为什么要用WAL（Write-Ahead Logging）机制

磁盘的写操作是随机IO，比较耗性能，所以如果把每一次的更新操作都先写入log中，那么就成了顺序写操作，实际更新操作由后台线程再根据log异步写入。这样对于client端，延迟就降低了。并且，由于顺序写入大概率是在一个磁盘块内，这样产生的io次数也大大降低。所以WAL的核心在于将随机写转变为了顺序写，降低了客户端的延迟，提升了吞吐量。

WAL

WAL 的全称是 Write-Ahead Logging，它的关键点就是先写日志，再写磁盘，也就是“先写粉板，等不忙的时候再写账本”。

WAL引入了什么问题

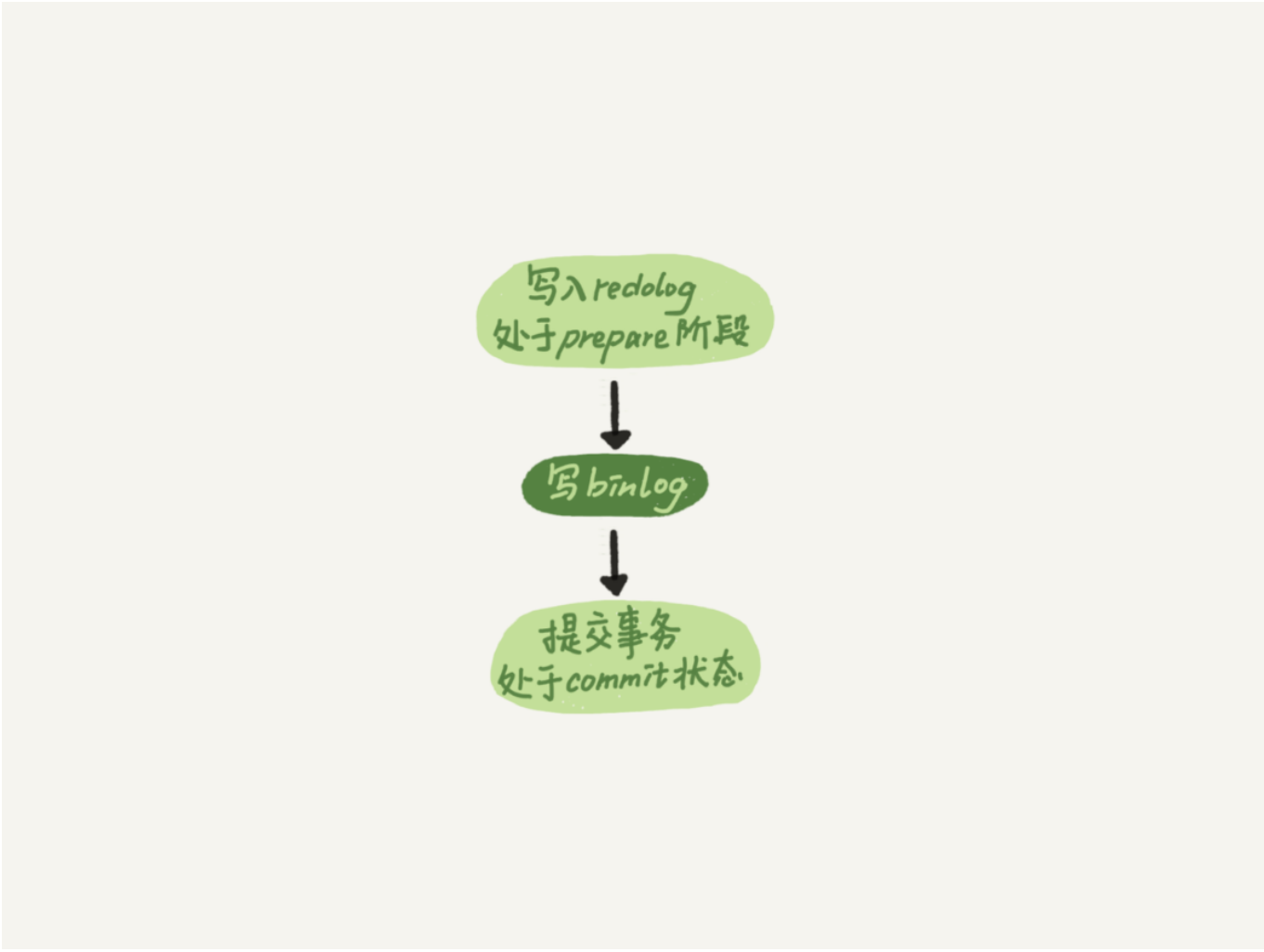
1. 日志刷盘问题

