Zookeeper源码解读

导读

ZK源码在当前各种框架中不算是多的，只有几万行，但是如果没有一个正确的指导或者方法，去读源码也是非常枯燥，没有头绪，而且非常容易放弃了，笔者这一系列文章是笔者源码读下来的历程，梳理出来给想要读zk源码的后来者一个读源码的思路，以及更好的理解zk原理而写的。文章会以顺序模式进行，会对单机下一个命令是怎么执行的到集群模式下数据的同步、领导者选举等等核心功能进行一一讲解，并不会对每一个命令行进行细致的解说，因为命令执行的模式基本上是一致的，重复的讲解只是在充字数，浪费篇幅和时间。内容太多也会让新学者产生距离感，不利于顺利读下去。我们要学的是ZK中使用的多线程的思路，以及对分布式更深层次的理解，而不是机械的把源码全部看一遍。本系列所有的代码会按照整体->分解->整体这样的思路去进行，因此阅读本文的最佳方法是对照着源码一步一步的阅读。因为源码解读的过程中肯定会涉及到方法的跳入和跳出，会酌情对冗余代码删减，并且分割上层代码，力求真实性。

单机模式之一：

很多人面对一个框架，会比较迷茫的一点就是要从哪里入手，其实一个很好的入手点就是架构中bin目录下的可执行文件。为什么这么说呢，因为这里是程序的入口，也是我们最好下手的地方，那么我们就先从ZK的客户端开始” zkCli.cmd”，笔者用的是win10系统，“zkCli.sh”内容也差不多，并不影响使用Linux的同学阅读本文，ZK在Windows和Linux上的区别，也就只有这些可执行程序不一样的区别了，所以完全不用担心自己看的和笔者的不一样。无论是” zkCli.cmd”还是“zkCli.sh”我们打开并且要找到的其实就只有org.apache.zookeeper.ZooKeeperMain这句话，它就是程序的执行入口，那么我们就要从源码入手，看看当我们运行客户端的时候源码里到底做了什么事情。

那么首先，我们要找到这个ZooKeeperMain.java类或者文件，根据版本的不同，新版本”3.4.14”、”3.5.\*”和”3.6.\*”独立出来，比如3.4.14，在“\zookeeper-3.4.14\zookeeper-server\src\main\java\org\apache\zookeeper”下，老一些的版本(比如3.4.13)在\zookeeper-3.4.13\src\java\main\org\apache\zookeeper下。版本大体的核心代码差别不多，这里请根据自己的喜好下载。

其实我们运行脚本的时候就是运行的ZooKeeperMain这个类的main()方法，那么我们就打开ZooKeeperMain.java找到main()方法看看里面写了什么。

首先初始化ZooKeeperMain main = new ZooKeeperMain(args);，初始化结束，开始run，这里就开始接受并处理外部传递进来的命令了main.run();，最先调用的就是ZooKeeperMain(args)构造方法，这里面并没有做什么只是传了参数，然后建立了连接connectToZK(cl.getOption("server"));，进入看下连接是怎么做的， 发现这里先判断了一下命令是不是只读模式，然后new了zk的实例，那么我们进入ZooKeeper这个类的构造方法，这个ZooKeeper其实就是原生的Zk客户端的类。在这里 解析连接地址把我们传入的一个或者多个hostname做成一个list然后包装成HostProvider，然后生成了一个上下文的实例cnxn，在这之前它先调用了getClientCnxnSocket()这个方法，这个方法就是拿到ClientCnxnSocket这个抽象类的实例，我们进入这个看下，首先我们看到系统自定义的包名，String clientCnxnSocketName = System.getProperty(ZOOKEEPER\_CLIENT\_CNXN\_SOCKET); 如果clientCnxnSocketName是空，就把ClientCnxnSocketNIO 这个类的名字给它clientCnxnSocketName = ClientCnxnSocketNIO.class.getName()，其实ZOOKEEPER\_CLIENT\_CNXN\_SOCKET这个属性的有默认值：zookeeper.clientCnxnSocket，这里两者拿到的都是一样的东西，因为ClientCnxnSocketNIO是ClientCnxnSocket这个抽象类的实现类上面的拿到了，也是ClientCnxnSocketNIO真正处理的，这里做了一保险验证。然后我们可以看到调用了newInstance()把ClientCnxnSocket的实例返回去出去了，此处很明显zk连接使用的socket对象其实是ClientCnxnSocketNIO提供的，并且zk底层的连接客户端和服务端用的是NIO的思想。然后把这个NIO的实例再传回到ClientCnxn，我们进入这个构造方法，进入this()，看到里面初始化了一些属性，里面有获取的配置文件的内容，计算的timeout时间，最关键的两个属性来了sendThread = new SendThread(clientCnxnSocket); eventThread = new EventThread();。到这里结局，我们就可以跳出去了，到cnxn.start();启动了。

在这个方法里，看到是启动了刚刚的两个线程，这两线程是做什么的呢？先去SendThread的run()方法里看下，clientCnxnSocket这个socket实例就是我们上面讲的NIO实例，state.isAlive()这个是判断clientCnxnSocket是不是存活的，如果这个state不是CLOSED 也不是AUTH\_FAILED那么就是存活的，在我们实例化的时候它是有一个默认值的NOT\_CONNECTED这也可以认为是存活的，其实就是在这里无限的循环，那么下面的if(!clientCnxnSocket.isConnected())就是判断连接了，以及是不是第一次连接(贴代码)如果是第一次连接那么就各种初始化，进行startConnect(serverAddress);而且这个方法里修改isFirstConnect的值，以判断第一次连接过了，下面我们看下是怎么连接的，进入startConnect(serverAddress)。(贴代码) clientCnxnSocket.connect(addr)重点就是在这里了，我们要找到ClientCnxnSocketNIO.connect()实现方法(贴代码)然后注册和连接(贴代码)。

到这里我们就可以总结下：

先启动了ZookeeperMain里面会

1. 初始化Zookeeper
2. 初始化ClientCnxn，在初始化过程中会，
3. 初始化两个线程SendThread和EventThread
4. Start()启动SendThread建立socket连接，成功后就可以传输数据了

这里就运行完了，后面的SendThread我们先不看，这里我们只讲连接，我们一层一层跳出来还回到main()，我们接着看run方法，有两个工具类要注意下Java的命令行工具：jline.ConsoleReader和org.apache.zookeeper.JLineZNodeCompletor用这两个工具才能识别输入的java语句，再往下到while循环，executeLine(line); 执行客户端输入的命令行，凡是输入的文字行都会送到这里来执行，点进去(贴代码)，着重就是processCmd(cl);处理命令，进入后发现就是一堆一场找到processZKCmd(co);，点进去，就发现这里有很多判断，看到这些判断就知道这些逻辑是干什么的，比如下面有个create: cmd.equals("create")，这里就是判断我们输入的命令是什么的地方，那么我们现在分析下，当我们创建一个节点这部分的逻辑是什么。进入这个if以后到zk.create(path, args[first+2].getBytes(), acl,flags);这句。进入(贴代码) “节点的验证单独写一篇帖子“ 重点就在cnxn.submitRequest(h, request, response, null);注意这里的cnxn就是我们之前说的NIO的实例，然后我们进入里面看下写了什么(贴代码)，还要重点解释下Packet packet = queuePacket()，进入以后我们发现其实所有的packet都会被加入到outgoingQueue里，那么我们先问题一个问题：什么时候取出outgoingQueue里的数据呢？跳出方法，发现此时被阻塞了wait，那么就必然有一个地方是唤醒的，到这里调用create其实在客户端就已经告一段落，需要等待服务器端返回才能接着往下走，那么我们先画个图总结一下：zookeeper客户端调用原理源码流程图.png最左边的图画下来03.44:37

单机模式之二：

上一篇 我们说到一个问题：什么时候取出outgoingQueue里的数据呢？其实我们上一篇略过了一个地方就是SendThread为什么略过呢？因为如果把这里一起写了可能会导致逻辑上不好理解。所以我这一篇就是重点讲解连接之后做了什么，既然如此我们还是回到SendThread.run()里面接着探究连接之后做什么。这里的try是非常有用的，而且还有第一次连接，为什么要睡一会儿呢？我们先按下不表，接着往下看连接成功做什么，首先判断有没有超时，过了这一段到真正连接的地方(贴代码从没有超时，连接成功if (state.isConnected())，因为外面是一个while所以这里其实是不断地去循环到这里，进入sendPing()贴代码，解释。再往下跳过只读模式的部分，到一个关键的方法来clientCnxnSocket.doTransport(to, pendingQueue, outgoingQueue, ClientCnxn.this)，其实到这里了，我们连接也成功了，数据也写入outgoingQueue里面了，最后就是网络传输要把outgoingQueue发送出去了。很明显这里就用到了outgoingQueue，还有一个pendingQueue，这也是一个很有用的queue，那么我们要点进ClientCnxnSocketNIO.doTransport()里面去(贴代码)，到doIO(pendingQueue, outgoingQueue, cnxn)里面去，这里面有两个重点，写就绪和读就绪，读就绪，我们先不看，先到写就绪里面(贴代码)。其实这里就很明白了，outgoingQueue是一个一个发送给服务端的指令queue，pendingQueue是等待服务端返回指令结果的queue，这两个里面存的都是packet。Pending就是已经发送出去了但是还没有结果的packect都存到这个queue里。读就绪(贴代码)，就是能从SocketChannel里读取数据了，就是说服务端传回数据了，可以读取了，进入sendThread.readResponse(incomingBuffer);解析服务端发来的数据流，进入这个方法贴代码，删除部分冗余代码。最终到finally这里，这里面有唤醒语句，和之前的submitRequest方法中的packet.wait()相呼应。finishPacket(packet);，进入贴代码，到这里我们客户端发送接受代码都结束了，接下来就开始要看服务端怎么处理这些来自客户端发送的数据了，那么总结一下zk客户端启动，键入命令后做了什么：

启动ZK

1. Zookeeper：初始化socket
2. Zookeeper：命令->Request->Packet->outgoingQueue
3. ClientCnxnSocketNIO.SendThread启动:

SendThread线程run():

while(true){

1. 如果socket没有连接，就回去连接，和重试机制，ping等等
2. 如果socket链接成功，服务端会发送一个ConnectRequest，接受服务器返回的ConnectResponse(Event, None)返回一个event=None的事件 doTransport()
3. 从outgoingQueue取packet，通过Socket发送出去，阻塞，同时如果说有需要等待结果的packet放到pendingQueue里 doIO()-写就绪
4. 从服务端收到数据，唤醒doIO()-读就绪

}

那么SendThread就做了这么一个非常重要的事情，画图贴上

EventThread

ClientCnxnSocketNIO

1. doTransport主要的发送方法
2. doIO传输数据

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.13--------------------------------------------------------------------------------------------------------

