[ZooKeeper 3](#_Toc2129694223)

[1 单机搭建环境 3](#_Toc1660861412)

[1.1 安装ant 3](#_Toc1105307778)

[1.2 导入操作 3](#_Toc1174278296)

[1.3 启动 4](#_Toc720604942)

[2 伪集群搭建环境 7](#_Toc1546974761)

[3. 主要类介绍 13](#_Toc420293898)

[3.1 QuorumPeerMain（单机和集群的启动类） 13](#_Toc805828703)

[3.1.1 main方法 14](#_Toc1531133339)

[3.1.2 initializeAndRun方法 15](#_Toc461486572)

[3.2 QuorumPeerConfig（配置解析类） 16](#_Toc1641366287)

[3.2.1 主要属性 16](#_Toc2015739894)

[3.2.2 parse方法 21](#_Toc1985867033)

[3.2.3 parseProperties方法 23](#_Toc276381957)

[3.3 DatadirCleanupManager（快照数据清理任务管理器） 29](#_Toc144422838)

[3.3.1 主要属性 29](#_Toc658117156)

[3.3.2 start方法 30](#_Toc1434258842)

[3.4 ZooKeeperServerMain（单机启动） 30](#_Toc84419919)

[3.4.1 主要属性 31](#_Toc1506371613)

[3.4.2 main方法 31](#_Toc902985208)

[3.4.3 initializeAndRun 32](#_Toc205457507)

[3.4.4 runFromConfig 33](#_Toc2118099420)

[3.4.5 shutdown 35](#_Toc60535621)

[3.5 ZooKeeperServer （zk服务器） 36](#_Toc1662417116)

[3.5.1 属性 36](#_Toc1482221142)

[3.5.2 startdata 37](#_Toc880428394)

[3.5.3 loadData 38](#_Toc1197690128)

[3.5.4 startup 启动 39](#_Toc1213757965)

[3.5.5 createSessionTracker 39](#_Toc682954902)

[3.5.6 startSessionTracker 40](#_Toc122944699)

[3.5.7 expire 40](#_Toc452287679)

[3.5.8 submitRequest(提交请求) 41](#_Toc1654394220)

[3.5.9 setupRequestProcessors 42](#_Toc1932877831)

[3.6 ServerCnxnFactory 43](#_Toc892577448)

[3.6.1 startup 43](#_Toc1375894241)

[3.7 NIOServerCnxnFactory（NIO） 44](#_Toc550597591)

[3.7.1 startup(启动类) 44](#_Toc386677014)

[3.7.2 start 45](#_Toc595058476)

[3.8 NettyServerCnxnFactory 45](#_Toc316462053)

[3.9 ZooKeeperThread (zk线程) 46](#_Toc1608214799)

[3.10 WorkerService(zk线程池) 46](#_Toc1036945651)

[3.11 SessionTrackerImpl(会话管理器，是个线程) 46](#_Toc1115760952)

[3.11.1 类结构 46](#_Toc767114660)

[3.11.2 内部类 46](#_Toc1551757679)

[3.11.3 主要属性 47](#_Toc1349901785)

[3.11.4 SessionTrackerImpl的构造函数 48](#_Toc1782053587)

[3.11.5 run方法 50](#_Toc20212000)

[3.12 ExpiryQueue(过期会话对列) 53](#_Toc400667774)

[3.12.1 主要属性 53](#_Toc1662044273)

[3.13 ServerCnxn(服务端连接管理类) 53](#_Toc1658299782)

[3.13.1 属性 53](#_Toc1001665308)

[3.14 Watcher 55](#_Toc864522723)

[3.15 RequestProcessor（请求处理器） 55](#_Toc159049859)

[3.15.1 PreRequestProcessor(第一个处理器) 55](#_Toc1681323345)

[3.15.2 SyncRequestProcessor 55](#_Toc1411632189)

[3.15.3 FinalRequestProcessor(最后一个处理器) 55](#_Toc2050352114)

# ZooKeeper

# 1 单机搭建环境

## 1.1 安装ant

下载地址：<https://ant.apache.org/bindownload.cgi>  
下载版本：1.10.7包：[apache-ant-1.10.7-bin.zip](http://mirror.bit.edu.cn/apache//ant/binaries/apache-ant-1.10.7-bin.zip) [ [PGP](https://www.apache.org/dist/ant/binaries/apache-ant-1.10.7-bin.zip.asc) ] [ [SHA512](https://www.apache.org/dist/ant/binaries/apache-ant-1.10.7-bin.zip.sha512) ]

解压，配置环境变量：

mac的配置

控制台：

vim ~/.bash\_profile

添加ANT\_HOME变量，将ANT\_HOM加入到PATH中

export ANT\_HOME=/Users/xxx/Documents/code\_application/apache-ant-1.10.7

PATH=$PATH:$JAVA\_HOME/bin:$PATH:$ANT\_HOME/bin:

win：

解压，配置环境变量ANT\_HOME=D:\apache-ant-1.10.7-bin\apache-ant-1.10.7  
编辑Path，增加%ANT\_HOME%\bin

