一、Zookeeper 简介

Zookeeper 是一个以文件管理为核心的分布式协调服务。

Zookeeper 中有三种角色：Leader、Follower 和 Observer。一个 Zookeeper 集群中只有一个 Leader，为客户端提供读服务和写服务，而 Follower 和 Observer 都不能提供写服务。Leader 和 Follower 通过“Leader 选举”产生，而 Observer 并不参加这个过程。尽管 Follower 并不提供写服务，但是会参与“过半写成功”策略，所以还是会影响写性能，因此 Observer 可以在不影响写性能的情况下提升集群的读性能。

在客户端访问 Zookeeper 时，会与服务器建立一个 TCP 连接，客户端可以通过这个连接进行和服务器的会话，可以向 Zookeeper 发送请求和接收相应，可以接收来自服务器的 Watch 事件通知。当因客户端导致连接断开后，只要在 SessionTimeout 规定的时间内重新连上集群的任意一台服务器即可使之前创建的会话仍然有效。

Zookeeper 的结构是一个树形结构，Leader 就相当于根节点，Follower 相当于其叶节点。Zookeeper 的节点分为两类：持久节点和临时结点。持久结点一旦被创建，除非主动对其进行移除操作，其都会保留在 Zookeeper 上；而临时结点则是由客户端的会话绑定，一旦客户端会话失效，这个客户端创建的临时节点将会被移除。

Zookeeper 允许用户向指定节点上增加 Watcher，用于监测一些关心的节点。在节点被删除，更改或是增加了子节点等情况下会向用户进行反馈。

二、Zookeeper 的功能

1. 数据的发布和订阅

对于发布和订阅的系统，一般来讲有两种模式——服务端主动向用户推送的“推模式”和用户向服务端发出请求的“拉模式”。Zookeeper 中采用的是两种相结合的模式，用户可以选择自己想要关注的节点，在节点发生变更时，服务端将推送“推”给客户端一个通知，之后有客户端自行决定是否将新的数据从服务端上“拉”回来。

2. 分布式协调

Zookeeper 中有着 Watcher 和异步通知机制，能够实现分布式环境下不同机器甚至是不同系统之间的通知和协调。

3. Leader 选举

Leader 选举是 Zookeeper 中重要的功能之一。

在没有 Leader 的情况下，可能是刚开始也可能是在之前的 Leader 挂掉后，此时便要开始 Leader 选举。

如果是在刚开始，所有服务器除 Observer 均进入 Looking 状态；如果是在之前的 Leader 挂掉后，所有 Follower 会发现无法与 Leader 取得联系，然后就进入 Looking 状态。总而言之，进行选举前所有参加选举的服务器均为 Looking 状态。

首先进行第一轮投票，每个服务器向其他机器发送自己的“选票”，其中包含了 myid 和 ZXID 两个内容。第一轮所有的服务器都会“选”自己，即把自己的 myid 和 ZXID 发送出去。

然后每个服务器就会接收到来自其他服务器的选票，接下来进行判断，判断这个选票是否有效——是否来自 Looking 状态的服务器，是否是本轮的投票。

判断为有效后，接下来与自己的选票进行对比，遵循 ZXID 优先于 myid，大优先于小这两条规则，更新自己的选票并重新投票。

当有超过半数的选票统一时，就“选”出了 Leader，其他的服务器则变成 Follower，继续正常的工作。此时选举整个过程结束。

4. Master 选举

Master 选举也是 Zookeeper 中重要的功能之一。

作为服务器，一般来讲会有要求全天无休工作的需求，而此时若是出现了宕机，带来的损失可想而知。因此我们需要多个“相同的”节点来保证在一个宕机时其他的可以顶上来。

不过什么时候顶也是个难题，如果在之前的节点没死掉的情况下就顶掉也会带来问题，因此备用节点可以向主节点发送 ping 包，若没有应答则判断主节点死掉了。然而有时可能会因为网络原因而导致没能收到应答，此时顶掉主节点会带来同样的问题。