单机模式之三：服务器

服务器同样也需要找到一个入口，我们打开“zkServer.cmd“找到里面的入口QuorumPeerMain，一样找到main()方法，main.initializeAndRun(args);args这里接收到的就是配置文件zoo.cfg，进入，解析QuorumPeerConfig config = new QuorumPeerConfig();配置类，这里面的属性都是配置文件(贴代码)，config.parse(args[0]);解析配置文件，//解析内容到Properties类中，进入parseProperties(cfg);贴代码，这里只介绍一些重点的判断，选择性的贴一些代码，跳过一大堆集群的判断，到servers.putAll(observers);这里来解析完了以后退出去，继续QuorumPeerMain.initializeAndRun()，就走到ZooKeeperServerMain.main(args);我们看看这个类是怎么启动的，一直点进去到runFromConfig(config);(节选代码),跳入解析，然后到cnxnFactory.startup(zkServer);贴代码，start();贴代码这里要介绍NIOServerCnxnFactory.run()，zks.startdata();初始化数据贴代码，进去，zks.startup();开始启动，进入贴代码，这里要重点介绍一下ZK的请求处理器(RequestProcessors)的逻辑，这个逻辑和我们一般意义上的链表很像，所以权且把这个逻辑叫做处理器链。

在介绍处理器链的逻辑之前有一个前提要给大家讲解一下，那就是服务器接收请求的步骤，这个步骤可以说是处理器链的理论基础。

我们知道ZK会对事务进行持久化，但是内存中也有数据，这个数据叫DataBase其包含了DataTree ,DataTree又包含了DataNode，这个DataNode就是我们最终创建的节点。

ZK当接收到任何一个命令的时候，主要的流程有4步

1.创建事务日志，保存到我们指定的目录下面;

2.在某一个点打快照创建dataBase.dataTree.dataNode到文件; 服务器累计多个事务才会打一个快照

3.更新内存，操作内存中的DataTree;

4.以上都执行成功了，返回错误或者正确的信息出去

有了以上逻辑以后，我们可以介绍一下处理器链有什么用，在单机模式下有三种处理器

Prep->做认证，生成txn事务对象，转交

Syn->事务对象持久化(生成事务文件)，打快照，转交

Final->做内存更改，处理事件响应啥的都是在这里弄的，返回客户端响应

要注意记住一点：第一个请求process是NIOServerCnxn中被调用的，也就是处理器链在服务端。

整条链就是这样的PrepRequestProcessor.next=SyncProcessor.next=FinalRequestProcessor

第一个处理器是PrepRequestProcessor，第二个就是syncProcessor，服务端收到客户端的socket以后取出其中的packet然后存到queue中，再由firstProcessor.run()从queue取出request，请求处理器链表来处理这个request，这个queue就是PrepRequestProcessor类中的linkedqueue：submittedRequests，但是这里要记得，这之前我们已经开启了NIOServerCnxnFactory线程，所以我们先去NIOServerCnxnFactory.run()，贴代码，到c.doIO(k);贴代码readPayload();正式加载数据，进入贴代码，再到readRequest();，再到zkServer.processPacket(),if (h.getType() == OpCode.auth) {//这个auth其实就是命令，addauth命令，往下跳过这里，因为后面acl还要讲，再到 if (h.getType() == OpCode.sasl) {//也不是sasl命令，到else里，贴代码submitRequest(si);//拿出后，提交request，介绍一下boolean validpacket = Request.isValid(si.type);这个valid方法，然后到firstProcessor.processRequest(si);，这个是个接口，有很多不同的实现方法，但是此时是PrepRequestProcessor实现的，所以我们就去PrepRequestProcessor. processRequest(si);里看下，贴代码submittedRequests.add(request); 这里的submittedRequests就是服务端的queue，那么谁来处理这个请求呢，我们应该还记得PrepRequestProcessor这个类是个线程类，那么必然有run()，而且刚才我们还把firstProcessor,start(),所以我们就转去PrepRequestProcessor.run()，看下里面怎么做的，转到run()贴代码，再到pRequest(request);//这里重点贴代码，首先还是判断是什么命令，我们还是以Create命令作为例子，介绍pRequest2Txn()，进入，介绍case OpCode.create:贴代码，getRecordForPath(String path)介绍，跳出来到case OpCode.create接着贴代码介绍，addChangeRecord(parentRecord)，介绍，然后break跳出去到pRequest()，直接到下面request.zxid = zks.getZxid();nextProcessor，初始化的时候还记得是first指向next再指向finalProcess的， nextProcessor就是咱们的SyncRequestProcessor，我们进入看下这里调用的SyncRequestProcessor.processRequest()里面写了什么nextProcessor.processRequest(request);，贴代码，我们要记得这个process也是一个线程，所以我们也转到run()方法中，把if (si != null)写一写，zks.takeSnapshot();//打快照，我们进入看下，快照的作用是什么呢？然后服务器启动的时候不可能直接从内存拿数据，服务器启动的时候也有一个逻辑：从文件取出数据加载到内存DataBase。所以Sync这个处理器就是持久化和打快照用的。最终也会调用nextProcessor.processRequest(si);//所以我们去看FinalRequestProcessor.processRequest();，synchronized (zks.outstandingChanges)贴代码，再到rc = zks.processTxn(hdr, txn)，进入贴代码，rc = getZKDatabase().processTxn(hdr, txn);//走进这里来，进入贴代码，一路进入，还是以create为例子，createNode()贴代码，跳出去，到此位置，事务也结束了，持久化也做了，更新内存也做了，就剩返回结果了，跳到FinalRequestProcessor.processRequest();的rc = zks.processTxn(hdr, txn)往下，try {//最后就是返回结果了，所以这里又有了switch，case OpCode.create: {//如果是create，贴代码跳出，try {//最终，这里调用的是socket层的东西，所以我们就又跳转到了NIOServerCnxn(ServerCnxn)里面，cnxn.sendResponse(hdr, rsp, "response");又回到了NIO里面去了，贴代码

总结一下贴图

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.15--------------------------------------------------------------------------------------------------------

单机模式之四：单机Event之客户端流程

前面几篇我们已经讲了单机模式下ZK客户端和服务端的启动，交互，SendThread和处理器链等等，我们还剩下了，EventThread，以及ACL，closeSession等等，那么我们这一篇就来说说EventThread这个重要的线程做了什么

首先我们先问一个问题，服务端除了接受客户端的请求以外，会不会做一些别的事情呢，比如主动推送一些内容给客户端呢？

当然是有的，最明显的就是Watch事件，比如客户端1在NodeA上绑定了一个监听器，然后客户端2改变了NodeA，那么客户端1怎么直到NodeA改变了呢，这里就需要服务端推送这个change给客户端1，其实这个推送就是和EventThread有关系。

首先我们还是要看接受请求的这里，因为不管是一个正常的结果返回还是一个监听事件，都是通过一个结果返回出去的，所以这部分客户端接受服务端返回的逻辑其实是在一个地方的，那么我们还找到SendThread，入口还是run()，这里上面几篇已经详细说过，就不再赘述，直接在While(state.isAlive())找到clientCnxnSocket.doTransport()，接着直接找到doIO(),，找到都就绪if (sockKey.isReadable())，一直走到sendThread.readResponse(incomingBuffer)，直接找到if (replyHdr.getXid() == -1) //xid=-1，WatcherEvent的逻辑，执行事件，这里贴上代码，到eventThread.queueEvent( we ); 这里是把服务端告诉我的事件存到这个线程的一个队列里。

那么通过代码我们总结一下：

服务端抛出来的事件，客户端会存在EventThread队列里waitingEvent里面。

进入eventThread.queueEvent( we )看看里面做了什么，贴代码WatcherSetEventPair pair = new WatcherSetEventPair(watcher.materialize(event.getState(), event.getType(),event.getPath()),event);和waitingEvents.add(pair); 那么看到这里这个waitingEvents队列是哪里处理的呢？我们说过这是个线程EventThread是个线程，那我们去找到EventThread.run()方法，你看run方法就把这个东西给取出来了Object event = waitingEvents.take();//这里就把上面的pair作为event取出来了，并且作为一个WatcherSetEventPair绑定上了，那么什么是pair？这里就是绑定事件，由watcher.materialize绑定一个事件，那么这个watch是什么意思呢，这个时候我们就得看下我们怎么去绑定的，

那么我们拿getData方法为例子看下，事件是怎么绑定的，贴getData的代码，我们可以看到拿/data的数据的时候就绑定了一个监听器，那么我们点进去getData(final String path, Watcher watcher, Stat stat)这个方法，跳到方法，看代码之前先想一个问题，比如我给nodeA绑定一个事件，记作nodeA:new Watch(), 客户端在绑定代码之前需不需要把nodeA:new Watch()告诉给服务端呢？换句话说需不需要把某个节点对应的某个watch告诉给服务端？答案是不需要的，因为监听器只要客户端存起来就可以了，服务端只关心哪个节点发生了什么事件，然后抛出去好了，客户端只要接收到这些事件以后检查自己有没有定义监听这个事件，如果定义了那么就触发自己定义的事件就可以了，这个信息只要客户端保存就好了，那么我们看代码是不是我们分析的，首先我们看传入了一个路径，一个new的watch，那么它又没发送给服务端呢，贴getData()里面的代码，request.setWatch(watcher != null);它只在这里判断了一下，只是一个true/false并没有传入实际的对象，那么为什么要把有没有watch告诉服务端呢？这里要联合服务端去看，我们先放下，先记住这里。但是这也说明了客户端并不需要把整个watch都给服务器，他只是在WatchRegistration这里处理了一下，new了一个注册，然后又把wcb传入了ReplyHeader r = cnxn.submitRequest(h, request, response, wcb);这个submit，进入贴代码，在这里封装成packet传给服务器，进入，贴代码，Packet packet = queuePacket(h, r, request, response, null, null, null, null, watchRegistration)，还记得这个packet是做什么的吗，之前讲过这个packet就是最终客户端要传过去的数据都会封装成packet，这里就有点问题了，不是说不会发给服务端吗，为什么还会把这个watchRegistration封装到packet里面呢？进入 queuePacket(h, r, request, response, null, null, null, null, watchRegistration)，贴代码，其实这里并不矛盾，就算是你放到了packet里服务端还是没有用，因为我们的packet最终还是要使用的，就是接受结果的时候，上篇我们说到了Sendthread，其实这个线程走到了最后会走到finishPacket(packet)这个方法里，先记住这个packet，我们先去ClientCnxn.Sendthread. finishPacket(packet p)里面，首先看到if (p.watchRegistration != null)如果watchRegistration不是null了，就首先进行注册p.watchRegistration.register()，进入看看，这个时候就会把刚刚设置的watch给存到一个map里，他就是在这里接受结果并注册的，这里就完成了客户端事件的注册。当一个事件发送给我们的客户端以后他是怎么触发的呢，这个就需要到到客户端xid=-1那里看了，找到这个ClientCnxn.readResponse()那里if (replyHdr.getXid() == -1)，贴入这个if的代码，再走到eventThread.queueEvent( we );这里，点进去，最后加到一个队列里waitingEvents.add(pair)，但是这里有一个方法比较重要WatcherSetEventPair pair = new WatcherSetEventPair(watcher.materialize(event.getState(), event.getType(),event.getPath()), event);，这时候就可以研究这个方法是做什么用的了watcher.materialize(event.getState(), event.getType(),event.getPath())，传入的参数是状态，类型，节点等等，重点在于事件的地址，类型都传进去了，进入，贴代码，然后到case NodeCreated:这里来，dataWatches这个map就是刚刚我们看的那个注册的map，然后addTo(dataWatches.remove(clientPath), result)他会remove掉刚才的那个值，这个值就是clientPath所绑定的监听器对吧，并且把remove掉的值放进result里面，然后就直接return出去了，在介绍分析watcher.materialize完毕以后我们知道，这个方法返回的就是event的结果而已，返回到上一层，waitingEvents.add(pair)

