## Mysql启动

mysql是CS架构, mysql服务器进程的默认名称是mysqld, mysql客户端进程的默认名称是mysql

1. unix下启动mysql服务器
   1. mysqld-----二进制可执行文件
   2. mysqld\_safe-----一个启动脚本，会间接地调用mysqld，还顺便启动了另外一个监控进程，这个监控进程在服务器进程挂了的时候，可以帮助重启它。
   3. mysql.server----一个启动脚本，间接调用mysqld\_safe, 可指定参数
2. unix下启动mysql客户端
   1. mysql (-h) -u -p

## 连接过程

1. TCP

监听3306端口

1. 命名管道

服务器程序启动时加上--enable\_named\_pipe， 客户端程序启动时加上--pipe

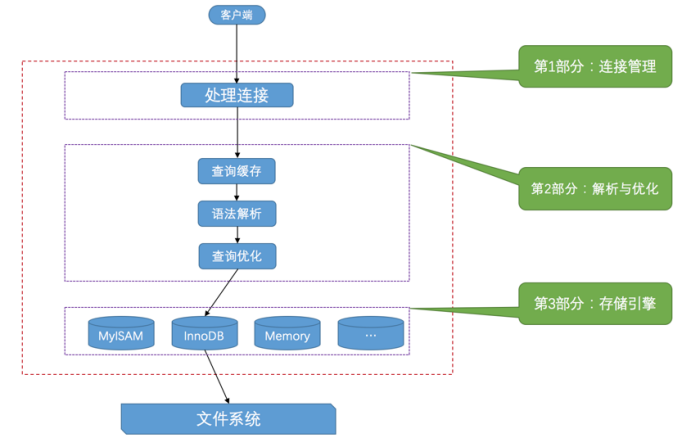
1. 共享内存

服务器程序启动时加上--shared-memory, (客户端程序加上proocol=memory)

1. unix域套接字

服务器程序默认监听unix套接字文件为/tmp/mysql.sock, 客户端程序启动时加上--protocol=socket

## 服务端处理流程



1. 连接管理

每当有一个客户端进程连接到服务器进程时，服务器进程就会创建一个线程来处理与这个客户端的交互，但在客户端断开连接后，线程不会被销毁而是被缓存以再利用。

1. 解析和优化

查询缓存(从mysql5.7.20开始，已经不推荐使用，再mysql8.0中已经被删除): mysql服务器会将最近处理过的请求和结果缓存起来，但如果两次请求再字符上有任何的不同（比如大小写，空格）以及请求包含一些函数、系统表都会导致缓存不被命中。mysql的缓存系统会监测表，如果对表使用了ddl、dml语句，那么该表的所有高速缓存都将被删除。

1. 语法解析

判断请求的语法是否正确，并且将查询的表和查询条件都提出来放在mysql服务器内部使用的一些数据结构上。

1. 查询优化

由于请求语句的执行效率可能不高，所以需要优化程序对语句进行优化，在此过程中会产生一个执行计划，可通过explain语句来查看。

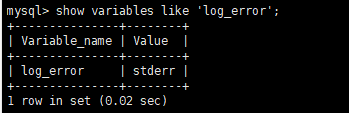
1. 存储引擎

mysql服务器把数据的存储和提取都存在了存储引擎模块里，存储引擎负责在物理上怎么表示数据记录、怎么从表读取数据，怎么写入数据等等。运用最广泛的存储引擎：InonoDB，第二是:MyISAM, 其余的存储引擎就不再概述, 可通过show engines命令查看支持的存储引擎和其特性

## mysql中的日志文件

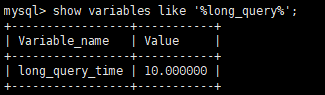
1. 错误日志

错误日志对mysqld进程的启动、运行、关闭过程进行了记录，通过查看相关变量可获取相关信息

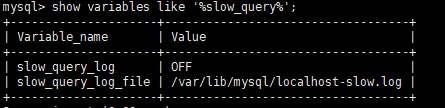


1. 慢查询日志

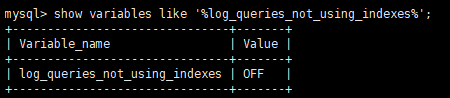
慢查询日志用来记录sql响应时间超过阈值的语句。阈值可见参数long\_query\_time



