

ZooKeeper 源码分析 (四) - Leader 选举

作者: [guobingwei](#)

原文链接: <https://ld246.com/article/1554032804309>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

前面几篇文章讲了服务器启动的流程，对于选举过程是一笔带过，这篇文章主要讲述Leader选举的细节。

Leader 选举是ZooKeeper中最重要技术之一，也是保证分布式数据一致性的关键所在。我们从选举算法概述、服务器启动Leader选举、服务器运行期间Leader选举三个方面探讨实现细节。

1. 选举算法概述

服务器启动期间Leader选举

选举的条件是集群中至少有两台机器。主要流程如下：

1. 每个server发出一个投票
2. 接收各个服务器的投票消息
3. 处理投票
4. 统计投票
5. 改变服务器状态

服务器运行期间的Leader选举

ZooKeeper集群在正常运行过程中，是不会进行leader选举过程的，只有当Leader机器挂掉之后，会进行新一轮的Leader选举。选举过程和**服务器启动期间Leader选举**是一致的。

1. 变更状态
2. 每个server发出一个投票
3. 接收来自各个服务器的投票
4. 处理投票
5. 统计投票
6. 改变服务器状态

QuorumPeer.OrderState定义了服务器的四种状态，分表是：

- **LOOKING**：寻找Leader状态，服务器处于该状态时，表示集群中没有leader，需要进入leader选举流程。
- **FOLLOWING**：跟随者状态，表明当前服务器角色是Follower
- **LEADER**：领导者状态，表明当前服务器角色是leader
- **OBSERVING**：观察者状态

2. 服务器启动Leader选举

服务器启动Leader选举只在集群模式启动时触发。根据上一篇文章

[ZooKeeper源码分析\(二\)—服务端启动之集群模式](#)，描述了项目启动过程，在执行QuorumPeer start()方法时会触发选举逻辑，下面详细看一下。

2.1. 启动入口

QuorumPeer start()方法中会执行startLeaderElection();方法，开始选举过程。

```

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

```

2.2. 生成投票

投票的数据结构如下：



各个字段的详细说明如下：

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，选举轮次
peerEpoch	被推举的Leader的epoch
state	当前服务器的状态
version	选举的版本

只有当server state为LOOKING状态是才触发选举过程。

```

if (getPeerState() == ServerState.LOOKING) {
    currentVote = new Vote(myid, getLastLoggedZxid(), getCurrentEpoch());
}

```

生成投票的规则为首次都选择自己作为Leader进行投票，传入myid,zxid,epoch值。分表代表机器编号

D, 事务ID, 当前的轮询次数。生成的投票结果再下面会用到, 会作为message发给其他机器。

2.3. 选择选举算法

根据`electionAlg`来决定实现哪种选举算法, 这个参数是在`zoo.cfg`配置文件中配置的。`electionAlg`的有0, 1, 2, 3四种。0是基于UDP实现的未进行安全认证的选举算法。1, 2也是基于UDP来实现, 只1是非授权模式, 2是授权模式。3为TCP版本的`FastLeaderElection`。3.4.0之后的版本都推荐使用`FastLeaderElection`模式, 下面主要讲这种实现。

2.4. QuorumCnxManager介绍

这个组件是基于TCP的用于leader选举的连接管理器。处理选举过程中的网络通信。他为每两个机器护着一个连接, 组件要解决的问题是如何保证两台机器之前只建立一个连接并能正确的通过网络进行信。如果有两台机器同时发起连接请求, 连接管理器会使用简单的`tie-breaking`机制来决定要删除哪连接。对每个连接对, 管理器维护着一个消息队列, 每次有新的消息, 都追加到队列尾部。队列是通监听器持有的。

连接管理器通过`AtomicReference`来持有, 为了保证支持跨线程的更新, 线程安全

```
private AtomicReference<QuorumCnxManager> qcmRef = new AtomicReference<>();
```

