Zookeeper 的一致性

Zookeeper 的来源

对于 zookeeper 的一致性问题，有很多同学有疑问，我这边再帮大家从来源层面梳理一遍一致性的问题
在第一节课，我们讲到了 zookeeper 的来源，是来自于 google chubby。为了解决在分布式环境下，如何从多个 server 中选举出 master server。那么这多个 server 就需要涉及到一致性问题，这个一致性体现的是多个 server 就 master 这个投票在分布式环境下达成一致性。简单来说就是最终听谁的。但是在网络环境中由于网络的不可靠性，会存在消息丢失和或者被篡改等问题。所以如何在这样一个环境中快速并且正确的在多个 server 中对某一个数据达成一致性并且保证不论发生任何异常，都不会破坏整个系统一致性呢?
所以在 Lamport 大神设计了一套 Paxos 的算法，多个 server 基于这个算法就可以达成一致。而 google chubby 就是基于 paxos 算法的实现，用来实现分布式锁服务。并且提供了 master 选举的服务

Paxos 在 Chubby 中的应用

很多同学会有疑问，Chubby 和 paxos 算法有什么关系？Chubby 本来应该设计成一个包含 Paxos 算法的协议库，是的应用程序可以基于这个库方

便的使用 Paxos 算法，但是它并没有这么做，而是把 Chubby 设计成了一个需要访问中心化节点的分布式锁服务。既然是一个服务，那么它肯定需要是一个高可靠的服务。所以 Chubby 被构建为一个集群，集群中存在一个中心节点（MASTER），采用 Paxos 协议，通过投票的方式来选举一个获得过半票数的服务器作为 Master，在 chubby 集群中，每个服务器都会维护一份数据的副本，在实际的运行过程中， 只有 master 服务器能执行事务操作，其他服务器都是使用 paxos 协议从 master 节点同步最新的数据。而 zookeeper 是 chubby 的开源实现，所以实现原理和 chubby 基本是一致的。

Zookeeper 的一致性是什么情况？

Zookeeper 的一致性，体现的是什么一致呢？

根据前面讲的 zab 协议的同步流程，在 zookeeper 集群内部的数据副本同步，是基于过半提交的策略，意味着它是最终一致性。并不满足强一致的要求。

其实正确来说，zookeeper 是一个顺序一致性模型。由于 zookeeper 设计出来是提供分布式锁服务，那么意味着它本身需要实现顺序一致性（http://zookeeper.apache.org/doc/r3.5.5/zookeeperProgrammers.html#ch_zkGuarantees）

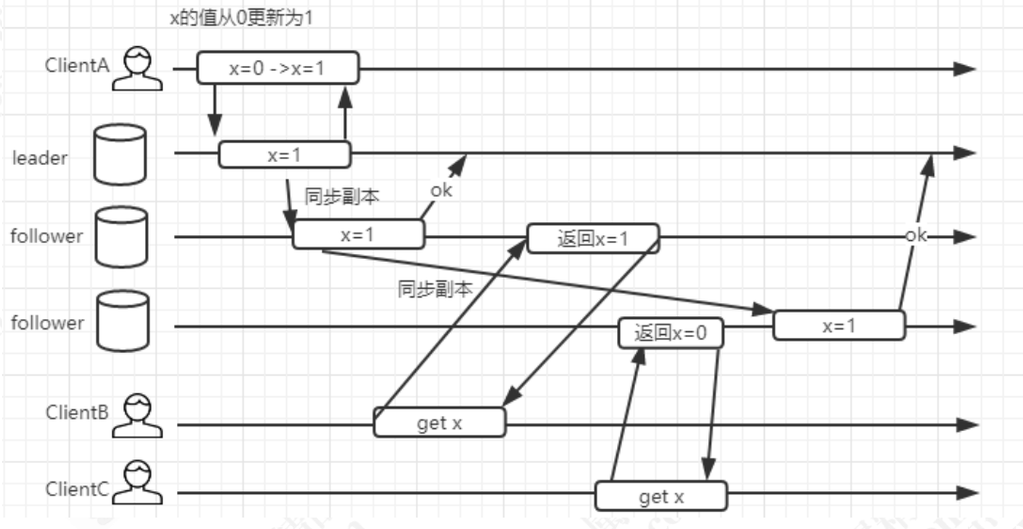
顺序一致性是在分布式环境中实现分布式锁的基本要求，比如当一个多个程序来争抢锁，如果 clientA 获得锁以后，后续所有来争抢锁的程序看到

的锁的状态都应该是被 clientA 锁定了，而不是其他状态。

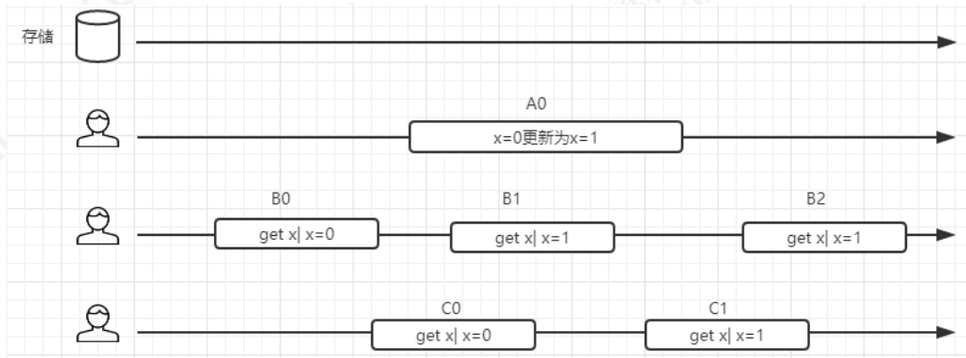
什么是顺序一致性呢？

在讲顺序一致性之前，咱们思考一个问题，假如说 zookeeper 是一个最终一致性模型，那么他会发生什么情况

ClientA/B/C 假设只串行执行， clientA 更新 zookeeper 上的一个值 x。ClientB 和 clientC 分别读取集群的不同副本，返回的 x 的值是不一样的。clientC 的读取操作是发生在 clientB 之后，但是却读到了过期的值。很明显，这是一种弱一致模型。如果用它来实现锁机制是有问题的。



顺序一致性提供了更强的一致性保证，我们来观察下面这个图，从时间轴来看，B0 发生在 A0 之前，读取的值是 0，B2 发生在 A0 之后，读取到的 x 的值为 1.而读操作 B1/C0/C1 和写操作 A0 在时间轴上有重叠，因此他们可能读到旧的值为 0，也可能读到新的值 1. 但是在强顺序一致性模型中，如果 B1 得到的 x 的值为 1，那么 C1 看到的值也一定是 1.



需要注意的是：由于网络的延迟以及系统本身执行请求的不确定性，会导致请求发起的早的客户端不一定会在服务端执行得早。最终以服务端执行的结果为准。

简单来说：顺序一致性是针对单个操作，单个数据对象。属于 CAP 中 C 这个范畴。一个数据被更新后，能够立马被后续的读操作读到。

但是 zookeeper 的顺序一致性实现是缩水版的，在下面这个网页中，可以看到官网对于一致性这块做了解释

http://zookeeper.apache.org/doc/r3.5.5/zookeeperProgrammers.html#ch_zkGuarantees

zookeeper 不保证在每个实例中，两个不同的客户端具有相同的 zookeeper 数据视图，由于网络延迟等因素，一个客户端可能会在另外一个客户端收到更改通知之前执行更新，

考虑到 2 个客户端 A 和 B 的场景，如果 A 把 znode /a 的值从 0 设置为 1，然后告诉客户端 B 读取 /a， 则客户端 B 可能会读取到旧的值 0，具

