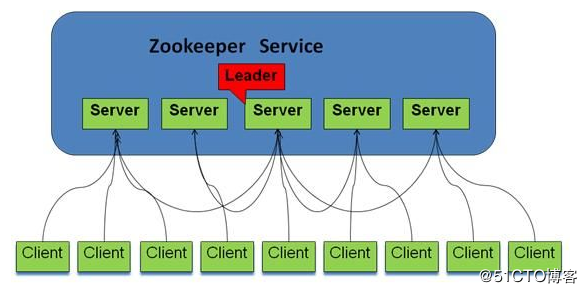
Apache Zookeeper

Apache Zookeeper是一个高性能的用于协调分布式应用程序的服务，其实一个集中式的服务，包括管理配置信息、命名服务、分布式锁、集群管理、选主与服务发现等。Zookeeper的设计模式是基于观察者模式设计的分布式服务管理框架，负责存储和管理数据，然后接受观察者的注册，一旦数据状态发生变化，通知观察者做相应的响应。

# 集群架构及概念

Apache Zookeeper本身是集群化的分布式服务，系统架构图如下所示：

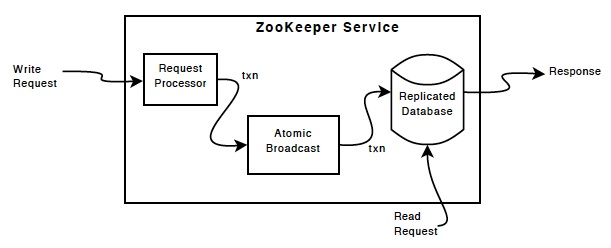


其各角色如下所示：

* 领导者（Leader），负责进行投票的发起和决议，更新系统状态
* 学习者（Learner）：跟随者（Follower），用户接受客户端请求并向客户端返回结果，在选主过程中参与投票；观察者（Observer），可以接收客户端连接，将写请求转发给Leader节点，但Observer不参加投票过程，只同步leader状态，Observer的目的是为了扩展系统，提高读取速度。
* 客户端（Client），请求的发起方

## 1.1 一致性协议

在Zookeeper的核心是原子广播，这个机制使用Zab协议保证了各个Server之间的同步，如下图所示：

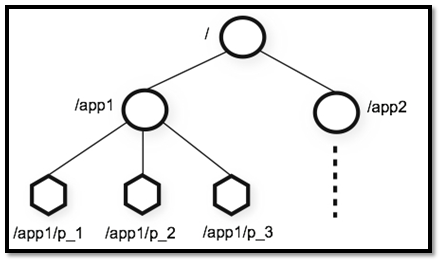


Zab协议有两种模式：

* 恢复模式（选主），当服务启动或者在领导者崩溃后，Zab进入恢复模式，当领导者被选出出来后，大多数Server完成和Leader的状态同步以后，保证Leader和Server具有相同的系统状态，恢复模式就结束了。
* 广播模式（同步），客户端发送更新请求，会生成全局唯一的递增编号（ZXID），Leader为每个Follower服务器创建一个队列，并按照FIFO的策略进行消息发送到Follower。Follower收到要处理的事务后，Follower以事务日志的形式写入服务器的磁盘中，并向Leader发送ACK响应。Leader收到超过一半的Follower的ACK响应后，向所有Follower服务器广播Commited消息，完成对事务的提交。

