主键 外键 索引

- 主键是表中唯一标识一条记录的，不允许有重复的、不能为空
- 外键是用来和其他表建立联系的，表的外键是另一个表的主键，可以有重复的可以为空
- 索引是对数据库中某些关键字段进行存储，类似数据中的目录，里面包含关键数据和数据的位置，不能太多一般一个表不能超过四个

索引

索引是帮助Mysql高效获得数据的排好序的数据结构

- **优点：**大大加快数据的检索速度，这是创建索引最主要的原因

  通过索引，在查询过程中，使用**隐藏优化器**，提高系统的性能

- **缺点：**创建索引和更新索引都需要消耗时间，当你在增删改查的时候，也需要去动态的修改维护索引，会降低增删改查的效率

  索引占用物理空间

聚簇索引与非聚簇索引的区别

索引存在磁盘

（1）聚簇索引也就是所谓的主键索引，一般只需要查询一次即可，但是如果是二级索引查询也就是非聚簇索引查询，则需要回表操作，多次查询，在大数据量的情况下，主键索引的效率会很高。

（2）聚簇索引：物理存储按照索引排序，**叶子节点包含了完整的数据记录，是数据节点**；非聚集索引：物理存储不按照索引排序，**叶子节点只包含主键的值，仍是索引节点**；可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。

Hash 索引和 B+ 树索引区别？设计索引怎么抉择?

- B+ 树可以进行范围查询，Hash 索引不能。
- B+ 树支持联合索引的最左侧原则，Hash 索引不支持。
- B+ 树支持 order by 排序，Hash 索引不支持。
- Hash 索引在等值查询上比 B+ 树效率更高。
- B+ 树使用 like 进行模糊查询的时候，like 后面（比如%开头）的话可以起到优化的作用，Hash 索引根本无法进行模糊查询。

最左前缀原则？最左匹配原则?

最左前缀原则，就是最左优先，在创建多列索引时，要根据业务需求，where 子句中使用最频繁的一列放在最左边。

当我们创建一个组合索引的时候，如 (a1,a2,a3)，相当于创建了（a1）、(a1,a2)和(a1,a2,a3)三个索引，这就是最左匹配原则。

不可以跳过左边索引直接用右边索引

B 树& B+树

B 树全称为 **多路平衡查找树** ，B+ 树是 B 树的一种变体。B 树和 B+树中的 B 是 `Balanced` （平衡）的意思。

目前大部分数据库系统及文件系统都采用 B-Tree 或其变种 B+Tree 作为索引结构。

B 树& B+树两者有何异同呢?

- B 树的所有节点既存放键(key) 也存放数据(data)，而 B+树只有叶子节点存放 key 和 data，其他内节点只存放 key。
- B 树的叶子节点都是独立的;B+树的叶子节点有一条引用链指向与它相邻的叶子节点。
- B 树的检索的过程相当于对范围内的每个节点的关键字做二分查找，可能还没有到达叶子节点，检索就结束了。而 B+树的检索效率就很稳定了，任何查找都是从根节点到叶子节点的过程，叶子节点的顺序检索很明显。

在 MySQL 中，MyISAM 引擎和 InnoDB 引擎都是使用 B+Tree 作为索引结构，但是，两者的实现方式不太一样。

MyISAM 引擎中，B+Tree 叶节点的 data 域存放的是数据记录的地址。在索引检索的时候，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 域的值为地址读取相应的数据记录。这被称为“非聚簇索引”。

InnoDB 引擎中，其数据文件本身就是索引文件。相比 MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按 B+Tree 组织的一个索引结构，树的叶节点 data 域保存了完整的数据记录。这个索引的 key 是数据表的主键，因此 InnoDB 表数据文件本身就是主索引。这被称为“聚簇索引（或聚集索引）”，而其余的索引都作为辅助索引，辅助索引的 data 域存储相应记录主键的值而不是地址，这也是和 MyISAM 不同的地方。在根据主索引搜索时，直接找到 key 所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，在走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。

事务四大特性，并解释这四大特性的含义

事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务最经典也经常被拿出来说例子就是转账了。假如小明要给小红转账1000元，这个转账会涉及到两个关键操作就是：将小明的余额减少1000元，将小红的余额增加1000元。万一在这两个操作之间突然出现错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

1. **原子性（undolog）：** 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. **一致性：** 执行事务前后，数据保持一致，例如转账业务中，无论事务是否成功，转账者和收款人的总额应该是不变的；
3. **隔离性（加锁和MVCC）：** 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
4. **持久性（redolog）：** 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

总结：事务是最小执行单位，不能再继续分割，原子性确保动作要么全部提交成功，要么全部失败回滚。执行事务前后，数据要保持一致。隔离性是隔离并发运行的多个事务之间的相互影响。事务提交成功，去修改会永远保存到数据库中，即使系统崩溃，修改的数据也不会丢失。

并发事务处理会带来哪些问题?