然后再到EventThread.run()方法里, processEvent(event);//执行事件，进入贴代码，然后这个watcher.process(pair.event)，进入后的process就是你自己要去实现的那个。到这里客户端处理watch的流程就结束了

总结一下：

事件的真正机制：

客户端有watcher，这个类中定义的有create，delete，change等等事件类型，服务端发生改变以后会抛出事件给客户端，然后客户端拿到事件后，就会自己查看自己这里定义的事件都有什么类型，有没有定义，如果定义了那么触发这个定义好了的事件就可以了

客户端Watcher 的注册流程

先发送请求，如果请求成功了，再注册Watcher到我们使用的客户端里。就是说当使用getData()的时候，客户端先发送请求，如果这个请求在服务端成功了，再返回来注册客户端event，保存起来。

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.16--------------------------------------------------------------------------------------------------------

单机模式之五：单机Event之服务端流程

为了讲解的更清楚，我们就以Set操作作为一个例子讲解，所以这篇文章也是再说ZK的Set命令是怎么触发一个事件的

既然触及到服务端的代码了，我们先找到服务端真正核心的代码是NIOServerCnxnFactory.doIO().readPayload().readRequest().processPacket(this, incomingBuffer). submitRequest(si)这里就会进入处理链，首先进入firstprocess: firstProcessor.processRequest(si)，然后进入到PrepRequestProcessor类的实现方法processRequest()，又加到了队列里，然后又到了PrepRequestProcessor.run()方法里, submittedRequests.take()取出请求开始处理，经过验证以后调用pRequest(request)方法，在里面判断命令，走到OpCode.setData，到pRequest2Txn()方法里，进入while(true)判断OpCode.setData，在这里验证并生成事务(日志和数据)，然后到最后交给下一个处理器链对象SyncRequestProcessor:nextProcessor.processRequest(request)，到里面以后发现，这里也是加入了一个线程队列里queuedRequests.add(request)，所以我们又要去SyncRequestProcessor类的run()方法里，我们发现了另一个while (true)以及各种验证、持久化、打快照等等到最后又交出去了nextProcessor.processRequest(si)，到了这里就是FinalRequestProcessor.processRequest(si)的内容了，之前分析我们将在这里更新内存以及触发事件，进入后经过一系列的取出操作我们找到这样一行zks.processTxn(hdr, txn)进入，接着进入getZKDatabase().processTxn(hdr, txn)，到datatree中去dataTree.processTxn(hdr, txn)，这里我们发现了又一个switch，找到OpCode.setData中的setData()，进入看是如何修改的，进入后通过synchronized的修饰，改完内存数据最后发现dataWatches.triggerWatch(path, EventType.NodeDataChanged)，看到这里大白天下，此时才刚刚触发事件，触发这个传入的node(path)的NodeDataChanged事件。

一路进入triggerWatch()，我们发现了这样一句话watchers = watchTable.remove(path);这里watchTable调用了remove()那么这里一定是服务端保存的东西，从这里取出watchers，经过一系列的确定path操作在最后的for循环里面，调用w.process(e)这里执行，而这个w.process(e)方法就是我们自己外面写的process()实现方法，自此一个set语句的流程就完成了。

但是还有一个问题，我们的watchTable是什么时候存进去数据的呢？查看声明看到watchTable是一个key=path, value=HashSet<Watcher>的一个map，默认情况下set执行以后会调用一个getdata去显示出来，那么我们去get里面找找，一样的处理链逻辑找到PrepRequestProcessor类中pRequest()找到case OpCode.getData分支，我们发现第一个处理器什么都没做，只是检验了一下session，到nextProcessor.processRequest(request)也就是SyncRequestProcessor里，但是我们之前说过Sync这个类是持久化类，所以很快就走到了FinalRequestProcessor.processRequest(si)，在方法里找到case OpCode.getData: lastOp = "GETD"; 由于getdata不需要更新zk内存里的东西，所以直接构造了一个response返回出去了，在最后我们找到byte b[] = zks.getZKDatabase().getData()，这个方法里传进去了一个判断“getDataRequest.getWatch() ? cnxn : null”，如果我们设置/系统设置了一个监听器，那么我们用cnxn(ServerCnxn)这个对象，这里是不是突然明白了什么，我们刚才getData()传入的watch其实是固定的ServerCnxn类对象。ServerCnxn类是服务端连接的类，连接没了这个类对象也会消失，我们进入这个类注意到这个implements了Stats和Watch两个接口，是他们的实现类，相当于服务端抽象出来的监听器，不管客户端定义传来了什么样的监听器，服务端最终都用ServerCnxn类去当作监听器注册，那么我们进入getData()方法看下里面又写了什么，一直进入发现又跑回了DataTree类，找到DataTree.getData()，我们发现里面有dataWatches.addWatch(path, watcher);看到这里传入的watch监听器被加到了WatchManager类对象里了，那么我们就进入addWatch()看看里面又写了什么，进入后我们终于看到了watchTable和watch2Paths，进一步分析，服务器接到的watch就是在这里被加入这两个map的。自此我们可以得出结论：服务端注册watch的时候注册的是一个ServerCnxn对象，然后对应的节点也会被存起来作为一个map返回出去。就是说只要客户端只要给node注册了一个事件，服务端都会给这个节点注册并保存这样的map信息：nodename=ServerCnxn(Watch)，然后交给客户端执行process方法达到watchevent的执行。我们知道Watch都有process方法，那么ServerCnxn就一定也有这个方法，我们去里面看下，我们发现ServerCnxn中的process方法其实又被NIOServerCnxn类实现了并且调用sendResponse(h, e, "notification")发给了客户端，这样就又转回来了，这一系列的操作就是用客户端和服务端一起通过NIO的逻辑实现的。到这里，一个完整的Set命令的事件触发流程就结束了。

总结：

客户端首先(getdata)的时候，首先要注册节点名字，watch实现类，这些信息会保存在一个Pair< Node: List<new Watch()>>对象里面对应一个具体的watch实现类，这个注册的动作在客户端是发生在服务端返回信息之后进行注册的。服务端收到请求后，也会注册但是服务端会把节点名字，watch这些信息保存在一个map里，而且watch注册的是一个NIOServerCnxn对象，然后返回给客户端。

服务端：当去改变内容的时候(setData)，服务端会在最后触发事件，调用NIOServerCnxn.process方法，这个process方法就是把这个节点名字，事件类型这些信息通过socket发送给客户端，客户端接收到以后就直接去pair去找，对应的node，读取并移除(remove)绑定的事件，然后触发这个事件，这也是为什么绑定的事件只能被触发一次。

以上整合就是：

客户端触发事件的时候，首先要注册节点名字，watch实现类，这些信息会保存在一个Pair< Node: List<new Watch()>>对象里面，去对应一个具体的watch实现类。这个注册的动作在客户端，但是发生在服务端返回信息之后，才进行注册的。服务端收到请求后，也会注册但是服务端会把节点名字，以及watcher这些信息保存在一个map里，而且watch注册的是一个NIOServerCnxn对象。然后服务端会在最后触发事件，调用NIOServerCnxn.process方法，这个process方法就是把这个节点名字，事件类型这些信息通过socket发送给客户端，客户端接收到以后就直接去pair去找，对应的node，读取并移除(remove)绑定的事件，然后触发这个事件，这也是为什么绑定的事件只能被触发一次。

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.17--------------------------------------------------------------------------------------------------------

Close Session:和临时节点有关系

当我们输入quit时，也是通过ZooKeeperMain中的main方法进入main.run()，找到最后进入processCmd(cl)，进入processZKCmd(co)找到quit的判断语句if (cmd.equals("quit"))，发现调用了一个zk.close()，进入后发现cnxn.close()《cnxn还记得这个是ClientCnxn(NIOClientCnxn)的对象，也是通过这个封装的socket》，进入close()方法，我们发现又是一个提交submitRequest(h, null, null, null);此时交给了服务器端，那么我们就去服务器端查看。通过刚才的请求我们直接找到请求处理链的第一个PrepRequestProcessor，找到run()找到pRequest()中的case OpCode.closeSession:同样里面调用了pRequest2Txn()方法，继续找到case OpCode.closeSession，然后我们发现了这样一句话HashSet es = zks.getZKDatabase().getEphemerals(request.sessionId)，这不就是放临时节点的set，我们知道临时节点是和session id绑定的，所以这里就是要拿到当前id的临时节点，进入getEphemerals看看这里拿出了什么，进入后发现这里只是把所有的节点名字(node)拿到。接着往下走发现有个队列outstandingChanges被锁住了，并且在第二个for循环里es被使用了，这里我们看到有一个addChangeRecord()方法进去后神奇发现了zks.outstandingChanges.add(c);这不就是我们被锁住的list吗，这里的逻辑其实这样的，当我们有临时节点的时候，就会循环es(如果没有临时节点，这里就会跳过，因为es就是存放临时节点的，当sessionclose时候，没有临时节点就不需要删除，直接关闭就好了)，然后加上一个改变的记录，在create或者set或者delete中我们可以看到PrepRequestProcessor都是这样做的都会先创建一个改变记录，而后面的处理器(final)会用到这个list，我们继续分析，new ChangeRecord()这里，把stat置null，拿到path2Delete这个节点名字，然后统一加到outstandingChanges这个队列里，于是我们又到了SyncRequestProcessor一样的逻辑略过，接着到FinalRequestProcessor.processRequest()，往下走又到了rc = zks.processTxn(hdr, txn)这个方法里，发现执行事务rc = getZKDatabase().processTxn(hdr, txn)，那我们进入后又走到了dataTree.processTxn(hdr, txn); 继续进入发现这里还有一个switch我们找到OpCode.closeSession: 这个分支，找到了killSession()这个方法，继续进入突然发现ephemerals.remove(session) 临时节点就在这里根据session被全部拿出来之后，调用deleteNode(path, zxid); 删除临时节点，然后我们跳出接着final的processRequest方法往下走，再往下找到一个flag: boolean closeSession = false;那么就一定有一个地方置这个flag为true，然后在调用closeSession方法把客户端关闭，接着往下，果然我们在紧接着的switch里找到了closeSession = true;跳出，看看接着发生了什么，跳出以后就找到了if (closeSession)这个判断，最终调用cnxn.sendCloseSession();关闭session并给服务端发去关闭的响应

