MySQL 的 crash-safe 原理解析

MySQL 保证数据不会丢的能力主要体现在两方面：

1. 能够恢复到任何时间点的状态；（依赖binlog来实现）
2. 能够保证MySQL在任何时间段突然奔溃，重启后之前提交的记录都不会丢失；（该能力就是crash-safe，依赖redo log和undo log两个日志实现）

MySQL中是怎么执行的，简单进行总结一下：

1. 从内存中找出这条数据记录，对其进行更新；
2. 将对数据页的更改记录到redo log中；
3. 将逻辑操作记录到binlog中；
4. 对于内存中的数据和日志，都是由后台线程，当触发到落盘规则后再异步进行刷盘；

上面演示了一条更新语句的详细执行过程，接下来咱们通过解答问题，带着问题来剖析这个crash-safe的设计原理。

二、WAL机制

问题：为什么不直接更改磁盘中的数据，而要在内存中更改，然后还需要写日志，最后再落盘这么复杂?

这个问题相信很多同学都能猜出来，MySQL更改数据的时候，之所以不直接写磁盘文件中的数据，最主要就是性能问题。因为直接写磁盘文件是**随机写**，开销大性能低，没办法满足MySQL的性能要求。所以才会设计成先在内存中对数据进行更改，再异步落盘。但是内存总是不可靠，万一断电重启，还没来得及落盘的内存数据就会丢失，所以还需要加上写日志这个步骤，万一断电重启，还能通过日志中的记录进行恢复。

写日志虽然也是写磁盘，但是它是顺序写，相比随机写开销更小，能提升语句执行的性能（针对顺序写为什么比随机写更快，可以比喻为你有一个本子，按照顺序一页一页写肯定比写一个字都要找到对应页写快得多）。

这个技术就是大多数存储系统基本都会用的**WAL(Write Ahead Log)技术，也称为日志先行的技术，指的是对数据文件进行修改前，必须将修改先记录日志。保证了数据一致性和持久性，并且提升语句执行性能。**

总结：WAL其实就是修改前先记录日志的机制。

三、核心日志模块

问题：更新SQL语句执行流程中，总共需要写3个日志，这3个是不是都需要，能不能进行简化?

更新SQL执行过程中，总共涉及**MySQL日志模块其中的三个核心日志，分别是redo log（重做日志）、undo log（回滚日志）、binlog（归档日志）。**这里提前预告，crash-safe的能力主要依赖的就是这三大日志。

接下来，针对每个日志将单独介绍各自的作用，然后再来评估是否能简化掉。

1、重做日志 redo log

redo log记录的是**数据库中每个页的修改**；redo log是顺序写，相比于更新数据文件的随机写，日志的写入开销更小，能显著提升语句的执行性能，提高并发量；redo log是固定大小的，所以只能循环写，从头开始写，写到末尾就又回到开头，相当于一个环形。当日志写满了，就需要对旧的记录进行擦除，擦除前需确保擦除的记录对应在内存中的数据页已经刷到磁盘中了，并且擦除旧记录腾出新空间这段期间，是不能再接收新的更新请求，所以有可能会导致MySQL卡顿。（所以针对并发量大的系统，适当设置redo log的文件大小非常重要！！！）

redo log是为**持久化**而生的，redo log记载的是**物理变化（xxxx页做了xxx修改）**，写完内存，如果数据库挂了，那我们可以通过redo log来恢复内存还没来得及刷到磁盘的数据，将redo log加载到内存里边，那内存就能恢复到挂掉之前的数据了。但是如果数据已经落入磁盘，redo数据就无效了，会被后面的覆盖，所以刷入磁盘后的数据想恢复得使用binlog和undo log。

2、归档日志 binlog

binlog在MySQL的server层产生，不属于任何引擎，主要记录用户对数据库操作的SQL语句（**除了查询语句**）和事务id（XID）。之所以将binlog称为归档日志，是因为**binlog不会像redo log一样擦掉之前的记录循环写，而是一直记录（超过有效期才会被清理），如果超过单日志的最大值（默认1G，可以通过变量max_binlog_size 设置），则会新起一个文件继续记录。**但由于日志可能是基于事务来记录的(如InnoDB表类型)，而事务是绝对不可能也不应该跨文件记录的，如果正好binlog日志文件达到了最大值但事务还没有提交则不会切换新的文件记录，而是继续增大日志，所以 max_binlog_size 指定的值和实际的binlog日志大小不一定相等。

