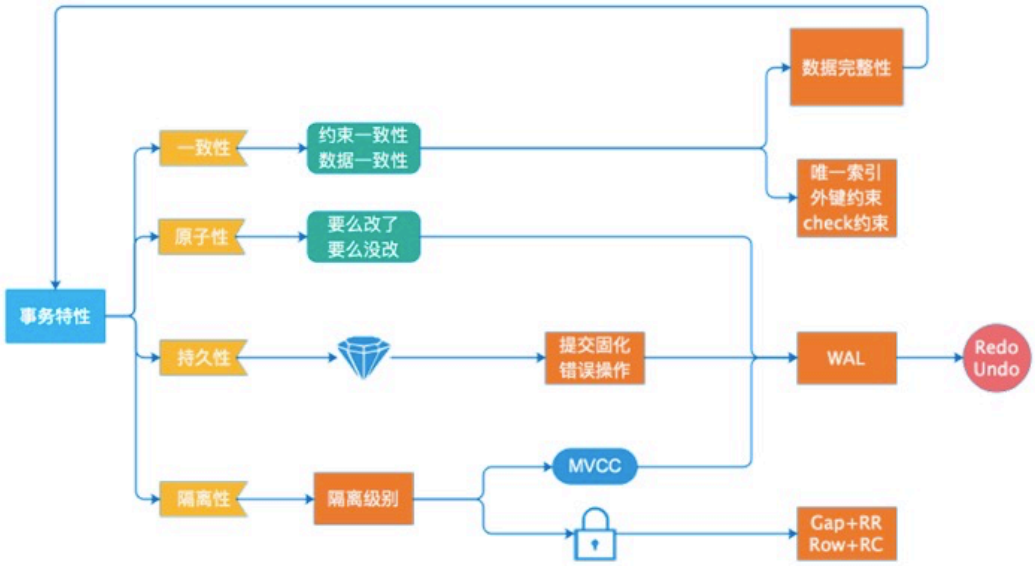
Mysql 事务 与 锁机制

拉钩高性能 mysql

一个逻辑工作单元要成为事务，在关系型数据库管理系统中，必须满足 4 个特性，即所谓的 ACID：原子性、一致性、隔离性和持久性。

- 一致性：事务开始之前和事务结束之后，数据库的完整性限制未被破坏。
- 原子性：事务的所有操作，要么全部完成，要么全部不完成，不会结束在某个中间环节。
- 持久性：事务完成之后，事务所做的修改进行持久化保存，不会丢失。
- 隔离性：当多个事务并发访问数据库中的同一数据时，所表现出来的相互关系。

ACID 及它们之间的关系如下图所示，比如 4 个特性中有 3 个与 WAL 有关系，都需要通过 Redo、Undo 日志来保证等。



一致性：

约束一致性： mysql 只支持 创建表结构时的外键、 唯一索引约束
数据一致性： 综合性的规定，由原子性、持久性、隔离性共同保证的结果 不是单一技术

原子性：

Mysql 通过 WAL （write ahread log） 实现。要么改要么不改，感受不到中间态。
WAL 如何保证原子?
例如： 事务提交，数据修改就生效了，buffer pool 的脏页没刷盘，就需要 redo log

恢复数据
事务未提交，buffer pool 刷盘了， 要通过undo log， undo log 又是redo log 保证的
所以最终是通过redo log 的WAL 机制

持久性：
事务一旦提交，对数据库的修改是永久的， 通过原子性，即使宕机，也可以从逻辑上将数据找回来重新写入物理存储

隔离性：
一个事务内部操作及使用数据对其他并发事务是隔离的。 锁 + 多版本控制符合隔离性

并发事务控制
单版本 控制 ——— 锁 （用独占的形式保证只有一个版本的情况下事务间相互隔离）

锁的实现与隔离级别有关，RR （repeated read）下，mysql 为了解决幻读， 以牺牲并行度为代价， 通过Gap锁（面试没答到的点）
来防止数据写入， 但是由于并行度问题， 经常死锁。 现在的Row模式 （是什么？） 可以避免冲突和死锁
所以推荐默认 Row+ Rc（read commited）模式

多版本控制———— MVCC

为了实现并发数据访问，对数据进行多版本处理，通过事务可见效保证事务看见自己应该看到对数据版本

如何生成多版本?
每次对数据库的修改， 都会在undo log 中记录当前修改记录事务号 & 修改前数据状态的存储地址 （Roll_PTR）
可以在必要时候回滚到老版本。 例如A、b两事务，新事务未提交，根据原子性读不到，但是可以从回滚段中拿到老版本数据 从而形成多版本

