**ZAB**

# 1.相关概念

* logicalClock：每次调用lookForLeader方法，logicalClock ++，logicalClock表示当前节点所处选举轮次，
* electionEpoch：它是发出Notification的节点的logicalClock
* peerEpoch：代表当前leader的任期，每次选举结束，会把leader最终的electionEpoch复制到peerEpoch中，选举期间，节点的peerEpoch不会变，electionEpoch++。
* ZXID：事务请求的唯一标识，由leader分配，高32位代表peerEpoch，低32位代表请求计数。
* lastProcessedZXID：最后一次commit事务请求ZXID
* committedLog：LinkedList<Proposal>，zookeeper会保存最近一段时间内执行的事务请求议案，默认长度为500，其范围在[max，min]之间
* outstandingProposals（待confirm）：ConcurrentHashMap<long，proposal>，属于leader属性

1. 每提出一个议案，都会放入其中
2. 一旦议案超过半数认可，从其中删除。
3. Azxid + 1 = Bzxid ,B先获得majority的ack，B也不会先提交，现判断outstandingProposals中是否存在A，存在说明A未提交，则B也不会提交

* toBeApplied（待apply）：ConcurrentLinkedQueue<Proposal>，leader拥有的属性，当majority ack后，会放入该列表中，一旦议案应用到zookeeper的内存树，然后就会从其中删除
* Notification：(SID,leader,state)，SID表示消息的发送者，leader表示投票获得者的SID，state为looking表示还在选举过程中，leading表示leader已经选出，此条Notification是通告
* recvSet：HashMap<long, Vote>即（SID，vote）用于收集looking、following、leading状态下的server投票，key值为SID表示消息的发送者，vote是SID的最新投票信息，当收到的消息后发现自己的任期peerEpoch小于投票中的任期，则清空recvSet
* outofelection：HashMap<long, vote>，用于收集following、leading状态下的server投票（状态表示leader选举已经完成）

# 2.node状态与角色

* looking：进入leader选举状态
* following：leader选举结束，进入follower状态
* leading：leader选举结束，进入leader状态
* observing：处于observer状态

## 2.1.observer和follower的区别

Observer和follower相似，其优点如下：

* Observer可以扩展zookeeper集群，因为observer的加入或者删除都会影响集群运行
* 大幅度提高读取速度
* 一定程度上提升容灾，observer宕机不会影响集群提供服务

区别如下：

* Observer没有选主投票和参与议案提交的过程
* Observer只会收到leader发来的commit消息，因为他没有参与议案过程

# 3.ZAB整体过程

* 选举----fastLeaderElection
* 发现----discovery
* 同步----synchronized
* 广播----broadcast

# 4.选举

主要采用fastleaderElection算法：

## 3.1.触发条件

* 节点刚刚启动，会进入looking状态
* Follower检测超时，进入looking状态
* Leader未收到多数派的心跳回复，进入looking状态

Raft中只有follower在electionTimeout超时后才有发起选举的可能，leader即使失去与多数派的联系，也不会放弃自己的leader地位，相反会不断发送消息给所有follower，直到leader收到新的选举信息term大于自身的任期号，或是收到leader发送的日志请求，leader会放弃领导地位，转为follower。

## 3.2.结束选举

* recvSet中半数节点投票给A，才会认可A为自己的leader，退出循环
* outOfElection中半数节点认可A为leader，才会认可A为leader，退出循环

## 3.3.选举过程

lookForLeader是其选主的函数。

1. 首先创建两个HashMap------ recvSet和outofelection
2. 更新logicalClock++，updateProposal(ID，ZXID，peerEpoch)
3. 向其他服务器发送自己的选票
4. 进入while循环直至找到leader，其循环体内容如下：
5. 从recvqueue接收队列中取出投票，如果没有选票，继续发送自己的选票
6. 如果recvqueue队列中有选票，则判断选票发送者的状态，根据状态做出相应的响应。

过程如下：

* 如果选票发送者的状态为looking，代表它还没有选出leader，下面针对electionEpoch和自己的logicalClock进行处理
* electionEpoch > logicalClock，则表示节点自身的recvSet中vote过时，进行recvset.clear()，logicalClock = electionEpoch，同时判断是否需要更新自己的选票（选票PK，规则先比较peerEpoch，再ZXID，最后SID），PK过后重新发送新的选票
* electionEpoch < logicalClock，表示节点的选举轮次更新，不做任何处理
* electionEpoch == logicalClock，进一步PK选票，对方获胜则更新选票，并发送新的选票信息
* recvset保存本轮次的所有外部投票（looking、following、leading）
* 根据recvset判断是否选出leader，如果选出leader的话将会等待200ms，看看recvqueue中是否会收到更优的leader，完成则退出lookForLeader
* 如果选票发送者的状态为following或leading，则表示已经选出leader
* 如果electionEpoch == logicalClock，将该节点的选票放入recvset中，然后根据recvset判断节点发送选票中的leader是否满足，如果完成了选举则退出lookForLeader
* 如果将投票信息放入outofelection，判断leader是否选出，依据为outofelection中有一条是来自leader，并且outofelection中有超过半数的服务器推荐了这个leader，如果完成了选举则退出lookForLeader

