# mysql各种引擎对比、实战

## 存储引擎概述

数据库引擎是用于存储、处理和保护数据的核心服务。利用数据库引擎可控制访问权限并快速处理事务，从而满足企业内大多数需要处理大量数据的应用程序的要求。 使用数据库引擎创建用于联机事务处理或联机分析处理数据的关系数据库。这包括创建用于存储数据的表和用于查看、管理和保护数据安全的数据库对象（如索引、视图和存储过程）。

存储引擎作用：

1. 设计并创建数据库以保存系统所需的关系或XML文档
2. 实现系统以访问和更改数据库中存储的数据。
3. 为单位或客户部署实现的系统。
4. 提供日常管理支持以优化数据库的性能。

## InnoDB（默认的存储引擎）

设计目标：处理大容量数据库系统，MySQL运行时会在内存中建立缓冲池，用于缓冲数据和索引。

事务型的存储引擎，行级锁定和外键约束。

提供对数据库ACID事务的支持，并且实现了SQL标准的四种隔离级别。不支持FULLTEXT类型的索引，而且没有保存表的行数。当SELECT COUNT(\*) FROM TABLE时需要扫描全表。

适用场景：

1. 经常更新的表，适合处理多重并发的更新请求
2. 支持事务
3. 可以从灾难中恢复
4. 外键约束
5. 支持自动增加列属性auto\_increment

A 事务的原子性(Atomicity)：指一个事务要么全部执行,要么不执行.也就是说一个事务不可能只执行了一半就停止了.比如你从取款机取钱,这个事务可以分成两个步骤:1划卡,2出钱.不可能划了卡,而钱却没出来.这两步必须同时完成.要么就不完成.

C 事务的一致性(Consistency)：指事务的运行并不改变数据库中数据的一致性.例如,完整性约束了a+b=10,一个事务改变了a,那么b也应该随之改变.

I 独立性(Isolation）:事务的独立性也有称作隔离性,是指两个以上的事务不会出现交错执行的状态.因为这样可能会导致数据不一致.

D 持久性(Durability）:事务的持久性是指事务执行成功以后,该事务所对数据库所作的更改便是持久的保存在数据库之中，不会无缘无故的回滚.

## MyIsam

不支持数据库事务，不支持行级锁和外键，因此当INSERT或UPDATE数据时即写操作需要锁定整个表，效率更低。

支持表级锁。

MyIsam 存储引擎独立于操作系统，也就是可以在windows上使用，也可以比较简单的将数据转移到linux操作系统上去。

引擎在创建表的时候，会创建三个文件，一个是.frm文件用于存储表的定义，一个是.MYD文件用于存储表的数据，另一个是.MYI文件，存储的是索引。操作系统对大文件的操作比较慢，这样将表分成三个文件，那么.MYD文件单独来存放数据自然可以优化数据库的查询等操作。

适用场景：

1. 不支持事务的设计，但是并不代表着有事务操作的项目不能用MyIsam存储引擎，可以在service层进行根据自己的业务需求进行相应的控制。
2. 不支持外键的表设计
3. 查询速度很快
4. MyIASM中存储了表的行数，于是SELECT COUNT(\*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么MyIASM也是很好的选择。

缺点：

不能再表损坏后恢复数据(不能主动恢复)

## Memory

使用存在内存中的内容来创建表。每个MEMORY表只实际对应一个磁盘文件。MEMORY类型的表访问非常的快，因为数据放在内存中，并且默认使用HASH索引。

但是一旦服务关闭，表中的数据就会丢失掉。 HEAP允许只驻留在内存里的临时表格。驻留在内存里让HEAP要比ISAM和MYISAM都快，但是它所管理的数据是不稳定的，而且如果在关机之前没有进行保存，那么所有的数据都会丢失。在数据行被删除的时候，HEAP也不会浪费大量的空间。HEAP表格在你需要使用SELECT表达式来选择和操控数据的时候非常有用。

适用场景：

1）那些内容变化不频繁的代码表，或者作为统计操作的中间结果表，便于高效地堆中间结果进行分析并得到最终的统计结果。

2）目标数据比较小，而且非常频繁的进行访问，在内存中存放数据，如果太大的数据会造成内存溢出。可以通过参数max\_heap\_table\_size控制Memory表的大小，限制Memory表的最大的大小。

3）数据是临时的，而且必须立即可用得到，那么就可以放在内存中。

4）存储在Memory表中的数据如果突然间丢失的话也没有太大的关系。

注意： Memory同时支持散列索引和B树索引，B树索引可以使用部分查询和通配查询，也可以使用<,>和>=等操作符方便数据挖掘，散列索引相等的比较快但是对于范围的比较慢很多。

特性要求：

1）要求存储的数据是数据长度不变的格式，比如，Blob和Text类型的数据不可用（长度不固定的）。

2）要记住，在用完表格之后就删除表格。

## InnoDB和MyIsam使用对比

1. 事务。MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持，提供事务支持已经外部键等高级数据库功能。InnoDB表的行锁也不是绝对的，假如在执行一个SQL语句时MySQL不能确定要扫描的范围，InnoDB表同样会锁全表，例如updatetable set num=1 where name like “a%”就是说在不确定的范围时，InnoDB还是会锁表的。
2. 性能主题。MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快。
3. 行数保存。InnoDB 中不保存表的具体行数，也就是说，执行select count() fromtable时，InnoDB要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count()语句包含where条件时，两种表的操作是一样的。
4. 索引存储。对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中，可以和其他字段一起建立联合索引。MyISAM支持全文索引（FULLTEXT）、压缩索引，InnoDB不支持；MyISAM的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而Innodb是索引和数据是紧密捆绑的，没有使用压缩从而会造成Innodb比MyISAM体积庞大不小。InnoDB存储引擎被完全与MySQL服务器整合，InnoDB存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB存储它的表＆索引在一个表空间中，表空间可以包含数个文件（或原始磁盘分区）。这与MyISAM表不同，比如在MyISAM表中每个表被存在分离的文件中。InnoDB 表可以是任何尺寸，即使在文件尺寸被限制为2GB的操作系统上。
5. 服务器数据备份。InnoDB必须导出SQL来备份，LOAD TABLE FROM MASTER操作对InnoDB是不起作用的，解决方法是首先把InnoDB表改成MyISAM表，导入数据后再改成InnoDB表，但是对于使用的额外的InnoDB特性(例如外键)的表不适用。而且MyISAM应对错误编码导致的数据恢复速度快。MyISAM的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作；InnoDB是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。
6. 锁的支持。\*\*MyISAM只支持表锁。InnoDB支持表锁、行锁 行锁大幅度提高了多用户并发操作的新能。但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的