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对统一数据进行操作）。并发虽然是必须的，但可能会导致以下的问题。

- **脏读（Dirty read）（读写）:** 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- **丢失修改（Lost to modify）（写写）:** 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。 例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- **不可重复读（Unrepeatableread）（读写）:** 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- **幻读（Phantom read）（读写）:** 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

总结：使用了别的事务修改后未提交的数据，可能导致脏读。在别的事务提交修改之前，对相同的数据进行修改，可能会导致覆盖别的事务的修改，造成丢失修改。在别的事务还没有结束之前，对数据进行了修改，可能导致别的事务两次读取数据不一致，称为不可重复读。在别的事务完成前，增/删了数据，导致产生了幻读。

不可重复度和幻读区别：

不可重复读的重点是修改，幻读的重点在于新增或者删除。

例1（同样的条件, 你读取过的数据, 再次读取出来发现值不一样了 ）：事务1中的A先生读取自己的工资为 1000的操作还没完成，事务2中的B先生就修改了A的工资为2000，导 致A再读自己的工资时工资变为 2000；这就是不可重复读。

例2（同样的条件, 第1次和第2次读出来的记录数不一样 ）：假某工资单表中工资大于3000的有4人，事务1读取了所有工资大于3000的人，共查到4条记录，这时事务2 又插入了一条工资大于3000的记录，事务1再次读取时查到的记录就变为了5条，这样就导致了幻读。

事务隔离级别

SQL 标准定义了四个隔离级别：

- **READ-UNCOMMITTED(读取未提交)：** 最低的隔离级别，允许读取尚未提交的数据变更，**可能会导致脏读、幻读或不可重复读。**
- **READ-COMMITTED(读取已提交)：** 允许读取并发事务已经提交的数据，**可以阻止脏读，但是幻读或不可重复读仍有可能发生。**
- **REPEATABLE-READ(可重复读)：** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**（Innodb引擎里在RR隔离级别下默认使用间隙锁，解决了幻读的问题）
- **SERIALIZABLE(可串行化)：** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读。**

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

```
MySQL InnoDB 存储引擎的默认支持的隔离级别是 **REPEATABLE-READ（可重读）
```

InnoDB 和 MyISM、MEMORY 的区别是什么

InnoDB

InnoDB 是 MySQL 的默认事务型引擎，用来处理大量短期事务。InnoDB 的性能和**自动崩溃恢复特性**使得它在非事务型存储需求中也很流行，除非有特别原因否则应该优先考虑 InnoDB。

InnoDB 采用 MVCC 来支持高并发，并且实现了四个标准的隔离级别，其默认级别是 REPEATABLE-READ ，并通过间隙锁策略防止幻读。

InnoDB 表是基于聚簇索引建立的，InnoDB 的索引结构和其他存储引擎有很大不同，聚簇索引对主键查询有很高的性能。

MyISAM（mysang）

MySQL5.1及之前，MyISAM 是默认存储引擎，MyISAM 支持全文索引，但不支持事务和行锁，**最大的缺陷就是崩溃后无法安全恢复。**对于只读的数据或者表比较小、可以忍受修复操作的情况仍然可以使用 MyISAM。

MyISAM 对整张表进行加锁，**读取时**会对需要读到的**所有表加共享锁**，**写入时**则对**表加排它锁**。但是在表有读取查询的同时，也支持并发往表中插入新的记录。

Memory

如果需要快速访问数据且这些数据不会被修改，重启以后丢失也没有关系，那么使用 Memory 表是非常有用的。Memory 表至少要比 MyISAM 表快一个数量级，因为所有数据都保存在内存，不需要磁盘 IO，Memory 表的结构在重启后会保留，但数据会丢失。

Memory 表适合的场景：查找或者映射表、缓存周期性聚合数据的结果、保存数据分析中产生的中间数据。

Memory 表支持哈希索引，因此查找速度极快。虽然速度很快但还是无法取代传统的基于磁盘的表，Memory 表使用表级锁，因此并发写入的性能较低。如果 MySQL 在执行查询的过程中需要使用临时表来保持中间结果，内部使用的临时表就是 Memory表。如果中间结果太大超出了Memory 表的限制，或者含有 BLOB 或 TEXT 字段，临时表会转换成MyISAM 表。