总结：

Quit，对应的操作叫做closeSession，其对应的这一步操作也需要做持久化到sync里，但是这个持久化做的是删除节点的那一步，然后到final那里更新内存删除临时节点，这就是整个close session发生的事情。为什么临时节点要做删除的持久化？因为你在创建的时候肯定做了持久化，所以关闭session删除临时节点就要做个对冲，把持久化冲掉，从这一点看临时节点和普通的节点没什么区别，唯一的区别在于临时节点和session绑定，做到session死亡临时节点自动删除，而普通的节点不绑定session，是否删除和session无关。

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.18--------------------------------------------------------------------------------------------------------

ACL:权限管理和验证

给xiaoming管理删除读的权限SetAcl /node auth:xiaoming:123123:adr但是没有写权限，那么xiaoming就不能写操作，这样当setData修改节点的值得时候，肯定有一个checkAcl得操作检查当前账号有没有权限。然后getAcl查看权限，还有添加权限addauth digest xiaoming:123123。也不仅仅是账号名字，也可以是ip，比如SetAcl /node auth:192.168.0.1:adr，中间得账号名字可以替换为ip，这就是给了客户端ip为192.168.0.1的adr权限

还是找到ZookeeperMain定位到cmd.equals("addauth")然后执行zk.addAuthInfo(args[1], b);，进入，发现authInfo.add()给自己添加上，然后仍然是加到队列里queuePacket()然后和别的命令一样传到服务端，但是这里要注意，我们输入的是addauth，但是封装的是OpCode.auth验证的操作，所以到服务端，我们要取看OpCode.auth。按照之前的命令传递流程，我们先去NIOServerCnxnFactory找到c.doIO(k)再到readPayload()再找到readRequest();在这个方法里加到packet里面processPacket()进入找到OpCode.auth，发现直接进入了，并没有往下走到处理器链的第一个Prep，接着看AuthPacket authPacket = new AuthPacket();先new一个authPacket对象，然后从authbuffer里读出来

ByteBufferInputStream.byteBuffer2Record(incomingBuffer, authPacket);然后根据不同的命令生成不同的AuthenticationProvider，这个接口有三个继承类，DigestAuthenticationProvider，IPAuthenticationProvider和SASLAuthenticationProvider会根据传入来生成不同的实例，比如我们传入的是digest则生成DigestAuthenticationProvider实例，以digest为例，往下走ap.handleAuthentication()这个方法就是调用的digestAP的实例，我们进去DigestAuthenticationProvider. handleAuthentication()实现方法里去，发现了一个签名的方法generateDigest(id)，这就是之前我们看到的ServerCnxn.authInfo变量在声明的时候中的Id类中的签名id，这个id就是generateDigest(id)这个方法生成的，进入发现就是一些字符串转换和加密。出来再往下digest.equals(superDigest)，“superDigest”这里就是检查超级管理员的地方，这里进入发现是读取配置的地方，也就是说我们启动前配置的超级管理员，就是在这里被加入服务器的。再往下cnxn.addAuthInfo()后面就没有了，也就是说自此addauth命令其实已经结束，就是把addauth后面的信息存到authInfo里，跳出继续走到cnxn.sendResponse(rh, null, null)返回给客户端，addauth命令彻底结束。

要注意：

当执行addauth ip 192.168.0.1后，就运行到IPAuthenticationProvider.handleAuthentication()但是这里要注意，这里拿到的将会是一个Scheme是ip，而id是客户端host地址的一个列表元素，而你自己写入的192.168.0.1是无效的。所以addauth ip 192.168.0.1是无效的。

通过上述add的代码分析，我们输入的信息都被存入了ServerCnxn.authinfo这个对象存里面，而authinfo对象的就是账户名字和密码，构造的id这些信息，但是这个什么时候用的，或者有什么用呢？那么我们就得看setACL这个方法，但是我们做这些操作的时候会首先验证权限，setACL其实就是对某一个节点存一个ACL的记录，所以我们得先关注checkACL方法。

找到服务器处理链PrepRequestProcessor. pRequest()，找到OpCode.setACL，进入pRequest2Txn()，再找到OpCode.setACL，略过拿数据验证去重，我们找到checkACL()这里传入了四个参数去验证，zks连接信息，acl是节点已经存在的节点规则，ZooDefs.Perms.ADMIN当前这个操作需要的权限，这个是个位操作的权限验证参数，验证使用异或与，request.authInfo是需要被验证的id信息，可能为多个，但是这样只要有一个supper用户在里面，所有的验证都会被跳过，这里是验证做的不好的地方。

往下走过几个系统和空验证，到for循环里发现首先验证节点的规则有没有我们要的权限，然后验证账号和ip是不是符合要求，然后返回。

但是这里有一个很坑的地方，比如我们本身有zhang:ad，然后我们添加两个用户，添加了xiao,添加了ming，由于zhang这个账户已经存在，所以如果我们不加上zhang，验证就不通过，就必需加上zhang，然后我们想要只修改xiao的账户给adrw的权限，用(也只能用)setAcl /node auth:xiao:123123:adrw，那么运行后我们会发现：xiao，ming，zhang三个账号同时都有adrw的权限这就是比较坑的地方，而这个功能就是在上面fixupACL()里面做的。里面有详细的原因。这里应该跟着视频调试一下

权限类ZooDef.java，里面定义的权限的位移个数，在这里面我们能找到rwcda的权限对应的16进制，在ZooDef.Perms()里面

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.20--------------------------------------------------------------------------------------------------------

集群模式：

集群模式下面有如下角色

Leader 只有leader能处理写请求，如果f/o收到写请求，就会转发给leader

Flower处理leader的提议，转发写请求，参与选举

Observer 处理leader的提议，转发写请求，不参与选举

所以启动的顺序就是

1. 加载配置
2. 启动socket
3. 选举leader
4. 同步数据
5. 初始RequestProcessor
6. 返回Response

集群服务器启动步骤：

首先还是从脚本里看到走QuorumPeerMain类找到main()，进入initializeAndRun()，看到config.parse(args[0]);解析配置文件，进入，往下cfg.load(in);把文件流装载成为Properties类实例，再往下parseProperties(cfg)这个方法就是核心的解析配置文件的方法，进入。发现里面是一堆我们写在zoo.cfg里面的参数，这些参数就是在这里解析并赋值的，这里有必要说的就是peerType这里只有两个值"observer"和"participant"，就是说如果你配了observer就是观察者模式，如果配了participant(参与者)就是follower，如果不配默认就是follower因为peerType声明的默认值就是LearnerType.PARTICIPANT，然后往下key.startsWith("server.")就是我们配置的server.1=localhost:2887:3887，这里会识别到server.开头的字符串，并以此决定是单机还是集群，如果发现了这个字段，就走下去，解析hostname，port之类的信息，还有是不是observer等等，再往下走，我们发现观察者和服务器存储的地方是不一样的observers.put()到这个map里，follower则放到servers.put()这个map里。

接着往下发现quorumVerifier = new QuorumMaj(servers.size())这里，QuorumMaj这个类就是过半机制验证的类，我们看下这个类，发现这个类实现了QuorumVerifier类，从名字看就知道这是个集群验证类，我们看到进入QuorumMaj类的构造方法只有this.half = n/2，这个half也在下面的一个containsQuorum()方法中使用了，当传入的set大于half的时候return (set.size() > half)一个true说明这是一个集群，而这个类传入的是servers.size()，这里可以看到取得是server中的一半，并不包含observer，因为上面说了观察者放到了observer这个map中，那么我们可以认定containsQuorum()这个方法就是验证集群用的，我们调用这个方法传一堆服务器进去，然后就去判断传入的服务器的数量是不是大于配置的集群的一半，如果过半就是说验证通过集群可用，如果不过半就不通过这个集群不能用，而且这个过半是指follower的服务器过半，并不包含observer，后面我们会经常看到containsQuorum()这个方法的使用的。

再往下我们发现这句servers.putAll(observers)，原来quorumVerifier验证器构造完以后才会把observer放到我们的servers的map中，此时这里才包含了我们所有的服务器了。

然后往下取myid，再往下发现这样一句话observers.containsKey(serverId) ? LearnerType.OBSERVER: LearnerType.PARTICIPANT，这里很明显再次验证observer，如果发现server.1=localhost:2887:3887:observer，如果发现myid对应的后面有关键字observer那么就把这个myid重新做成OBSERVER，而紧跟着的if语句会更新peerType，说明observer是由配置server.1=localhost:2887:3887:observer决定的，而不是文件中peerType=observer决定的，即便配置成peerType=participant，后面host只要配置了observer，还是会被刷新掉的，自此解析配置文件结束。

跳出来，走到if语句config.servers.size() > 0就会走到集群模式里面，那么就到了方法runFromConfig(config)这里，可以看到这里做了一个socket连接cnxnFactory，然后打开了socket准备接收客户端的请求cnxnFactory.configure(config.getClientPortAddress(),config.getMaxClientCnxns())，但是这里只是打开了客户端的请求，并没有打开和别的server的连接。下面quorumPeer = getQuorumPeer() 新构造一个集群中的服务器，再下面一堆set方法设置参数，设置完毕以后，初始化quorumPeer.initialize();，然后我们看quorumPeer.start()进入，发现loadDataBase()这里和单机模式一样，从快照里导出数据到内存，进入zkDb.loadDataBase()进入，看snapLog.restore()方法进入，看到snapLog.deserialize(dt, sessions) 把快照的日志反序列化出来数据存到datatree(参数dt)中，然后我们继续进入fastForwardFromEdits(dt, sessions, listener)方法中，具体的解释在代码里……此处略过……，

跳出回到quorumPeer.start()方法，cnxnFactory.start();开启一个线程，接受客户端的请求，startLeaderElection();领导者选举方法我们后面再说，再后面super.start(); 这里就会启动QuorumPeer本身的run方法，因为QuorumPeer本身就是一个线程类，那么我们就去QuorumPeer本身的run方法里面看下。

我们着重看run()里面的while (running) 这里其实就是一个永久循环，但是我们要关注getPeerState()//紧接着就是去取PeerState，这个PeerState一共有四种：LOOKING：代表还没有确定好角色，就是正在进行领导者选举，如果选举完了，PeerState状态会改变为下面两种之一(FOLLOWING/ LEADING)；FOLLOWING：服务器确定是一个follower；LEADING：服务器确定是一个leader；OBSERVING：服务器确定是一个观察者。

那么我们就先来看，如果选举为Leader后，会做什么操作，到case LEADING:这里，首先看到makeLeader(logFactory); 当一台服务器选为leader后，先new一个leader，然后调用lead()方法，这个才是核心方法，进入看leader.lead();，我们知道当选为leader后要做什么呢？首先leader会跟很多个follower和observer，我们统称为learner。作为leader肯定要和各个服务器通信，就是用的配置文件中的第二个端口号，首先先开启socket为了连接各个learner，然后会给每个单独learner开启一个线程，这个线程会不断地从socket里面获取数据，处理，传输等等任务。去代码查看。