编辑classpath，增加%ANT\_HOME%\lib

## 1.2 导入操作

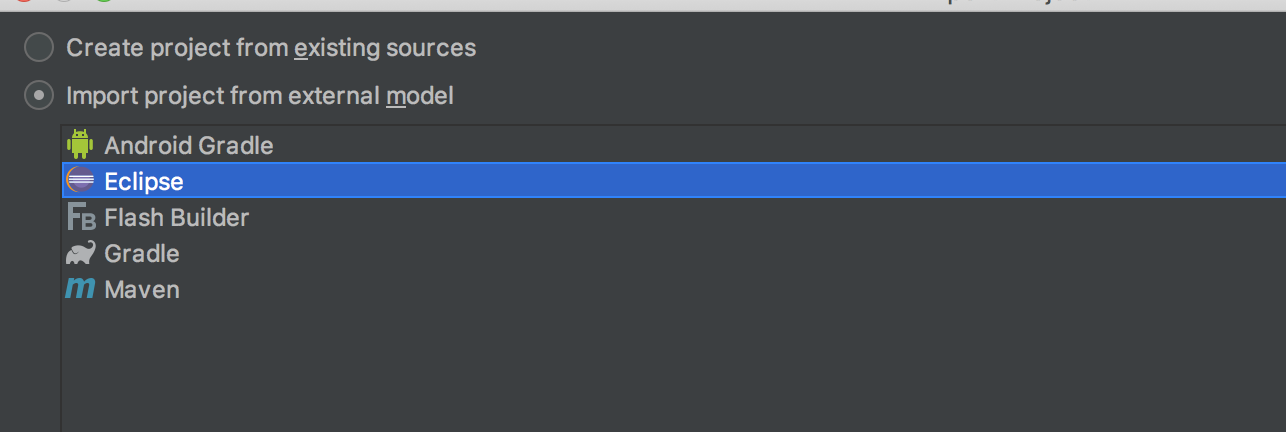
1、下载zk源码： <https://github.com/apache/zookeeper>；版本：3.5.6

2、控制台切换到zk源码目录，执行

ant eclipse（正常要下载一会依赖，未出现异常）

1. 导入idea

导出过程中，要选Eclipse，不要选其他的，一路next

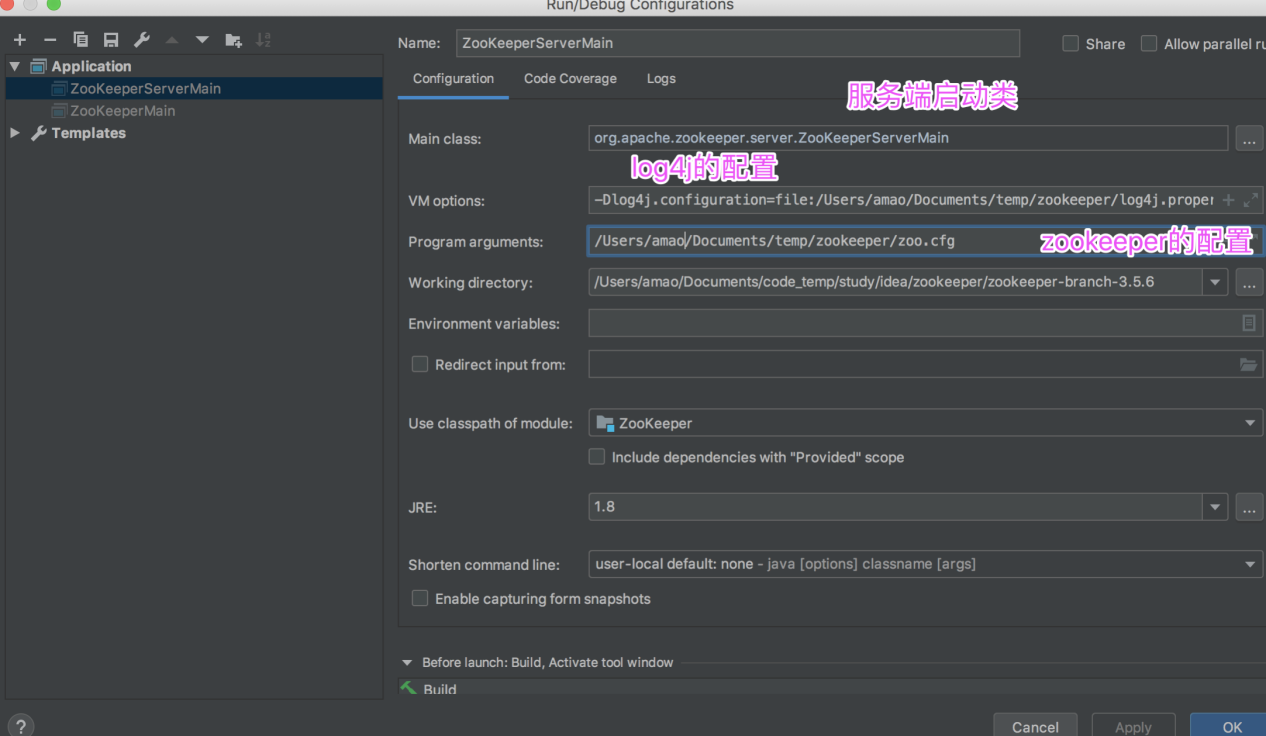


## 1.3 启动

1. 需要在org.apache.zookeeper.version包下面新建一个类(直接启动会包缺少这个类错误)