对比两者：

1.是否支持行级锁

MyISAM 只有表级锁(table-level locking)，而 InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁。

也就说，MyISAM 一锁就是锁住了整张表，这在并发写的情况下性能很差！这也是为什么 InnoDB 在并发写的时候，性能更牛！

2.是否支持事务

MyISAM 不提供事务支持。

InnoDB 提供事务支持，具有提交(commit)和回滚(rollback)事务的能力，也有ACID insert update等等。

3.是否支持外键

MyISAM 不支持，而 InnoDB 支持。

拓展一下：

一般我们也是不建议在数据库层面使用外键的，应用层面可以解决。不过，这样会对数据的一致性造成威胁。具体要不要使用外键还是要根据你的项目来决定。

4.是否支持数据库异常崩溃后的安全恢复

MyISAM 不支持，而 InnoDB 支持。

使用 InnoDB 的数据库在异常崩溃后，数据库重新启动的时候会保证数据库恢复到崩溃前的状态。这个恢复的过程依赖于 `redo log` 。

拓展一下：

- MySQL InnoDB 引擎使用 **redo log(重做日志)** 保证事务的**持久性**，使用 **undo log(回滚日志)** 来保证事务的**原子性**。
- MySQL InnoDB 引擎通过 **锁机制**、**MVCC** 等手段来保证事务的隔离性（ 默认支持的隔离级别是 `REPEATABLE-READ` ）。
- 保证了事务的持久性、原子性、隔离性之后，一致性才能得到保障。

5.是否支持 MVCC

MyISAM 不支持，而 InnoDB 支持。

MyISAM 连行级锁都不支持。MVCC 可以看作是行级锁的一个升级，可以有效减少加锁操作，提供性能。

关于 MyISAM 和 InnoDB 的选择问题

大多数时候我们使用的都是 InnoDB 存储引擎，在某些读密集的情况下，使用 MyISAM 也是合适的。不过，前提是你的项目不介意 MyISAM 不支持事务、崩溃恢复等缺点（可是~我们一般都会介意啊！）。

《MySQL 高性能》上面有一句话这样写到:

> 不要轻易相信“MyISAM 比 InnoDB 快”之类的经验之谈，这个结论往往不是绝对的。在很多我们已知场景中，InnoDB 的速度都可以让 MyISAM 望尘莫及，尤其是用到了聚簇索引，或者需要访问的数据都可以放入内存的应用。

一般情况下我们选择 InnoDB 都是没有问题的，但是某些情况下你并不在乎可扩展能力和并发能力，也不需要事务支持，也不在乎崩溃后的安全恢复问题的话，选择 MyISAM 也是一个不错的选择。但是一般情况下，我们都是需要考虑到这些问题的。

因此，对于咱们日常开发的业务系统来说，你几乎找不到什么理由再使用 MyISAM 作为自己的 MySQL 数据库的存储引擎。

InnoDB 行锁实现方式

MyISAM 和 InnoDB 存储引擎使用的锁：

- MyISAM 采用表级锁(table-level locking)。
- InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁

表级锁和行级锁对比：

- **表级锁：** MySQL 中锁定 **粒度最大** 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM 和 InnoDB 引擎都支持表级锁。
- **行级锁：** MySQL 中锁定 **粒度最小** 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

InnoDB 存储引擎的锁的算法有三种：

- Record lock：记录锁，单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，不包括记录本身
- Next-key lock：record+gap临界锁，锁定一个范围，包含记录本身

在 MySQL 中，InnoDB 行锁通过给索引上的索引项加锁来实现，如果没有索引，InnoDB 将通过隐藏的聚簇索引来对记录加锁。

InnoDB 支持 3 种行锁定方式：

- 行锁（Record Lock）：直接对索引项加锁。
- 间隙锁（Gap Lock）：锁加在索引项之间的间隙，也可以是第一条记录前的“间隙”或最后一条记录后的“间隙”。
- Next-Key Lock：行锁与间隙锁组合起来用就叫做 Next-Key Lock。前两种的组合，对记录及其前面的间隙加锁。

默认情况下，InnoDB 工作在可重复读（默认隔离级别）下，并且以 Next-Key Lock 的方式对数据行进行加锁，这样可以有效防止幻读的发生。

Next-Key Lock 是行锁与间隙锁的组合，这样，当 InnoDB 扫描索引项的时候，会首先对选中的索引项加上行锁（Record Lock），再对索引项两边的间隙（向左扫描扫到第一个比给定参数小的值， 向右扫描扫到第一个比给定参数大的值， 然后以此为界，构建一个区间）加上间隙锁（Gap Lock）。如果一个间隙被事务 T1 加了锁，其它事务不能在这个间隙插入记录。

