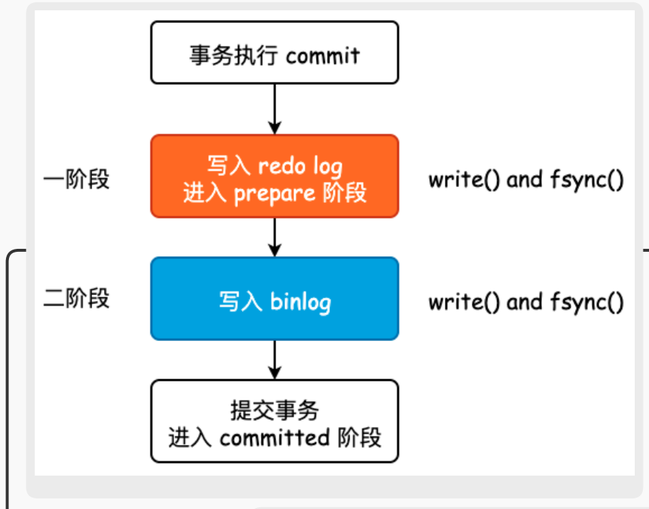
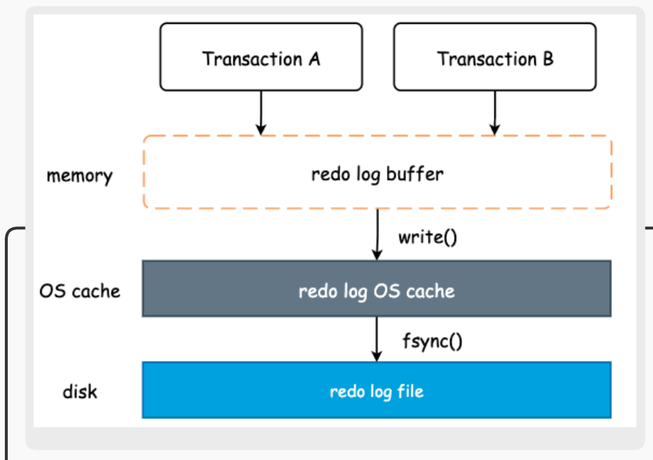
InnoDB Group Commit

2PC



在 InnoDB 的两阶段提交中，不管是 redo log 还是 binlog，都需要确切地将数据写入硬盘中，而不是 OS cache

redo log 写入机制



redo log buffer 为全局共有

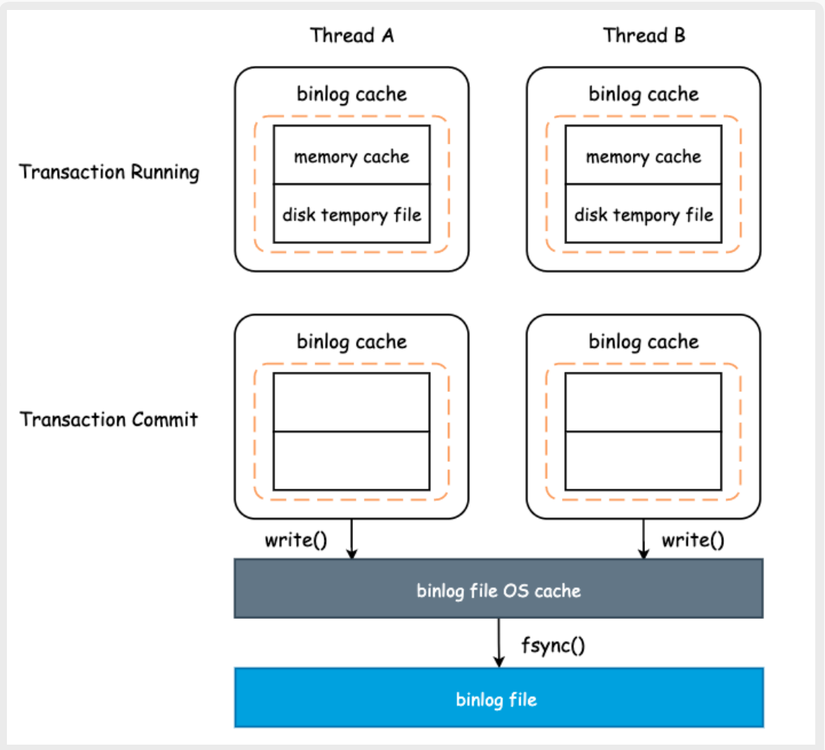
前面我们已经提到了 Linux 中的文件 I/O 分为两部分: 调用 write() 写入内核高速缓存，以及调用 fsync() 将其持久化至硬盘。wirte() 调用速度非常快，因为仅仅是将数据从用户空间拷贝至内核空间而已，而 fsync() 相对来说则慢很多