package org.apache.zookeeper.version;  
  
public interface Info {  
  
 int *MAJOR*=1;  
 int *MINOR*=0;  
 int *MICRO*=0;  
 String *QUALIFIER*=null;  
 int *REVISION*=-1; //*TODO: remove as related to SVN VCS* String *REVISION\_HASH*="1";  
 String *BUILD\_DATE*="2019-3-4";  
  
}

1. 启动ZooKeeperServerMain。会出现警告，需要配置

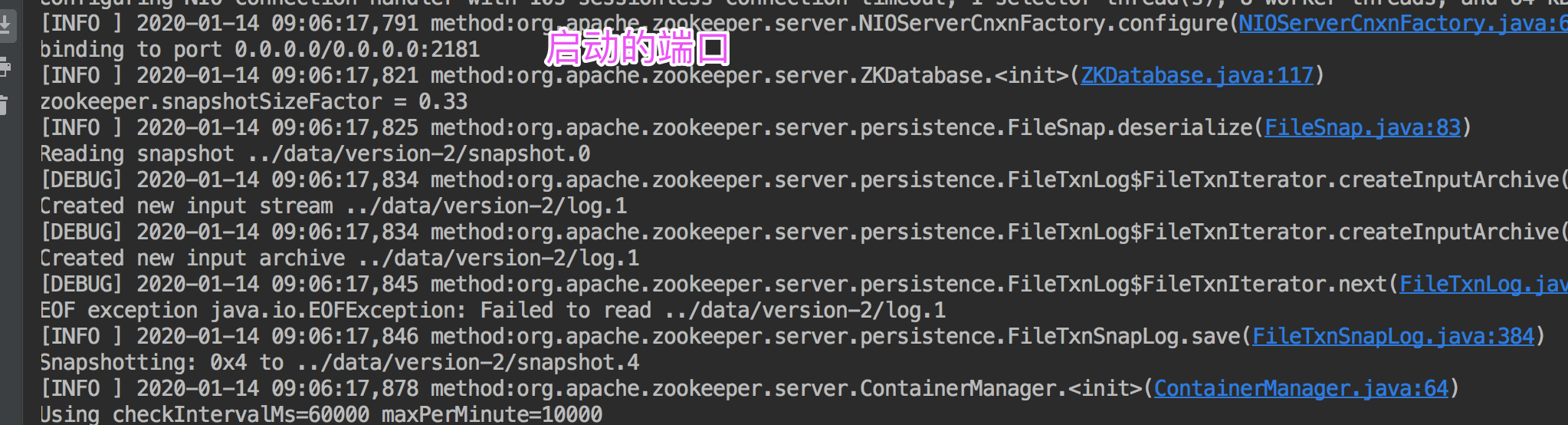


log4j.properties注意修改路径

|  |
| --- |
| log4j.rootLogger = debug,stdout,D,E  ### 输出信息到控制抬 ###  log4j.appender.stdout = org.apache.log4j.ConsoleAppender  log4j.appender.stdout.Target = System.out  log4j.appender.stdout.layout = org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n  ### 输出DEBUG 级别以上的日志到=E://logs/error.log ###  log4j.appender.D = org.apache.log4j.DailyRollingFileAppender  log4j.appender.D.File = /Users/amao/Documents/temp/zookeeper/logs/log.log  log4j.appender.D.Append = true  log4j.appender.D.Threshold = DEBUG  log4j.appender.D.layout = org.apache.log4j.PatternLayout  log4j.appender.D.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n  ### 输出ERROR 级别以上的日志到=E://logs/error.log ###  log4j.appender.E = org.apache.log4j.DailyRollingFileAppender  log4j.appender.E.File =/Users/amao/Documents/temp/zookeeper/logs/error.log  log4j.appender.E.Append = true  log4j.appender.E.Threshold = ERROR  log4j.appender.E.layout = org.apache.log4j.PatternLayout  log4j.appender.E.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n |

