ZooKeeper

寇伊　2015K8009915002

一、 ZooKeeper 的简述

从命名的角度讲，ZooKeeper，顾名思义，动物园的管理员，结合以往了解到的 IT 届各种项目的命名规则和以动物作为吉祥物的癖好，大数据处理器 Hadoop 是一只黄色的大象；负责数据仓库的 Hive 是虚拟的蜂巢；管理 web 容器的 tomcat 是一直雄猫；数据分析项目 Apache Pig 则是一头猪；那么 ZooKeeper 就非常形象的成为了所有系统的管理者，因此它是一个负责分布式协调的项目。

官方给出的解释是 ZooKeeper 分布式服务框架是 Apache Hadoop 的一个子项目，它主要用于解决分布式应用中的常见数据管理问题，比如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理。ZooKeeper 通过封装服务以达到将接口简单化，使性能高效化以及使功能稳定化从而简化分布式协调及管理的难度，提供高性能的分布式服务。

二、 ZooKeeper 中主要介绍的功能——Leader 选举

Leader 选举是保证分布式数据一致性的关键所在。当 Zookeeper 集群中的一台服务器出现以下两种情况之一时，需要进入 Leader 选举。

⑴ 服务器初始化启动。

⑵ 服务器运行期间无法和 Leader 保持连接。

三、 Leader 选举的算法介绍

在 3.4.0 后的 Zookeeper 的版本只保留了 TCP 版本的 FastLeaderElection 选举算法。当一台机器进入 Leader 选举时，当前集群可能会处于以下两种状态：

集群中已经存在 Leader；

集群中不存在 Leader；

对于集群中已经存在 Leader 而言，此种情况一般都是某台机器启动得较晚，在其启动之前，集群已经在正常工作，对这种情况，该机器试图去选举 Leader 时，会被告知当前服务器的 Leader 信息，对于该机器而言，仅仅需要和 Leader 机器建立起连接，并进行状态同步即可。而在集群中不存在 Leader 情况下则会进行 Leader 选举，其步骤如下：

⑴ 第一次投票：

无论哪种导致进行 Leader 选举，集群的所有机器都处于试图选举出一个 Leader 的状态，即 LOOKING 状态，LOOKING 机器会向所有其他机器发送消息，该消息称为投票。投票中包含了 SID（服务器的唯一标识）和 ZXID（事务 ID），(SID, ZXID)形式来标识一次投票信息。假定 Zookeeper 由 5 台机器组成，SID 分别为 1、2、3、4、5，ZXID 分别为 9、9、9、8、8，并且此时 SID 为 2 的机器是 Leader 机器，某一时刻，1、2 所在机器出现故障，因此集群开始进行 Leader 选举。在第一次投票时，每台机器都会将自己作为投票对象，于是 SID 为 3、4、5 的机器投票情况分别为(3, 9)，(4, 8)， (5, 8)。

⑵ 变更投票：

每台机器发出投票后，也会收到其他机器的投票，每台机器会根据一定规则来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票，这个规则也是整个 Leader 选举算法的核心所在，其中涉及到四个变量：

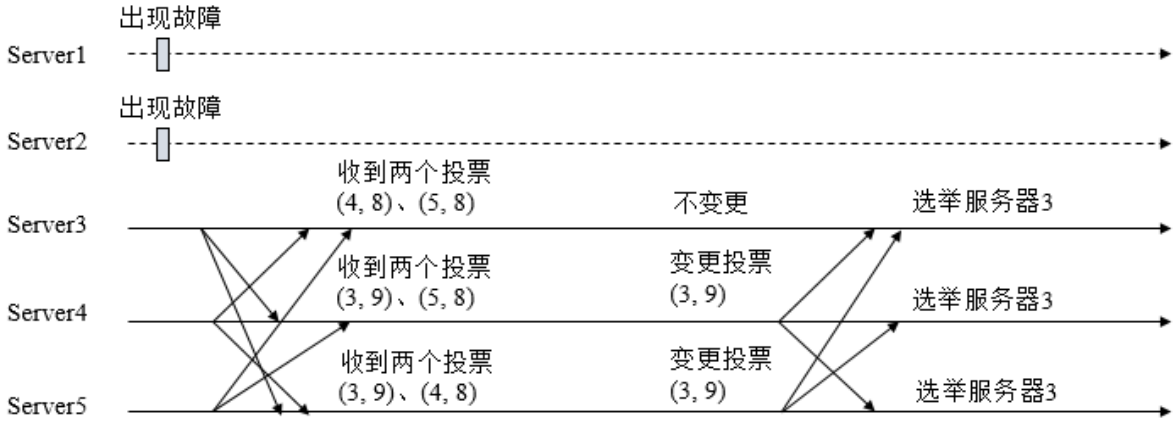
- vote_sid：接收到的投票中所推举 Leader 服务器的 SID。
- vote_zxid：接收到的投票中所推举 Leader 服务器的 ZXID。
- self_sid：当前服务器自己的 SID。
- self_zxid：当前服务器自己的 ZXID。