2.5. 初始化连接管理器

通过原子操作`getAndSet`来初始化连接管理器, 如果发现已经初始化了管理器, 则执行管理器的销毁作。重新进行选举过程。为什么要销毁? 这个管理器可能是上个Leader存活期间创建的, 保留了上次举的一些信息, 现在要重新选举, 要清除这些信息

```
QuorumCnxManager qcm = createCnxnManager();
QuorumCnxManager oldQcm = qcmRef.getAndSet(qcm);
if (oldQcm != null) {
    LOG.warn("Clobbering already-set QuorumCnxManager (restarting leader election?");
    oldQcm.halt();
}
```

`oldQcm.halt()`执行销毁动作, 中断线程, 重置连接信息

2.5.1. 初始化监听器

如果连接管理器创建成功, 需要注册一个监听器, 监听器里维护着两个线程, 进行消息发送和接收

```
QuorumCnxManager.Listener listener = qcm.listener;
if (listener != null) {
    listener.start();
    FastLeaderElection fle = new FastLeaderElection(this, qcm);
    fle.start();
    le = fle;
} else {
    LOG.error("Null listener when initializing cnx manager");
}
```

这个Listener继承自`ZooKeeperThread`, 有三种socket实现, 默认、统一协议socket、基于SSL协议socket

```

if (self.shouldUsePortUnification()) {
    LOG.info("Creating TLS-enabled quorum server socket");
    ss = new UnifiedServerSocket(self.getX509Util(), true);
} else if (self.isSslQuorum()) {
    LOG.info("Creating TLS-only quorum server socket");
    ss = new UnifiedServerSocket(self.getX509Util(), false);
} else {
    ss = new ServerSocket();
}

```

2.5.2. 注册端口，监听事件

如果SASL安全认证可用时，需要异步处理连接过程，因为SASL认证需要一定的耗时，同步操作的话阻塞其他的连接请求。

```

if (quorumSaslAuthEnabled) {
    receiveConnectionAsync(client);
} else {
    receiveConnection(client);
}

```

`receiveConnectionAsync`会把逻辑放在线程池里异步去执行。

```

connectionExecutor.execute(new QuorumConnectionReceiverThread(sock));
connectionThreadCnt.incrementAndGet();

```

同步执行的逻辑在`receiveConnection`里，如果当前服务器被选为leader，就会放弃当前连接的请求否则会初始化`SendWorker`，`RecvWorker`，接收请求并处理。

2.6. 初始化选票

`SendWorker`主要职责是发送消息给集群中其他机器，在其他机器存活的情况下，他会第一时间发送求。如果队列中没有消息发送，他会发送lastMessage，确认集群中其他机器都收到了消息。

`RecvWorker`用来接收消息，监听socket端口。如果channel关闭，RecvWorker会从线程池中移除。

```

//If wins the challenge, then close the new connection.
if (sid < self.getId()) {
    /*
     * This replica might still believe that the connection to sid is
     * up, so we have to shut down the workers before trying to open a
     * new connection.
     */
    SendWorker sw = senderWorkerMap.get(sid);
    if (sw != null) {
        sw.finish();
    }

    /*
     * Now we start a new connection
     */
    LOG.debug("Create new connection to server: {}", sid);
    closeSocket(sock);
}

```

```

    if (electionAddr != null) {
        connectOne(sid, electionAddr);
    } else {
        connectOne(sid);
    }
} else {
    // Otherwise start worker threads to receive data.
    SendWorker sw = new SendWorker(sock, sid);
    RecvWorker rw = new RecvWorker(sock, din, sid, sw);
    sw.setRecv(rw);

    SendWorker vsw = senderWorkerMap.get(sid);

    if (vsw != null) {
        vsw.finish();
    }

    senderWorkerMap.put(sid, sw);

    queueSendMap.putIfAbsent(sid,
        new ArrayBlockingQueue<ByteBuffer>(SEND_CAPACITY));

    sw.start();
    rw.start();
}

