# 6. Zookeeper源码分析

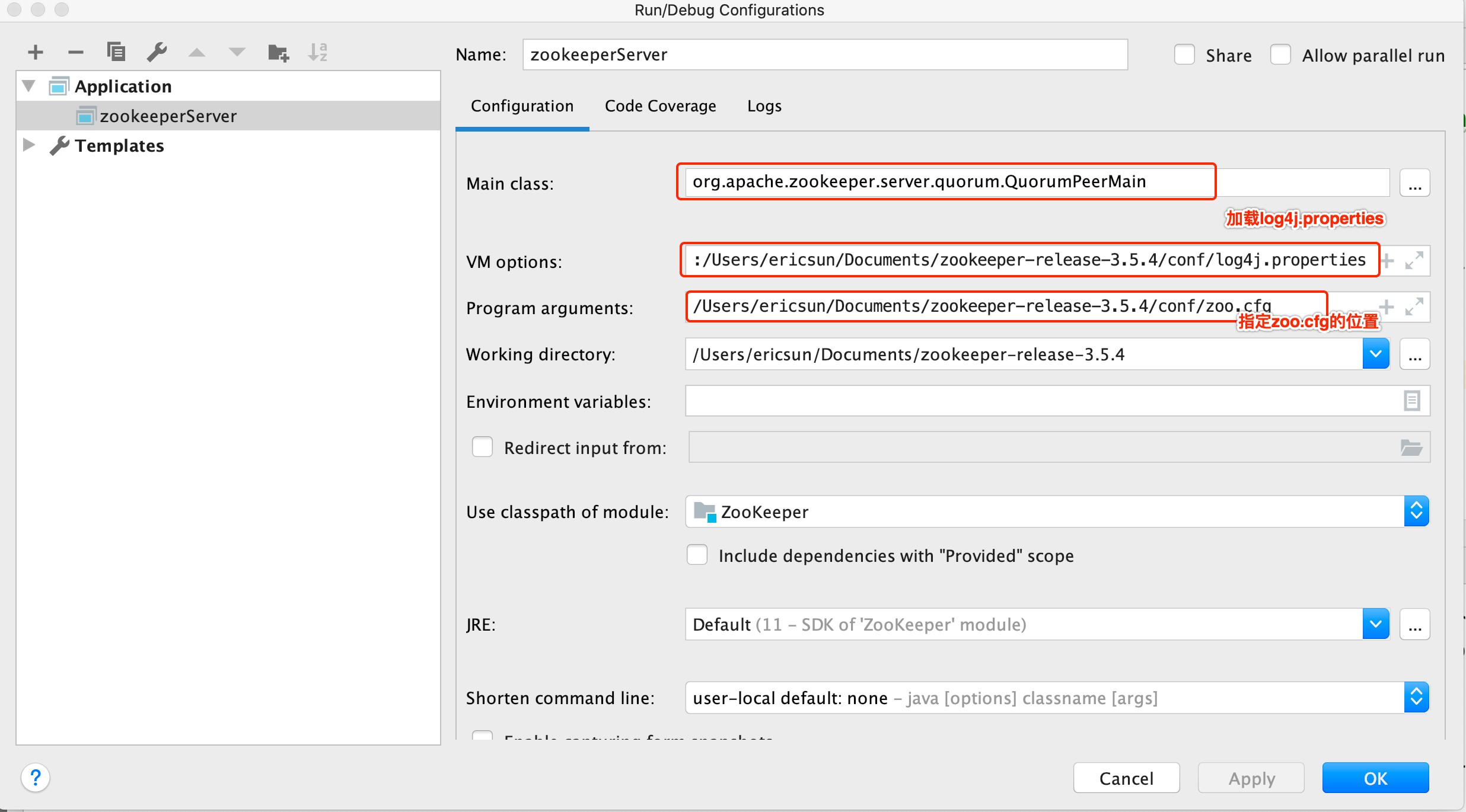
## 源码环境搭建

zk源码下载地址：https://github.com/apache/zookeeper/tree/release-3.5.4

注意：因为zk是由ant来构建的，所以需要使⽤ant命令来转换成⼯程，然后导⼊idea 将准备好的zookeeper-release-3.5.4导⼊idea中

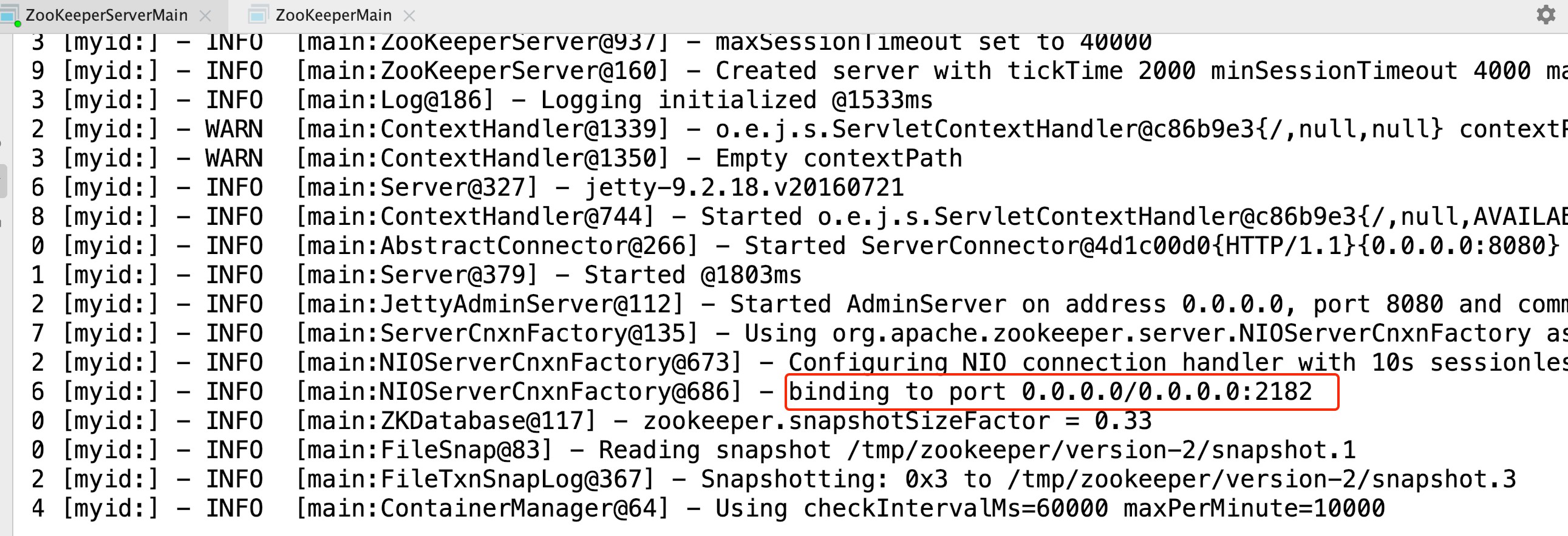
#### 启动服务端

运⾏主类 org.apache.zookeeper.server.QuorumPeerMain ，将zoo.cfg的完整路径配置在Program arguments。