这里进行了两次判断，一次时判断recvset，一次时判断outofelection，当electionEpoch=logicalClock时，表明正在选举中，判断时采用recvset，当electionEpoch！=logicalClock时，表明选举结束，判断outofelection中的选票是否满足选举结束

# 5.同步

一旦leader选举完成，就进入恢复阶段，即follower同步leader上的数据信息。

1. 通信初始化

leader会创建一个serverSocket，接收follower的连接，leader会为每一个连接用一个learnerHandler线程来进行服务

1. 重新选出一个peerEpoch

follower会向leader发送一个Leader.FollowerInfo信息，包含自己的peerEpoch，leader收到上述的peerEpoch，选出最大的peerEpoch+1作为新的peerEpoch并发送给所有follower一个Leader.LeaderInfo信息，包含上述新的peerEpoch，leader的所有learnerHandler会记录上述各自follower的lastProcessedZXID，然后根据这个lastProcessedZXID和leader的进行比较，或补偿，或删除。

1. 判断follower的lastProcessedZXID是否在leader的[minCommittedLog，maxCommittedLog]之间：

* Follower的lastProcessedZXID=leader的LastProcessedZXID，说明已经保持同步了
* 若在[max, min]之间，则表示lastProcessedZXID到max之间的议案follower没有，则将这部分内容发送给follower并发送议案的commit命令，上述可能存在一个问题，follower的lastProcessedZXID虽然在最小和最大之间，但leader中并没有这个议案，则leader会告诉follower删除这部分日志，重新发送新的议案并提交
* Follower的lastProcessedZXID > maxCommittedLog，删除follower大于leader的部分日志
* Follower的lastProcessedZXID < minCommittedLog，直接采用快照的方式恢复

1. 未处理的事务议案的同步

* LearnerHandler还会从leader的toBeApplied（多数派ack但未commit，应用到内存删除）数据中将大于该follower的lastProcessedZXID的议案进行发送和提交
* learnerHandler还会从leader的outstandingProposals（刚提出，多数派ack后删除）中大于该follower的lastProcessedZXID的议案就行发送但并不提交

1. leader发送leader.NewLeader以及leader.UpToDate命令

该命令在同步结束之后发送，follower收到该命令后会执行版本快照等初始化操作，如果收到ack表明follower完成同步并初始化成功。Leader进入心跳检测阶段，不断向follower发送心跳命令， 检测是否过半节点回复，没有过半重新选主，接着leader向所有follower发送leader.UpToDate命令，follower接收后进行broadcast处理过程。

# 6.广播

1. leader针对client的请求，包装为一个proposal，并分配唯一ZXID，将该proposal放入outStandingProposals（concurrentHashMap<ZXID,proposal>）中
2. leader开始向所有follower发送该proposal，如果过半follower回复OK，leader将会提交该议案，并将其从outstandingProposals中删除，然后放到toBeApplied（concurrentLinkedQueue）中
3. leader对该议案进行提交，会向所有follower发送提交议案命令，同时自己也开始提交过程，会将请求的响应内容放到zookeeper的内存树中，然后跟新lastProcessedZXID，同时将该请求放入到committedLog（LinkedList）中，更新maxCommittedLog和minCommittedLog
4. leader向client回复，同时将该请求的议案从ToBeApplied队列中删除

# 7.持久化

* broadcast过程持久化

leader针对每次请求都会生成一个议案，然后向所有follower发送议案，follower收到议案后，所做的操作就是将该议案记录到事务日志中，默认每当满100000个，事务日志执行flush操作，同时开启一个新的文件来记录事务日志，同时还会执行内存树的快照

* leader shutdown过程持久化

一旦leader过半心跳检测失败，执行shutdown方法，然后对事务日志进行flush操作

# 8.恢复

* 事务快照恢复
  1. 会在事务快照文件目录下找到最近的100个快照文件，并排序，最新的在前面
  2. 对上述快照文件依次进行恢复和验证，一旦验证成功则退出，否则利用下一个快照文件进行恢复，恢复完成更新最新的lastProcessedZXID
* 事务日志恢复

1. 从事务日志文件目录下找到ZXID大于上述lastProcessedZXID的事务日志
2. 然后对上述事务日志进行遍历，应用到zookeeper内存树中，同时更新lastProcessedZXID
3. 同时将上述事务日志存储到committedLog中，并更新maxCommittedLog和minCommittedLog

可以看出来初始化恢复的时候，是会将所有最新事务日志作为已经commit的事务来处理，也就说这里可能会有部分事务日志实际并没有提交，而这里全部当做已提交来处理。