## 原理对比

聚簇索引：对磁盘上实际数据重新组织以按指定的一个或多个列的值排序算法。特点是存储数据的顺序和索引顺序一致

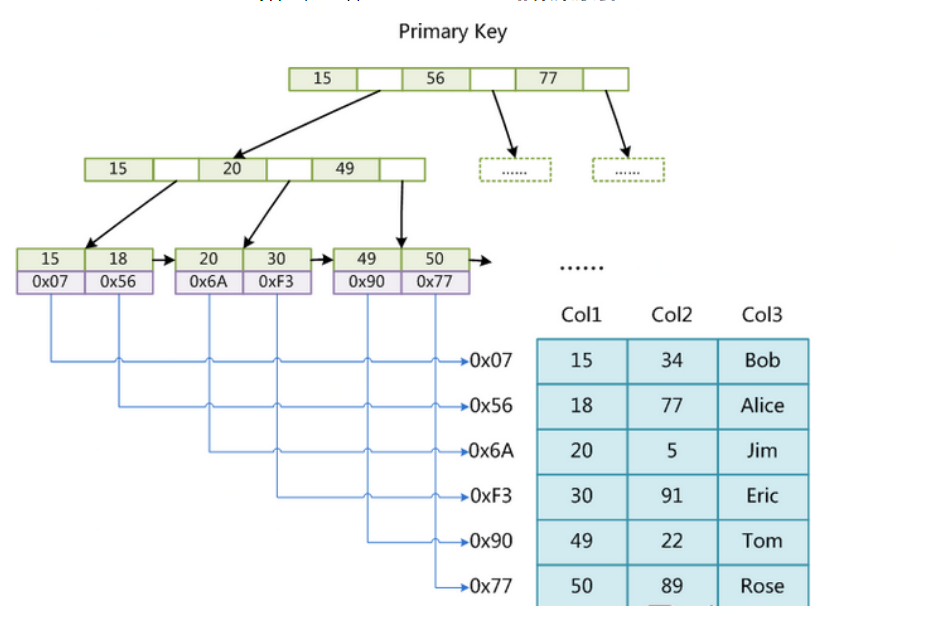
一般情况下主键会默认创建聚簇索引，且一张表只允许存在一个聚簇索引。

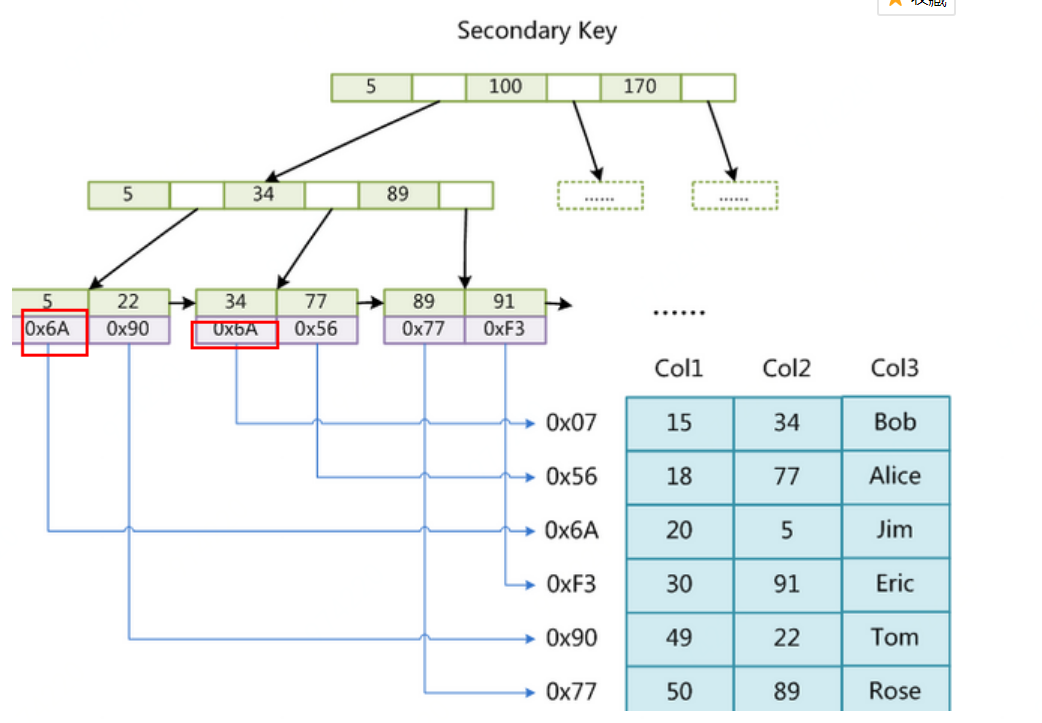
在《数据库原理》一书中是这么解释聚簇索引和非聚簇索引的区别的：  
聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。

因此，MYSQL中不同的数据存储引擎对聚簇索引的支持不同就很好解释了。

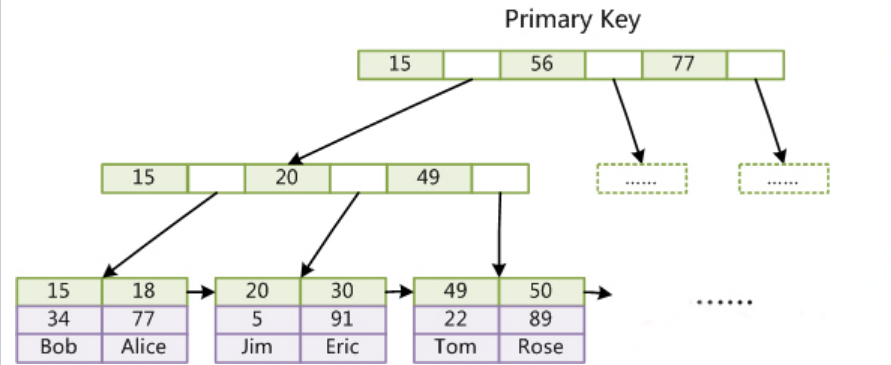
MyIASM引擎的索引结构：B+树来存储数据，MyISAM索引的指针指向的是键值的地址，地址存储的是数据。

