

# 简介

## 数据库

数据库指的是数据的仓库，常用来指数据库软件，可以帮助高效管理数据。

**数据库三范式**

**第一范式（1NF）**

字段具有原子性,不可再分。所有关系型数据库系统都满足第一范式。

数据库表中的字段都是单一属性的，不可再分。例如，姓名字段，其中的姓和名必须作为一个整体，无法区分哪部分是姓，哪部分是名，如果要区分出姓和名，必须设计成两个独立的字段。

**第二范式（2NF）**

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。

要求数据库表中的每个实例或行必须可以被唯一地区分。通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键。

第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，第二范式就是非主属性非部分依赖于主关键字。

**第三范式（3NF）**

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。

所以第三范式具有如下特征：

1，每一列只有一个值

2，每一行都能区分。

3，每一个表都不包含其他表已经包含的非主关键字信息。

## SQL语言

**SQL**语句（Structured Query Language）是结构化查询语言、非过程性的语言，每一条SQL执行完都会有一个具体的结果出现，具有定义、操纵、查询和控制关系型数据库的四方面功能。

**DDL**（Data Definition Language）数据定义语言，用来定义数据库的对象，如数据表、视图、索引等，是对数据库或表的结构进行操作。

**DML**（Data Manipulation Language）数据处理语言，用来在数据库表中更新，增加和删除记录，是对表数据进行操作，如update，insert，delete。

**DQL**（Data Query Language）数据查询语言，如select。

**DCL**（Data Control Language）数据控制语言，指用于设置用户权限和控制事务语句。如grant，revoke，if…else，while，begin transaction。

## MySQL

### 性能

最大数据量

MySQL 没有限制单表最大记录数，它取决于操作系统对文件大小的限制。单表行数超过500万行或者单表容量超过2GB，可以考虑分库分表。

|  |  |
| --- | --- |
| 文件系统 | 单文件大小限制 |
| FAT32 | 最大4G |
| NTFS | 最大64G |
| NTFS5.0 | 最大2TB |
| EXT2 | 块大小为1024字节，文件最大容量16GB；块大小为4096字节，文件最大容量为2TB |
| EXT3 | 块大小为4KB，文件最大容量为4TB |
| EXT4 | 理论可以大于16TB |

最大并发数

并发数是指同一时刻数据库能处理多少个请求，由 max\_connections 和 max\_user\_connections 决定。

max\_connections 是指 MySQL 实例的最大连接数，上限值是 16384，max\_user\_connections 是指每个数据库用户的最大连接数。

MySQL 会为每个连接提供缓冲区，缓冲区消耗更多的内存。如果连接数设置太高硬件吃不消，太低又不能充分利用硬件。一般要求两者比值超过 10%，计算方法如下：

max\_used\_connections / max\_connections \* 100% = 3/100 \*100% ≈ 3%

查看最大连接数与响应最大连接数：

|  |
| --- |
| show variables like '%max\_connections%'; show variables like '%max\_user\_connections%'; |

在配置文件 my.cnf 中修改最大连接数：

max\_connections = 100

max\_used\_connections = 20

### 数据类型

**数值型**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **大小** | **范围（有符号）** | **范围（无符号）** | **用途** |
| TINYINT | 1字节 | -128~127 | 0~255 | 小整数值 |
| SMALLINT | 2字节 | -32768~32767 | 0~65535 | 大整数值 |
| MEDIUMINT | 3字节 | -8 388 608~8 388 607 | 0~16777215 | 大整数值 |
| INT或INTEGER | 4字节 | - | 0~ | 大整数值 |
| BIGINT | 8字节 | - | 0~ | 极大整数值 |
| FLOAT | 4字节 | - | 0~ | 单精度浮点数值 |
| DOUBLE | 8字节 | - | 0~ | 双精度浮点数值 |
| DECIMAL | 对DECIMAL(M,D),如果M>D，为M+2，否则为D+2 | 依赖于M和D的值 | 依赖于M和D的值 | 小数值 |

decimal(m,d)是数值类型中专门用来存储和钱有关的数据，m表示整个数的长度，d表示m中占几个小数位。

**时间和日期类型**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **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 | 8 | 1970-01-01 00:00:00~2037年某时 | YYYYMMDD HHMMSS | 混合日期和时间值，时间戳 |

每个时间类型都有一个有效值范围和一个零值，当指定不合法的MySQL不能表示的值时使用零值。

timestamp这个类型，会在每次修改记录的时候自动的更新为修改的时间。

datetime类型在5.6中字段长度是5个字节，datetime类型在5.5中字段长度是8个字节。

**字符串类型**

|  |  |  |
| --- | --- | --- |
| **类型** | **大小** | **用途** |
| CHAR | 0~255 | 固定长度字符串 |
| VARCHAR | 0~65535 | 可变长度字符串 |
| 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 | 0~16 777 215 | 二进制形式的中等长度文本数据 |
| MEDIUMTEXT | 0~16 777 215 | 中等长度文本数据 |
| LONGBLOB | 0~4 294 967 295 | 二进制形式的极大文本数据 |
| LONGTEXT | 0~4 294 967 295 | 极大文本数据 |

**varchar 和char的比较**

char和varchar类型类似，但两者保存和检索的方式不同，最大长度和是否尾部空格被保留等方面也不同，在存储或检索过程中不进行大小写转换。

char不可变长度，浪费空间，性能较好。

varchar可变长度，节省空间，性能较低。

例如：存储abcd字符串，如果定义成varchar(10)，此时存储的数据不足10个，这时会把多余的取消。如果使用char(10),则会把存储的数据添加到10个，不足的部分用\0替代。

如果存储的字符串长度超出了表中列的长度，则会存储报错。

**数据类型的选择**

**数据类型的选择原则是更简单或者占用空间更小**；

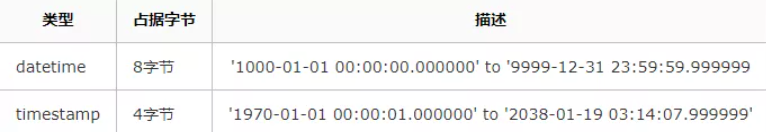
1、如果长度能够满足，整型尽量使用 tinyint、smallint、medium\_int 而非 int。

2、如果字符串长度确定，采用 char 类型。

3、如果 varchar 能够满足，不采用 text 类型。

4、精度要求较高的使用 decimal 类型，也可以使用 BIGINT，比如精确两位小数就乘以 100 后保存。

5、尽量采用 timestamp 而非datetime，相比 datetime，timestamp 占用更少的空间，以 UTC 的格式储存自动转换时区。



**避免空值；**

MySQL 中字段为 NULL 时依然占用空间，会使索引、索引统计更加复杂。从 NULL 值更新到非 NULL 无法做到原地更新，容易发生索引分裂影响性能。

因此尽可能将 NULL 值用有意义的值代替，也能避免 SQL 语句里面包含 is not null 的判断。

**Text 类型优化**；

由于 Text 字段储存大量数据，表容量会很早涨上去，影响其他字段的查询性能。建议抽取出来放在子表里，用业务主键关联。

### 字符集的使用

mysql有六处使用了字符集，如下所示：

**client**是客户端使用的字符集。

**connection**是连接数据库的字符集设置类型，如果程序没有指明连接数据库使用的字符集类型就按照服务器端默认的字符集设置。

**database**是数据库服务器中某个库使用的字符集设定，如果建库时没有指明，将使用服务器安装时指定的字符集设置。

**results**是数据库给客户端返回时使用的字符集设定，如果没有指明，使用服务器默认的字符集。

**server**是服务器安装时指定的默认字符集设定。

**system**是数据库系统使用的字符集设定。（utf-8不可修改）

**字符集的修改**

第一种：当前窗口临时修改 set names gbk ;只对当前窗口有效，关闭后就会失效

第二种：配置mysql/my.ini 文件

[mysql] 客户端配置

[mysqld] 服务器端配置

修改客户端字符集 [mysql] 后字符集 default-character-set=gbk

**查看字符集**

|  |
| --- |
| mysql> show variables like '%char%';  mysql> show variables like 'character%'; |

# 语法

**注释**

|  |
| --- |
| -- 单行注释，和注释之间要有空格分开 ##单行注释，和注释之间不需要空格分开 /\* 多行注释 \*/ |

## 库相关

**创建数据库**

|  |
| --- |
| -- 创建名为database\_name的数据库 CREATE DATABASE `demo` character set utf8mb4; -- 如果创建的数据库已存在就会报错，可以用下列语句避免 CREATE DATABASE if not exists `demo`; |

**删除数据库**

|  |
| --- |
| -- 删除数据库 drop database `demo`; |

**修改数据库**

|  |
| --- |
| -- 修改数据库指定表默认的数据编码方式  alter database `demo` character set utf8mb4; |

**查询数据库**

|  |
| --- |
| -- 查询所有数据库 show databases; -- 查询数据库定义 show create database `demo`; -- 查询当前正在使用的数据库 select *database*();  -- 切换到指定数据库  use `demo`; |

## 表相关

在创建表之前，要注意先使用use database语法指定需要创建数据库表的库。

**创建数据库表**

|  |
| --- |
| -- 创建数据库表，每一行的结束位置用“,”隔开，最后一行不要使用“,”及其他符号 create table `goods` (  `id` bigint(0) not null auto\_increment comment '主键',  `classify` bigint(0) NOT null comment '商品分类ID',  `title` varchar(30) unique comment '商品名称',  `price` decimal(10, 2) default 0.0 comment '商品价格',  `sales` int(0) default 0 comment '商品销量',  `create\_time` timestamp null default null on update *current\_timestamp* comment '创建时间',  `update\_time` datetime default null comment '最后修改时间',  `is\_deleted` int(4) default 0 comment '逻辑删除',  primary key (`id`) );  -- 复制数据库表结构 create table `table2` like `table`; -- 复制数据库表结构及数据 create table `table3` select \* from `table`; |

MySQL中约束表示通过某些限制来规定数据表中的某列数据是否可以为null、是否可以重复等限制。约束的目的是为了保证存储在数据表中的数据完整性和有效性。

**1、主键约束primary key：**主键一般表示数据库表中的某一列，使用这一列来区分数据表中的数据和其他数据不同。这一列的数据在整个数据表中是不允许重复的，且不能为空。如果表的主键是int 类型，这时可以在主键的后面添加 **auto\_increment** ，这时表中这一列在添加数据的时候，会自动的增长。

**2、唯一约束unique：**表示该列的值不允许重复，一张表中可有多个唯一约束，只能有一个(两个)作为主键约束。

**3、非空约束not null：**表示该字段的值不能为空。

**4、默认约束default：**可以为该字段设置默认值。

**删除数据库表**

|  |
| --- |
| -- 删除指定数据库表 drop table `table`; |

**修改数据库表**

|  |
| --- |
| -- 修改数据库表名 alter table `old\_name` RENAME TO `new\_name`; -- 设置数据库表默认字符集 alter table `table` CHARACTER SET utf8mb4; -- 添加数据库表新列 alter table `table` ADD `column\_name` VARCHAR(20) UNIQUE ; -- 修改数据库表指定列的类型、长度和约束属性 alter table `table` MODIFY `column\_name` VARCHAR(30) DEFAULT 'mangmang.xu'; -- 修改数据库表指定列的名称、类型、长度和约束属性 alter table `table` CHANGE `old\_column\_name` `new\_column\_name`  VARCHAR(40) DEFAULT 'UNKNOWN'; -- 删除数据库表指定列 alter table `table` drop `column\_name`; |

