咕泡学院 VIP 课：深入分析 Zookeeper 的实现原理

课程目标

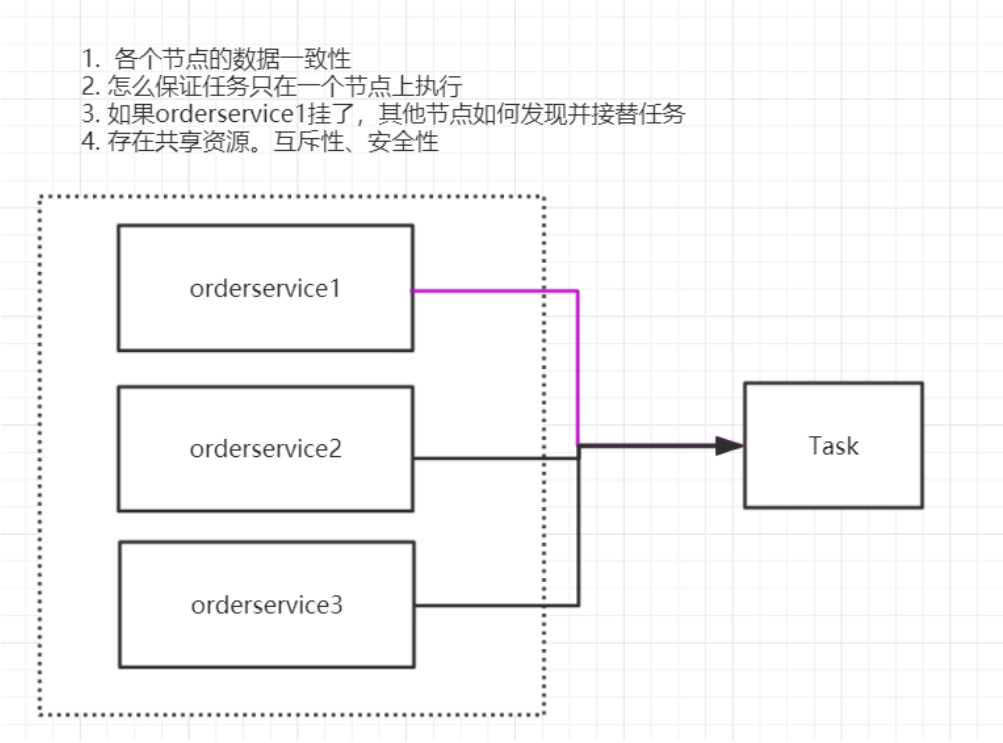
1. 了解 zookeeper 及 zookeeper 的设计猜想
2. zookeeper 集群角色
3. 深入分析 ZAB 协议
4. 从源码层面分析leader选举的实现过程
5. 关于 zookeeper 的数据存储

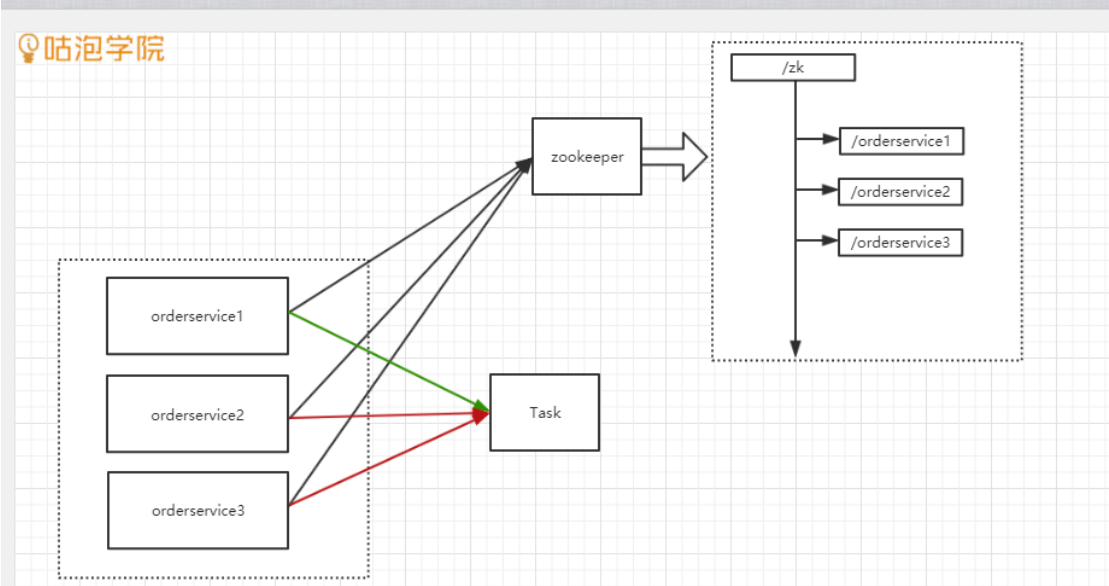
回顾内容

1. zookeeper 集群安装 （myid / zoo.cfg）

2. zookeeper 的数据模型（znode）
3. 节点的特性（持久化、临时节点、有序节点、同级节点必须唯一、临时节点不能存在子节点）
4. 节点的 stat 信息
5. 简单了解了 Watcher 机制。
6. zookeeper 的应用场景

zookeeper 的由来





Zookeeper 的前世今生

从上面的案例可以看出，分布式系统的很多难题，都是由于缺少协调机制造成的。在分布式协调这块做得比较好的，有 Google 的 Chubby 以及 Apache 的 Zookeeper。

Google Chubby 是一个分布式锁服务，通过 Google Chubby 来解决分布式协作、Master 选举等与分布式锁服务相关的问题。

Zookeeper 也是类似，因为当时在雅虎内部的很多系统都需要依赖一个系统来进行分布式协调，但是谷歌的 Chubby 是不开源的，所以后来雅虎基于 Chubby 的思想开发了 zookeeper，并捐赠给了 Apache。

在上面这个架构下 zookeeper 以后，可以用来解决 task 执行问题，各个服务先去 zookeeper 上去注册节点，然后获得权限以后再来访问 task

zookeeper 的设计猜想

zookeeper 主要是解决分布式环境下的服务协调问题而产生的，如果我们要去实现一个 zookeeper 这样的中间件，我们需要做什么？

1. 防止单点故障

   如果要防止 zookeeper 这个中间件的单点故障，那就势必要做集群。而且这个集群如果要满足高性能要求的话，还得是一个高性能高可用的集群。高性能意味着这个集群能够分担客户端的请求流量，高可用意味着集群中的某一个节点宕机以后，不影响整个集群的数据和继续提供服务的可能性。

   结论： 所以这个中间件需要考虑到集群,而且这个集群还需要分摊客户端的请求流量

2. 接着上面那个结论再来思考，如果要满足这样的一个高性能集群，我们最直观的想法应该是，每个节点都能接收到请求，并且每个节点的数据都必须要保持一致。要实现各个节点的数据一致性，就势必要一个 leader 节点负责协调和数据同步操作。这个我想大家都知道，如果在这样一个集群中没有 leader 节点，每个节点都可以接