正是由于binlog有归档的作用，所以binlog主要用作主从同步和数据库基于时间点的还原。

那么回到刚才的问题，binlog可以简化掉吗？这里需要分场景来看：

1. 如果是主从模式下，binlog是必须的，因为从库的数据同步依赖的就是binlog；
2. 如果是单机模式，并且不考虑数据库基于时间点的还原，binlog就不是必须，因为有redo log就可以保证crash-safe能力了；但如果万一需要回滚到某个时间点的状态，这时候就无能为力，所以建议binlog还是一直开启；

根据上面对三个日志的详解，我们可以对这个问题进行解答：在主从模式下，三个日志都是必须的；在单机模式下，binlog可以视情况而定，保险起见最好开启。

总结：binlog有回滚到某个时间点的状态的能力，但是redo log没有，binlog无论MySQL用什么引擎，都会有的。

查看指定binlog文件的内容命令：

```
mysql> show binlog events in 'mysql-bin.000001';
```

binlog日志内容：

```
[python]

mysql> show binlog events in 'mysql-bin.000001';
+------------------+------+-------------+-----------+-------------+---------------------------------------------------------------+
| Log_name         | Pos  | Event_type  | Server_id | End_log_pos | Info                                                          |
+------------------+------+-------------+-----------+-------------+---------------------------------------------------------------+
| mysql-bin.000001 |    4 | Format_desc |       195 |         106 | Server ver: 5.1.73-log, Binlog ver: 4                         |
| mysql-bin.000001 |  106 | Query       |       195 |         198 | use `hadoop`; delete from user where id=3                     |
| mysql-bin.000001 |  198 | Intvar      |       195 |         226 | INSERT_ID=4                                                   |
| mysql-bin.000001 |  226 | Query       |       195 |         332 | use `hadoop`; INSERT INTO user (id,name)VALUES (NULL,1)       |
| mysql-bin.000001 |  332 | Query       |       195 |         424 | use `hadoop`; delete from user where id=3                     |
| mysql-bin.000001 |  424 | Intvar      |       195 |         452 | INSERT_ID=5                                                   |
| mysql-bin.000001 |  452 | Query       |       195 |         560 | use `hadoop`; INSERT INTO user (id,name)VALUES (NULL,222)     |
| mysql-bin.000001 |  560 | Query       |       195 |         660 | use `hadoop`; DELETE FROM `user` WHERE (`id`='1')             |
| mysql-bin.000001 |  660 | Intvar      |       195 |         688 | INSERT_ID=6                                                   |
| mysql-bin.000001 |  688 | Query       |       195 |         795 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('555')      |
| mysql-bin.000001 |  795 | Intvar      |       195 |         823 | INSERT_ID=7                                                   |
| mysql-bin.000001 |  823 | Query       |       195 |         930 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('555')      |
| mysql-bin.000001 |  930 | Intvar      |       195 |         958 | INSERT_ID=8                                                   |
| mysql-bin.000001 |  958 | Query       |       195 |        1065 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('555')      |
| mysql-bin.000001 | 1065 | Intvar      |       195 |        1093 | INSERT_ID=9                                                   |
| mysql-bin.000001 | 1093 | Query       |       195 |        1200 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('555')      |
| mysql-bin.000001 | 1200 | Query       |       195 |        1300 | use `hadoop`; DELETE FROM `user` WHERE (`id`='9')             |
| mysql-bin.000001 | 1300 | Query       |       195 |        1400 | use `hadoop`; DELETE FROM `user` WHERE (`id`='8')             |
| mysql-bin.000001 | 1400 | Query       |       195 |        1500 | use `hadoop`; DELETE FROM `user` WHERE (`id`='7')             |
| mysql-bin.000001 | 1500 | Query       |       195 |        1600 | use `hadoop`; DELETE FROM `user` WHERE (`id`='4')             |
| mysql-bin.000001 | 1600 | Query       |       195 |        1700 | use `hadoop`; DELETE FROM `user` WHERE (`id`='5')             |
| mysql-bin.000001 | 1700 | Query       |       195 |        1800 | use `hadoop`; DELETE FROM `user` WHERE (`id`='6')             |
| mysql-bin.000001 | 1800 | Intvar      |       195 |        1828 | INSERT_ID=10                                                  |
| mysql-bin.000001 | 1828 | Query       |       195 |        1935 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('555')      |
| mysql-bin.000001 | 1935 | Intvar      |       195 |        1963 | INSERT_ID=11                                                  |
| mysql-bin.000001 | 1963 | Query       |       195 |        2070 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('666')      |
| mysql-bin.000001 | 2070 | Intvar      |       195 |        2098 | INSERT_ID=12                                                  |
| mysql-bin.000001 | 2098 | Query       |       195 |        2205 | use `hadoop`; INSERT INTO `user` (`name`) VALUES ('777')      |
+------------------+------+-------------+-----------+-------------+---------------------------------------------------------------+
```