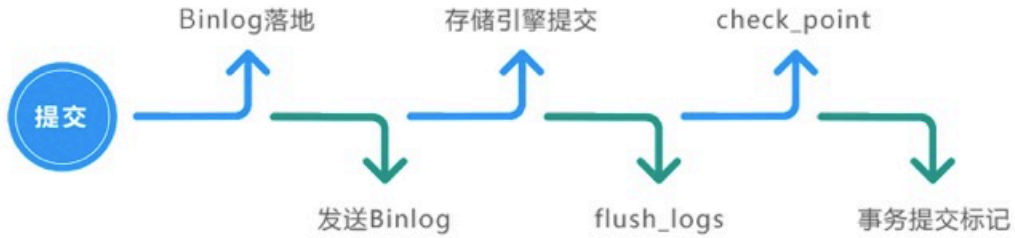
原子性的技术实现：
每一个写事务，都会修改buffer pool 产生相应 redo log， 都记录到 ib_logfiles 中。 因为WAL ，所以事务是顺序记录的

Buffer pool 页刷到磁盘前， 都会先写 日志文件：
1、 buffer pool 刷失败， 数据库挂了， 重启后，通过redo log 可以恢复，保证脏页写下去的数据不丢失， 所以要保证redolog 先写

2、 因为buffer pool 空间有限， 载入新页时， 要从LRU 链表淘汰一些页， 这些页必须要刷盘后，才可以使用，
而刷盘需要保证对应的 LSN 日志先写到 ib_logfiles ．不写，数据对应的回滚日志可能不存在，未提交的事务就无法回滚，从而不能保证原子性

持久性技术实现：

再来看持久性，如下图所示，一个“提交”动作触发的操作有：binlog 落地、发送 binlog、存储引擎提交、flush_logs， check_point、事务提交标记等。这些都是数据库保证其数据完整性、持久性的手段。



那这些操作如何做到持久性呢？前面讲过，通过原子性可以保证逻辑上的持久性，通过存储引擎的数据刷盘可以保证物理上的持久性。这个过程与前面提到的 Redo 日志、事务状态、数据库恢复、参数 innodb_flush_log_at_trx_commit 有关，还与 binlog 有关。这里多提一句，在数据库恢复时，如果发现某事务的状态为 Prepare，则会在 binlog 中找到对应的事务并将其在数据库中重新执行一遍，来保证数据库的持久性。

隔离性的技术：

接下来看隔离性，InnoDB 支持的隔离性有 4 种，隔离性从低到高分别为：读未提交、读提交、可重复读、可串行化。
读未提交（RU，Read Uncommitted）。它能读到一个事务的中间过程，违背了 ACID 特性，存在脏读的问题，所以基本不会用到，可以忽略。

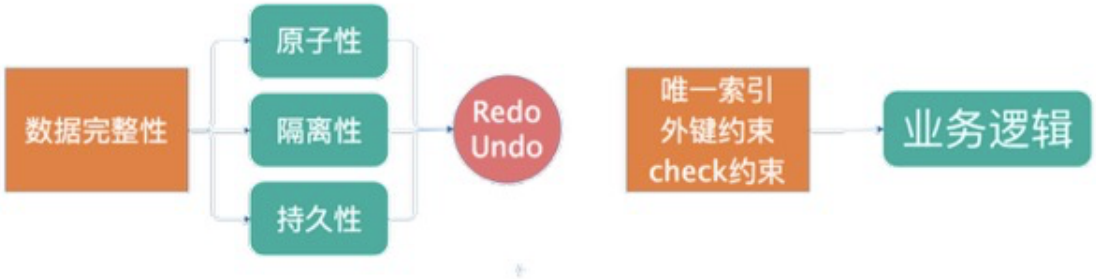
读提交（RC，Read Committed）。它表示如果其他事务已经提交，那么我们就可以看到，这也是一种最普遍适用的级别。但由于一些历史原因，可能 RC 在生产环境中用的并不多。

可重复读（RR，Repeatable Read），是目前被使用得最多的一种级别。其特点是有 Gap 锁、目前还是默认的级别、在这种级别下会经常发生死锁、低并发等问题。

可串行化，这种实现方式，其实已经并不是多版本了，又回到了单版本的状态，因为它所有的实现都是通过锁来实现的。

一致性：

致性可以归纳为数据的完整性。根据前文可知，数据的完整性是通过其他三个特性来保证的，包括原子性、隔离性、持久性，而这三个特性，又是通过 Redo/Undo 来保证的