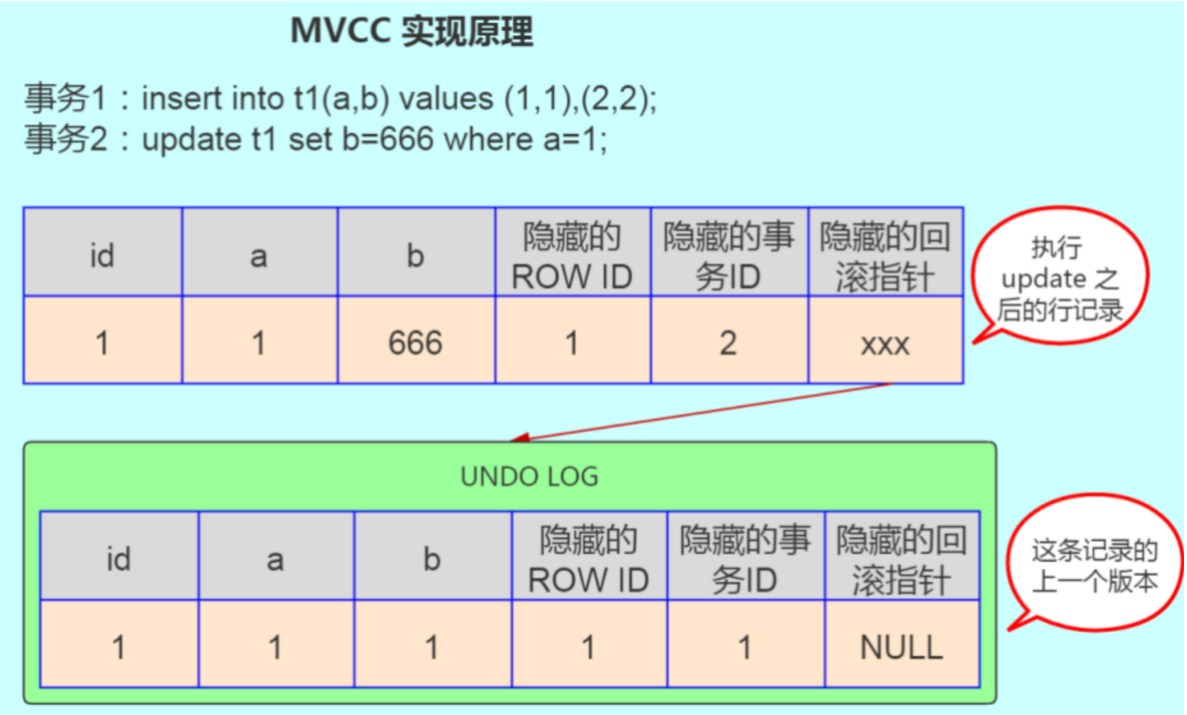
要禁止间隙锁的话，可以把隔离级别降为读已提交（READ COMMITTED），或者开启参数 innodb_locks_unsafe_for_binlog。

介绍MVCC

MVCC：多版本并发控制，是通过保存数据在某个时间点的快照来实现的，每个事务对同一张表。其实指的是一条记录会有多个版本，每次修改记录都会存储这条记录被修改之前的版本，多版本之间串联起来形成一条版本链

MVCC作用在MYSQL innDB中的实现主要为了提高数据库的并发性能，用更好的方式去处理读写冲突，做到即使有读写冲突时，也能做到不加锁，非阻塞并发读

MVCC实现原理:



如图，首先insert语句向表t1中插入了一条数据，a字段为1，b字段为1，ROWID 也为1，事务ID假设为1，回滚指针假设为null。当执行update t1 set b=666 where a=1时，大致步骤如下:

- 数据库会先对满足a=1的行加排他锁;
- 然后将原记录复制到undo表空间中;
- 修改b字段的值为666， 修改事务ID为2;
- 并通过隐藏的回滚指针指向undolog中的历史记录;
- 事务提交，释放前面对满足a=1的行所加的排他锁;

session2 查询的结果是session1修改之前的记录，这个记录就是来自undolog中;

因此可以总结出MVCC实现的原理大致是:
InnoDB每一行数据都有一个隐藏的回滚指针，用于指向该行修改前的最后一个历史版本，这个历史版本存放在undo log中。如果要执行更新操作，会将原记录放入undo log中，并通过隐藏的回滚指针指向undolog中的原记录。其它事务此时需要查询时，就是查询undolog中这行数据的最后一个历史版本。