## **1.2相关概念**

1. 数据节点（ZNodes），zk数据模型中的最小数据单元，数据模型是一棵树，由斜杠/分割的路径名唯一标识：



数据节点可以存储数据内容及一系列属性信息，同时还可以挂载子节点，构成一个层次化的命名空间

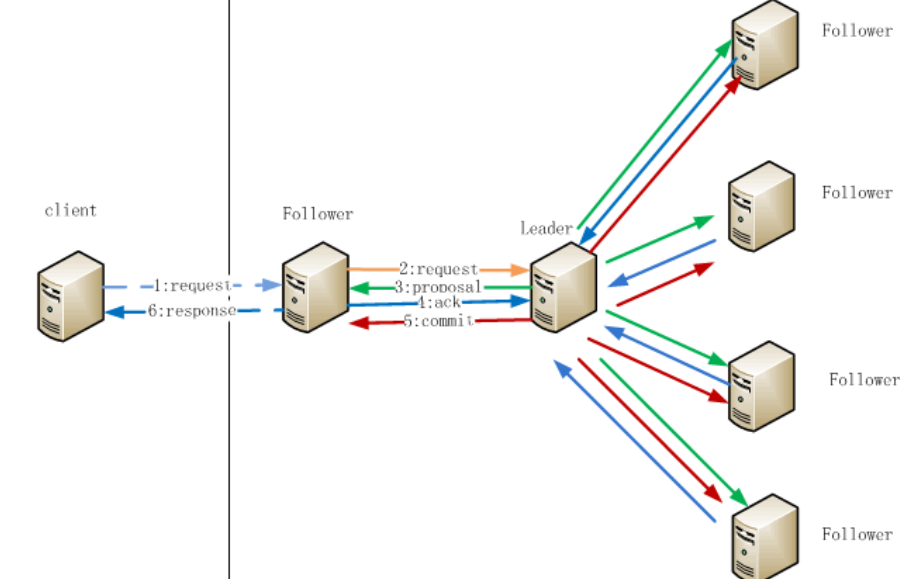
1. 会话（Session），客户端与服务器之间的会话，会话是通过客户端和服务器之间的TCP长连接来实现的。通过这个长连接，客户端能够使用心跳检测与服务器保持有效的会话，也能向服务器发送请求并接收响应，还可以接收服务器的Watcher事件通知。SessionTimeout，是会话超时时间，在这段时间内，客户端未与服务器发生任何沟通（心跳或请求），服务器会清楚该Session数据，客户端的TCP长连接将不可用，这种情况下，客户端需要重新实例化一个Zookeeper对象
2. 事务及ZXID，事务是指能够改变ZK服务器状态的操作，包括数据节点的创建与删除、数据节点内容的更新及客户端会话创建与失效等操作。对于每个事务操作，ZK都会为其分配全局唯一的事务ID(ZXID，64位数字)，高32位表示该事务发生的集群选举周期（每选举一次，值加1），低32为表示事务在当前选择周期内的递增次序（Leader每处理一次事务请求，值加1，发生一次Leader选择，低32为清0）

4）事务日志，所有的事务操作都是需要记录到日志文件中，可通过dataLogDir配置文件目录，文件是以写入第一条事务zkid为后缀，方便后续的定位查找。Zk会采取磁盘空间预分配的策略，来避免磁盘Seek频率，提高服务器处理能力，默认每条事务日志写入操作都会实时刷入磁盘。

1. 数据快照，数据存储的另一个核心运行机制，数据快照用来记录zk服务器上某一个时刻的全量内存数据内容，并将其写入到指定磁盘文件中，可以通过dataDir配置文件目录，可配置参数为snapCount，设置两次快照之间的事务操作个数。Zk节点记录完事务日志时，会统计判断是否需要做数据快照。
2. 过半，所谓过半是指大于集群集群数量的一半，即大于或者等于（n/2+1），此外的集群机器数量不包括Observer角色节点。Leader广播一个事务消息后，当收到半数以上的ack信息时，就认为集群中所有节点都收到了消息，然后leader就不需要等待剩余节点的ack，直接广播commit消息，提交事务。选举中的投票提议及数据同步时，也是如此，Leader不需要等到所有Learner节点的反馈，值要收到过半的反馈就可进行下一步的操作。

## **1.3 节点数据操作流程**

客户端数据操作流程如下所示：



针对客户端的写请求全部转交给Leader来处理，Leader需确保变更实时同步给所有Follower及Observer，具体执行如下：

1. 客户端向Follower发出一个写请求
2. Follower把请求发送给Leader
3. Leader接收到以后开始发起投票并通知Follower进行投票
4. Follower把投票结果发送给Leader
5. Leader将结果汇总后，如果需要写入，则开始写如，然后把写入响应通知给Leader，然后再Commit
6. Follower最后把请求结果返回给Client