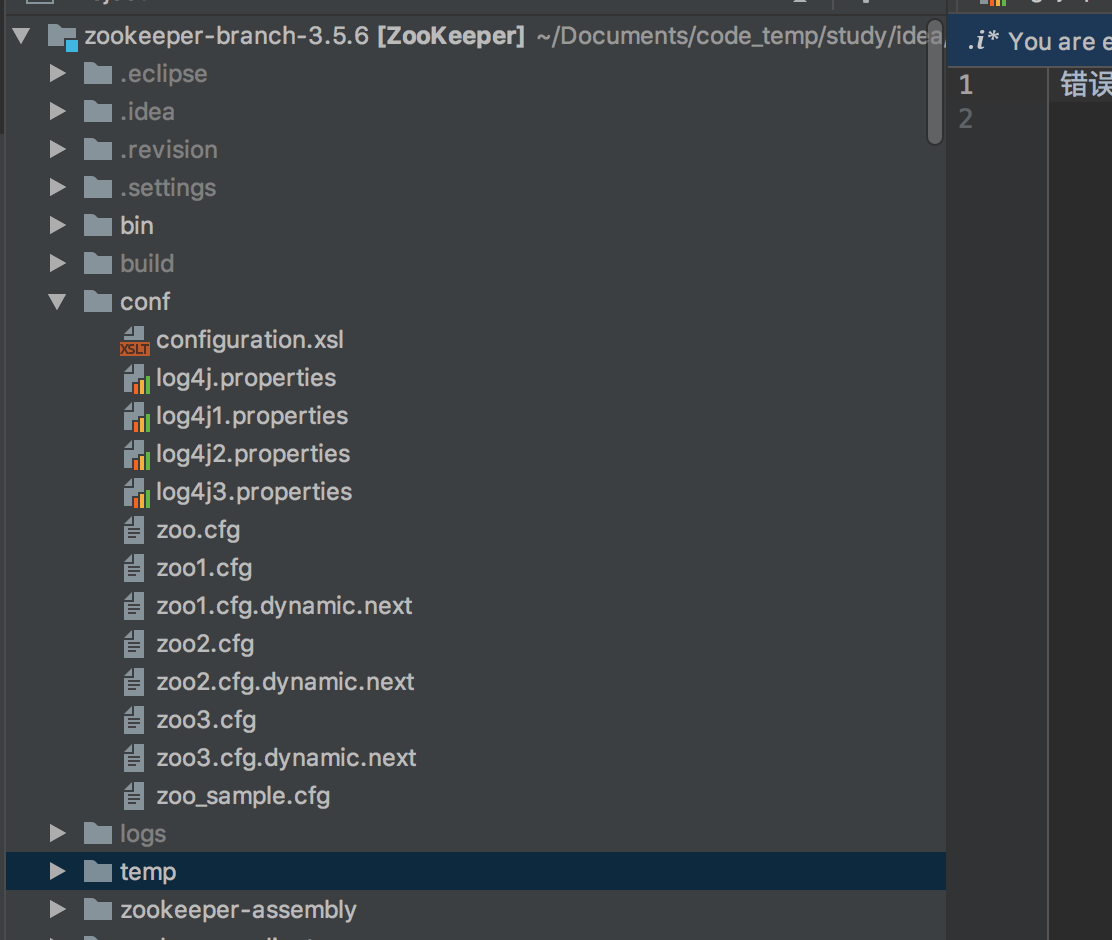
zoo.cfg

|  |
| --- |
| # The number of milliseconds of each tick  tickTime=2000  # The number of ticks that the initial  # synchronization phase can take  initLimit=10  # The number of ticks that can pass between  # sending a request and getting an acknowledgement  syncLimit=5  # the directory where the snapshot is stored.  # do not use /tmp for storage, /tmp here is just  # example sakes.  dataDir=../data  # the port at which the clients will connect  clientPort=2181  # the maximum number of client connections.  # increase this if you need to handle more clients  #maxClientCnxns=60  #  # Be sure to read the maintenance section of the  # administrator guide before turning on autopurge.  #  # http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance  #  # The number of snapshots to retain in dataDir  #autopurge.snapRetainCount=3  # Purge task interval in hours  # Set to "0" to disable auto purge feature  #autopurge.purgeInterval=1 |



# 2 伪集群搭建环境

在导入idea中的有个bin目录，目录中有conf目录，该目录中log4j.properties和zoo\_sample.cfg，confuguration.xsl 三个配置文件



将log4j.properties和zoo\_sample.cfg分别复制份，做如下配置

log4j.properties ,三个只有zookeeper.log.dir的值是三个文件夹，如下

/Users/amao/Documents/code\_temp/study/idea/zookeeper/zookeeper-branch-3.4.14/logs/log1

/Users/amao/Documents/code\_temp/study/idea/zookeeper/zookeeper-branch-3.4.14/logs/log2