体取决于他连接到那个服务器，如果客户端 A 和 B 要读取必须要读取到相同的值，那么 client B 在读取操作之前执行 sync 方法。

除此之外，zookeeper 基于 zxid 以及阻塞队列的方式来实现请求的顺序一致性。如果一个 client 连接到一个最新的 follower 上，那么它 read 读取到了最新的数据，然后 client 由于网络原因重新连接到 zookeeper 节点，而这个时候连接到一个还没有完成数据同步的 follower 节点，那么这一次读到的数据不久是旧的数据吗？实际上 zookeeper 处理了这种情况，client 会记录自己已经读取到的最大的 zxid，如果 client 重连到 server 发现 client 的 zxid 比自己大。连接会失败

Single System Image 的理解

zookeeper 官网还说它保证了“Single System Image”，其解释为“A client will see the same view of the service regardless of the server that it connects to.”。实际上看来这个解释还是有一点误导性的。其实由上面 zxid 的原理可以看出，它表达的意思是“client 只要连接过一次 zookeeper，就不会有历史的倒退”。

https://github.com/apache/zookeeper/pull/931

leader 选举的原理

接下来再我们基于源码来分析 leader 选举的整个实现过程。

leader 选举存在与两个阶段中，一个是服务器启动时的 leader 选举。 另一个是运行过程中 leader 节点宕机导致的 leader 选举 ；

在开始分析选举的原理之前，先了解几个重要的参数

服务器 ID（myid）

比如有三台服务器，编号分别是 1,2,3。

编号越大在选择算法中的权重越大。

zxid 事务 id

值越大说明数据越新，在选举算法中的权重也越大

逻辑时钟（epoch – logicalclock）

或者叫投票的次数，同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加，然后与接收到的其它服务器返回的投票信息中的数值相比，根据不同的值做出不同的判断。

选举状态

LOOKING，竞选状态。

FOLLOWING，随从状态，同步 leader 状态，参与投票。

OBSERVING，观察状态,同步 leader 状态，不参与投票。

LEADING，领导者状态。

服务器启动时的 leader 选举

每个节点启动的时候状态都是 LOOKING，处于观望状态，接下来就开始进行选主流程

若进行 Leader 选举，则至少需要两台机器，这里选取 3 台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器 Server1 启动时，其单独无法进行和完成 Leader 选举，当第二台服务器 Server2 启动时，此时两台机器可以相互通信，每台机器都试图找到 Leader，于是进入 Leader 选举过程。选举过程如下

(1) 每个 Server 发出一个投票。由于是初始情况，Server1 和 Server2 都会将自己作为 Leader 服务器来进行投票，每次投票会包含所推举的服务器的 myid 和 ZXID、epoch，使用(myid, ZXID,epoch)来表示，此时 Server1 的投票为(1, 0)，Server2 的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) 接受来自各个服务器的投票。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票（epoch）、是否来自 LOOKING 状态的服务器。

(3) 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行 PK，PK 规则如下

i. 优先比较 epoch

ii. 其次检查 ZXID。ZXID 比较大的服务器优先作为 Leader

iii. 如果 ZXID 相同，那么就比较 myid。myid 较大的服务器作为 Leader 服务器。

对于 Server1 而言，它的投票是(1, 0)，接收 Server2 的投票为(2, 0)，首先会比较两者的 ZXID，均为 0，再比较 myid，此时 Server2 的 myid 最大，于是更新自己的投票为(2, 0)，然后重新投票，对于 Server2 而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

(4) 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于 Server1、Server2 而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了 Leader。

(5) 改变服务器状态。一旦确定了 Leader，每个服务器就会更新自己的状态，如果是 Follower，那么就变更为 FOLLOWING，如果是 Leader，就变更为 LEADING。

运行过程中的 leader 选举

当集群中的 leader 服务器出现宕机或者不可用的情况时，那么整个集群将无法对外提供服务，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 选举基本过程是一致的。

(1) 变更状态。Leader 挂后，余下的非 Observer 服务器都会将自己的服

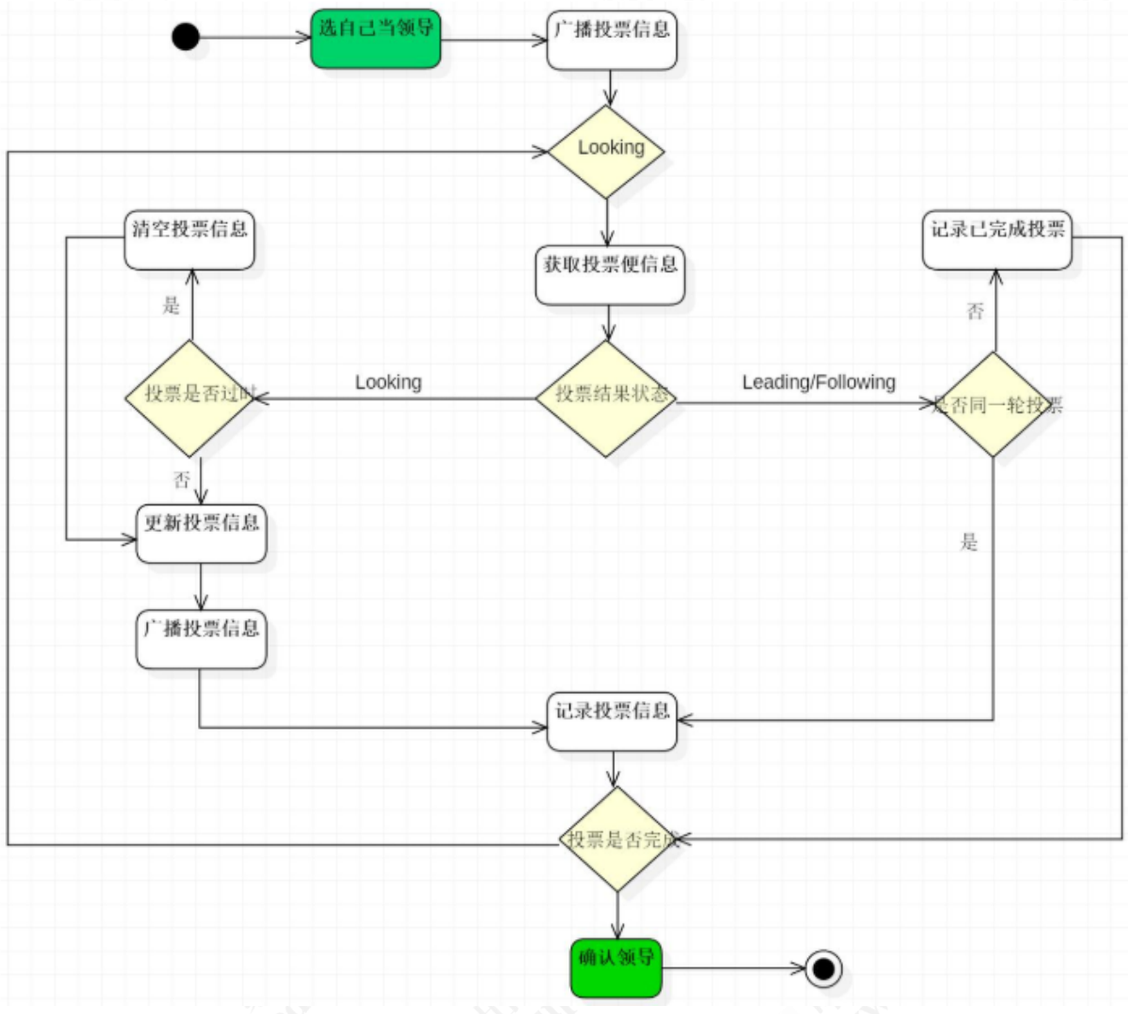
务器状态变更为 LOOKING，然后开始进入 Leader 选举过程。

(2) 每个 Server 会发出一个投票。在运行期间，每个服务器上的 ZXID 可能不同，此时假定 Server1 的 ZXID 为 123，Server3 的 ZXID 为 122；在第一轮投票中，Server1 和 Server3 都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。接收来自各个服务器的投票。与启动时过程相同。