查看慢查询日志



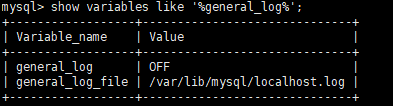
同时，参数log\_queries\_not\_using\_indexes 控制如果运行的sql语句没有使用索引，那么也会被记录到慢查询日志文件中



慢查询日志分析工具: mysqldumpslow

1. 查询日志

查询日志记录了mysqld运行时处理的所有sql语句信息， 不论这个语句是否得到了正确的执行。



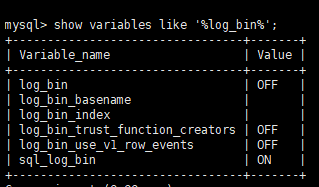
1. 二进制日志

二进制日志记录对数据库所有执行更改的操作，不包括select和show这些。二进制日志有如下几种作用:

恢复: 可以通过二进制日志进行point-in-time的恢复

赋值：通过复制和执行二进制日志使得主从数据库进行实时同步

审计：通过二进制日志中的信息来进行审计，判断是否有对数据库进行注入攻击



## Mysql设置

一般在/etc/my.cnf中设置。

global和session的区别:

set session/global .....

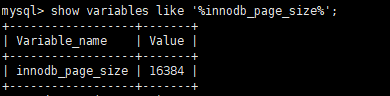
session 只对当前会话生效

global 对当前mysqld生效，重启失效

# InnoDB存储引擎下的索引

## InnoDB页简介

InnoDB的数据都是存在磁盘上的，但是处理数据的过程却又是发送在内存中的，那就需要不断地把磁盘上的数据加载到内存中，执行写入和修改请求时，还要不断地把内存上的数据刷新到磁盘中，这其中就存在磁盘速度特别慢的问题，如果一条一条记录地读，一条一条记录地写，那磁盘寻道时间开销将是非常大的。InnoDB存储引擎将数据划为若干页，以页作为磁盘和内存之间交互的基本单位，大小一般为16KB。



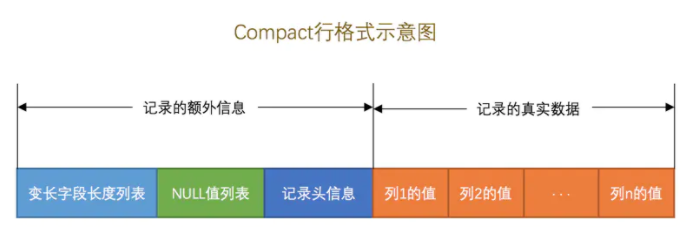
这样就一次最少从磁盘读16kb内容，一次最少刷新16kb内容。

由于客户端平时是以记录为单位向表中插入数据的，所以对于记录在磁盘上的存储格式成为行格式/记录格式，目前有4种不同的行格式：Compact、Redundant、Dynamic、Compressed。

CREATE TABLE 表名 (列的信息) ROW\_FORMAT=行格式名称

ALTER TABLE 表名 ROW\_FORMAT=行格式名称

## Compact行格式简介



一条记录被分为额外信息和真实数据两部分。

1. 变长字段长度列表

在MySQL中支持一些变长数据类型: varchar、varbinary、text、blob， 而拥有这些数据类型的列也被称为变长字段。因为变长字段中存储多少字节的数据是不固定的，所以mysql需要把变长字段中数据占用的字节数也存起来。

1. NULL值列表

因为表中某些列可能会存储NULL值，如果把这些NULL值都放在记录的真实数据中存储会很占地方，所以Compact行格式把这些值为NULL的列统一管理起来，存储到NULL值列表中。如果表中没有允许存储NULL的列，那么NULL值列表也不存在了。

1. 记录头信息

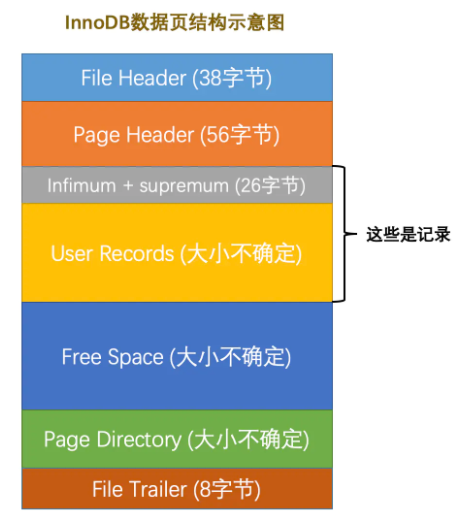
由固定的5字节组成，其中每个位表示不同的信息

对于记录的真实数据中，除了用户自己定义的列之外，mysql还会为每个记录默认地添加一些列(也称为隐藏列): row\_id(6字节， 行ID，唯一标识一条记录)、transaction\_id(6字节， 事务ID)、roll\_pointer(7字节， 回滚指针)。