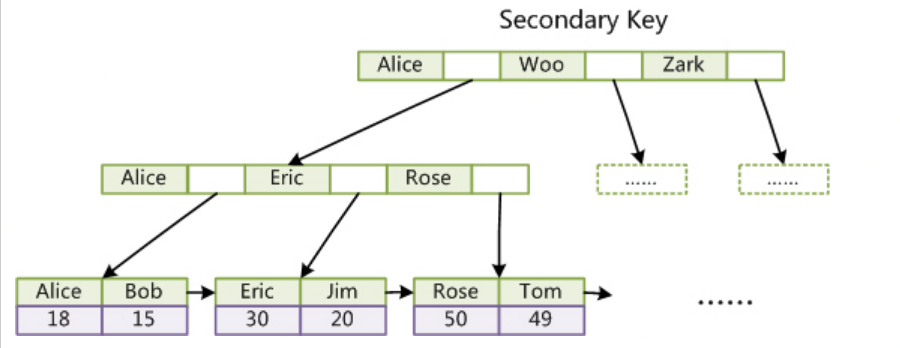
B+树的数据域存储的内容为实际数据的地址，也就是说它的索引和实际的数据是分开的，只不过是用索引指向实际的数据。





因此，过程为： **MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，根据data域的值去读取相应数据记录。**





**InnoDB引擎的索引结构：**

**B+ Tree索引结构，索引文件本身就是数据文件，即B+ Tree的数据域存储的就是实际的数据，这种索引就是聚集索引。这个索引的Key就是数据表的主键，因此InnoDB表数据文件本身就是主索引。**

InnoDB的辅助索引数据域存储的也是相应记录主键的值而不是地址，所以当以辅助索引查找时，会先根据辅助索引找到主键，再根据主键索引找到实际的数据。所以Innodb不建议使用过长的主键，否则会使辅助索引变得过大。所以Innodb不建议使用过长的主键，否则会使辅助索引变得过大。建议使用自增的字段作为主键，这样B+Tree的每一个结点都会被顺序的填满，而不会频繁的分裂调整，会有效的提升插入数据的效率。

# 索引

存储引擎用于快速找到记录的一种数据结构

<https://kb.cnblogs.com/page/45712/>

## 索引结构类型：

在MySQL中，索引是在存储引擎层而不是服务器层实现的，所以，并没有统一的索引标准：不同的存储引擎的索引的工作方式并不一样，也不是所有的存储引擎都支持所有类型的索引。

优点：可以让服务器快速定位到表的指定位置。

1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性
2. 可以大大加快数据的检索速度
3. 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
4. 在使用分组和排序 子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
5. 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能

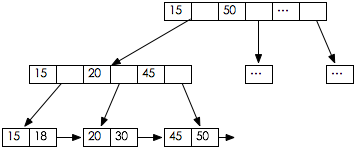
缺点：

1. 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。
2. 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。
3. 当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

### B- Tree索引

MySQL一般以B+树作为其索引结构，那么B+树有什么特点呢？

* 树度为n的话，每个节点指针上限为2n+1
* 非叶子节点不存储数据，只存储指针索引；叶子节点存储所有数据，不存储指针
* 在经典B+树基础上增加了**顺序访问指针**，每个叶子节点都有指向相邻下一个叶子节点的指针，如图所示。主要为了**提高区间访问的性能**，例如要找key为20到50的所有数据，只要按着顺序访问路线一次性访问所有数据节点。



局部性原理和磁盘预读：

局部性原理：为了减少磁盘IO，磁盘往往会进行数据预读，会从某位置开始，预先向后读取一定长度的数据放入内存。预读长度一般为页的整数倍，主存和磁盘以页为单位交换数据。当需要读取的数据不在内存时，触发缺页中断，系统会向磁盘发出读取磁盘数据的请求，磁盘找到数据的起始位置并向后连续读取一页或几页数据载入内存，然后中断返回，系统继续运行。

一般来说，索引本身较大，不会全部存储在内存中，会以索引文件的形式存储在磁盘上。所以索引查找数据过程中就会产生磁盘IO操作，而磁盘IO相对于内存存取非常缓慢，因此索引结构要尽量减少磁盘IO的存取次数。并且一般数据库设计时会将B+树节点的大小设置为一页，这样每个节点的载入只需要一次IO。

常见的索引类型，大部分引擎都支持B树索引。

* **普通索引**  
  这是最基本的索引类型，而且它没有唯一性之类的限制。普通索引可以通过以下几种方式创建：  
  （1）创建索引: CREATE INDEX 索引名 ON 表名(列名1，列名2,...);  
  （2）修改表: ALTER TABLE 表名ADD INDEX 索引名 (列名1，列名2,...);  
  （3）创建表时指定索引：CREATE TABLE 表名 ( [...], INDEX 索引名 (列名1，列名 2,...) );
* **UNIQUE索引**  
  表示唯一的，不允许重复的索引，如果该字段信息保证不会重复例如身份证号用作索引时，可设置为unique：  
  （1）创建索引：CREATE UNIQUE INDEX 索引名 ON 表名(列的列表);  
  （2）修改表：ALTER TABLE 表名ADD UNIQUE 索引名 (列的列表);  
  （3）创建表时指定索引：CREATE TABLE 表名( [...], UNIQUE 索引名 (列的列表) );
* **主键：PRIMARY KEY索引**  
  主键是一种唯一性索引，但它必须指定为“PRIMARY KEY”。  
  （1）主键一般在创建表的时候指定：“CREATE TABLE 表名( [...], PRIMARY KEY (列的列表) ); ”。  
  （2）但是，我们也可以通过修改表的方式加入主键：“ALTER TABLE 表名ADD PRIMARY KEY (列的列表); ”。  
  每个表只能有一个主键。 （主键相当于聚合索引，是查找最快的索引）  
  注：不能用CREATE INDEX语句创建PRIMARY KEY索引

### 哈希索引

只有Memory引擎支持，使用场景简单。

### R-Tree索引(空间索引)

空间索引是MyISAM的一种特殊索引类型，主要用于地理空间数据类型。

### Full-test(全文索引)

全文索引也是MyISAM的一种特殊索引类型，主要用于全文索引，InnoDB从MYSQL5.6版本提供对全文索引的支持。

## 复合索引

复合索引也叫联合索引，表示索引建立在多个列上。

我们删除t\_users上的索引，然后创建一个复合索引。

|  |  |
| --- | --- |
| 1  2  3  4 | mysql> drop index IDX\_FOR\_NUM on t\_users;  mysql> drop index IDX\_FOR\_NAME on t\_users;  mysql> create index INDEX\_FOR\_NAME\_AGE on t\_users(name, age); |