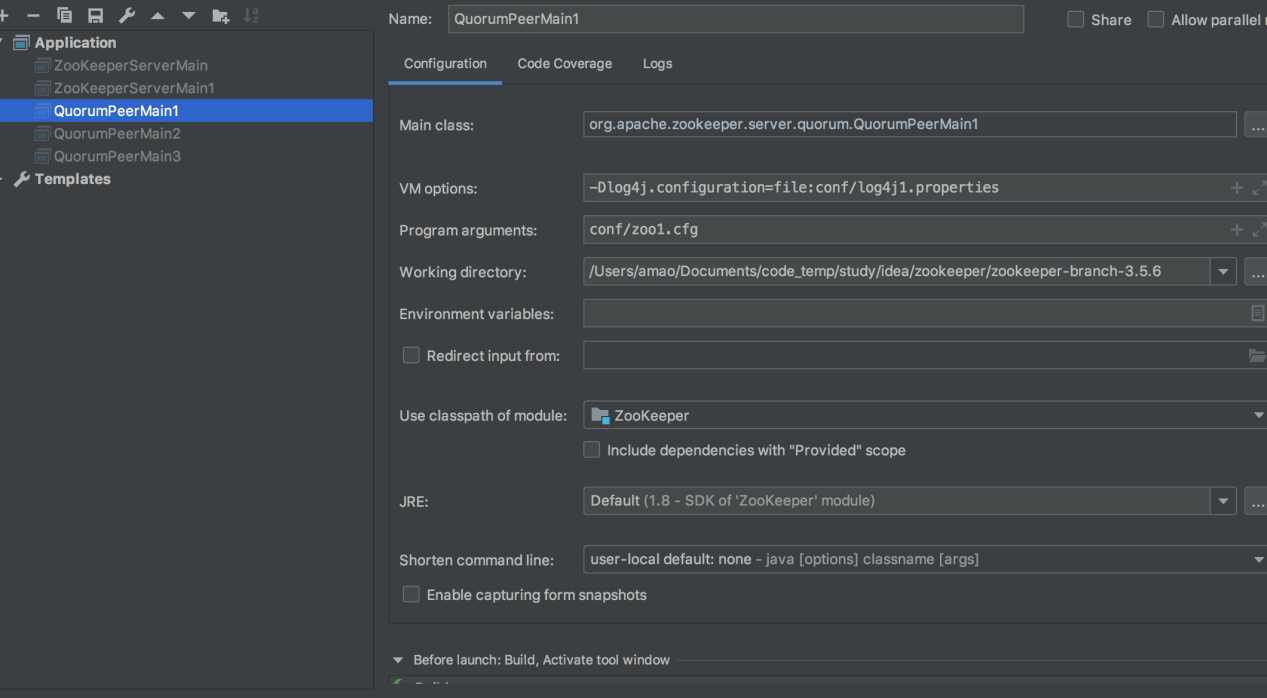
/Users/amao/Documents/code\_temp/study/idea/zookeeper/zookeeper-branch-3.4.14/logs/log3

|  |
| --- |
| # Copyright 2012 The Apache Software Foundation # # Licensed to the Apache Software Foundation (ASF) under one # or more contributor license agreements. See the NOTICE file # distributed with this work for additional information # regarding copyright ownership. The ASF licenses this file # to you under the Apache License, Version 2.0 (the # "License"); you may not use this file except in compliance # with the License. You may obtain a copy of the License at # # http://www.apache.org/licenses/LICENSE-2.0 # # Unless required by applicable law or agreed to in writing, software # distributed under the License is distributed on an "AS IS" BASIS, # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. # See the License for the specific language governing permissions and # limitations under the License.  # Define some default values that can be overridden by system properties zookeeper.root.logger=DEBUG, CONSOLE, ROLLINGFILE  zookeeper.console.threshold=INFO  zookeeper.log.dir=/Users/amao/Documents/code\_temp/study/idea/zookeeper/zookeeper-branch-3.4.14/logs/log1 zookeeper.log.file=zookeeper.log zookeeper.log.threshold=INFO zookeeper.log.maxfilesize=256MB zookeeper.log.maxbackupindex=20  zookeeper.tracelog.dir=${zookeeper.log.dir} zookeeper.tracelog.file=zookeeper\_trace.log  log4j.rootLogger=${zookeeper.root.logger}  # # console # Add "console" to rootlogger above if you want to use this # log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender log4j.appender.CONSOLE.Threshold=${zookeeper.console.threshold} log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] - %-5p [%t:%C{1}@%L] - %m%n  # # Add ROLLINGFILE to rootLogger to get log file output # log4j.appender.ROLLINGFILE=org.apache.log4j.RollingFileAppender log4j.appender.ROLLINGFILE.Threshold=${zookeeper.log.threshold} log4j.appender.ROLLINGFILE.File=${zookeeper.log.dir}/${zookeeper.log.file} log4j.appender.ROLLINGFILE.MaxFileSize=${zookeeper.log.maxfilesize} log4j.appender.ROLLINGFILE.MaxBackupIndex=${zookeeper.log.maxbackupindex} log4j.appender.ROLLINGFILE.layout=org.apache.log4j.PatternLayout log4j.appender.ROLLINGFILE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] - %-5p [%t:%C{1}@%L] - %m%n  # # Add TRACEFILE to rootLogger to get log file output # Log TRACE level and above messages to a log file # log4j.appender.TRACEFILE=org.apache.log4j.FileAppender log4j.appender.TRACEFILE.Threshold=TRACE log4j.appender.TRACEFILE.File=${zookeeper.tracelog.dir}/${zookeeper.tracelog.file}  log4j.appender.TRACEFILE.layout=org.apache.log4j.PatternLayout ### Notice we are including log4j's NDC here (%x) log4j.appender.TRACEFILE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] - %-5p [%t:%C{1}@%L][%x] - %m%n |