   由于所有对数据的修改都需要写日志，当并发量很大的时候，必然会导致日志的写入量也很大，为了性能考虑，往往需要先写到一个日志缓冲区，然后再按照一定规则刷入磁盘，此外日志缓冲区大小有限，而用户会源源不断的生产日志，数据库需要不断的把缓存区中的日志刷入磁盘，缓存区才可以复用，由此可见，这里构成了一个典型的生产者和消费者模型。

2. 数据刷盘问题

   在用户收到操作成功的时候，用户的数据不一定已经被持久化了，很有可能修改还没有落盘，这就需要数据库有一套刷数据的机制，专业术语叫做刷脏页算法。脏页(内存中被修改的但是还没落盘的数据页)在源源不断的产生，然后要持续的刷入磁盘，这里又凑成一个生产者消费者模型，影响数据库的性能。

两阶段提交



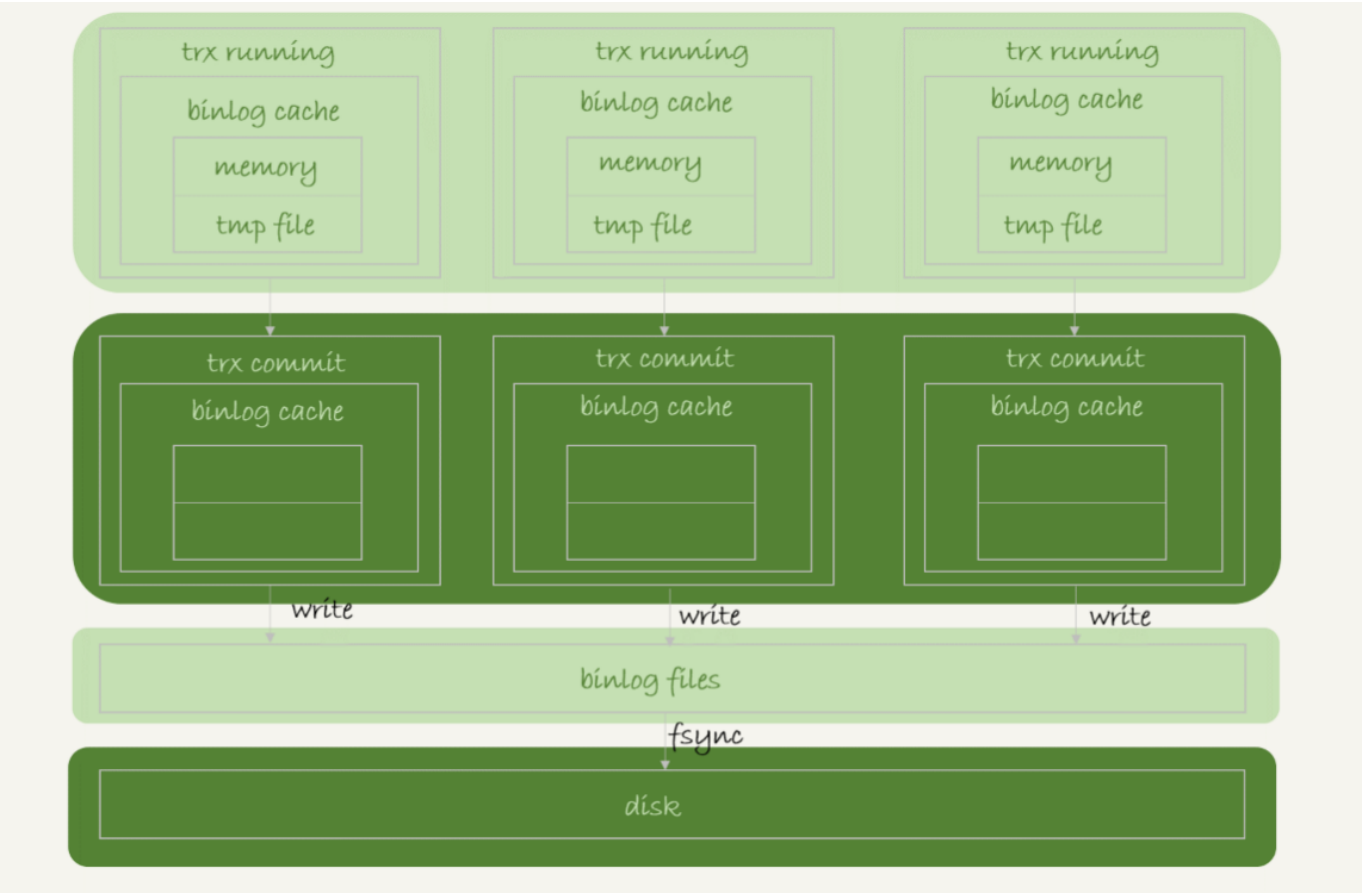
将 redo log 的写入拆成了两个步骤：prepare 和 commit，这就是"两阶段提交"。