注：MySQL需要保证redo log和binlog的**数据是一致**的，如果不一致，那就乱套了。

MySQL通过**两阶段提交**来保证redo log和binlog的数据是一致的。

过程：

- 阶段1：InnoDBredo log 写盘，InnoDB 事务进入 prepare 状态
- 阶段2：binlog 写盘，InooDB 事务进入 commit 状态（实际是在redo log里面写上一个commit记录）
- 每个事务binlog的末尾，会记录一个 XID event，标志着事务是否提交成功，也就是说，恢复过程中，binlog 最后一个 XID event 之后的内容都应该被清除。

redo log**事务开始**的时候，就开始记录每次的变更信息，而binlog是在**事务提交**的时候才记录。

3、回滚日志 undo log

undo log主要有两个作用：回滚和多版本控制(MVCC)，Undo Log和Redo Log正好相反，记录的是数据**被修改前**的信息，并且只记录**逻辑**变化（相反的sql语句），基于Undo Log进行的回滚只是对数据库进行一个相反的操作，而不是直接恢复物理页。

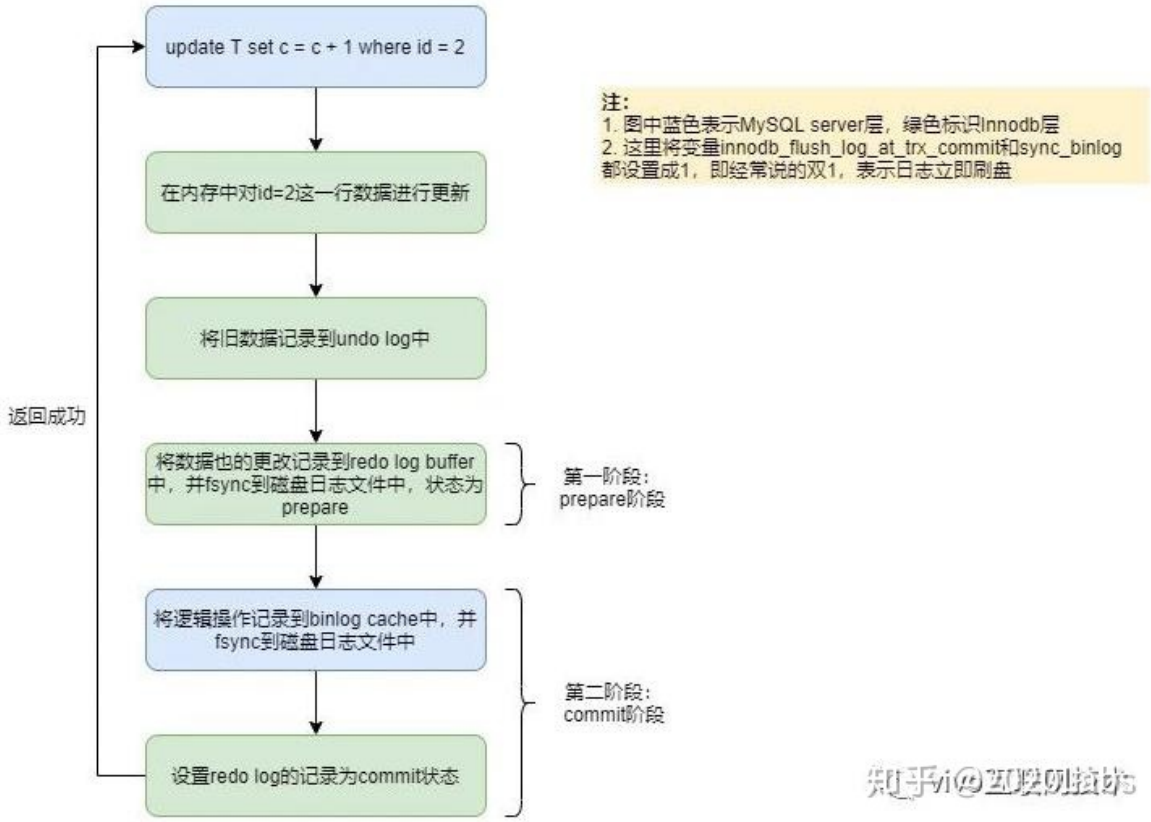
四、两阶段提交

问题：为什么redo log要分两步写，中间再穿插写binlog呢?

从上面可以看出，因为redo log影响主库的数据，binlog影响从库的数据，所以redo log和binlog必须保持一致才能保证主从数据一致，这是前提。

分布式事务，要想保持一致，就必须使用分布式事务的解决方案来处理。而将redo log分成了两步，其实就是使用了两阶段提交协议（Two-phase Commit，2PC）。

下面对更新语句的执行流程进行简化，看一下MySQL的两阶段提交是如何实现的：



从图中可看出，事务的提交过程有两个阶段，就是将redo log的写入拆成了两个步骤：prepare和commit，中间再穿插写入binlog。

过程：

- 阶段1：InnoDB`redo log` 写盘，InnoDB 事务进入 `prepare` 状态
- 阶段2：`binlog` 写盘，InooDB 事务进入 `commit` 状态（实际是在redo log里面写上一个commit记录）
