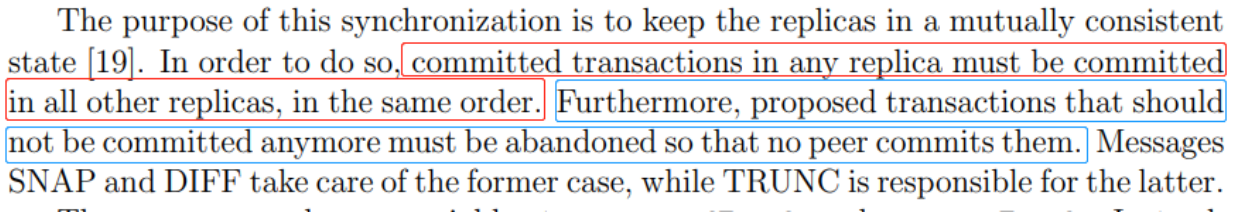
探求动机

zab论文中指出，



The purpose of this synchronization is to keep the replicas in a mutually consistent state [19]. In order to do so, committed transactions in any replica must be committed in all other replicas, in the same order. Furthermore, proposed transactions that should not be committed anymore must be abandoned so that no peer commits them. Messages SNAP and DIFF take care of the former case, while TRUNC is responsible for the latter.

为了维持一致性，必须保证:

1. 已经提交的事务最终要在所有服务器上提交
2. 已经被丢弃的事务不能被之后的服务器提交

要保证zookeeper做到第一点，我们需要证明: committed事务一定完整存在于之后的所有Leader上。，因为如果之后的某个Leader不存在该事务，就相当于该事务丢失了，破坏了一致性的原则。