# Leader选举

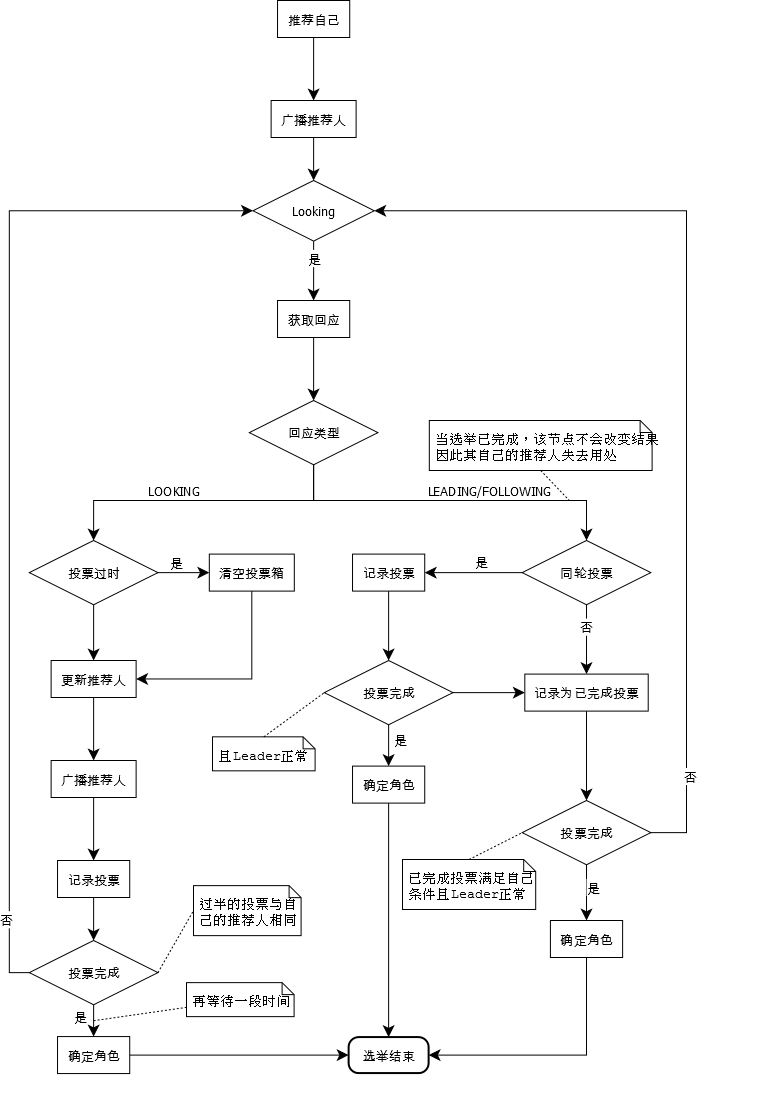
根据前文的介绍，Leader选举是保证分布式数据一致性的关键，当ZK集群中的一台服务器出现下面的两种情况：服务器初始化启动或则Leader在运行期间无法和大多数Follower保持连接，需要进入Leader选举。Zookeeper的Leader选举算法有两种：basic及fast paxos（系统默认），其核心的思想是一致的：每一个节点相当于一个公民，他们都有自己的推荐人，最开始的时候推荐自己。谁更适合成为Leader有一个简单的规则（配置中的SID够大，或者持有的数据够新zxid），每个选民都告诉其他选民自己的推荐人是谁，类似于宣传拉拢其他选民。每个选民发现有比自己更适合的人时就转而推荐更适合的人。最后大部分人意见一致时，就可以结束选举

* 1. **Leader选举的算法分析**

目前集群中支持的Leader选举算法为FastLeaderElection，其相关术语：

* SID，服务器ID，标识一台Zookeeper集群中的机器，配置为myid
* ZXID，事务ID，唯一标识服务器状态的变更，在某个时刻，集群每台机器的ZXID值不一定全都一致
* Vote，投票，集群机器发现自己无法检测到Leader的时候，就尝试投票
* Quorum，过半机器数，quorum = (n/2+1)

算法的执行流程图如下：



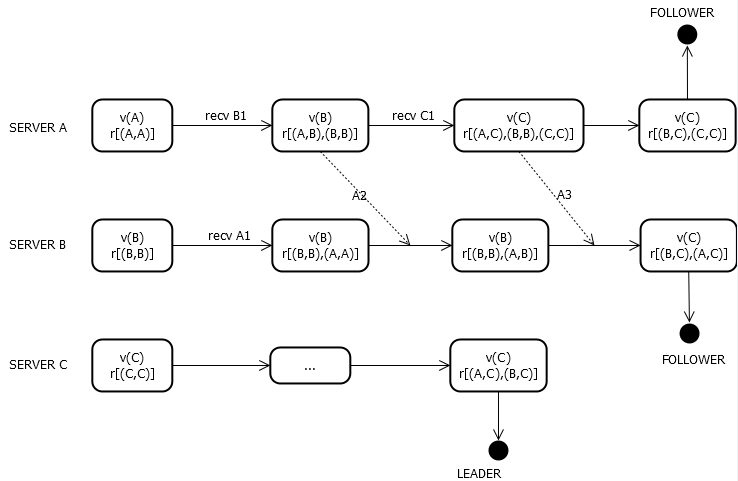
当服务器启动时，集群中已经有Leader的时候，当试图选举Leader时，会被告知当前服务器Leader的信息，仅需要和Leader机器建立连接，并进行状态同步即可。下面分析集群中没有Leader的情况下，进入Leader选举的过程。

