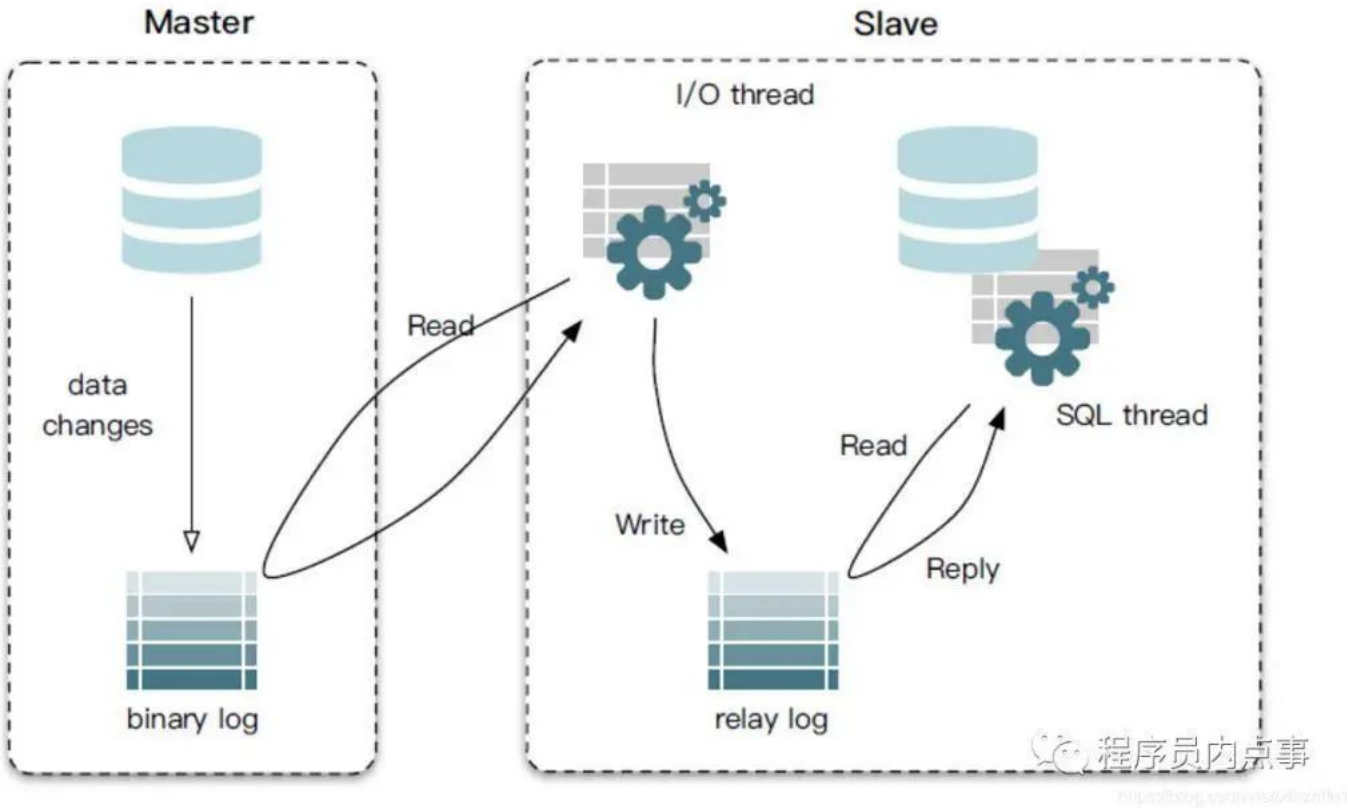
1.主从复制



复制的原理
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 线程会从中继日志读取 binlog，然后执行 binlog 日志中的内容，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。
主从同步的延迟
就是从库同步主库数据的过程是串行化的（单线程串行化的执行），所以会出现延迟，要过几十毫秒，甚至几百毫秒才能读取到。
半同步复制
也叫 semi-sync 复制，指的就是主库写入 binlog 日志之后，就会将强制此时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。
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。
解决主从复制延迟的问题
分库:将一个主库拆分为多个主库，每个主库的写并发就减少了几倍，此时主从延迟可以忽略不计。
并行复制:多个库并行复制。如果说某个库的写入并发就是特别高，单库写并发达到了 2000/s，并行复制还是没意义。
重写代码:写代码的同学，要慎重，插入数据时立马查询可能查不到。
强制走主库:如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库。
不推荐这种方法，你要是这么搞，读写分离的意义就丧失了。