在InnoDB存储引擎中：优先使用用户定义的主键作为主键，如果用户没有定义则选取一个unique键作为主键，如果连unique键都没有，那么就会默认添加一个row\_id隐藏列作为主键，并且会为每条记录都添加transaction\_id和roll\_pointer两个列。

## InnoDB数据页结构简介

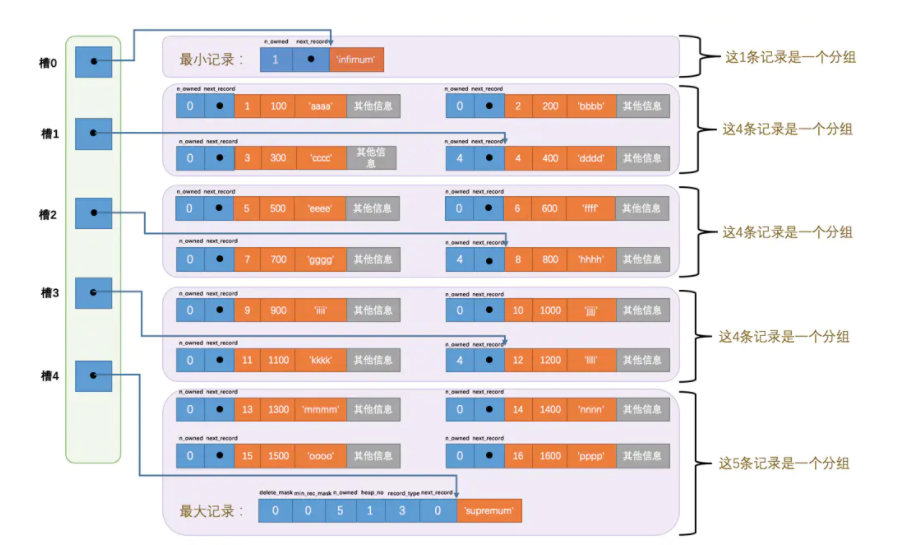
InnoDB中数据页的大小一般为16kb， 16kb的空间又可以被划分为多个部分， 不同的部分有不同的功能。

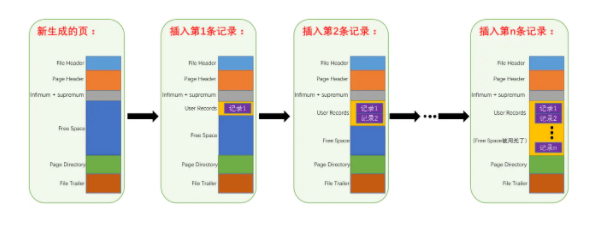


当记录被刷新到磁盘时，就会按照指定的行格式存储到User Records部分。User Records部分一开始空间为0，每当插入一条记录，就会从Free Space分配一个记录大小的空间到User Records。而当Free Space的空间全部被用完时，那就意味着这个页的空间已经使用完了，如果有新记录要插入，就需要去申请新的页了。在每个页中还有两条伪记录：最大记录和最小记录，存储在Infimum+Supermum区域。



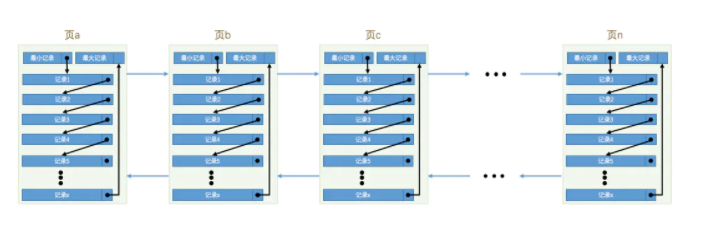
## 页目录



在File Header中描述了对各种类型的页都通用的信息，其中的Fil\_PAGE\_PREV和FIL\_PAGE\_NEXT部分分别表示上一个页的页号以及下一个页的页号。