在VM options配置，即指定到conf⽬录下的log4j.properties：

-Dlog4j.configuration=file:/Users/ericsun/Documents/zookeeper-release- 3.5.4/conf/log4j.properties

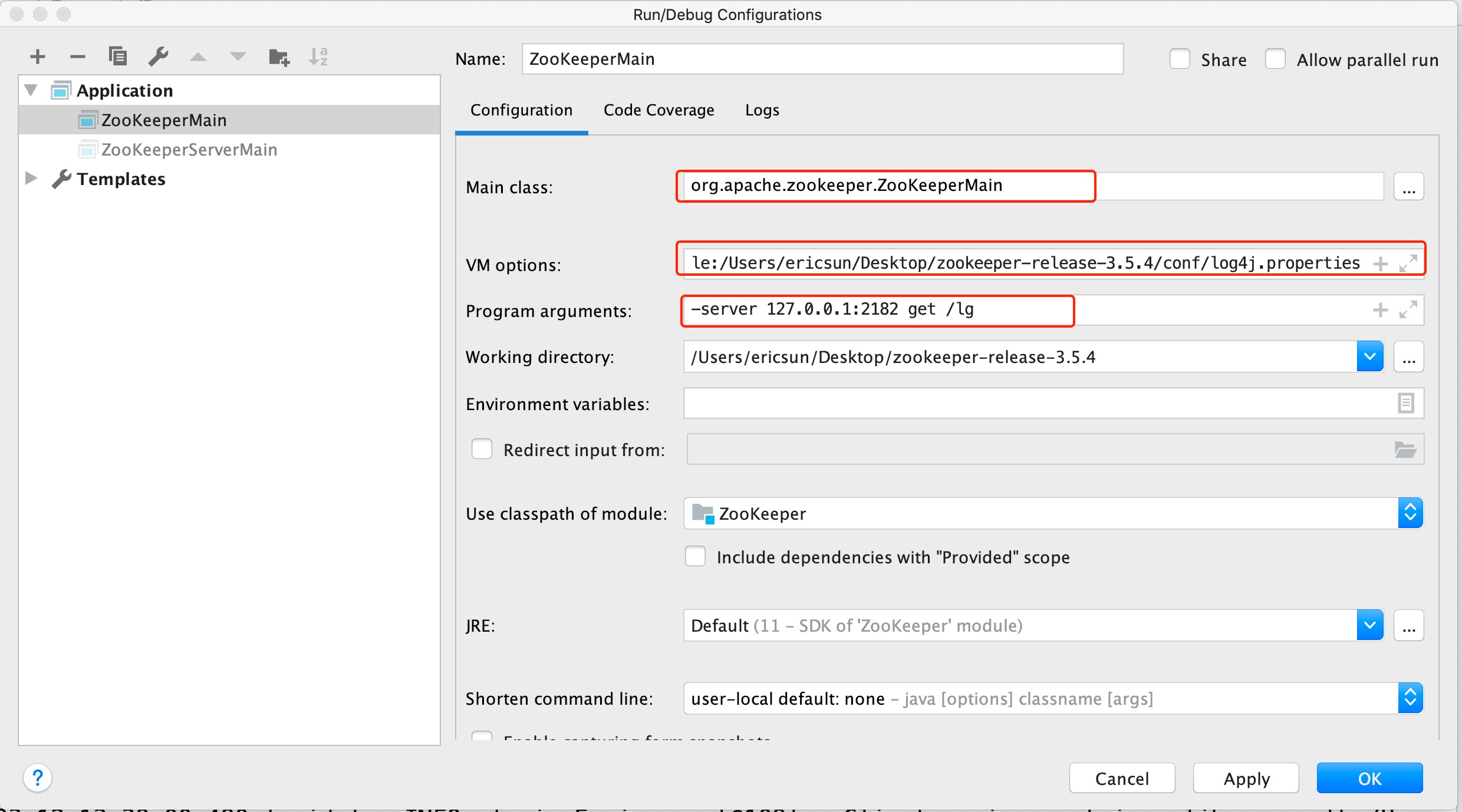
运⾏输出⽇志如下

可以得知单机版启动成功，单机版服务端地址为127.0.0.1:2182。

#### 运⾏客户端

通过运⾏ZooKeeperServerMain 得到的⽇志，可以得知ZooKeeper服务端已经启动，服务的地址为 127.0.0.1:2181 。启动客户端来进⾏连接测试。