2.mysql事务的实现原理
原子性：undolog来实现的 回滚日志是反向的操作
持久性：redolog来实现的
redo log 的存储是顺序存储，而缓存同步是随机操作。
缓存同步是以数据页为单位的，每次传输的数据大小大于redo log。
隔离性：读写锁+MVCC 实现的目的是为了高并发
级别越低的隔离级别可以执行越高的并发，但同时实现复杂度以及开销也越大。
MySQL隔离级别有以下四种（级别由低到高）：
READUNCOMMITED(未提交读) **脏读**
READCOMMITED(提交读) **不可重读以及幻读**
REPEATABLEREAD(可重复读) **幻读**
SERIALIZABLE (可重复读)
一致性：原子性、持久性、隔离性

3.MVCC的实现
系统版本号：一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。
事务版本号：事务开始时的系统版本号。

创建版本号：创建一行数据时，将当前系统版本号作为创建版本号赋值
删除版本号：删除一行数据时，将当前系统版本号作为删除版本号赋值

可重复读 是部分解决了幻读的问题
快照读

当前读

```
select * from T where number = 1 for update; 加排他锁
select * from T where number = 1 lock in share mode; 加共享锁
```

4.如何解决幻读

```
幻度现象：
Mysql官方给出的幻读解释是：只要在一个事务中，第二次select多出了row就算幻读。
a事务先select，b事务insert确实会加一个gap锁，但是如果b事务commit，这个gap锁就会释放（释放后a事务可以随意dml操作），a事务再select出来的结果在MVCC下还和第一次select一样，接着a事务不加条件地update，这个update会作用在所有行上（包括b事务新加的），a事务再次select就会出现b事务中的新行，并且这个新行已经被update修改了，实测在RR级别下确实如此。
如果这样理解的话，Mysql的RR级别确实防不住幻读

在快照读读情况下，mysql通过mvcc来避免幻读。
在当前读读情况下，mysql通过next-key来避免幻读。
select * from t where a=1;属于快照读
select * from t where a=1 lock in share mode;属于当前读
不能把快照读和当前读得到的结果不一样这种情况认为是幻读，这是两种不同的使用。所以我认为mysql的rr级别是解决了幻读的。
```

使用串行化读的隔离级别

MVCC+next-key locks：next-key locks由record locks(索引加锁) 和 gap locks(间隙锁，每次锁住的不光是需要使用的数据，还会锁住这些数据附近的数据)

5.什么时候开启事务?