进入lead(), 找到cnxAcceptor = new LearnerCnxAcceptor();看到了创建线程类，而后有个start()方法，老规矩，我们要去LearnerCnxAcceptor.run()里看看里面写了什么，我们看到Socket s = ss.accept();开启socket等待连接，然后new LearnerHandler(s, is, Leader.this);，很明显这个LearnerHandler也是一个线程，那么我们大概有这样一个概念了，还记得之前说的每一个socket都会单独开启一个线程，这个外面是一个线程，然后这个线程会接收socket连接，一旦有一个socket来了ss.accept()，那么Socket s就会单独new一个线程LearnerHandler fh.start();，从名字上看线程(LearnerHandler)是为了处理leader和follower数据交互的。那么就有就一个问题，作为follower到底会不会主动和leader连接呢，所以我们就看下follower里面又写了什么，找到QuorumPeer.run().case FOLLOWING:我们可以看到几乎是一样的逻辑，先makeFollower就是new一个Follower，然后调用follower.followLeader();，那么我们进入看看follower会不会发一个请求给Leader，

进入后我们发现它会QuorumServer leaderServer = findLeader(); 首先找到哪个是leader，然后再用connectToLeader()方法，去连接，进入我们看到sock.connect(addr, self.tickTime \* self.syncLimit);给leader地址address，发送一个socket请求，主动连接Leader。

那么leader接收到这个socket请求以后，又会干什么呢？我们就需要转到LearnerHandler.run()里面看下leader的动作，第一步肯定是建立一个socket，然后我们看到ia.readRecord(qp, "packet")，这不就是立刻读取了数据，所以说明learner首先要向leader发数据，那么我们返回去查看Learner.followLeader(),发现里面果然有方法发数据registerWithLeader(Leader.FOLLOWERINFO)，进入查看发现qp = new QuorumPacket();构造一个数据包，然后writePacket(qp, true);写数据，所以learnerHandler那边就可以read数据了，

建立完socket就会发送一些following，observing这些信息，还包含了选举第几轮的信息(Epoch), leader会同步各个服务器的epoch，并选出一个最大的作为当前的版本，这个选举就是用leader.getEpochToPropose()选举出来的，……代码详细注释略过……

有了这个新的有了这个新的Epoch以后就要通知各个follower一个LeaderInfo，就是在new QuorumPacket(Leader.LEADERINFO, ZxidUtils.makeZxid(newEpoch, 0), ver, null);这里，然后写出去oa.writeRecord(newEpochPacket, "packet");，到这里我们去Follower里面找到followLeader().registerWithLeader()，进入后在Learner. registerWithLeader()就直接readPacket(qp);，如完以后拿到数据又会构造一个ackNewEpoch(QuorumPacket ackNewEpoch = new QuorumPacket(Leader.ACKEPOCH, lastLoggedZxid, epochBytes, null);),然后又写出去writePacket(ackNewEpoch, true);，转到LearnerHandler.run()里面读出来ia.readRecord(ackEpochPacket, "packet");接收到ack数据，这样就选举出了一个最新的Epoch逻辑，这里就是我们说的leader和follower的ack机制。那么我们新一届的Epoch已经确定了，下面就是要做数据同步了。

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.22--------------------------------------------------------------------------------------------------------

同步数据：

如果新的follower加进来

1. leader直接发快照给follower，加入内存就可以了
2. 如果快照不是最新的怎么办？我们去服务(Server)端的FinalRequestProcessor. processRequest().zks.getZKDatabase().addCommittedProposal(request);方法里去，……分析代码里……，Follower还可以去committedLog里面拿到最新的事务

只要知道上面两点，我们再回到LearnerHandler.run()里面接着去看，我们找到rl.lock();里面的内容发现，这里就说到的committedLog这个链表的地方了，往下走我们看到拿到了maxCommittedLog和minCommittedLog，然后proposals = leader.zk.getZKDatabase().getCommittedLog()拿到了最新的事务链表committedLog，紧接着是一个if(peerLastZxid == leader.zk.getZKDatabase().getDataTreeLastProcessedZxid())，这里是说如果这个事务id和leader里面当前的最新的事务id是一样的，那么既不需要进行什么逻辑处理。如果说proposals(committedLog)里面是有值的，那么就判断peerLastZxid是不是在服务器zxid之间的内容，if ((maxCommittedLog >= peerLastZxid) && (minCommittedLog <= peerLastZxid)) 比如minCommittedLog=100，maxCommittedLog=400，传过来的peerLastZxid是200，那么就需要同步201到400这些事务，这个操作就叫做DIFF(packetToSend = Leader.DIFF)，往下prevProposalZxid = minCommittedLog这里同步最小值，再往下有一个for循环，紧接着的if语句我们看到这里就是把大于的min的事务的zxid找出来，如果小于，那么直接更新最小值，如果大于如果大于最小值，就添加到一个packet队列中 queuePacket(propose.packet)，这里就是添加，并没有发送，记住这个队列，然后又构造一个qcommit实例，这个commit请求就是我们说集群的时候，leader会有一个提议给follower，然后follower会返回leader一个ack然后，leader再回给follower一个commit，这个commit就是在这里发送给follower的，follower收到以后就会直接提交数据，不会再发ack之类的内容了，往后看queuePacket(qcommit);同样也是先添加到queuedPackets，并不会在这里发送。后面如果说peerLastZxid > maxCommittedLog，说明follower给leader的id比leader当前的还要大，那就得回滚掉，回滚的操作叫做TRUNC(packetToSend = Leader.TRUNC)。但是还有一种情况，那就是如果follower传递来的信息小于leader的zxid最小值怎么办，比如follower传来100，但是leader的min是200，那么就会去取默认值SNAP(int packetToSend = Leader.SNAP)快照，就是让follower直接从leader的快照中更新就行了，后面有SNAP的逻辑if (packetToSend == Leader.SNAP)。那么什么时候发呢，拉到最后有一个线程的run()里面的sendPackets()才会发送，进入后发现进入后会发现这里是一个while(true)的发送，只要此线程不停，凡是被添加到queuedPackets里的事务都会被发送,然后有p = queuedPackets.poll();取出队列，然后oa.writeRecord(p, "packet")发送出去。这里要区分一下，SNAP，TRUNC和DIFF是直接通过Socket发送出去的，而COMMIT是加到队列里，统一发出去的。

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.23--------------------------------------------------------------------------------------------------------

我们去看下follower是怎么处理SNAP，DIFF，TRUNC的，还是找到Follower. followLeader()找到syncWithLeader(newEpochZxid)，看名字也能判断出来这里就会接收到Leader告诉我的同步方式和需要同步的数据，进入后看到qp.getType() == Leader.DIFF) Diff好像什么都没有做就是赋了一个false值，qp.getType() == Leader.SNAP) 首先把当前的数据库清空zk.getZKDatabase().clear();然后反序列化leader发回来的数据leaderIs，zk.getZKDatabase().deserializeSnapshot(leaderIs) leaderIs这个是在connectToLeader()这个方法里拿到的，其实这两步就是先清空再赋值。qp.getType() == Leader.TRUNC，zk.getZKDatabase().truncateLog(qp.getZxid())这里的truncateLog()就是删数据了，传入的就是超出的id。那么我们接着看，下面又是一个for循环，如果还在运行的话self.isRunning()，那么我们就读取qp readPacket(qp);如果我们的qp是DIFF发送的，那么就一定又commit，所以这个循环里的switch就会走到case Leader.COMMIT这里，我们可以看到这里处理完了以后，就会加到一个队列里，然后break继续循环，因为我们的self.isRunning()不调用shutdown或者不挂掉的话永远都是true，那么什么时候跳出来呢，我们可以看到在这个Switch里有个Leader.UPTODATE:被执行到的时候有一个break outerLoop，就直接跳到了循环外面，那么这个update是什么时候传递进去的呢？我们回到LearnerHandler继续看rl.lock();这个锁解开了以后做了什么，发现这里又给Learner发了一个NEWLEADER的标记，明明leader已经选举出来了，为什么要还发送这个NewLeader呢，Learner这个类里面已经告诉你了， NEWLEADER这里就是要告诉新加的follower，不用选举了，我就是新的leader，而且里面做的什么也都说明了，就是在更新Epoch，然后用QuorumPeer.loadDataBase()这个方法去加载那些没有被打上快照之前就挂掉的事务。那么我们接着往下，走过packetToSend == Leader.SNAP，走过sendPackets();终于我们发现queuedPackets.add(new QuorumPacket(Leader.UPTODATE, -1, null, null));这里把update加到了queuedPackets这个队列里，其实到了这儿也就是说leader已经把所有的数据同步都告诉了follower，剩下的就是follower要同步数据了，而后Learner端碰到Update退出循环，这些同步的数据在之前都存到了packetsCommitted这里，然后执行到后面的fzk.commit(zxid);提交事务完成数据同步。到这里同步数据完成。

总结一下

Leader：

1. 找到要同步的数据，存到queue里
2. 发送出去
3. 发送一个update的执行，告诉follower数据可以使用了

Follower：

收到数据

while(true){

不停的加入指令到packetsCommitted队列里

直到收到update，退出while

}

执行指令

Update再往后面，每个LearnerHandler线程都会开启一个while(True)，这里是做什么的呢？这里就是在监听Learner会不会给我发送一些指令，比如ACK，Ping这种，但是其中有一个Request比较特殊。我们知道如果一个客户端连接了follower，并且给这个follower发送了一个写请求，那么这个写请求会被转发到Leader上，这个情况就是用的Leader.REQUEST这个分支，这个请求就是在这里发的。这里先不看。我们在转去Follower.followLeader()同步以后syncWithLeader(newEpochZxid);以后又一个while循环，这里和LearnerHandler的while(ture)是一样的，readPacket(qp);监听leader给我的信息，processPacket(qp);处理这个信息

初始化

到这里数据同步已经结束了，我们马上要初始化了，回到Leader.lead()方法，往下就看到了startZkServer();，同时follower是在syncWithLeader(newEpochZxid);中跳出循环以后开始初始化的。outLoop以后往下就有了zk.startup();开始初始化，其实startZkServer();这里调用的也是zk.startup()，这里和单机模式也是一样的，服务器接收客户端的请求最核心的就是RequestProcess那些处理器链，但是又和单机模式有些区别。

我们去看这些区别，首先先进入startZkServer()方法，在进入zk.startup();发现这个和单机模式没有任何区别嘛，都是调用的setupRequestProcessors()。但是其实，这里是不一样的，如果是从Leader这个类进入的，那么这个zk就是LeaderZooKeeperServer的实例对象，而不是ZooKeeperServer的实例对象。同理，如果是从Follower这个类进入的那么zk就是LearnerZooKeeperServer这个类的实例对象。所以直接ctrl进去是看不到的，所以我们要找的应该是LeaderZooKeeperServer. setupRequestProcessors()这样你才能看到集群和单机模式的区别，当然Follower和Observer也有相应的实现方法。