每次对收到的投票的处理，都是对(vote_sid, vote_zxid)和(self_sid, self_zxid)对比的过程，依据如下规则：

- 规则一：如果 vote_zxid 大于 self_zxid，就认可当前收到的投票，并再次将该投票发送出去。
- 规则二：如果 vote_zxid 小于 self_zxid，那么坚持自己的投票，不做任何变更。

- 规则三：如果 vote_zxid 等于 self_zxid，那么就对比两者的 SID，如果 vote_sid 大于 self_sid，那么就认可当前收到的投票，并再次将该投票发送出去。
- 规则四：如果 vote_zxid 等于 self_zxid，并且 vote_sid 小于 self_sid，那么坚持自己的投票，不做任何变更。

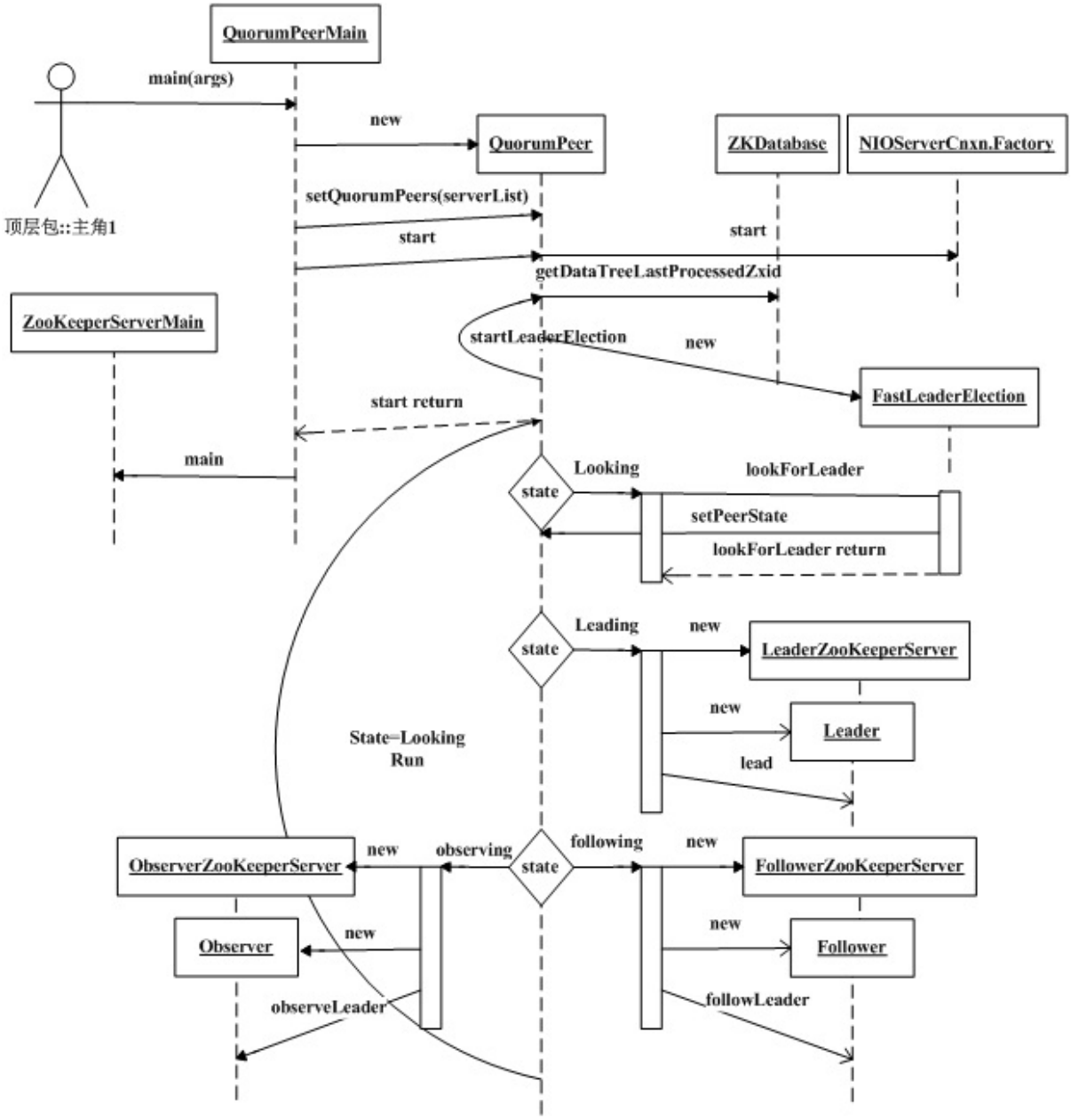
举个例子



⑶ 确定 Leader：

经过第二轮投票后，集群中的每台机器都会再次接收到其他机器的投票，然后开始统计投票，如果一台机器收到了超过半数的相同投票，那么这个投票对应的 SID 机器即为 Leader。此时 Server3 将成为 Leader。

四、 ZooKeeper 集群的启动过程



这张图非常清晰地描述了 ZooKeeper 整个集群从预启动到初始化再到 Leader 选举的过程。

首先是预启动过程，QuorumPeerMain 作为启动类，在读取 zoo.cfg 配置文件之后创建并启动历史文件清理器并且判断当前是集群模式还是单机模式。

接下来进入到初始化，创建并初始化 ServerCnxnFactory 线程，随后初始化 NIO 服务器，创建数据管理器。创建 QuorumPeer 实例，Quorum 作为集群模式下特有的对象是 ZooKeeper 服务器实力的托管者，从集群层面来看 QuorumPeer 代表着 ZooKeeper 集群中的一台服务器，在运行期间他会不断检查当前服务器实例运行的状态，然后根据情况进行 leader 选举。紧接着创建内存数据库 ZKDatabase，并初始化

QuorumPeer 注册核心组件包括 FileTxnSnapLog、ServerCnxnFactory、ZKDatabase 并配置参数。

随后，启动 ServerCnxnFactory 主线程，进行 Leader 选举。

五、 ZooKeeper 启动类与集群初始化

ZooKeeper 的启动类是 org.apache.zookeeper.server.quorum.QuorumPeerMain，其启动时传入配置文件 zoo.cfg 的路径。QuorumPeerMain 解析各项配置，如果发现 server 列表只有一个，那么直接通过 ZooKeeperServerMain 来启动单机版的 Server；如果有多个，那么读取 server 列表和 myid 文件，启动 QuorumPeer 线程。

```
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();

    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);
    }
}
```

对于 QuorumPeer 线程，每个 QuorumPeer 线程启动之前都会先启动一个 cnxnFactory 线程，作为 NIO server(Non-blocking Server)接受客户端请求。

```
ServerCnxnFactory cnxnFactory = null;
ServerCnxnFactory secureCnxnFactory = null;
```