1. 服务器启动后，从配置中读取对应的myId值，将自己作为Leader服务器开始投票，投票的基本元素<myid,ZXID>的形式来表示，然后将各自的投票发给集群中其他所有集群
2. 服务器接收来自各个服务器的投票，首先会判断该投票的有效性，包括检查是否是本轮投票，是否来自LOOKING状态的服务器
3. 在接收到来自其他服务器的投票后，针对每个投票，服务器都需要将别人的投票和自己的投票进行PK，PK的规则如下：

* 优先检查ZXID，比较大的服务器优先作为Leader
* ZXID相同，比较myid，值的服务器作为Leader服务器。

1. 统计投票，服务器统计所有投票后，判断是否已经有过半的服务器接收到相同的投票信息
2. 一旦确定Leader，每个服务器就会更新自己的状态，如果是Follower，就会更新为FOLLOWING，如果是Leader，那么变更为LEADING

下图是选举及状态转换的执行图：



V(A)表示当前推荐为A,r[]表示收到的投票集合。初始投票集合中只有自己的投票，代码中会将推荐自己。

在Zookeeper集群正常运行期间，一旦选出Leader，那么所有服务器的集群角色不会变化，Leader服务器一致作为集群的Leader，即使集群中有非Leader集群挂了或者新机器加入集群都不会影响Leader，但是一旦Leader所在的机器挂了，那么整个集群将暂时无法对外服务，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选举过程是一致的

* 变更状态，当Leader挂了以后，其余的非Observer服务器都将自己的服务器状态变更为LOOKING，然后开始进入Leader选举流程
* 每个Server发出一个投票（myid,ZXID），在运行期间ZXID的值可能不同
* 接收来自各个服务器的投票，处理规则与启动期间处理规则一致
* 统计投票，并改变服务器的状态
  1. **Leader选举相关数据结构**

FastLeaderElection算法并不复杂，但是在生产环境的产品中，需要定义很多实际的问题，选举相关类图：



http://iwinit.iteye.com/blog/1773531

1. 服务器状态

为了能够清楚地对Zookeeper集群每台机器的状态进行标识，在ServerState中列举了4中服务器状态：

* LOOKING，寻找Leader状态，处于该状态，其认为当前集群中没有Leader，需要进入Leader选举流程
* FOLLOWING，跟随者状态
* LEADING，领导者状态
* OBSERVERING，观察者状态

1. 投票数据结构

Leader的选举通过投票来实现，每个投票中包含最基本的信息：所推举服务器的SID和ZXID，其类结构：



|  |  |
| --- | --- |
| 属性 | 说明 |
| id | 被推举的Leader的SID值 |
| zxid | 被推举的Leader的事务ID |
| electionEpoch | 逻辑时钟，用来判断多个投票是否在同一轮选举周期中，该值在服务端是一个自增序列，每次进入新一轮的投票，都会对该值进行加1操作 |
| peerEpoch | 被推举的Leader的epoch |
| state | 当前服务器的状态 |

1. QuorumCnxManager

在Leader选举过程中，使用QuorumCnxManager来处理网络IO，每台服务器启动的时候，都会启动一个QuorumCnxManager负责各台服务器之间的底层Leader选举过程中的网络通信。

* 消息队列，其内部维护一系列队列，用于保存接收到的、待发送的消息，以及消息的发送器。这些队列按SID分组形成队列集合：
* recvQueue，消息接收队列，用于存放从其他服务器接收到的消息
* queueSendMap，消息发送队列，用于保存哪些待分组消息
* senderWorkerMap，发送器集合，负责消息发送
* lastMessageSend，最近发送过的消息
* 消息的接收和发送，消息的接收由RecvWorker来负责，不断从TCP连接读取消息，并其保存到recvQueue队列中。消息的发送由SendWorker来完成，不断从对应的消息发送队列获取一个消息发送即可。
  1. **选举算法的实现**

在Zookeeper中通过QuorumCnxManager来管理服务器之间的投票发送和接收，其执行流程如下所示：



基本流程是Election.lookForLeader方法的逻辑，当ZK服务器检测到当前服务器状态变为LOOKING时，就会触发Leader选举

1. 在FastLeaderElection中，有一个logicalclock属性，用于标识当前Leader的选举轮次，Zookeeper规定所有有效的投票必须在同一个轮次中，因此新一轮的投票时，logicalclock进行自增：