客户端的启动类为org.apache.zookeeper.ZooKeeperMain ，进⾏如下配置：

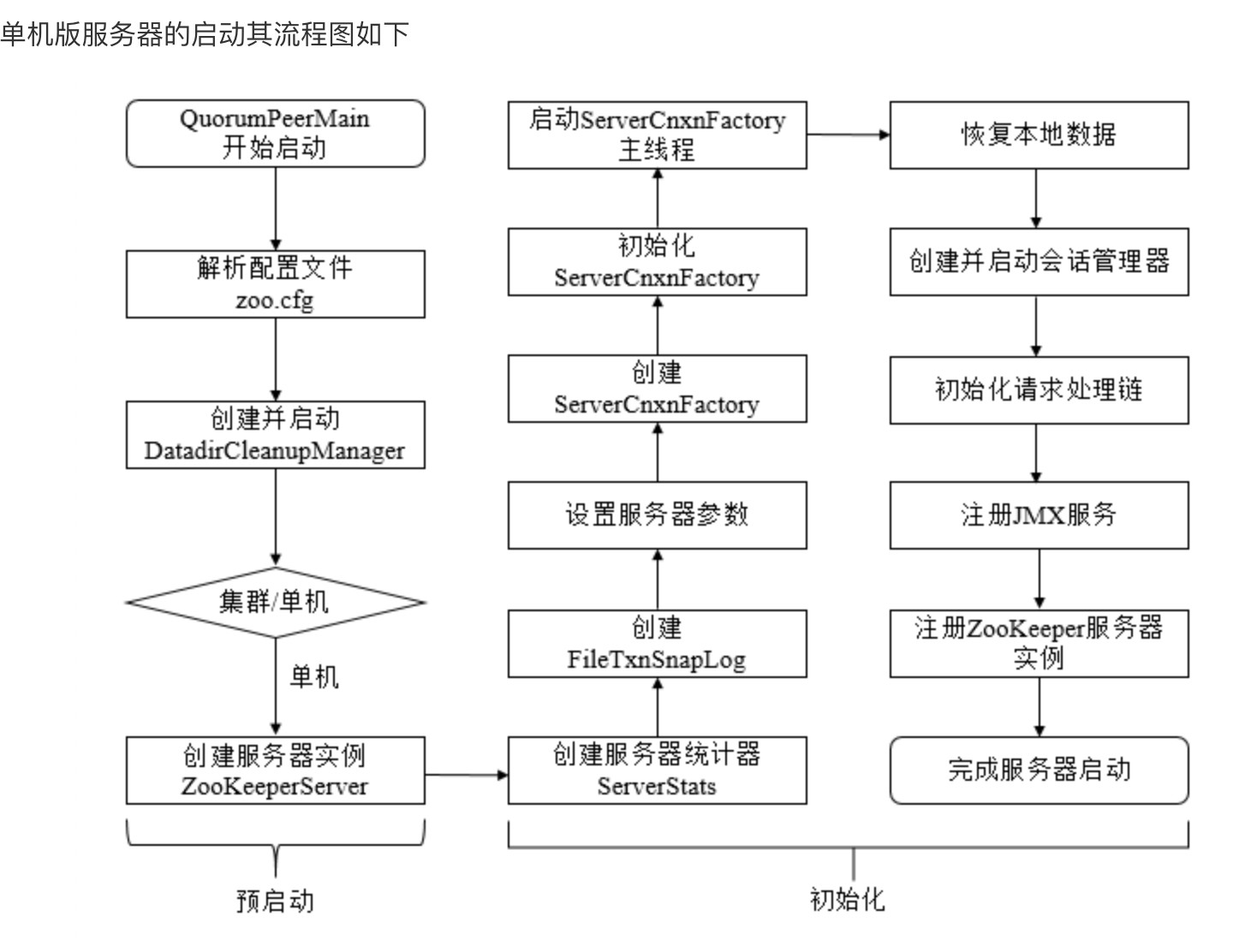


即客户端连接127.0.0.1:2182，获取节点/lg 的信息。

## zookeeper源码分析之单机模式服务端启动

#### ⼀、执⾏过程概述

单机模式的ZK服务端逻辑写在ZooKeeperServerMain类中，由⾥⾯的main函数启动，整个过程如下：



单机模式的委托启动类为：ZooKeeperServerMain

#### 服务端启动过程

看下ZooKeeperServerMain⾥⾯的main函数代码：

public static void main(String[] args) {

ZooKeeperServerMain main = new ZooKeeperServerMain(); main.initializeAndRun(args);

}

protected void initializeAndRun(String[] args)

throws ConfigException, IOException, AdminServerException

{

ServerConfig config = new ServerConfig();

//如果⼊参只有⼀个，则认为是配置⽂件的路径if (args.length == 1) {

config.parse(args[0]);

} else {

//否则是各个参数config.parse(args);

}

runFromConfig(config);

}

//省略部分代码，只保留了核⼼逻辑

public void runFromConfig(ServerConfig config) throws IOException,AdminServerException {

FileTxnSnapLog txnLog = null; try {

//初始化⽇志⽂件

txnLog = new FileTxnSnapLog(config.dataLogDir, config.dataDir);

//初始化ZkServer对象

final ZooKeeperServer zkServer = new ZooKeeperServer(txnLog, config.tickTime, config.minSessionTimeout, config.maxSessionTimeout, null);

txnLog.setServerStats(zkServer.serverStats());

if (config.getClientPortAddress() != null) {

//初始化server端IO对象，默认是NIOServerCnxnFactory cnxnFactory = ServerCnxnFactory.createFactory();

//初始化配置信息

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

//启动服务

cnxnFactory.startup(zkServer);

节点时

}

//container ZNodes是3.6版本之后新增的节点类型，Container类型的节点会在它没有⼦

// 被删除（新创建的Container节点除外），该类就是⽤来周期性的进⾏检查清理⼯作

containerManager = new ContainerManager(zkServer.getZKDatabase(),

zkServer.firstProcessor,

Integer.getInteger("znode.container.checkIntervalMs", (int) TimeUnit.MINUTES.toMillis(1)),

Integer.getInteger("znode.container.maxPerMinute", 10000)

);

containerManager.start();

//省略关闭逻辑

} catch (InterruptedException e) { LOG.warn("Server interrupted", e);

} finally {

if (txnLog != null) { txnLog.close();

}

}

}

⼩结：

zk单机模式启动主要流程： 1、注册jmx

2、解析ServerConﬁg配置对象

3、根据配置对象，运⾏单机zk服务

4、创建管理事务⽇志和快照FileTxnSnapLog对象，zookeeperServer对象，并设置zkServer的统计对 象

5、设置zk服务钩⼦，原理是通过设置CountDownLatch，调⽤ZooKeeperServerShutdownHandler的 handle⽅法，可以将触发shutdownLatch.await⽅法继续执⾏，即调⽤shutdown关闭单机服务

6、基于jetty创建zk的admin服务

7、创建连接对象cnxnFactory和secureCnxnFactory（安全连接才创建该对象），⽤于处理客户端的请 求

8、创建定时清除容器节点管理器，⽤于处理容器节点下不存在⼦节点的清理容器节点⼯作等

可以看到关键点在于解析配置跟启动两个⽅法，先来看下解析配置逻辑，对应上⾯的conﬁgure⽅法：

//依旧省略掉了部分逻辑

public void configure(InetSocketAddress addr, int maxcc, boolean secure) throws IOException {

maxClientCnxns = maxcc;

//会话超时时间sessionlessCnxnTimeout =

Integer.getInteger(ZOOKEEPER\_NIO\_SESSIONLESS\_CNXN\_TIMEOUT, 10000);

//过期队列

cnxnExpiryQueue = new ExpiryQueue<NIOServerCnxn>(sessionlessCnxnTimeout);

//过期线程，从cnxnExpiryQueue中读取数据，如果已经过期则关闭expirerThread = new ConnectionExpirerThread();

//根据CPU个数计算selector线程的数量

int numCores = Runtime.getRuntime().availableProcessors(); numSelectorThreads =

Integer.getInteger(ZOOKEEPER\_NIO\_NUM\_SELECTOR\_THREADS, Math.max((int) Math.sqrt((float) numCores/2), 1));

if (numSelectorThreads < 1) {

throw new IOException("numSelectorThreads must be at least 1");

}

//计算woker线程的数量

numWorkerThreads = Integer.getInteger(ZOOKEEPER\_NIO\_NUM\_WORKER\_THREADS, 2

* numCores);