zoo1.cfg

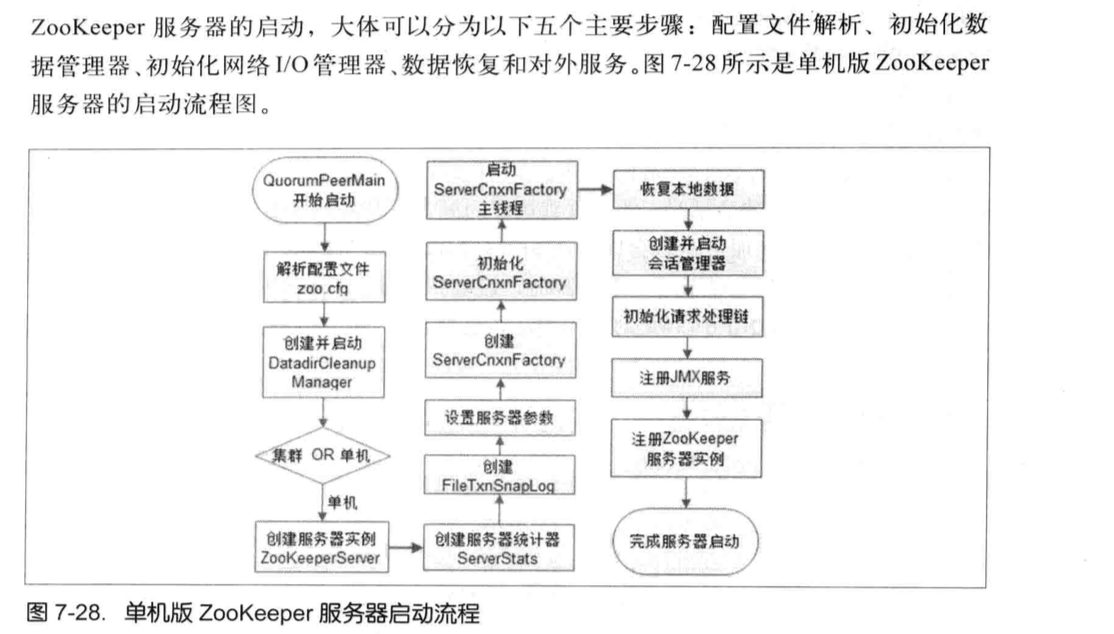
|  |
| --- |
| # The number of milliseconds of each tick tickTime=2000 # The number of ticks that the initial # synchronization phase can take initLimit=10 # The number of ticks that can pass between # sending a request and getting an acknowledgement syncLimit=5 # the directory where the snapshot is stored. # do not use /tmp for storage, /tmp here is just # example sakes. dataDir=/Users/amao/Documents/temp/zookeeper356/zk1 # the port at which the clients will connect  # 这个要注意每个不一样 clientPort=2181 # the maximum number of client connections. # increase this if you need to handle more clients #maxClientCnxns=60 # # Be sure to read the maintenance section of the # administrator guide before turning on autopurge. # # http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance # # The number of snapshots to retain in dataDir #autopurge.snapRetainCount=3 # Purge task interval in hours # Set to "0" to disable auto purge feature #autopurge.purgeInterval=1  # 这个要注意每个不一样 admin.serverPort=9091  server.1=127.0.0.1:2287:3387 server.2=127.0.0.1:2288:3388 server.3=127.0.0.1:2289:3389  zoo2.cfg  # The number of milliseconds of each tick tickTime=2000 # The number of ticks that the initial # synchronization phase can take initLimit=10 # The number of ticks that can pass between # sending a request and getting an acknowledgement syncLimit=5 # the directory where the snapshot is stored. # do not use /tmp for storage, /tmp here is just # example sakes. dataDir=/Users/amao/Documents/temp/zookeeper356/zk2 # the port at which the clients will connect clientPort=2182 # the maximum number of client connections. # increase this if you need to handle more clients #maxClientCnxns=60 # # Be sure to read the maintenance section of the # administrator guide before turning on autopurge. # # http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance # # The number of snapshots to retain in dataDir #autopurge.snapRetainCount=3 # Purge task interval in hours # Set to "0" to disable auto purge feature #autopurge.purgeInterval=1  admin.serverPort=9092  server.1=127.0.0.1:2287:3387 server.2=127.0.0.1:2288:3388 server.3=127.0.0.1:2289:3389  zoo3.cfg  # The number of milliseconds of each tick tickTime=2000 # The number of ticks that the initial # synchronization phase can take initLimit=10 # The number of ticks that can pass between # sending a request and getting an acknowledgement syncLimit=5 # the directory where the snapshot is stored. # do not use /tmp for storage, /tmp here is just # example sakes. dataDir=/Users/amao/Documents/temp/zookeeper356/zk3 # the port at which the clients will connect clientPort=2183 # the maximum number of client connections. # increase this if you need to handle more clients #maxClientCnxns=60 # # Be sure to read the maintenance section of the # administrator guide before turning on autopurge. # # http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc\_maintenance # # The number of snapshots to retain in dataDir #autopurge.snapRetainCount=3 # Purge task interval in hours # Set to "0" to disable auto purge feature #autopurge.purgeInterval=1  admin.serverPort=9093  server.1=127.0.0.1:2287:3387 server.2=127.0.0.1:2288:3388 server.3=127.0.0.1:2289:3389 |