- 每个事务`binlog`的末尾，会记录一个 `XID event`，标志着事务是否提交成功，也就是说，恢复过程中，`binlog` 最后一个 XID event 之后的内容都应该被清除。

`redo log`**事务开始**的时候，就开始记录每次的变更信息，而`binlog`是在**事务提交**的时候才记录。

如果这时候你很疑惑，为什么一定要用两阶段提交呢，如果不用两阶段提交会出现什么情况，比如先写redo log，再写binlog或者先写binlog，再写redo log不行吗？下面我们用反证法来进行论证。

我们继续用update T set c=c+1 where id=2这个例子，假设id=2这一条数据的c初始值为0。那么在redo log写完，binlog还没有写完的时候，MySQL进程异常重启。由于redo log已经写完了，系统重启后会通过redo log将数据恢复回来，所以恢复后这一行c的值是1。但是由于binlog没写完就crash了，这时候binlog里面就没有记录这个语句。因此，不管是现在的从库还是之后通过这份binlog还原临时库都没有这一次更新，c的值还是0，与原库的值不同。

同理，如果先写binlog，再写redo log，中途系统crash了，也会导致主从不一致，这里就不再详述。

所以将**redo log分成两步写**，即两阶段提交，才能保证redo log和binlog内容一致，从而保证主从数据一致。

注：两阶段提交的核心是redo log分成两步写。

如何保证顺序性？

两阶段提交虽然能够保证单事务两个日志的内容一致，但在多事务的情况下，却不能保证两者的提交顺序一致，比如下面这个例子，假设现在有3个事务同时提交：

```
T1 (--prepare--binlog--------------------commit)
T2 (-----prepare-----binlog----commit)
T3 (--------prepare-------binlog------commit)
```

解析：

redo log prepare的顺序：T1 --》T2 --》T3

binlog的写入顺序：T1 --》 T2 --》T3

redo log commit的顺序：T2 --》 T3 --》T1

结论：由于binlog写入的顺序和redo log提交结束的顺序不一致，导致binlog和redo log所记录的事务提交结束的顺序不一样，最终导致的结果就是主从数据不一致。