收所有请求，那么这个集群的数据同步的复杂度是非常大。

结论：所以这个集群中涉及到数据同步以及会存在 leader 节点

3. 继续思考，如何在这些节点中选举出 leader 节点，以及 leader 挂了以后，如何恢复呢？

   结论：所以 zookeeper 用了基于 paxos 理论所衍生出来的 ZAB 协议

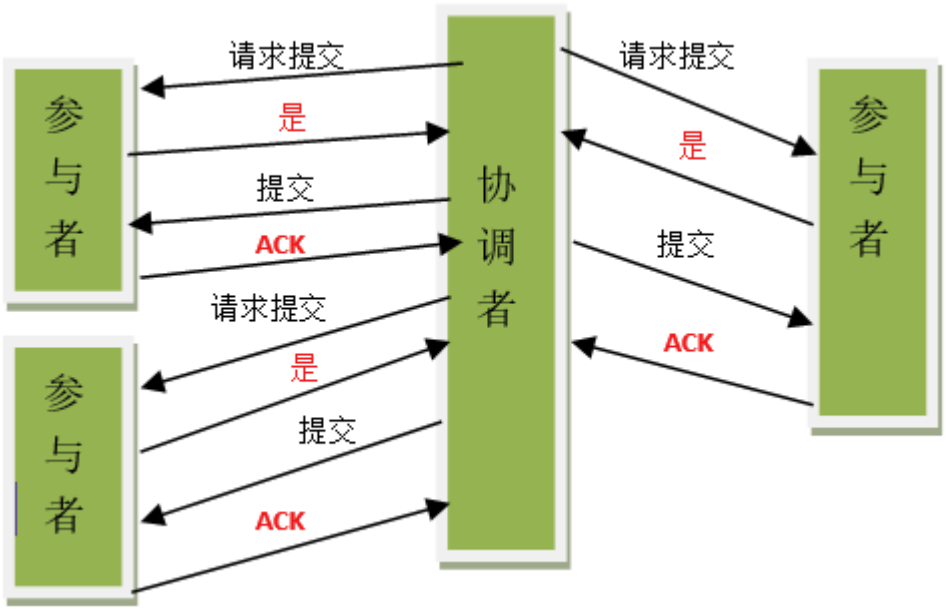
4. leader 节点如何和其他节点保证数据一致性，并且要求是强一致的。在分布式系统中，每一个机器节点虽然都能够明确知道自己进行的事务操作过程是成功和失败，但是却无法直接获取其他分布式节点的操作结果。所以当一个事务操作涉及到跨节点的时候，就需要用到分布式事务，分布式事务的数据一致性协议有 2PC 协议和 3PC 协议。

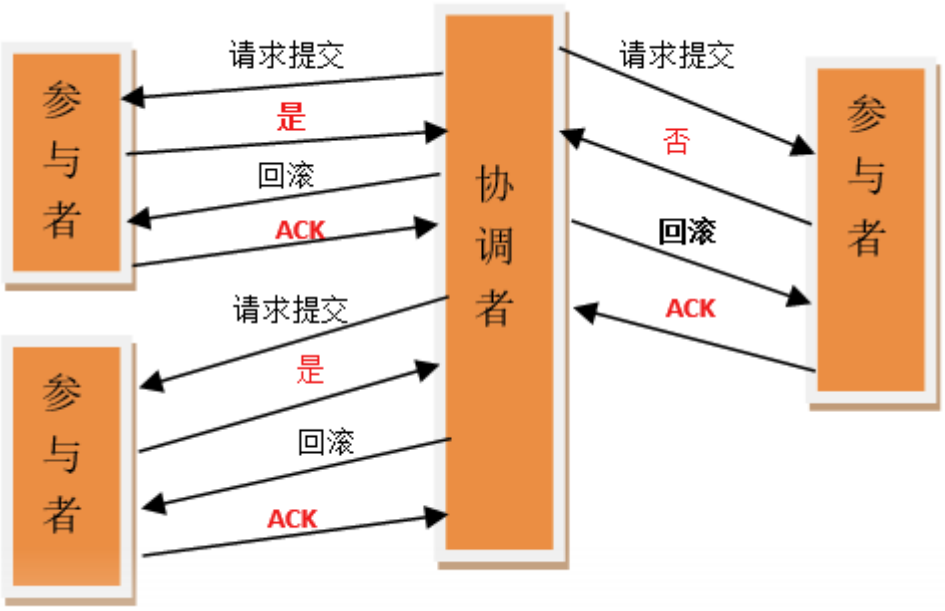
基于这些猜想，我们基本上知道 zookeeper 为什么要用到 zab 理论来做选举、为什么要做集群、为什么要用到分布式事务来实现数据一致性了。接下来我们逐步去剖析 zookeeper 里面的这些内容

关于 2PC 提交

（Two Phase Commitment Protocol）当一个事务操作需

要跨越多个分布式节点的时候，为了保持事务处理的 ACID 特性，就需要引入一个“协调者”（TM）来统一调度所有分布式节点的执行逻辑，这些被调度的分布式节点被称为 AP。TM 负责调度 AP 的行为，并最终决定这些 AP 是否要把事务真正进行提交；因为整个事务是分为两个阶段提交，所以叫 2pc





阶段一：提交事务请求（投票）

1. 事务询问

   协调者向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应

2. 执行事务

   各个参与者节点执行事务操作，并将 Undo 和 Redo 信息记录到事务日志中，尽量把提交过程中所有消耗时间的操作和准备都提前完成确保后面 100%成功提交事务

3. 各个参与者向协调者反馈事务询问的响应

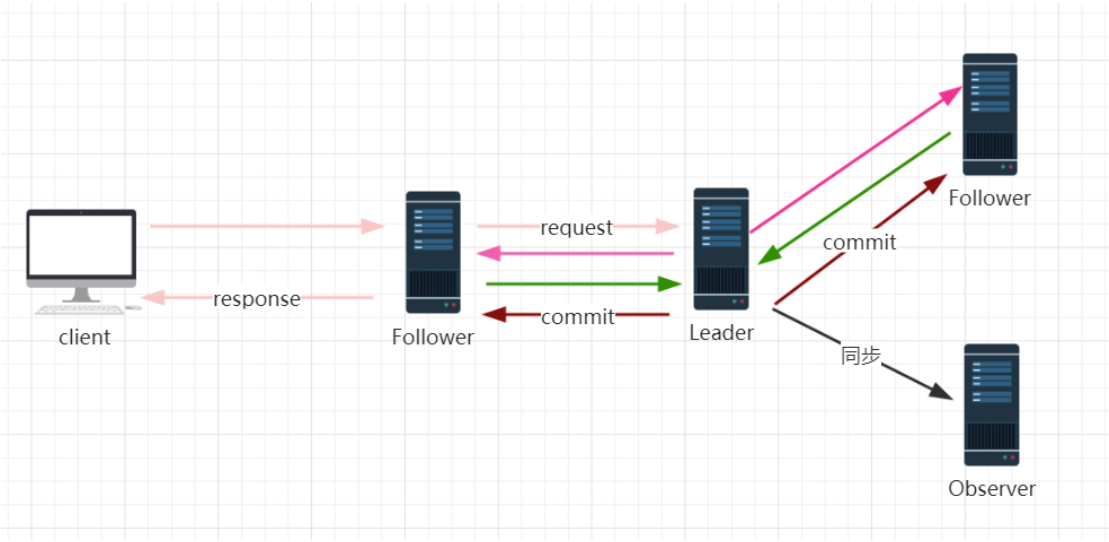
   如果各个参与者成功执行了事务操作，那么就反馈给参与者 yes 的响应，表示事务可以执行；如果参与者没有成功执行事务，就反馈给协调者 no 的响应，表示事务不可以执行，上面这个阶段有点类似协调者组织各个参与者对一次事务操作的投票表态过程，因此 2pc 协议的第一个阶段称为“投票阶段”，即各参与者投票表名是否需要继续执行接下去的事务提交操作。