```

`WorkerSender`的处理逻辑在`private void process(ToSend m)`中，它根据message type来判断消息类型进行不同的处理。用switch case来处理，这里写的又有点low..hahh。总共有四类不同的消息类型

crequest: 发起选举的请求信息

challenge: 选举信息

notification: 通知消息

ack: 响应消息

```

static enum mType {
    crequest, challenge, notification, ack
}

```

```

ToSend(mType type, long tag, long leader, long zxid, long epoch,
    ServerState state, InetAddress addr) {

```

```

    switch (type) {
    case crequest:
        this.type = 0;
        this.tag = tag;
        this.leader = leader;
        this.zxid = zxid;
        this.epoch = epoch;
        this.state = state;
        this.addr = addr;

```

```

        break;
    case challenge:
        this.type = 1;
        this.tag = tag;
        this.leader = leader;
        this.zxid = zxid;
        this.epoch = epoch;
        this.state = state;
        this.addr = addr;

        break;
    case notification:
        this.type = 2;
        this.leader = leader;
        this.zxid = zxid;
        this.epoch = epoch;
        this.state = QuorumPeer.ServerState.LOOKING;
        this.tag = tag;
        this.addr = addr;

        break;
    case ack:
        this.type = 3;
        this.tag = tag;
        this.leader = leader;
        this.zxid = zxid;
        this.epoch = epoch;
        this.state = state;
        this.addr = addr;

        break;
    default:
        break;
    }
}

```

case 0:构造一个选举开始请求给其他机器。

```

/*
 * Building challenge request packet to send
 */
requestBuffer.clear();
requestBuffer.putInt(ToSend.mType.crequest.ordinal());
requestBuffer.putLong(m.tag);
requestBuffer.putInt(m.state.ordinal());
zeroes = new byte[32];
requestBuffer.put(zeroes);
requestPacket.setLength(48);
requestPacket.setSocketAddress(m.addr);

if (challengeMap.get(m.tag) == null) {
    mySocket.send(requestPacket);
}

```

case 1: 发送选举信息给其他机器

```
/*
 * Building challenge packet to send
 */
long newChallenge;
ConcurrentHashMap<Long, Long> tmpMap = addrChallengeMap.get(m.addr);
if(tmpMap != null){
    Long tmpLong = tmpMap.get(m.tag);
    if (tmpLong != null) {
        newChallenge = tmpLong;
    } else {
        newChallenge = genChallenge();
    }

    tmpMap.put(m.tag, newChallenge);

    requestBuffer.clear();
    requestBuffer.putInt(ToSend.mType.challenge.ordinal());
    requestBuffer.putLong(m.tag);
    requestBuffer.putInt(m.state.ordinal());
    requestBuffer.putLong(newChallenge);
    zeroes = new byte[24];
    requestBuffer.put(zeroes);
    requestPacket.setLength(48);
    requestPacket.setSocketAddress(m.addr);
    mySocket.send(requestPacket);
}
```

case 2: 构造通知消息去发送, 有重试机制, 最多重试`maxAttempts`次

case 3: 发送ack消息

```
case 3:
    requestBuffer.clear();
    requestBuffer.putInt(m.type);
    requestBuffer.putLong(m.tag);
    requestBuffer.putInt(m.state.ordinal());
    requestBuffer.putLong(m.leader);
    requestBuffer.putLong(m.zxid);
    requestBuffer.putLong(m.epoch);
    requestPacket.setLength(48);
    try {
        requestPacket.setSocketAddress(m.addr);
    } catch (IllegalArgumentException e) {
    }
    try {
        mySocket.send(requestPacket);
    } catch (IOException e) {
        LOG.warn("Exception while sending ack: ", e);
    }
    break;
```