因此，**在两阶段提交的流程基础上，还需要加一个锁来保证提交的原子性**，从而保证多事务的情况下，两个日志的提交顺序一致。所以在早期的MySQL版本中，通过使用prepare_commit_mutex锁来保证事务提交的顺序，在一个事务获取到锁时才能进入prepare，一直到commit结束才能释放锁，下个事务才可以继续进行prepare操作。通过加锁虽然完美地解决了顺序一致性的问题。

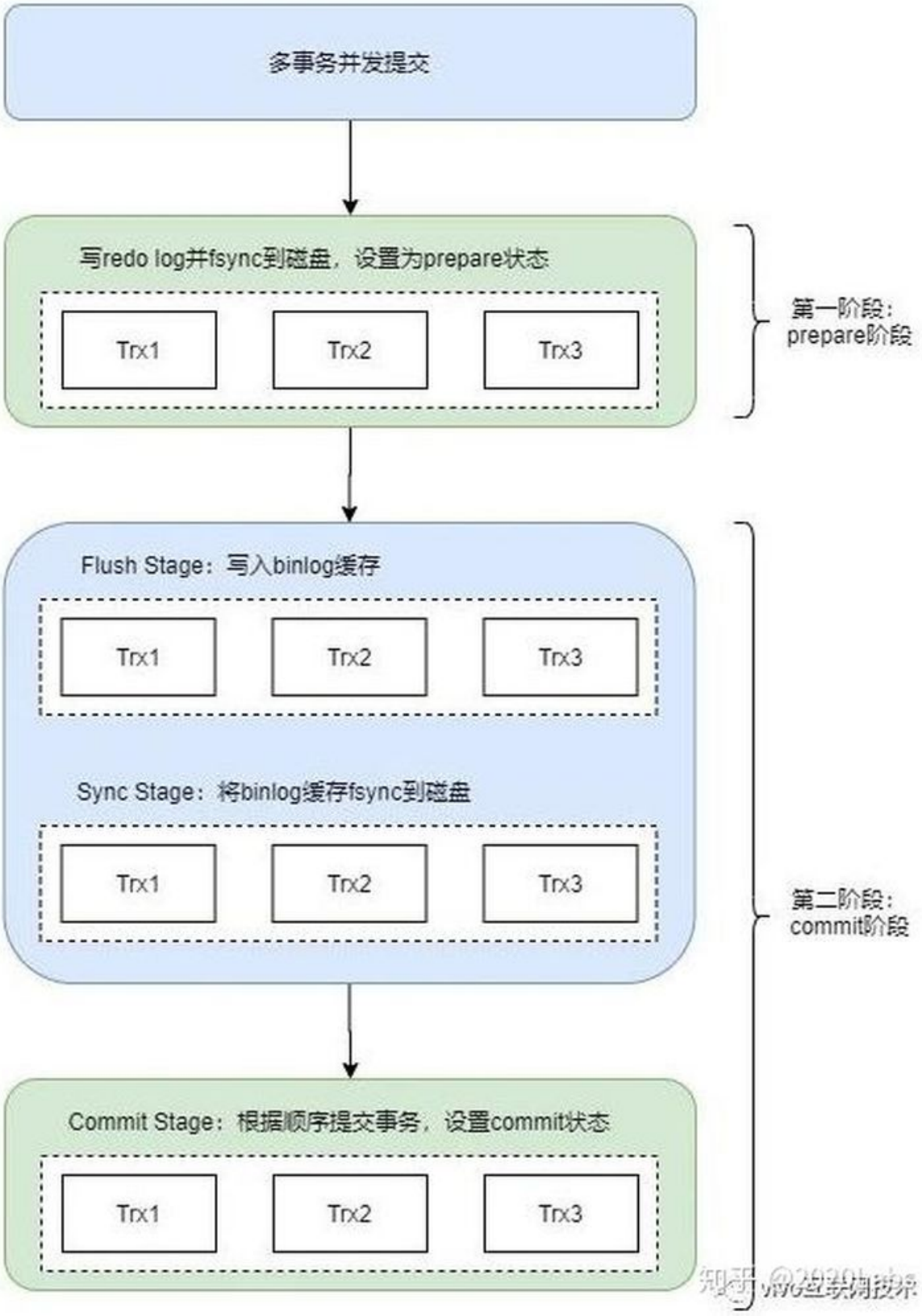
两阶段缺点：

1. 并发量较大的时候，就会导致对锁的争用。
2. 每个事务提交都会进行两次fsync（写磁盘），一次是redo log落盘，另一次是binlog落盘。大家都知道，写磁盘是昂贵的操作，对于普通磁盘，**每秒的QPS大概也就是几百**，于是采用组提交方式来改进性能问题。

五、组提交

问题：针对通过在两阶段提交中加锁控制事务提交顺序这种实现方式遇到的性能瓶颈问题，有没有更好的解决方案呢?