//worker线程关闭时间

workerShutdownTimeoutMS = Long.getLong(ZOOKEEPER\_NIO\_SHUTDOWN\_TIMEOUT, 5000);

//初始化selector线程

for(int i=0; i<numSelectorThreads; ++i) { selectorThreads.add(new SelectorThread(i));

}

this.ss = ServerSocketChannel.open();

ss.socket().setReuseAddress(true); ss.socket().bind(addr); ss.configureBlocking(false);

//初始化accept线程，这⾥看出accept线程只有⼀个，⾥⾯会注册监听ACCEPT事件

acceptThread = new AcceptThread(ss, addr, selectorThreads);

}

再来看下启动逻辑：

public void startup(ZooKeeperServer zkServer) throws IOException, InterruptedException {

startup(zkServer, true);

}

//启动分了好⼏块，⼀个⼀个看

public void startup(ZooKeeperServer zks, boolean startServer) throws IOException, InterruptedException {

start(); setZooKeeperServer(zks); if (startServer) {

zks.startdata(); zks.startup();

}

}

//⾸先是start⽅法public void start() {

stopped = false;

//初始化worker线程池

if (workerPool == null) {

workerPool = new WorkerService("NIOWorker", numWorkerThreads, false);

}

//挨个启动select线程

for(SelectorThread thread : selectorThreads) { if (thread.getState() == Thread.State.NEW) {

thread.start();

}

}

//启动acceptThread线程

if (acceptThread.getState() == Thread.State.NEW) { acceptThread.start();

}

//启动expirerThread线程

if (expirerThread.getState() == Thread.State.NEW) { expirerThread.start();

}

}

//初始化数据结构

public void startdata() throws IOException, InterruptedException {

//初始化ZKDatabase，该数据结构⽤来保存ZK上⾯存储的所有数据

if (zkDb == null) {

//初始化数据数据，这⾥会加⼊⼀些原始节点，例如/zookeeper zkDb = new ZKDatabase(this.txnLogFactory);

}

//加载磁盘上已经存储的数据，如果有的话if (!zkDb.isInitialized()) {

loadData();

}

}

//启动剩余项⽬

public synchronized void startup() {

//初始化session追踪器

if (sessionTracker == null) { createSessionTracker();

}

//启动session追踪器startSessionTracker();

//建⽴请求处理链路

setupRequestProcessors(); registerJMX(); setState(State.RUNNING); notifyAll();

}

//这⾥可以看出，单机模式下请求的处理链路为：

//PrepRequestProcessor -> SyncRequestProcessor -> FinalRequestProcessor protected void setupRequestProcessors() {

RequestProcessor finalProcessor = new FinalRequestProcessor(this); RequestProcessor syncProcessor = new SyncRequestProcessor(this,

finalProcessor); ((SyncRequestProcessor)syncProcessor).start();

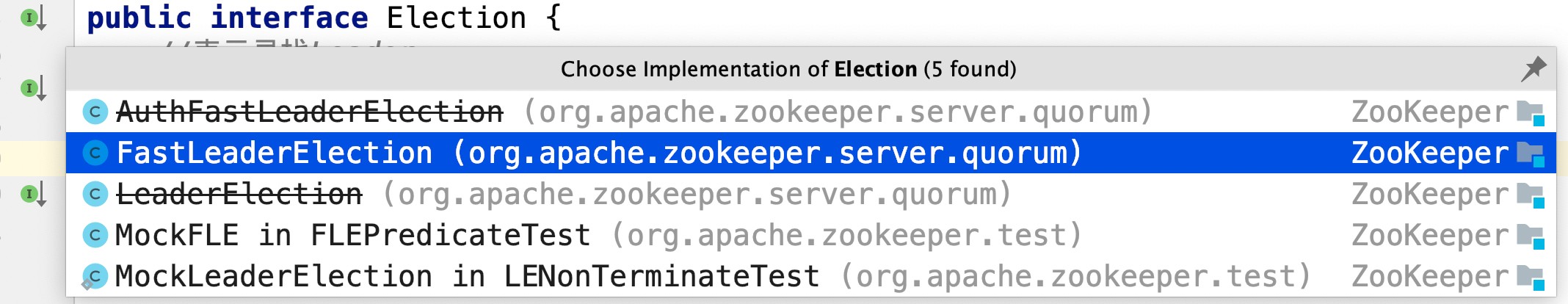
firstProcessor = new PrepRequestProcessor(this, syncProcessor); ((PrepRequestProcessor)firstProcessor).start();

}

## 源码分析之Leader选举（⼀）

分析Zookeeper中⼀个核⼼的模块，Leader选举。

#### 总体框架图

对于Leader选举，其总体框架图如下图所示

AuthFastLeaderElection，LeaderElection其在3.4.0之后的版本中已经不建议使⽤。

#### Election源码分析

public interface Election {

public Vote lookForLeader() throws InterruptedException; public void shutdown();

}

说明：

选举的⽗接⼝为Election，其定义了lookForLeader和shutdown两个⽅法，lookForLeader表示寻 找Leader，shutdown则表示关闭，如关闭服务端之间的连接。