复合索引支持最左原则，也就是说INDEX\_FOR\_NAME\_AGE索引支持name字段的查找，支持name，age字段的查找，但是不支持age，name字段的查找以及age字段的查找。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | mysql> explain select \* from t\_users where name = 'format-100000';  +----+-------------+---------+------+--------------------+--------------------+---------+-------+------+-----------------------+  | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  +----+-------------+---------+------+--------------------+--------------------+---------+-------+------+-----------------------+  | 1 | SIMPLE | t\_users | ref | INDEX\_FOR\_NAME\_AGE | INDEX\_FOR\_NAME\_AGE | 767 | const | 1 | Using index condition |  +----+-------------+---------+------+--------------------+--------------------+---------+-------+------+-----------------------+  mysql> explain select \* from t\_users where name = 'format-100000' and age = 100000;  +----+-------------+---------+------+--------------------+--------------------+---------+-------------+------+-----------------------+  | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  +----+-------------+---------+------+--------------------+--------------------+---------+-------------+------+-----------------------+  | 1 | SIMPLE | t\_users | ref | INDEX\_FOR\_NAME\_AGE | INDEX\_FOR\_NAME\_AGE | 775 | const,const | 1 | Using index condition |  +----+-------------+---------+------+--------------------+--------------------+---------+-------------+------+-----------------------+  mysql> explain select \* from t\_users where age = 100000;  +----+-------------+---------+------+---------------+------+---------+------+--------+-------------+  | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  +----+-------------+---------+------+---------------+------+---------+------+--------+-------------+  | 1 | SIMPLE | t\_users | ALL | NULL | NULL | NULL | NULL | 996677 | Using where |  +----+-------------+---------+------+---------------+------+---------+------+--------+-------------+ |

使用age，name字段查找：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | mysql> explain select \* from t\_users where age = 100000 and name = 'format-100000';  +----+-------------+---------+------+--------------------+--------------------+---------+-------------+------+-----------------------+  | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  +----+-------------+---------+------+--------------------+--------------------+---------+-------------+------+-----------------------+  | 1 | SIMPLE | t\_users | ref | INDEX\_FOR\_NAME\_AGE | INDEX\_FOR\_NAME\_AGE | 775 | const,const | 1 | Using index condition |  +----+-------------+---------+------+--------------------+--------------------+---------+-------------+------+-----------------------+ |

我们发现使用age，name字段的查找使用了复合索引，这是因为MySQL内部有个查询优化器帮我们进行了优化。

## hash索引和btree索引的区别

1. hash索引不能使用范围查询，只能使用一些比如 “=”, “<>”, “in”查询。因为hash索引会计算索引列的hash值，计算出后的hash值经过了hash算法与原先的值完全不一样，只能进行等值的过滤，不能基于范围的过滤
2. hash索引遇到大量hash值相同的情况下，性能比btree要差
3. hash索引并不一定一次可以定位到数据。因为基于索引列计算出的hash值会有重复，重复的话需要扫描hash表进行比较
4. 由于hash索引中存放的是经过hash计算之后的hash值，而且hash值的大小关系并不一定和hash运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算
5. 对于组合索引，hash索引在计算hash值的时候是组合索引键合并后再一起计算hash值，而不是单独计算hash值，所以通过组合索引的前面一个或几个索引键进行查询的时候，hash索引也无法被利用
6. InnoDB和MyISAM引擎不支持hash索引

### 索引选择原则

**1. 较频繁的作为查询条件的字段应该创建索引**  
**2. 唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件**  
**3. 更新非常频繁的字段不适合创建索引**

**4. 不会出现在 WHERE 子句中的字段不该创建索引**

# MySQL数据库视图

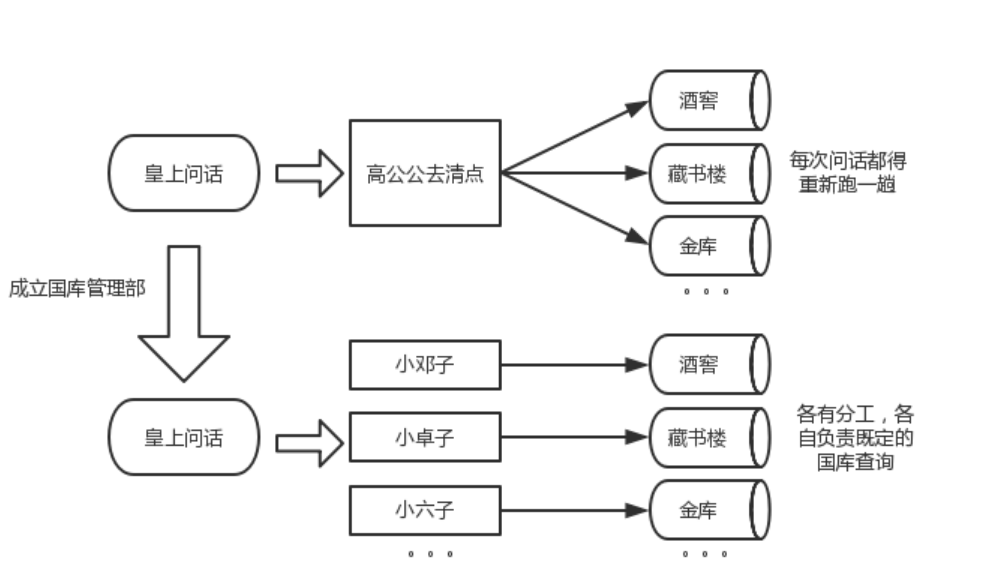
视图是指计算机数据库中的视图，是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列和行数据。但是，**视图并不在数据库中以存储的数据值集形式存在。行和列数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。**

| **操作指令** | **代码** |
| --- | --- |
| 创建视图 | CREATE VIEW 视图名(列1，列2...) AS SELECT (列1，列2...) FROM ...; |
| 使用视图 | 当成表使用就好 |
| 修改视图 | CREATE OR REPLACE VIEW 视图名 AS SELECT [...] FROM [...]; |
| 查看数据库已有视图 | >SHOW TABLES [like...];（可以使用模糊查找） |
| 查看视图详情 | DESC 视图名或者SHOW FIELDS FROM 视图名 |
| 视图条件限制 | [WITH CHECK OPTION] |