QuorumPeer 线程启动

```
protected QuorumPeer quorumPeer;
```

```
quorumPeer = getQuorumPeer();
quorumPeer.setRootMetricsContext(metricsProvider.getRootContext());
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(), false);
```

```
/**
 * This class manages the quorum protocol. There are three states this server
 * can be in:
 * <ol>
 * <li>Leader election - each server will elect a leader (proposing itself as a
 * leader initially).</li>
 * <li>Follower - the server will synchronize with the leader and replicate any
 * transactions.</li>
 * <li>Leader - the server will process requests and forward them to followers.
 * A majority of followers must log the request before it can be accepted.
 * </ol>
```

```
* This class will setup a datagram socket that will always respond with its
* view of the current leader. The response will take the form of:
*
* <pre>
* int xid;
*
* long myid;
*
* long leader_id;
*
* long leader_zxid;
* </pre>
*
* The request for the current leader will consist solely of an xid: int xid;
*/
```

启动线程后，首先做 Leader election。一个 QuorumPeer 线程代表一个 ZooKeeper 节点，或者说一个 ZooKeeper 进程。

QuorumPeer 共有 4 个状态：LOOKING, FOLLOWING, LEADING, OBSERVING; 启动时初始状态是 LOOKING，表示正在寻找确定 leader 中。

```
public enum ServerState {
    LOOKING, FOLLOWING, LEADING, OBSERVING;
}
```

Leader election 的默认算法是基于 TCP 实现的 fast Paxo 算法，由 FastLeaderElection 实现。

```
@SuppressWarnings("deprecation")
protected Election createElectionAlgorithm(int electionAlgorithm){
    Election le=null;

    //TODO: use a factory rather than a switch
    switch (electionAlgorithm) {
    case 1:
        le = new AuthFastLeaderElection(this);
        break;
    case 2:
        le = new AuthFastLeaderElection(this, true);
        break;
    case 3:
        qcm = createCnxnManager();
        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");
        }
        break;
    default:
        assert false;
    }
```

QuorumPeer 线程调用 FastLeaderElection.lookForLeader 选择 leader（之后会详细说），该方法会在确定 leader 之后改变 QuorumPeer 的状态为 LEADING, FOLLOWING 或 OBSERVING。

QuorumPeer 根据 Leader election 确定的这 3 个状态之一对应创建 LeaderZooKeeperServer、FollowerZooKeeperServer、ObserverZooKeeperServer 和 Leader、Follower、Observer 对象，并调用各自的 lead、followLeader、observeLeader 方法。

```
 protected Leader makeLeader(FileTxnSnapLog logFactory) throws IOException, X509Exception {
     return new Leader(this, new LeaderZooKeeperServer(logFactory, this, this.zkDb));
 }

protected Follower makeFollower(FileTxnSnapLog logFactory) throws IOException {
    return new Follower(this, new FollowerZooKeeperServer(logFactory, this, this.zkDb));
}

protected Observer makeObserver(FileTxnSnapLog logFactory) throws IOException {
    return new Observer(this, new ObserverZooKeeperServer(logFactory, this, this.zkDb));
}
```

```
public Follower follower;
public Leader leader;
public Observer observer;

protected Follower makeFollower(FileTxnSnapLog logFactory) throws IOException {
    return new Follower(this, new FollowerZooKeeperServer(logFactory, this, this.zkDb));
}

protected Leader makeLeader(FileTxnSnapLog logFactory) throws IOException, X509Exception {
    return new Leader(this, new LeaderZooKeeperServer(logFactory, this, this.zkDb));
}

protected Observer makeObserver(FileTxnSnapLog logFactory) throws IOException {
    return new Observer(this, new ObserverZooKeeperServer(logFactory, this, this.zkDb));
}
```

```
while (running) {
    switch (getPeerState()) {
    case LOOKING:
```

```
case LEADING:
    LOG.info("LEADING");
    try {
        setLeader(makeLeader(logFactory));
        leader.lead();
        setLeader(null);
    } catch (Exception e) {
        LOG.warn("Unexpected exception",e);
    } finally {
        if (leader != null) {
            leader.shutdown("Forcing shutdown");
            setLeader(null);
        }
        updateServerState();
    }
    break;
```

```
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;
```

```
case OBSERVING:
    try {
        LOG.info("OBSERVING");
        setObserver(makeObserver(logFactory));
        observer.observeLeader();
    } catch (Exception e) {
        LOG.warn("Unexpected exception",e );
    } finally {
        observer.shutdown();
        setObserver(null);
       updateServerState();
    }
    break;
```