阶段二：执行事务提交

在这个阶段，协调者会根据各参与者的反馈情况来决定最终是否可以进行事务提交操作，正常情况下包含两种可能:执行事务、中断事务

zookeeper 的集群

在 zookeeper 中，客户端会随机连接到 zookeeper 集群中的一个节点，如果是读请求，就直接从当前节点中读取数据，如果是写请求，那么请求会被转发给 leader 提交事务，然后 leader 会广播事务，只要有超过半数节点写入成功，那么写请求就会被提交（类 2PC 事务）



所有事务请求必须由一个全局唯一的服务器来协调处理，这个服务器就是 Leader 服务器，其他的服务器就是 follower。leader 服务器把客户端的失去请求转化成一个事务 Proposal（提议），并把这个 Proposal 分发给集群中的所有 Follower 服务器。之后 Leader 服务器需要等待所有

Follower 服务器的反馈，一旦超过半数的 Follower 服务器进行了正确的反馈，那么 Leader 就会再次向所有的 Follower 服务器发送 Commit 消息，要求各个 follower 节点对前面的一个 Proposal 进行提交;

集群角色

Leader 角色

Leader 服务器是整个 zookeeper 集群的核心，主要的工作任务有两项

1. 事物请求的唯一调度和处理者，保证集群事物处理的顺序性
2. 集群内部各服务器的调度者

Follower 角色

Follower 角色的主要职责是

1. 处理客户端非事物请求、转发事物请求给 leader 服务器
2. 参与事物请求 Proposal 的投票（需要半数以上服务器通过才能通知 leader commit 数据; Leader 发起的提案，要求 Follower 投票）
3. 参与 Leader 选举的投票

Observer 角色

Observer 是 zookeeper3.3 开始引入的一个全新的服务器角色，从字面来理解，该角色充当了观察者的角色。
观察 zookeeper 集群中的最新状态变化并将这些状态变化同步到 observer 服务器上。Observer 的工作原理与 follower 角色基本一致，而它和 follower 角色唯一的不同在于 observer 不参与任何形式的投票，包括事物请求 Proposal 的投票和 leader 选举的投票。简单来说，observer 服务器只提供非事物请求服务，通常在于不影响集群事物处理能力的前提下提升集群非事物处理的能力

集群组成

通常 zookeeper 是由 2n+1 台 server 组成，每个 server 都知道彼此的存在。对于 2n+1 台 server，只要有 n+1 台 (大多数) server 可用，整个系统保持可用。我们已经了解到，一个 zookeeper 集群如果要对外提供可用的服务，那么集群中必须要有过半的机器正常工作并且彼此之间能够正常通信，基于这个特性，如果向搭建一个能够允许 F 台机器 down 掉的集群，那么就要部署 2*F+1 台服务器构成的 zookeeper 集群。因此 3 台机器构成的 zookeeper 集群，能够在挂掉一台机器后依然正常工作。一个 5 台机器集群的服务，能够对 2 台机器怪调的情况下进行容灾。如果一

台由 6 台服务构成的集群，同样只能挂掉 2 台机器。因此，5 台和 6 台在容灾能力上并没有明显优势，反而增加了网络通信负担。系统启动时，集群中的 server 会选举出一台 server 为 Leader，其它的就作为 follower（这里先不考虑 observer 角色）。

之所以要满足这样一个等式，是因为一个节点要成为集群中的 leader，需要有超过及群众过半数的节点支持，这个涉及到 leader 选举算法。同时也涉及到事务请求的提交投票

ZAB 协议

ZAB（Zookeeper Atomic Broadcast） 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

zab 协议介绍

ZAB 协议包含两种基本模式，分别是

1. 崩溃恢复
2. 原子广播

当整个集群在启动时，或者当 leader 节点出现网络中断、崩溃等情况时，ZAB 协议就会进入恢复模式并选举产生新的 Leader，当 leader 服务器选举出来后，并且集群中有过半的机器和该 leader 节点完成数据同步后（同步指的是数据同步，用来保证集群中过半的机器能够和 leader 服务器的数据状态保持一致），ZAB 协议就会退出恢复模式。
当集群中已经有过半的 Follower 节点完成了和 Leader 状态同步以后，那么整个集群就进入了消息广播模式。这个时候，在 Leader 节点正常工作时，启动一台新的服务器加入到集群，那这个服务器会直接进入数据恢复模式，和 leader 节点进行数据同步。同步完成后即可正常对外提供非事务请求的处理。

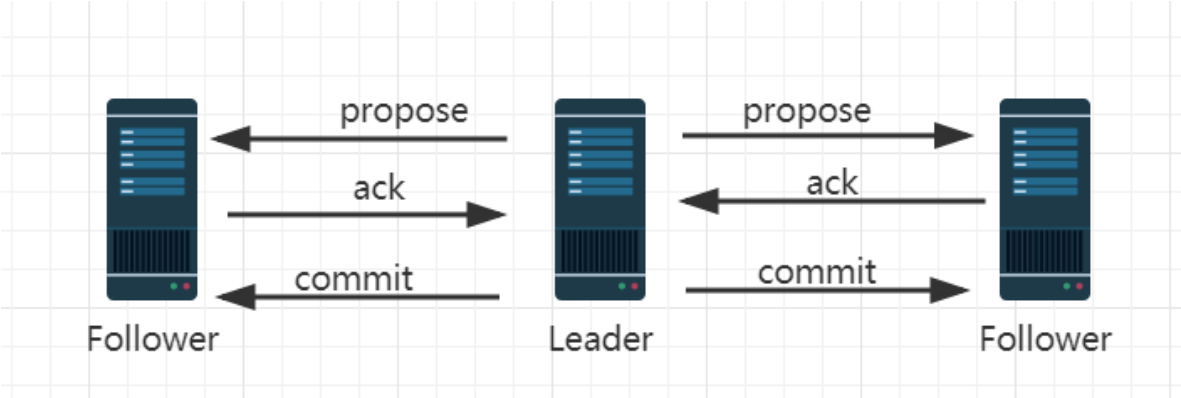
消息广播的实现原理

如果大家了解分布式事务的 2pc 和 3pc 协议的话（不了解也没关系，我们后面会讲），消息广播的过程实际上是一个简化版本的二阶段提交过程

1. leader 接收到消息请求后，将消息赋予一个全局唯一的 64 位自增 id，叫：zxid，通过 zxid 的大小比较既可以实现因果有序这个特征
2. leader 为每个 follower 准备了一个 FIFO 队列(通过 TCP 协议来实现，以实现了全局有序这一个特点)将带有 zxid