答案自然是有的，在MySQL 5.6 就引入了binlog组提交，即BLGC（Binary Log Group Commit）。binlog组提交的基本思想是，引入队列机制保证InnoDB commit顺序与binlog落盘顺序一致，并将事务分组，组内的binlog刷盘动作交给一个事务进行，实现组提交目的。具体如图：



第一阶段（prepare阶段）：

持有prepare_commit_mutex锁，并且write/fsync redo log到磁盘，设置为prepared状态，完成后就释放prepare_commit_mutex，binlog不作任何操作。

第二个阶段（commit阶段）：

这里拆分成了三步，**每一步的任务分配给一个专门的线程处理**：

1. **Flush Stage（写入binlog缓存）**
   ① 持有Lock_log mutex [leader持有，follower等待]

② 获取队列中的一组binlog(队列中的所有事务)
③ 写入binlog缓存

2. **Sync Stage（将binlog落盘）**
   ①释放Lock_log mutex，持有Lock_sync mutex[leader持有，follower等待]
   ②将一组binlog落盘（fsync动作，最耗时，假设sync_binlog为1）。
3. **Commit Stage（InnoDB commit，清楚undo信息）**
   ①释放Lock_sync mutex，持有Lock_commit mutex[leader持有，follower等待]
   ② 遍历队列中的事务，逐一进行InnoDB commit
   ③ 释放Lock_commit mutex

每个Stage都有自己的队列，队列中的第一个事务称为leader，其他事务称为follower，leader控制着follower的行为。**每个队列各自有互斥锁（mutex）保护，队列之间是顺序的。**只有flush完成后，才能进入到sync阶段的队列中；sync完成后，才能进入到commit阶段的队列中。但是这三个阶段的作业是可以同时并发执行的，即当一组事务在进行commit阶段时，其他新事务可以进行flush阶段，实现了真正意义上的组提交，大幅度降低磁盘的IOPS消耗。

针对组提交为什么比两阶段提交加锁性能更好，简单做个总结：组提交虽然在每个队列中仍然保留了prepare_commit_mutex锁，但是锁的粒度变小了，变成了原来两阶段提交的1/4，所以锁的争用性也会大大降低；另外，组提交是**批量刷盘**，相比之前的单条记录都要刷盘，能大幅度降低磁盘的IO消耗。