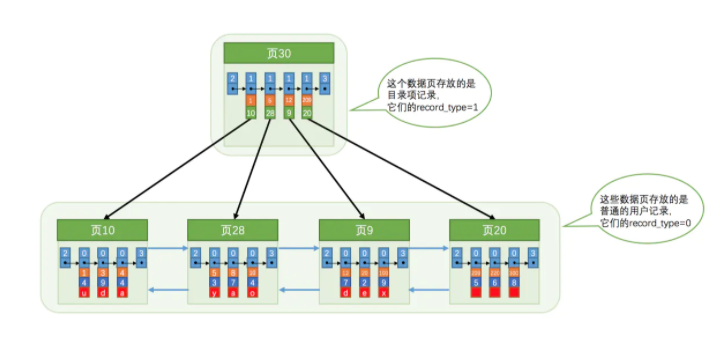
## C:\Users\Administrator\Desktop\批注 2020-08-27 143033.png批注 2020-08-27 143033B+树索引

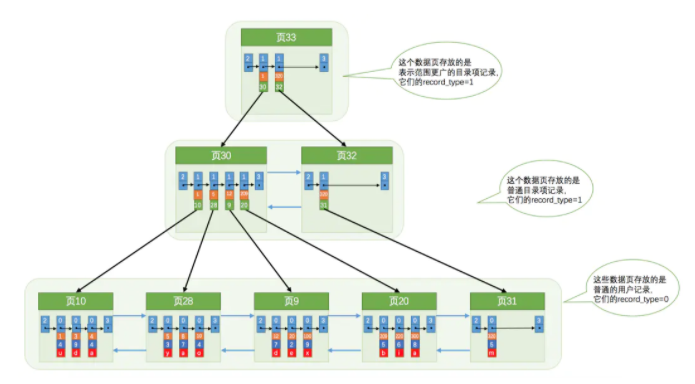
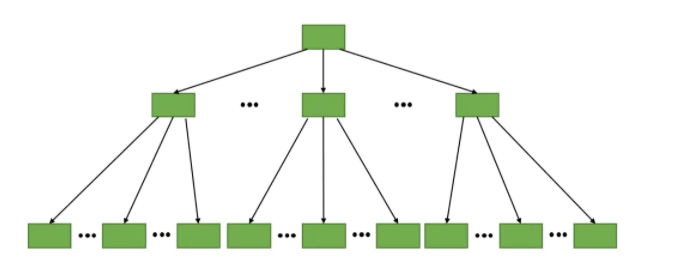
在InnoDB中，各个数据页可以组成一个双向链表，而在数据页中每个记录又会按照主键值从小到大的顺序组成一个单向链表。



如果没有索引的情况下，查找一个记录就需要先定位到记录所在的页，再在页中找到相应的记录。这样的话只能从第一个页开始沿着双向链表一直往下找，再在每个页中先按照二分法找到数据所在的槽，再在槽中按照单向链表查找。

InnoDB的索引方案

InnoDB中使用目录页来管理数据页，目录页结构和数据页结构是一样的，只是通过页头信息中的record\_type属性来区分，0表示普通用户记录，1表示目录项记录，目录页中只有主键值和页号两个列。

如果假设目录页最多存放1000条记录，数据页最多存放100条记录。

那么如果B+树只有一层，最多能存放100条记录。

如果B+树有两层，那么最多能放100000条记录

如果B+树有三层，那么最多能放100000000条记录

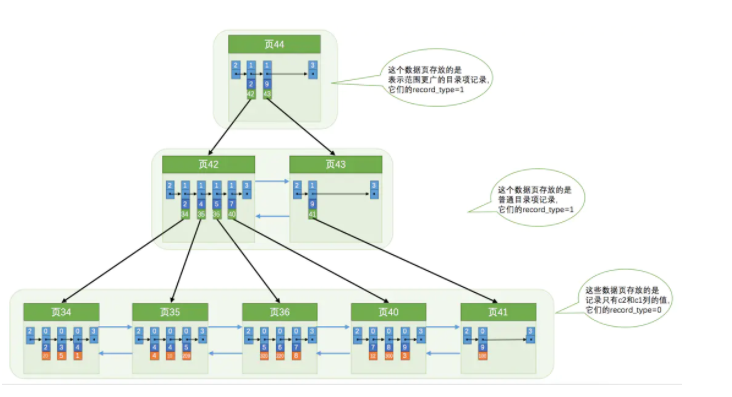
如果B+树有四层，那么最多能放100000000000条记录，并且只需要访问4个页！并且因为页中也有页目录，在页中也可以通过2分法定位。