(3) 处理投票。与启动时过程相同，此时，Server1 将会成为 Leader。

(4) 统计投票。与启动时过程相同。

(5) 改变服务器的状态。与启动时过程相同



leader 选举的源码分析

源码分析，最关键的是要找到一个入口，对于 zk 的 leader 选举，并不是由客户端来触发，而是在启动的时候会触发一次选举。因此我们可以直接去看启动脚本 zkServer.sh 中的运行命令

ZOOMAIN 就是 QuorumPeerMain。那么我们基于这个入口来看

```
140     nohup "$JAVA" "-Dzookeeper.log.dir=${ZOO_LOG_DIR}" "-Dzookeeper.root.logger=${ZOO_LOG4J_PROP}" \
141     -cp "$CLASSPATH" $JVMFLAGS $ZOOMAIN "$ZOOCFG" > "$_ZOO_DAEMON_OUT" 2>&1 < /dev/null &
```

QuorumPeerMain.main 方法

main 方法中，调用了 initializeAndRun 进行初始化并且运行

```
protected void initializeAndRun(String[] args)
    throws ConfigException, IOException{
    //这段代码比较简单，设置配置参数，如果 args 不为空，可以基于外部的配置路径来进行解析
    QuorumPeerConfig config = new QuorumPeerConfig();
    if (args.length == 1) {
        config.parse(args[0]);
    }

    // 这里启动了一个线程，来定时对日志进行清理，从命名来看也很容易理解
    DatadirCleanupManager purgeMgr = new DatadirCleanupManager(config
```

```
            .getDataDir(),
config.getDataLogDir(), config
            .getSnapRetainCount(),
config.getPurgeInterval());
    purgeMgr.start();
    //如果是集群模式，会调用 runFromConfig.
servers 实际就是我们在 zoo.cfg 里面配置的集群节点
    if (args.length == 1 &&
config.servers.size() > 0) {
        runFromConfig(config);
    } else {//否则直接运行单机模式
        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);
    }
}
```

runFromConfig

从名字可以看出来，是基于配置文件来进行启动。

所以整个方法都是对参数进行解析和设置 ， 因为这些参数暂时还没用到，所以没必要去看。直接看核心的代码

quorumPeer.start() ， 启动一个线程，那么从这句代码可以看出来 QuorumPeer 实际是继承了线程。那么它里面一定有一个 run 方法

```
    public void runFromConfig(QuorumPeerConfig
config) throws IOException {
  try {
      ManagedUtil.registerLog4jMBeans();
  } catch (JMException e) {
      LOG.warn("Unable to register log4j JMX
control", e);
  }

  LOG.info("Starting quorum peer");
  try {
      ServerCnxnFactory cnxnFactory =
ServerCnxnFactory.createFactory();
```

```
cnxnFactory.configure(config.getClientPortAddress()
,
                      config.getMaxClientCnxns());

    quorumPeer = getQuorumPeer();


quorumPeer.setQuorumPeers(config.getServers());
    quorumPeer.setTxnFactory(new FileTxnSnapLog(
            new File(config.getDataLogDir()),
            new File(config.getDataDir())));

quorumPeer.setElectionType(config.getElectionAlg())
;
    quorumPeer.setMyid(config.getServerId());
    quorumPeer.setTickTime(config.getTickTime());

quorumPeer.setInitLimit(config.getInitLimit());

quorumPeer.setSyncLimit(config.getSyncLimit());
```

```
        //投票决定方式，默认超过半数就通过

quorumPeer.setQuorumListenOnAllIPs(config.getQuorum
ListenOnAllIPs());
        quorumPeer.setCnxnFactory(cnxnFactory);

quorumPeer.setQuorumVerifier(config.getQuorumVerifi
er());

quorumPeer.setClientPortAddress(config.getClientPor
tAddress());

quorumPeer.setMinSessionTimeout(config.getMinSessio
nTimeout());

quorumPeer.setMaxSessionTimeout(config.getMaxSessio
nTimeout());
        quorumPeer.setZKDatabase(new
ZKDatabase(quorumPeer.getTxnFactory()));
```

```
quorumPeer.setLearnerType(config.getPeerType());

quorumPeer.setSyncEnabled(config.getSyncEnabled());

    // sets quorum sasl authentication
configurations

quorumPeer.setQuorumSaslEnabled(config.quorumEnable
Sasl);
    if(quorumPeer.isQuorumSaslAuthEnabled()){

quorumPeer.setQuorumServerSaslRequired(config.quoru
mServerRequireSasl);

quorumPeer.setQuorumLearnerSaslRequired(config.quor
umLearnerRequireSasl);

quorumPeer.setQuorumServicePrincipal(config.quorumS
ervicePrincipal);
```

```
quorumPeer.setQuorumServerLoginContext(config.quoru
mServerLoginContext);

quorumPeer.setQuorumLearnerLoginContext(config.quor
umLearnerLoginContext);
      }

quorumPeer.setQuorumCnxnThreadsSize(config.quorumCn
xnThreadsSize);
      quorumPeer.initialize();
      //启动主线程
      quorumPeer.start();
      quorumPeer.join();
  } catch (InterruptedException e) {
      // warn, but generally this is ok
      LOG.warn("Quorum Peer interrupted", e);
  }
}
```

QuorumPeer.start

QuorumPeer.start 方法，重写了 Thread 的 start。也就是在线程启动之前，会做以下操作

1. 通过 loadDataBase 恢复快照数据
2. cnxnFactory.start() 启动 zkServer，相当于用户可以通过 2181 这个端口进行通信了，这块后续在讲。我们还是以 leader 选举为主线

```
@Override
public synchronized void start() {
    loadDataBase();
    cnxnFactory.start();
    startLeaderElection();
    super.start();
}
```

startLeaderElection

看到这个方法，有没有两眼放光的感觉？没错，前面铺垫了这么长，终于进入 leader 选举的方法了

```
synchronized public void startLeaderElection() {
   try {
```

```
        //构建一个票据，用于投票
        currentVote = new Vote(myid,
getLastLoggedZxid(), getCurrentEpoch());
    } catch(IOException e) {
        RuntimeException re = new
RuntimeException(e.getMessage());
        re.setStackTrace(e.getStackTrace());
        throw re;
    }
     //这个 getView 返回的就是在配置文件中配置的
server.myid=ip:port:port。view 在哪里解析的呢？
    for (QuorumServer p : getView().values()) {
        if (p.id == myid) {//获得当前 zkserver myid 对应
的 ip 地址
            myQuorumAddr = p.addr;
            break;
        }
    }
    if (myQuorumAddr == null) {
        throw new RuntimeException("My id " + myid +
```

```
" not in the peer list");
    }
    //根据 electionType 匹配对应的选举算法,electionType 默认值为 3.可以在配置文件中动态配置
    if (electionType == 0) {
        try {
            udpSocket = new DatagramSocket(myQuorumAddr.getPort());
            responder = new ResponderThread();
            responder.start();
        } catch (SocketException e) {
            throw new RuntimeException(e);
        }
    }
    this.electionAlg = createElectionAlgorithm(electionType);
}
```

quorumPeer. createElectionAlgorithm

根据对应的标识创建选举算法

```
    protected Election createElectionAlgorithm(int
electionAlgorithm){
    Election le=null;
    //TODO: use a factory rather than a switch
    switch (electionAlgorithm) {
    case 0:
        le = new LeaderElection(this);
        break;
    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();  //启动监听器，这个监听具体
```

```
做什么的暂时不管，后面遇到需要了解的地方再回过头来看
            le = new FastLeaderElection(this, qcm);//
初始化 FastLeaderElection
          } else {
            LOG.error("Null listener when
initializing cnx manager");
          }
        break;
    default:
        assert false;
    }
    return le;
}
```

FastLeaderElection

初始化FastLeaderElection，QuorumCnxManager是一个很核心的对象，用来实现领导选举中的网络连接管理功能，这个后面会用到

```
    public FastLeaderElection(QuorumPeer
self, QuorumCnxManager manager){
    this.stop = false;
```

```
    this.manager = manager;
    starter(self, manager);
}
```