那么看下代码发现firstProcessor就是PrepRequestProcessor 这个和单机是一样的，它的next是ProposalRequestProcessor再next是CommitProcessor再next是ToBeAppliedRequestProcessor最后才是FinalRequestProcessor

Lead就是这样的一个链：

firstProcessor=PrepRequestProcessor.next- >ProposalRequestProcessor.next- >CommitProcessor .next->ToBeAppliedRequestProcessor.next->FinalRequestProcessor

要注意到有一句proposalProcessor.initialize();，进入后发现syncProcessor.start();所以其实这里SyncRequestProcessor其实还是在用的，而且还单独开启了一个线程SyncRequestProcessor.next->AckRequestProcessor。

我们再去看下follower是怎么样的。

查看代码，拿到了另一个链firstProcessor=FollowerRequestProcessor.next-> CommitProcessor.next-> FinalRequestProcessor

然后也单独开了一个Sync链SyncRequestProcessor.next->SendAckRequestProcessor

最后再去看Observer的链

firstProcessor=ObserverRequestProcessor.next->CommitProcessor.next->FinalRequestProcessor

这里 区别就在于Observer控制了SyncProcess是否要进行持久化和打快照if (syncRequestProcessorEnabled)这个if判断决定了Observer是否启动SyncProcess

PrepRequestProcessor：checkACL,构造txn

SyncRequestProcessor：持久化txn，快照

FinalRequestProcessor：更新内存，返回response给客户端

ProposalRequestProcessor：投票

CommitProcessor：提交

我们先看lead的

PrepRequestProcessor启动以后，老规矩 进入run()拉到最后会触发执行下一个process所以我们找到nextProcessor.processRequest(request)，那么这nextProcessor是谁呢？我们就发现了nextP其实再PrepRequestProcessor()这个构造方法里被赋值了，而这个构造方法是在哪里被使用的呢，就是firstProcessor = new PrepRequestProcessor(this, proposalProcessor);传入的就是ProposalRequestProcessor，所以这里我们再使用的processRequest()就是ProposalRequestProcessor这个类实现的了，所以我们找到ProposalRequestProcessor.processRequest()。

结果进入后发现如果不是Learner的同步请求就直接走到了nextProcessor.processRequest(request);，这就不太对了，因为我们知道如果一个create之类的请求进来，肯定是要先发起来提议的，经过ACK确认以后才会提交，怎么就会直接到了CommitProcessor里面去提交这个请求了呢？那么我们打开CommitProcessor.processRequest()一起对比看下。进入后，首先会加到queuedRequests队列里面来，因为在LeaderZooKeeperServer. setupRequestProcessors()已经开始commitProcessor.start();了，所以一旦加进来就会立刻到run()方法里，通过代码里分析，这里会被阻塞掉wait();，并不会立刻执行。那么既然阻塞了，就必然会被唤醒，那就得清楚我们等的是什么，之前分析了，事务要经过投票，投票过半响应才能执行，那么这里一定等待的就是投票的结果。于是我们在nextProcessor.processRequest(request);这里被阻塞以后接着往下走，进入if (request.hdr != null)这个if语句里，我们就发现了投票方法zks.getLeader().propose(request);进入看到pp = new QuorumPacket(Leader.PROPOSAL, request.zxid, data, null);请求被封装成一个packet，注意这里的type就是PROPOSAL提议了，data也被封装了进去。按道理这里发出去了，我们的follower就应该接收到这个proposal，下面把proposal放到队列里去outstandingProposals.put(lastProposed, p)，这个队列我们后面还会在Leader里用到，然后发送出去sendPacket(pp);这里没有任何的阻塞，我们就直接返回出去到了syncProcessor.processRequest(request);又开始调用持久化了，这是为什么呢？还记得上面leader走两条链吗？那么我们就去syncProcessor的下一步AckRequestProcessor里面看看，sync持久化后，又做了什么。根据处理器链的逻辑sync最后一定是AckRequestProcessor.processRequest()那么这里就是说，持久化结束了，才会做这个ack，

这个时候我们就可以去看下follower那边是怎么处理的。

我们现在去找到Follower.followLeader()里面关于接收Leader消息的那段代码。找到while()里面的processPacket(qp)进入查看，发现了Leader.PROPOSAL接收这个提议，然后取出数据，到fzk.logRequest(hdr, txn);这个方法里看，发现也是调用了syncProcessor.processRequest(request);持久化，那我们就得看follower这边调用的持久化后面又是谁，我们进入看发现Follower这边的sync后面是SendAckRequestProcessor发送ACK，也就是说follower这边也是持久化以后发还ack给leader。那么我们就明白了提议的逻辑原来是这样的：首先leader接收到一个事务，然后leader进行持久化，leader持久化以后发送一个提议给follower，follower怎么判断这个提议能不能成功呢？也是持久化，follower接到这个提议以后，直接就去持久化，如果follower的持久化成功了，就会调用发送ack的process，进入SendAckRequestProcessor.processRequest(request);也能看到这个process很简单，就是发送一个ack: learner.writePacket(qp, false)，也就是会告诉leader，我这个follower持久化没问题可以做这个事务，同意这个提议。

到这里我们就又得返回LearnerHandle.run()方法里找到while(true)里的ACK分支，因为要记住我们这个线程一直再跑，而while(true)是一直监听的消息的。那么我们找到Leader.ACK，拿到ack以后就会处理leader.processAck()进入看,看到一堆提取数据的代码，略过，找到p.ackSet.add(sid) 把返回ack的follower的server id加入packet里面，把收到并返回ack的follower都加入进来，然后又发现了过半验证的判断if (self.getQuorumVerifier().containsQuorum(p.ackSet))，然后验证是不是过半了，一旦验证过半了就进入if，然后提交commit(zxid)这个事务。进入，里面就只有创建打包为一个packet，然后sendPacket(qp)给所有的follower，跳出还有一个inform(p);这里就是发送一个INFORM的type给我们的observer。发送以后，我们的follower自然就可以在Follower.followerLeader().while(isrunning).processPacket().Leader.COMMIT分支里处理这个事务fzk.commit(qp.getZxid())，进入后我们发现这个提交用的就是commitProcessor.commit(request)那么我们进入这个commit，在这里面，我们就发现了两个notifyAll()唤醒和committedRequests.add(request)。

这时候在CommitProcessor里面就能够直接走到finalProcessor里面了，并且刚才阻塞的事务可以被提交了。所以CommitProcessor做了这样的逻辑，首先，不能提交的我就先阻塞住等待ack，可以提交的我就直接调用下一个process去提交更新内存。

总结一下：

Client发了一个写请求给leader，leader首先会发送一个提议给followers，followers会返回ack给leader，然后leader会进行一个过半验证，验证过了会给followers发送提交认证，followers接收到以后就会直接调用finalprocessor更新内存，因为持久化已经在发送ack之前就做完了。Leader持久化做好了，提议也通过了，接下来leader就也走到了finalprocessor更新内存。

那么我们看下如果client发给follower一个请求会有什么不同：

如果一个客户端个follower发送了一个写请求，那么根据我们上面的分析，第一个接到写请求的应该是FollowerRequestProcessor.run(),因为这里也是放到一个队列里面，然后有一个线程去取出来，到这里面以后看到这里也是直接执行了nextProcessor.processRequest(request)，那么这里就到了CommitProcessor里面了，还记得这里面的不合标准的请求会直接阻塞，然后我们接着走到下面的switch里去，我们看到只要不是同步，它都会调用zks.getFollower().request(request)把这个请求发送出去，request(request)这个方法就是初始化了一些参数，然后直接包装成一个packet写到socket里writePacket(qp, true)，这里就是follower往leader转发的过程，我们看这里封装的就是REQUEST。这个就是之前我们提到的在LearnerHandler里面的Leader.REQUEST，这里就发现了请求提交的方法leader.zk.submitRequest(si)进去，我们要注意这里的zk是LeaderZookeeperServer的实例，所以这个方法里的firstProcessor就是我们leader处理链中的PrepRequestProcessor，这样follower接收到的事物，就转发到了Leader上面，Observer也是同理，这样zk的一致性保证就结束了。

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.28--------------------------------------------------------------------------------------------------------

领导者选举算法

步骤：

1. 确定节点的角色
2. 同步数据
3. 接收初始化RequestProcess
4. 接受请求

选举领导的标准：Zk定义Leader一定是数据最新的那一个

数据新旧标准：

1. zxid大的就是新的
2. 如果zxid是一样的，那么myid最大的是就是最可能被选中的

集群启动的时候

1. 每个机器先投自己一票，这时这个票包含的是<sid,zxid>哪台服务器，zxid是多大
2. 把票发给对方，各个机器接到不同的机器的票数以后，存到投票箱里，在投票箱里对比数据，数据优势的一方会更改掉数据弱势的一方的票，然后再重新发出去
3. 优势的机器收到重发发出的票以后，会统计自己收到的票，超过半数成为leader

集群挂掉：

1. 如果挂掉的是Leader，剩下机器会重新选举一个新的leader
2. 如果挂掉follower，剩下的机器未必会重新选举

源码：

首先我们还是先启动，进入QuorumPeerMain.main()，进入加载类main.initializeAndRun(args)，进入集群模式runFromConfig(config)，略过初始化参数，找到并进入quorumPeer.start()，然后就看到了我们今天要看的领导者选举方法startLeaderElection()，首先我们就看到了一句话currentVote = new Vote(myid, getLastLoggedZxid(), getCurrentEpoch())先给自己投票，经过上面的原理分析，我们知道每台服务器都有一个自己的投票箱，因为不可能再创建一个投票服务器，且投票箱里存放的应该是当前自己的投票和别人的选票，那么我们总结三个重要的概念，投票箱、投票、选票。回来这句话这个myid是自己的myid，我们进入Vote看下。我们解释下几个选举用到的参数this.id = id：myid，this.zxid = zxid：事务id，this.electionEpoch = -1：这个是记录领导者选举的轮数的，服务器内自己记录的，这个数值越大，投票的作用越大，对传入的低数值epoch有否决权，this.peerEpoch = peerEpoch;当前选举周期，届号，传递进来的参数。

往下找到选举算法的方法createElectionAlgorithm(electionType)进入，发现zk其实之前有好多中选举的机制，但是这些在当前版本已经是过期的了，那么现在所留下来的也就只有唯一的一种FastLeaderElection(this, qcm)，我们直接走到le = new FastLeaderElection(this, qcm)里面发现，这里就是new了一个类，并不是真正做算法的地方，那么QuorumPeer.start().startLeaderElection()看起来像是选举方法，但是其实这个方法只是进行领导者选举，确定服务器角色，在针对不同的服务器角色初始化，并不是领导者选举逻辑所在的地方。

