in**：in 常用于 where 表达式中，其作用是查询某个范围内的数据

**any 和 some 一样**：可以与=、>、>=、<、<=、<>结合起来使用，分别表示等于、大于、大于等于、小于、小于等于、不等于其中的任何一个数据即满足要求。

**all**：可以与=、>、>=、<、<=、<>结合是来使用，分别表示等于、大于、大于等于、小于、小于等于、不等于其中的所有数据。

下文中会经常用到这些关键字。

**子查询+分组函数，示例**

查询最低工资大于 50 号部门最低工资的部门 id 和其最低工资【having】

|  |
| --- |
| /\*查询最低工资大于50号部门最低工资的部门id和其最低工资【having】\*/ /\*①查询50号部门的最低工资\*/ SELECT **min**(salary) FROM employees WHERE department\_id = 50; /\*②查询每个部门的最低工资\*/ SELECT   **min**(salary),   department\_id FROM employees GROUP BY department\_id; /\*③在②的基础上筛选，满足min(salary)>①\*/ SELECT   **min**(a.salary) minsalary,   department\_id FROM employees a GROUP BY a.department\_id HAVING **min**(a.salary) > (SELECT min(salary)                         FROM employees                         WHERE department\_id = 50); |

**列子查询(子查询结果集一列多行)**

列子查询需要搭配多行操作符使用：in(not in)、any/some、all。

为了提升效率，最好去重一下**distinct**关键字。

返回 location\_id 是 1400 或 1700 的部门中的所有员工姓名

|  |
| --- |
| SELECT a.last\_name FROM employees a WHERE a.department\_id = ANY (SELECT DISTINCT department\_id                              FROM departments                              WHERE location\_id **IN** (1400, 1700)); |

返回其他工种中比 job\_id 为'IT\_PROG'工种任意工资低的员工的员工号、姓名、job\_id、salary

|  |
| --- |
| SELECT   last\_name,   employee\_id,   job\_id,   salary FROM employees WHERE salary < ANY (SELECT DISTINCT salary                     FROM employees                     WHERE job\_id = 'IT\_PROG') AND job\_id != 'IT\_PROG';  /\*或者\*/ SELECT   last\_name,   employee\_id,   job\_id,   salary FROM employees WHERE salary < (SELECT **max**(salary)                 FROM employees                 WHERE job\_id = 'IT\_PROG') AND job\_id != 'IT\_PROG'; |

**行子查询（子查询结果集一行多列）**

**示例**

查询员工编号最小并且工资最高的员工信息，3 种方式。

|  |
| --- |
| /\*查询员工编号最小并且工资最高的员工信息\*/ /\*①查询最小的员工编号\*/ SELECT **min**(employee\_id) FROM employees; /\*②查询最高工资\*/ SELECT **max**(salary) FROM employees; /\*③方式1：查询员工信息\*/ SELECT \* FROM employees a WHERE a.employee\_id = (SELECT **min**(employee\_id)                        FROM employees)       AND salary = (SELECT **max**(salary)                     FROM employees);  /\*方式2\*/ SELECT \* FROM employees a **WHERE** (a.employee\_id, a.salary) = (SELECT                                      **min**(employee\_id),                                      **max**(salary)                                    FROM employees); /\*方式3\*/ SELECT \* FROM employees a **WHERE** (a.employee\_id, a.salary) **in** (SELECT                                      min(employee\_id),                                      **max**(salary)                                    FROM employees); |

**exists 后面（也叫做相关子查询）**

1. 语法：exists(完整的查询语句)。
2. exists 查询结果：1 或 0，exists 查询的结果用来判断子查询的结果集中是否有值。
3. 一般来说，能用 exists 的子查询，绝对都能用 in 代替，所以 exists 用的少。
4. 和前面的查询不同，这**先执行主查询，然后主查询查询的结果**，再根据子查询进行过滤，子查询中涉及到主查询中用到的字段，所以叫相关子查询。

查询所有员工的部门名称

|  |
| --- |
| /\*查询所有员工部门名\*/ SELECT department\_name FROM departments a WHERE **exists**(SELECT 1              FROM employees b              WHERE a.department\_id = b.department\_id);  /\*使用in实现\*/ SELECT department\_name FROM departments a WHERE a.department\_id **IN** (SELECT department\_id                           FROM employees); |