## 源码分析之Leader选举（⼆）之FastLeaderElection

刚刚介绍了Leader选举的总体框架，接着来学习Zookeeper中默认的选举策略，FastLeaderElection。

#### FastLeaderElection源码分析

类的继承关系

public class FastLeaderElection implements Election {}

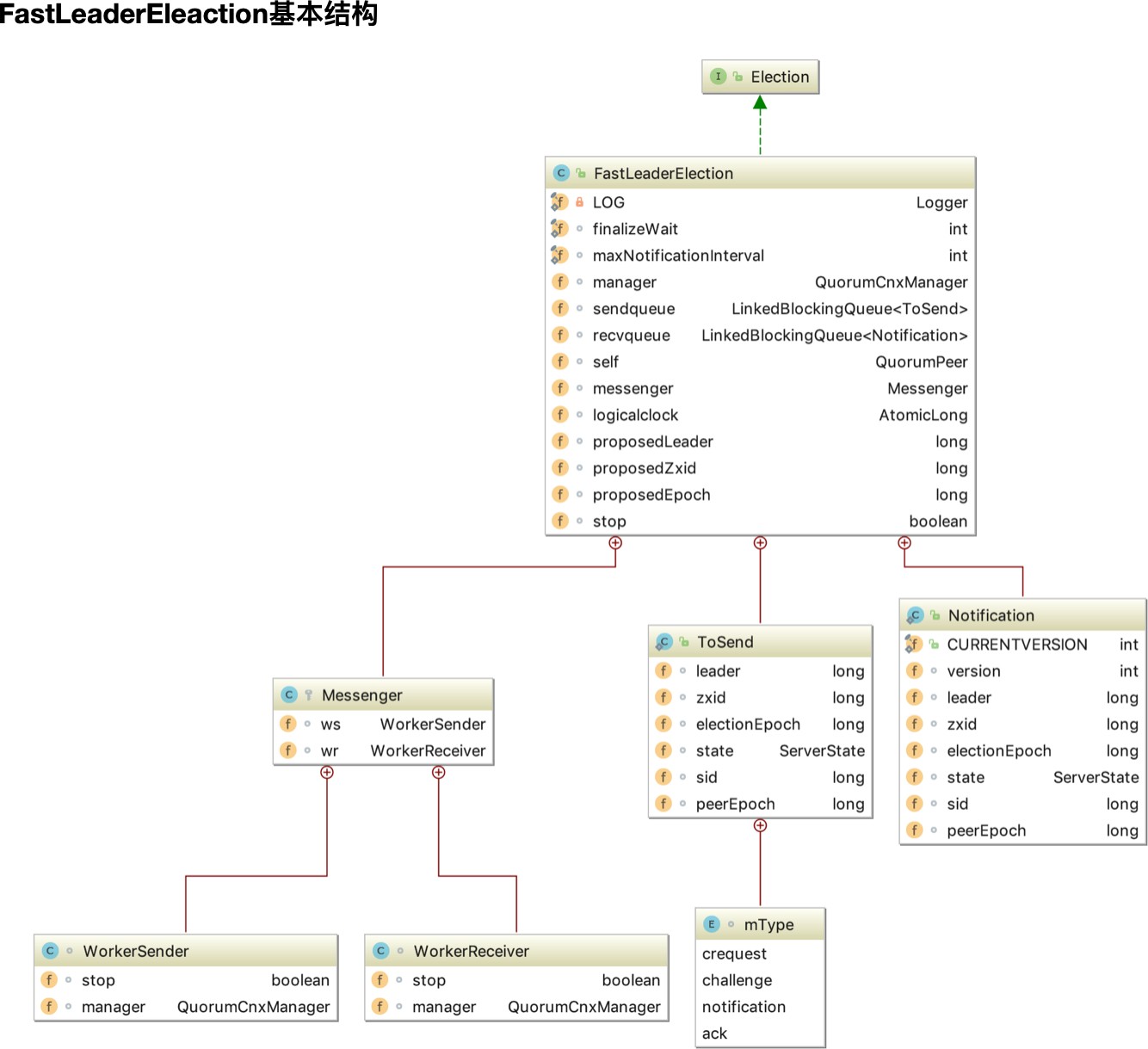
说明：FastLeaderElection实现了Election接⼝，重写了接⼝中定义的lookForLeader⽅法和shutdown

⽅法

在源码分析之前，我们⾸先介绍⼏个概念:

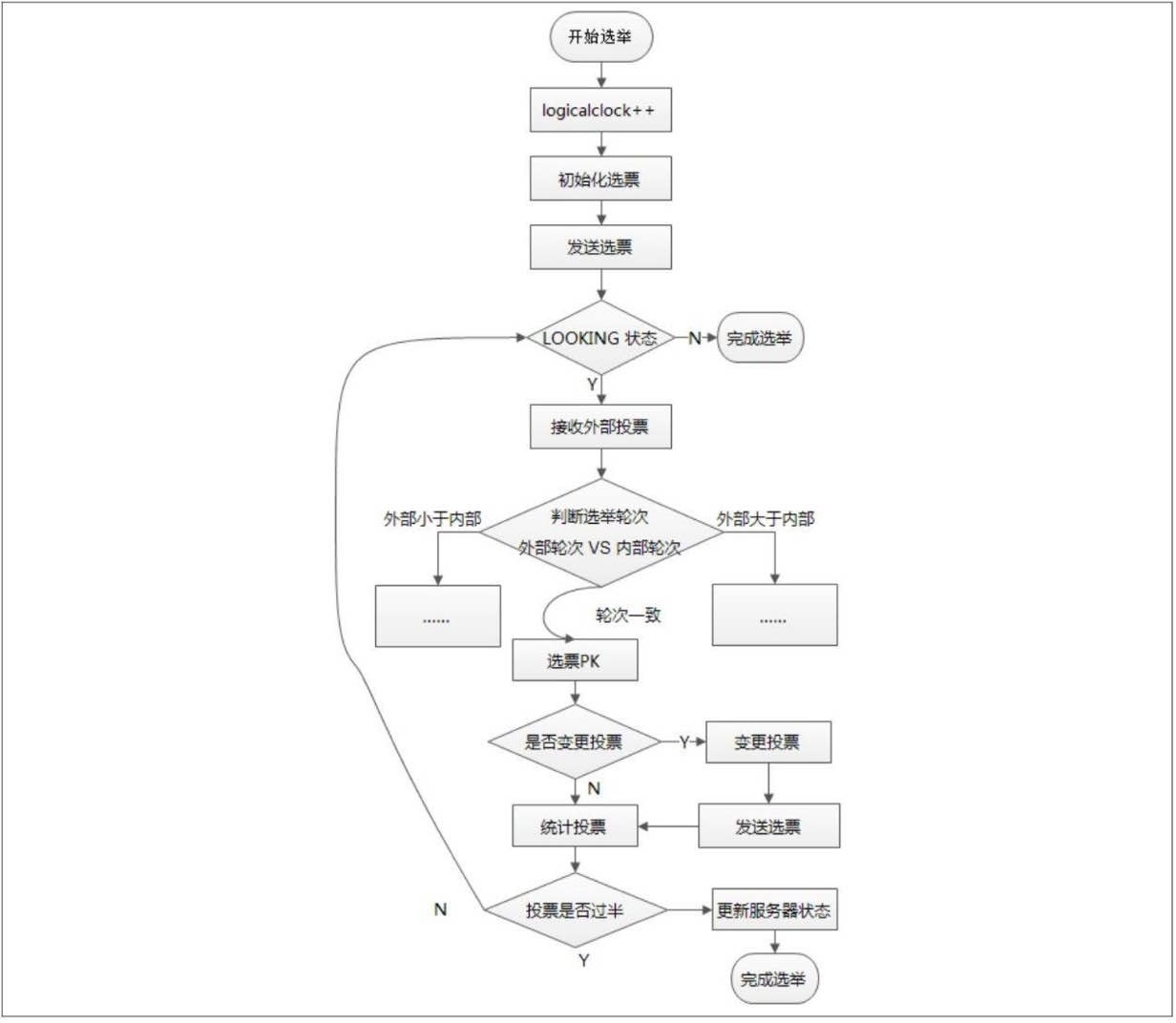
外部投票：特指其他服务器发来的投票。内部投票：服务器⾃身当前的投票。

选举轮次：ZooKeeper服务器Leader选举的轮次，即logical clock（逻辑时钟）。PK：指对内部投票和外部投票进⾏⼀个对⽐来确定是否需要变更内部投票。选票管理sendqueue： 选 票 发 送 队 列 ， ⽤ 于 保 存 待 发 送 的 选 票 。 recvqueue：选票接收队列，⽤于保存接收到的外部投票。