Q：如果不用两阶段提交，会有什么问题

1. 先写redo log后写binlog
2. 先写binlog后写redo log

binlog的写入

事务执行过程中，先把日志写到 binlog cache，事务提交的时候，再把 binlog cache 写到 binlog 文件中。



相关参数

- sync_binlog

write 和 fsync 的时机，是由参数 sync_binlog 控制的：sync_binlog=0 的时候，表示每次提交事务都只 write，不 fsync；sync_binlog=1 的时候，表示每次提交事务都会执行 fsync；sync_binlog=N(N>1) 的时候，表示每次提交事务都 write，但累积 N 个事务后才 fsync。

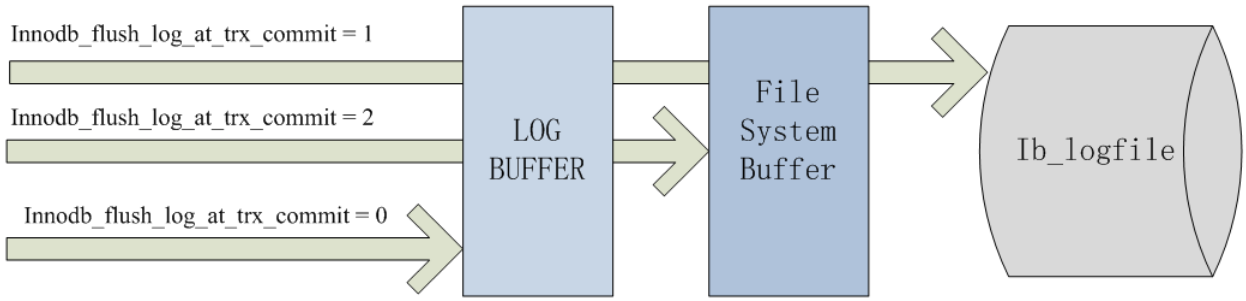
因此，在出现 IO 瓶颈的场景里，将 sync_binlog 设置成一个比较大的值，可以提升性能。在实际的业务场景中，考虑到丢失日志量的可控性，一般不建议将这个参数设成 0，比较常见的是将其设置为 100~1000 中的某个数值。但是，将 sync_binlog 设置为 N，对应的风险是：如果主机发生异常重启，会丢失最近 N 个事务的 binlog 日志。