FastLeaderElection. starter

starter 方法里面，设置了一些成员属性，并且构建了两个阻塞队列，分别是 sendQueue 和 recvqueue。并且实例化了一个 Messager

```
    private void starter(QuorumPeer self,
QuorumCnxManager manager) {
    this.self = self;
    proposedLeader = -1;
    proposedZxid = -1;

    sendqueue = new
LinkedBlockingQueue<ToSend>();
    recvqueue = new
LinkedBlockingQueue<Notification>();
    this.messenger = new Messenger(manager);
}
```

Messenger

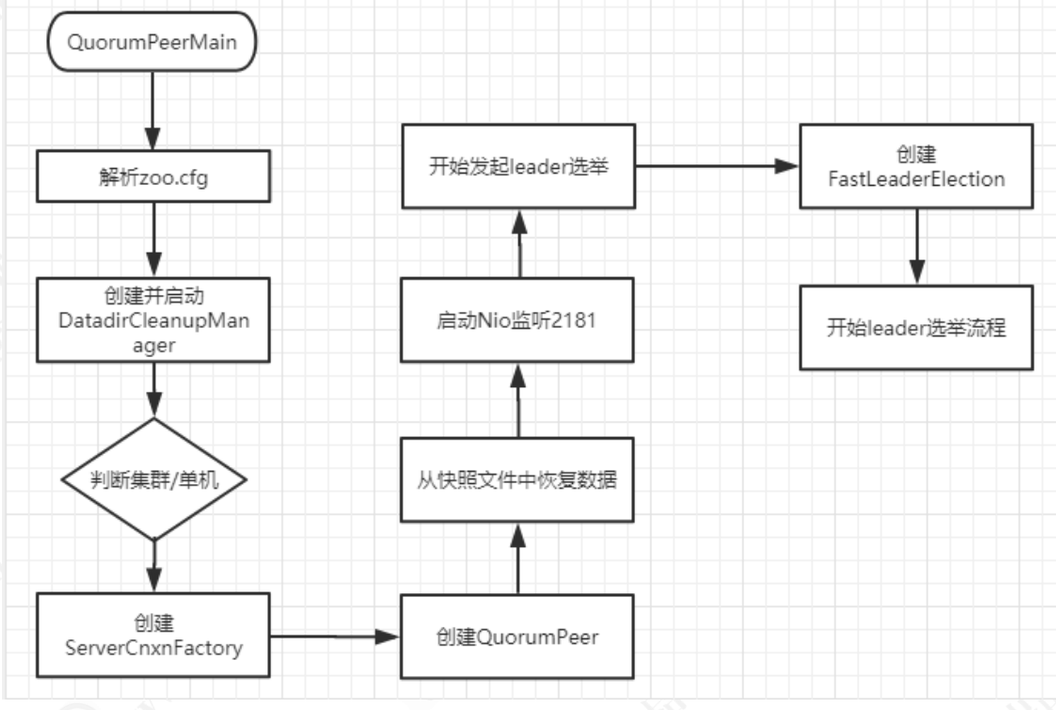
在 Messenger 里面构建了两个线程，一个是 WorkerSender，一个是 WorkerReceiver。 这两个线程是分别用来发送和接收消息的线程。具体做什么，暂时先不分析。

```
    Messenger(QuorumCnxManager manager) {

    this.ws = new WorkerSender(manager);

    Thread t = new Thread(this.ws,
            "WorkerSender[myid=" +
self.getId() + "]");
    t.setDaemon(true);
    t.start();

    this.wr = new WorkerReceiver(manager);

    t = new Thread(this.wr,
            "WorkerReceiver[myid=" +
self.getId() + "]");
    t.setDaemon(true);
```

```
        t.start();
    }
```

阶段性总结

ok，分析到这里，先做一个简单的总结，通过一个流程图把前面部分的功能串联起来。



getView 的解析流程

```
public Map<Long,QuorumPeer.QuorumServer>
getView() {
    return
```

```
Collections.unmodifiableMap(this.quorumPeers);
}
```

getView 里面实际上返回的是一个 quorumPeers，就是参与本次投票的成员有哪些。这个属性在哪里赋值的呢?

我们又得回到 runFromConfig 方法中

QuorumPeerMain.runFromConfig

设置了一个值为 config.getServers（）

```
    //...
    quorumPeer.setQuorumPeers(config.getServers());
    //...
```

config 这个配置信息又是通过在 initializeAndRun 方法中初始化的，

```
    protected void initializeAndRun(String[] args)
    throws ConfigException, IOException
{
    QuorumPeerConfig config = new QuorumPeerConfig();
```

```
    if (args.length == 1) {
        config.parse(args[0]);
    }
```

QuorumPeerConfig.parse

这里会根据一个外部的文件去进行解析，然后其中有一段是这样，解析对应的集群配置数据放到 servers 这个集合中

```
    } else if (key.startsWith("server.")) {
    int dot = key.indexOf('.');
    long sid = Long.parseLong(key.substring(dot + 1));
    String parts[] = splitWithLeadingHostname(value);
    if ((parts.length != 2) && (parts.length != 3) && (parts.length !=4)) {
        LOG.error(value
           + " does not have the form host:port or host:port:port " +
```

```
            " or host:port:port:type");
    }
```

ZkServer 服务启动的逻辑

在讲 leader 选举的时候，有一个 cnxnFactory.start()方法来启动 zk 服务，这块具体做了什么呢？我们来分析看看

QuorumPeerMain.runFromConfig

在 runFromConfig 中，有构建了一个 ServerCnxnFactory.

```
public void runFromConfig(QuorumPeerConfig
config) throws IOException {
  //…
  LOG.info("Starting quorum peer");
  try {
      ServerCnxnFactory cnxnFactory =
ServerCnxnFactory.createFactory();

cnxnFactory.configure(config.getClientPortAddres
```

```
s(),

config.getMaxClientCnxns());
  //…
     quorumPeer.setCnxnFactory(cnxnFactory);
```

并且将这个 factory 设置给了 quorumPeer 的成员属性

```
  //…
```

这个很明显是一个工厂模式，基于这个工厂类创建什么呢？ 打开 createFactory 方法看看就知道了

ServerCnxnFactory.createFactory

这个方法里面是根据 ***ZOOKEEPER_SERVER_CNXN_FACTORY*** 来决定创建 NIO server 还是 Netty Server

而默认情况下，应该是创建一个 NIOServerCnxnFactory

```
static public ServerCnxnFactory createFactory()
throws IOException {
    String serverCnxnFactoryName =

System.getProperty(ZOOKEEPER_SERVER_CNXN_FACTORY);
    if (serverCnxnFactoryName == null) {
```

```
        serverCnxnFactoryName =
NIOServerCnxnFactory.class.getName();
    }
    try {
        ServerCnxnFactory serverCnxnFactory =
(ServerCnxnFactory)
Class.forName(serverCnxnFactoryName)
                .getDeclaredConstructor().newInstance(
);
        LOG.info("Using {} as server connection
factory", serverCnxnFactoryName);
        return serverCnxnFactory;
    } catch (Exception e) {
        IOException ioe = new IOException("Couldn't
instantiate "
                + serverCnxnFactoryName);
        ioe.initCause(e);
        throw ioe;
    }
}
```

QuorumPeer.start

因此，我们再回到 QuorumPeer.start()方法中，cnxnFactory.start()，应该会调用 NIOServerCnxnFactory 这个类去启动一个线程

```
    public synchronized void start() {
    loadDataBase();
    cnxnFactory.start();
    startLeaderElection();
    super.start();
}
```

NIOServerCnxnFactory.start

这里通过 thread.start 启动一个线程，那 thread 是一个什么对象呢?

```
    public void start() {
    // ensure thread is started once and
only once
    if (thread.getState() ==
Thread.State.NEW) {
        thread.start();
    }
}
```