#### lookForLeader函数

当 ZooKeeper 服务器检测到当前服务器状态变成 LOOKING 时，就会触发 Leader选举，即调⽤lookForLeader⽅法来进⾏Leader选举。



public Vote lookForLeader() throws InterruptedException {

synchronized(this){

// ⾸先会将逻辑时钟⾃增，每进⾏⼀轮新的leader选举，都需要更新逻辑时钟

logicalclock++;

// 更新选票（初始化选票）

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

}

LOG.info("New election. My id = " + self.getId() +

", proposed zxid=0x" + Long.toHexString(proposedZxid));

// 向其他服务器发送⾃⼰的选票（已更新的选票）

sendNotifications();

之后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，就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，则⻢上建⽴连接，如 果已经建⽴了连接，则再次发送⾃⼰当前的内部投票，其流程如下

// 从recvqueue接收队列中取出投票

Notification n = recvqueue.poll(notTimeout,

TimeUnit.MILLISECONDS);

/\*

* Sends more notifications if haven't received enough.
* Otherwise processes new notification.

\*/

if(n == null){ // ⽆法获取选票

if(manager.haveDelivered()){ // manager已经发送了所有选票消息

（表示有连接）

// 向所有其他服务器发送消息

sendNotifications();

} else { // 还未发送所有消息（表示⽆连接）

// 连接其他每个服务器

manager.connectAll();

}

/\*

\* Exponential backoff

\*/

int tmpTimeOut = notTimeout\*2;

notTimeout = (tmpTimeOut < maxNotificationInterval? tmpTimeOut : maxNotificationInterval);

LOG.info("Notification time out: " + notTimeout);

}

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。

* + **外部投票的选举轮次⼤于内部投票**。若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，那么就会⽴即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。

* + **外部投票的选举轮次⼩于内部投票**。若服务器接收的外选票的选举轮次落后于⾃身的选举轮 次，那么Zookeeper就会直接忽略该外部投票，不做任何处理。
  + **外部投票的选举轮次等于内部投票**。此时可以开始进⾏选票PK，如果消息中的选票更优，则 需要更新本服务器内部选票，再发送给其他服务器。

之后再对选票进⾏归档操作，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，其中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， 然后开始统计投票，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，然后再进⾏最后⼀次确认，判断是否⼜有更优的选票产⽣，若

⽆，则终⽌投票，然后最终的选票，其流程如下

if (n.electionEpoch > logicalclock) { // 其选举周期⼤于逻辑时钟

// 重新赋值逻辑时钟

logicalclock = n.electionEpoch;

// 清空所有接收到的所有选票

recvset.clear();

n.peerEpoch,

if(totalOrderPredicate(n.leader, n.zxid,

getInitId(), getInitLastLoggedZxid(),

getPeerEpoch())) { // 进⾏PK，选出较优的服务器

// 更新选票

updateProposal(n.leader, n.zxid, n.peerEpoch);

} else { // ⽆法选出较优的服务器

// 更新选票

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

}

// 发送本服务器的内部选票消息

sendNotifications();

} else if (n.electionEpoch < logicalclock) { // 选举周期

⼩于逻辑时钟，不做处理，直接忽略

if(LOG.isDebugEnabled()){

LOG.debug("Notification election epoch is smaller than logicalclock. n.electionEpoch = 0x"

+ Long.toHexString(n.electionEpoch)

+ ", logicalclock=0x" +

Long.toHexString(logicalclock));

}

break;

} else if (totalOrderPredicate(n.leader, n.zxid,

n.peerEpoch,

{ // PK，选出较优的服务器

}

proposedLeader, proposedZxid, proposedEpoch))

// 更新选票

updateProposal(n.leader, n.zxid, n.peerEpoch);

// 发送消息

sendNotifications();

if(LOG.isDebugEnabled()){

LOG.debug("Adding vote: from=" + n.sid + ", proposed leader=" + n.leader + ", proposed zxid=0x" +

Long.toHexString(n.zxid) +

Long.toHexString(n.electionEpoch));

}

", proposed election epoch=0x" +

部投票

// 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

recvset.put(n.sid, new Vote(n.leader, n.zxid,

n.electionEpoch, n.peerEpoch));

if (termPredicate(recvset,

能选出leader

new Vote(proposedLeader, proposedZxid, logicalclock, proposedEpoch))) { // 若

leader

经接收的投票集合

n.peerEpoch,

// Verify if there is any change in the proposed

while((n = recvqueue.poll(finalizeWait,

TimeUnit.MILLISECONDS)) != null){ // 遍 历 已

if(totalOrderPredicate(n.leader, n.zxid, proposedLeader, proposedZxid,

proposedEpoch)){ // 选票有变更，⽐之前提议的Leader有更好的选票加⼊

// 将更优的选票放在recvset中

recvqueue.put(n); break;

}

}

new

self.getId()) ?

/\*

* This predicate is true once we don't read any
* relevant message from the reception queue

\*/

if (n == null) { // 表示之前提议的Leader已经是最优的

// 设置服务器状态

self.setPeerState((proposedLeader ==

ServerState.LEADING: learningState());

// 最终的选票

Vote endVote = new Vote(proposedLeader,

proposedZxid, logicalclock, proposedEpoch);

// 清空recvqueue队列的选票

leaveInstance(endVote);

// 返回选票

return endVote;

}

}

1. ⾃增选举轮次。 在 FastLeaderElection 实现中，有⼀个 logicalclock 属性，⽤于标识当前Leader的选举轮次，ZooKeeper规定了所有有效的投票都必须在同⼀轮次中。ZooKeeper在开始新⼀轮的投票 时，会⾸先对logicalclock进⾏⾃增操作。
2. 初始化选票。 在开始进⾏新⼀轮的投票之前，每个服务器都会⾸先初始化⾃⼰的选票。在图7-33中我们已经讲解了 Vote 数据结构，初始化选票也就是对 Vote 属性的初始化。在初始化阶段，每台服务器都会将⾃⼰推举为Leader，表7-10展示了⼀个初始化的选票。
3. 发送初始化选票。 在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队列中，由发送器WorkerSender负
4. 接收外部投票。 每台服务器都会不断地从 recvqueue 队列中获取外部投票。如果服务器发现⽆法获取到任何的外部投票，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连接。如果发现没有 建⽴连接，那么就会⻢上建⽴连接。如果已经建⽴了连接，那么就再次发送⾃⼰当前的内部投票。
5. 判断选举轮次。 当发送完初始化选票之后，接下来就要开始处理外部投票了。在处理外部投票的时候，会根据选举轮次来进⾏不同的处理。 · 外部投票的选举轮次⼤于内部投票。如果服务器发现⾃⼰的选举轮次已经落后于该外部投票对应服务器的选举轮次，那么就会⽴即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（关于P K的逻辑会在步骤6中统⼀讲解），最终再将内部投票发送出去。 · 外部投票的选举轮次