在实际开发中，一般在表创建好之后旧不会再对表中的结构进行修改，因此会在创建表的时候预留多个备用列来防止以后需要在表中添加或修改列。

**查询数据库表**

|  |
| --- |
| -- 查询当前数据库下所有的表 show tables;  -- 查询数据库表结构 desc `table`;  -- 查询数据库表定义 show create table `table` |

## 表数据相关

### 增删改数据

**添加数据**

|  |
| --- |
| -- 添加数据语法 insert into `goods`(`classify`, `title`, `price`, `sales`, `create\_time`, `update\_time`)  values (1, '手机', 2998.00, 300, *now*(), *now*()); -- 添加完整的一条记录 insert into `goods` values (2, 1, '电脑', 5998.00, 100, *now*(), *now*(), 0); -- 批量添加记录 insert into `goods` values (3, 2, '三国演义', 28.80, 20, *now*(), *now*(), 0),  (4, 2, '西游记', 28.80, 40, *now*(), *now*(), 0); |

1. 前面的列名和后面的values中的值一定要一一对应，且数据的大小应在列的范围内。
2. 在添加字符串和日期数据时，需要将其使用单引号或双引号引起来，推荐使用单引号。
3. 主键不能为null，非空约束也不能为null。
4. 当某一列的数据有默认值时，该列可以不在插入的数据中。
5. 没有设置自增的列，必须手动为其赋值；有设置自增的列，可以不在插入的数据中。
6. 当添加一条完整的数据时，所有的列都可以省略；如果添加的数据不完整，则所有的列均不可省略。

MyISAM存储引擎的数据库表中大批量插入数据时，可以使用DISABLE KEYS和ENABLE KEYS用来打开或者关闭MyISAM表非唯一索引的更新来优化插入速度。

|  |
| --- |
| alter table t\_name DISABLE KEYS; loading the data... alter table t\_name ENABLE KEYS; |

**删除数据**

|  |
| --- |
| -- 删除数据库表中id为5的数据 delete from `table` where `id` = 5; -- 删除数据库表中所有的数据并创建一个一模一样的空表 truncate `table`; |

1、删除数据时，如果没有添加where条件语句，则会将表中所有的数据删除。

2、delete不可以删除某一列的值，此时可以使用update完成

**delete语句和truncate语句的区别：**

1、delete配合where条件语句可以准确删除指定数据；truncate不能配合where删除指定数据。

2、delete是在原表上进行删除数据操作；truncate是将原表删除之后再创建出一个一模一样的空表。

3、delete删除后执行添加语句时，所有自增数据的值都是从原表最后一条数据记录开始往后自增；truncate删除全部数据后将自增恢复为从1开始自增。

**修改数据**

|  |
| --- |
| -- 将id为4的数据的`sales`修改为50，`price`修改为38.88 update `goods` set `sales` = 50, `price` = 38.88 where `id` = 4; |

1、如果没有添加where条件语句判断需要修改的是哪些数据，则会将所有数据的该列的值进行修改。

### 简单查询

#### SQL执行顺序

**手写SQL语句的顺序 MySQL读取的顺序**

|  |  |
| --- | --- |
| select distinct <select\_list>  from <left\_table> <join\_type>  join <right\_table>  on <join\_condition>  where <where\_condition>  group by < group\_by\_list>  having <having\_condition>  order by <order\_by\_condition>  limit < limit\_number> | from <left\_table>  on <join\_condition><join\_type>  join <right\_table>  where <where\_condition>  group by <group\_by\_list>  having <having\_condition>  select distinct <select\_list>  order by <order\_by\_condition>  limit <limit\_number> |

#### 常量、运算和函数查询

|  |
| --- |
| select 100; -- 查询常量 select 'tom';  select *version*(); -- 查询当前数据库版本 select *now*(); -- 查询当前时间 select *database* (); -- 查询当前正在使用的数据库  select *lower*('abcdABCD'); -- 将大写字符串转换为小写 select *upper*('abcdABCD'); -- 将小写字符串转换为大写 select *concat*('a','b','c','d,'); -- 字符串拼接操作 select *substr*('123456',2); -- 从指定位置截取子字符串 select *substring*('123456',2,2); -- 从指定位置开始截取指定长度的字符串 select *instr*('123456','5'); -- 查找子字符串出现的开始索引位置 select *trim*(' 123456 '); -- 去掉字符串前后空格 select *lpad*('666',10,'-----'); -- 用指定字符串从左边填充到指定位数 select *rpad*('666',10,'-----'); -- 用指定字符串从右边填充到指定位数 select *replace*('123456','456','123'); -- 替换字符串 select 100+100; -- 查询运算+、-、\*、/、% select 100-100;  select *ceil*(12.12); -- 向上取整 select *floor*(12.12); -- 向下取整 select *round*(12.5); -- 四舍五入 select *rand*() \* 100; -- 随机数0~1之间 select *abs*(-100); -- 求绝对值 select *mod*(10,3); -- 取余/模 select *truncate*(1.23456,3); -- 截断指定数位的小数  select *now*(); -- 当前时间（年月日，时分秒） select *curdate*(); -- 当前日期（年月日） select *curtime*(); -- 当前时间（时分秒） select *year*(*now*()); -- 获取给定时间的年份 select *month*(*now*()); -- 获取给定时间的月份 select *day*(*now*()); -- 获取给定时间的天数 select *hour*(*now*()); -- 获取给定时间的小时数 select *minute*(*now*()); -- 获取给定时间的分钟数 select *second*(*now*()); -- 获取给定时间的秒数  /\* %Y 四位年 %m 二位月 %d 二位日 %H 二位小时数 %i 二位分钟数 %s 二位秒钟数 \*/ -- 格式化日期时间，把日期转换为字符串 select *date\_format*(*now*(),'%Y年%m月%d日%H时%i分%s秒'); -- 转换字符串为日期 select *str\_to\_date*('2019年05月11日14时50分42秒','%Y年%m月%d日%H时%i分%s秒'); |

#### 聚合(统计)函数的查询

|  |
| --- |
| -- 获取该表的所有行数，包括为null的行 select *count*(\*) from `goods`; -- 获取该表中name列的行数，跳过为null的行 select *count*(`title`) from `goods`; -- 获取某列的总和，sum仅对数值起作用，否则会报错 select *sum*(`price`) from `goods`; -- 获取某列的平均值 select *avg*(`price`) from `goods`; -- 获取某列的最大值 select *max*(`price`) from `goods`; -- 获取某列的最小值 select *min*(`price`) from `goods`; |

#### 列的查询

查询时尽量不要使用\*来查询所有的列，明确指定要查询的列，避免不必要的消耗（CPU、IO、内存和网络宽带等）。而且select \* 增加了使用覆盖索引的可能性，还有当表结构发生改变时，之前的语句也会需要更新。

查询时不仅列名可以使用别名来显示，表名同样可以使用别名来显示。而且同一张表可以通过设置不同的别名来将其视为两张表来使用。

|  |
| --- |
| -- 查询列，多列使用逗号隔开 select `title`, `price` from `goods`; -- 查询全部列 select \* from `goods`; -- 混合查询，可以查询列、常量、函数等 select `title`, 100, 100\*100, *now*() from `goods`; -- 查询并以别名显式列名 select `title` as `图书名`, `price` as `价格` from `goods`; -- 查询并以别名显式列名，省略as关键字 select `title` `图书名`, `sales` `销量` from `goods`; -- 查询列并去除重复数据，distinct表示去重，多列使用逗号隔开 select distinct `title`, `price` from `goods`; |

#### 条件查询

条件查询语法：**select 列名 from 表名 where 条件;**

**> 、 < 、>= 、 <= 、 = 、<>**

|  |
| --- |
| -- 获取`sales`大于50的全部数据 select \* from `goods` where `sales` > 50; -- 获取`sales`大于等于50的全部数据 select \* from `goods` where `sales` >= 50; -- 获取`title`不等于'三国演义'的全部数据 select \* from `goods` where `title` <> '三国演义'; -- 获取`title`等于'西游记'的全部数据 select \* from `goods` where `title` = '西游记'; |

**and(与)和or(或)**

|  |
| --- |
| -- 查询`title`为'三国演义'且`sales`等于300的全部数据 select \* from `goods` where `title` = '三国演义' and `sales` = 300 ; -- 查询`title`为'三国演义'或`sales`等于300的全部数据 select \* from `goods` where `title` = '三国演义' or `sales` = 300; |

**is null**

|  |
| --- |
| -- 查询`title`为null的全部数据 select \* from `goods` where `title` is null; |

**like**

|  |
| --- |
| -- 查询`title`为4个字符，且第2、3和4个字符为'国演义'的全部数据，\_表示1个字符 select \* from `goods` where `title` like '\_国演义'; -- 查询`title`包含有'演义'的全部数据， %表示任意个字符 select \* from `goods` where `title` like '%演义%'; -- 查询`title`为4个字符且第2、3个字符为'国演'的全部数据 select \* from `goods` where `title` like '\_国演\_'; |

**between and**

|  |
| --- |
| -- 查询`sales`为50到100的全部数据 select \* from `goods` where `sales` between 50 and 100; |

**in和exists**

in关键字查询表示只要满足括号中任意一个值，即可被查询到，相当于多个or条件的叠加。in查询的子条件返回结果中只能有1个字段。括号中包含的值不应过多，对于连续的数值，能用between就不要使用in。

**in查询就是先将子查询条件的记录全都查出来，假设结果集为B，共有m条记录，然后再将子查询条件的结果集分解成m个，再进行m次查询。**

|  |
| --- |
| -- 查询id为3、4的全部数据 select \* from `goods` where `id` in(3,4);  select \* from `goods` where *exists*(select \* from `goods`); |

exists查询是对外表用loop逐条查询，每次查询都会查看exists的条件语句。当 exists里的条件语句能够返回记录行时(无论记录行是多少，只要能返回)，条件就为真 , 返回当前loop到的这条记录。反之如果exists里的条件语句不能返回记录行，条件为假，则当前loop到的这条记录被丢弃。

**总的来说，如果外表有n条记录，那么exists查询就是将这n条记录逐条取出，然后判断n遍exists条件。**

|  |
| --- |
| -- 查询`id`大于3的数据 select \* from `goods` t1 where *exists*(  select \* from `goods` t2 where t2.`id` > 2 and t1.`id` = t2.`id` ); |

**in和exists的区别**

in和exists主要区别是驱动顺序的改变，MySQL中的in语句是把外表和内表作join连接，而exists语句是对外表作nest loop循环，每次loop循环再对内表进行查询。如果是exists，那么以外层表为驱动表，先被访问，如果是IN，那么先执行子查询。**所以IN适合于外表大而内表小的情况；EXISTS适合于外表小而内表大的情况。如果查询的两个表大小相当，那么用in和exists差别不大。**

由于查询优化器的不断升级，很多场景这两者性能差不多一样了。但是一般使用in查询时可以考试能否替换为join查询。

**any、some和all**

any 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。语句some是any的别名，用法相同。

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、不等于其中的其中的所有数据。

|  |
| --- |
| -- 查询`id`大于子查询中所有值的数据 select \* from `goods` where `id` > all(select `id` from `goods` where `id` < 2); -- 查询`id`大于子查询中任意一个值的数据 select \* from `goods` where `id` > any(select `id` from `goods` where `id` > 2); -- 查询`id`大于子查询中任意一个值的数据 select \* from `goods` where `id` > some(select `id` from `goods` where `id` > 2); |