*logicalclock.incrementAndGet()*

1. 初始化选票，在开始新一轮的投票之前，要先初始化自己的选票，在初始化阶段每台服务器都会自己推举为Leader，在上文中已经介绍了Vote的数据结构，初始化代码如下：

*updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());*

1. 发送初始化选票，在完成选票初始化之后，发起第一次投票，将刚刚初始化的选票放入sendQueue队列中，由发送器WorkerSender负责发送出去

*private void sendNotifications() {*

*for (long sid : self.getCurrentAndNextConfigVoters()) {*

*QuorumVerifier qv = self.getQuorumVerifier();*

*ToSend notmsg = new ToSend(ToSend.mType.notification,*

*proposedLeader,*

*proposedZxid,*

*logicalclock.get(),*

*QuorumPeer.ServerState.LOOKING,*

*sid,*

*proposedEpoch, qv.toString().getBytes());*

*sendqueue.offer(notmsg);*

*}*

*}*

1. 接收外部投票，每台服务器不断从recvQueue队列中获取外部投票，如果无法获取到任何的外部投票，那么就会立即确认自己是否和其他服务器保持有效连接，没有建立连接则会马上连接

*Notification n = recvqueue.poll(notTimeout, TimeUnit.MILLISECONDS);*

1. 判断选举轮次，当发送完初始选票后，就要开始处理外部投票，在处理外部投票时会根据选举轮次进行不同的处理：

* 外部投票的选举轮次大于内部投票，立即更新自己的选举轮次，并且清空所有收到的图片，然后使用初始化投票来进行PK以确定是否变更内部投票，最终再由内部投票发送出去

*if (n.electionEpoch > logicalclock.get()) {*

*logicalclock.set(n.electionEpoch);*

*recvset.clear();*

*if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,*

*getInitId(), getInitLastLoggedZxid(), getPeerEpoch())) {*

*updateProposal(n.leader, n.zxid, n.peerEpoch);*

*} else {*

*updateProposal(getInitId(),getInitLastLoggedZxid(),getPeerEpoch());*

*}*

*sendNotifications();}*

* 外部投票的选举轮次小于内部投票，如果接受到的选票的选举轮次落后于服务器自身，那么Zookeeper会直接忽略外部投票

*else if (n.electionEpoch < logicalclock.get()) {*

*break;*

*}*

* 外部投票的选举轮次和内部投票一致，这也是绝大多数投票的场景，一致时开始选票PK

1. 选票PK，在收到来自其他服务器有效的外部投票后，就开始进行选票PK，FastLeaderElection.totalOrderPredicate方法的核心逻辑，其目的是为了确认当前服务器是否需要变更投票，主要从选举轮次，ZXID和SID三个因素去考虑，执行如下：

*protected boolean totalOrderPredicate( long newId, long newZxid, long newEpoch,*

*long curId, l ong curZxid, long curEpoch) {*

*if(self.getQuorumVerifier().getWeight(newId) == 0){*

*return false;*

*}*

*return ((newEpoch > curEpoch) ||*

*((newEpoch == curEpoch) &&*

*((newZxid > curZxid) || ((newZxid == curZxid) && (newId > curId)))));*

*}*

7）选票进行PK后，如果外部投票由于内部投票，变更投票的执行如下：

*updateProposal(n.leader, n.zxid, n.peerEpoch);*

使用外部投票信息来覆盖内部投票，然后将这个变更后的内部投票发送出去。

8）选票归档，无论是否进行投票变更，都会将受到的外部投票放入选票集合recvset中进行归档，执行如下：

*recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));*

recvset用于记录当前服务器在本轮次的Leader选举中受到的所有外部投票，按照服务器对应的SID来区分

1. 统计投票，完成选票归档之后，就可以开始统计投票，统计集群中是否有过半的服务器认可当前的内部投票。如果确定有过半的服务器任何该内部投票，则终止投票

*private boolean termPredicate(Map<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());*

*}*

*for (Map.Entry<Long, Vote> entry : votes.entrySet()) {*

*if (vote.equals(entry.getValue())) {*

*voteSet.addAck(entry.getKey());*

*}}*

*return voteSet.hasAllQuorums();}*

1. 更新服务器状态，统计投票后如果确定可以终止投票，那么就开始更新服务器状态。首先判断当前被过半服务器认可的投票所对应的Leader是否为自己，如果是自己，那么将自己的服务器状态更新为LEADING。如果不是被选出产生的Leader，那么根据具体情况来确定自己是FOLLOWING或者OBSERVING。状态由WorkerReiver在运行过中设置：