MVCC最大的好处是读不加锁，读写不冲突，极大地增加了MySQL的并发性。通过MVCC，保证了事务ACID中的1 (隔离性)特性。

MVCC用来实现实现哪几个隔离级别：

- **读已提交**，一个事务中的每一次 SELECT查询都会获取一次Read View,同样的查询语句都会重新获取一次Read View,这时如果Read View不同，就可能产生不可重复读或者幻读的情况。
- **可重复读**，就避免了不可重复读，这是因为一个事务只在第一次SELECT的时候会获取一次Read View,而后面所有的SELECT都会复用这个Read View。

MVCC最大的好处是读不加锁，读写不冲突，极大地增加了MySQL的并发性。通过MVCC，保证了事务ACID中的(隔离性)特性。

MVCC带来的好处是

- 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能
- 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题

readview

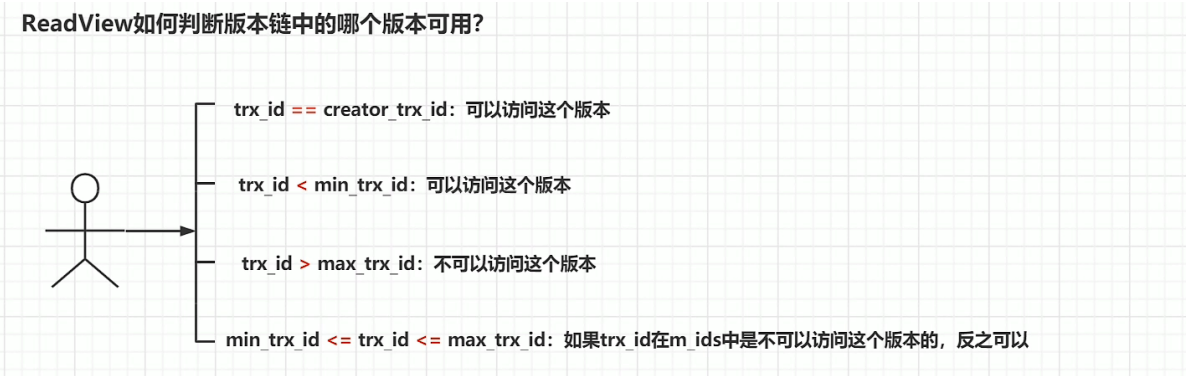
相当于告诉你什么事务的数据可见，什么事务数据不可见，从而构成乐观锁思想的MVCC机制

trx_id指的是当前系统中活跃的读写事务的id列表

min_trx_id指的是系统活跃的读写事务中最小的事务id

max_trx_id重点不是最大事务id而是系统应该分配给下个事务的id

creator_trx_id 表示生成该ReadView的事务的事务id



第一点说明 修改这条事务就是当前事务，所以可见

第二点 当前事务的id小于最小活跃事务的id 说明已经commit可见

第三点 大于则代表当前事务id是在readView生成后才出现，对当前事务不可见

第四点 在这个最小活跃事务id和最大活跃事务id之间，不可见，反之可见 譬如h活跃13567 我是2 那么可见 我是3就是不可见

慢查询

慢查询可以帮我们**定位执行慢的SQL语句。**我们可以通过设置long_query_time参数定义“慢"的阈值，如果SQL执行时间超过了
long_ query_ time,则会认为是慢查询。

```
#查看慢SQL是否开启
show variables like "slow_ query_log%";
#查看慢查询设定的阈值单位:秒
show variables like "long_query_time";
```

乐观锁和悲观锁

- 乐观锁：在操作数据的时候很乐观，觉得别的线程不会同时修改数据，所以不会上锁，但是在更新的时候会判断一下此期间别的线程有没有更新这个数据（适合读操作多的场景）
- 悲观锁：在操作数据的时候很悲观，觉得每次修改数据的时候，别的线程也同时在修改，所以在每次拿数据的时候都会上锁，这样别的线程就会阻塞直到前一个线程释放锁，CPU来唤醒等待的线程，它才能拿到锁（适合并发写操作多的场景）

死锁

死锁是指:一组互相竞争资源的线程，因互相等待，导致永久阻塞的现象

原因：互斥条件：共享线程x、y只能被一个线程占用（无法被破坏，因为锁本身就是通过互斥来解决线程安全问题的）

> 占有且等待：线程t1已经取得共享资源x，在等待共享资源y的时候，不释放共享资源x
>
> 不可抢占：其他线程不能强行抢占
>
> 循环等待

如何处理死锁

- 一次性申请所有的资源，这样就不存在等待了
- 占有部分资源的线程，进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源
- 线性申请，先申请资源序号小的，再申请资源序号大的