⼩于内部投票。 如果接收到的选票的选举轮次落后于服务器⾃身的，那么ZooKeeper就会直接忽略该外部投票，不做任何处理，并返回步骤4。

* 外部投票的选举轮次和内部投票⼀致。 这也是绝⼤多数投票的场景，如外部投票的选举轮次和内部投票⼀致的话，那么就开始进⾏选票PK。 总的来说，只有在同⼀个选举轮次的投票才是有效的投票。

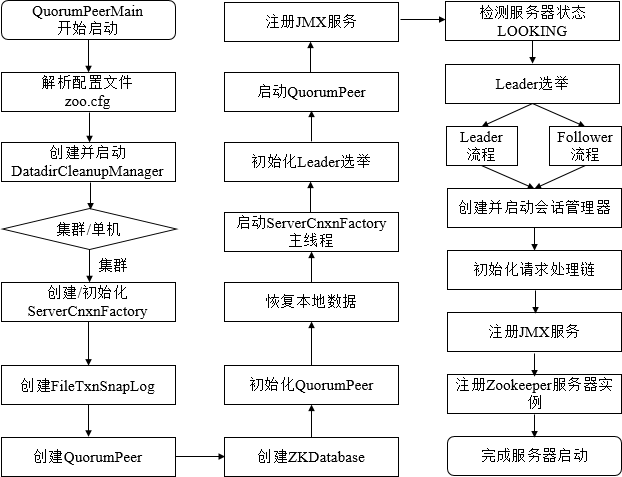
1. 选票PK。 在步骤5中提到，在收到来⾃其他服务器有效的外部投票后，就要进⾏选票PK了——也就是FastLeaderElection.totalOrderPredicate⽅法的核⼼逻辑。选票PK的⽬的是为了确定当前服务器是否 需要变更投票，主要从选举轮次、ZXID和 SID 三个因素来考虑，具体条件如下：在选票 PK 的时候依次判断，符合任意⼀个条件就需要进⾏投票变更。 · 如果外部投票中被推举的Leader服务器的选举轮次⼤于内部投票，那么就需要进⾏投票变更。 · 如果选举轮次⼀致的话，那么就对⽐两者的ZXID。如果外部投票的ZXID⼤于内部投票，那么就需要进⾏投票变更。 · 如果两者的 ZXID ⼀致，那么就对⽐两者的SID。如果外部投票的 SID ⼤于内部投票，那么就需要进⾏投票变更。 7.变更投票。 通过选票PK后，如果确定了外部投票优于内部投票（所谓的“优于”，是指外部投票所推举的服务器更适合成为Leader）， 那么就进⾏投票变更——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。 ⽆论是否进⾏了投票变更，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——按照服务器对应 的SID来区分，例如，{（1，vote1），（2，vote2），…}。 9.统计投票。 完成了选票归档之后，就可以开始统计投票了。统计投票的过程就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的服务器认可了该内部投票，则终⽌投票。否则返回步骤4。 10.更新服务器状态。 统计投票后，如果已经确定可以终⽌投票，那么就开始更新服务器状态。服务器会⾸先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，如果是⾃⼰的话，那么就会将⾃⼰的服务器状态更新为 LEADING。如果⾃⼰不是被选举产⽣的 Leader 的话，那么就会根据具体情况来确定⾃⼰是FOLLOWING或是OBSERVING。 以上 10 个步骤，就是 FastLeaderElection 选举算法的核⼼步骤，其 中步骤 4～9 会经过⼏轮循环，直到Leader选举产⽣。另外还有⼀个细节需要注意，就是在完成步骤9 之后，如果统计投票发现已经有过半的服务器认可了当前的选票，这个时候，ZooKeeper 并不会⽴即进

⼊步骤 10 来更新服务器状态，⽽是会等待⼀段时间（默认是 200 毫秒）来确定是否有新的更优的投票

## zookeeper源码分析之集群模式服务端

#### 执⾏流程图



**源码分析**

集群模式下启动所有的ZK节点启动⼊⼝都是QuorumPeerMain类的main⽅法。 main⽅法加载配置⽂件以后，最终会调⽤到QuorumPeer的start⽅法，来看下：

public synchronized void start() {

//校验ServerId是否合法

if (!getView().containsKey(myid)) {

throw new RuntimeException("My id " + myid + " not in the peer list");

}

//载⼊之前持久化的⼀些信息loadDataBase();

//启动线程监听

startServerCnxnFactory(); try {

adminServer.start();

} catch (AdminServerException e) { LOG.warn("Problem starting AdminServer", e); System.out.println(e);

}

//初始化选举投票以及算法startLeaderElection();

//当前也是⼀个线程，注意run⽅法

super.start();

}

我们已经知道了当⼀个节点启动时需要先发起选举寻找Leader节点，然后再根据Leader节点的事务信息进⾏同步，最后开始对外提供服务，这⾥我们先来看下初始化选举的逻辑，即上⾯的startLeaderElection⽅法：

synchronized public void startLeaderElection() { try {

//所有节点启动的初始状态都是LOOKING，因此这⾥都会是创建⼀张投⾃⼰为Leader的票

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

getCurrentEpoch());

}

} catch(IOException e) {

//异常处理

}

//初始化选举算法，electionType默认为3

this.electionAlg = createElectionAlgorithm(electionType);

}

protected Election createElectionAlgorithm(int electionAlgorithm){ Election le = null;

switch (electionAlgorithm) { case 1:

//忽略

case 2:

//忽略case 3:

//electionAlgorithm默认是3，直接⾛到这⾥

qcm = createCnxnManager();

//监听选举事件的listener

QuorumCnxManager.Listener listener = qcm.listener; if(listener != null){

//开启监听器

listener.start();

//初始化选举算法

FastLeaderElection fle = new FastLeaderElection(this, qcm);

//发起选举fle.start(); le = fle;

} else {

LOG.error("Null listener when initializing cnx manager");

}

break; default:

//忽略

}

return le;

}

接下来，回到QuorumPeer类中start⽅法的最后⼀⾏super.start()，QuorumPeer本身也是⼀个线程 类，⼀起来看下它的run⽅法：

public void run() { try {

while (running) {

//根据当前节点的状态执⾏不同流程switch (getPeerState()) { case LOOKING:

try {

//寻找Leader节点setCurrentVote(makeLEStrategy().lookForLeader());

