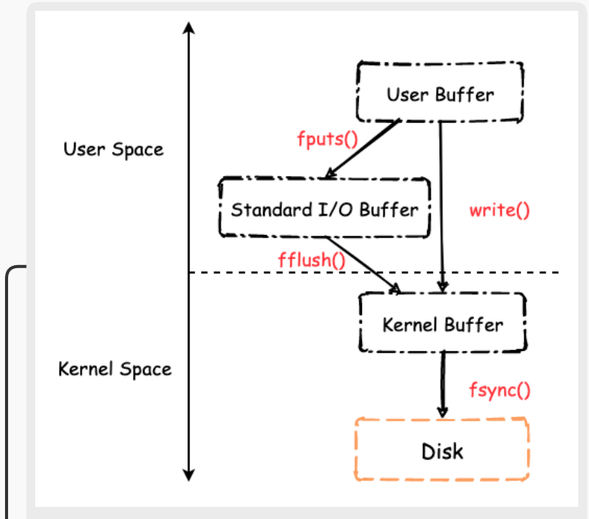
InnoDB-WAL

Linux 文件 I/O



在 Linux 系统下，对文件内容的写入均会首先首先写入至 Kernel Buffer，也就是内核高速缓冲区。应用程序要么主动调用 fsync() 将文件元数据和内容数据一起强制刷新至硬盘中，要么由内核 I/O 线程定时地将缓冲区内容刷新至硬盘

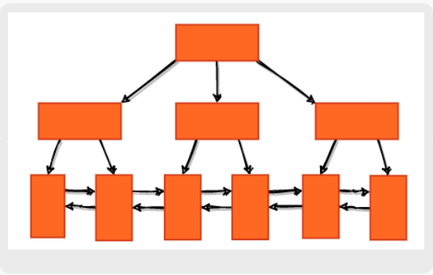
前者由进程主动控制，能够保证数据的持久性，但是由于每次写数据都需要进行硬盘 I/O 操作，所以整体性能不会很高。后者由内核定时任务定期执行，能够提高进程的 I/O 效率，但是当系统崩溃时，可能会丢失部分 Kernel Buffer 中的所有数据

困境
- 每次写入数据均调用 fsync() —— 数据能确保落入硬盘，但是效率很低
- 将 fsync() 的调用交给 OS 定时任务 —— 效率很高，但是数据不能保证一定落入硬盘

WAL

WAL 是 Write Ahead Log 的缩写，也就是当我们更新数据库数据时，首先将数据写入日志，然后再写入至真正的 DB 文件中，其目的是为了解决数据库写入效率问题

前因



MySQL 聚簇索引大致上如左图所示，B+Tree 节点与节点的指针使用文件偏移量表示

由于 B+Tree 的特性，所有的行数据均保存在其叶子节点中，那么当我们创建、更新一条数据时，就需要从根节点出发，沿着 B+Tree 的路径一步一步的到达叶子节点，然后进行插入、更新操作。在最坏情况下，每一层都需要进行一次随机的硬盘 I/O

而对于插入新数据而言，还可能会涉及到 B+Tree 节点的页分裂问题，原因在于 InnoDB 的 Page 大小固定为 16KB

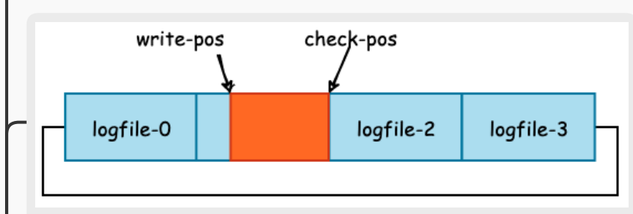
因此，插入和更新一条数据并不像 Linux 文件 I/O 图所绘制的那样，简单的执行下 fsync() 就可以了，内部其实是一个非常复杂并且相对于直接内存操作而言是一个很耗时的动作

所以，如果我们每次插入、更新数据都把数据写入至 .ibd 文件的话，数据库的性能和效率都会大打折扣

后果 —— 为了解决写入性能问题，以及保证数据的持久性，InnoDB 引入了 WAL 机制，更确切的说，就是 redo log

redo log

InnoDB redo log 是一个顺序写入的、大小固定的环形日志



假设我们配置了 4 个 redo log 文件，每个文件大小为 1 GB，那么我们总计可以写入 4 GB 的内容

数据写入时首先往 logfile-0 写入，当写到 logfile-3 并且没有空间时，又回到 logfile-0 开始写入

因此需要维护两个指针。一个是 write-pos，表示当前 redo log 的写入位置， 另一个是 check-pos，用于当前要擦除的位置，同样是往后推移且循环的，擦除记录的话就需要把数据持久化至 .ibd 文件中

write-pos 到 check-pos 中间的位置就是我们当前能够写入的空间。如果 write-pos 追上了 check-pos 的话，就需要把 check-pos 往后挪挪，也就是把一部分数据给持久化到 DB 文件中