使用**not in**的方式查询没有员工的部门，如下：

|  |
| --- |
| mysql> SELECT \*     -> FROM departments a     -> WHERE a.department\_id NOT **IN** (SELECT department\_id     ->                               FROM employees b); Empty **set** (0.00 sec) |

not in 的情况下，子查询中列的值为 NULL 的时候，外查询的结果为空。

**任何值和 NULL 使用运算符（>、<、>=、<=、!=、<>）或者（in、not in、any/some、all）比较时，返回值都为 NULL，NULL 作为布尔值的时候，不为 1 也不为 0。**

**IN、NOT IN 和 NULL 比较**

**IN 和 NULL 比较**

|  |
| --- |
| mysql> select \* from test1; +------+------+ | a    | b    | +------+------+ |    1 |    1 | |    1 | NULL | | NULL | NULL | +------+------+ 3 rows in **set** (0.00 sec)  mysql> select \* from test1 where a **in** (**null**); Empty **set** (0.00 sec)  mysql> select \* from test1 where a **in** (**null**,1); +------+------+ | a    | b    | +------+------+ |    1 |    1 | |    1 | NULL | +------+------+ 2 rows in **set** (0.00 sec) |

**结论：当 IN 和 NULL 比较时，无法查询出为 NULL 的记录。**

**NOT IN 和 NULL 比较**

|  |
| --- |
| mysql> select \* from test1 where a not **in** (1); Empty **set** (0.00 sec)  mysql> select \* from test1 where a not **in** (**null**); Empty **set** (0.00 sec)  mysql> select \* from test1 where a not **in** (**null**,2); 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) |

**结论：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。**

查询语句中使用 exists、not exists 对比 test1.a=test2.a，**因为=不能比较 NULL**，结果和预期一致。

**判断 NULL 只能用 IS NULL、IS NOT NULL**

**结论：count(字段)无法统计字段为 NULL 的值，count(\*)可以统计值为 null 的行。**

**什么是事务？**

**数据库中的事务是指对数据库执行一批操作，这些操作最终要么全部执行成功，要么全部失败，不会存在部分成功的情况。**

**事务的几个特性(ACID)**

**原子性(Atomicity)**

事务的整个过程如原子操作一样，最终要么全部成功，或者全部失败，这个原子性是从最终结果来看的，从最终结果来看这个过程是不可分割的。

**一致性(Consistency)**

一个事务必须使数据库从一个一致性状态变换到另一个一致性状态。

首先回顾一下一致性的定义。所谓一致性，指的是数据处于一种有意义的状态，这种状态是**语义上的**而不是**语法上的**。最常见的例子是转帐。例如从帐户 A 转一笔钱到帐户 B 上，如果帐户 A 上的钱减少了，而帐户 B 上的钱却没有增加，那么我们认为此时数据处于不一致的状态。

**从这段话的理解来看，所谓一致性，即，从实际的业务逻辑上来说，最终结果是对的、是跟程序员的所期望的结果完全符合的**

**隔离性(Isolation)**

一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

**持久性(Durability)**

一个事务一旦提交，他对数据库中数据的改变就应该是永久性的。当事务提交之后，数据会持久化到硬盘，修改是永久性的。

**Mysql 中事务操作**

mysql 中事务默认是隐式事务，执行 insert、update、delete 操作的时候，数据库自动开启事务、提交或回滚事务。

是否开启隐式事务是由变量autocommit控制的。

所以事务分为**隐式事务**和**显式事务**。

**隐式事务**

事务自动开启、提交或回滚，比如 insert、update、delete 语句，事务的开启、提交或回滚由 mysql 内部自动控制的。

**显式事务**

事务需要手动开启、提交或回滚，由开发者自己控制。

2 种方式手动控制事务：

**方式 1：**

语法：

//设置不自动提交事务  
set autocommit=0;  
//执行事务操作  
commit|rollback;

start transaction;//开启事务  
//执行事务操作  
commit|rollback;

**savepoint 关键字**

在事务中我们执行了一大批操作，可能我们只想回滚部分数据，怎么做呢？

我们可以将一大批操作分为几个部分，然后指定回滚某个部分。可以使用savepoin来实现，效果如下：

演示savepoint效果，认真看：

|  |
| --- |
| 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> savepoint part1;//设置一个保存点 Query OK, 0 rows **affected** (0.00 sec)  mysql> insert into test1 **values** (2); Query OK, 1 row **affected** (0.00 sec)  mysql> rollback to part1;//将savepint = part1的语句到当前语句之间所有的操作回滚 Query OK, 0 rows **affected** (0.00 sec)  mysql> commit;//提交事务 Query OK, 0 rows **affected** (0.00 sec)  mysql> select \* from test1; +------+ | a    | +------+ |    1 | +------+ 1 row in **set** (0.00 sec) |