**not**

not关键字可以与is null、like、between and等关键字连用，表示非的意思。

|  |
| --- |
| -- 查询`title`不为null的全部数据 select \* from `goods` where `title` is not null;  -- 查询`title`不包含有'演义'的全部数据 select \* from `goods` where `title` not like '%演义%';  -- 查询`sales`不为50到100的全部数据 select \* from `goods` where `sales` not between 50 and 100;  -- 查询id不为3、4的全部数据 select \* from `goods` where `id` not in(3,4);  -- 查询`id`不大于3的数据 select \* from `goods` t1 where not *exists*(  select \* from `goods` t2 where t2.`id` > 2 and t1.`id` = t2.`id` ); |

#### 排序查询

asc 默认排序，从小到大，表示升序；desc 降序，从大到小。

|  |
| --- |
| -- 按照`price`升序排序 select \* from `goods` order by `price` asc; -- 按照`price`降序排序 select \* from `goods` order by `price` desc; -- 按照`price`升序排序，`classify`降序排序 select \* from `goods` order by `classify` desc, `price` asc ; |

#### 分页查询

**语法 : limit begin,size;**

begin表示每页的第一个数据在表中的位置，且该位置从0开始标记，类似于数组下标；size表示每页显示的数据个数。在只需要使用1条数据的情况下，使用limit 1。

begin可以求公式求得:（页码 - 1）\* 每页数量

假设： 1页，每页3条 得 ：( 1 - 1 ) \* 3 = 0

假设： 2页，每页3条 得 ：( 2 - 1 ) \* 3 = 3

假设： 3页，每页3条 得 ：( 3 - 1 ) \* 3 = 6

|  |
| --- |
| -- 将表中的第1+0条至第3+0条数据显示到同一页，且该页显示的数据共3条 select \* from `user` limit 0,3; -- 将表中的第1+7条至第7+6条数据显示到同一页，且该页显示的数据共6条，  -- 若余下数据不足6条则显示余下全部数据 select \* from `user` limit 7,6; |

Limit 用于分页查询时越往后翻性能越差，解决的原则是尽量缩小扫描范围。

随着表中数据量的增加，直接使用limit分页查询会越来越慢。解决的办法是获取前一页的最大行数的id，然后根据这个最大的id来限制下一页中的起点。

|  |
| --- |
| select `id`, `title` from `goods` limit 866613, 20; -- 以前一页最后一条数据的某一属性作为基准优化后的SQL select `id`, `title` from `goods` where id> 866612 limit 20; |

#### 分组过滤查询

分组查询的目的是为了统计，没有统计的分组没有意义。havaing和where均可实现过滤，但是having可以使用聚合函数，通常位于group by后，作用于分组。

**简单分组查询语法**

**select \* from 表名 where 条件 group by 列名**

**分组统计查询语法**

**select 分组（统计）函数 from 表名 where 条件 group by 列名**

**分组过滤查询语法**

**select \* from 表名 where 条件 group by 列名 having 分组函数过滤**

|  |
| --- |
| -- 按`classify`分组图书信息 select \* from `goods` group by `classify`; -- 统计每个`publishing`出版了几本书 select *count*(\*), *sum*(sales) from `goods` group by `classify`; -- 统计每个`author`卖出了几本书，并过滤掉少于30的作者 select *sum*(sales), `classify` from `goods` group by `classify` having *sum*(sales) > 100; |

#### 流程控制

**两个分支**

**语法 : if(expression 1, expression 2, expression 3)**

如果expression 1值为真，返回expression 2的值，如果expression 1值为假，返回expression 3的值。

|  |
| --- |
| -- 查询所有图书信息，根据库存情况对图书销售情况给出说明。0库存说明卖完，有库存显示正在热卖。 se·lect \*,*if* (`sales` = 0,'sellout','hot') `status` from `goods` where id = 3; |
|  |

**多个分支**

**语法 :**

**case (字段)**

**when 条件1 then expression 1或value 1 (sql语句)**

**when 条件2 then expression 2或value 2 (sql语句)**

**……**

**else expression n或value n**

**end**

|  |
| --- |
| -- 1、查询所有图书信息，根据库存情况对图书销售情况给出说明。如果0库存。说明卖完，如果销量大于50显示爆款，其他显示热卖 select \*,  case  when `sales` = 0 then '卖完'  when `sales` > 50 then '爆款'  else '热卖'  end 'sale\_status' from `goods`; -- 2、如果库存为零。显示价格为原来1.5倍，如果销量大于50，价格是1.3倍，其他维持原价。 select \*,  case  when `sales` > 50 then `price` \* 1.3  when `sales` = 0 then `price` \* 1.5  else `price`  end 'new\_price' from `goods`; |

### 多表查询

#### 多表数据对应关系

**一对一对应关系**

在处理两张表之间数据的一对一对应关系时，需要添加一个对应关系列通过外键约束来将两张无关的表关联起来。该列中用于记录两张表中的对应关系，且可以任意添加到两张表中的任意一张表中。

|  |
| --- |
| -- 创建订单表，一个订单只能对应一个商品，和`goods`表是一对一的关系 create table `order`(  `id` bigint(64) not null auto\_increment comment '订单主键id',  `order\_no` varchar(50) not null comment '订单号',  `goods\_id` bigint(64) not null comment '商品id',  primary key (`id`) );  -- `goods`表中添加`order\_id`字段表示和`order`表的关系 alter table `goods` add column `order\_id` bigint(64) not null comment '订单id'; |

**添加外键约束**

外键实际上是一种约束，表示这个属性是另一个实体集中的主标识属性，即另一个表中的主键。外键用于建立和加强两个表数据之间的链接的一列或多列，保证数据的一致性。

|  |
| --- |
| -- 替`goods`表中的`order\_id`字段添加外键约束 alter table `goods` add constraint `order\_id` foreign key (`order\_id`)  references `order` (`id`) on DELETE restrict on update restrict; |

添加外键约束时有可能会提示如下错误：

Cannot add or update a child row: a foreign key constraint fails

这种报错一般都是由于表与外键取值范围参照表的相关值不匹配。

|  |
| --- |
| -- mysql禁用外键约束: SET FOREIGN\_KEY\_CHECKS=0; -- mysql启动外键约束: SET FOREIGN\_KEY\_CHECKS=1; -- 查看当前FOREIGN\_KEY\_CHECKS值： SELECT @@FOREIGN\_KEY\_CHECKS; |

因此需要在`goods`表中的`order\_id`字段添加数据并与`order`表中的`id`字段对应。

|  |
| --- |
| -- 查询不匹配的数值，更新外键约束参照表 select \* from `goods` where `order\_id` not in (select `id` from `order`);  -- 向`order`表中添加一条`id`为1的订单数据 insert `order` values (1, '20200623'); -- 然后修改`goods`表中的`order\_id`字段的值为1，与`order`表中添加的`id`为1的数据对应 update `goods` set `order\_id` = 1 where `id` > 3; |

解决值不匹配的问题后添加再次执行外键操作即可成功添加外键。

添加外键成功后，`goods`表中的`order\_id`字段就不能取`order`表中`id`字段中不存在的值了，否侧还会出现上述异常。

MySQL的外键约束较为损耗性能，因此一般在实际开发中，不会使用外键约束相关的数据库表，而是通过程序来保证数据的一致性。

|  |
| --- |
| -- 删除外键约束： alter table d`goods` drop foreign key `order\_id`; |

**一对多对应关系**

在处理两张表之间数据的一对多对应关系时，需要添加一个对应关系字段通过外键约束来将多张无关的表关联起来。该字段中用于记录多张表中的对应关系，必须添加到多张表中的表示多的那张表中。

**多对多对应关系**

在处理两张表之间数据的多对多对应关系时，将其简化为一对多的对应关系，具体实施方式为创建一张第三方表，并创建两个字段分别与两张表中的数据形成一对多的对应关系。

#### 多表关联查询

在进行查询时，每一次的查询结果都可以作为一张新的表来使用，并可以为其添加别名，具体使用方式就是在整个查询语句外使用小括号包裹即可。

在进行多表关联查询时，如果不将其中的数据关联起来，就会出现笛卡儿积的问题，从而导致查询的数据错误。

**多表关联查询语法**

**select column\_1, column\_2 ... from table\_1, table\_2 ... where expression\_1 ...**

|  |
| --- |
| -- 查询商品名称为'三国演义'的商品的订单号及其他字段 select t1.`id`, t1.`title`, t2.`order\_no` from `goods` t1, `order` t2 where t1.`order\_id` = t2.`id` and t1.`title` = '三国演义'; |

#### 子查询

子查询即是指将查询的结果作为查询的条件来使用，子查询（内查询）在主查询执行之前执行，主查询（外查询）使用子查询的结果。

**单行子查询**

单行子查询指的是查询的结果只有一条记录，并将其作为查询的条件来使用。

|  |
| --- |
| -- 查询最贵的商品信息 select \* from `goods` where `price` = (select *max*(`price`) from `goods`);  -- 查询价格在平均价格之上的所有商品信息 select \* from `goods` where `price` > (select *avg*(`price`) from `goods`); |

**多行子查询**

多行子查询指的是查询的结果有多条记录，并将这些记录作为查询的条件来使用。

|  |
| --- |
| -- 查询所有`id`大于子查询结果中最小值的商品信息 select \* from `goods` where `id` > any(select `id` from `goods` where `price` > 100); -- 查询`id`为2、3、4的图书信息 select \* from `goods` where `id` in(select `id` from `goods` where `id` > 1); |

多行子查询有两种方式，分别为不成对比较和成对比较:

|  |
| --- |
| -- 不成对比较 select \* from `goods` where `id` in (select `id` from `goods` where `id` <= 2) and `price` in (select `price` from `goods` where `id` <= 2); -- 成对比较 select \* from `goods` where (`id`, `price`) in (select `id`, `price` from `goods` where `id` <= 2); |

**from子句中使用子查询**

|  |
| --- |
| -- 获取各个商品分类的平均价格的最大值 select *max*(t.avg\_price) from (  select *avg*(t1.`price`) avg\_price from `goods` t1 group by t1.`classify`  ) t;  -- 返回比所属分类平均价格高的商品的`title`, `classify`, `price`及平均工资 -- 方法一 select `title`, `classify`, `price`,  (select *avg*(`price`) from `goods` where `classify` = t1.`classify`) avg\_price from `goods` t1 where `price` > (  select *avg*(`price`) from `goods` t2 where t2.`classify` = t1.`classify` ); -- 方法二 select t1.`title`, t1.`classify`, t1.`price`, t2.avg\_price from `goods` t1,(  select classify, *avg*(`price`) avg\_price from `goods` group by `classify`) t2 where t1.`classify` = t2.`classify` and t1.`price` > t2.avg\_price; |

**单列子查询表达式**

单列子查询表达式是在一行中只返回一列的子查询，只在下列情况下可以使用，例如：

1、select语句（from和where子句）

2、insert语句中的values列表中

3、case

4、select中除group by子句外的所有子句中

|  |
| --- |
| -- 查询与`id`为1的数据的`classify`相同的数据，并设置`classify\_title`为'Non\_Books' SELECT `title`, `price`,(  CASE  WHEN `classify` = (SELECT `classify` FROM `goods` WHERE `id` = 1) THEN 'Non\_Books'  ELSE 'Books'  END) `classify\_title` FROM `goods`; |