的消息作为一个提案 (proposal) 分发给所有的 follower

3. 当 follower 接收到 proposal，先把 proposal 写到磁盘，写入成功以后再向 leader 回复一个 ack
4. 当 leader 接收到合法数量 (超过半数节点) 的 ACK 后，leader 就会向这些 follower 发送 commit 命令，同时会在本地执行该消息
5. 当 follower 收到消息的 commit 命令以后，会提交该消息



leader 的投票过程，不需要 Observer 的 ack，也就是 Observer 不需要参与投票过程，但是 Observer 必须要同步 Leader 的数据从而在处理请求的时候保证数据的一致性

崩溃恢复(数据恢复)

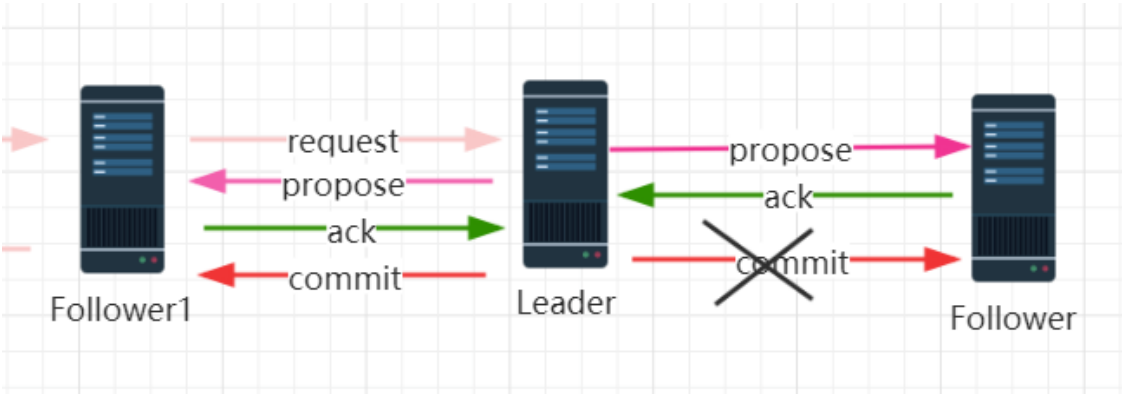
ZAB 协议的这个基于原子广播协议的消息广播过程，在正常情况下是没有任何问题的，但是一旦 Leader 节点崩溃，或者由于网络问题导致 Leader 服务器失去了过半的

Follower 节点的联系（leader 失去与过半 follower 节点联系，可能是 leader 节点和 follower 节点之间产生了网络分区，那么此时的 leader 不再是合法的 leader 了），那么就会进入到崩溃恢复模式。在 ZAB 协议中，为了保证程序的正确运行，整个恢复过程结束后需要选举出一个新的 Leader

为了使 leader 挂了后系统能正常工作，需要解决以下两个问题

1. 已经被处理的消息不能丢失

   当 leader 收到合法数量 follower 的 ACKs 后，就向各个 follower 广播 COMMIT 命令，同时也会在本地执行 COMMIT 并向连接的客户端返回「成功」。但是如果在各个 follower 在收到 COMMIT 命令前 leader 就挂了，导致剩下的服务器并没有执行都这条消息。



leader 对事务消息发起 commit 操作，但是该消息在 follower1 上执行了，但是 follower2 还没有收到 commit，就已经挂了，而实际上客户端已经收到该事务消息处理成

功的回执了。所以在 zab 协议下需要保证所有机器都要执行这个事务消息

2. 被丢弃的消息不能再次出现

当 leader 接收到消息请求生成 proposal 后就挂了，其他 follower 并没有收到此 proposal，因此经过恢复模式重新选了 leader 后，这条消息是被跳过的。 此时，之前挂了的 leader 重新启动并注册成了 follower，他保留了被跳过消息的 proposal 状态，与整个系统的状态是不一致的，需要将其删除。

ZAB 协议需要满足上面两种情况，就必须要设计一个 leader 选举算法：能够确保已经被 leader 提交的事务 Proposal 能够提交、同时丢弃已经被跳过的事务 Proposal。针对这个要求

1. 如果 leader 选举算法能够保证新选举出来的 Leader 服务器拥有集群中所有机器最高编号（ZXID 最大）的事务 Proposal，那么就可以保证这个新选举出来的 Leader 一定具有已经提交的提案。因为所有提案被 COMMIT 之前必须有超过半数的 follower ACK，即必须有超过半数节点的服务器的事务日志上有该提案的 proposal，因此，

只要有合法数量的节点正常工作，就必然有一个节点保存了所有被 COMMIT 消息的 proposal 状态

另外一个，zxid 是 64 位，高 32 位是 epoch 编号，每经过一次 Leader 选举产生一个新的 leader，新的 leader 会将 epoch 号+1，低 32 位是消息计数器，每接收到一条消息这个值+1, 新 leader 选举后这个值重置为 0.这样设计的好处在于老的 leader 挂了以后重启，它不会被选举为 leader，因此此时它的 zxid 肯定小于当前新的 leader。当老的 leader 作为 follower 接入新的 leader 后，新的 leader 会让它将所有的拥有旧的 epoch 号的未被 COMMIT 的 proposal 清除

关于 ZXID

zxid，也就是事务 id，

为了保证事务的顺序一致性，zookeeper 采用了递增的事务 id 号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，它高 32 位是 epoch（ZAB 协议通过 epoch 编号来区分 Leader 周期变化的策略）用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch=（原来的 epoch+1），标识当前属于那个 leader 的统治时期。低 32 位用于递增计数。

epoch：可以理解为当前集群所处的年代或者周期，每个 leader 就像皇帝，都有自己的年号，所以每次改朝换代，leader 变更之后，都会在前一个年代的基础上加 1。这样就算旧的 leader 崩溃恢复之后，也没有人听他的了，因为 follower 只听从当前年代的 leader 的命令。

epoch 的变化大家可以做一个简单的实验，

1. 启动一个 zookeeper 集群。
2. 在 /tmp/zookeeper/VERSION-2 路径下会看到一个 currentEpoch 文件。文件中显示的是当前的 epoch
3. 把 leader 节点停机，这个时候在看 currentEpoch 会有变化。 随着每次选举新的 leader，epoch 都会发生变化

leader 选举

Leader 选举会分两个过程

启动的时候的 leader 选举、 leader 崩溃的时候的的选举

服务器启动时的 leader 选举

每个节点启动的时候状态都是 LOOKING，处于观望状态，接下来就开始进行选主流程

进行 Leader 选举，至少需要两台机器（具体原因前面已经讲过了），我们选取 3 台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器 Server1 启动时，它本身是无法进行和完成 Leader 选举，当第二台服务器 Server2 启动时，这个时候两台机器可以相互通信，每台机器都试图找到 Leader，于是进入 Leader 选举过程。选举过程如下