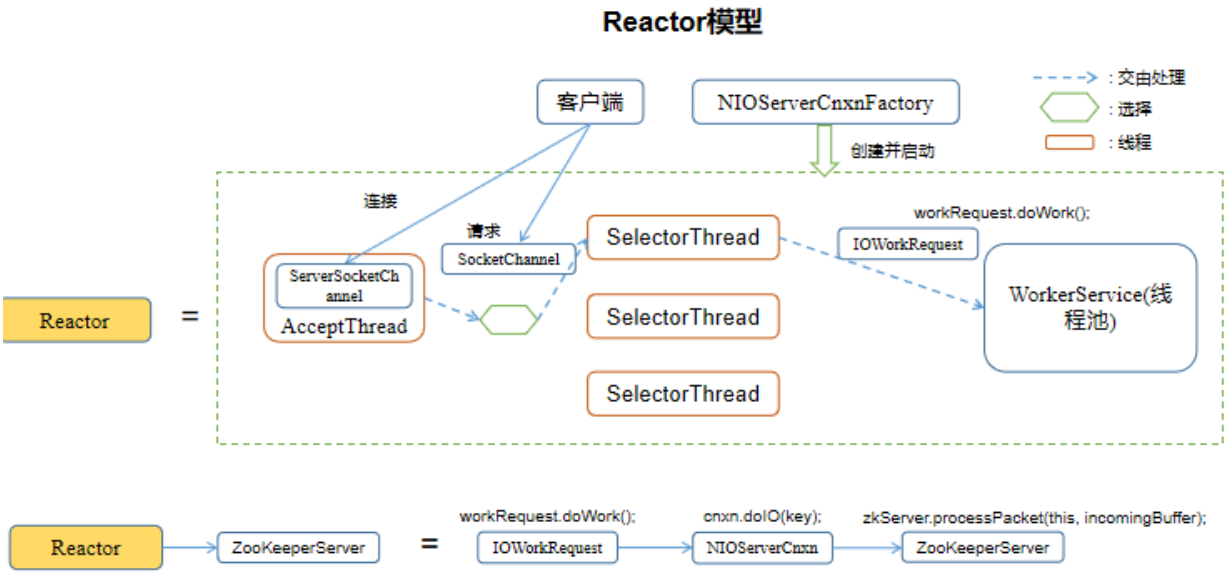
同时，我们还要确定句子中**committed的含义。**

另外我曾疑惑，为什么Leader在收到过半ACK后就敢在本地commmit？如果这时发生停电，集群全部宕机，还能保证之后的Leader将事务提交吗？

此处我先将答案写下，之后会以图文的形式一探究竟。读者需根据图中的流程图进行调试，自行思考。

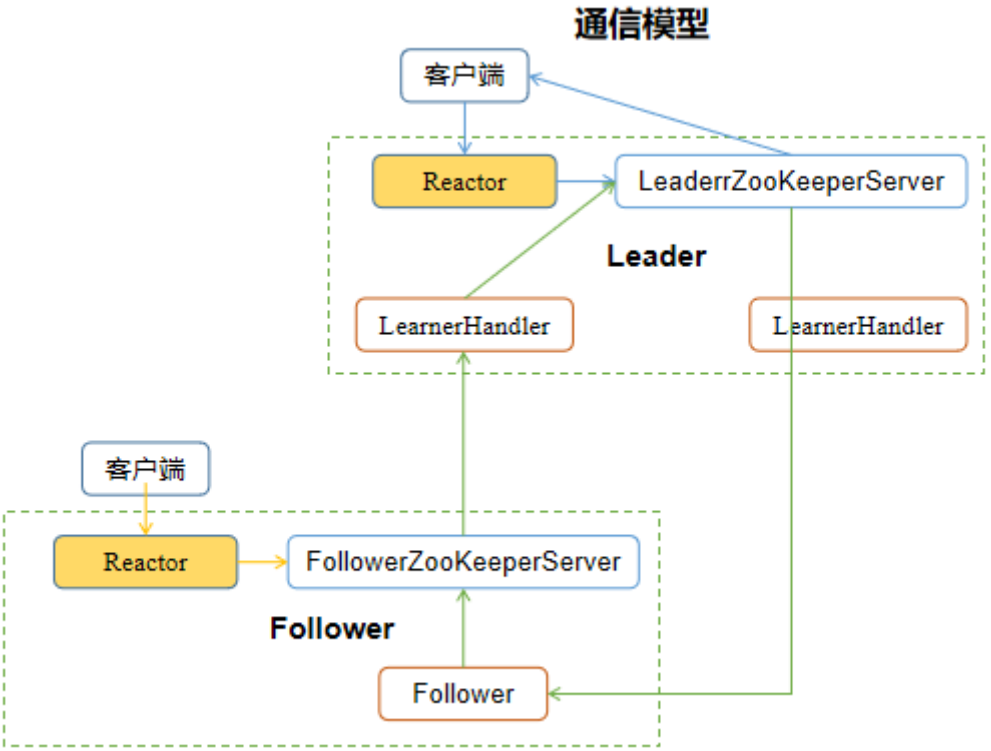
> 其实，Follower完成后，才回复ACK。那么当Leader收到过半ACK时，说明过半Follower已经将事务备份完毕。这个时刻的Leader已经能安全提交了，因为即使停电，由于事务持久化在过半服务器上，再结合**崩溃恢复的机制**，该事务终会被之后的Leader提交。

服务端Reactor模型



仅供参考，便于调试理解

通信模型



仅供参考，便于调试理解

崩溃恢复机制理解

新任leader同步的机制，关键在于利用NEWLADER的提交使之前的都提交。

与raft相似

zab协议实现的zookeeper跟raft协议基本一样：

- raft中，新Leader会写一条no-op来确保旧事务commit。
- zookeeper中，新leader会使用新的epoch写一条叫NewLeader的日志来保证旧leader可能被commit的日志最终得以commit。

步骤

崩溃恢复的关键在于Leader的同步步骤:

1. 新任leader视情况要求Follower执行TRUNC或DIFF。若有冲突则需要执行TRUNC，让Follower将冲突部分丢弃。
2. 冲突部分丢弃后，新任leader让follower同步自己的日志(依次发送每条缺少的日志和commit请求)，并最后还要求写一条NEWLEADER日志。
3. 当NEWLEADER日志收到过半ack时，代表该日志连带之前的日志都已经commit，给follower发送UPTODATE。
4. Follower收到UPTODATE后才结束同步，开始"对外服务""。

> 在步骤2.中，每条日志后紧跟一条commit请求的做法是安全的，因为只有完成步骤3.，Follower收到了UPTODATE后，才结束同步，对外服务。同步期间是不处理客户端请求的。所以即使在执行commit的期间，服务器宕机了，在重启后，不论Leader在宕机期间有没更换，Leader也会先要求服务器丢弃冲突部分，再进行同步。