单机版 Server、Leader、Follower、Observer 分别对应 ZooKeeperServer、LeaderZooKeeperServer、FollowerZooKeeperServer、ObserverZooKeeperServer。4 种 Server 共享 Processor 处理器，各自将某几个 Processor 按顺序组合为一个 Processor 链。在每个 Server 中请求总是从第一个 Processor 开始处理，处理完交给下一个，直到走完整个 Processor 链。

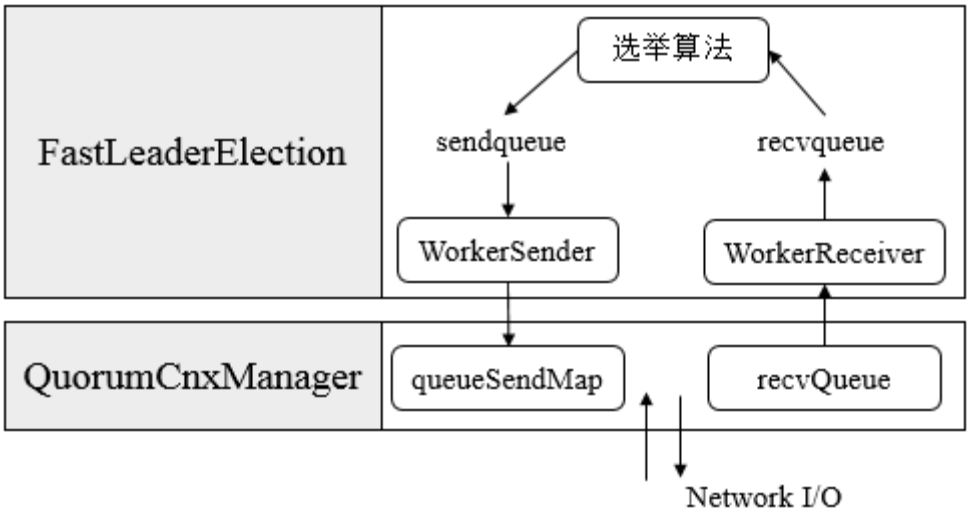
六、 Leader 选举与 QuorumCnxManager 的交互关系

每台服务器在启动的过程中，会启动一个 QuorumPeerManager，负责各台服务器之间的底层 Leader 选举过程中的网络通信。QuorumCnxManager 内部维护了一系列的

队列，用来保存接收到的、待发送的消息以及消息的发送器，除接收队列以外，其他队列都按照 SID 分组形成队列集合，如一个集群中除了自身还有 3 台机器，那么就会为这 3 台机器分别创建一个发送队列，互不干扰。

```
public QuorumCnxManager createCnxnManager() {
    return new QuorumCnxManager(this,
            this.getId(),
            this.getView(),
            this.authServer,
            this.authLearner,
            this.tickTime * this.syncLimit,
            this.getQuorumListenOnAllIPs(),
            this.quorumCnxnThreadsSize,
            this.isQuorumSaslAuthEnabled());
}
```

具体交互关系的实现：



- recvqueue：选票接收队列，用于保存接收到的外部投票。
- sendqueue：选票发送队列，用于保存待发送的选票。
- WorkerReceiver：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，在选票接收过程中，如果发现该外部选票的选举轮次小于当前服务器的，那么忽略该外部投票，同时立即发送自己的内部投票。

- WorkerSender：选票发送器，不断地从 sendqueue 中获取待发送的选票，并将其传递到底层 QuorumCnxManager 中。

下面就针对交互流程进行介绍：

1. 自增选举轮次：Zookeeper 规定所有有效的投票都必须在同一轮次中，在开始新一轮投票时，会首先对 logicalclock 进行自增操作。（在这里 logicalclock 表示本次选举的 id，逻辑时钟的值，这个值从 0 开始递增,每次选举对应一个值,如果在同一次选举中，这个值是一样的，逻辑时钟值越大，说明该节点上的这一次选举 leader 的进程更加新）。