这时，Zookeeper 中的临时节点就发挥作用了。临时节点在经过了 SessionTimeout 设定的时间之后会自动被删除掉，所以我们可以让主节点在 Zookeeper 的某处建立一个临时节点，相当于“宣布”了“我是主节点”。在有其他的节点也想要当主节点时，会因为已经有这个临时节点存在而失败。此时这个节点并不是直接走掉，而是将这个节点订阅，在这个节点删除时，也就是之前的主节点死掉的时候。这时这些备用节点就可以来“抢”这个主节点的位置了，当然和之前一样，只要有一个节点抢到了这个位置，它就成为了主节点，其他的节点也就不能再抢了。

通过以上的行为便可以保证主节点存在且唯一，这样就能满足全天无休的工作了。

三、对于 Leader 选举的具体设计

首先，对于 Leader 选举来讲，Zookeeper 中包含了普通的 LeaderElection、FastLeaderElection 和 AuthFastLeaderElection。这三种 Leader 选举都会有相同的 lookForLeader 和 shutdown 两个函数，因此专门为 Election 设置了一个接口。

```
1.  //Election 的接口
2.  public interface Election {
3.  public Vote lookForLeader() throws InterruptedException;
4.  public void shutdown();
5.  }
```

接下来具体对 FastLeaderElection 进行分析。

首先在 FastLeaderElection 中找到了它的构造函数。

```
1.  public class FastLeaderElection implements Election {
2.      …
3.      public FastLeaderElection(QuorumPeer self, QuorumCnxManager manager){
4.          this.stop = false;
5.          this.manager = manager;
6.          starter(self, manager);
7.      }
8.      …
9.  }
```

其中 this.stop 置为 false 很好理解，若置为 true 相当于终止了这个 election。而 manager 则是用来管理连接的。FastLeaderElection 是通过 TCP 进行交流的，而这个 manager 会来管理这些链接。接下来便是 starter 了。

```
1.    private void starter(QuorumPeer self, QuorumCnxManager manager) {
2.        this.self = self;
3.        proposedLeader = -1;
4.        proposedZxid = -1;
5.
6.        sendqueue = new LinkedBlockingQueue<ToSend>();
7.        recvqueue = new LinkedBlockingQueue<Notification>();
8.        this.messenger = new Messenger(manager);
9.    }
```

首先，proposedLeader 和 proposedZxid 会被初始化，由于 myid（对应 leader）和 zxid 均非负，且最后要选取最大的，因此初始化为-1。接下来 starter 新建了 sendqueue 和 recvqueue，还新建了一个 messenger。sendqueue 和 recvqueue 分别对应了发送队列和接收队列。

接下来来看一下 lookForLeader 的部分。由于在主体的 Leader 选举过程中节点都是以 LOOKING 状态存在的，因此来着重分析 LOOKING 状态下的操作。

```
                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.sid);
                        break;
                    }
                    // If notification > current, replace and send messages out
                    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();
                    } 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;
                        } 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));
                        }

                        // don't care about the version if it's in LOOKING state
                        recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch,
n.peerEpoch));

                        voteSet = getVoteTracker(
                                recvset, new Vote(proposedLeader, proposedZxid,
                                        logicalclock.get(), proposedEpoch));

                        if (voteSet.hasAllQuorums()) {

                            // Verify if there is any change in the proposed leader
                            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
                             */
                            if (n == null) {
                                setPeerState(proposedLeader, voteSet);
```

```
                            Vote endVote = new Vote(proposedLeader,
                                    proposedZxid, logicalclock.get(),
                                    proposedEpoch);
                            leaveInstance(endVote);
                            return endVote;
                        }
                    }
                    break;
```