#### JOIN查询

join 的实现是采用 Nested Loop Join 算法，就是通过驱动表的结果集作为基础数据，通过该结数据作为过滤条件到下一个表中循环查询数据，然后合并结果。如果有多个 Join，则将前面的结果集作为循环数据，再次到后一个表中查询数据。驱动表和被驱动表尽可能增加查询条件，满足 ON 的条件而少用 Where，用小结果集驱动大结果集。被驱动表的 Join 字段上加上索引，无法建立索引的时候，设置足够的 Join Buffer Size。禁止 Join 连接三个以上的表，尝试增加冗余字段。

|  |
| --- |
| -- 内连接，多表内连接查询和多表关联查询的结果相同，A、B两表共有，两个结果集的交集 select \* from `goods` inner join `order` on `goods`.`id` = `order`.`goods\_id`; -- 左外连接，以左表为主表，右表为从表，A、B两表共有+A的独有，即A结果集的全部内容 select \* from `goods` left join `order` on `goods`.`id` = `order`.`goods\_id`; -- 右外链接，以右表为主表，左表为从表，A、B两表共有+B的独有，即B结果集的全部内容 select \* from `goods` right join `order` on `goods`.`id` = `order`.`goods\_id`; -- A的独有 select \* from `goods` left join `order` on `goods`.`id` = `order`.`goods\_id` where `order`.`id` is null; -- B的独有 select \* from `goods` right join `order` on `goods`.`id` = `order`.`goods\_id` where `goods`.`id` is null; -- AB全有，因为MySQL不支持FULL JOIN，下面是替代方法：left join + union + right join。 select \* from `goods` left join `order` on `goods`.`id` = `order`.`goods\_id` union select \* from `goods` right join `order` on `goods`.`id` = `order`.`goods\_id`; -- A的独有+B的独有 select \* from `goods` left join `order` on `goods`.`id` = `order`.`goods\_id` where `order`.`id` is null union select \* from `goods` right join `order` on `goods`.`id` = `order`.`goods\_id` where `goods`.`id` is null; |

#### UNION和UNION ALL

union和union all的区别之一在于对重复结果的处理。

|  |
| --- |
| -- union会对查询出来的结果进行去重，去重操作会降低效率 select \* from `goods` where `id` >= 2 union select \* from `goods` where `id` <= 2; -- union all则不会对查询出来的结果进行去重，效率更快 select \* from `goods` where `id` >= 2 union all select \* from `goods` where `id` <= 2; |

### 批量添加修改数据脚本

**设置参数log\_bin\_trust\_function\_creators**

|  |
| --- |
| -- 查看是否开启二进制日志 show variables like 'log\_bin'; -- 查看是否信任存储函数创建者配置 show variables like 'log\_bin\_trust\_function\_creators'; -- 开启全局信任函数创建者配置，1表示开启，0表示关闭 set global log\_bin\_trust\_function\_creators = 1; |

当开启二进制日志后，如果变量log\_bin\_trust\_function\_creators为OFF，那么创建或修改存储函数就会报如下错误：

“ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you \*might\* want to use the less safe log\_bin\_trust\_function\_creators variable)”

当二进制日志启用后，**log\_bin\_trust\_function\_creators**变量就会启用。它控制是否可以信任存储函数创建者，不会创建写入二进制日志引起不安全事件的存储函数。如果设置为0（默认值）者表示用户不得创建或修改存储函数，除非具有除create routine或alter routine特权之外的super权限。设置为0还强制使用deterministic特性或reads sql data或no sql特性声明函数的限制。如果该参数设置为1，则MySQL不会对创建存储函数实施这些限制，此变量也适用于触发器的创建。

这样添加了参数以后，如果mysqld重启，上述参数又会消失。永久方法：

windows下 my.ini[mysqld]加上log\_bin\_trust\_function\_creators=1

linux下 /etc/my.cnf下my.cnf[mysqld]加上log\_bin\_trust\_function\_creators=1

**创建函数**

创建函数，保证每条数据都不相同

|  |
| --- |
| -- 生成随机字符串 DELIMITER $$ CREATE FUNCTION *rand\_string*(n INT) RETURNS VARCHAR(255) BEGIN  DECLARE chars\_str VARCHAR(100)  DEFAULT 'abcdefghijklmnopqrstuvwxyzABCDEFJHIJKLMNOPQRSTUVWXYZ';  DECLARE return\_str VARCHAR(255) DEFAULT '';  DECLARE i INT DEFAULT 0;  WHILE i < n DO  SET return\_str = *CONCAT*(return\_str, *SUBSTRING*(chars\_str, *FLOOR*(1 + *RAND*() \* 52), 1));  SET i = i + 1;  END WHILE;  RETURN return\_str; END $$  -- 删除函数 drop function *rand\_string*; |

**随机产生数字**

|  |
| --- |
| -- 生成随机数字 DELIMITER $$ CREATE FUNCTION *rand\_num*( )  RETURNS INT(5) BEGIN  DECLARE i INT DEFAULT 0;  SET i = *FLOOR*(100+*RAND*() \* 10);  RETURN i; END $$ |

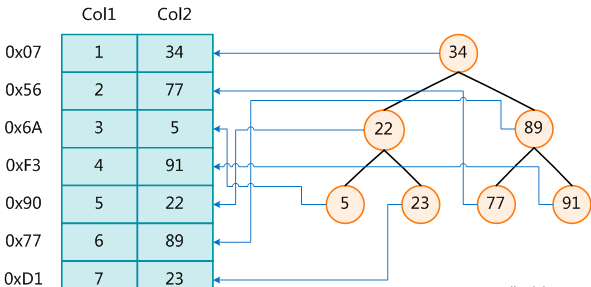
**创建并执行存储过程**

|  |
| --- |
| -- 创建存储过程 DELIMITER $$ CREATE PROCEDURE *insert\_goods*(IN START INT(10),IN max\_num INT(10)) BEGIN  DECLARE i INT DEFAULT 0;  #set autocommit =0 把autocommit设置成0  SET autocommit = 0;  REPEAT  SET i = i + 1;  INSERT INTO demo.`goods` (`id`, `classify`, `title` , `price` , `sales` , `create\_time`, `update\_time`, `is\_deleted`)  VALUES ((START+i) ,(START+i) % 5 , *rand\_string*(6), *rand\_num*(), *rand\_num*(), *now*(), *now*(), 0);  UNTIL i = max\_num  END REPEAT;  COMMIT; END $$ -- 调用存储过程 call *insert\_goods*(10, 100); -- 删除存储过程 DELIMITER ; drop PROCEDURE *insert\_goods*; |

# 索引

索引（Index）是帮助MySQL高效获取数据的数据结构，索引的本质是数据结构。索引的目的在于提高查询效率，可以类比字典。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。下图就是一种可能的索引方式示例：



左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址

为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在一定的复杂度内获取到相应数据，从而快速的检索出符合条件的记录。

数据本身之外，数据库还维护着一个满足特定查找算法的数据结构，这些数据结构以某种方式指向数据，这样就可以在这些数据结构的基础上实现高级查找算法，这种数据结构就是索引。

一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上，平常所说的索引，如果没有特别指明，都是指B树(Balance Tree多路搜索树，并不一定是二叉的)结构组织的索引。

MySQL的索引结构除了B-Tree结构，还有Hash索引、full-text全文索引和R-Tree索引。

**普通索引**：最基本的索引。

**组合索引**：多个字段上建立的索引，能够加速复合查询条件的检索。

**唯一索引**：与普通索引类似，但索引列的值必须唯一，允许有空值。

**组合唯一索引**：列值的组合必须唯一。

**主键索引**：特殊的唯一索引，用于唯一标识数据表中的某一条记录，不允许有空值，用primary key约束。

**全文索引**：用于海量文本的查询，MySQL 5.6 之后的 InnoDB 和 MyISAM 均支持全文索引。由于查询精度以及扩展性不佳，更多的企业选择 Elasticsearch。

## 基本语法

**增删改索引**

|  |
| --- |
| -- 创建索引 create unique index `idx\_title` on `goods`(`title`) using hash ; -- 删除索引 drop index `idx\_title` on `goods`; -- 修改索引 alter table `goods` drop index `idx\_title`,  add fulltext index `idx\_title`(`title`) using btree ; |

**查看索引**

|  |
| --- |
| -- 查看指定数据库表的索引 show index from `goods`; |

执行查看索引结果如下：

|  |  |
| --- | --- |
| Table |  |
| Non\_unique |  |
| Key\_name |  |
| Seq\_in\_index |  |
| Column\_name |  |
| Collation |  |
| Cardinality |  |
| Sub\_part |  |
| Packed |  |
| Null |  |
| Index\_type |  |
| Comment |  |
| Index\_comment |  |

## explain关键字

### explain概述

使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理SQL语句的。分析查询语句或是表结构的性能瓶颈。

使用explain关键字可以获取如下信息：

1、表的读取顺序；

2、数据读取操作的操作类型；

3、哪些索引可以使用；

4、哪些索引被实际使用；

5、表之间的引用；

6、每张表有多少行被优化器查询。

使用方式：explain + SQL语句；执行后的结果包含如下信息：id、select\_type、table、type、possible\_key、key、key\_len、ref、rows、Extra。

### explain结果字段详情

**id字段**

select查询的序列号,包含一组数字，表示查询中执行select子句或操作表的顺序，共有如下三种情况：

1、id相同，执行顺序由上至下

2、id不同，如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行

3、id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行。衍生 = DERIVED。

**select\_type字段**

select\_type字段表示查询的类型，主要是用于区别普通查询、联合查询、子查询等复杂查询，其属性值共有如下六种：simple、primary、subquery、derived、union和union result。

|  |  |
| --- | --- |
| simple | 简单的 select 查询,查询中不包含子查询或者UNION |
| primary | 查询中若包含任何复杂的子部分，最外层查询则被标记为primary |
| subquery | 在SELECT或WHERE列表中包含了子查询 |
| derived | 在FROM列表中包含的子查询被标记为DERIVED(衍生) MySQL会递归执行这些子查询, 把结果放在临时表里 |
| union | 若第二个SELECT出现在UNION之后，则被标记为UNION； 若UNION包含在FROM子句的子查询中,外层SELECT将被标记为：DERIVED |
| union result | 从UNION表获取结果的SELECT |

**table字段**

table字段用于显示这一行的数据是关于哪张表的。

**type字段**

type显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是：

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

显示查询使用了什么类型，一般来说，得保证查询至少达到range级别，最好能达到ref。

|  |  |
| --- | --- |
| **system** | 表只有一行记录（等于系统表），这是const类型的特列，平时不会出现，这个也可以忽略不计 |
| **const** | 表示通过索引一次就找到了,const用于比较primary key或者unique索引。因为只匹配一行数据，所以很快 如将主键置于where列表中，MySQL就能将该查询转换为一个常量 |
| **eq\_ref** | 唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描 |
| **ref** | 非唯一性索引扫描，返回匹配某个单独值的所有行. 本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而， 它可能会找到多个符合条件的行，所以他应该属于查找和扫描的混合体。 |
| **range** | 只检索给定范围的行,使用一个索引来选择行。key 列显示使用了哪个索引 一般就是在你的where语句中出现了between、<、>、in等的查询 这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一点，不用扫描全部索引。 |
| **index** | Full Index Scan，index与ALL区别为index类型只遍历索引树。这通常比ALL快，因为索引文件通常比数据文件小。 （也就是说虽然all和Index都是读全表，但index是从索引中读取的，而all是从硬盘中读的）。 |
| **all** | Full Table Scan，将遍历全表以找到匹配的行 |