## 聚簇索引

1. 页中的记录是按照主键大小顺序排成一个单向链表。
2. 页之间也是按照主键大小排成一个双向链表。
3. 叶子节点存储的是完整的用户记录。

在InnoDB存储引擎中，聚簇索引就是数据的存储方式，所有的用户记录都存储在了叶子节点。

## 二级索引

1. 页内的记录是按照指定索引列的大小顺序排成一个单向链表。
2. 页之间也是按照指定索引列大小排成一个双向链表。
3. 叶子节点存储的是指定索引列数据和主键值。

在二级索引中找到指定记录只能获取到主键值，还需要进行回表操纵去聚簇索引取全部的记录数据。

## 联合索引

（c1，c2）

排序规则：先按照c1的大小排序，在c1相同的情况下再按照c2大小排序， 本质上也是一个二级索引。

## 索引的坏处

索引并不是建的越多越好，首先索引有空间代价，每一个索引都对应一颗B+树，每一颗B+树上的每一个节点都是16kb大小的页，而B+树由很多页构成，这会是一片很大的存储空间。其次，索引存在时间代价，每次对表中数据进行增删改操作时，都需要去修改B+树索引，而这些操作可能会对节点和记录的排序造成破坏，就会产生一些记录移位、页面分裂、页面回收等操作，会消耗时间。

所以一个表上索引创建的越多，存储空间就会占用越多，增删改时的性能就越差。

## 索引的适用条件

B+树不是万能的，并不是所有的查询语句都能用到索引。

例子:

CREATE TABLE person\_info( id INT NOT NULL auto\_increment,

name VARCHAR(100) NOT NULL,

birthday DATE NOT NULL,

phone\_number CHAR(11) NOT NULL,

country varchar(100) NOT NULL,

PRIMARY KEY (id),

KEY idx\_name\_birthday\_phone\_number (name, birthday, phone\_number) );

1. 全值匹配

SELECT \* FROM person\_info WHERE name = 'Ashburn' AND birthday = '1990-09-27' AND phone\_number = '15123983239';

因为B+树的数据页和记录先是按照name列的值进行排序的，所以先可以很快定位name列的值是Ashburn的记录位置。

在name列相同的记录里又是按照birthday列的值进行排序的，所以在name列的值是Ashburn的记录里又可以快速定位birthday列的值是'1990-09-27'的记录。

如果很不幸，name和birthday列的值都是相同的，那记录是按照phone\_number列的值排序的，所以联合索引中的三个列都可能被用到。

如果调换几个搜索列的顺序对查询执行过程有影响吗？

比如：SELECT \* FROM person\_info WHERE birthday = '1990-09-27' AND phone\_number = '15123983239' AND name = 'Ashburn';

没有影响，因为mysql有一个叫查询优化器的东西，会分析这些搜索条件并且按照可以使用的索引中列的顺序来决定先使用哪个搜索条件，后使用哪个搜索条件。

1. 匹配左边的列

在搜索引擎中可以不用包含所有联合索引中的列，只需要包含左边的就行。

例如：

SELECT \* FROM person\_info WHERE name = 'Ashburn';

SELECT \* FROM person\_info WHERE name = 'Ashburn' AND birthday = '1990-09-27';

但是下面这条语句就不能使用索引

SELECT \* FROM person\_info WHERE birthday = '1990-09-27';

用不到，因为B+树的数据页和记录先是按照name列的值排序的，在name列的值相同的情况下才使用birthday列进行排序，也就是说name列的值不同的记录中birthday的值可能是无序的。而现在你跳过name列直接根据birthday的值去查找，臣妾做不到呀～ 那如果我就想在只使用birthday的值去通过B+树索引进行查找咋办呢？这好办，你再对birthday列建一个B+树索引就行了。

所以如果我们想使用联合索引里尽可能多的列，搜索条件中的各个列必须是联合索引中从左边连续的列。

例如：

SELECT \* FROM person\_info WHERE name = 'Ashburn' AND phone\_number = '15123983239';