(1) 每个 Server 发出一个投票。由于是初始情况，Server1 和 Server2 都会将自己作为 Leader 服务器来进行投票，每次投票会包含所推举的服务器的 myid 和 ZXID、epoch，使用(myid, ZXID,epoch)来表示，此时 Server1 的投票为(1, 0)，Server2 的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) 接受来自各个服务器的投票。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票(epoch)、是否来自 LOOKING 状态的服务器。

(3) 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行 PK，PK 规则如下

   i. 优先检查 ZXID。ZXID 比较大的服务器优先作为 Leader

ii. 如果 ZXID 相同，那么就比较 myid。myid 较大的服务器作为 Leader 服务器。

对于 Server1 而言，它的投票是(1, 0)，接收 Server2 的投票为(2, 0)，首先会比较两者的 ZXID，均为 0，再比较 myid，此时 Server2 的 myid 最大，于是更新自己的投票为(2, 0)，然后重新投票，对于 Server2 而言，它不需要更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

(4) 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于 Server1、Server2 而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了 Leader。

(5) 改变服务器状态。一旦确定了 Leader，每个服务器就会更新自己的状态，如果是 Follower，那么就变更为 FOLLOWING，如果是 Leader，就变更为 LEADING。

运行过程中的 leader 选举

当集群中的 leader 服务器出现宕机或者不可用的情况时，那么整个集群将无法对外提供服务，而是进入新一轮的 Leader 选举，服务器运行期间的 Leader 选举和启动时期的 Leader 选举基本过程是一致的。

(1) 变更状态。Leader 挂后，余下的非 Observer 服务器都会将自己的服务器状态变更为 LOOKING，然后开始进入 Leader 选举过程。

(2) 每个 Server 会发出一个投票。在运行期间，每个服务器上的 ZXID 可能不同，此时假定 Server1 的 ZXID 为 123，Server3 的 ZXID 为 122；在第一轮投票中，Server1 和 Server3 都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。接收来自各个服务器的投票。与启动时过程相同。

(3) 处理投票。与启动时过程相同，此时，Server1 将会成为 Leader。

(4) 统计投票。与启动时过程相同。

(5) 改变服务器的状态。与启动时过程相同

Leader 选举源码分析

有了理论基础以后，我们先带大家读一下源码，看看他的实现逻辑。然后我们再通过图形的方式帮助大家理解

从入口函数 QUORUMPEERMAIN 开始

```java
public static void main(String[] args) {
    QuorumPeerMain main = new QuorumPeerMain();
    try {
        main.initializeAndRun(args);
```

```java
protected void initializeAndRun(String[] args)
    throws ConfigException, IOException, AdminServerException
{
    QuorumPeerConfig config = new QuorumPeerConfig();
    if (args.length == 1) {
        config.parse(args[0]);
    }

    // Start and schedule the the purge task
    DatadirCleanupManager purgeMgr = new DatadirCleanupManager(config
            .getDataDir(), config.getDataLogDir(), config
            .getSnapRetainCount(), config.getPurgeInterval());
    purgeMgr.start();
    //判断是standalone模式还是集群模式
    if (args.length == 1 && config.isDistributed()) {
        runFromConfig(config);
    } else {
        LOG.warn("Either no config or no quorum defined in config, running "
                + " in standalone mode");
        // there is only server in the quorum -- run as standalone
        ZooKeeperServerMain.main(args);
    }
}
```

```
public void runFromConfig(QuorumPeerConfig config)
        throws IOException, AdminServerException
{
  try {
      ManagedUtil.registerLog4jMBeans();
  } catch (JMException e) {
      LOG.warn("Unable to register log4j JMX control", e);
  }

  LOG.info("Starting quorum peer");
  try {
      ServerCnxnFactory cnxnFactory = null;
      ServerCnxnFactory secureCnxnFactory = null;
      //为客户端提供读写的server，  也就是2181这个端口的访问功能
      if (config.getClientPortAddress() != null) {
          cnxnFactory = ServerCnxnFactory.createFactory();
          cnxnFactory.configure(config.getClientPortAddress(),
                  config.getMaxClientCnxns(),
                  secure: false);
      }
```

```
//ZK的逻辑主线程，负责选举、投票
quorumPeer = getQuorumPeer();
quorumPeer.setTxnFactory(new FileTxnSnapLog(
            config.getDataLogDir(),
            config.getDataDir()));
quorumPeer.enableLocalSessions(config.areLocalSessionsEnabled());
quorumPeer.enableLocalSessionsUpgrading(
    config.isLocalSessionsUpgradingEnabled());
//quorumPeer.setQuorumPeers(config.getAllMembers());
quorumPeer.setElectionType(config.getElectionAlg());
quorumPeer.setMyid(config.getServerId());
quorumPeer.setTickTime(config.getTickTime());
quorumPeer.setMinSessionTimeout(config.getMinSessionTimeout());
quorumPeer.setMaxSessionTimeout(config.getMaxSessionTimeout());
quorumPeer.setInitLimit(config.getInitLimit());
quorumPeer.setSyncLimit(config.getSyncLimit());
quorumPeer.setConfigFileName(config.getConfigFilename());
quorumPeer.setZKDatabase(new ZKDatabase(quorumPeer.getTxnFactory()));
quorumPeer.setQuorumVerifier(config.getQuorumVerifier(), writeToDisk: false);
if (config.getLastSeenQuorumVerifier()!=null) {
    quorumPeer.setLastSeenQuorumVerifier(config.getLastSeenQuorumVerifier(),
}
quorumPeer.initConfigInZKDatabase();
quorumPeer.setCnxnFactory(cnxnFactory);
```

设置各种参数

启动主线程，QuorumPeer 重写了 Thread.start 方法

```
quorumPeer.setQuorumListenOnAllIPs(config.getQuorumL
//启动主线程
quorumPeer.start();
quorumPeer.join();
```

调用 QUORUMPEER 的 START 方法

```java
@Override
public synchronized void start() {
    if (!getView().containsKey(myid)) {
        throw new RuntimeException("My id " + myid + " not in the peer list");
    }
    loadDataBase();  //恢复DB
    startServerCnxnFactory();
    try {
        adminServer.start();
    } catch (AdminServerException e) {
        LOG.warn("Problem starting AdminServer", e);
        System.out.println(e);
    }
    startLeaderElection();//选举初始化
    super.start();
}
```

loaddatabase， 主要是从本地文件中恢复数据，以及获取最新的 zxid

```java
private void loadDataBase() {
    try {
        zkDb.loadDataBase(); //从本地文件恢复db

        // load the epochs
        //从最新的zxid恢复epoch变量、zxid64位，前32位是epoch的值，后32位是zxid
        long lastProcessedZxid = zkDb.getDataTree().lastProcessedZxid;
        long epochOfZxid = ZxidUtils.getEpochFromZxid(lastProcessedZxid);
        try {
            //从文件中读取当前的epoch
            currentEpoch = readLongFromFile(CURRENT_EPOCH_FILENAME);
        } catch(FileNotFoundException e) {
            // pick a reasonable epoch number
            // this should only happen once when moving to a
            // new code version
            currentEpoch = epochOfZxid;
            LOG.info(CURRENT_EPOCH_FILENAME
                    + " not found! Creating with a reasonable default of {}. This should only happen
                    currentEpoch);
            writeLongToFile(CURRENT_EPOCH_FILENAME, currentEpoch);
        }
        if (epochOfZxid > currentEpoch) {
            throw new IOException("The current epoch, " + ZxidUtils.zxidToString(currentEpoch) + ", is
```