*int rstate = response.buffer.getInt();*

*......*

*QuorumPeer.ServerState ackstate = QuorumPeer.ServerState.LOOKING;*

*switch (rstate) {*

*case 0:*

*ackstate = QuorumPeer.ServerState.LOOKING;*

*break;*

*case 1:*

*ackstate = QuorumPeer.ServerState.FOLLOWING;*

*break;*

*case 2:*

*ackstate = QuorumPeer.ServerState.LEADING;*

*break;*

*case 3:*

*ackstate = QuorumPeer.ServerState.OBSERVING;*

*break;*

*default:*

*continue;*

以上步骤执行后，选举出了Leader。

# 选主后的数据同步

经过第二部分的介绍，Zookeeper集群完成了Leader的选举，Learner完成注册后进入数据同步环节，执行如下：

*case FOLLOWING:*

*setFollower(makeFollower(logFactory));*

*follower.followLeader();*

数据同步过程就是Leader服务器将在Leader上提交的事务请求同步给Learner服务器，注册过程如下所示：

*void addLearnerHandler(LearnerHandler learner) {*

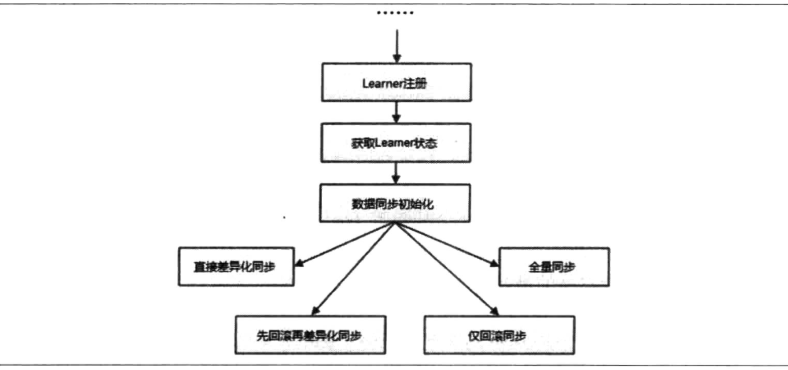
*synchronized (learners) {*

*learners.add(learner);*

*}*

*}*

Leader为所有Follower和observer创建LearnerHandler线程用于接收同步数据请求。在Leader端同步的大体过程如下：



1）获取Learner状态，注册的最后节点会向Leader服务器发送ACKEPOCH数据包，Leader会从整个数据包中解析出该Learner的currentEpoch和lastZxid

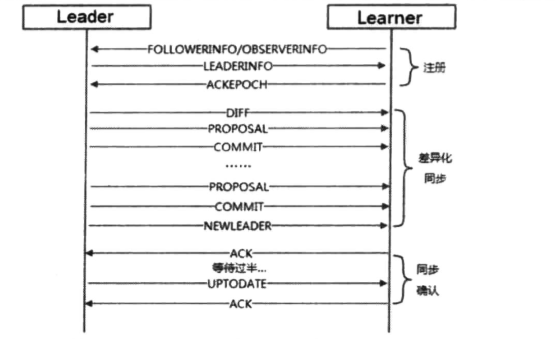
*QuorumPacket ackNewEpoch = new QuorumPacket(Leader.ACKEPOCH,*

*lastLoggedZxid, epochBytes, null);*

*writePacket(ackNewEpoch, true);*

1. 数据同步初始化，Leader服务器进行数据同步初始化，从Zookeeper的内在诉苦中提取出事务请求瑞鹰的提议缓存队列。集群数据同步通常分为：DIFF同步、先回滚再差异化同步（TRUNC+DIFF同步）、仅回滚同步（TRUNC同步）和全量同步（SNAP同步）。在初始化阶段，Leader服务器会优先初始化以全量同步的方式来同步数据。
2. 直接差异化同步（DIFF同步）,当peerLastZxid介于minCommitedLog和maxCommittedLog之间时，进行差异化同步的方式即可。

Leader服务器会首先向这个Learner发送DIFF命令，用于通知Learner进入差异化数据同步阶段，Leader服务器即将一些Proposal同步给自己，通过的过程如下所示：



4.

http://zookeeper.apache.org/doc/r3.5.0-alpha/zookeeperOver.html