只能用到name列的索引，birthday和phone\_number的索引就用不上了，因为name值相同的记录先按照birthday的值进行排序，birthday值相同的记录才按照phone\_number值进行排序。

1. 匹配列前缀

对于字符串类型的索引列来说，我们只匹配它的前缀也是可以快速定位记录的，比方说我们想查询名字以'As'开头的记录，那就可以这么写查询语句：

SELECT \* FROM person\_info WHERE name LIKE 'As%';

这是因为字符串列是排好序的，先安装字符串的第一个字符进行排序，如果第一个字符相同再按照第二个字符进行排序，如果第二个字符相同则按照第三个字符排序，以此类推。

如果只给出后缀或者中间的某个字符串，比如这样：

SELECT \* FROM person\_info WHERE name LIKE '%As%';

MySQL就无法快速定位记录位置了，因为字符串中间有'As'的字符串并没有排好序，所以只能全表扫描了。

如果一定要用字符串后缀作索引的话，可以选择将字符串反转存入数据库。

1. 匹配范围值

例子:

SELECT \* FROM person\_info WHERE name > 'Asa' AND name < 'Barlow';

由于B+树中的数据页和记录是先按name列排序的，所以我们上边的查询过程其实是这样的：

找到name值为Asa的记录。

找到name值为Barlow的记录。

哦啦，由于所有记录都是由链表连起来的（记录之间用单链表，数据页之间用双链表），所以他们之间的记录都可以很容易的取出来喽～

找到这些记录的主键值，再到聚簇索引中回表查找完整的记录。

注意：如果对多个列同时进行范围查找的话，只有对索引最左边的那个列进行范围查找的时候才能用到B+树索引

例子：

SELECT \* FROM person\_info WHERE name > 'Asa' AND name < 'Barlow' AND birthday > '1980-01-01';

上边这个查询可以分成两个部分：

通过条件name > 'Asa' AND name < 'Barlow'来对name进行范围，查找的结果可能有多条name值不同的记录，

对这些name值不同的记录继续通过birthday > '1980-01-01'条件继续过滤。

这样子对于联合索引idx\_name\_birthday\_phone\_number来说，只能用到name列的部分，而用不到birthday列的部分，因为只有name值相同的情况下才能用birthday列的值进行排序，而这个查询中通过name进行范围查找的记录中可能并不是按照birthday列进行排序的，所以在搜索条件中继续以birthday列进行查找时是用不到这个B+树索引的。

1. 精确匹配某一列并范围匹配另外一列

对于同一个联合索引来说，虽然对多个列都进行范围查找时只能用到最左边那个索引列，但是如果左边的列是精确查找，则右边的列可以进行范围查找，比方说这样：

SELECT \* FROM person\_info WHERE name = 'Ashburn' AND birthday > '1980-01-01' AND birthday < '2000-12-31' AND phone\_number > '15100000000';

这个查询的条件可以分为3个部分：

name = 'Ashburn'，对name列进行精确查找，当然可以使用B+树索引了。

birthday > '1980-01-01' AND birthday < '2000-12-31'，由于name列是精确查找，所以通过name = 'Ashburn'条件查找后得到的结果的name值都是相同的，它们会再按照birthday的值进行排序。所以此时对birthday列进行范围查找是可以用到B+树索引的。

phone\_number > '15100000000'，通过birthday的范围查找的记录的birthday的值可能不同，所以这个条件无法再利用B+树索引了，只能遍历上一步查询得到的记录。

## explain

一条查询语句在经过mysql查询优化器各种基于成本和规则的优化后会生成一个所谓的执行计划，展示了接下来具体执行的查询方式。如果想看某个查询的执行计划，那就可以在查询语句前加上explain

explain 关键字解释：

|  |  |
| --- | --- |
| id | 在一个大的查询语句中每个SELECT关键字都对应一个唯一的id |
| select\_type | SELECT关键字对应的那个查询的类型 |
| table | 表名 |
| partitions | 匹配的分区信息 |
| type | 针对单表的访问方法 |
| possible\_keys | 可能用到的索引 |
| key | 实际上使用的索引 |
| key\_len | 实际使用到的索引长度 |
| ref | 当使用索引列等值查询时，与索引列进行等值匹配的对象信息 |
| rows | 预估的需要读取的记录条数 |
| filtered | 某个表经过搜索条件过滤后剩余记录条数的百分比 |
| Extra | 一些额外的信息 |

## information\_schema