2. 初始化选票：在开始进行新一轮投票之前，每个服务器都会初始化自身的选票，并且在初始化阶段，每台服务器都会将自己推举为 Leader。

3. 发送初始化选票：完成选票的初始化后，服务器就会发起第一次投票。Zookeeper 会将刚刚初始化好的选票放入 sendqueue 中，由发送器 WorkerSender 负责发送出去。

4. 接收外部投票：每台服务器会不断地从 recvqueue 队列中获取外部选票。如果服务器发现无法获取到任何外部投票，那么就会立即确认自己是否和集群中其他服务器保持着有效的连接，如果没有连接，则马上建立连接，如果已经建立了连接，则再次发送自己当前的内部投票。

5. 判断选举轮次：在发送完初始化选票之后，接着开始处理外部投票。在处理外部投票时，会根据选举轮次来进行不同的处理。

6. 选票 PK：在进行选票 PK 时，符合任意一个条件就需要变更投票。

7. 变更投票：经过 PK 后，若确定了外部投票优于内部投票，那么就变更投票，即使用外部投票的选票信息来覆盖内部投票，变更完成后，再次将这个变更后的内部投票发送出去。

8. 选票归档：无论是否变更了投票，都会将刚刚收到的那份外部投票放入选票集合 recvset 中进行归档。recvset 用于记录当前服务器在本轮次的 Leader 选举中收到的所有外部投票（按照服务队的 SID 区别，如{(1, vote1), (2, vote2)...}）。

9. 统计投票：完成选票归档后，就可以开始统计投票，统计投票是为了统计集群中是否已经有过半的服务器认可了当前的内部投票，如果确定已经有过半服务器认可了该投票，则终止投票。否则返回步骤 4。

10. 更新服务器状态：若已经确定可以终止投票，那么就开始更新服务器状态，服务器首选判断当前被过半服务器认可的投票所对应的 Leader 服务器是否是自己，若是

自己，则将自己的服务器状态更新为 LEADING，若不是，则根据具体情况来确定自己是 FOLLOWING 或是 OBSERVING。

以上 10 个步骤就是 Leader 选举和 QuorumCnxManage 交互的核心，其中步骤 4-9 主要是 FastLeaderElection 实现的过程，其会经过几轮循环，直到有 Leader 选举产生。

七、 FastLeaderElection

zookeeper 默认选举算法为 FastLeaderElection.java。其主要方法为 FastLeaderElection.lookForLeader，该接口是一个同步接口，直到选举结束才会返回。

我们知道在 Leader 选举过程中，一个很重要的事情就是每个 node 之间的相互通信，在整个选举过程中这部分由 sendNotifications()实现。它负责每个 node 分别向所有的 node 发送 notification 消息，其主方法：

```
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
                  " (n.round), " + sid + " (recipient), " + self.getId() +
                  " (myid), 0x" + Long.toHexString(proposedEpoch) + " (n.peerEpoch)");
        }
        sendqueue.offer(notmsg);
    }
}
```

mType	type	消息类型
long	leader	推荐的 leader 的 id，就是配置文件中写好的每个服务器的 id
long	zxid	推荐的 leader 的 zxid，zookeeper 中的每份数据,都有一个对应的 zxid 值，越新的数据，zxid 值就越大
long	epoch	logicalclock
ServerState	state	本节点的状态
long	sid	本节点的 id，即 myid

发送消息的问题解决了，现在需要关注的是在节点的状态为 LOOKING 时且没有 stop，就一直 loop 到选出 leader 为止的实现过程：

首先，从消息队列中接收消息：

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

如没有接收到消息，则检查 manager.haveDelivered()，如果已经全部发送出去了，就继续发送，一直到选出 leader 为止。否则就重新连接。

```
/*
 * Sends more notifications if haven't received enough.
 * Otherwise processes new notification.
 */
if(n == null){
    if(manager.haveDelivered()){
        sendNotifications();
    } else {
        manager.connectAll();
    }

    /*
     * Exponential backoff
     */
    int tmpTimeOut = notTimeout*2;
    notTimeout = (tmpTimeOut < maxNotificationInterval?
            tmpTimeOut : maxNotificationInterval);
```