redo log buffer 刷新时机

- InnoDB 后台线程定期扫描，将 redo log buffer 中的 redo log 持久化至硬盘
- **redo log buffer 所占用的空间达到了 innodb_log_buffer_size 的一半，此时后台线程会将 buffer 中的内容调用 write() 写入至内核高速缓冲区中，但是由于事务没有提交，所以不会调用 fsync()**
- **当事务提交时，InnoDB 不仅仅会将相应的 redo log 写入至内核缓冲区，同时也会调用 fsync() 进行持久化**

也就是说，redo log file 中会存在尚未提交的事务，但是这并不影响 InnoDB 数据的准确性，因为这些 page 页并没有进入 prepare 阶段

binlog 写入机制



binlog 的写入机制要稍微复杂一些，原因在于 binlog 中的 binlog cache 是每个线程一个

当执行的事务比较大并且 memory cache 无法容纳的话，会将这部分的数据暂存至硬盘中。为什么不能直接调用 write() 将数据写入高速缓冲区呢? 其原因就在于 binlog 必须有序且完整，如果进行分批写入的话，就会出现同一个事务的 binlog 出现在文件的不同区域

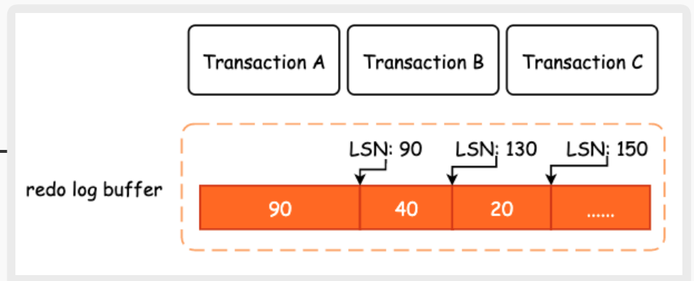
当事务提交时，redo log 写入完毕后会将 binlog cache 中的内容（内存+硬盘临时数据）一起调用 write() 写入 OS cache，并调用 fsync() 进行持久化，同时清空 binlog cache 中的全部内容

可以看到，在上述的 2PC 提交中，一个事务的提交一共涉及到了 2 次 fsync() 的调用。如果 MySQL 的 TPS 是 2000 的话，那么使用磁盘工具检测的结果应为 4000，但实际上 MySQL 在提交一个事务时只进行了一次 fsync() 系统调用

Group Commit

LSN

LSN 是 Log Sequence Number 的缩写，特指 redo log 的写入点。LSN 单调递增，每次写入长度为 N 的 redo log，LSN 的值就会增加 N

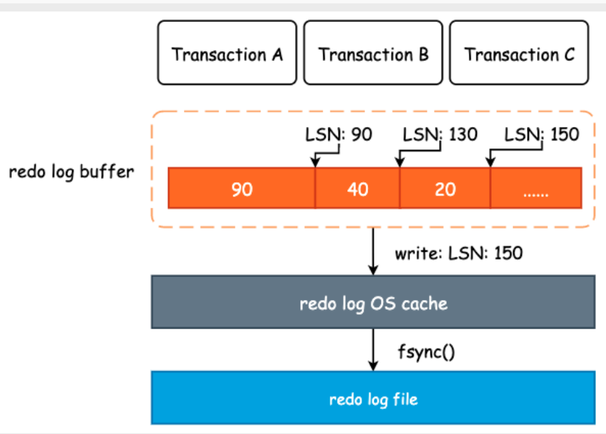


如上图所示，假设有 3 个事务并发执行，其中 A 写入长度为 90 的 redo log，B 事务写入长度为 40 的 redo log，C 事务写入长度为 20 的 redo log，LSN 则按照 redo log 的长度进行增长