## ****1 视图****

百度百科定义了什么是视图，但是对缺乏相关知识的人可能还是难以理解或者只有一个比较抽象的概念，笔者举个例子来解释下什么是视图。

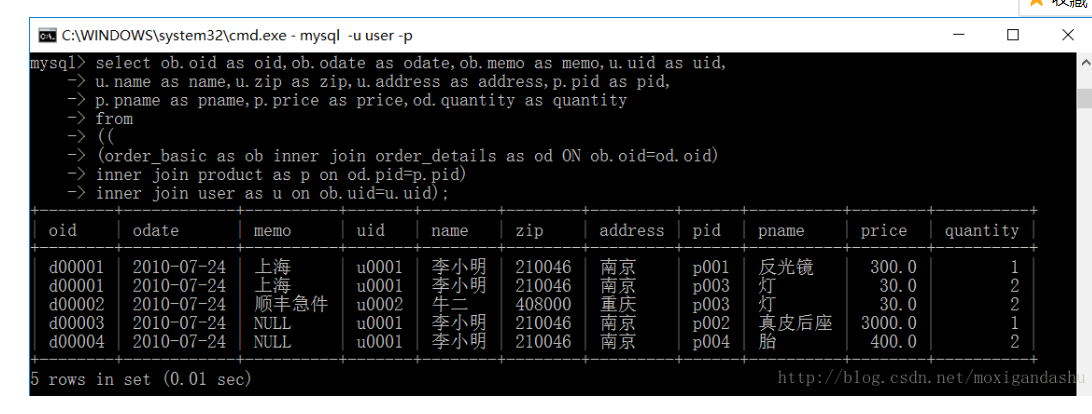
朕想要了解皇宫的国库的相关情况，想知道酒窖有什么酒，剩多少，窖藏多少年，于是派最信任的高公公去清点，高公公去国库清点后报给了朕；朕又想知道藏书情况，于是又派高公公去清点并回来报告给朕，又想知道金银珠宝如何，又派高公公清点。。。过一段时间又想知道藏书情况，高公公还得重新再去清点，皇上问一次，高公公就得跑一次路。

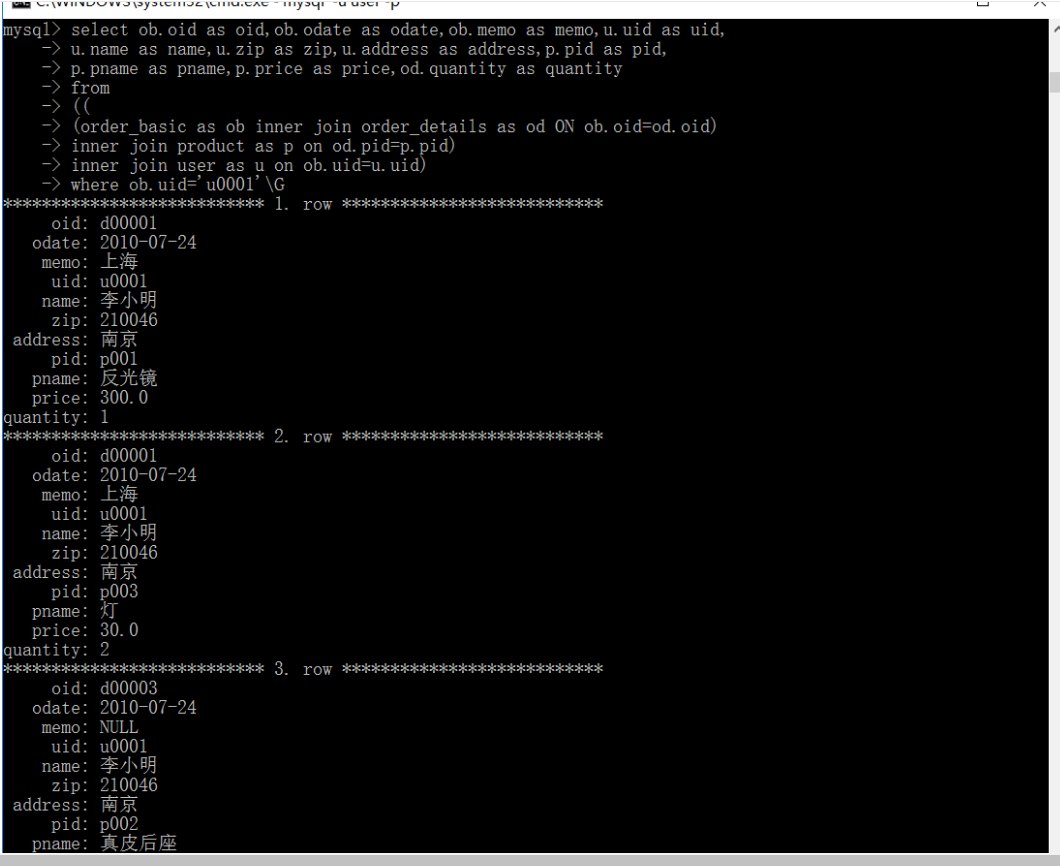
后来皇上觉得高公公不容易，就成立了国库管理部门，小邓子负责酒窖，小卓子负责藏书，而小六子负责金库的清点。。。后来皇上每次想了解国库就直接问话负责人，负责人就按照职责要求进行汇报。   


安排专人管理后，每次皇上想要了解国库情况，就不必让高公公每次都跑一趟，而是指定的人员按照指定的任务完成指定的汇报工作就可以了。

和数据库相对应，每次进行查询工作，都需要编写查询代码进行查询；而视图的作用就是不必每次都重新编写查询的SQL代码，而是通过视图直接查询即可。因此：

**视图是虚拟表，本身不存储数据，而是按照指定的方式进行查询。**

比如，我们希望从前文提到的四张表，order\_baisc,order\_details，user和product中查找所有记录，需要写入代码指令：   
  
想再次查询这几个表中uid为u0001的用户的记录，有需要键入一次操作指令：

  
  
也就是说，每次查询都得重新键入查询指令SQL代码，这种费时费力的体力活，对于时间就是生命的你我来说，是不划算的。所以借助视图，来执行相同或相似的查询。

## ****2 创建视图****

**2.1 创建视图create view**   
创建视图的代码为：

>CREATE VIEW 视图名(列1，列2...)

AS SELECT (列1，列2...)