**possible\_keys字段**

显示可能应用在这张表中的索引，一个或多个。查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用。

**key字段**

实际使用的索引，如果为NULL，则没有使用索引。查询中若使用了覆盖索引，则该索引和查询的select字段重叠。

**key\_len字段**

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精确性的情况下，长度越短越好。key\_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key\_len是根据表定义计算而得，不是通过表内检索出的。

key\_len表示索引使用的字节数，根据这个值，就可以判断索引使用情况，特别是在组合索引的时候，判断所有的索引字段是否都被查询用到。

char和varchar跟字符编码也有密切的联系,latin1占用1个字节，gbk占用2个字节，utf8占用3个字节。（不同字符编码占用的存储空间不同）。

**案例演示**

n为该索引字段建表时定义的长度

1、索引字段为char类型+不可为Null时

key\_len = n\*（UTF-8=3、GBK=2、LATIN=1）

2、索引字段为char类型+允许为Null时

key\_len = n\*（UTF-8=3、GBK=2、LATIN=1）+ 1（NULL）

3、索引字段为varchar类型+不可为Null时，varchar需要额外的2个字节来记录长度

key\_len = n\*（UTF-8=3、GBK=2、LATIN=1）+ 2

4、索引字段为varchar类型+允许为Null时

key\_len = n\*（UTF-8=3、GBK=2、LATIN=1）+1（NULL）+ 2

5、正数、浮点数、时间类型的索引长度

key\_len = 字段本身的字段长度（不允许为NULL时）

key\_len = 字段本身的字段长度+1(允许为NULL时，因为要有是否为空的标记，需要占用1个字节)

注意：datetime类型在5.6中字段长度是5个字节，datetime类型在5.5中字段长度是8个字节

变长字段需要额外的2个字节（VARCHAR值保存时只保存需要的字符数，另加一个字节来记录长度(如果列声明的长度超过255，则使用两个字节)，所以VARCAHR索引长度计算时候要加2），固定长度字段不需要额外的字节。

而NULL都需要1个字节的额外空间,所以索引字段最好不要为NULL，因为NULL让统计更加复杂并且需要额外的存储空间。

所以，复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之和，这也可以用来判定复合索引是否部分使用，还是全部使用。

**ref字段**

显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列上的值。

由key\_len可知t1表的idx\_col1\_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac' 查询中与其它表关联的字段，外键关系建立索引。

**rows字段**

根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数。

根据底层使用存储引擎的不同，受影响的行数这个指标可能是一个估计值，也可能是一个精确值。即使受影响的行数是一个估计值（例如当使用InnoDB存储引擎管理表存储时），通常情况下这个估计值也足以使优化器做出一个有充分依据的决定。

**Extra字段**

extra字段包含不适合在其他列中显示但十分重要的额外信息，共有如下几种情况：

using filesort、using temporary、using index、using where、using join buffer和impossible where。

**using filesort**

说明mysql会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。 MySQL中无法利用索引完成的排序操作称为“文件排序”。

查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度

**using temporary**

使了用临时表保存中间结果,MySQL在对查询结果排序时使用临时表。常见于排序 order by 和分组查询 group by。

**using index**

表示相应的select操作中使用了覆盖索引(Covering Index)，避免访问了表的数据行，效率不错！ 如果同时出现using where，表明索引被用来执行索引键值的查找; 如果没有同时出现using where，表明索引用来读取数据而非执行查找动作。

如果同时出现using where，表明索引被用来执行索引键值的查找;

如果没有同时出现using where，表明索引用来读取数据而非执行查找动作。

覆盖索引（Covering Index）一说为索引覆盖。

理解方式一:就是select的数据列只用从索引中就能够取得，不必读取数据行，MySQL可以利用索引返回select列表中的字段，而不必根据索引再次读取数据文件,换句话说查询列要被所建的索引覆盖。

理解方式二:索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此它不必读取整个行。毕竟索引叶子节点存储了它们索引的数据;当能通过读取索引就可以得到想要的数据，那就不需要读取行了。一个索引包含了(或覆盖了)满足查询结果的数据就叫做覆盖索引。

注意：

如果要使用覆盖索引，一定要注意select列表中只取出需要的列，不可select \*，因为如果将所有字段一起做索引会导致索引文件过大，查询性能下降。

**using where**

表明使用了where过滤。

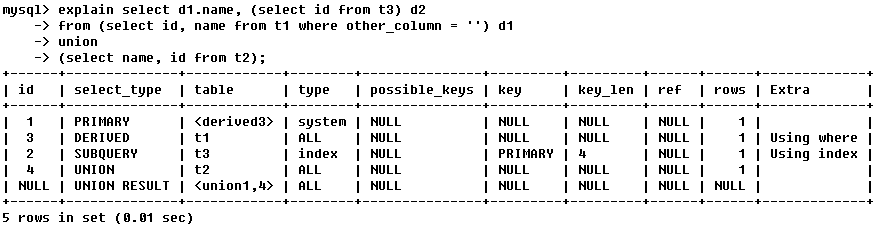
**using join buffer**

使用了连接缓存。

**impossible where**

where子句的值总是false，不能用来获取任何元组。

### explain使用实例



第一行（执行顺序4）：id列为1，表示是union里的第一个select，select\_type列的primary表 示该查询为外层查询，table列被标记为<derived3>，表示查询结果来自一个衍生表，其中derived3中3代表该查询衍生自第三个select查询，即id为3的select。【select d1.name......】

第二行（执行顺序2）：id为3，是整个查询中第三个select的一部分。因查询包含在from中，所以为derived。【select id,name from t1 where other\_column=''】

第三行（执行顺序3）：select列表中的子查询select\_type为subquery，为整个查询中的第二个select。【select id from t3】

第四行（执行顺序1）：select\_type为union，说明第四个select是union里的第二个select，最先执行【select name,id from t2】

第五行（执行顺序5）：代表从union的临时表中读取行的阶段，table列的<union1,4>表示用第一个和第四个select的结果进行union操作。【两个结果union操作】

## 索引失效

|  |
| --- |
| CREATE TABLE staffs (  id INT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR (24) NOT NULL DEFAULT '' COMMENT '姓名',  age INT NOT NULL DEFAULT 0 COMMENT '年龄',  pos VARCHAR (20) NOT NULL DEFAULT '' COMMENT '职位',  add\_time TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '入职时间'  ) CHARSET utf8 COMMENT '员工记录表' ;  INSERT INTO staffs(NAME,age,pos,add\_time) VALUES('z3',22,'manager',NOW());  INSERT INTO staffs(NAME,age,pos,add\_time) VALUES('July',23,'dev',NOW());  INSERT INTO staffs(NAME,age,pos,add\_time) VALUES('2000',24,'dev',NOW());  INSERT INTO staffs(NAME,age,pos,add\_time) VALUES('2000',25,'dev',NOW());  SELECT \* FROM staffs;  +----+------+-----+---------+---------------------+  | id | NAME | age | pos | add\_time |  +----+------+-----+---------+---------------------+  | 1 | z3 | 22 | manager | 2019-08-20 08:59:40 |  | 2 | July | 23 | dev | 2019-08-20 08:59:40 |  | 3 | 2000 | 24 | dev | 2019-08-20 08:59:40 |  | 4 | 2000 | 25 | dev | 2019-08-20 08:59:40 |  +----+------+-----+---------+---------------------+  #将name、age和pos设置为联合索引  ALTER TABLE staffs ADD INDEX idx\_staffs\_nameAgePos(name, age, pos);  ALTER TABLE staffs ADD INDEX idx\_staffs\_name(name); |

### 案例分析

1、全值匹配我最爱

|  |
| --- |
| EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 74 | const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+-------------+**  EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' AND age = 25;  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 78 | const,const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------+------+-------------+**  EXPLAIN SELECT \* FROM staffs WHERE NAME = 'July' AND age = 25 AND pos = 'dev';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 140 | const,const,const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+** |

2、最佳左前缀法则，如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列。

|  |
| --- |
| mysql> EXPLAIN SELECT \* FROM staffs WHERE age = 25 AND pos = 'dev';  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  mysql> EXPLAIN SELECT \* FROM staffs WHERE pos = 'dev';  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  mysql> EXPLAIN SELECT \* FROM staffs WHERE name = 'July';  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos,idx\_staffs\_name | idx\_staffs\_nameAgePos | 74 | const | 1 | Using where |**  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+** |

3、不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），会导致索引失效而转向全表扫描。

|  |
| --- |
| EXPLAIN SELECT \* FROM staffs WHERE left(NAME,4) = 'July';  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 4 | Using where |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+** |

4、存储引擎不能使用索引中范围条件右边的列

|  |
| --- |
| mysql> EXPLAIN SELECT \* FROM staffs WHERE name='z4';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 74 | const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+-------------+**  mysql> EXPLAIN SELECT \* FROM staffs WHERE name='z4'and age=22;  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 78 | const,const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------+------+-------------+**  mysql> EXPLAIN SELECT \* FROM staffs WHERE name='z4'and age=22 and pos='manager';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 140 | const,const,const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  mysql> EXPLAIN SELECT \* FROM staffs WHERE name='z4'and age>11 and pos='manager';  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | range | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 78 | NULL | 1 | Using where |**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+** |

5、尽量使用覆盖索引(只访问索引的查询(索引列和查询列一致))，减少select \*

|  |
| --- |
| mysql> EXPLAIN SELECT \* FROM staffs WHERE name='z3'and age=22 and pos='manager';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 140 | const,const,const | 1 | Using where |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+-------------+**  mysql> EXPLAIN SELECT name,age,pos FROM staffs WHERE name='z3'and age=22 and pos='manager';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+--------------------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+--------------------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 140 | const,const,const | 1 | Using where; Using index |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------------------+------+--------------------------+**  **mysql> EXPLAIN SELECT \* FROM staffs WHERE name='z3'and age>22 and pos='manager';**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | range | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 78 | NULL | 1 | Using where |**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  mysql> EXPLAIN SELECT name,age,pos FROM staffs WHERE name='z3'and age>22 and pos='manager';  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+--------------------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+--------------------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 74 | const | 1 | Using where; Using index |**  **+----+-------------+--------+------+-----------------------+-----------------------+---------+-------+------+--------------------------+** |

6、mysql 在使用不等于(!= 或者<>)的时候有时候无法使用索引会导致全表扫描

|  |
| --- |
| mysql> explain select \* from staffs where name='July';  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos,idx\_staffs\_name | idx\_staffs\_nameAgePos | 74 | const | 1 | Using where |**  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  mysql> explain select \* from staffs where name!='July';  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | idx\_staffs\_nameAgePos,idx\_staffs\_name | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  mysql> explain select \* from staffs where name<>'July';  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | idx\_staffs\_nameAgePos,idx\_staffs\_name | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+** |