NIOServerCnxnFactory.configure

thread 其实构建的是一个 zookeeperThread 线程，并且线程的参数为 this，表示当前 NIOServerCnxnFactory 也是实现了线程的类，那么它必须要重写

run 方法，因此定位到 NIOServerCnxnFactory.run。

到此，NIOServer 的初始化以及启动过程就完成了。并且对 2181 的这个端口进行监听。一旦发现有请求进来，就执行相应的处理即可。这块后续在分析数据同步的时候再做详细了解

```
    Thread thread;
@Override
public void configure(InetSocketAddress addr,
int maxcc) throws IOException {
    configureSaslLogin();

    thread = new ZooKeeperThread(this,
"NIOServerCxn.Factory:" + addr);
    thread.setDaemon(true);
    maxClientCnxns = maxcc;
```

```
    this.ss = ServerSocketChannel.open();
    ss.socket().setReuseAddress(true);
    LOG.info("binding to port " + addr);
    ss.socket().bind(addr);
    ss.configureBlocking(false);
    ss.register(selector,
SelectionKey.OP_ACCEPT);
}
```

选举流程分析

前面分析这么多，还没有正式分析到 leader 选举的核心流程，前期准备工作做好了以后，接下来就开始正式分析 leader 选举的过程

```
    public synchronized void start() {
    loadDataBase();
    cnxnFactory.start();
    startLeaderElection();
    super.start(); //启动线程
}
```

很明显，super.start() 表示当前类 QuorumPeer 继承了线程，线程必须要重写 run 方法，所以我们可以在 QuorumPeer 中找到一个 run 方法

QuorumPeer.run

这段代码的逻辑比较长。粗略看一下结构，好像也不难

PeerState 有几种状态，分别是

1. LOOKING，竞选状态。

2. FOLLOWING，随从状态，同步 leader 状态，参与投票。

3. OBSERVING，观察状态,同步 leader 状态，不参与投票。

4. LEADING，领导者状态。

对于选举来说，默认都是 LOOKING 状态，

只有 LOOKING 状态才会去执行选举算法。每个服务器在启动时都会选择自己做为领导，然后将投票信息发送出去，循环一直到选举出领导为止。

```
    @Override
public void run() {
    setName("QuorumPeer" + "[myid=" + getId() + "]"
+
            cnxnFactory.getLocalAddress());
    //... 根据选举状态，选择不同的处理方式
    while (running) {
    switch (getPeerState()) {
    case LOOKING:
        LOG.info("LOOKING");
```

```
        //判断是否为只读模式,通过”readonlymode.enabled”开启
        if (Boolean.getBoolean("readonlymode.enabled")) {
            //只读模式的启动流程
            } else {
            try {
                setBCVote(null);
                    //设置当前的投票，通过策略模式来决定当前用哪个选举算法来进行领导选举

setCurrentVote(makeLEStrategy().lookForLeader());
            } catch (Exception e) {
                LOG.warn("Unexpected exception", e);
                setPeerState(ServerState.LOOKING);
            }
        }
        break;
        //…后续逻辑暂时不用管
    }
```

FastLeaderElection.lookForLeader

开始发起投票流程

```
    public Vote lookForLeader() throws
InterruptedException {
    //...
    try {
    HashMap<Long, Vote> recvset = new HashMap<Long,
Vote>();
    HashMap<Long, Vote> outofelection = new
HashMap<Long, Vote>();
    int notTimeout = finalizeWait;


    synchronized(this){
        logicalclock.incrementAndGet();//更新逻辑时钟，用来
判断是否在同一轮选举周期
        //初始化选票数据：这里其实就是把当前节点的 myid，zxid，
epoch 更新到本地的成员属性
```

```
        updateProposal(getInitId(),
getInitLastLoggedZxid(), getPeerEpoch());
    }


    LOG.info("New election. My id =  " + self.getId() +
            ", proposed zxid=0x" +
Long.toHexString(proposedZxid));
    sendNotifications();//异步发送选举信息
    /*
     * Loop in which we exchange notifications until we
find a leader
     */
    //这里就是不断循环，根据投票信息进行进行 leader 选举
    while ((self.getPeerState() == ServerState.LOOKING)
&&
            (!stop)){
        /*
         * Remove next notification from queue, times
out after 2 times
```

```
             * the termination time
             */
            //从 recvqueue 中获取消息
            Notification n = recvqueue.poll(notTimeout,
                    TimeUnit.MILLISECONDS);

            /*
             * Sends more notifications if haven't received
enough.
             * Otherwise processes new notification.
             */
            if(n == null){  //如果没有获取到外部的投票，有可能是集
群之间的节点没有真正连接上
                if(manager.haveDelivered()){//判断发送队列是否有
数据，如果发送队列为空，再发一次自己的选票
                    sendNotifications();
                } else {//在此发起集群节点之间的连接
                    manager.connectAll();
                }
                /*
```

```
                * Exponential backoff
                */
               int tmpTimeOut = notTimeout*2;
               notTimeout = (tmpTimeOut <
maxNotificationInterval?
                       tmpTimeOut : maxNotificationInterval);
               LOG.info("Notification time out: " +
notTimeout);
           }
       //…
       }
```

选票的判断逻辑（核心代码）

```
//判断收到的选票中的 sid 和选举的 leader 的 sid 是否存在于我们集群所配置的 myid 范围

else if(validVoter(n.sid) && validVoter(n.leader))
{
    //判断接收到的投票者的状态，默认是 LOOKING 状态,说明当前发起投票的服务器也是在找 leader
```

```
    switch (n.state) {
    case LOOKING: 说明当前发起投票的服务器也是在找 leader
        // 如果收到的投票的逻辑时钟大于当前的节点的逻辑时钟
        if (n.electionEpoch > logicalclock.get()) {
            logicalclock.set(n.electionEpoch);//更新成新一轮的逻辑时钟
            recvset.clear();
            //比较接收到的投票和当前节点的信息进行比较，比较的顺序
            epoch、zxid、myid,如果返回 true，则更新当前节点的票据（sid,zxid,epoch），
            那么下次再发起投票的时候，就不再是选自己了
            if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
                    getInitId(), getInitLastLoggedZxid(), getPeerEpoch())) {
                updateProposal(n.leader, n.zxid, n.peerEpoch);
            } else {//否则，说明当前节点的票据优先级更高，再次更新自己的票据
```

```
                updateProposal(getInitId(),
                        getInitLastLoggedZxid(),
                        getPeerEpoch());
            }
            sendNotifications();//再次发送消息把当前的票据发出去
        } else if (n.electionEpoch < logicalclock.get()) {//如果小于，说明收到的票据已经过期了，直接把这张票丢掉
            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;
            //这个判断表示收到的票据的 epoch 是相同的，那么按照 epoch、zxid、myid 顺序进行比较
```

```
                比较成功以后，把对方的票据信息更新到自己的节点
            } else if (totalOrderPredicate(n.leader,
n.zxid, n.peerEpoch,
                    proposedLeader, proposedZxid,
proposedEpoch)) {
                updateProposal(n.leader, n.zxid,
n.peerEpoch);
                sendNotifications();//把收到的票据再发出去，
告诉大家我要选 n.leader 为 leader
            }

            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));
            }
            //将收到的投票信息放入投票的集合 recvset 中，用来作为
```