FROM ...;

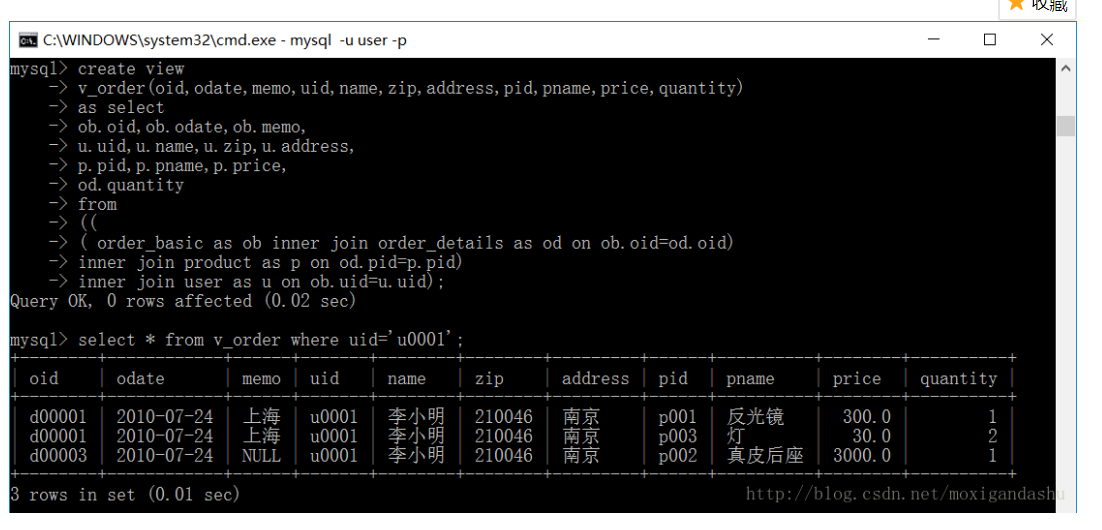
* 1
* 2
* 3

可以看到，创建视图和查询相比，增加了前面的CREATE VIEW 视图名 AS

**2.2 视图运用**

使用视图和使用表完全一样，只需要把视图当成一张表就OK了。**视图是一张虚拟表。**

eg：创建order\_baisc,order\_details，user和product的查询视图，并通过视图查找uid为u0001的记录：



**2.3 修改视图CREATE OR REPLACE VIEW**

修改和创建视图可以使用代码：

CREATE OR REPLACE VIEW 视图名 AS SELECT [...] FROM [...];

* 1

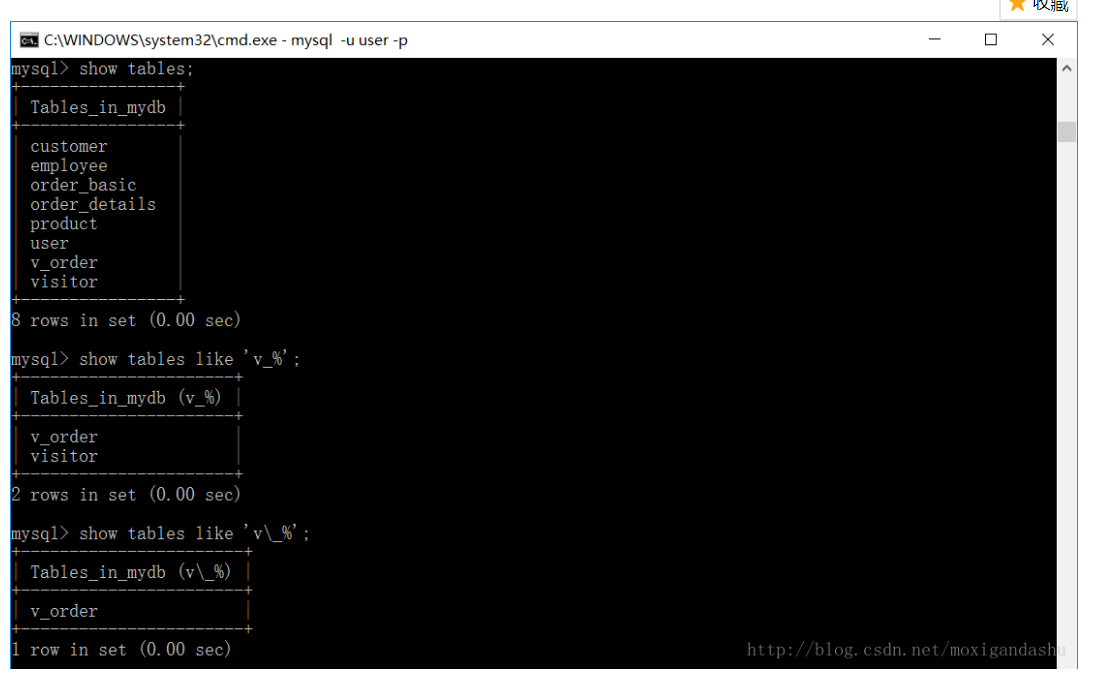
eg: 

**2.4 查看视图**   
**(1)查看数据库中有哪些视图 show tables**   
前面提到，视图就是虚拟的表，因此，查看视图的方法和查看表的方法是一样的：

>SHOW TABLES;

* 1

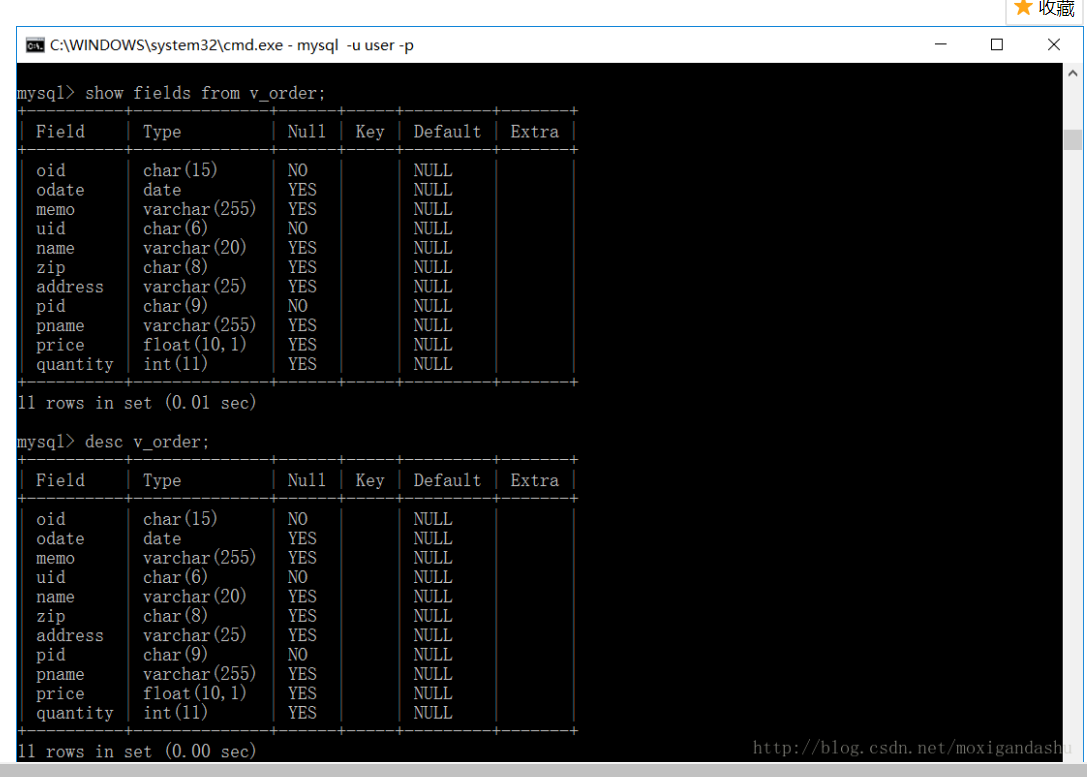
通过show tables;反馈得到所有的表和视图。同样的，我们可以通过模糊检索的方式专门查看视图，这个时候，视图的命令统一采用v\_视图名v\_视图名的优势就体现出来了。