7、注意null/not null对索引的可能影响，当前数据库表中的name字段定义时为not null。

|  |
| --- |
| mysql> EXPLAIN SELECT \* FROM staffs WHERE NAME IS NULL;  **+----+-------------+-------+------+---------------+------+---------+------+------+------------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+-------+------+---------------+------+---------+------+------+------------------+**  **| 1 | SIMPLE | NULL | NULL | NULL | NULL | NULL | NULL | NULL | Impossible WHERE |**  **+----+-------------+-------+------+---------------+------+---------+------+------+------------------+**  mysql> EXPLAIN SELECT \* FROM staffs WHERE NAME IS NOT NULL;  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | idx\_staffs\_nameAgePos,idx\_staffs\_name | NULL | NULL | NULL | 4 | Using where |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+** |

创建数据库表staffs2，并将name字段定义为可以为null，其他内容都与staffs数据库表相同。

|  |
| --- |
| CREATE TABLE staffs2 (  id INT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR (24) ,  age INT NOT NULL DEFAULT 0 COMMENT '年龄',  pos VARCHAR (20) NOT NULL DEFAULT '' COMMENT '职位',  add\_time TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP COMMENT '入职时间'  ) CHARSET utf8 COMMENT '员工记录表' ;  INSERT INTO staffs2(NAME,age,pos,add\_time) VALUES('z3',22,'manager',NOW());  ALTER TABLE staffs2 ADD INDEX idx\_staffs\_nameAgePos(name, age, pos);  ALTER TABLE staffs2 ADD INDEX idx\_staffs\_name(name); |

|  |
| --- |
| mysql> EXPLAIN SELECT \* FROM staffs2 WHERE name IS NULL;  **+----+-------------+---------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+---------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| 1 | SIMPLE | staffs2 | ref | idx\_staffs\_nameAgePos,idx\_staffs\_name | idx\_staffs\_nameAgePos | 75 | const | 1 | Using where |**  **+----+-------------+---------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  mysql> EXPLAIN SELECT \* FROM staffs2 WHERE name IS NOT NULL;  **+----+-------------+---------+-------+---------------------------------------+-----------------------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+---------+-------+---------------------------------------+-----------------------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs2 | range | idx\_staffs\_nameAgePos,idx\_staffs\_name | idx\_staffs\_nameAgePos | 75 | NULL | 1 | Using where |**  **+----+-------------+---------+-------+---------------------------------------+-----------------------+---------+------+------+-------------+** |

8、like以通配符开头('%abc...')mysql索引失效会变成全表扫描的操作

|  |
| --- |
| mysql> explain select \* from staffs where name='July';  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  **| 1 | SIMPLE | staffs | ref | idx\_staffs\_nameAgePos,idx\_staffs\_name | idx\_staffs\_nameAgePos | 74 | const | 1 | Using where |**  **+----+-------------+--------+------+---------------------------------------+-----------------------+---------+-------+------+-------------+**  mysql> explain select \* from staffs where name like '%July%';  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  mysql> explain select \* from staffs where name like '%July';  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | NULL | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------+------+---------+------+------+-------------+**  mysql> explain select \* from staffs where name like 'July%';  **+----+-------------+--------+-------+---------------------------------------+-----------------------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+-------+---------------------------------------+-----------------------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | range | idx\_staffs\_nameAgePos,idx\_staffs\_name | idx\_staffs\_nameAgePos | 74 | NULL | 1 | Using where |**  **+----+-------------+--------+-------+---------------------------------------+-----------------------+---------+------+------+-------------+** |

9、字符串不加单引号索引失效

|  |
| --- |
| mysql> EXPLAIN SELECT \* FROM staffs WHERE name = 917;  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | idx\_staffs\_nameAgePos,idx\_staffs\_name | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+---------------------------------------+------+---------+------+------+-------------+** |

10、少用or,用它来连接时会索引失效

|  |
| --- |
| mysql> explain select \* from staffs where name='July' or name='z3';  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | range | idx\_staffs\_nameAgePos | idx\_staffs\_nameAgePos | 74 | NULL | 2 | Using where |**  **+----+-------------+--------+-------+-----------------------+-----------------------+---------+------+------+-------------+**  **mysql> explain select \* from staffs where name='July' or age=23;**  **+----+-------------+--------+------+-----------------------+------+---------+------+------+-------------+**  **| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |**  **+----+-------------+--------+------+-----------------------+------+---------+------+------+-------------+**  **| 1 | SIMPLE | staffs | ALL | idx\_staffs\_nameAgePos | NULL | NULL | NULL | 12 | Using where |**  **+----+-------------+--------+------+-----------------------+------+---------+------+------+-------------+**  mysql> select \* from staffs where name='July' or name='z3';  **+----+------+-----+---------+---------------------+**  **| id | NAME | age | pos | add\_time |**  **+----+------+-----+---------+---------------------+**  **| 2 | July | 23 | dev | 2019-08-20 08:59:40 |**  **| 1 | z3 | 22 | manager | 2019-08-20 08:59:40 |**  **+----+------+-----+---------+---------------------+** |

### 索引失效总结

假设index(a,b,c)

|  |  |
| --- | --- |
| Where语句 | 索引是否被使用 |
| where a = 3 | Y,使用到a |
| where a = 3 and b = 5 | Y,使用到a，b |
| where a = 3 and b = 5 and c = 4 | Y,使用到a,b,c |
| where b = 3 或者 where b = 3 and c = 4 或者 where c = 4 | N |
| where a = 3 and c = 5 | 使用到a， 但是c不可以，b中间断了 |
| where a = 3 and b > 4 and c = 5 | 使用到a和b， c不能用在范围之后，b断了 |
| where a = 3 and b like 'kk%' and c = 4 | Y,使用到a,b,c |
| where a = 3 and b like '%kk' and c = 4 | Y,只用到a |
| where a = 3 and b like '%kk%' and c = 4 | Y,只用到a |
| where a = 3 and b like 'k%kk%' and c = 4 | Y,使用到a,b,c |

全职匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

LIKE百分写最右，覆盖索引不写\*；

不等空值还有OR，索引影响要注意；

VAR引号不可丢， SQL优化有诀窍。

## 索引实际使用注意事项

1、分页查询很重要，如果查询数据量超过 30%，MySQL 不会使用索引。

2、单表索引数不超过 5 个、单个索引字段数不超过 5 个。

3、字符串可使用前缀索引，前缀长度控制在 5-8 个字符。

4、字段唯一性太低，增加索引没有意义，如：是否删除、性别。

5、合理使用覆盖索引，如下所示，login\_name, nick\_name两个字段建立组合索引，比login\_name简单索引要更快。

|  |
| --- |
| select login\_name, nick\_name from member where login\_name = ? |

6、由于!=可能会使索引失效，因此尽量使用<>来替换!=

7、可以使用union替换or来避免无法使用组合索引的情况

8、查询条件列运算会导致索引失效，因此尽量避免在查询列上进行运算

9、如果不查询表中所有的列，避免使用 SELECT \*，它会进行全表扫描，不能有效利用索引。

10、like模糊查询，%在左边时无法命中索引

# 事务

## ACID概念

事务是由一组SQL语句组成的逻辑处理单元，事务具有以下4个属性，通常简称为事务的ACID属性。

### 原子性（Atomicity）

事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。

### 一致性（Consistent）

在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（如B树索引或双向链表）也都必须是正确的。

### 隔离性（Isolation）

数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。

### 持久性（Durable）

事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

## 并发事务问题

### 更新丢失（lost update）

当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题－－最后的更新覆盖了由其他事务所做的更新。

例如，两个程序员修改同一java文件。每程序员独立地更改其副本，然后保存更改后的副本，这样就覆盖了原始文档。最后保存其更改副本的编辑人员覆盖前一个程序员所做的更改。

如果在一个程序员完成并提交事务之前，另一个程序员不能访问同一文件，则可避免此问题。

### 脏读（dirty reads）

一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象地叫做”脏读”。

一句话：事务A读取到了事务B已修改但尚未提交的的数据，还在这个数据基础上做了操作。此时，如果B事务回滚，A读取的数据无效，不符合一致性要求。

### 不可重复读（non-repeatable reads）

一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现其读出的数据已经发生了改变、或某些记录已经被删除了！这种现象就叫做“不可重复读”。

一句话：事务A读取到了事务B已经提交的修改数据，不符合隔离性。

### 幻读（phantom reads）

一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象就称为“幻读”。

一句话：事务A读取到了事务B体提交的新增数据，不符合隔离性。

幻读和脏读有点类似，脏读是事务B里面修改了数据，幻读是事务B里面新增了数据。

## 事务的隔离级别

脏读、不可重复读和幻读其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。



数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上 “串行化”进行，这显然与“并发”是矛盾的。同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。

查看当前数据库的事务隔离级别：show variables like 'tx\_isolation';

## 事务相关语句

1.查看当前会话隔离级别

select @@tx\_isolation;

2.查看系统当前隔离级别

select @@global.tx\_isolation;

3.设置当前会话隔离级别

set session transaction isolation level read uncommitted;

set session transaction isolation level read committed;

set session transaction isolation level repeatable read;

set session transaction isolation level serializable;

4.开启事务

start transaction;

5.提交事务

commit;

6.回滚事务

rollback;

# 锁

## 锁的概述

锁是计算机协调多个进程或线程并发访问某一资源的机制。

在数据库中，除传统的计算资源（如CPU、RAM、I/O等）的争用以外，数据也是一种供许多用户共享的资源。如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题，锁冲突也是影响数据库并发访问性能的一个重要因素。从这个角度来说，锁对数据库而言显得尤其重要，也更加复杂。

打个比方，到淘宝上买一件商品，商品只有一件库存，这时还有另一个人买，如何解决是谁买到的问题？这里要用到事务，先从库存表中取出物品数量，然后插入订单，付款后插入付款表信息，然后更新商品数量。在这个过程中，使用锁可以对有限的资源进行保护，解决隔离和并发的矛盾。

## MySQL锁的分类

选择锁时需要根据开销、加锁速度、死锁、粒度、并发性能等来考虑，只能就具体应用的特点来说哪种锁更合适。

### 从对数据操作的类型分类

#### 读锁（共享锁）

针对同一份数据，多个读操作可以同时进行而不会互相影响。

#### 写锁（排它锁）

当前写操作没有完成前，它会阻断其他写锁和读锁。

### 从对数据操作的粒度分类

为了尽可能提高数据库的并发度，每次锁定的数据范围越小越好，理论上每次只锁定当前操作的数据的方案会得到最大的并发度，但是管理锁是很耗资源的事情（涉及获取，检查，释放锁等动作），因此数据库系统需要在高并发响应和系统性能两方面进行平衡，这样就产生了“锁粒度（Lock granularity）”的概念。根据对数据操作的粒度不同来将锁分为页锁、表锁和行锁。

#### 表锁

偏向MyISAM存储引擎，开销小，加锁快；无死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。

**表锁分析--建表SQL**

create table mylock(

id int not null primary key auto\_increment,

name varchar(20)

)engine myisam;

insert into mylock(name) values('a');

insert into mylock(name) values('b');

insert into mylock(name) values('c');

insert into mylock(name) values('d');

insert into mylock(name) values('e');

select \* from mylock;

**查看加过的锁的表**

show open tables;

**手动增加表锁**

lock table 表名字 read(write)，表名字2 read(write)，其它;

lock table mylock read;[lock table mylock write;]

show open tables;

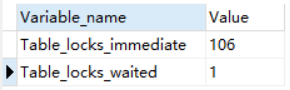


**释放表锁**

unlock tables;

**分析系统上的表锁定**

show status like 'table%';