数据库中的死锁问题

- 通过innodb_ lock wait timeout 来设置超时时间，一直等待直到超时;InnoDB默认50s。
- 发起死锁检测，发现死锁之后，主动回滚死锁中的某一个事务，让其它事务继续执行。

日志

redo log

redolog是innodb特有的引擎日志

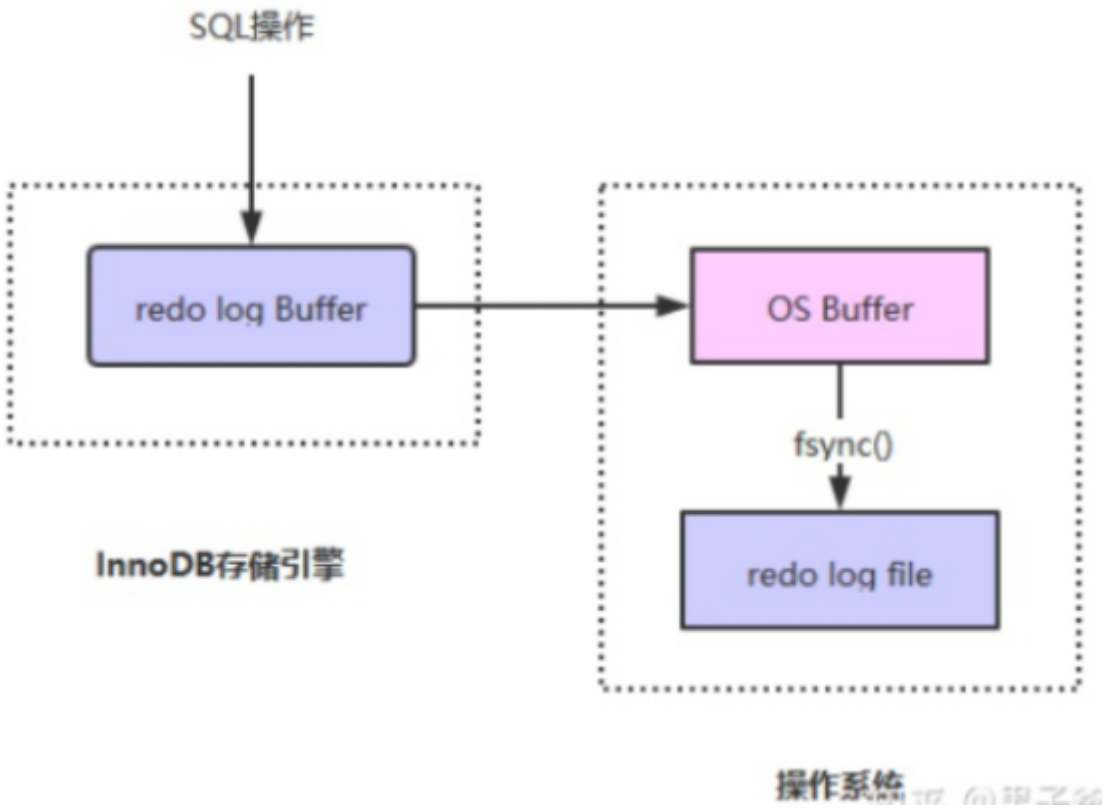
innodb引擎利用redolog可以保证提交事务数据持久性，解决事务提交后，服务器发生故障，也能将我们修改的数据持久化到磁盘

- 内存中的redo log buffer是日志缓冲区，这部分数据容易丢失

- 磁盘中的redo log fle是日志文件，这部分数据已经持久化磁盘，不容易丢失

Write-Ahead Logging

- 先写日志，再写磁盘。
- 对数据页，先在内存中修改，然后使用io顺序写的方式持久化到redo log文件;然后异步去处理redo log,将数据页的修改持久化到磁盘中，效率非常高。
  同步到磁盘里就意味着每次写操作就得产生随机写盘操作，速度很慢。



redo log file 写入方式

在Innodb中，redo log的大小是固定的，只能是以循环的方式进行写入了

- 当一个事务commit的时候，刚好发现redo log不够了，此时会先停下来处理redo log中的内容，然后在进行后续的操作，遇到这种情况时，整个事物响应会稍微慢一些。

undolog

undolog用来回滚行记录到某个版本。事务未提交之前，Undo保存了未提交之前的版本数据，Undo中的数据可作为数据旧版本快照供其他并发事务进行快照读。是为了实现事务的原子性而出现的产物,在MySQL innodb存储引擎中用来实现多版本并发控制。