初始化 LEADERELECTION

```
public synchronized void start() {
    if (!getView().containsKey(myid)) {
        throw new RuntimeException("My id " + my
    }
    loadDataBase();  //恢复DB
    startServerCnxnFactory();
    try {
        adminServer.start();
    } catch (AdminServerException e) {
        LOG.warn("Problem starting AdminServer"
        System.out.println(e);
    }
    startLeaderElection();//选举初始化
    super.start();
}
```

```
synchronized public void startLeaderElection() {
    try {
        //如果当前节点的状态是LOOKING，则投票给自己
        if (getPeerState() == ServerState.LOOKING) {
            currentVote = new Vote(myid, getLastLoggedZxid(), getCurrentEpoch());
        }
    } catch(IOException e) {
        RuntimeException re = new RuntimeException(e.getMessage());
        re.setStackTrace(e.getStackTrace());
        throw re;
    }
    //根据配置获取选举选法
    this.electionAlg = createElectionAlgorithm(electionType);
}
```

配置选举算法，选举算法有 3 种，可以通过在 zoo.cfg 里面进行配置，默认是 fast 选举

```
protected Election createElectionAlgorithm(int electionAlgorithm){
    Election le=null;

    //TODO: use a factory rather than a switch
    switch (electionAlgorithm) {
        case 1:
            le = new AuthFastLeaderElection( self: this);
            break;
        case 2:
            le = new AuthFastLeaderElection( self: this,  auth: true);
            break;
        case 3:
            //Leader选举IO负责类
            qcm = new QuorumCnxManager( self: this);
            QuorumCnxManager.Listener listener = qcm.listener;
            if(listener != null){
                listener.start(); //启动已绑定端口的选举线程，等待集群中其他机器连接
                //基于TCP的选举算法
                FastLeaderElection fle = new FastLeaderElection( self: this, qcm);
                fle.start();
                le = fle;
            } else {
                LOG.error("Null listener when initializing cnx manager");
```

继续看 FastLeaderElection 的初始化动作，主要初始化了业务层的发送队列和接收队列

```
public FastLeaderElection(QuorumPeer self, QuorumCnxM
    this.stop = false;
    this.manager = manager;
    starter(self, manager);
}
```

```
private void starter(QuorumPeer self, QuorumCn
    this.self = self;
    proposedLeader = -1;
    proposedZxid = -1;
    //业务层发送队列，业务对象ToSend
    sendqueue = new LinkedBlockingQueue<ToSend
    //业务层接收队列，业务对象Notification
    recvqueue = new LinkedBlockingQueue<Notifi
    this.messenger = new Messenger(manager);
}
```

接下来调用 fle.start()，也就是会调用 FastLeaderElection start()方法，该方法主要是对发送线程和接收线程的初始化，左边是 FastLeaderElection 的 start，右边是 messager.start()

```
/**
 * This method starts the sender and receiver threads.
 */
public void start() {
    this.messenger.start();
}
```

```
/**
 * Starts instances of WorkerSender and WorkerReceiver
 */
void start(){
    this.wsThread.start();//启动业务层发送线程，讲消息发送给IO负责类QuorumCnxManager
    this.wrThread.start();//启动业务接收线程，从IO负责类QuorumCnxManager接收消息
}
```

wsThread 和 wrThread 的初始化动作在

FastLeaderElection 的 starter 方法里面进行，这里面有两个内部类，一个是 WorkerSender，一个是 WorkerReceiver，负责发送投票信息和接收投票信息

```
private void starter(QuorumPeer self, QuorumCnxManager manager) {
    this.self = self;
    proposedLeader = -1;
    proposedZxid = -1;
    //业务层发送队列，业务对象ToSend
    sendqueue = new LinkedBlockingQueue<ToSend>();
    //业务层接收队列，业务对象Notification
    recvqueue = new LinkedBlockingQueue<Notification>();
    this.messenger = new Messenger(manager);
}
```

```
Messenger(QuorumCnxManager manager) {

    this.ws = new WorkerSender(manager);

    this.wsThread = new Thread(this.ws,
            name: "WorkerSender[myid=" + self.getId() + "]");
    this.wsThread.setDaemon(true);

    this.wr = new WorkerReceiver(manager);

    this.wrThread = new Thread(this.wr,
            name: "WorkerReceiver[myid=" + self.getId() + "]");
    this.wrThread.setDaemon(true);
}
```

然后再回到 QuorumPeer.java。 FastLeaderElection 初始化完成以后，调用 super.start()，最终运行 QuorumPeer 的 run 方法

```
@Override
public synchronized void start() {
    if (!getView().containsKey(myid)) {
        throw new RuntimeException("My id " + myid + " not in
    }
    loadDataBase();  //恢复DB
    startServerCnxnFactory();
    try {
        adminServer.start();
    } catch (AdminServerException e) {
        LOG.warn("Problem starting AdminServer", e);
        System.out.println(e);
    }
    startLeaderElection();//选举初始化
    super.start();
}
```

QuorumPeer.java

前面部分主要是做 JMX 监控注册

```
@Override
public void run() {
    updateThreadName();

    LOG.debug("Starting quorum peer");
    try { //此处通过JMX来监控一些属性
        jmxQuorumBean = new QuorumBean( peer: this);
        MBeanRegistry.getInstance().register(jmxQuorumBean,  parent: null);
        for(QuorumServer s: getView().values()){
            ZKMBeanInfo p;
            if (getId() == s.id) {
                p = jmxLocalPeerBean = new LocalPeerBean(this);
                try {
                    MBeanRegistry.getInstance().register(p, jmxQuorumBean);
                } catch (Exception e) {
                    LOG.warn("Failed to register with JMX", e);
                    jmxLocalPeerBean = null;
                }
            } else {
                RemotePeerBean rBean = new RemotePeerBean(s);
                try {
                    MBeanRegistry.getInstance().register(rBean, jmxQuorumBean);
                    jmxRemotePeerBean.put(s.id, rBean);
```

重要的代码在这个 while 循环里

```
while (running) {
    switch (getPeerState()) { //判断当前节点的状态
    case LOOKING: //如果是LOOKING，则进入选举流程
        LOG.info("LOOKING");

        if (Boolean.getBoolean( name: "readonlymode.enabled")) {
            LOG.info("Attempting to start ReadOnlyZooKeeperServer");

            // Create read-only server but don't start it immediately
            final ReadOnlyZooKeeperServer roZk =
                new ReadOnlyZooKeeperServer(logFactory, self: this, this.zkDb);

            // Instead of starting roZk immediately, wait some grace
            // period before we decide we're partitioned.
            //
            // Thread is used here because otherwise it would require
            // changes in each of election strategy classes which is
            // unnecessary code coupling.
            Thread roZkMgr = run() → {
                    try {
                        // lower-bound grace period to 2 secs
                        sleep(Math.max(2000, tickTime));
                        if (ServerState.LOOKING.equals(getPeerState())) {
                            roZk.startup();
```