根据消息判断是否为 LOOKING 状态：

```
switch (n.state) {
case LOOKING:
```

如果该节点的 epoch 大于 logicalclock，表示当前是新一轮的选举 。服务器自身的选举轮次落后于该外部投票对应服务器的选举轮次，更新自己的选举轮次 (logicalclock)，并且清空所有已经收到的投票，然后使用初始化的投票来进行 PK 以确定是否变更内部投票。最终再将内部投票发送出去。

```
// 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();
```

调用 totalOrderPredicate 决定是否更新自己的投票，依次比较选举轮数 epoch，事务 zxid，服务器编号 server id(myid) ，依据如下规则 PK 选票：

- 若外部投票中推举的 Leader 服务器的选举轮次大于内部投票，那么需要变更投票。
- 若选举轮次一致，那么就对比两者的 ZXID，若外部投票的 ZXID 大，那么需要变更投票。
- 若两者的 ZXID 一致，那么就对比两者的 SID，若外部投票的 SID 大，那么就需要变更投票。

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

如果该节点的 epoch 小于 logicalclock，则忽略 。

```
 } else if (n.electionEpoch < logicalclock.get()) {
    if(LOG.isDebugEnabled()){
        LOG.debug("Notification election epoch is smaller than logicalclock. n.e
                + Long.toHexString(n.electionEpoch)
                + ", logicalclock=0x" + Long.toHexString(logicalclock.get()));
    }
    break;
```

若等于则直接进行选票的 PK，如上所述。

```
} else if (totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
        proposedLeader, proposedZxid, proposedEpoch)) {
    updateProposal(n.leader, n.zxid, n.peerEpoch);
    sendNotifications();
}
```

投票完成，则把从该节点的信息发到 recvset 中，表明已经收到该节点的回应。

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

判断 recvset 是否已经达到法定 quorum，默认超过半数就通过。

```
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;
        }
    }
```

确定 Leader：

```
 /*
  * 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, proposedEpoch);
     leaveInstance(endVote);
     return endVote;
```

八、 ZooKeeper 中 Leader 选举过程面向对象思想的应用

1、 封装：

QuorumPeer 作为 Leader 选举中的各个服务器的线程是一个封装好的类。

2、 继承：

QuorumPeer 根据 Leader election 确定的这 3 个状态之一对应创建 LeaderZooKeeperServer、FollowerZooKeeperServer、ObserverZooKeeperServer 这三种状态对应的类又是对它的一种继承。

3、 多态：

在不同状态对应的对象中都会调用 SetupManager 进行信息交互，并都会调用方法配合 Leader 选举，但对不同的状态也就是不同的类所创建的对象，在相同的方法下是多态的。

九、 Leader 选举过程中遵循的面向对象设计原则

1、 单一职责原则：

在 Leader 选举中，不同状态所对应的类创建的对象拥有单一的职责，并且被完整的封装在类中，observer，leader，follower 各自有自己的职责。

2、 合成复用原则：

Leader 选举中，各个服务期间多是合成复用而非继承关系，尽量使用对象组合而并没有使用继承来达到复用服务器来进行 Leader 选举的目的。

十、 ZooKeeper 的设计意图

1、 最终一致性：client 不论连接到哪个 Server，展示给它都是同一个视图，Leader 选举达成的就是这一终极目的，这是 zookeeper 最重要的性能。

2、 可靠性：具有简单、健壮、良好的性能，如果消息被到一台服务器接受，那么它将被所有的服务器接受，最典型的应用就是 Leader 选举，所有服务器对于 Leader 的认可。

3、 实时性：Zookeeper 保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，Zookeeper 不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用 sync()接口，在 Leader 选举过程中，所有信息的传输都具有实时性。

4、 等待无关（wait-free）：慢的或者失效的 client 不得干预快速的 client 的请求，使得每个 client 都能有效的等待。

5、 原子性：更新只能成功或者失败，没有中间状态。

6、 顺序性：包括全局有序和偏序两种：全局有序是指如果在一台服务器上消息 a 在消息 b 前发布，则在所有 Server 上消息 a 都将在消息 b 前被发布；偏序是指如果一个消息 b 在消息 a 后被同一个发送者发布，a 必将排在 b 前面。