MySQL中是如何实现事务提交和回滚的?
为了保证数据的持久性，数据库在执行SQL操作数据之前会先记录redo log和undo log

- redo log是重做日志，通常是物理日志，记录的是物理数据页的修改，它用来恢复提交后的物理数据页，用于保障已提交事务的持久性。
- undo log是回滚日志，用来回滚行记录到某个版本，undo log一般是逻辑日志，根据行的数据变化进行记录，用于保障未提交事务的原子性。
- redo/undo log都是写先写到日志缓冲区，再通过缓冲区写到磁盘日志文件中进行持久化保存。
  undo日志还有一个用途就是用来控制数据的多版本(MVCC) 。

binlog

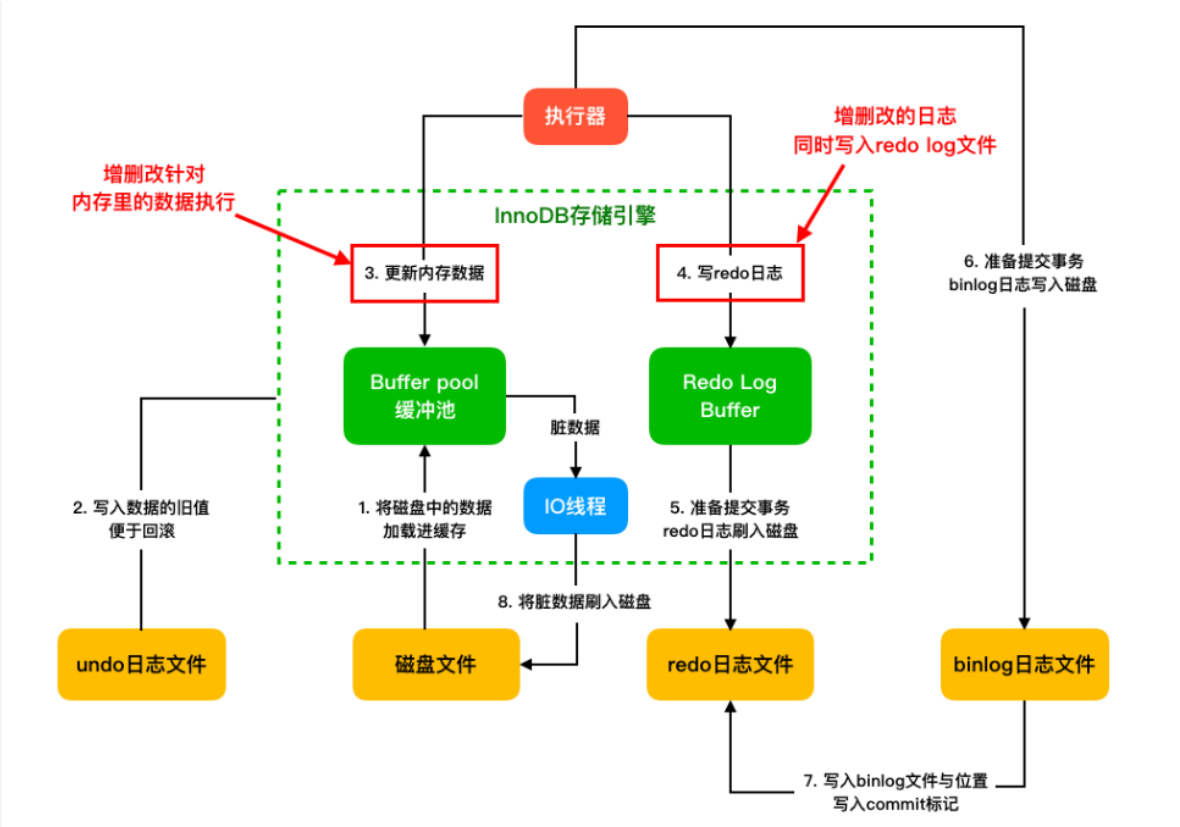
Binlog 二进制日志，它记录了所有引起或可能引起数据库改变的事件binlog不会记录SELECT和SHOW这类的操作，因为这类操作对数据本身并没有修改，如果updata操作没有造成数据变化，也是会记录binlog

binlog的作用：

- **数据异常恢复**
- **主从复制**（）
  - binlog线程：记录下所有改变了数据库数据的语句，放进master上的binlog中;
  - io线程：在使用start slave之后，负责从master上拉取binlog 内容，放进自己的relay log中;
  - sql执行线程： 执行relay log中的语句;

binlog

- 为了加快写的速度，mysql先把整个过程中产生的binlog日志先写到内存中的binlog cache缓存中， 后面再将binlog cache中内容一次性持久化到binlog文件中。
- 一个事务的binlog是不能被拆开的，因此不论这个事务多大，也要确保事务一次性写入。这就涉及到了binlog cache的保存问题。系统给binlog cache分配了一片内存， 每个线程一个 ,参数binlog. cache_ size 用于控制单个线程内binlog cache所占内存的大小。如果超过了这个参数规定的大小，就要暂存到磁盘。