  
**（2）查看视图详情**   
查看视图详情的方法有两种，一种是和查看表详情一样使用desc 视图名，另外一种方法是show fields from 视图名：

>DESC 视图名;

或者

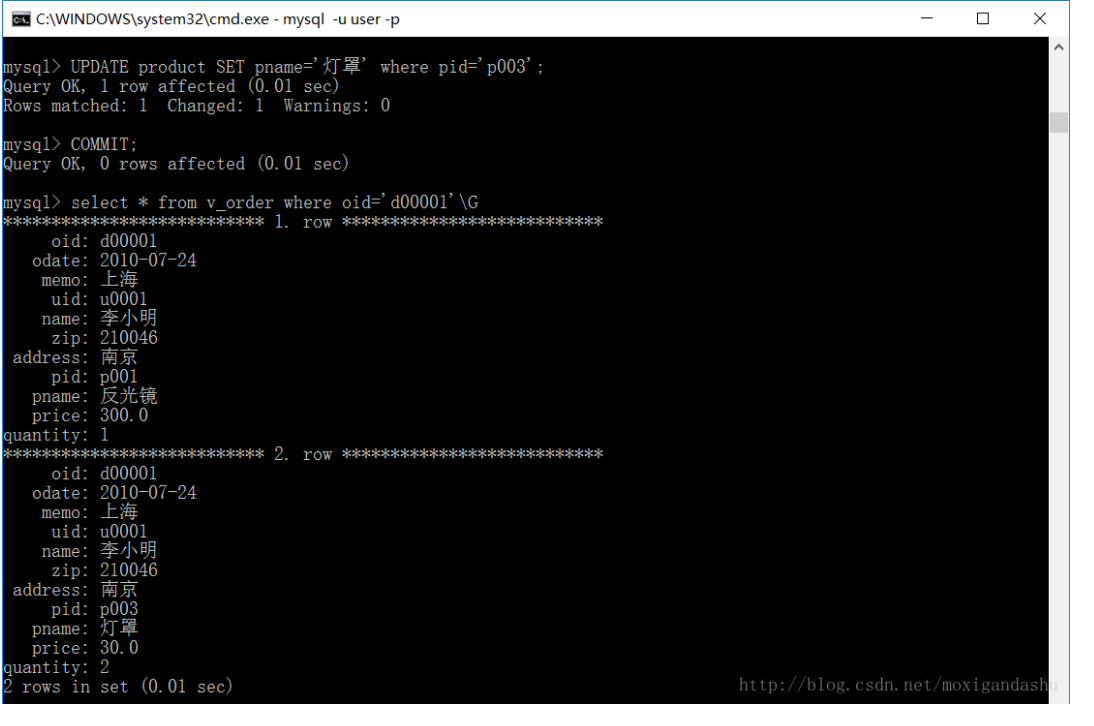
>SHOW FIELDS FROM 视图名;



两种方法得到的详情都是一毛一样的。

## ****3 视图与数据变更****

**3.1 表格数据变更**   
将表product中的数据进行更新，在通过视图检索：



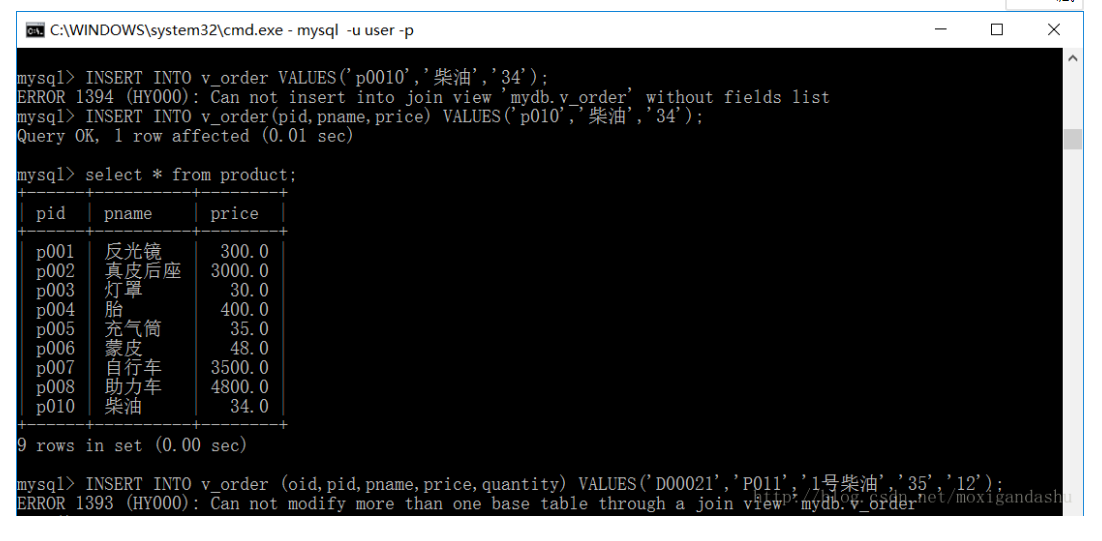
可以看到表格数据变化后，在通过视图检索，得到的结果也同步发生了变化，因此，在此证明了：

**视图不是表，不保存数据，知识一张虚拟表；**

**3.2 通过视图变更数据**

* （1）插入数据

>INSERT INTO v\_order(pid,pname,price) VALUES('p010','柴油','34');



在此查询视图，发现插入了数据。

* （2）跨表插入数据   
  通过上图，我们可以看到，跨表插入数据系统反馈报错，提示不能修改超过一个表的数据。

**因此，可以通过视图插入数据，但是只能基于一个基础表进行插入，不能跨表更新数据。**

* **（3）WITH CHECK OPTION**   
  如果在创建视图的时候制定了“WITH CHECK OPTION”，那么更新数据时不能插入或更新不符合视图限制条件的记录。

eg:对表product创建一个单价超过3000的视图，并加上“WITH CHECK OPTION”，之后插入一个价格为42的记录：

可以看到系统提示错误CHECK OPTION FAILED。因为视图限制了价格要高于3000.   
后面再次尝试了不加“WITH CHECK OPTION”的视图，后者可以成功插入。