} catch (Exception e) { setPeerState(ServerState.LOOKING);

}

break;

case OBSERVING:

try {

// 当 前 节 点 启 动 模 式 为 Observer setObserver(makeObserver(logFactory));

//与Leader节点进⾏数据同步

observer.observeLeader();

} catch (Exception e) {

} finally {

}

break;

case FOLLOWING:

try {

// 当 前 节 点 启 动 模 式 为 Follower setFollower(makeFollower(logFactory));

//与Leader节点进⾏数据同步

follower.followLeader();

} catch (Exception e) {

} finally {

}

break; case LEADING:

try {

// 当 前 节 点 启 动 模 式 为 Leader setLeader(makeLeader(logFactory));

//发送⾃⼰成为Leader的通知

leader.lead(); setLeader(null);

} catch (Exception e) {

} finally {

}

break;

}

}

}

}

节点初始化的状态为LOOKING，因此启动时直接会调⽤lookForLeader⽅法发起Leader选举，⼀起看 下：

public Vote lookForLeader() throws InterruptedException { try {

Map<Long, Vote> recvset = new HashMap<Long, Vote>(); Map<Long, Vote> outofelection = new HashMap<Long, Vote>();

//向所有投票节点发送⾃⼰的投票信息

sendNotifications();

while ((self.getPeerState() == ServerState.LOOKING) && (!stop)){

//读取各个节点返回的投票信息

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

//超时重发

if(n == null){

//如果前⾯待发送的消息已经全部发送，则重新发送if(manager.haveDelivered()){

sendNotifications();

} else {

//否则尝试与各个节点建⽴连接manager.connectAll();

}

//退避算法修改下次等待时间

int tmpTimeOut = notTimeout\*2;

notTimeout = (tmpTimeOut < maxNotificationInterval? tmpTimeOut : maxNotificationInterval);

}

else if (validVoter(n.sid) && validVoter(n.leader)) { switch (n.state) {

case LOOKING:

//如果节点的周期⼤于⾃⼰的

if (n.electionEpoch > logicalclock.get()) { logicalclock.set(n.electionEpoch);

//清除已收到的投票信息

recvset.clear();

//两个节点根据epoch，zxid，serverId来判断新的投票信息if(totalOrderPredicate(n.leader, n.zxid,

n.peerEpoch, getInitId(), getInitLastLoggedZxid(), getPeerEpoch())) {

updateProposal(n.leader, n.zxid, n.peerEpoch);

} else {

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

}

//修改选举周期以及投票信息，发起新⼀轮投票sendNotifications();

} else if (n.electionEpoch < logicalclock.get()) {

n.peerEpoch,

{

//这⾥的break是跳出switch语句，别跟循环弄混break;

} else if (totalOrderPredicate(n.leader, n.zxid, proposedLeader, proposedZxid, proposedEpoch))

//如果对⽅的epoch，zxid，serverId⽐⾃⼰⼤

//则更新⾃⼰的投票给n的投票节点

updateProposal(n.leader, n.zxid, n.peerEpoch);

//重新发送⾃⼰新的投票信息sendNotifications();

}

//把节点的投票信息记录下

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

//统计投票信息，判断当前选举是否可以结束，也就是收到的票数信息已

经⾜够确认Leader

proposedZxid,

if (termPredicate(recvset, new Vote(proposedLeader,

logicalclock.get(), proposedEpoch))) { while((n = recvqueue.poll(finalizeWait,

TimeUnit.MILLISECONDS)) != null){

if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch, proposedLeader, proposedZxid, proposedEpoch)){

recvqueue.put(n); break;

}

}

//如果没有多余的投票信息则可以结束本次选举周期if (n == null) {

//根据serverId修改当前节点的类型

self.setPeerState((proposedLeader == self.getId()) ? ServerState.LEADING: learningState());

Vote endVote = new Vote(proposedLeader,

proposedZxid, proposedEpoch);

}

//清空接收消息队列leaveInstance(endVote);

//返回最终的投票信息

return endVote;

}

break;

case OBSERVING:

//Observer节点不参与投票，忽略break;

case FOLLOWING:

case LEADING:

//如果周期相同，说明当前节点参与了这次选举if(n.electionEpoch == logicalclock.get()){

//保存投票信息

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

//判断当前节点收到的票数是否可以结束选举

if(termPredicate(recvset, new Vote(n.leader,

n.zxid, n.electionEpoch,

n.peerEpoch, n.state)) n.leader, n.electionEpoch)) {

&& checkLeader(outofelection, self.setPeerState((n.leader == self.getId()) ?

ServerState.LEADING: learningState());

n.peerEpoch);

Vote endVote = new Vote(n.leader, n.zxid,

leaveInstance(endVote); return endVote;

}

}

的信息是否⼀致

//把Leader跟Follower的投票信息加⼊outofelection，确认下它们

outofelection.put(n.sid, new Vote(n.leader,

IGNOREVALUE, IGNOREVALUE, n.peerEpoch, n.state));

if (termPredicate(outofelection, new Vote(n.leader, IGNOREVALUE, IGNOREVALUE, n.peerEpoch,

n.state)) IGNOREVALUE)) {

n.peerEpoch);

&& checkLeader(outofelection, n.leader,

synchronized(this){ logicalclock.set(n.electionEpoch); self.setPeerState((n.leader == self.getId()) ?

ServerState.LEADING: learningState());

}

Vote endVote = new Vote(n.leader, n.zxid,

leaveInstance(endVote); return endVote;

}

break; default:

break;

}

}

}

return null;

}

经过上⾯的发起投票，统计投票信息最终每个节点都会确认⾃⼰的身份，节点根据类型的不同会执⾏以 下逻辑：

1. 如果是Leader节点，⾸先会想其他节点发送⼀条NEWLEADER信息，确认⾃⼰的身份，等到各个节点的ACK消息以后开始正式对外提供服务，同时开启新的监听器，处理新节点加⼊的逻辑。
2. 如果是Follower节点，⾸先向Leader节点发送⼀条FOLLOWERINFO信息，告诉Leader节点⾃⼰已 处理的事务的最⼤Zxid，然后Leader节点会根据⾃⼰的最⼤Zxid与Follower节点进⾏同步，如果Follower节点落后的不多则会收到Leader的DIFF信息通过内存同步，如果Follower节点落后的很 多则会收到SNAP通过快照同步，如果Follower节点的Zxid⼤于Leader节点则会收到TRUNC信息忽 略多余的事务。
3. 如果是Observer节点，则与Follower节点相同