MVCC的实现原理：
注意：MVCC 只在 Read Commited 和 Repeatable Read 两种隔离级别下工作

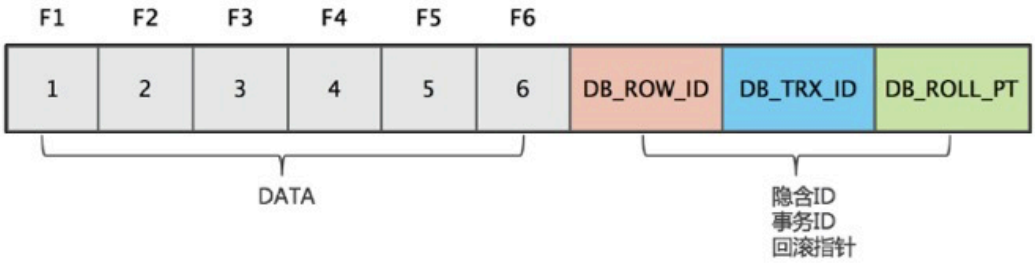
InnoDB 实现的是基于多版本的并发控制协议 MVCC，而不是基于锁的并发

MVCC 最大的好处是读不加锁， 读写不冲突。

快照读：读取的是记录的可见版本（有可能是历史版本），不用加锁。 简单的 select

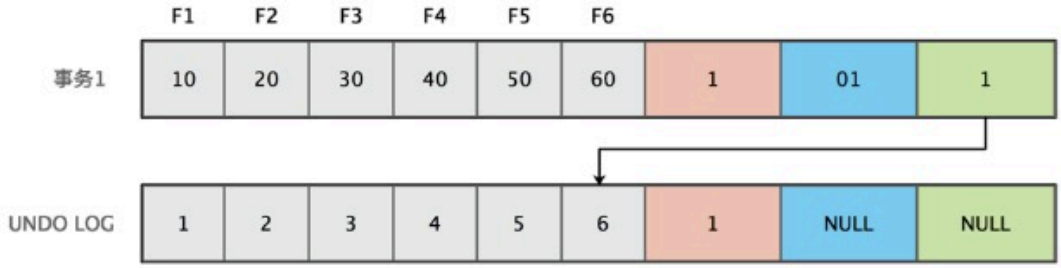
当前读：读取的是记录的最新版本，并且当前读返回的记录，都会加锁，保证其他事务不会再并发修改这条记录。 特殊的读（插入、更新、删除）

- 假设 F1~F6 是表中字段的名字，1~6 是其对应的数据。后面三个隐含字段分别对应该行的隐含 ID、事务号和回滚指针，如下图所示。



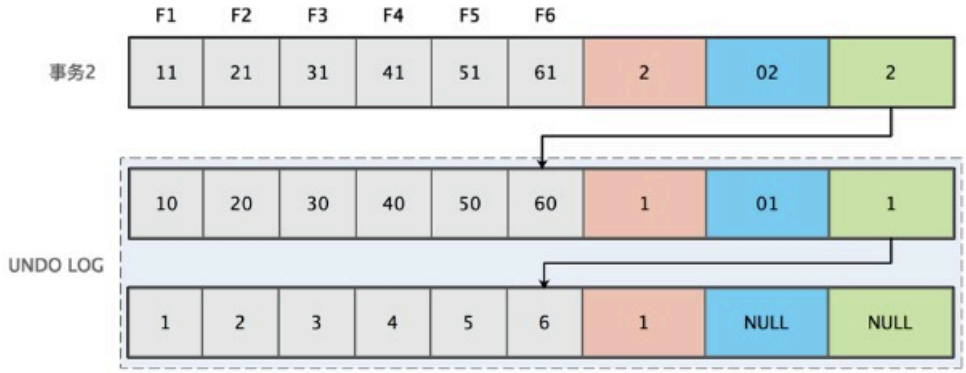
- 隐含 ID（DB_ROW_ID），6 个字节，当由 InnoDB 自动产生聚集索引时，聚集索引包括这个 DB_ROW_ID 的值。
- 事务号（DB_TRX_ID），6 个字节，标记了最新更新这条行记录的 Transaction ID，每处理一个事务，其值自动 +1。
- 回滚指针（DB_ROLL_PT），7 个字节，指向当前记录项的 Rollback Segment 的 Undo log记录，通过这个指针才能查找之前版本的数据。

刚插入时： ID 为1，另外两个 null
然后进行更改



- 用排他锁锁定该行；记录 Redo log；
- 把该行修改前的值复制到 Undo log，即图中下面的行；
- 修改当前行的值，填写事务编号，使回滚指针指向 Undo log 中修改前的行。

接下来，与事务 1 相同，此时 Undo log 中有两行记录，并且通过回滚指针连在一起。因此，如果 Undo log 一直不删除，则会通过当前记录的回滚指针回溯到该行创建时的初始内容，所幸的是在 InnoDB 中存在 purge 线程，它会查询那些比现在最老的活动事务还早的 Undo log，并删除它们，从而保证 Undo log 文件不会无限增长，如下图所示。