同样的，在不加“WITH CHECK OPTION”的情况下，通过视图修改记录，也可以成功执行： 

通过视图修改，可能导致数据无故消失，因此：

**没有特殊的理由，建议加上“WITH CHECK OPTION”命令。**

**注意点：**   
1. 视图不是表，不直接存储数据，是一张虚拟的表；   
2. 一般情况下，在创建有条件限制的视图时，加上“WITH CHECK OPTION”命令。

# MySQL存储过程

存储过程是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数(如果该存储过程带有参数)来调用执行它。

存储过程是一些预编译的SQL语句。

更加直白的理解：存储过程可以说是一个记录集，它是由一些T-SQL语句组成的代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。

存储过程是一个预编译的代码块，执行效率比较高

一个存储过程替代大量T\_SQL语句，可以降低网络通信量，提高通信速率

可以一定程度上确保数据安全

存储过程是可编程的函数，在数据库中创建并保存，可以由SQL语句和控制结构组成。当想要在不同的应用程序或平台上执行相同的函数或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟，它允许控制数据的访问方式。

-------------创建名为GetUserAccount的存储过程----------------

create Procedure GetUserAccountRe1

@UserName nchar(20),

@UserID int output

as

if(@UserName>5)

select @UserID=COUNT(\*) from UserAccount where UserID>25

else

set @UserID=1000

return @@rowcount

go

-------------执行上面的存储过程----------------

exec GetUserAccountRe1 '7',null

常见数据库面试题：！！！很重要！！！

<https://zhuanlan.zhihu.com/p/23713529>

操作系统面试题

<https://zhuanlan.zhihu.com/p/23755202>

# MySQL的复制原理以及流程

保证主服务器和从服务器的数据是一致的，向Master插入数据后，Slave会自动从Master把修改的数据同步过来(有一定的延迟)。

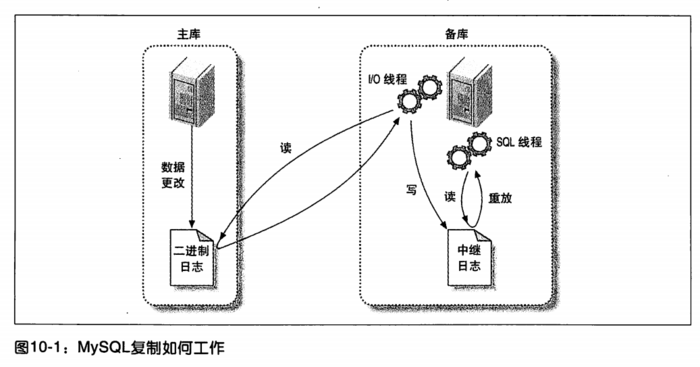
## Mysql 复制能解决什么问题

* 1. **高可用和故障切换**  
     复制能够帮避免MySql单点失败，因为数据都是相同的，所以当Master挂掉后，可以指定一台Slave充当Master继续保证服务运行，因为数据是一致性的（如果当插入Master就挂掉，可能不一致，因为同步也需要时间），当然这种配置不是简单的把一台Slave充当Master，毕竟还要考虑后续的Salve同步Master
* **二、负载均衡**  
  因为读写分离也算是负载均衡的一种，所以就不单独写了，因为一般都是有多台Slave的，所以可以将读操作指定到Slave服务器上（需要代码控制），然后再用负载均衡来选择那台Slave来提供服务，同时也可以吧一些大量计算的查询指定到某台Slave，这样就不会影响Master的写入以及其他查询
* **三、数据备份**  
  一般我们都会做数据备份，可能是写定时任务，一些特殊行业可能还需要手动备份，有些行业要求备份和原数据不能在同一个地方，所以主从就能很好的解决这个问题，不仅备份及时，而且还可以多地备份，保证数据的安全
  1. **四、业务模块化**  
     可以一个业务模块读取一个Slave，再针对不同的业务场景进行数据库的索引创建和根据业务选择MySQL存储引擎， 不同的slave可以根据不同需求设置不同索引和存储引擎

**主从配置需要注意的点**  
(1)主从服务器操作系统版本和位数一致；  
(2) Master和Slave数据库的版本要一致；  
(3) Master和Slave数据库中的数据要一致；  
(4) Master开启二进制日志，Master和Slave的server\_id在局域网内必须唯一；

## 复制如何工作

复制有三个步骤：  
1、Master将数据改变记录到二进制日志(binary log)中，也就是配置文件log-bin指定的文件，这些记录叫做二进制日志事件(binary log events)  
2、Slave通过I/O线程读取Master中的binary log events并写入到它的中继日志(relay log)  
3、Slave重做中继日志中的事件，把中继日志中的事件信息一条一条的在本地执行一次，完成数据在本地的存储，从而实现将改变反映到它自己的数据(数据重放)



第一步Master记录二进制日志， 每次提交事务完成数据更新前，Master将数据更新的时间记录到二进制日志中，MySql会按事务提交的顺序而非每条语句的执行顺序来记录二进制日志。再记录二进制日志后，主库会告诉存储引擎可以提交事务了。

第二步，Slave将Master的二进制日志复制到本地的中继日志中，首先，Slave会启动一个工作线程，成为I/O线程， I/O线程跟Master建立一个普通的客户端链接，然后再Master上启动一个特殊的二进制转储（binlog dump）线程（该线程没有对应的SQL命令），这个二进制转储线程会读取主库上的二进制日志中的事件。从库I/O线程将接受到时间记录到中继日志中。

第三步从库的SQL线程执行最后异步，该线程的从中继日志中读取事件并在从库执行，从而实现从库数据更新。

## 复制类型

**1、基于语句的复制**  
在Master上执行的SQL语句，在Slave上执行同样的语句。MySQL默认采用基于语句的复制，效率比较高。一旦发现没法精确复制时，会自动选着基于行的复制

**2、基于行的复制**  
把改变的内容复制到Slave，而不是把命令在Slave上执行一遍。从MySQL5.0开始支持

**3、混合类型的复制**  
默认采用基于语句的复制，一旦发现基于语句的无法精确的复制时，就会采用基于行的复制

作者：cooffeelis  
链接：https://www.jianshu.com/p/63c1a1babfd1  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

作者：cooffeelis  
链接：https://www.jianshu.com/p/63c1a1babfd1  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

作者：cooffeelis  
链接：https://www.jianshu.com/p/63c1a1babfd1  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。