调用 setCurrentVote(makeLEStrategy().lookForLeader());，最终根据策略应该运行 FastLeaderElection 中的选举算法

```
try {
    roZkMgr.start();
    reconfigFlagClear();
    if (shuttingDownLE) {
        shuttingDownLE = false;
        startLeaderElection();
    }
    //此处通过策略模式来决定当前用哪个选举算法来进行领导选举
    setCurrentVote(makeLEStrategy().lookForLeader());
} catch (Exception e) {
    LOG.warn("Unexpected exception", e);
    setPeerState(ServerState.LOOKING);
} finally {
```

LOOKFORLEADER 开始选举

```
public Vote lookForLeader() throws InterruptedException {
    try {
        self.jmxLeaderElectionBean = new LeaderElectionBean();
        MBeanRegistry.getInstance().register(
                self.jmxLeaderElectionBean, self.jmxLocalPeerBean);
    } catch (Exception e) {
        LOG.warn("Failed to register with JMX", e);
        self.jmxLeaderElectionBean = null;
    }
    if (self.start_fle == 0) {
       self.start_fle = Time.currentElapsedTime();
    }
    try {
        //收到的投票
        HashMap<Long, Vote> recvset = new HashMap<~>();
        //存储选举结果
        HashMap<Long, Vote> outofelection = new HashMap<~>();

        int notTimeout = finalizeWait;
```

```
synchronized(this){
    logicalclock.incrementAndGet(); //增加逻辑时钟
    //吃要自己的zxid和epoch
    updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
}

LOG.info("New election. My id =  " + self.getId() +
        ", proposed zxid=0x" + Long.toHexString(proposedZxid));
sendNotifications(); //发送投票，包括发送给自己

/*
 * Loop in which we exchange notifications until we find a leader
 */

while ((self.getPeerState() == ServerState.LOOKING) &&
        (!stop)){//主循环，直到选举出leader
    /*
     * Remove next notification from queue, times out after 2 times
     * the termination time
     */
    // 从IO线程里拿到投票消息，自己的投票也在这里处理
    Notification n = recvqueue.poll(notTimeout,
            TimeUnit.MILLISECONDS);
```

```
            /*
             * Sends more notifications if haven't received enough.
             * Otherwise processes new notification.
             */
            if(n == null){
                //如果空间的情况下，消息发完了，继续发送，一直到选出leader为止
                if(manager.haveDelivered()){
                    sendNotifications();
                } else {
                    //消息还没投递出去，可能是其他server还没启动，尝试再连接
                    manager.connectAll();
                }

                /*
                 * Exponential backoff
                 */
                //延长超时时间
                int tmpTimeOut = notTimeout*2;
                notTimeout = (tmpTimeOut < maxNotificationInterval?
                        tmpTimeOut : maxNotificationInterval);
                LOG.info("Notification time out: " + notTimeout);
            }
            //收到了投票消息，判断收到的消息是不是属于这个集群内
            else if (self.getCurrentAndNextConfigVoters().contains(n.sid)) {
```

```
//收到了投票消息，判断收到的消息是不是属于这个集群内
else if (self.getCurrentAndNextConfigVoters().contains(n.sid)) {
    /*
     * Only proceed if the vote comes from a replica in the current or next
     * voting view.
     */
    switch (n.state) {//判断收到消息的节点的状态
    case LOOKING:
        if (getInitLastLoggedZxid() == -1) {
            LOG.debug("Ignoring notification as our zxid is -1");
            break;
        }
        if (n.zxid == -1) {
            LOG.debug("Ignoring notification from member with -1 zxid" + n.si
            break;
        }
        // If notification > current, replace and send messages out
        //判断接收到的节点epoch大于logicalclock，则表示当前是新一轮的选举
        if (n.electionEpoch > logicalclock.get()) {
            logicalclock.set(n.electionEpoch); //更新本地的logicalclock
            recvset.clear(); //清空接收队列
            //检查收到的这个消息是否可以胜出，一次比较epoch，zxid、myid
            if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
```

```
protected boolean totalOrderPredicate(long newId, long newZxid, long newEpoch, lon
    LOG.debug("id: " + newId + ", proposed id: " + curId + ", zxid: 0x" +
            Long.toHexString(newZxid) + ", proposed zxid: 0x" + Long.toHexString(cu
    if(self.getQuorumVerifier().getWeight(newId) == 0){
        return false;
    }

    /*
     * We return true if one of the following three cases hold:
     * 1- New epoch is higher
     * 2- New epoch is the same as current epoch, but new zxid is higher
     * 3- New epoch is the same as current epoch, new zxid is the same
     *  as current zxid, but server id is higher.
     */
     // 1.判断消息里的epoch是不是比当前的大，如果大则消息中id对应的服务器就是leader
     // 2. 如果epoch相等则判断zxid，如果消息里的zxid大，则消息中id对应的服务器就是leade
     // 3. 如果前面两个都相等那就比较服务器id，如果大，则其就是leader

    return ((newEpoch > curEpoch) ||
            ((newEpoch == curEpoch) &&
            ((newZxid > curZxid) || ((newZxid == curZxid) && (newId > curId)))));
}
```

```
                if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
                        getInitId(), getInitLastLoggedZxid(), getPeerEpoch())) {
                    //胜出以后，把投票改为对方的票据
                    updateProposal(n.leader, n.zxid, n.peerEpoch);
                } else {//否则，票据不变
                    updateProposal(getInitId(),
                            getInitLastLoggedZxid(),
                            getPeerEpoch());
                }
                sendNotifications();//继续广播消息，让其他节点知道我现在的票据
                //如果收到的消息epoch小于当前节点的epoch，则忽略这条消息
            } else if (n.electionEpoch < logicalclock.get()) {
                if(LOG.isDebugEnabled()){
                    LOG.debug("Notification election epoch is smaller than logicalclock. n.electionEpoch = 0x"
                            + Long.toHexString(n.electionEpoch)
                            + ", logicalclock=0x" + Long.toHexString(logicalclock.get()));
                }
                break;
                //如果是epoch相同的话，就继续比较zxid、myid，如果胜出，则更新自己的票据，并且发出广播
```

```
            //如果是epoch相同的话，就继续比较zxid、myid，如果胜出，则更新自己的票据，并且发出广播
        } else if (totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
                proposedLeader, proposedZxid, proposedEpoch)) {
            updateProposal(n.leader, n.zxid, n.peerEpoch);
            sendNotifications();
        }

        if(LOG.isDebugEnabled()){
            LOG.debug("Adding vote: from=" + n.sid +
                    ", proposed leader=" + n.leader +
                    ", proposed zxid=0x" + Long.toHexString(n.zxid) +
                    ", proposed election epoch=0x" + Long.toHexString(n.electionEpoch));
        }
        //添加到本机投票集合，用来做选举终结判断
        recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));
```

```
            //判断选举是否结束，默认算法是超过半数server同意
            if (termPredicate(recvset,
                    new Vote(proposedLeader, proposedZxid,
                            logicalclock.get(), proposedEpoch))) {
                // Verify if there is any change in the proposed leader
                //一直等新的notification到达，直到超时
                while((n = recvqueue.poll(finalizeWait,
                        TimeUnit.MILLISECONDS)) != null){
                    if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
                            proposedLeader, proposedZxid, proposedEpoch)){
                        recvqueue.put(n);
                        break;
                    }
                }
                /*
                 * This predicate is true once we don't read any new
                 * relevant message from the reception queue
                 */
                //确定leader
                if (n == null) {
                    //修改状态，LEADING or FOLLOWING
```

```
private boolean termPredicate(HashMap<Long, Vote> votes, Vote vote) {
    SyncedLearnerTracker voteSet = new SyncedLearnerTracker();
    voteSet.addQuorumVerifier(self.getQuorumVerifier());
    if (self.getLastSeenQuorumVerifier() != null
            && self.getLastSeenQuorumVerifier().getVersion() > self
                    .getQuorumVerifier().getVersion()) {
        voteSet.addQuorumVerifier(self.getLastSeenQuorumVerifier());
    }

    /*
     * First make the views consistent. Sometimes peers will have differen
     * zxids for a server depending on timing.
     */
    //遍历已经接收的投票集合，把等于当前投票的项放入set
    for (Map.Entry<Long, Vote> entry : votes.entrySet()) {
        if (vote.equals(entry.getValue())) {
            voteSet.addAck(entry.getKey());
        }
    }
    //统计set集合，查看投某个id的票数是否超过一半
    return voteSet.hasAllQuorums();
}
```

```
                //确定leader
                if (n == null) {
                    //修改状态，LEADING or FOLLOWING
                    self.setPeerState((proposedLeader == self.getId()) ?
                            ServerState.LEADING: learningState());
                    //返回最终投票结果
                    Vote endVote = new Vote(proposedLeader,
                            proposedZxid, proposedEpoch);
                    leaveInstance(endVote);
                    return endVote;
                }
            }
            break;
```

```
            //如果收到的选票状态不是LOOKING，比如这台机器刚加入一个已经正在运行的zk集群时
            //OBSERVING机器不参数选举
        case OBSERVING:
            LOG.debug("Notification from observer: " + n.sid);
            break;
            //这2种需要参与选举
        case FOLLOWING:
        case LEADING:
            /*
             * Consider all notifications from the same epoch
             * together.
             */
            if(n.electionEpoch == logicalclock.get()){ //判断epoch是否相同
                //加入到本机的投票集合
                recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));
                //投票是否结束，如果结束，再确认LEADER是否有效
                //如果结束，修改自己的状态并返回投票结果
                if(termPredicate(recvset, new Vote(n.leader,
                                n.zxid, n.electionEpoch, n.peerEpoch, n.state))
                                && checkLeader(outofelection, n.leader, n.electionEpoch)) {
                    self.setPeerState((n.leader == self.getId()) ?
                            ServerState.LEADING: learningState());
```