redo log的写入

1. 日志刷盘问题

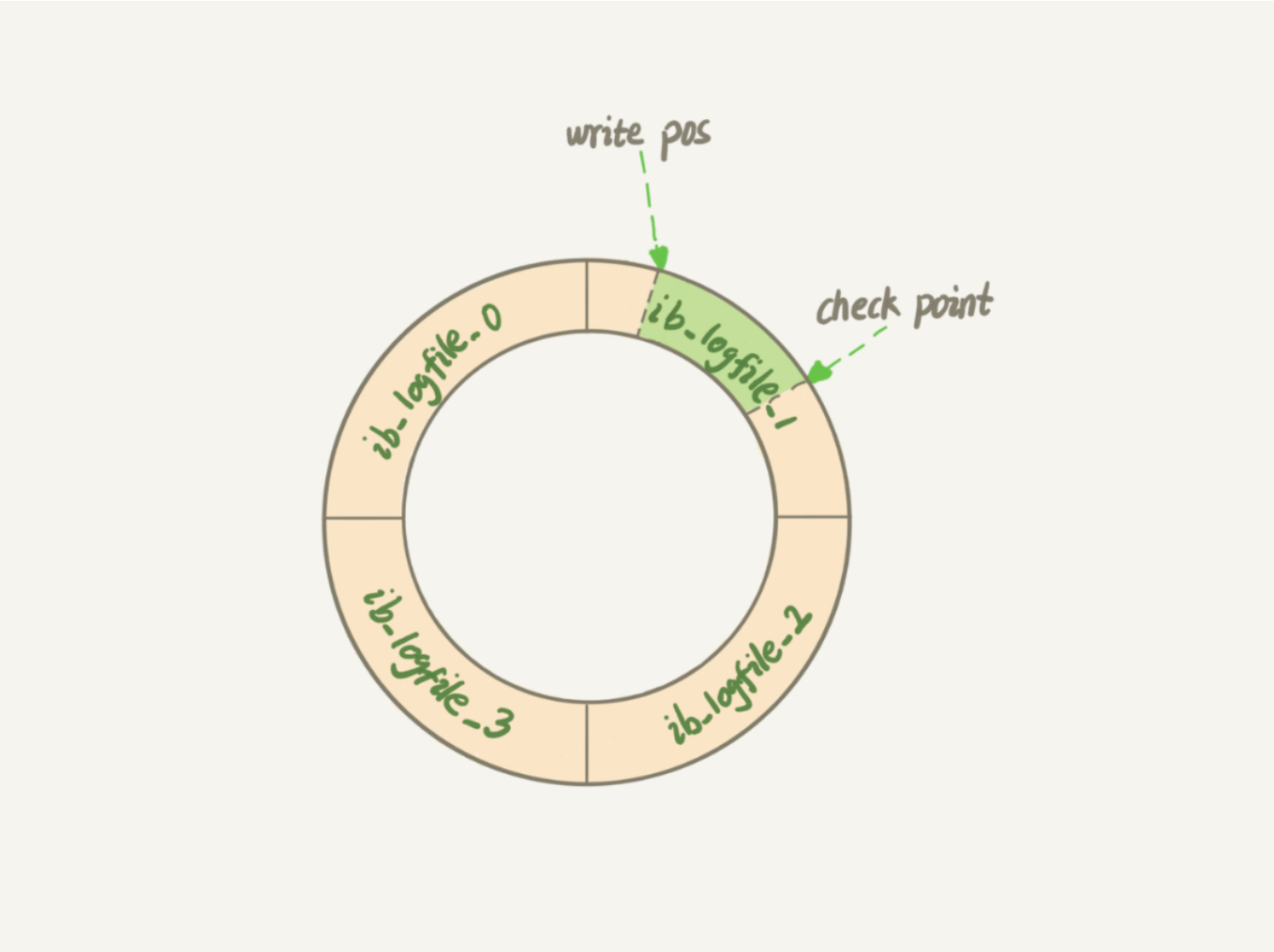
相关参数

- innodb_flush_log_at_trx_commit



- 后台线程每秒一次的轮询操作外，还有两种场景会让一个没有提交的事务的 redo log 写入到磁盘中。
- redo log buffer 占用的空间即将达到 innodb_log_buffer_size 一半的时候，后台线程会主动写盘。注意，由于这个事务并没有提交，所以这个写盘动作只是 write，而没有调用 fsync，也就是只留在了文件系统的 page cache。
- 并行的事务提交的时候，顺带将这个事务的 redo log buffer 持久化到磁盘。假设一个事务 A 执行到一半，已经写了一些 redo log 到 buffer 中，这时候有另外一个线程的事务 B 提交，如果 innodb_flush_log_at_trx_commit 设置的是 1，那么按照这个参数的逻辑，事务 B 要把 redo log buffer 里的日志全部持久化到磁盘。这时候，就会带上事务 A 在 redo log buffer 里的日志一起持久化到磁盘。