这里有两个状态变量记录MySQL内部表级锁定的情况，两个变量说明如下：

Table\_locks\_immediate：产生表级锁定的次数，表示可以立即获取锁的查询次数，每立即获取锁值加1 ；

Table\_locks\_waited：出现表级锁定争用而发生等待的次数(不能立即获取锁的次数，每等待一次锁值加1)，此值高则说明存在着较严重的表级锁争用情况；

此外，Myisam的读写锁调度是写优先，这也是myisam不适合做写为主表的引擎。因为写锁后，其他线程不能做任何操作，大量的更新会使查询很难得到锁，从而造成永远阻塞。

##### 表共享读锁（Table Read Lock）

|  |  |
| --- | --- |
| **session\_1** | **session\_2** |
| 获得表mylock的READ锁定 | 连接终端 |
| 当前session可以查询该表记录 | 其他session也可以查询该表的记录 |
| 当前session不能查询其它没有锁定的表 | 其他session可以查询或者更新未锁定的表 |
| 当前session中插入或者更新锁定的表都会提示错误 | 其他session插入或者更新锁定表会一直等待获得锁，即session\_1执行释放锁的操作 |
| 释放锁 | Session2获得锁，插入操作完成： |

##### 表独占写锁（Table Write Lock）

|  |  |
| --- | --- |
| **session\_1** | **session\_2** |
| 获得表mylock的WRITE锁定 | 待Session1开启写锁后，session2再连接终端 |
| 当前session对锁定表的查询+更新+插入操作都可以执行 | 其他session对锁定表的查询被阻塞，需要等待锁被释放  备注：如果可以，请换成不同的id来进行测试，因为mysql聪明有缓存，第2次的条件会从从缓存取得，影响锁效果演示 |
| 释放锁 | Session2获得锁，查询返回 |

MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行增删改操作前，会自动给涉及的表加写锁。 对MyISAM表进行操作会有以下情况：

1、对MyISAM表的读操作（加读锁），不会阻塞其他进程对同一表的读请求，但会阻塞对同一表的写请求。只有当读锁释放后，才会执行其它进程的写操作。

2、对MyISAM表的写操作（加写锁），会阻塞其他进程对同一表的读和写操作，只有当写锁释放后，才会执行其它进程的读写操作。

简而言之，就是读锁会阻塞写，但是不会堵塞读。而写锁则会把读和写都堵塞。

#### 行锁

偏向InnoDB存储引擎，开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。InnoDB与MyISAM的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁。

通过select \* from test\_innodb\_lock where a = 8 for update;来锁定该行，其他会话的操作都会被阻塞，直到锁定该行的本会话提交commit。

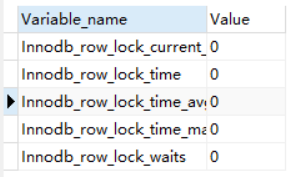
Innodb存储引擎由于实现了行级锁定，虽然在锁定机制的实现方面所带来的性能损耗可能比表级锁定会要更高一些，但是在整体并发处理能力方面要远远优于MyISAM的表级锁定的。当系统并发量较高的时候，Innodb的整体性能和MyISAM相比就会有比较明显的优势了。

但是，Innodb的行级锁定同样也有其脆弱的一面，使用不当时可能会让Innodb的整体性能表现不仅不能比MyISAM高，甚至可能会更差。

##### 行锁分析

通过检查InnoDB\_row\_lock状态变量来分析系统上的行锁的争夺情况

show status like 'innodb\_row\_lock%';



对各个状态量的说明如下：

Innodb\_row\_lock\_current\_waits：当前正在等待锁定的数量；

Innodb\_row\_lock\_time：从系统启动到现在锁定总时间长度；

Innodb\_row\_lock\_time\_avg：每次等待所花平均时间；

Innodb\_row\_lock\_time\_max：从系统启动到现在等待最常的一次所花的时间；

Innodb\_row\_lock\_waits：系统启动后到现在总共等待的次数；

对于这5个状态变量，比较重要的主要是

Innodb\_row\_lock\_time\_avg（等待平均时长），

Innodb\_row\_lock\_waits（等待总次数）

Innodb\_row\_lock\_time（等待总时长）这三项。

尤其是当等待次数很高，而且每次等待时长也不小的时候，我们就需要分析系统中为什么会有如此多的等待，然后根据分析结果着手指定优化计划。

**行锁分析--建表SQL**

create table test\_innodb\_lock (a int(11),b varchar(16))engine=innodb;

insert into test\_innodb\_lock values(1,'b2');

insert into test\_innodb\_lock values(3,'3');

insert into test\_innodb\_lock values(4,'4000');

insert into test\_innodb\_lock values(5,'5000');

insert into test\_innodb\_lock values(6,'6000');

insert into test\_innodb\_lock values(7,'7000');

insert into test\_innodb\_lock values(8,'8000');

insert into test\_innodb\_lock values(9,'9000');

insert into test\_innodb\_lock values(1,'b1');

create index test\_innodb\_a\_ind on test\_innodb\_lock(a);

create index test\_innodb\_lock\_b\_ind on test\_innodb\_lock(b);

select \* from test\_innodb\_lock;

##### 行锁基本演示

|  |  |
| --- | --- |
| **Session\_1** | **Session\_2** |
| set autocommit=0; | set autocommit=0; |
| 更新但是不提交，没有手写commit;  update test\_innodb\_lock set b = ‘b1’ where a =1; | Session\_2被阻塞，只能等待  update test\_innodb\_lock set b = ‘b1’ where a =1; |
| 提交更新  commit; | 解除阻塞，更新正常进行  update test\_innodb\_lock set b = ‘b1’ where a =1; |
|  | commit命令执行 |

##### 无索引行锁升级为表锁

|  |  |
| --- | --- |
| **Session\_1** | **Session\_2** |
| 正常情况，各自锁定各自的行，互相不影响，一个2000另一个3000 |  |
| update test\_innodb\_lock set b = ‘2’ where b = ‘2000’; | update test\_innodb\_lock set b = ‘2’ where a = 3; |
| 由于在column字段b上面建了索引，如果没有正常使用，会导致行锁变表锁 |  |
| 比如没加单引号导致索引失效，行锁变表锁  update test\_innodb\_lock set b = ‘2’ where b = 2000; | 被阻塞，等待。只到Session\_1提交后才阻塞解除，完成更新 |

##### 间隙锁危害

|  |  |
| --- | --- |
| **Session\_1** | **Session\_2** |
| update test\_innodb\_lock set b = a \* 20 where a>1 and a<5; | 阻塞产生，暂时不能插入  insert into test\_innodb\_lock values(2,‘200’); |
| commit; | 阻塞解除，完成插入 |

当用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁）。

因为Query执行过程中通过过范围查找时，会锁定整个范围内所有的索引键值，即使这个键值并不存在。

间隙锁有一个比较致命的弱点，就是当锁定一个范围键值之后，即使某些不存在的键值也会被无辜的锁定，而造成在锁定的时候无法插入锁定键值范围内的任何数据。在某些场景下这可能会对性能造成很大的危害

##### 行锁优化

1、尽可能让所有数据检索都通过索引来完成，避免无索引行锁升级为表锁。

2、合理设计索引，尽量缩小锁的范围

3、尽可能较少检索条件，避免间隙锁

4、尽量控制事务大小，减少锁定资源量和时间长度

5、尽可能低级别事务隔离

#### 页锁

页锁开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。因此了解一下即可。

# 分库分表

## 分库

## 分表

# 应用注意事项

1、充分利用但不滥用索引，须知索引也消耗磁盘和 CPU。

2、不推荐使用数据库函数格式化数据，交给应用程序处理。

3、不推荐使用外键约束，用应用程序保证数据准确性。

4、写多读少的场景，不推荐使用唯一索引，用应用程序保证唯一性。

5、适当冗余字段，尝试创建中间表，用应用程序计算中间结果，用空间换时间。

6、不允许执行极度耗时的事务，配合应用程序拆分成更小的事务。

7、预估重要数据表（比如订单表）的负载和数据增长态势，提前优化。

8、分批处理，大量的不带分页参数的查询或者影响大量数据的 update 和 delete 操作可能会堵死其他SQL，因此需要分批处理。

# 安装与卸载

## Windows

### Windows安装

**安装步骤如下**

->选择Custom自定义安装

->将MySQL Server和Server data files的目录选择到同一个目录下，便于以后删除

->开始安装，安装完成后勾选Instance Configuration Wizard配置向导进行配置

->选择Detailed Configuration详细配置

->选择Developer Machine

->选择Multifunctional Database

->选择Manual Setting将数值设置为20或者直接勾选Decision Support(DSS)/OLAP

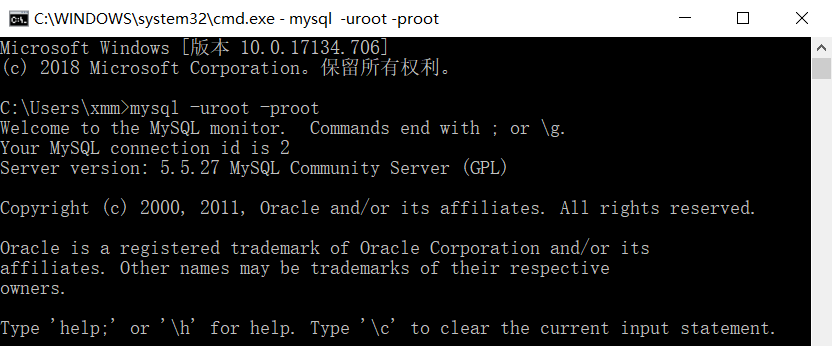
->默认设置端口值为3306，并添加防火墙例外

->将字符集设置为UTF-8，或者直接勾选Best Support For Multilingualism

->勾选服务器并设置服务器名为MySQL，在勾选添加到系统路径选项

**测试是否安装成功**

打开cmd命令行窗口输入：mysql -uroot -p你的密码，若安装成功则显式下图：

****

### Windows卸载

在控制面板的卸载程序中卸载，除了删除Mysql程序之外，还要删除Mysql之前的Data数据。在原来安装的Mysql程序目录下的my.ini配置文件，找到如下的datadir的值，删除指定的路径的目录。

另外还要删除注册表中mysql相关选项，在开始中输入regedit，找到如下三个目录中的MySQL文件夹并删除。

 HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services

 HKEY\_LOCAL\_MACHINE\SYSTEM\ControlSet001\Services

 HKEY\_LOCAL\_MACHINE\SYSTEM\ControlSet002\Services

也可以在注册表中直接搜索MySQL，将所有与其相关的文件删除。完成以上步骤后重启电脑，然后再重新安装MySQL数据库。

### Windows启动和关闭

界面操作

控制面板->管理工具->服务->关闭或启动MySQL

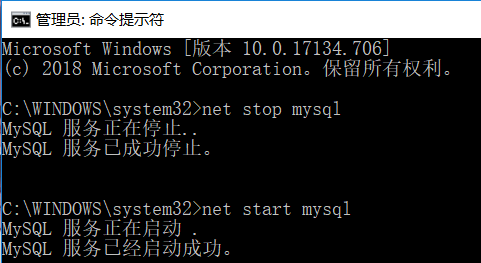
运行搜索 services.msc打开服务->关闭或启动MySQL

命令行操作

必须要以**管理员身份**运行cmd

停止mysql服务器的命令： net stop mysql

启动mysql服务器的命令： net start mysql



### 重置root密码

#### 方法一

在my.ini的[mysqld]字段加入：skip-grant-tables

重启mysql服务，这时的mysql不需要密码即可登录数据库，然后进入mysql

mysql>use mysql;

mysql>更新 update user set password=password('新密码') WHERE User='root';

mysql>flush privileges;