最终的 "过半原则" 判断

```
        recvset.put(n.sid, new Vote(n.leader, n.zxid,
n.electionEpoch, n.peerEpoch));
        //判断选举是否结束
        if (termPredicate(recvset,
                new Vote(proposedLeader, proposedZxid,
                        logicalclock.get(),
proposedEpoch))) {
            //进入这个判断，说明选票达到了 leader 选举的要求
            //在更新状态之前，服务器会等待 finalizeWait 毫秒
时间来接收新的选票，以防止漏下关键选票。
            如果收到可能改变 Leader 的新选票，则重新进行计票
            while((n = recvqueue.poll(finalizeWait,
                    TimeUnit.MILLISECONDS)) != null){
                if(totalOrderPredicate(n.leader,
n.zxid, n.peerEpoch,
                        proposedLeader, proposedZxid,
proposedEpoch)){
                    recvqueue.put(n);
                    break;
```

```
                }
            }
            //如果 notifaction 为空，说明 Leader 节点是可以确定好了
            if (n == null) {
                设置当前当前节点的状态（判断 leader 节点是不是我自己，如果是，直接更新当前节点的 state 为 LEADING）
                否则，根据当前节点的特性进行判断，决定是 FOLLOWING 还是 OBSERVING
                self.setPeerState((proposedLeader == self.getId()) ?
                        ServerState.LEADING: learningState());
                //组装生成这次 Leader 选举最终的投票的结果
                Vote endVote = new Vote(proposedLeader,
                                        proposedZxid,
                                        logicalclock.get(),
                                        proposedEpoch);
```

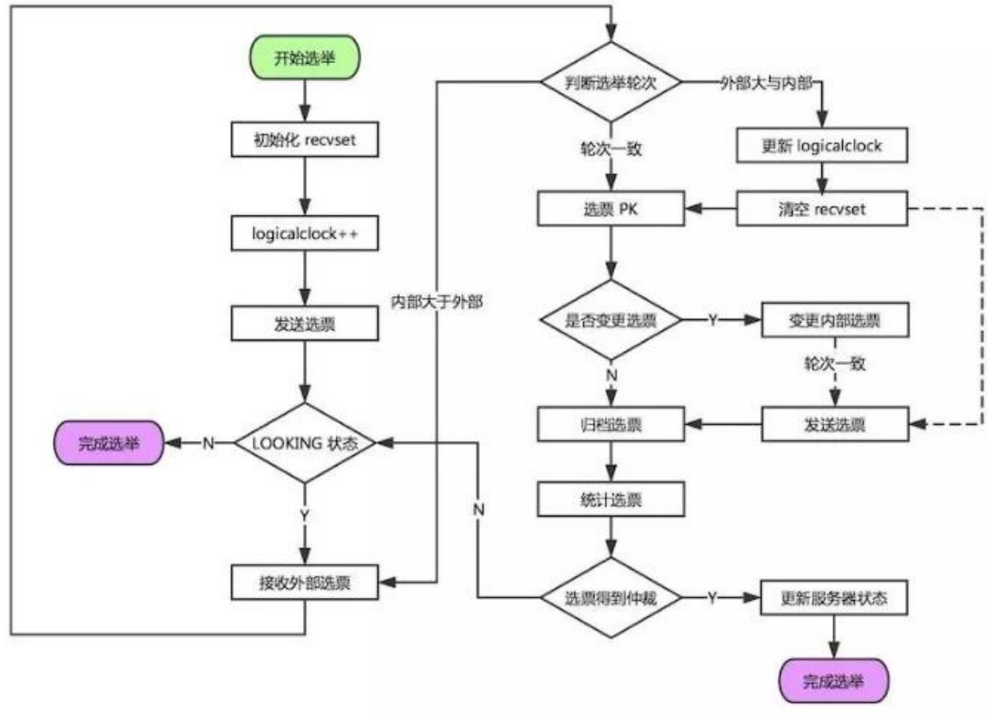
```
                leaveInstance(endVote);// 清空
recvqueue
                return endVote; //返回最终的票据
            }
        }
        break;
    case OBSERVING://OBSERVING 不参与 leader 选举
        LOG.debug("Notification from observer: " +
n.sid);
        break;
    case FOLLOWING:
    case LEADING:
        /*
         * Consider all notifications from the same
epoch
         * together.
         */
        if(n.electionEpoch == logicalclock.get()){
            recvset.put(n.sid, new Vote(n.leader,
                                        n.zxid,
```

```
                                                n.electionEpoch,
                                                n.peerEpoch));


                    if(ooePredicate(recvset, outofelection,
n)) {
                        self.setPeerState((n.leader ==
self.getId()) ?
                                ServerState.LEADING:
learningState());

                        Vote endVote = new Vote(n.leader,
                                n.zxid,
                                n.electionEpoch,
                                n.peerEpoch);
                        leaveInstance(endVote);
                        return endVote;
                    }
                }

                /*
```

```
            * Before joining an established ensemble,
verify
            * a majority is following the same leader.
            */
           outofelection.put(n.sid, new Vote(n.version,
                                             n.leader,
                                             n.zxid,
n.electionEpoch,
                                             n.peerEpoch,
                                             n.state));

           if(ooePredicate(outofelection, outofelection,
n)) {
               synchronized(this){
                   logicalclock.set(n.electionEpoch);
                   self.setPeerState((n.leader ==
self.getId()) ?
                           ServerState.LEADING:
learningState());
```

```
                }
                Vote endVote = new Vote(n.leader,
                                        n.zxid,
                                        n.electionEpoch,
                                        n.peerEpoch);
                leaveInstance(endVote);
                return endVote;
            }
            break;
        default:
            LOG.warn("Notification state unrecognized: {}
(n.state), {} (n.sid)",
                    n.state, n.sid);
            break;
        }
}
```

投票处理的流程图



termPredicate

这个方法是使用过半原则来判断选举是否结束，如果返回 true，说明能够选出 leader 服务器

votes 表示收到的外部选票的集合

vote 表示但前服务器的选票

```
protected boolean termPredicate(
        HashMap<Long, Vote> votes,
        Vote vote) {
```

```
    HashSet<Long> set = new HashSet<Long>();
    //遍历接收到的所有选票数据
    for (Map.Entry<Long,Vote> entry :
votes.entrySet()) {
        //对选票进行归纳，就是把所有选票数据中和当前
节点的票据相同的票据进行统计
            if
(vote.equals(entry.getValue())){
            set.add(entry.getKey());
        }
    }//对选票进行判断
    return
self.getQuorumVerifier().containsQuorum(set
);
}
```

QuorumMaj. containsQuorum

判断当前节点的票数是否是大于一半，默认采用 QuorumMaj 来实现

```
public boolean containsQuorum(Set<Long>
set){
    return (set.size() > half);
}
```

这个 half 的值是多少呢?

可以在 QuorumPeerConfig.parseProperties 这个方法中，找到如下代码。

```
398                 LOG.info("Defaulting to major
399                 quorumVerifier = new QuorumMa
```