2.7 接收外部投票

`WorkerReceiver`也是根据消息类型来进行处理的。

当message type = 0时，表示其他机器发起了选举的请求，当前机器也会生成内部投票消息去发送。每台服务器都会不断的从recvqueue队列中获取外部投票，如果服务器无法获取任何外部投票时，会即确认自己是否和集群中其他服务器保持着有效连接，如果没有建立连接，那么会马上建立连接，如已经建立连接，那么就再次发送当前的内部投票

case 0:

```
// Receive challenge request
ToSend c = new ToSend(ToSend.mType.challenge, tag,
    current.getId(), current.getZxid(),
    logicalclock.get(), self.getPeerState(),
    (InetSocketAddress) responsePacket.getSocketAddress());
sendqueue.offer(c);
break;
```

type = 1时，接收其他机器发来的选举信息，保存到本地。是通过`challengeMap`来保存的，是个`ConcurrentHashMap`

case 1:

```
// Receive challenge and store somewhere else
long challenge = responseBuffer.getLong();
saveChallenge(tag, challenge);
break;
```

2.8 判断选举轮次

type = 2时，接收通知消息，会先判断收到消息的选举轮次

- 如果通知消息的选举轮次比本身的高，则更新自己的选举轮次，并接收通知中的选举信息作为自己选举信息进行发送。然后把通知消息放入recvqueue中，生成的自身的选举消息放入sendqueue中。
- 如果外部投票的选举轮次小于内部投票，那么会忽略该外部投票，不做任何处理。
- 外部投票和内部投票选举轮次一致，则开始进行选票PK。

```
if ((myMsg.lastEpoch <= n.epoch)
    && ((n.zxid > myMsg.lastProposedZxid)
    || ((n.zxid == myMsg.lastProposedZxid)
    && (n.leader > myMsg.lastProposedLeader)))) {
    myMsg.lastProposedZxid = n.zxid;
    myMsg.lastProposedLeader = n.leader;
    myMsg.lastEpoch = n.epoch;
}
```

```
recvqueue.offer(n);
ToSend a = new ToSend(ToSend.mType.ack, tag,
    current.getId(), current.getZxid(),
    logicalclock.get(), self.getPeerState(),
    (InetSocketAddress) responsePacket
        .getSocketAddress());
sendqueue.offer(a);
```

2.9 选票PK

`totalOrderPredicate`会判断一个外部选票是否大于内部选票。判断的逻辑为：

- 外部选票的选举轮次更高
- 外部选票的选举轮次跟内部一样，但是zxid更高
- 外部选票的选举轮次跟内部一样，zxid也相同，但是sid更高

这三种情况都会是外部选票胜出。

```
protected boolean totalOrderPredicate(long newId, long newZxid, long newEpoch, long curId,
long curZxid, long curEpoch) {
    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)))));
}
```

2.10 变更投票

通过选票PK，确定了外部选票优于内部投票，那么就进行选票变更。使用外部投票的选票信息覆盖内投票。变更完成后，再次将这个变更后的内部投票发送出去。

```
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;
}
```

发送通知：

```
/**
 * Send notifications to all peers upon a change in our vote
 */
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());
    if(LOG.isDebugEnabled()){
        LOG.debug("Sending Notification: " + proposedLeader + " (n.leader), 0x" +
            Long.toHexString(proposedZxid) + " (n.zxid), 0x" + Long.toHexString(logicalclock.
et()) +
            " (n.round), " + sid + " (recipient), " + self.getId() +
            " (myid), 0x" + Long.toHexString(proposedEpoch) + " (n.peerEpoch)");
    }
    sendqueue.offer(notmsg);
}
}
}

```

2.11 选票归档

无论是否进行了投票变更，都会将收到的外部投票放入选票集合recvset中进行归档，recvset用于记当前服务器在本轮次的选举中收到的所有外部投票

```

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;
        }
    }
}
}

```

2.12 统计投票

投票统计的过程就是为了统计集群中是否已经有了过半的服务器认可了当前的内部投票，如果是，则止投票

2.13 更新服务器状态

统计投票后，如果已经确定可以终止投票，那么就更新服务器状态。先判断投票结果的Leader是否是己，如果是的话，就会将自己的服务器状态更新为Leading，如果不是自己的话，根据情况来确定自己是FOLLOWING还是OBSERVING