2. 数据刷盘问题



write pos 是当前记录的位置，一边写一边后移，写到第 3 号文件末尾后就回到 0 号文件开头。checkpoint 是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到数据文件。

write pos 和 checkpoint 之间的是“粉板”上还空着的部分，可以用来记录新的操作。如果 write pos 追上 checkpoint，表示“粉板”满了，这时候不能再执行新的更新，得停下来先擦掉一些记录，把 checkpoint 推进一下。

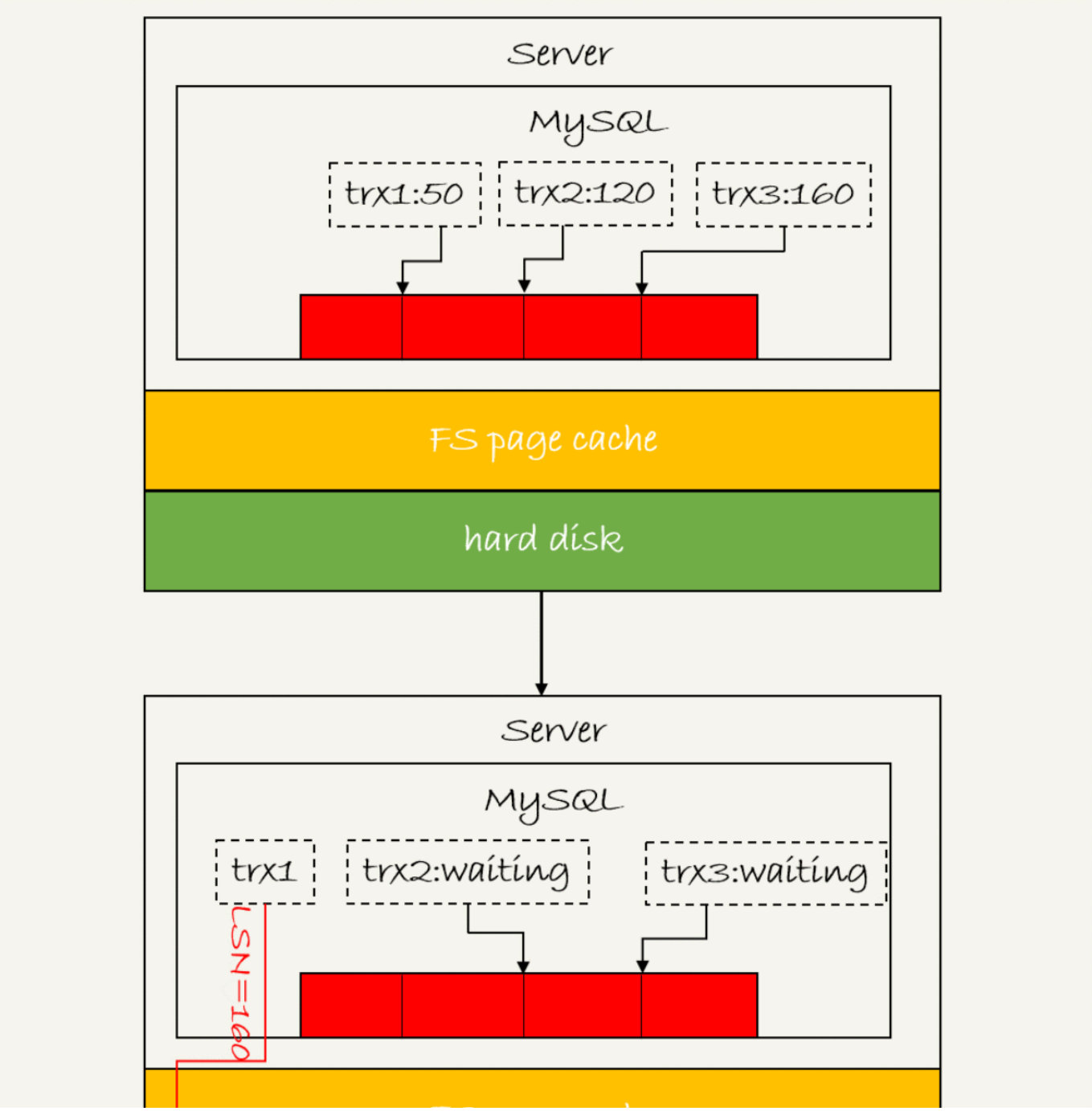
有了 redo log，InnoDB 就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力称为 crash-safe。