从上面可以看出，执行了 2 次插入操作，最后只插入了 1 条数据。

savepoint需要结合rollback to sp1一起使用，可以将保存点sp1到rollback to之间的操作回滚掉。

**只读事务**

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。

用法如下：

start transaction read only;

只读事务中执行 delete 会报错。

**事务中的一些问题**

这些问题主要是基于数据在多个事务中的可见性来说的。

**脏读**

一个事务在执行的过程中读取到了其他事务还没有提交的数据。这个还是比较好理解的。

**读已提交**

从字面上我们就可以理解，即一个事务操作过程中可以读取到其他事务已经提交的数据。

事务中的每次读取操作，读取到的都是数据库中其他事务已提交的最新的数据（相当于当前读）

**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**

脏读、不可重复读、可重复读、幻读，其中最难理解的是幻读

以 mysql 为例：

**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**

幻读现象例子：

可重复读模式下，比如有个用户表，手机号码为主键，有两个事物进行如下操作

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）

上面操作对 A 来说就像发生了幻觉一样，明明查询 X（A 中第二步、第四步）不存在，但却无法插入成功

**幻读可以这么理解：事务中后面的操作（插入号码 X）需要上面的读取操作（查询号码 X 的记录）提供支持，但读取操作却不能支持下面的操作时产生的错误，就像发生了幻觉一样。**

如果还是理解不了的，继续向下看，后面会详细的演示。

**事务的隔离级别**

当多个事务同时进行的时候，如何确保当前事务中数据的正确性，比如 A、B 两个事物同时进行的时候，A 是否可以看到 B 已提交的数据或者 B 未提交的数据，这个需要依靠事务的隔离级别来保证，不同的隔离级别中所产生的效果是不一样的。

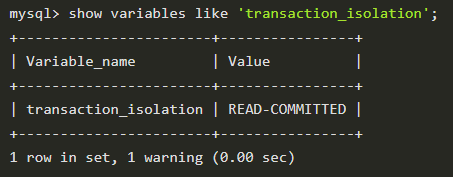
事务隔离级别主要是解决了上面多个事务之间数据可见性及数据正确性的问题。

**隔离级别分为 4 种：**

1. **读未提交：READ-UNCOMMITTED**
2. **读已提交：READ-COMMITTED**
3. **可重复读：REPEATABLE-READ**
4. **串行：SERIALIZABLE**

上面 4 中隔离级别越来越强，会导致数据库的并发性也越来越低。

**查看隔离级别**



**隔离级别的设置**

分 2 步骤，修改文件、重启 mysql，如下：

修改 mysql 中的 my.init 文件，我们将隔离级别设置为：READ-UNCOMMITTED，如下：

|  |
| --- |
| # 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行 transaction-isolation=READ-UNCOMMITTED |

以管理员身份打开 cmd 窗口，重启 mysql

**各种隔离级别中会出现的问题**

| **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 |

表格中和网上有些不一样，主要是幻读这块，幻读只会在可重复读级别中才会出现，其他级别下不存在。

下面我们来演示一下，各种隔离级别中可见性的问题，开启两个窗口，叫做 A、B 窗口，两个窗口中登录 mysql。

**READ-UNCOMMITTED：读未提交**

将隔离级别置为READ-UNCOMMITTED：

# 隔离级别设置,READ-UNCOMMITTED读未提交,READ-COMMITTED读已提交,REPEATABLE-READ可重复读,SERIALIZABLE串行  
transaction-isolation=READ-UNCOMMITTED

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。

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; |  |

看一下:

T2-A：无数据，T6-A：有数据，T6 时刻 B 还未提交，此时 A 已经看到了 B 插入的数据，**说明出现了脏读**。

T2-A：无数据，T6-A：有数据，查询到的结果不一样，**说明不可重复读**。

**结论：读未提交情况下，可以读取到其他事务还未提交的数据，多次读取结果不一样，出现了脏读、不可重复读**

**READ-COMMITTED：读已提交**

将隔离级别置为READ-COMMITTED

transaction-isolation=READ-COMMITTED

重启 mysql

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; |  |

看一下:

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**

将隔离级别置为REPEATABLE-READ

transaction-isolation=REPEATABLE-READ

重启 mysql

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; 有数据 |  |

看一下:

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。**

**幻读演示**

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。

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 insert into t\_user values (1,'路人甲 '); 成功 |
| T4 |  | select \* from t\_user; 有数据 |
| T5 | -- 查看路人甲是否存在 select \* from t\_user where name='路人甲'; 否 |  |
| T6 |  | commit; |
| T7 | -- 插入路人甲 insert into t\_user values (2,'路人甲');  unique异常 |  |
| T8 | -- 查看路人甲Java是否存在 select \* from t\_user where name='路人甲'; 否 |  |
| T9 | commit; |  |

看一下:

A 想插入数据路人甲，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。

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。

这个类似于 java 中的java.util.concurrent.lock.ReentrantReadWriteLock类产生的效果。

下面演示读写互斥的效果。

将隔离级别置为SERIALIZABLE

transaction-isolation=SERIALIZABLE

重启 mysql

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。

上面这个演示的是读写互斥产生的效果，大家可以自己去写一下写读、写写互斥的效果。

**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。**

**关于隔离级别的选择**

1. 需要对各种隔离级别产生的现象非常了解，然后选择的时候才能游刃有余
2. 隔离级别越高，并发性也低，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。

**视图**

**需求背景**

电商公司领导说：给我统计一下：当月订单总金额、订单量、男女订单占比等信息，我们啪啦啪啦写了一堆很复杂的 sql，然后发给领导。

这样一大片 sql，发给领导，你们觉得好么？

如果领导只想看其中某个数据，还需要修改你发来的 sql，领导日后想新增其他的统计指标，你又会发送一大坨 sql 给领导，对于领导来说这个 sql 看起来很复杂，难以维护。

实际上领导并不关心你是怎么实现的，他关心的只是这些指标，并且方便查看、查询，而你却把复杂的实现都发给了领导。

那我们有什么办法隐藏这些细节，只暴露简洁的结果呢？

数据库已经帮我们想到了：使用视图来解决这个问题。

**什么是视图**

**概念**

视图是在 mysql5 之后出现的，是一种虚拟表，行和列的数据来自于定义视图时使用的一些表中，**视图的数据是在使用视图的时候动态生成的，视图只保存了 sql 的逻辑，不保存查询的结果**。

**使用场景**

多个地方使用到同样的查询结果，并且该查询结果比较复杂的时候，我们可以使用视图来隐藏复杂的实现细节。

**视图和表的区别**

|  | **语法** | **实际中是否占用物理空间** | **使用** |
| --- | --- | --- | --- |
| 视图 | create view | 只是保存了 sql 的逻辑 | 增删改查，实际上我们只使用查询 |
| 表 | create table | 保存了数据 | 增删改查 |

**视图的好处**

* 简化复杂的 sql 操作，不用知道他的实现细节
* 隔离了原始表，可以不让使用视图的人接触原始的表，从而保护原始数据，提高了安全性

**创建视图**

**语法**

create view 视图名  
as  
查询语句;

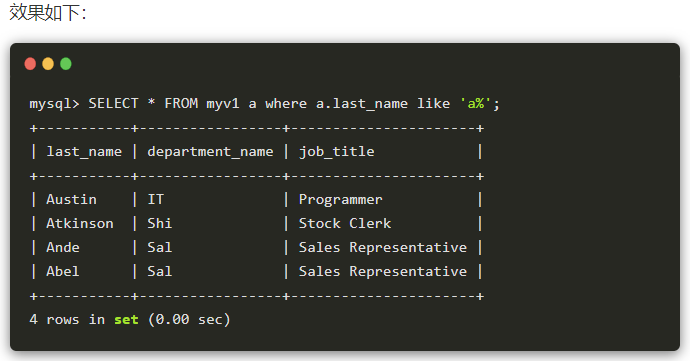
**视图的使用步骤**

* 创建视图
* 对视图执行查询操作

**案例 1**

查询姓名中包含 a 字符的员工名、部门、工种信息

|  |
| --- |
| /\*案例1：查询姓名中包含a字符的员工名、部门、工种信息\*/ /\*①创建视图myv1\*/ CREATE VIEW myv1 AS   SELECT     t1.last\_name,     t2.department\_name,     t3.job\_title   FROM employees t1, departments t2, jobs t3   WHERE t1.department\_id = t2.department\_id         AND t1.job\_id = t3.job\_id;  /\*②使用视图\*/ SELECT \* FROM myv1 a where a.last\_name like 'a%'; |



上面我们创建了一个视图：myv1，我们需要看员工姓名、部门、工种信息的时候，不用关心这个视图内部是什么样的，只需要查询视图就可以了，sql 简单多了。