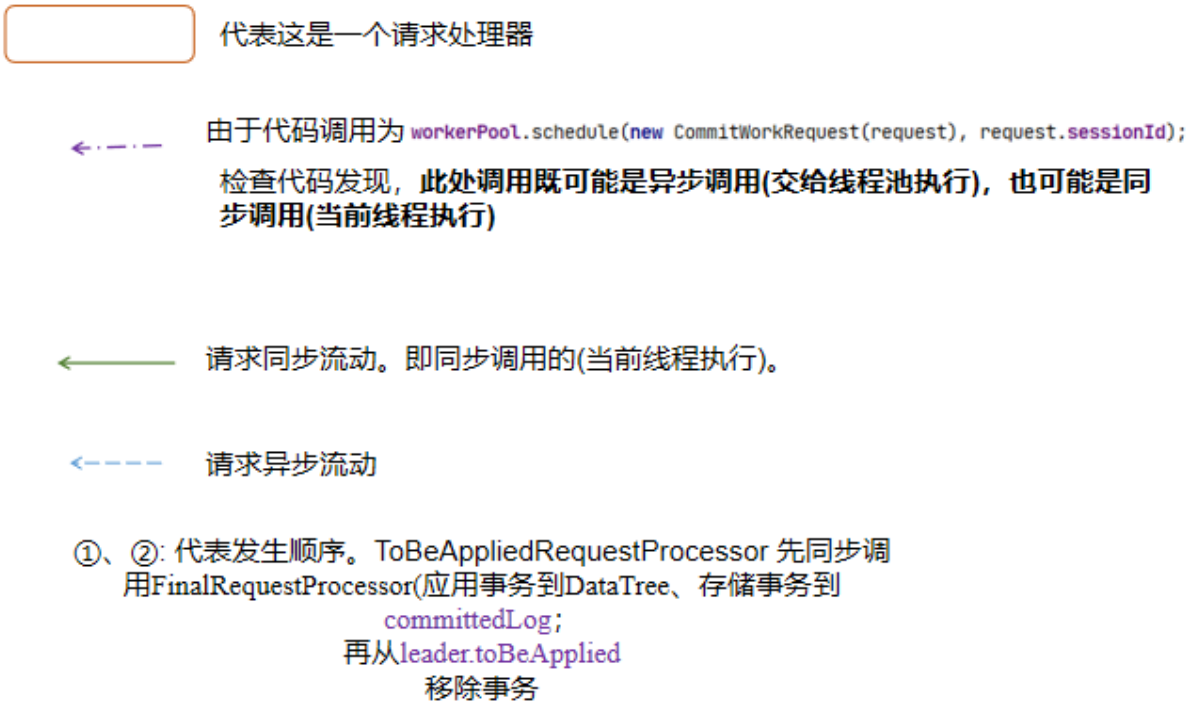
写操作流程

介绍

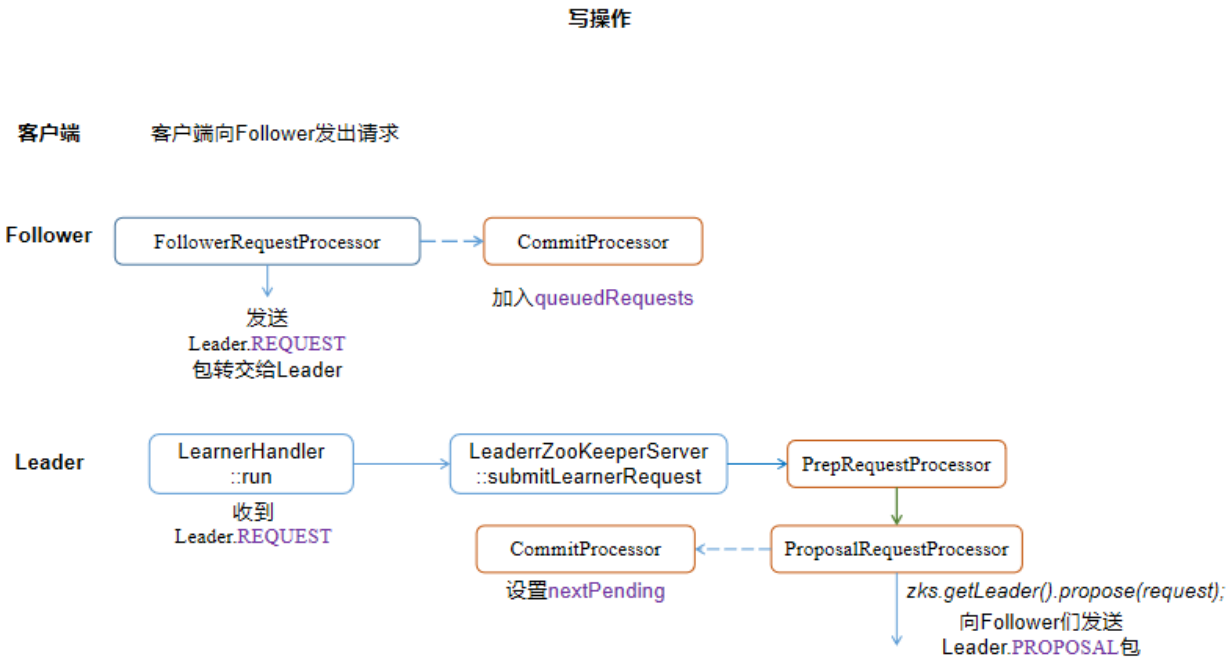
本章假设读者已经理解请求链和请求处理器。 本章会剖析客户端将写请求(比如 `set / 3`)提交给Follower后，请求在Follower与Leader的请求链上流动的情况。 本章试图通过剖析请求的流动，证明两点：

1. 已经commit的消息一定会出现在之后的Leader中
2. 已经丢弃的消息一定不出现在之后的Leader中

首先展示一下会用到的符号的解释:

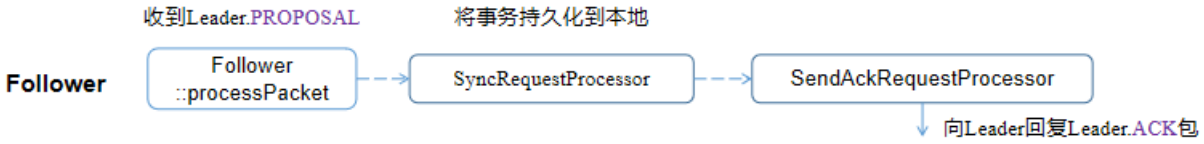


1. Follower转交Leader发送写请求



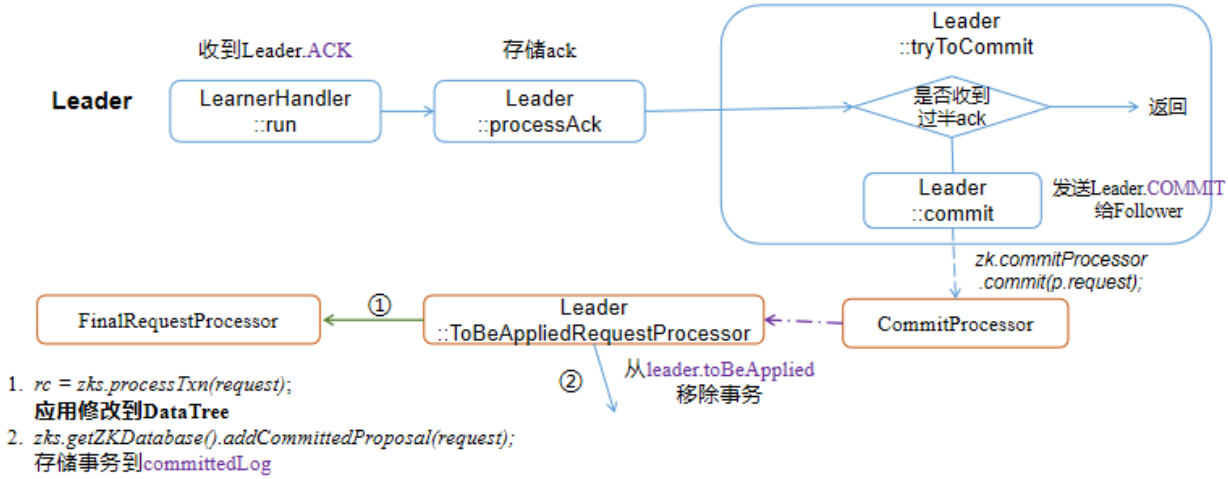
1. Follower收到写请求后，转交给Leader
2. Leader收到后，将请求分发给所有Follower

2. Follower响应



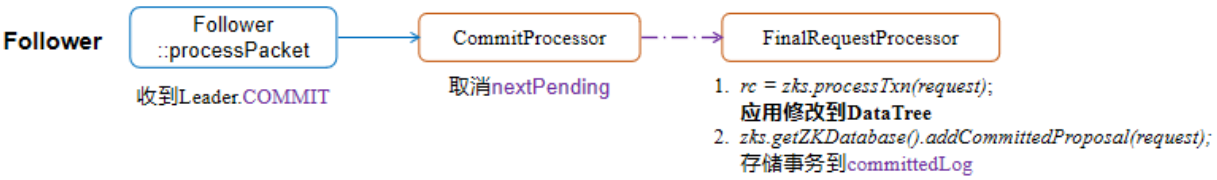
1. Follower收到写请求，先将事务**完成**持久化到本地(假设持久化顺利)。SyncRequestProcessor采用按批写入的方式，保证事务持久化完成后才交给下个请求处理器(SyncRequestProcessor的机制可参考Zookeeper能保证持久化不丢失吗？)。
2. Follower回复ACK给Leader

3. Leader提交



1. Leader先发送Commit给Follower，在DataTree应用写操作

4. Follower提交



1. Follower收到Commit后，在本地DataTree应用写操作。

总结

由上可得事件的顺序是：

1. Follower将本地事务持久化完成后，才回复ACK
2. Leader收到过半ACK后，才发送Commit给Follower

由1. 2.结合可得，**Leader收到过半ACK时，说明过半Follower已经将事务备份完毕。**

然后不管发送Commit的操作时集群是否会发生崩溃，在集群恢复后，**这条事务都会存储在过半的服务器上。**

结合崩溃恢复的机制，可以确保新Leader包含这条事务。

从这个角度来看，当一条消息被过半服务器持久化完毕时，它就可以被视为"committed"了，因为他一定会存在于之后的Leader上。