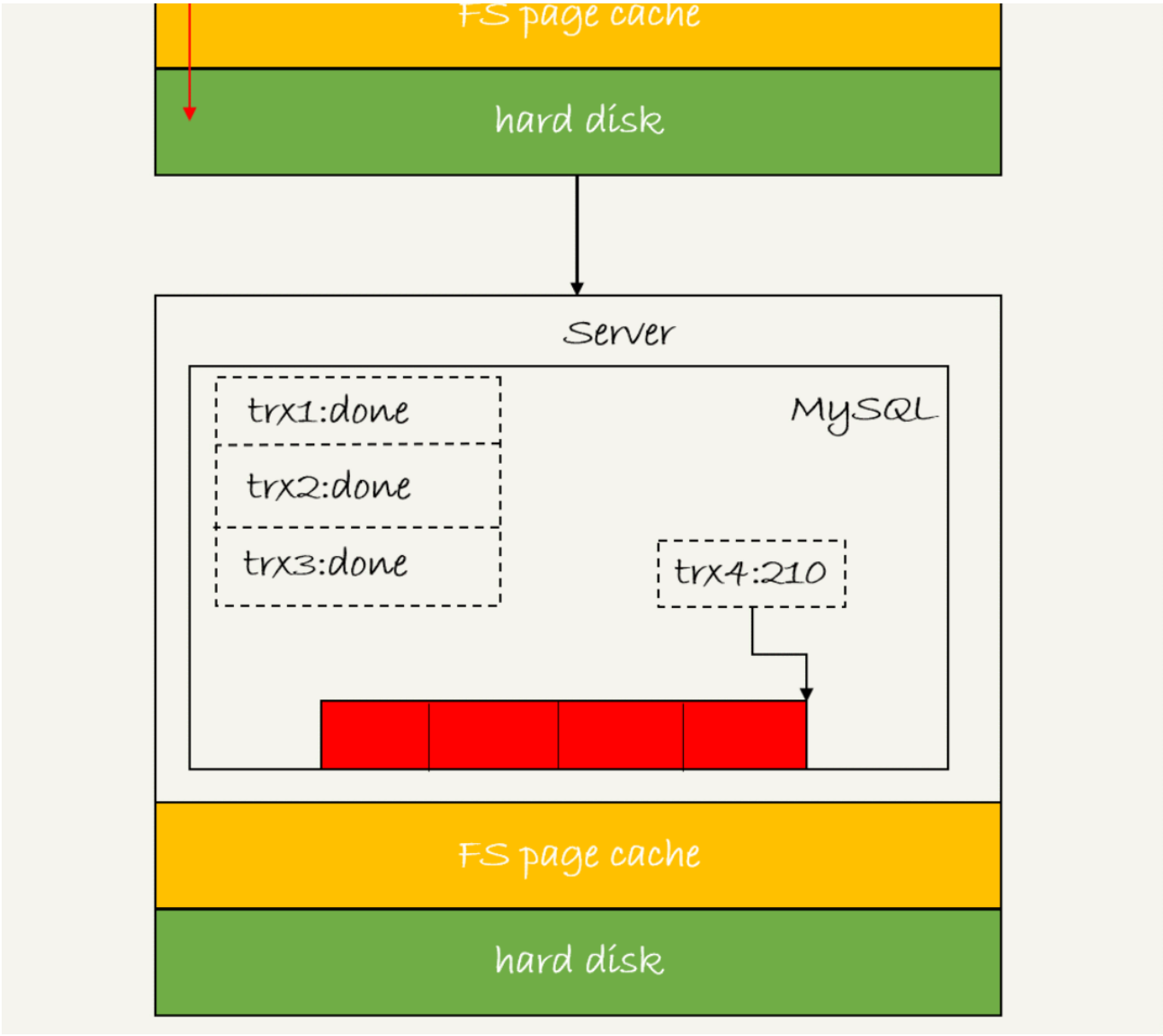
“双 1”配置，指的就是 sync_binlog 和 innodb_flush_log_at_trx_commit 都设置成 1。也就是说，一个事务完整提交前，需要等待两次刷盘，一次是 redo log（prepare 阶段），一次是 binlog。