我们先看下electionType=3到底要做什么，首先我们先说明下，选举也是通过socket链接进行的，那我们看case 3: qcm = createCnxnManager();进入，new QuorumCnxManager()进入，再进入this()，到QuorumCnxManager类里面去，这个类就是负责底层传输的，所以我们解析一下这个类里面的重要属性

1. ConcurrentHashMap<Long, SendWorker> senderWorkerMap

格式：serverId(myid):SendWorker

Long：存的其他服务器的编号serverId，也就是myid

SendWorker：负责发送数据的类

也就是说这个Map其实就是保存了当前服务器去向其他服务器发送的数据的SendWorker

1. ConcurrentHashMap<Long,ArrayBlockingQueue<ByteBuffer>>queueSendMap

格式：myid:Queue

Long：存的其他服务器的编号serverId，也就是myid

ArrayBlockingQueue：是一个数据队列， 是要向其他服务器发送的数据的队列

所以这个map保存的就是，当前的服务器要向其他服务器发送的数据的队列

SendWorker是一个线程，这个线程就负责去ArrayBlockingQueue这个队列里取出数据发送出去，这里是不是和之前同步数据的地方很像，只不过这里就是发送的选票。

1. ArrayBlockingQueue<Message> recvQueue: 保存本台服务器接收到的消息
2. ConcurrentHashMap<Long, ByteBuffer> lastMessageSent;

格式：myid: ByteBuffer

Long：存的其他服务器的编号serverId，也就是myid

ByteBuffer：存的就是最新发出去的消息

所以这个map保存的就是，本服务器发送给每台服务器最新的消息，这里也是用了一个map去存的，这里存的只有最新发送出去的消息。

请大家牢牢记住这几个属性，下面的解析离不开他们

再往下又碰见一个很关键的属性listener = new Listener();，我们点进去发现这个Listener是一个QuorumCnxManager的内部类，而且还是一个线程类，这里只是new了一个实例，并没有启动，那么我们接着case 3往下走，QuorumCnxManager.Listener listener = qcm.listener;把listener取出来，然后start()，那么这里就开始运行Listener中的run()方法了，我们去去看下这个run方法中做了什么，进入后发现了while循环里面有socket，ss = new ServerSocket();但是注意这里使用的是java中的Socket是TCP而不是NIO，经过一些过滤的逻辑走到了ss.bind(addr);绑定地址，这个地址electionAddr的端口就是3887，下面又一个while循环，这里的Socket client = ss.accept();会被阻塞，等待其他的服务器来连接，连接通过以后就会走到下面receiveConnection(client)，我们注意到了这里面的while几乎也是一个while(true)，所以其实每一个socket都会调用一次这个方法，那么我们进去receiveConnection()看下里面，

进入后看到取出数据din = new DataInputStream()，然后调用handleConnection(sock, din)进行连接，进入，

里面有一个很长的判断，首先if (sid < 0)sid小于0肯定不合法，然后如果是观察者if (sid == QuorumPeer.OBSERVER\_ID)，也先略过，重点看if (sid < this.mySid)，这个sid是对方连接过来的服务器传递进来的myid，在之前sid = din.readLong();拿到的，this.mySid这个很明显就是自己的sid，那么为什么这里要进行一个sid的大小比较呢？比如1号服务器要连接2号服务器，那么返回来2号也要链接1号服务器，如果不加以限制，那么这两条服务器就会启动两条socket相互连接，这样明显就是多余了，所以按照myid的大小进行了一个连接的规则，也就是说领导者选举里面小的sid连接大的sid是不允许的。

那么我们进入这个if以后就发现，这里会先closeSocket(sock);关掉外部传来的socket，然后主动去connectOne(sid);去连接对方。我们先看如果正好符合这个规则，那么进入else，先new一个SendWorker sw，然后再new一个RecvWorker rw，再后面我们就看到了一个map，SendWorker vsw = senderWorkerMap.get(sid)，if(vsw != null)这里是一个验证，如果这里不是null，说明这里已经发送过了，就不用再发送了，验证过了以后就会走到下面两个map的put方法中，一个存入SendWorker，一个存入发送数据的queue中，然后启动这两个线程start()。

那么就先去SendWorker的run()中看看里面写了什么，首先看到bq = queueSendMap.get(sid)数据被拿出来了，然后经过一个if，如果里面没有数据(null)那么我们把数据从lastMessageSent.get(sid)拿出来,再转化为ByteBuffer b，然后发送出去send(b)。如果里面有数据，那么就在这个while循环里不断的拿出队列bq = queueSendMap.get(sid);然后从队列里拿出数据b = pollSendQueue()，最后send(b)发送。

SendWorker.start()运行完以后，就该RecvWorker.start()了，所以我们接着看RecvWorker.run()，

从名字上看这个应该是接受数据的，果然这个线程首先从socket里面读出数据din.readFully(msgArray, 0, length);，然后包装成ByteBuffer实例ByteBuffer message = ByteBuffer.wrap(msgArray)，再调用addToRecvQueue(new Message(message.duplicate(), sid))包装字节数据成Message实例放到recvQueue里面，这个方法里面并没有什么逻辑只是一个判空然后add一个message到队列里。

所以总结一下：我们到目前为止，应该有这样一个逻辑

queueSendMap(存发送命令)--取走-->SendWorker线程(这是Map) --Socket发送到-->其他

recvQueue(存收到命令)<--存入--RecvWorker线程(这也是个Map)<--Socket发送到--其他

所以这部分的重点就在于SendWorker和RecvWorker这两个线程的逻辑，这两个类都是属于QuorumCnxManager这个传输层的类。  
那么我们现在有两个问题，1. queueSendMap这里的数据从何而来，2. recvQueue收到的数据是谁来使用的？

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.29--------------------------------------------------------------------------------------------------------

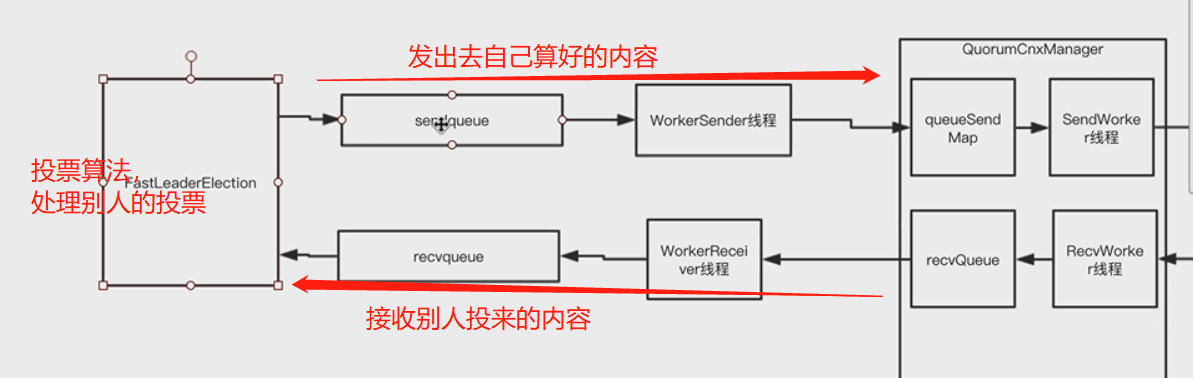
这两个线程走完以后只有一个return;，那么我们还得一步步的返回去，说到这里我们只是看完了listener.start()这个线程的run方法，所以我们回到QuorumPeer. createElectionAlgorithm()方法中，接着listener.start()继续看，开始正式解析FastLeaderElection做了什么事情，

进入构造方法，就有看到两个queue：sendqueue和revcqueue和一个message。点进Message，发现里面又启动了两个线程WorkerSender和WorkerReceiver。我们一个一个来看，先点进去WorkerSender，看到了run方法，可以看到就是在这里取出了sendqueue里面的内容ToSend m = sendqueue.poll()，然后调用了process(m)方法，继续进去process()方法，里面只有一行manager.toSend(m.sid, requestBuffer)，这个manager又是什么呢，他其实就是QuorumCnxManager这个类的对象，那我们进入toSend()方法。这里有一个逻辑判断if (this.mySid == sid)这里是说如果是发给自己的，则不需要经过网络，自己转成Message，加入到recvQueue即可，else如果不是发给自己的，发送给其他服务器的，就应该加入到服务器所对应的发送队列中，这里就发现了queueSendMap，这里逻辑很简单其实就是把要发的数据放入queueSendMap中，那么到这里我们的第一个问题就回答出来了：queueSendMap中的数据就是来自sendqueue，被WorkerSender线程从sendqueue读出来放进去的。

我们回头看WorkerReceiver，进入run()，看

到首先又是调用的manager.pollRecvQueue()，接着点进去，就发现了我们的recvQueue.poll(timeout, unit)，很明显WorkerReceiver这个线程就是取数据的线程，接着走，如果为null继续，如果不为null又有一大堆的逻辑，到这里就很明显就是处理消息的逻辑了，这里的代码具体什么用的先不解释，但是到这里我们的第二个问题也有答案了WorkerReceiver就是消费recvQueue内容的线程，然后再把消息根据逻辑的不同一部分放到sendqueue一部分放到revcqueue里。那么这两个队列的后面，就是领导者选举的算法的核心部分，真正的算法部分。

如图



算法只要把要发送的投票发送到sendqueue这个队列里可以了，而取投票只要去revcqueue里面取就可以了，就代表了本机从别人那里去到了投票。

到这里我们就可以往上返回了，返回到最初我们进入的方法startLeaderElection()，下一步就是super.start();，这里就要启动本类的run()方法了，那么我们现在去QuorumPeer本类的run()看下，上面我们说了LOOKING的状态，代表还没有确定好角色，就是正在进行领导者选举，所以领导者选举的方法应该就是在这里，由于getPeerState()中的status默认的就是LOOKING状态，所以一台机器启动的时候就会进到这个case里去，跳过只读模式部分的内容，真正关心的方法就是lookForLeader()，这是个接口，找到FastLeaderElection类对这个方法的实现类，跳过一些初始化，找到recvset = new HashMap<Long, Vote>()，这个map就是我们提到的投票箱的那个箱子，Key(Long)代表其他服务器，Value(Vote)代表该服务器投的票，如果有更新，按照key来更新就好了。往下我们看到一个synchronized包围的代码，这里其实就是先投票给自己的逻辑，logicalclock.incrementAndGet()记录自己的投票数然后加1(首次启动，初始化为1)，然后updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch())更新提议包含(myid,lastZxid,epoch)，更新为自己，这里只是设置了内容，后面发送自己的投票sendNotifications()会用到，

进入sendNotifications()方法，立即就看到了一个for循环，而且是循环QuorumServer的，很明显这个循环就是发现那些能够投票的机器的，点进去self.getVotingView().values()就可以很明显看出来，这个循环里面取了刚才update设置的内容，然后new了一个ToSend再放入发送队列sendqueue.offer(notmsg)，那么这个循环就会不断地生成投给自己的投票，这个数目就是配置的机器数目,当然也包括自己，第一个步骤投给自己的逻辑就走完了。