不是begin语句就开启了事务，实际上是对数据进行了增删改查等操作后才开启了一个事务

6.意向锁：主要作用是处理行锁和表锁之间的矛盾，能够显示“某个事务正在某一行上持有了锁，或者准备去持有锁”。

7.Mysql死锁

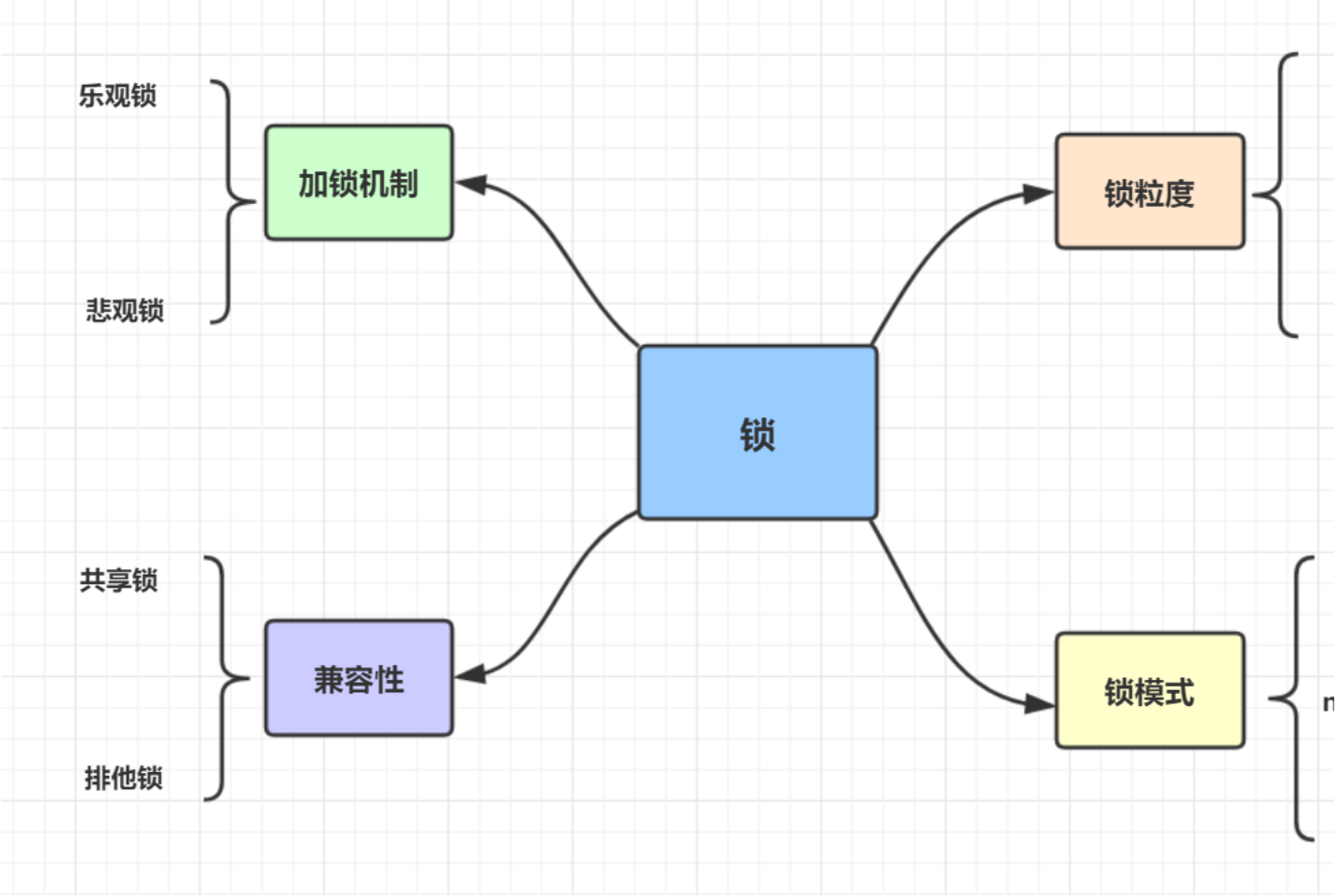
https://blog.csdn.net/tr1912/article/details/81668423

```
Deadlock found when trying to get lock; try restarting transaction

show engine innodb status

SHOW VARIABLES LIKE 'innodb_lock_wait_timeout';
把超时等待时间修改为5秒： SET innodb_lock_wait_timeout=5;

autocommit = 1 事务自动提交

死锁恢复
Mysql默认解决死锁的方式是将最少行级排他锁的事务进行回滚
外部锁的死锁检测
发生死锁后，InnoDB 一般都能自动检测到，并使一个事务释放锁并回退，另一个事务获得锁，继续完成事务。但在涉及外部锁，或涉及表锁的情况下，InnoDB 并不能完全自动检测到死锁， 这需要通过设置锁等待超时参数 innodb_lock_wait_timeout 来解决
死锁影响性能
死锁会影响性能而不是会产生严重错误，因为InnoDB会自动检测死锁状况并回滚其中一个受影响的事务。在高并发系统上，当许多线程等待同一个锁时，死锁检测可能导致速度变慢。 有时当发生死锁时，禁用死锁检测（使用innodb_deadlock_detect配置选项）可能会更有效，这时可以依赖innodb_lock_wait_timeout设置进行事务回滚。

InnoDB避免死锁
* 为了在单个InnoDB表上执行多个并发写入操作时避免死锁，可以在事务开始时通过为预期要修改的每个元祖（行）使用SELECT ... FOR UPDATE语句来获取必要的锁，即使这些行的更改语句是在之后才执行的。
* 在事务中，如果要更新记录，应该直接申请足够级别的锁，即排他锁，而不应先申请共享锁、更新时再申请排他锁，因为这时候当用户再申请排他锁时，其他事务可能又已经获得了相同记录的共享锁，从而造成锁冲突，甚至死锁
* 如果事务需要修改或锁定多个表，则应在每个事务中以相同的顺序使用加锁语句。在应用中，如果不同的程序会并发存取多个表，应尽量约定以相同的顺序来访问表，这样可以大大降低产生死锁的机会
* 通过SELECT ... LOCK IN SHARE MODE获取行的读锁后，如果当前事务再需要对该记录进行更新操作，则很有可能造成死锁。
* 改变事务隔离级别
```

8.mysql什么时候不会走索引

```
mysql走索引查询的数据占全表30%以上时，mysql就不会选择走索引
(1) 组合索引未使用最左前缀，例如组合索引（A，B），where B=b不会使用索引；
(2) like未使用最左前缀，where A like '%China';
(3) 搜索一个索引而在另一个索引上做order by，where A=a order by B，只使用A上的索引，因为查询只使用一个索引 ；
(4) or会使索引失效。如果查询字段相同，也可以使用索引。例如where A=a1 or A=a2（生效），where A=a or B=b（失效）
(5) 如果列类型是字符串，要使用引号。例如where A='China'，否则索引失效（会进行类型转换）；
(6) 在索引列上的操作，函数（upper()等）、or、！=(<>)、not in等
```

9.锁分类


乐观锁
加锁机制
悲观锁
锁粒度
锁
共享锁
兼容性
排他锁
锁模式