运行之后最后去掉my.ini中的skip-grant-tables，重启mysqld即可。

#### 方法二

不使用修改my.ini重启服务的方法，通过非服务方式加skip-grant-tables运行mysql来修改mysql密码

停止mysql服务

打开命令行窗口，在bin目录下使用mysqld-nt.exe启动，即在命令行窗口执行: mysqld-nt --skip-grant-tables

然后另外打开一个命令行窗口，登录mysql，此时无需输入mysql密码即可进入。

按以上方法修改好密码后,关闭命令行运行mysql的那个窗口，此时即关闭了mysql，如果发现mysql仍在运行的话可以结束掉对应进程来关闭。

启动mysql服务

## Linux

### Linux安装

1、安装MySQL服务端

|  |
| --- |
| [root@xmm opt]# rpm -ivh MySQL-server-5.5.52-1.el6.x86\_64.rpm  警告：MySQL-server-5.5.52-1.el6.x86\_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  准备中... ################################# [100%]  正在升级/安装...  1:MySQL-server-5.5.52-1.el6 ( 1################################# [100%]  190827 21:10:12 [Note] /usr/sbin/mysqld (mysqld 5.5.52) starting as process 2699 ...  190827 21:10:12 [Note] /usr/sbin/mysqld (mysqld 5.5.52) starting as process 2706 ...  PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !  To do so, start the server, then issue the following commands:  /usr/bin/mysqladmin -u root password 'new-password'  /usr/bin/mysqladmin -u root -h xmm password 'new-password'  Alternatively you can run:  /usr/bin/mysql\_secure\_installation  which will also give you the option of removing the test  databases and anonymous user created by default. This is  strongly recommended for production servers.  See the manual for more instructions.  Please report any problems at http://bugs.mysql.com/ |

2、安装MySQL客户端

|  |
| --- |
| [root@xmm opt]# rpm -ivh MySQL-client-5.5.52-1.el6.x86\_64.rpm  警告：MySQL-client-5.5.52-1.el6.x86\_64.rpm: 头V3 DSA/SHA1 Signature, 密钥 ID 5072e1f5: NOKEY  准备中... ################################# [100%]  正在升级/安装...  1:MySQL-client-5.5.52-1.el6 ( 2################################# [100%] |

3、查看MySQL服务并启动

|  |
| --- |
| [root@xmm opt]# chkconfig | grep mysql  注：该输出结果只显示 SysV 服务，并不包含  原生 systemd 服务。SysV 配置数据  可能被原生 systemd 配置覆盖。  要列出 systemd 服务，请执行 'systemctl list-unit-files'。  查看在具体 target 启用的服务请执行  'systemctl list-dependencies [target]'。  mysql 0:关 1:关 2:开 3:开 4:开 5:开 6:关  [root@xmm opt]# service mysql status  ERROR! MySQL is not running  [root@xmm opt]# service mysql start  Starting MySQL.. SUCCESS! |

4、设置root账号密码，这里设置的密码为root

|  |
| --- |
| [root@xmm opt]# mysqladmin -u root password  New password:  Confirm new password: |

5、登陆MySQL服务器

|  |
| --- |
| [root@xmm opt]# mysql -u root -p  Enter password:  Welcome to the MySQL monitor. Commands end with ; or \g.  Your MySQL connection id is 2  Server version: 5.5.52 MySQL Community Server (GPL)  Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.  Oracle is a registered trademark of Oracle Corporation and/or its  affiliates. Other names may be trademarks of their respective  owners.  Type 'help;' or '\h' for help. Type '\c' to clear the current input statement. |

#### 其他操作

复制配置文件到/etc目录

|  |
| --- |
| [root@xmm mysql]# cp /usr/share/mysql /etc/my.cnf |

导出sql文件到当前目录下，导出sql文件语句格式如下：

mysqldump -u 数据库链接用户名 -p 目标数据库 > 存储的文件名

|  |
| --- |
| [root@xmm mysql]# mysqldump -u root -p test > test.sql  Enter password: |

导入sql文件，导入sql文件语句格式如下：

mysql -u 用户名 -p 数据库名 < 数据库名.sql

|  |
| --- |
| [root@xmm mysql]# mysql -u root -p test < test.sql  Enter password: |

#### 远程访问MySQL

1、先 ping 一下数据库服务器的ip 地址确认网络畅通。

2、关闭数据库服务的防火墙

service iptables stop

3、 确认Mysql中已经有可以通过远程登录的账户

select \* from mysql.user where user='li4' and host='%';

4、如果没有用户,先执行如下命令：

grant all privileges on \*.\* to li4@'%' identified by '123123';

可能出现的异常

1、1130 - Host is not allowed to connect to this MySQL server

解决方式：

|  |
| --- |
| mysql> use mysql  mysql> select host from user;  +-----------+  | host |  +-----------+  | 127.0.0.1 |  | ::1 |  | localhost |  | localhost |  | xmm |  | xmm |  +-----------+  mysql> update user set host='%' where user='root';  ERROR 1062 (23000): Duplicate entry '%-root' for key 'PRIMARY'  mysql> flush privileges;  Query OK, 0 rows affected (0.00 sec) |

### Linux卸载

1、查看当前Linux系统是否安装了MySQL

|  |
| --- |
| [root@xmm ~]# rpm -qa | grep -i mysql  MySQL-server-5.5.52-1.el6.x86\_64  MySQL-client-5.5.52-1.el6.x86\_64 |

2、卸载之前安装的MySQL

|  |
| --- |
| [root@xmm ~]# rpm -e --nodeps MySQL-server-5.5.52-1.el6.x86\_64  [root@xmm ~]# rpm -e --nodeps MySQL-client-5.5.52-1.el6.x86\_64 |

如果提示依赖包错误，则使用以下命令尝试

rpm -ev MySQL-client-5.5.25a-1.rhel5 --nodeps

如果提示错误：error: %preun(xxxxxx) scriptlet failed, exit status 1则用以下命令尝试：

rpm -e --noscripts MySQL-client-5.5.25a-1.rhel5

3、查找并删除MySQL的目录、文件和库，注意还有etc目录下的my.cnf配置文件需要删除

|  |
| --- |
| [root@xmm ~]# find / -name mysql  /var/lib/mysql  /var/lib/mysql/mysql  /etc/selinux/targeted/active/modules/100/mysql  /usr/lib64/mysql  [root@xmm ~]# rm -rf /var/lib/mysql  [root@xmm ~]# rm -rf /var/lib/mysql/mysql  [root@xmm ~]# rm -rf /etc/selinux/targeted/active/modules/100/mysql  [root@xmm ~]# rm -rf /usr/lib64/mysql  [root@xmm ~]# rm -rf /etc/my.cnf |

# 扩展

## 其他类型数据库

**常用数据库**

Oracle ：它是 Oracle 甲骨文 公司的大型关系型数据库，它是收费的。

DB2 ：IBM公司的数据库，它是收费的。

SQLServer ：微软数据库。

Sybase ：Sybase公司的。

Mysql ：早期瑞典一个公司发明，后期被sun公司收购，后期被Oracle。mysql数据库分为：企业版（收费）和 社区版（免费）

Java开发应用程序主要使用的数据库：MySQL、Oracle、DB2。

|  |  |  |
| --- | --- | --- |
| **分类** | **数据库** | **特性** |
| 键值型 | Memcache | 用于内容缓存，大量数据的高访问负载 |
| 键值型 | Redis | 用于内容缓存，较Memcache支持更多数据结构，并能持久化数据 |
| 列式存储 | HBase | Hadoop体系核心数据库，海量结构化数据存储，大数据必备 |
| 文档型 | MongoDb | 知名文档型数据库，也可用户缓存 |
| 文档型 | CouchDB | Apache开源项目，专注于易用性，支持REST API |
| 文档型 | SequoiaDB | 国内知名文档型数据库 |
| 图形 | Neo4J | 用于社交网络构建关系图谱，推荐系统等 |

# 习题

|  |
| --- |
| -- 查询全部学生信息  **select** \* **from** t\_student;  -- 查询所有男学生信息  **select** \* **from** t\_student **where** `sex` = '男';  -- 统计男、女生总人数  **select** count(\*) '总人数',`sex` '性别' **from** t\_student **group** **by** `sex`;  -- 查询选修课成绩 大于 > 60 分的学生信息  **select** \* **from** t\_student **where** id **in**(  -- 先获取在score表中成绩大于60的学生id，并去除重复的id  **select** **distinct** s\_id **from** t\_score **where** score > 60);  -- 查询所有选修了 韩顺平 老师的所有学生信息  **select** \* **from** t\_student **where** id **in** (  **select** s\_id **from** t\_score **where** c\_id = (  **select** id **from** t\_course **where** t\_id = (  **select** id **from** t\_teacher **where** name = '韩顺平')));  -- 查询北京学生所有选修课信息（学生编号，学生姓名、课程名称、成绩）。  **select**  t\_student.id '学生编号',t\_student.name '学生姓名',t\_course.name '课程名称',t\_score.score '成绩'  **from**  t\_score,t\_course,t\_student  **where**  t\_student.id = t\_score.s\_id  **and**  t\_course.id = t\_score.c\_id  **and**  t\_student.city = '北京';    -- 查询"雷"姓老师的数量  **select** count(\*) **from** t\_teacher **where** `name` **like** '雷%';  -- 查询没有选修了 雷丰阳 老师的所有学生信息  **select** \* **from** t\_student **where** id **not** **in**(  **select** s\_id **from** t\_score **where** c\_id = (  **select** id **from** t\_course **where** id = (  **select** id **from** t\_teacher **where** `name` = '雷丰阳')));  -- 查询学过 java 或 C++ 课程的学生  **select** \* **from** t\_student **where** id **in** (  **select** **distinct** s\_id **from** t\_score **where** c\_id **in** (  **select** id **from** t\_course **where** name = 'Java' **or** name = 'C++'));  -- 查询学过 java 和 C++ 课程的学生  **select** \* **from** t\_student **where**  id **in** (  **select** **distinct** s\_id **from** t\_score **where** c\_id **in** (  **select** id **from** t\_course **where** name = 'Java'))  **and**  id **in** (  **select** **distinct** s\_id **from** t\_score **where** c\_id **in** (  **select** id **from** t\_course **where** name = 'C++'));  -- 查询平均成绩大于等于60分的同学的学生编号和学生姓名和平均成绩  **select**  t\_student.`id` '学生编号',t\_student.`name` '学生姓名',`avgscore` '平均成绩'  **from**  t\_student,(**select** s\_id,avg(score) 'avgscore'**from** t\_score **group** **by** s\_id **having** avg(score) > 60) t\_t  **where**  t\_student.`id` = t\_t.s\_id ;  -- 查询所有Java课程比C++课程学的好的学生信息  **select**  a.\*,b.`c\_id`,b.`score`  **from**  t\_score a,t\_score b  **where**  a.`s\_id` = b.`s\_id`  **and**  a.`c\_id` = (**select** id **from** t\_course **where** name = 'Java')  **and**  b.`c\_id` = (**select** id **from** t\_course **where** name = 'C++')  **and**  a.score > b.score; |

# default