我们可以不用太关心 redo log 文件中记录的具体格式，只需要知道它记录了对物理数据页的修改即可

日志与数据写入
- 从上面 redo log 的格式我们可以看出，redo log 的写入是顺序写入的，不需要找到某一个具体的索引位置，而是简单的从 write-pos 位置追加
- 那么也就是说，当一个写事务或者更新事务执行时，InnoDB 首先取出对应的 Page，然后进行修改。当事务提交时，将位于内存中的 redo log buffer 强制刷新至硬盘中，如果不考虑 binlog 的话，我们可以认为事务执行可以返回成功了
- 再然后，可由 InnoDB 的 Master Thread 定时地将缓冲池中的脏页，也就是上面儿我们修改的页，刷新至磁盘，此时被修改数据真正的写入至 .ibd 文件

因为顺序 I/O 很快，所以使用 redo log 既能够提高系统运行效率，同时也能够保证数据的准确性，因为事务提交时，redo log 必须被刷新至硬盘中 —— 虽然 fsync() redo log buffer 的时机可以通过配置项进行修改，当时强烈建议不要对其进行改动

binlog

binlog 的作用主要有两个
- 记录 MySQL Table 中的逻辑操作，比如更新了哪条数据，对这条数据做了哪些更新，可用于恢复因误删库导致的数据丢失
- 用于主从复制，主库和从库之间利用 binlog 进行数据复制，主库执行什么语句、进行何种更新，从库跟着执行

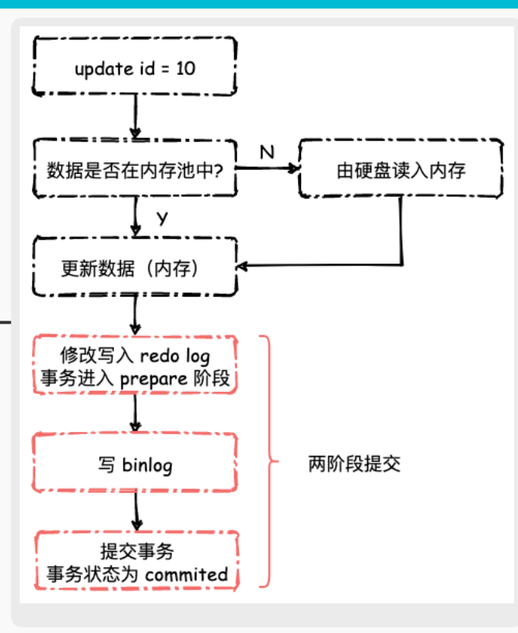
因此，当我们执行一个事务时，还需要写入 binlog，并且我们需要保证 redo log 和 binlog 之前的一致性

binlog 的日志格式主要有两种，Statement 和 Row
- Statement
  - 仅记录执行的 SQL 语句，例如 UPDATE table set name = "smart" WHERE id = 1，占用硬盘空间较少
  - 在主从复制中如果 SQL 语句调用了系统函数的话，很容易出现主从不一致的情况，因此很少使用
- Row —— Row 格式的 binlog 会记录两条数据，更新前的更新后的，非常详细，便于日志的恢复。但是占用硬盘空间较多，主从复制时的网络传输开销也会更多一些

到底使用哪种格式的 binlog，需要视情况而定

实际上 Row 格式的 binlog 也有 3 中子格式，分别为 FULL、MINIMAL 和 NOBLOB。FULL 记录数据修改的全部上下文，MINIMAL 则仅记录被修改字段的前后值

两阶段提交



两阶段提交的目的就是让 redo log 和 binlog 在逻辑上保持一致

有了两阶段提交以后，MySQL 在崩溃重启时可根据 redo log 和 binlog 之间的差异进行按需恢复/回滚事务。开发运维人员可以根据 binlog + 定时备份将 InnoDB 中的数据恢复至任意时刻

总结

redo log 的作用 —— 提高 InnoDB 存储引擎写入数据的效率

redo log 和 binlog 的区别
- redo log 中记录了 InnoDB Page 的具体修改，也就是物理页。binlog 中记录的是逻辑日志
- redo log 为 InnoDB 引擎所特有，而 binlog 则是在 MySQL Server 层所实现的，任何一个存储引擎都可以写入 binlog
- redo log 大小固定，本质上是一个"循环队列"，而 binlog 则可以一直往下写，并创建新的 binlog 文件

两阶段提交的目的是为了使得 redo log 和 binlog 在逻辑上保持一致。如果关闭 MySQL 的 binblog 记录功能，也就没有了两阶段提交，同时也就失去了数据恢复、主从复制等功能。当然，这么做肯定会有性能的提升

WAL 的本质作用还是优化数据库写入时的性能，将数据写入 DB 则由另外的线程异步进行 —— Postgres、etcd 等数据库均使用了 WAL