组提交的缺点：因为是批量刷盘，假如要等待100个才能提交，如果等到99个机器出问题，那么之前的等待又白费了。

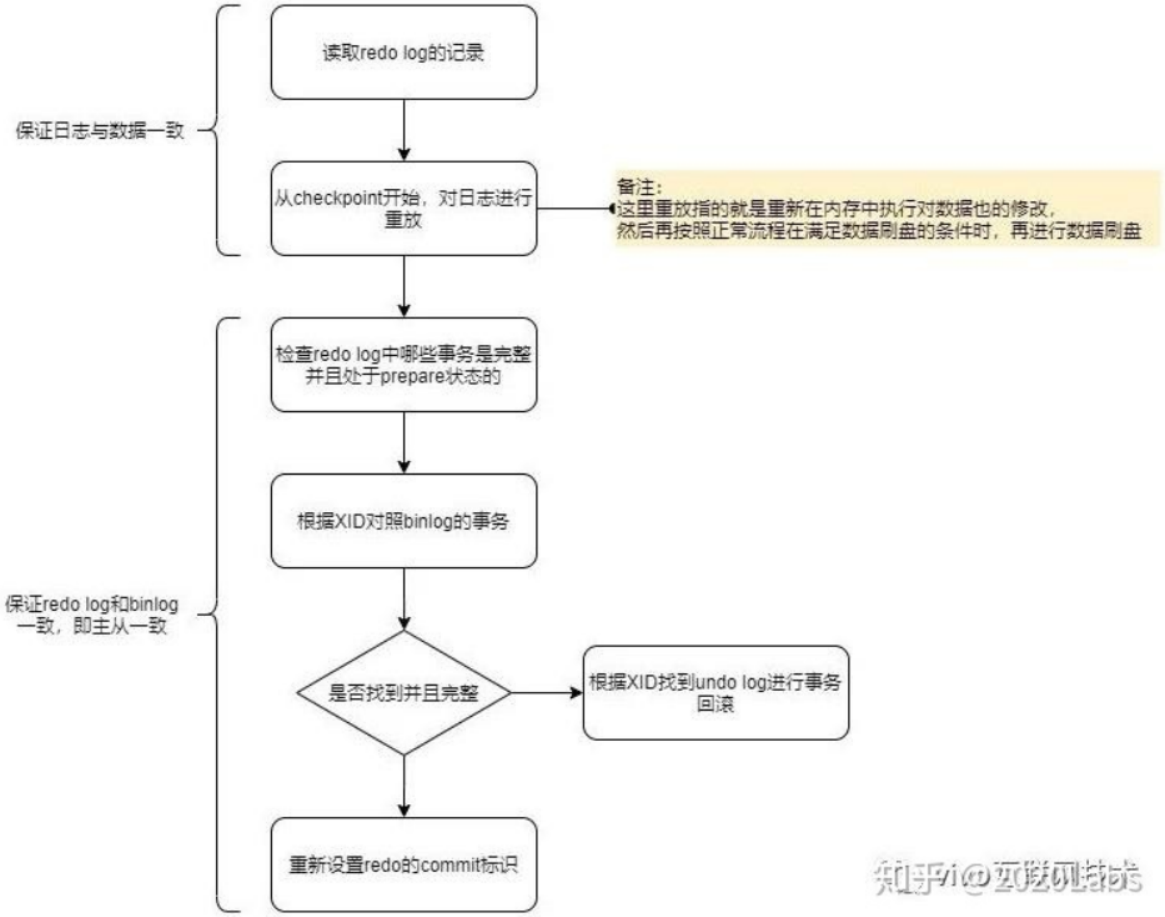
组提交相对于两阶段的改进：

1. 每个流程都有一个线程控制。
2. 每个流程里面都有队列，队列中的第一个事务称为leader，其他事务称为follower，通过队列实现等到事务达到阈值一起提交，减少刷盘次数。
3. 为了保证流程顺序性，每个流程使用互斥锁。