Bin log与Redo log完整流程

- mysql收到start transaction后，生成一个全局的事务编号trx. id,比如trx_ id=10
- user_ id=666这个记录我们就叫r1, user, _id=888这个记录叫r2
- 找到r1记录所在的数据页p1，将其从磁盘中加载到内存中
- 在内存中对p1进行修改
- 将p1修改操作记录到redo log buffer中
- 将p1修改记录流水记录到binlog cache中

- 找到r2记录所在的数据页p2，将其从磁盘中加载到内存中
- 在内存中对p2进行修改
- 将p2修改操作记录到redo log buffer中
- 将p2修改记录流水记录到binlog cache中
- mysq收到commit指令
- 将redo log buffer携带trx_ id=10写入到redo log文件，持久化到磁盘，这步操作叫做redo log prepare

当我们要对某个数据进行修改的时候，我们就要先将数据所在的数据页从磁盘加载至内存，然后把数据的旧值写入undolog;文件中便于回滚。然后通过执行器对数据进行修改，将对数据的修改顺序写入redo log Buffer中，redolog日志写入磁盘处于prepare状态，然后将binlog日志中记录的事务的操作内容写入磁盘判断是否一致，如果一致redolog状态改成commit写入磁盘随即将数据持久化保存在磁盘中，不一致回滚操作。

Mysql的优化

一大表数据查询，如何优化

- 使用索引，优化索引；
- SQL优化；
- 缓存频繁查询的数据；
- 主从复制，读写分离；
- 分库分表，垂直拆分或者是水平拆分；
- 慢查询

主从复制

MySQL主从复制是指数据可以从一个MySQL数据库服务器主节点复制到一个或多个从节点。MySQL 默认采用异步复制方式，这样
从节点可以复制主数据库中的所有数据库或者特定的数据库、或者特定的表。

作用：

- 主服务器故障后，可以切换到从数据库继续工作，避免数据的丢失
- 提升I/O性能，业务量大了之后，数据库I/O访问频率越来越高，单机无法满足，做多库的存储，这样可以有效降低I/O访问的频率
- 读写分离，写请求分发到master主数据库，读请求分发到slave从数据库，数据库可以支持更大的并发

原理：

主库将变更写入 binlog 日志，然后从库连接到主库之后，从库有一个 IO 线程，将主库的 binlog 日志拷贝到自己本地，写入一个 relay 中继日志中。接着从库中有一个 SQL 线程会从中继日志中读取内容解析成sql语句，逐一执行，也就是在自己本地再次执行一遍 SQL，这样就可以保证自己跟主库的数据是一样的。

解决主库数据丢失问题：

- **半同步复制**：主库写入 binlog 日志之后，就会将强制立即将数据同步到从库，从库将日志写入自己本地的 relay log 之后，接着会返回一个 ack 给主库，主库接收到至少一个从库的 ack 之后才会认为写操作完成了。

解决主从同步延时问题：

- **并行复制**：从库开启多个线程，并行读取relay log 中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。

分库分表

分库分表的优点

单表数据量太大的时候会极大的影响到sql的性能。因此我们可以进行分库分表的操作。

分表后使用连接查询。

分库分表的优点就是：

- 使MySQL从单机到多机，抗多并发的能力更强；
- 拆分为多个库，数据库服务器的磁盘的使用率大大降低；
- 单表数据量减小，执行效率显著提升。

垂直拆分：垂直拆分的意思，就是把**一个有很多字段的表给拆分个成多个表，或者是多个库上去。每个库表的结构都不一样，包含的字段不一样。**一般来说，会将较少的访问频率很高的字段放到一个表里去，然后将较多的访问频率很低的字段放到另外一个个表里去。因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。

水平拆分：水平拆分是**把一个表的数据给弄到多个库的多个表里去，但是每个库的表结构都一样，只不过每个库表放的数据是不同的。**水平拆分的意义，就是将数据均匀放更多的库里，然后用多个库来抗更高的并发，还有就是用多个库的存储容量来进行扩容。