跳出sendNotifications()走到下面的while ((self.getPeerState() == ServerState.LOOKING) && (!stop))里面，

如果服务器还在运行，且状态是Looking就会再往下走，就看到了一个很熟悉的队列Notification n = recvqueue.poll((notTimeout, TimeUnit.MILLISECONDS);在这里获取其他服务器的投票，首次启动的时候这里并不会接到其他的服务器发来的数据，但是其实也还会有数据的，这个数据就是自己投票给自己的数据，不需要经过网络，WorkSender这个线程就直接调用方法把自己的票放到了recvQueue中，然后(上图)就又到了recvqueue里，这里还要注意WorkSender也会把要发往别的服务器的投票从sendqueue取出放到queueSendMap中，这里放的当然就是配置中的需要经过网络的要到其他服务器的投票。当然这里的n就是取到的结果也就是我们说的选票。紧接着是一个判断，if(n == null)获取的选票是空的，说明没有收到别的服务器发来的数据，如果没有收到别的服务器发来的票，那么就要判断一下是不是我的票没有发出去，或者发出去了但是没有收到回复，manager.haveDelivered()就是判断这一点的，进入以后会循环queueSendMap，如果发现里面有数据，说明投票没有发出去，那么就返回false，如果发现里面有queue没有数据，queue.size()==0说明，我的这个票发成功了，但是只要进入这个if说明，在外面我拿到的选票n一定是null，意味着我肯定没有得到回复，返回true，再次发送这个数据。所以当这个if是true的时候就sendNotifications()既没有别人来的选票，我发的选票也没有得到其他的回复，那么我就把我配置的选票再发送一次，再次等待目标机的回复。else：queueSendMap这个map里面有数据，我的票没有发出去，那么可能的原因就是还没有建立连接，调用manager.connectAll()连接所有配置的服务器，

进入这里可以看到服务器的id是从queueSendMap取出来的，queueSendMap.keys()，哪台的没有发出去，就去连接哪台connectOne(sid)。进入以后就看见一个判断if (!connectedToPeer(sid)) 如果真的没有连接，才回去重新连接，然后获取地址建立socket，初始化initiateConnection(sock, sid)进入，接着进入startConnection(sock, sid)，这里面就是一样的逻辑了if (sid > this.mySid)连接的控制，新建SendWorker和RecvWorker然后启动，这样就完成了connectAll()，所以这个n==null这一块的逻辑其实就是一个保险机制，用来检验投票的发送和连接成功与否用的。那么验证过了n!=null就是选举的核心逻辑了，到这里说明收到了外来的选票而且合法，

----------------------------------------------------------------------------------------------------------以上已经写完，时间2020.7.29--------------------------------------------------------------------------------------------------------

那么只要是正在选举的服务器应该都是Looking状态，我们先去case LOOKING看，首先就是判断谁的时钟周期大if (n.electionEpoch > logicalclock.get()如果接收到的投票周期比自己的周期高，更新自己的时钟为外来的周期logicalclock.set(n.electionEpoch)，然后recvset.clear();清空自己的选票箱子，这里为什么要清空呢，如果时钟周期不一样，那就没有意义了，这一步其实就是在重置所有服务器的时钟周期。重置以后，比较选票对应的服务器和本机，如果选票对应的服务器就更新为选票对应的服务器，否则投给自己，然后发送出去。

如果if (n.electionEpoch < logicalclock.get()别人投票的周期，比我的小，就 break;直接放弃了跳出switch进行下一个while循环，如果时钟周期相等，那就比较sid，Zxid，Epoch谁的服务器内容是更高等级的if (totalOrderPredicate(n.leader, n.zxid, n.peerEpoch, proposedLeader, proposedZxid, proposedEpoch))，n.leader, n.zxid, n.peerEpoch这些很明显就是外面传递进来的，proposedLeader, proposedZxid, proposedEpoch这三个是自己的投票，这三个的值有可能是自己，也有可能会随着投票的进行更新出来的候选者的数据，走到这里就说名传递进来的比较强大，所以更新选票为外来的updateProposal(n.leader, n.zxid, n.peerEpoch);然后发送投票sendNotifications();。接着保存自己接收的选票recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));，也就是把合法的投票放到投票箱，以上逻辑都是在更新proposed的选票，也就是在选准leader是谁，并不会清空外来的选票，除非突然来一个时钟周期更高的，此时自己持有的选票全部都无效了，所以会调用clearAll清空选票箱，因为要根据接收到的票决定自己是不是leader，所以有必要保留所有的票。所以这里没接收一个投票统计一个投票。

然后就开始过滤过半机制，紧接着的if (termPredicate(recvset, new Vote(proposedLeader, proposedZxid, logicalclock.get(), proposedEpoch)))就是过半的机制判断，我们进入看下，首先说明下参数，votes是投票箱，vote是自己的认可的选票(proposed选票)，里面new了一个set，这里和后面的for循环一起就可以看出来，这个一块的逻辑主要做的就是把和我投一样票的机器找出来并且加入到一个set中，然后通过QuorumVerifier判断是不是set.size() > half。

跳出到了这个if内说明过了过半机制，本台机器就认为是选出了leader，这里先叫做准leader，紧接着一个while是一个验证，如果我们已经找到一个准leader以后，又来了选票怎么办呢，我就得再判断这个选票和我的准leader哪个厉害，如果说新来的选票厉害，那么就把新来的选票放入recvqueue中，退出这个循环，进入上层循环，在进行一轮准leader的投票，如果来的选票竞争失败了，那么就一直循环，直到无法获取新的选票了。

那么我们跳出while以后，看到了if (n == null)这就说明确实没有票了，leader就选出来了，那么就要开始设置状态了self.setPeerState((proposedLeader == self.getId()) ? ServerState.LEADING: learningState());对比准leader的sid是不是和自己的一样，如果一样就更新自己服务器状态到Leading，如果不是就根据配置文件设置为相应的状态FOLLOWING，或者OBSERVING。最后生成一个新的Vote设置成选出的leader的内容，返回出去return endVote。那么做完这些，谁是leader，谁是follower基本上就确定了，然后就会跳出，一直往上直到QuorumPeer,run()里面，接着执行while(running)，那么改是Leading的，就走到case LEADING:，改是following的case FOLLOWING:进行同步数据等等，这个之前同步数据那里已经说过了，到这里服务器启动时候的选举逻辑就已经结束了。

我们还有一个情况，启动起来以后，由服务器挂了，这种时候是怎么选举的呢？其实还是在这里这个while(running)，因为同步数据就是在这里进行的。

1. Leader挂了：其他服务器会在case FOLLOWING:这里follower.followLeader();这里报错，因为这里是同步数据的，Leader挂了会导致follower无法通过这个方法同步数据，然后经过捕获错误，到finally后重新设定状态为Looking，然后所有机器一起选举setPeerState(ServerState.LOOKING);，Observer也是一样只不过报错的地方是observer.observeLeader();但是Observer不参与选举所以其实对于本节来说无所谓了。
2. Follower挂了：这里只有当follower挂的超过了一半了才会导致重新选举，如果超过一半了这里报错的地方则是case LEADING的leader.lead();，进入跳过数据同步的地方走到这个方法里的最后一段逻辑，while (true){}这里，首先我们看到Thread.sleep(self.tickTime / 2)，这个while其实很明显就是leader在呼叫别的follower是不是还活着的，往下看有一个set，首先leader先把自己放进去了syncedSet.add(self.getId());，紧跟着就是一个for循环，我们之前已经分析过，每一个learner都会有一个LearnerHandler，这里的for循环就是在找哪些follower是活的，看里面的if条件，如果拿出来的f是活的f.synced()，并且f的Type是Participant f.getLearnerType() == LearnerType.PARTICIPANT，那么就把这个机器加到syncedSet中，如果不是活的那么就不会加到这个set中，跳出这个for以后，再往下就又是一个过半验证if (!tickSkip && !self.getQuorumVerifier().containsQuorum(syncedSet))，如果syncedSet里面没有超过一半的机器，那么就是直接shutdown()，因为这里是一个while(true)正常情况下这个线程是不应该结束的，然是现在结束了于是就挑出了leader.lead();，继续走到finally里面，如果leader活着，先停了，再把自己的状态变成Looking重新进行选举。总结一下，其实就是leader在自己运行的时候不断地去ping自己的follower如果发现自己的follower小于等于配置的机器的一半，就自废武功重新举行一个选举，重新确定谁是新的leader。
3. 新加入一台服务器：当新加入一台服务器的时候，第一件事情是做什么呢？当然是发送投票，因为新加入的机器不知道谁是leader肯定会发送自己的投票给所有的机器，投票机制都一样主要就是其他机器接到投票后怎么处理的问题，所以我们就得重新找到WorkerReceiver这个线程类，进入run()方法，

看到接到数据以后就有一个if(!validVoter(response.sid)) 如果不是合法的的，比如集群的配置里就没有这个机器，就把自己的投票发给他sendqueue.offer(notmsg);，如果是合法的，那么就先解析对方服务器的状态，是LOOKING，FOLLOWING，LEADING还是OBSERVING，然后获取对方的其他值sid、zxid、electionEpoch等等，到下面的if(self.getPeerState() == QuorumPeer.ServerState.LOOKING)，如果自己的也是Looking，说明正在进行选举就先给自己发一份recvqueue.offer(n);，进入以后如果发现对方也是looking，发现对方也是looking，就给对方也发一份我自己的，后面就交给选举算法了。如果说我不是looking状态，那么我就是Leader或者Follower，我就把我认为的leader的票发给对方，这里的n.version是一个兼容性检测，其实看内容可以把这个忽略掉，无论是current还是bcVote最终都是current作为选票的内容，最后发送出去sendqueue.offer(notmsg)，正式结束。

Zookeeper的分布式锁和分布式配置中心

ThreadLocal 学习是干嘛的

CountDownLatch 学习

服务器启动的时候也有一个逻辑：从文件取出数据加载到内存DataBase()进入zkDb.loadDataBase()进入，看snapLog.restore()方法进入，看到snapLog.deserialize(dt, sessions) 把快照的日志反序列化出来数据存到datatree(参数dt)中，然后我们继续进入fastForwardFromEdits(dt, sessions, listener)方法中，这里在写第一个create的博客的时候要注意，写清楚fastForwardFromEdits这里的内容

一个命令的周期

ZookeeperMain🡪建立Socket连接初始化ClientCnxn，由ClientCnxnSocket接口和其实现类ClientCnxnSocketNIO实现，并最终实例化为一个cnxn对象，此时ClientCnxn实例化一个SendThread对象和一个EventThread对象，由cnxn.start()启动这两个对象。在SendThread.run()中，有重试机制的具体写法，以及客户端联系服务器不断使用的ping方法，最终交给clientCnxnSocket.doTransport()发送给服务器🡪Zookeeper.create()，首先进行验证，然后调用submitRequest方法来实现的，所谓提交请求，就是把请求包装成一个packet然后封装到一个队列中去🡪doTransport