LSN

LSN:日志逻辑序列号（log sequence number。LSN 是单调递增的，用来对应 redo log 的一个个写入点。每次写入长度为 length 的 redo log， LSN 的值就会加上 length。

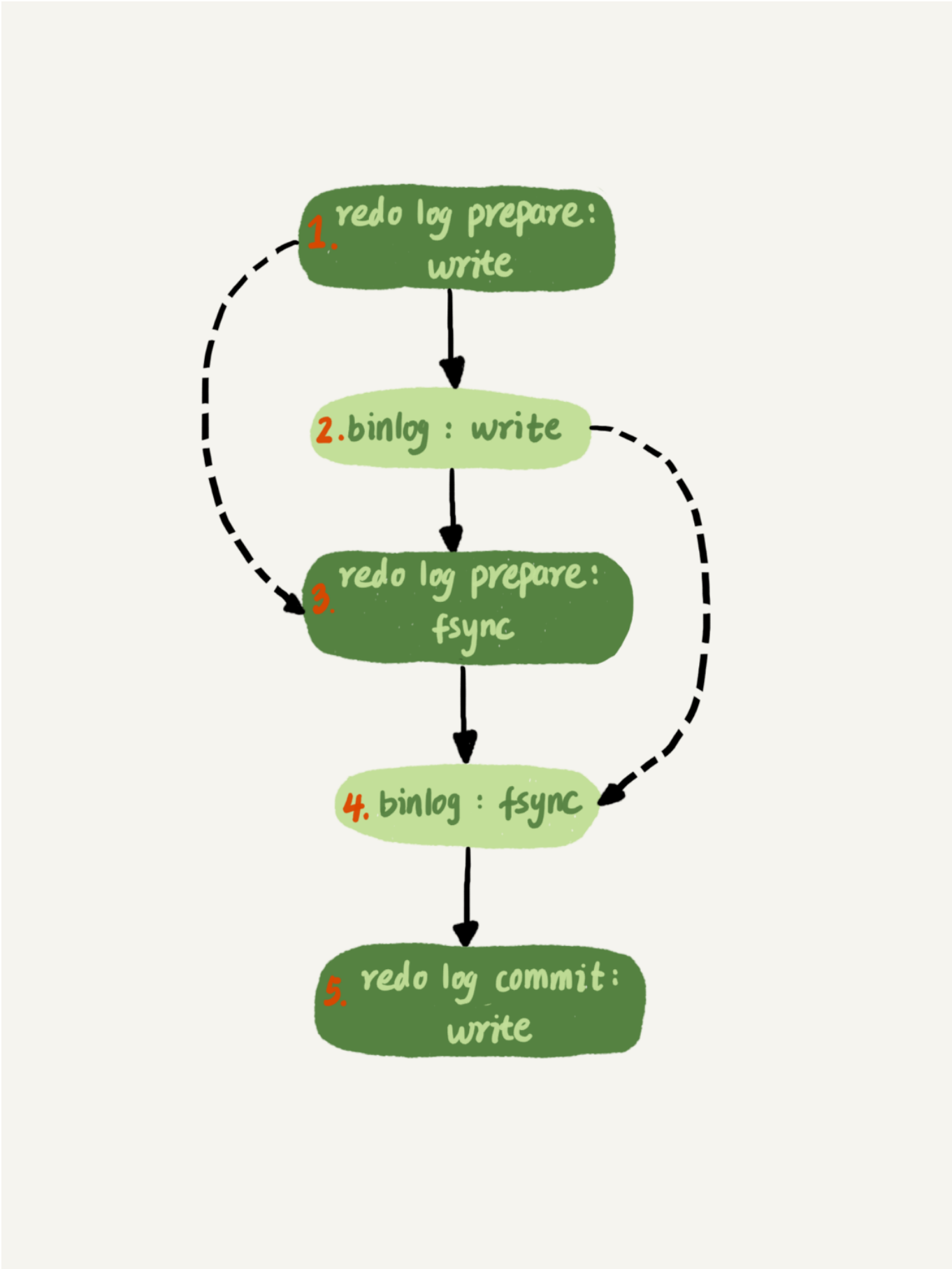
redo log 组提交





1. trx1 是第一个到达的，会被选为这组的 leader；
2. 等 trx1 要开始写盘的时候，这个组里面已经有了三个事务，这时候 LSN 也变成了 160；
3. trx1 去写盘的时候，带的就是 LSN=160，因此等 trx1 返回时，所有 LSN 小于等于 160 的 redo log，都已经被持久化到磁盘；
4. 这时候 trx2 和 trx3 就可以直接返回了。

在并发更新场景下，第一个事务写完 redo log buffer 以后，接下来这个 fsync 越晚调用，组员可能越多，节约 IOPS 的效果就越好。



相关参数

- binlog_group_commit_sync_delay： 延迟多少微秒后才调用 fsync
- binlog_group_commit_sync_no_delay_count：累积多少次以后才调用 fsync

MySQL IO性能瓶颈解决方法

1. 设置 binlog_group_commit_sync_delay 和 binlog_group_commit_sync_no_delay_count 参数，减少 binlog 的写盘次数。这个方法是基于“额外的故意等待”来实现的，因此可能会增加语句的响应时间，但没有丢失数据的风险。 【默认0】
2. 将 sync_binlog 设置为大于 1 的值（比较常见是 100~1000）。这样做的风险是，主机掉电时会丢 binlog 日志。【默认1】
3. 将 innodb_flush_log_at_trx_commit 设置为 2。这样做的风险是，主机掉电的时候会丢数据。【默认 2】

参考文献

1. https://blog.csdn.net/u010900754/article/details/106630704
2. http://mysql.taobao.org/monthly/2018/07/01/
3. https://time.geekbang.org/column/article/76161 MySQL实战45讲第23讲
4. https://time.geekbang.org/column/article/73161 MySQL实战45讲第15讲