在没有 Group Commit 的情况下，当 A 事务 commit 时，只会将 LSN 为 90 的 redo log fsync 至硬盘

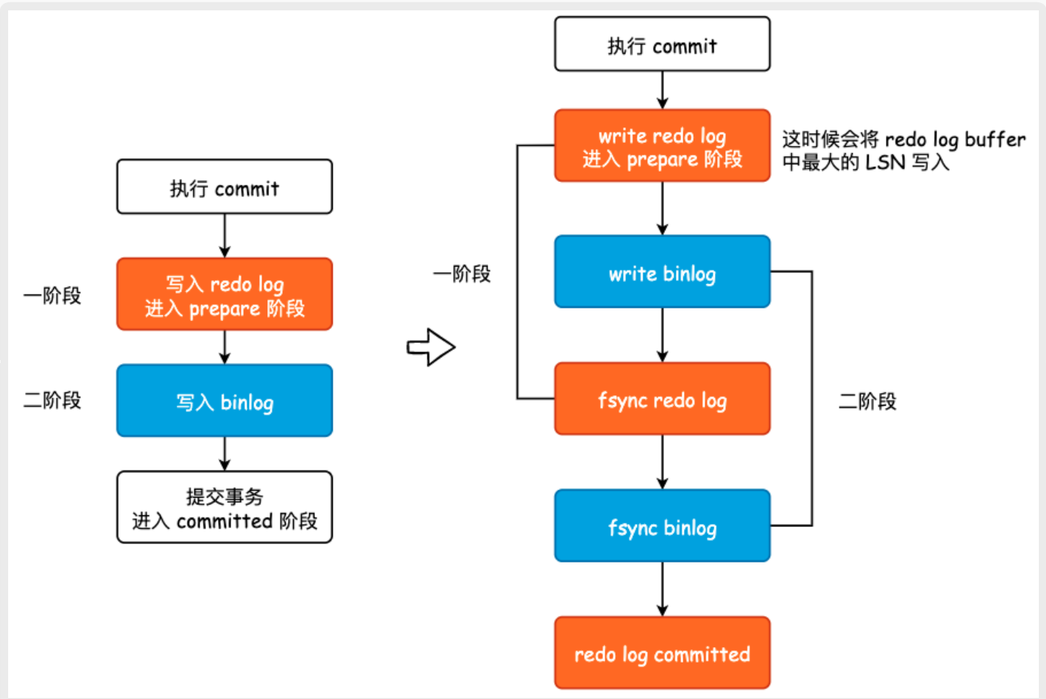
redo log group commit

redo log 的组提交其实就是将 redo log buffer 中的那个最大的 LSN 之前的所有 redo log 都持久化至硬盘中。事务到底有没有真正的提交，redo log 并不关心，反正最后是通过事务的 commit 标志位来决定当前数据是否真正被提交



和 Raft 协议中的日志复制有些类似。Leader 节点在写入数据时，会将日志发送给所有的 Follower，Follower 进行日志的追加，但是并不更新该数据的标志位，只有 Leader 收到大多数 Follower 的 ACK 以后，才会告诉 Follower Append 刚才那条日志

既然 redo log 中的内容和事务提不提交没有任何关系，就算事务没有提交依然可以将修改持久化至硬盘中，那么 InnoDB 就利用了该特性进行了一个有趣的优化



MySQL 将 redo log 和 binlog 的 write()、fsync() 这两个系统调用拆开，其目的就是为了让 binlog 能够累积到更多，从而一次 fsync() 能够将更多的 binlog 持久化至硬盘

同时我们也可以看到，这种两阶段提交的方式与系统并发度有关，能够并发执行的事务越多，那么调用 fsync() 的次数就会越少。相反，如果锁争抢非常严重的话，每次都只能有一个事务进入 redo log buffer 中，那么这时候组提交的效率就会回到最原始的版本中

提升 binlog Group Commit 的效果

设置 binlog_group_commit_sync_delay，表示延迟多少微秒以后才调用 binlog 的 fsync()

设置 binlog_group_commit_sync_no_delay_count ，表示累积多少次以后才调用 binlog 的 fsync()

5.7 版本的并行复制

MySQL 主从复制中的多线程复制一直是备受业界关注的问题，从库是否可以使用多线程执行 binlog 中的内容将决定了主库和从库之间的延迟量

在 MySQL 5.7 中，当 slave-parallel-type 配置项为 LOGICAL_CLOCK 时，就选用了基于 Group Commit 的并行复制方式

LOGICAL_CLOCK 利用了这样的一个事实: 多个事务如果能够同时进入 prepare 阶段，那么这些事务一定已经经过了锁校验，它们所更新、修改或者删除的数据一定是不会发生冲突的，换而言之，它们是可以并发执行的

同时，处于 prepare 和 committed 状态之间的事务，其实也是可以并发执行的。因此，从库将会对这些事务进行并发执行，这些事务全部处理完毕后，再转而并发地执行下一批事务