分别org.apache.zookeeper.server.quorum.QuorumPeerMain复制三份，叫QuorumPeerMain1，QuorumPeerMain2，QuorumPeerMain3，这个三就是集群的启动类

启动类配置如下，配置启动的log4j和zoo.cfg，分别对应上述三个，正常启动第一个和第二个都会报错，知道第三个启动后，就都启动正常，我这边启动第二个的时候，QuorumPeerMain2就有LOOKING状态变为了LEADING，说明QuorumPeerMain2是leader，启动第三个的时候LOOKING状态变为了FOLLOWING，说明第三个是follower

# 主要类介绍

## 3.1 QuorumPeerMain（单机和集群的启动类）



单机启动完成的工作

1. 配置文件解析
2. 初始化数据管理器
3. 初始化网络IO管理器
4. 数据恢复和对外服务

### 3.1.1 main方法

|  |
| --- |
| public static void main(String[] args) {  QuorumPeerMain main = new QuorumPeerMain();  try {  // 初始化和运行都是这个方法  main.initializeAndRun(args);  } catch (IllegalArgumentException e) {  *LOG*.error("Invalid arguments, exiting abnormally", e);  *LOG*.info(*USAGE*);  System.*err*.println(*USAGE*);  System.*exit*(2);  } catch (ConfigException e) {  *LOG*.error("Invalid config, exiting abnormally", e);  System.*err*.println("Invalid config, exiting abnormally");  System.*exit*(2);  } catch (DatadirException e) {  *LOG*.error("Unable to access datadir, exiting abnormally", e);  System.*err*.println("Unable to access datadir, exiting abnormally");  System.*exit*(3);  } catch (AdminServerException e) {  *LOG*.error("Unable to start AdminServer, exiting abnormally", e);  System.*err*.println("Unable to start AdminServer, exiting abnormally");  System.*exit*(4);  } catch (Exception e) {  *LOG*.error("Unexpected exception, exiting abnormally", e);  System.*exit*(1);  }  *LOG*.info("Exiting normally");  System.*exit*(0); } |

### 3.1.2 initializeAndRun方法

1. 完成zoo.cfg的配置解析工作，主要是QuorumPeerConfig完成
2. 创建一个DatadirClearupManager管理器，主要启动一个定时器，执行定时清理数据快照等文件的任务。这个
3. 判断是集群启动还是单集启动，集群走runFromConfig，单机直接启动ZooKeeperServerMain