3. 服务器运行期间Leader选举

服务器启动完成之后，QuorumPeer的后台线程一直运行着，在run()方法中，有个循环逻辑不断的测服务器状态，如果服务器状态变为LOOKING时，会触发lookForLeader逻辑，开始进行选举过程。

3.1. LOOKING状态

`zoo.cfg`配置文件中可以配置`readonlymode.enabled`的值，如果启用只读模式时，即使当服务器从群中分隔出去，客户端仍可以从服务器读取值，只是不能进行更新操作，也无法感知其他客户端的变。

如果`readonlymode.enabled`为`true`，则会在休眠一定时间后启动只读服务器。

然后再执行`startLeaderElection()`方法，开始leader选举，并发送选票信息。选举过程之前已经讲过

如果`readonlymode.enabled`为`false`，直接进行leader选举。

case LOOKING:

```
// 允许启动只读模式
if (Boolean.getBoolean("readonlymode.enabled")) {
    final ReadOnlyZooKeeperServer roZk =
        new ReadOnlyZooKeeperServer(logFactory, this, this.zkDb);

    Thread roZkMgr = new Thread() {
        public void run() {
            try {
                sleep(Math.max(2000, tickTime));
                if (ServerState.LOOKING.equals(getPeerState())) {
                    roZk.startup();
                }
            } catch (InterruptedException e) {
                LOG.info("Interrupted while attempting to start ReadOnlyZooKeeperServer, not s
                arted");
            } catch (Exception e) {
                LOG.error("FAILED to start ReadOnlyZooKeeperServer", e);
            }
        }
    };
    try {
        // 启动只读服务器
        roZkMgr.start();
        reconfigFlagClear();
        if (shuttingDownLE) {
            shuttingDownLE = false;
            // 开始选举
            startLeaderElection();
        }
        // 生成选票
        setCurrentVote(makeLEStrategy().lookForLeader());
    } catch (Exception e) {
        LOG.warn("Unexpected exception", e);
        setPeerState(ServerState.LOOKING);
    } finally {
        roZkMgr.interrupt();
        roZk.shutdown();
    }
    // 只读模式禁用
} else {
    try {
```

```

    reconfigFlagClear();
    if (shuttingDownLE) {
        shuttingDownLE = false;
        // 开始选举
        startLeaderElection();
    }
    // 生成选票
    setCurrentVote(makeLEStrategy().lookForLeader());
} catch (Exception e) {
    LOG.warn("Unexpected exception", e);
    setPeerState(ServerState.LOOKING);
}
}
break;

```

3.2. OBSERVING状态

当服务器状态为OBSERVING时，设置observer信息，充当leader的一个观察者，其观察集群状态的变化并将这些变化同步过来。对于非事务请求可以独立处理，对于事务请求，则转发给leader进行处理他不参与任何形式的投票。

```

case OBSERVING:
    try {
        // 设置observer信息
        setObserver(makeObserver(logFactory));
        // 实现observer leader的主要逻辑
        observer.observeLeader();
    } catch (Exception e) {
        LOG.warn("Unexpected exception", e);
    } finally {
        observer.shutdown();
        setObserver(null);
        updateServerState();

        // Add delay jitter before we switch to LOOKING
        // state to reduce the load of ObserverMaster
        if (isRunning()) {
            Observer.waitForReconnectDelay();
        }
    }
}
break;

```

3.3. FOLLOWING状态

当服务器状态是FOLLOWING时，会把当前服务器设置为Follower，作为ZooKeeper集群状态的跟者。它的主要工作有三个：

- 处理客户端非事务请求，转发事务请求给Leader服务器
- 参与事务请求proposal的投票
- 参与Leader选举投票

```

case FOLLOWING:

```

```
try {
    LOG.info("FOLLOWING");
    setFollower(makeFollower(logFactory));
    follower.followLeader();
} catch (Exception e) {
    LOG.warn("Unexpected exception", e);
} finally {
    follower.shutdown();
    setFollower(null);
    updateServerState();
}
break;
```

4. 流程总结

选举实现的主要组件协作关系如图所示：