Mysql 锁机制：

Innodb 中的锁： 行锁、 表锁

行锁：
共享锁 S： 允许一个事务读一行，阻止其他事务活动相同数据集排他锁
排他锁 X： 允许获得排他锁的事务更新数据， 阻止其他获得排他锁、 共享锁

表锁又分为三种。
意向共享锁（IS）：事务计划给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。
意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。
自增锁（AUTO-INC Locks）：特殊表锁，自增长计数器通过该“锁”来获得子增长计数器最大的计数值。

在加行锁之前必须先获得表级意向锁，否则等待 innodb_lock_wait_timeout 超时后

根据innodb_rollback_on_timeout 决定是否回滚事务。

InnoDB 行锁：

行锁实现 主要是通过 对索引数据页上记录加锁

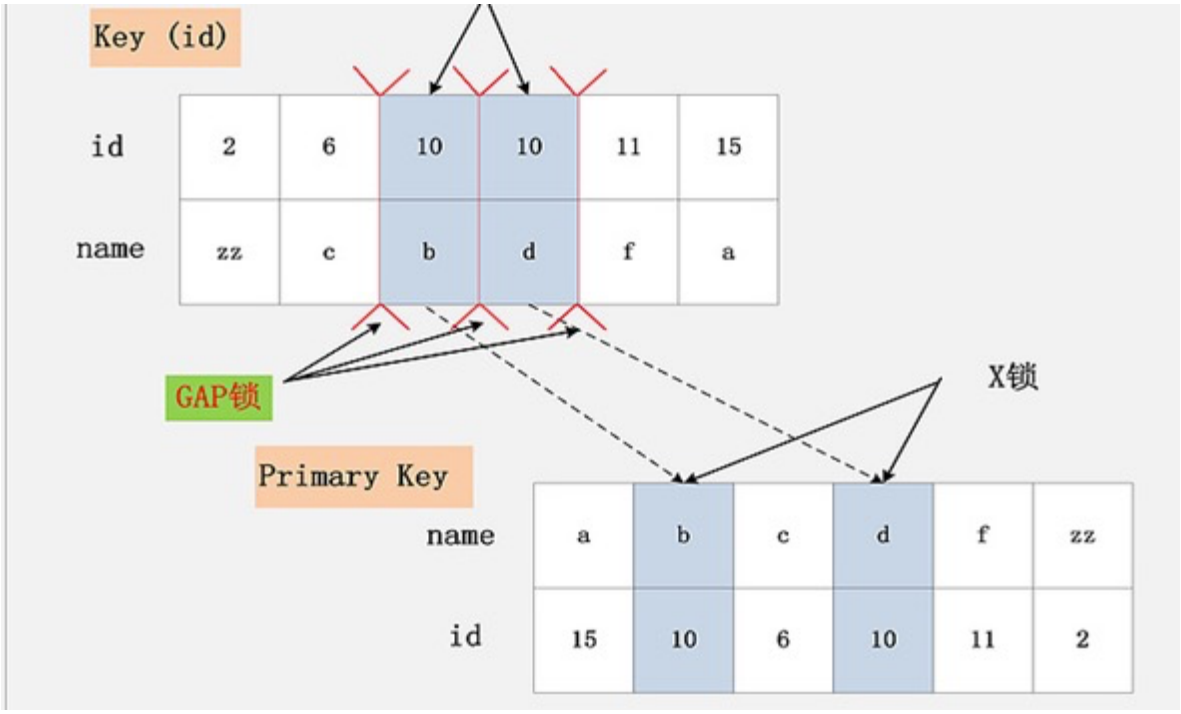
Record lock： 单个行记录锁 不锁gap
Gap lock： 间隙锁， 锁定一个范围，不含记录本身（不锁数据，仅仅锁数据前面的gap）
next_key lock： 同时锁住数据 & 数据前面gap

从四个场景分析加锁行为
update t1 set name='XX' where id=10。

主键 + RR。 id=10的主键索引记录加X锁

唯一键 + RR。 先在唯一索引id上加锁， 再在id=10的主键索引记录上加X锁，id=10不存在，加间隙锁

非唯一键 + RR。 先通过id=10 找到第一个满足条件记录，记录加x锁，并在范围内加gap 防止幻读，主键索引name加记录的x锁
到id=11 发现没有满足记录到，不需要加x锁，但要加一个gap lock



无索引 + RR。 锁表 所有行、间隙都加X锁

Innodb死锁：

更新 where 用索引
加锁索引准确，缩小锁定范围
减少范围更新，尤其非主键、唯一索引上的范围更新
控制事务大小 减少锁定数据量和锁定时间长度 （innodb_row_lock_time_avg）
加锁顺序一致，尽可能一次性锁定所有需要的数据行