可以看到在整个 LOOKING 的状态下，有两个值很重要，n.electionEpoch 和 logicalclock，n.electionEpoch 相当于选举开始的时间，在最初它应当是比 logicalclock 大的，因此会进入第一个分支。在第一个分支中干了两件重要的事，一个是将 logicalclock 设置成与选举开始时间相同，即进入这个选举，然后通过 totalOrderPredicate 和 updateProposal 进行选票的初始化。在之后经过这里的时候会进入第三个分支，即通过 totalOrderPredicate 和 updateProposal 对选票进行刷新。至于第二个分支，理论上是不会出现这种情况的，因此里面是 debug 的 log。

这里还有个很巧妙的点，因为发生了选举会使得 n.electionEpoch 大于 logicalclock，通过这个方式很自然的将其初始化一次，而不用每次都对其“是否进行过初始化”进行专门的判断，这种判断一般来讲还需要再新增变量，而且可能还需要新的方法。

由于 totalOrderPredicate 和 updateProposal 被使用了多次，又因为虽然二者会同时使用，但要将相对功能高内聚低耦合，根据面向对象的设计原则便将其分别提出来单独作为一个函数。

```
    protected boolean totalOrderPredicate(long newId, long newZxid, long newEpoch, long curId,
long curZxid, long curEpoch) {
        LOG.debug("id: " + newId + ", proposed id: " + curId + ", zxid: 0x" +
                Long.toHexString(newZxid) + ", proposed zxid: 0x" +
Long.toHexString(curZxid));
        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.
         */

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

totalOrderPredicate 实际上只是实现了一个很简单的功能，它相当于一个信号用于输出“是否要更新现在的选票”。判断依据有三条：若已经在进行下一次投票则一定要更新选票、在通一次投票中新的 Zxid 如果比目前的大则更新、Zxid 也相同的情况下 id 若更大则更新。而在之前出现的两次调用中，在初始化时传入的“目前的选票”是初始化的选票，在之后更新时则是用的真正的目前的选票。虽然干了两件不太相同的事，但因为高度的相似性便将其抽象成一个方法，体现了面向对象的设计思想。

```
synchronized void updateProposal(long leader, long zxid, long epoch){
    if(LOG.isDebugEnabled()){
        LOG.debug("Updating proposal: " + leader + " (newleader), 0x"
                + Long.toHexString(zxid) + " (newzxid), " + proposedLeader
                + " (oldleader), 0x" + Long.toHexString(proposedZxid) + " (oldzxid)");
    }
    proposedLeader = leader;
    proposedZxid = zxid;
    proposedEpoch = epoch;
}
```

updateProposal 中也只是实现了一个非常简单的功能，只是将输入的三个值赋给了目前的选票。可以想到这一步除了在这里可能还会有很多的地方用到，因此将其也抽象成了一个方法。

反过来想一下，如果要设计能达到相同目的的方法，还有另一种设计方式——设计两个大的方法，分别为初始化和更新选票，每个方法中要先判断输入的选票和目前的选票的“大小”，然后再进行赋值。这样虽然能起到相同的效果，但是代码的简洁性就丢失了。在相似作用的两个方法中又存在不同的操作又导致了高耦合低内聚。

接下来来看一下选举的结束阶段。

```
                if (voteSet.hasAllQuorums()) {

                    // Verify if there is any change in the proposed leader
                    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
                     */
                    if (n == null) {
                        setPeerState(proposedLeader, voteSet);
                        Vote endVote = new Vote(proposedLeader,
                                proposedZxid, logicalclock.get(),
```

```
                    proposedEpoch);
            leaveInstance(endVote);
            return endVote;
        }
    }
```

首先会有一个 while 循环，这个循环用来判断每一次是否更新了目前的选票，如果更新了则跳出这个循环。由于整个 lookForLeader 是有一个大循环的，因此它会再一次的向外发送自己的选票，然后再一次的进入这个循环，直到所有的选票都变成了最大的，也就是最终选出来的 leader，这时这个循环会结束，此时 n 会为 null，进入了最后的结尾。

结尾中的 endVote 便是目前选出来的 leader 对应的选票，然后将这个选票返回，即这个值，对应着选出来的 leader，会作为整个 lookForLeader 的返回值。至此 leader 选举的主要部分完成。