消息如何广播，看 SENDNOTIFICATIONS

```
private void sendNotifications() {
    for (long sid : self.getCurrentAndNextConfigVoters()) {//循环发送
        QuorumVerifier qv = self.getQuorumVerifier();
        //消息实体
        ToSend notmsg = new ToSend(ToSend.mType.notification,
                proposedLeader,
                proposedZxid,
                logicalclock.get(),
                QuorumPeer.ServerState.LOOKING,
                sid,
                proposedEpoch, qv.toString().getBytes());
        if(LOG.isDebugEnabled()){
            LOG.debug("Sending Notification: " + proposedLeader + " (n.leader), 0x"  +
                  Long.toHexString(proposedZxid) + " (n.zxid), 0x" + Long.toHexString(logicalclock.
                  " (n.round), " + sid + " (recipient), " + self.getId() +
                  " (myid), 0x" + Long.toHexString(proposedEpoch) + " (n.peerEpoch)");
        }
        sendqueue.offer(notmsg); //添加到发送队列，这个队列会被workerSender消费
    }
}
```

WORKERSENDER

```
class WorkerSender extends ZooKeeperThread {
    volatile boolean stop;
    QuorumCnxManager manager;

    WorkerSender(QuorumCnxManager manager){
        super( threadName: "WorkerSender");
        this.stop = false;
        this.manager = manager;
    }

    public void run() {
        while (!stop) {
            try {
                //从发送队列中获取消息实体
                ToSend m = sendqueue.poll( timeout: 3000, TimeUnit.MILLISECONDS);
                if(m == null) continue;

                process(m);
            } catch (InterruptedException e) {
                break;
            }
        }
```

```
    void process(ToSend m) {
        ByteBuffer requestBuffer = buildMsg(m.state.ordinal(),
                                            m.leader,
                                            m.zxid,
                                            m.electionEpoch,
                                            m.peerEpoch,
                                            m.configData);

        manager.toSend(m.sid, requestBuffer);

    }
}
```

```
public void toSend(Long sid, ByteBuffer b) {
    /*
     * If sending message to myself, then simply enqueue it (loopback).
     */
    if (self.getId() == sid) { //如果是自己，就不走网络发送，直接添加到本地接收队列
        b.position(0);
        addToRecvQueue(new Message(b.duplicate(), sid));
        /*
         * Otherwise send to the corresponding thread to send.
         */
    } else {
        /*
         * Start a new connection if doesn't have one already.
         */
        ////发送给别的节点，判断之前是不是发送过
        ArrayBlockingQueue<ByteBuffer> bq = new ArrayBlockingQueue<~>(
            SEND_CAPACITY);
        //这个SEND_CAPACITY的大小是1，所以如果之前已经有一个还在等待发送，则会把之前的一个删除掉，发送新的
        ArrayBlockingQueue<ByteBuffer> oldq = queueSendMap.putIfAbsent(sid, bq);
        if (oldq != null) {
            addToSendQueue(oldq, b);
        } else {
            addToSendQueue(bq, b);
        }
        }
        //这里是真正的发送逻辑了
        connectOne(sid);

    }
}
```

FastLeaderElection 选举过程

其实在这个投票过程中就涉及到几个类，

FastLeaderElection：FastLeaderElection 实现了 Election 接口，实现各服务器之间基于 TCP 协议进行选举

Notification：内部类，Notification 表示收到的选举投票信息（其他服务器发来的选举投票信息），其包含了被选举者的 id、zxid、选举周期等信息

ToSend：ToSend 表示发送给其他服务器的选举投票信息，

也包含了被选举者的 id、zxid、选举周期等信息

Messenger： Messenger 包含了 WorkerReceiver 和 WorkerSender 两个内部类；

WorkerReceiver 实现了 Runnable 接口，是选票接收器。其会不断地从 QuorumCnxManager 中获取其他服务器发来的选举消息，并将其转换成一个选票，然后保存到 recvqueue 中

WorkerSender 也实现了 Runnable 接口，为选票发送器，其会不断地从 sendqueue 中获取待发送的选票，并将其传递到底层 QuorumCnxManager 中