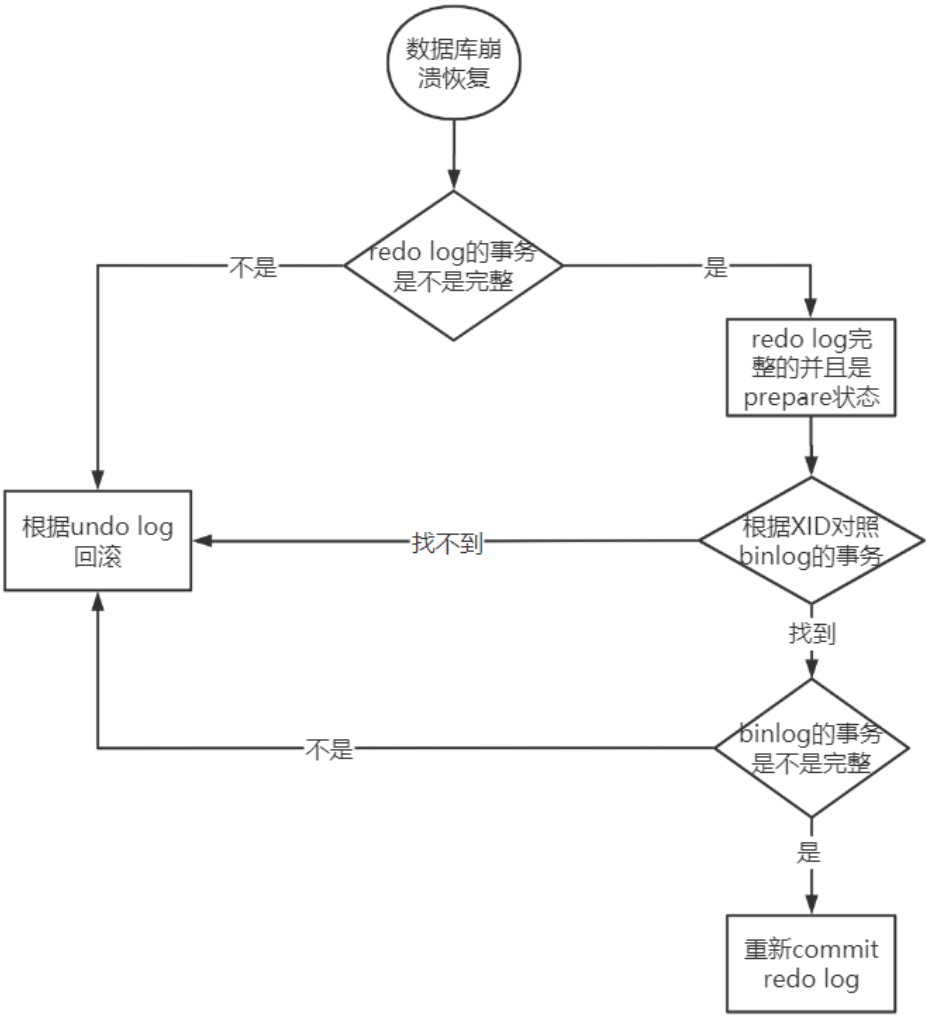
六、数据恢复流程

问题：假设事务提交过程中，MySQL进程突然奔溃，重启后是怎么保证数据不丢失的?

下图就是MySQL重启后，提供服务前会先做的事 -- 恢复数据的流程：



对上图进行简单描述就是，整个数据恢复的流程：**奔溃重启后会检查redo log中是完整并且处于prepare状态的事务，然后根据XID（事务ID），从binlog中找到对应的事务，如果找不到，则回滚；找到并且事务完整则重新commit redo log，完成事务的提交。**（整个数据恢复的流程，可以解决时刻B、C、D的问题，时刻A是还没刷盘的情况）



下面我们根据事务提交流程，在不同的阶段时刻，看看MySQL突然奔溃后，按照上述流程是如何恢复数据的。

1. **时刻A**（刚在内存中更改完数据页，还没有开始写redo log的时候奔溃）：
   因为内存中的脏页还没刷盘，也没有写redo log和binlog，即这个事务还没有开始提交，所以奔溃恢复跟该事务没有关系；
2. **时刻B**（正在写redo log或者已经写完redo log并且落盘后，处于prepare状态，还没有开始写binlog的时候奔溃）：
   恢复后会判断redo log的事务是不是完整的，如果不是则根据undo log回滚；如果是完整的并且是prepare状态，则进一步判断对应的事务binlog是不是完整的，如果不完整则一样根据undo log进行回滚；
3. **时刻C**（正在写binlog或者已经写完binlog并且落盘了，还没有开始commit redo log的时候奔溃）：
   恢复后会跟时刻B一样，先检查redo log中是完整并且处于prepare状态的事务，然后判断对应的事务binlog是不是完整的，如果不完整则一样根据undo log回滚，完整则重新commit redo log；
4. **时刻D**（正在commit redo log或者事务已经提交完的时候，还没有反馈成功给客户端的时候奔溃）：
   恢复后跟时刻C基本一样，都会对照redo log和binlog的事务完整性，来确认是回滚还是重新提交。

redolog、binlog、undolog如何保证断电数据一致?

七、总结

至此对MySQL 的crash-safe原理细节就基本讲完了，简单回顾一下：

1. 首先简单介绍了WAL日志先行技术，包括它的定义、流程和作用。WAL是大部分数据库系统实现一致性和持久性的通用设计模式。；
2. 接着对MySQL的日志模块，redo log、undo log、binlog、两阶段提交和组提交都进行了详细介绍；
3. 最后讲解了数据恢复流程，并从不同时刻加以验证。