也就是说，在构建 QuorumMaj 的时候，传递了当前集群节点的数量，这里是 3

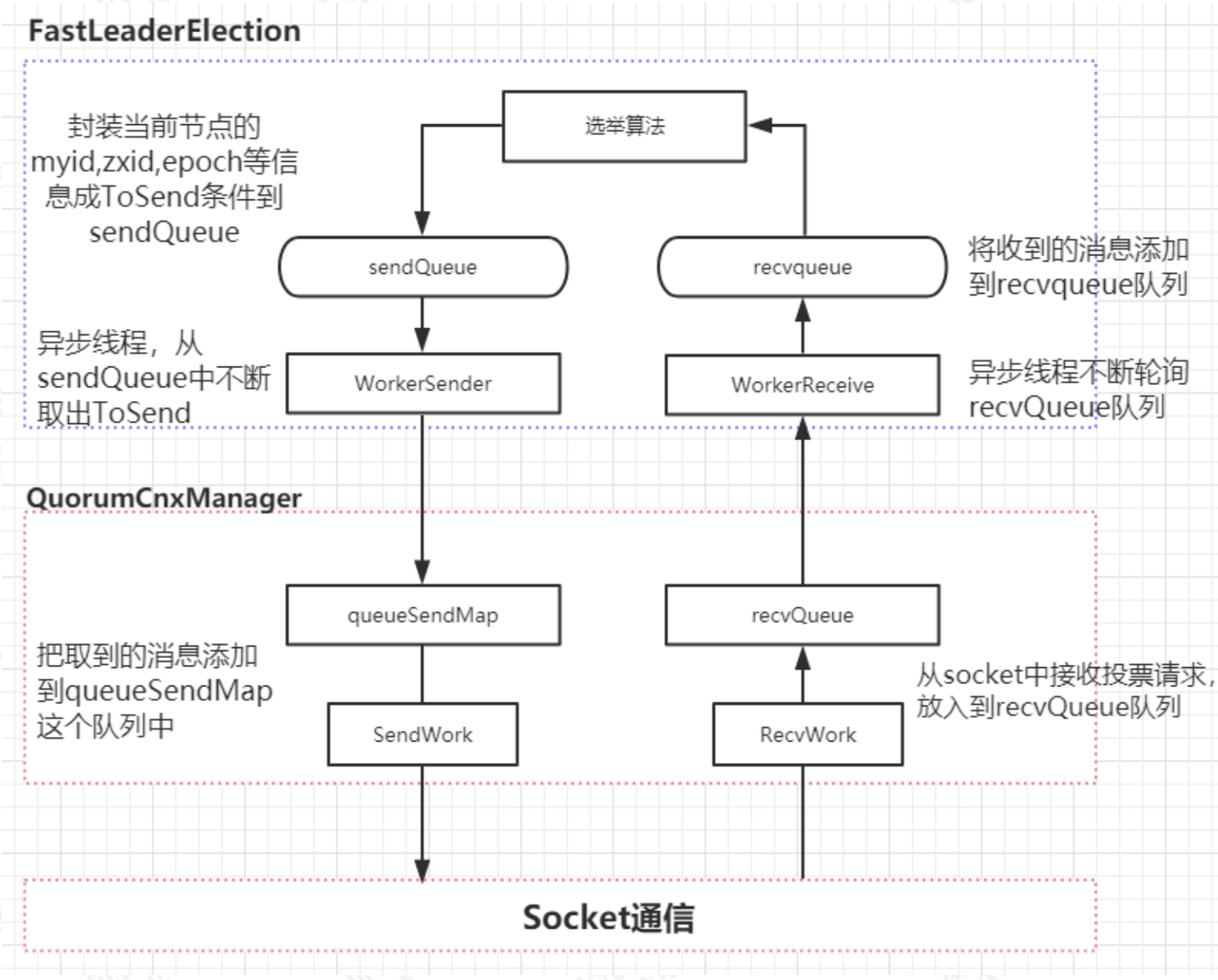
那么，hafl=3/2=1

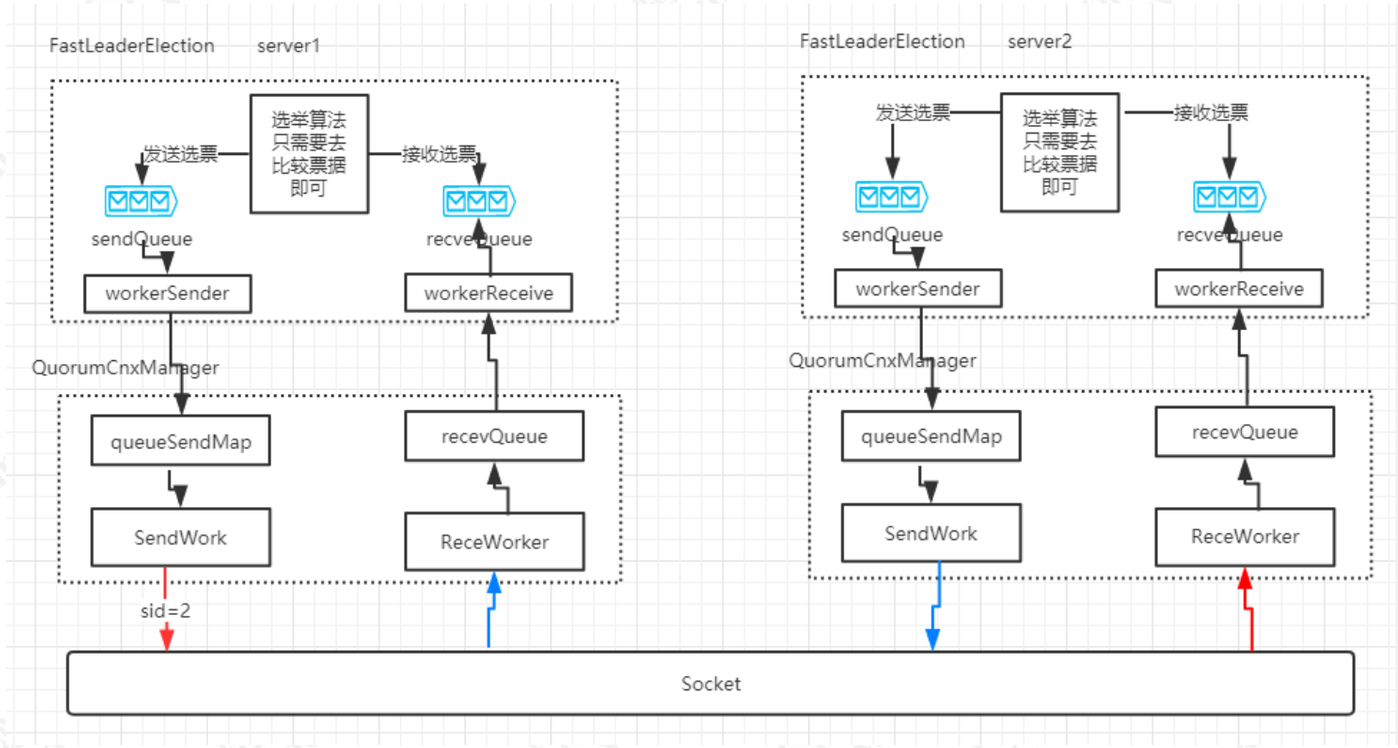
```
public QuorumMaj(int n){
    this.half = n/2;
}
```

那么 set.size()>1. 意味着至少要有两个节点的票据是选择你当 leader，否则，还得继续投

投票的网络通信流程

通信流程图





接收数据 Notification 和发送 ToSend

ToSender	Notification
leader； 被推荐的服务器 sid	leader； //被推荐的服务器 sid
zxid; 被推荐的服务器当前最新的事务 id	zxid； 被推荐的服务器最新事务 id
peerEpoch; 被推荐的服务器当前所处的 epoch	peerEpoch; 被推荐的服务器当前所处的 epoch
electionepoch； 当前服务器所处的 epoch	electionEpoch 选举服务器所处的 epoch
stat 当前服务器状态	stat； 选举服务器当前的状态

sid 接收消息的服务器 sid（myid）	sid； 选举服务器的 sid

通信过程源码分析

每个 zk 服务启动后创建 socket 监听

```
protected Election
createElectionAlgorithm(int
electionAlgorithm){
//….
case 3:
    qcm = createCnxnManager();
    QuorumCnxManager.Listener listener =
qcm.listener;
    if(listener != null){
        listener.start(); //启动监听
}
```

```
listener 实现了线程，所以在 run 方法中可以看到构建
ServerSocket 的请求，这里专门用来接收其他
zkServer 的投票请求，//这块后续再分析
@Override
public void run() {
```

```
    int numRetries = 0;
    InetSocketAddress addr;
    while((!shutdown) && (numRetries < 3)){
        try {
            ss = new ServerSocket();
//…
}
```

FastLeaderElection.lookForLeader

这个方法在前面分析过，里面会调用 sendNotifications 来发送投票请求

```
public Vote lookForLeader() throws
InterruptedException {
    //省略部分代码
    sendNotifications(); //这个方法，会把当前
zk 服务器的信息添加到 sendqueue
     /*
      * Loop in which we exchange
notifications until we find a leader
      */
```

```
    while ((self.getPeerState() ==
ServerState.LOOKING) &&
    //省略部分代码
}
```

FastLeaderElection.sendqueue

sendQueue 这个队列的数据，是通过 WorkerSender 来进行获取并发送的。而这个 WorkerSender 线程，在构建 fastLeaderElection 时，会启动

```
    class WorkerSender extends
ZooKeeperThread {
    public void run() {
    while (!stop) {
        try {
            //从队列中获取 ToSend 对象
            ToSend m = sendqueue.poll(3000,
TimeUnit.MILLISECONDS);
            if(m == null) continue;

            process(m);
```

```
    //省略部分代码
    void process(ToSend m) {
    ByteBuffer requestBuffer =
buildMsg(m.state.ordinal(),

m.leader,
                                        m.zxid,

m.electionEpoch,

m.peerEpoch);
    manager.toSend(m.sid, requestBuffer); //
这里就是调用 QuorumCnxManager 进行消息发送
}
```

QuorumCnxManager.toSend

```
public void toSend(Long sid, ByteBuffer b)
{
```

```
    if (this.mySid == sid) {//如果接受者是自己，直接放置到接收队列
        b.position(0);
        addToRecvQueue(new Message(b.duplicate(), sid));
    } else {  //否则发送到对应的发送队列上
        ArrayBlockingQueue<ByteBuffer> bq = new ArrayBlockingQueue<ByteBuffer>(SEND_CAPACITY);
         //判断当前的 sid 是否已经存在于发送队列，如果是，则直接把已经存在的数据发送出去
        ArrayBlockingQueue<ByteBuffer> bqExisting = queueSendMap.putIfAbsent(sid, bq);
        if (bqExisting != null) {
            addToSendQueue(bqExisting, b);
        } else {
            addToSendQueue(bq, b);
```

```
            }
            connectOne(sid);//连接申请
```

调用链 connectOne-->initiateConnection-->startConnection ， startConnection 就是发送方启动入口

```
        }
}
```

startConnection

```
private boolean startConnection(Socket sock, Long sid)
    //省略部分代码
    if (sid > this.mySid) {
        //为了防止重复建立连接，只允许 sid 大的主动连接 sid 小的
        closeSocket(sock);
    } else {
        //构建一个发送线程和接收线程,负责针对当前连接的数据传递,后续的逻辑比较简单，就不做分析
```

```
    SendWorker sw = new SendWorker(sock,
    sid);
    RecvWorker rw = new RecvWorker(sock,
    din, sid, sw);
    sw.setRecv(rw);
  }
```

SendWorker 会监听对应 sid 的阻塞队列，启动的时候回如果队列为空时会重新发送一次最前最后的消息，以防上一次处理是服务器异常退出，造成上一条消息未处理成功；然后就是不停监听队里，发现有消息时调用 send 方法

RecvWorker：RecvWorker 不停监听 socket 的 inputstream，读取消息放到消息接收队列中,消息放入队列中，qcm 的流程就完毕了。

QuorumCnxManager.Listener

listener 监听到客户端请求之后，开始处理消息

```
public void run() {
  //省略部分代码
   while (!shutdown) {
       Socket client = ss.accept();
       setSockOpts(client);
```

```
        LOG.info("Received connection request
    "
                +
    client.getRemoteSocketAddress());
        if (quorumSaslAuthEnabled) {
            receiveConnectionAsync(client);
        } else {
            receiveConnection(client); //接收客
    户端请求
        }
}
```

QuorumCnxManager.receiveConnection

```
public void receiveConnection(final Socket sock)
{
    DataInputStream din = null;
    try {
        //获取客户端的数据包
        din = new DataInputStream(
```

```
                new
BufferedInputStream(sock.getInputStream()));

        handleConnection(sock, din);//调用 handle 进
行处理
    } catch (IOException e) {
        LOG.error("Exception handling connection,
addr: {}, closing server connection",
                  sock.getRemoteSocketAddress());
        closeSocket(sock);
    }
}
```

handleConnection

```
private void handleConnection(Socket sock,
DataInputStream din)
        throws IOException {
    Long sid = null;
```

```
try {
    //获取客户端的sid，也就是myid
    sid = din.readLong();
    if (sid < 0) {
        sid = din.readLong();

    if (sid < this.mySid) {
        //为了防止重复建立连接，只允许sid大的主动连接sid小的
        SendWorker sw =
    senderWorkerMap.get(sid);
        if (sw != null) {
            sw.finish();//关闭连接
        }
        LOG.debug("Create new connection
    to server: " + sid);
        closeSocket(sock);//关闭连接
        connectOne(sid);//向sid发起连接
    } else {//同样，构建一个SendWorker和
    RecvWorker进行发送和接收数据
```

```
        SendWorker sw = new
    SendWorker(sock, sid);
        RecvWorker rw = new
    RecvWorker(sock, din, sid, sw);
        sw.setRecv(rw);
```

leader 选举完成之后的处理逻辑

通过 lookForLeader 方法选举完成以后，会设置当前节点的 PeerState，要么为 Leading、要么就是 FOLLOWING、或者 OBSERVING
到这里，只是表示当前的 leader 选出来了，但是 QuorumPeer.run 方法里面还没执行完，我们再回过头看看后续的处理过程

QuorumPeer.run

分别来看看 case 为 FOLLOWING 和 LEADING，会做什么事情

```
@Override
public void run() {
    setName("QuorumPeer" + "[myid=" +
getId() + "]" +
            cnxnFactory.getLocalAddress());
```

```
while (running) {
  switch (getPeerState()) {
  case LOOKING:
  case OBSERVING:
  case FOLLOWING:
      try {
          LOG.info("FOLLOWING");

  setFollower(makeFollower(logFactory))
  ;
          follower.followLeader();
      } catch (Exception e) {
          LOG.warn("Unexpected
  exception",e);
      } finally {
          follower.shutdown();
          setFollower(null);

  setPeerState(ServerState.LOOKING);
      }
```

```
            break;
        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);
                }

        setPeerState(ServerState.LOOKING);
            }
```

```
            break;
        }
    //省略部分代码
```

makeFollower

初始化一个 Follower 对象

构建一个 FollowerZookeeperServer，表示 follower 节点的请求处理服务

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

follower.followLeader();

```
void followLeader() throws InterruptedException
{
```

```
//省略部分代码
 try {
     //根据 sid 找到对应 leader，拿到 lead 连接信息
     QuorumServer leaderServer = findLeader();
     try {
         //连接到 Leader
         connectToLeader(leaderServer.addr,
leaderServer.hostname);
         //将 Follower 的 zxid 及 myid 等信息封装好发
送到 Leader，同步 epoch。
         也就是意味着接下来 follower 节点只同步新
epoch 的数据信息
         long newEpochZxid =
registerWithLeader(Leader.FOLLOWERINFO);

         //如果 leader 的 epoch 比当前 follow 节点的
poch 还小，抛异常
         long newEpoch =
ZxidUtils.getEpochFromZxid(newEpochZxid);
         if (newEpoch <
```

```
self.getAcceptedEpoch()) {
                LOG.error("Proposed leader epoch "
+ ZxidUtils.zxidToString(newEpochZxid)
                        + " is less than our
accepted epoch " +
ZxidUtils.zxidToString(self.getAcceptedEpoch()))
;
                throw new IOException("Error: Epoch
of leader is lower");
            }
            //和 leader 进行数据同步
            syncWithLeader(newEpochZxid);
            QuorumPacket qp = new QuorumPacket();
            while (this.isRunning()) { //接受 Leader
消息，执行并反馈给 leader，线程在此自旋
                readPacket(qp);//从 leader 读取数据包
                processPacket(qp);//处理 packet
            }
        } catch (Exception e) {
            LOG.warn("Exception when following the
```

```
leader", e);
            try {
                sock.close();
            } catch (IOException e1) {
                e1.printStackTrace();
            }

            // clear pending revalidations
            pendingRevalidations.clear();
        }
    } finally {
        zk.unregisterJMX((Learner)this);
    }
}
```

makeLeader

初始化一个 Leader 对象，构建一个 LeaderZookeeperServer，用于表示 leader 节点的请求处理服务

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

leader.lead();

在 Leader 端，则通过 lead()来处理与 Follower 的交互

leader 和 follower 的处理逻辑这里就不再展开来分析，大家课后可以自己去分析并且画出他们的交互图