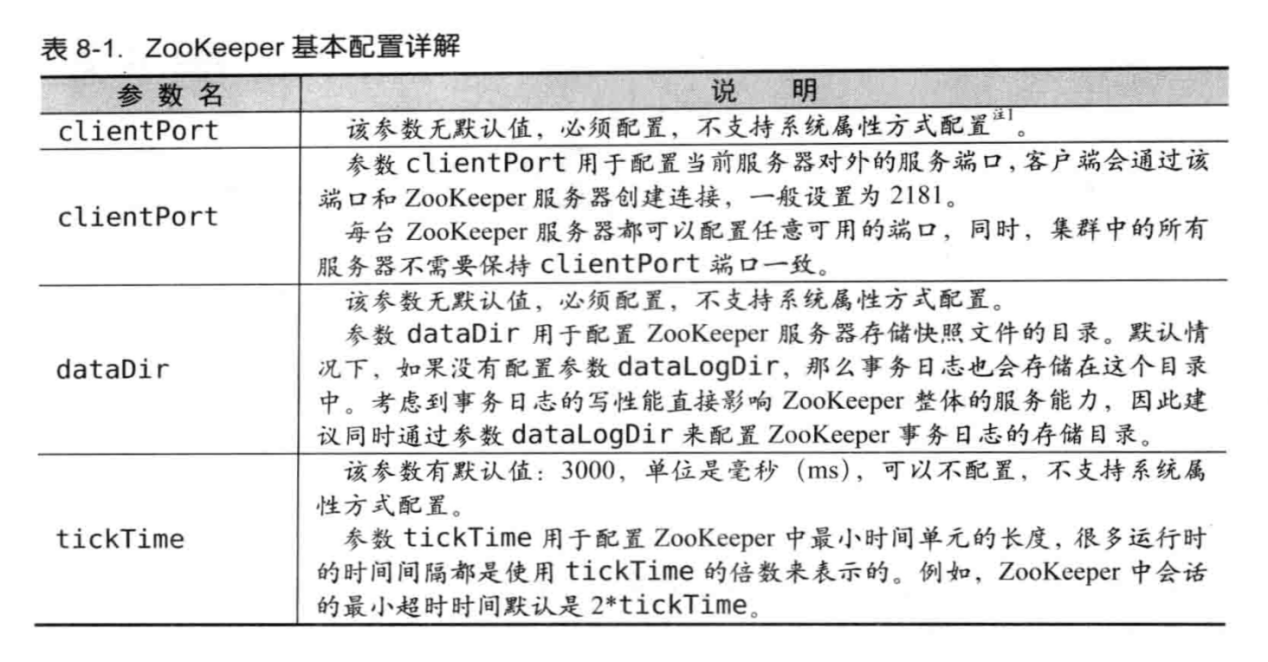
分析单机启动先

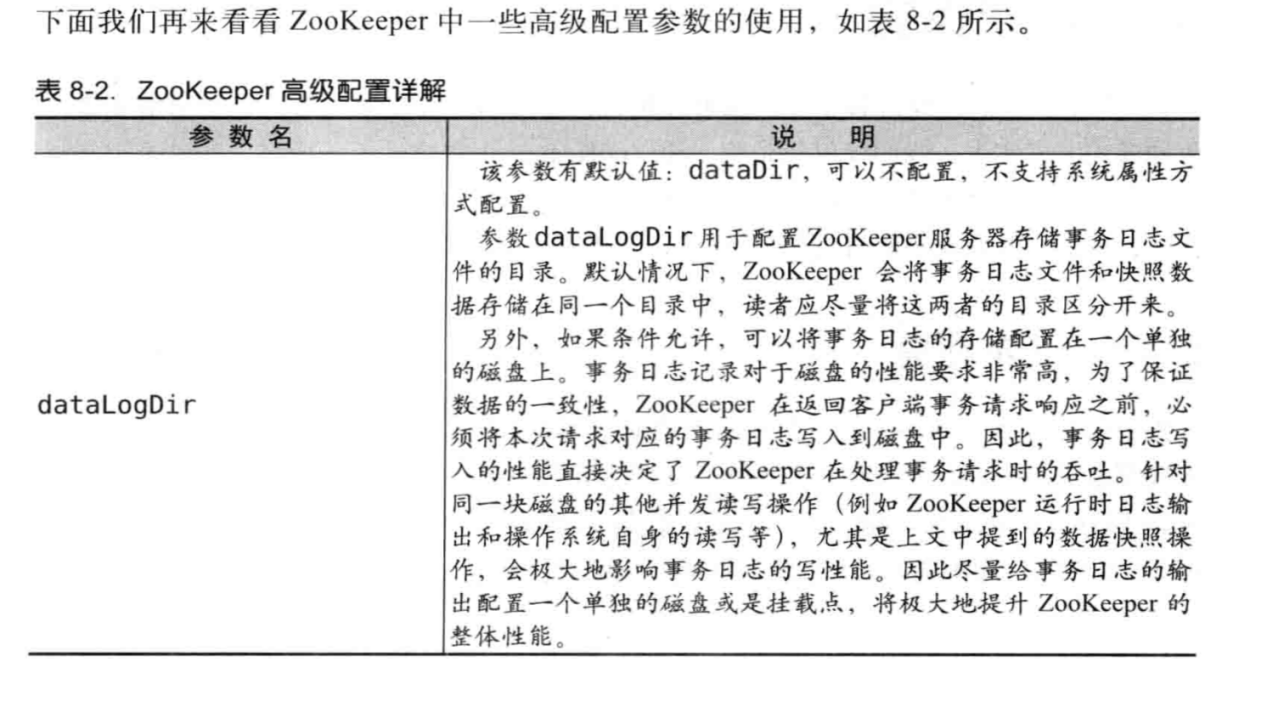
|  |
| --- |
| protected void initializeAndRun(String[] args)  throws ConfigException, IOException, AdminServerException {  // 解析配置文件  QuorumPeerConfig config = new QuorumPeerConfig();  if (args.length == 1) {  config.parse(args[0]);  }   // Start and schedule the the purge task  // 启动一个定时清理快照数据的任务  DatadirCleanupManager purgeMgr = new DatadirCleanupManager(config  .getDataDir(), config.getDataLogDir(), config  .getSnapRetainCount(), config.getPurgeInterval());  purgeMgr.start();   if (args.length == 1 && config.isDistributed()) {  // 集群的启动  runFromConfig(config);  } else {  *LOG*.warn("Either no config or no quorum defined in config, running "  + " in standalone mode");  // there is only server in the quorum -- run as standalone  // 单机启动  ZooKeeperServerMain.*main*(args);  } } |

## 3.2 QuorumPeerConfig（配置解析类）

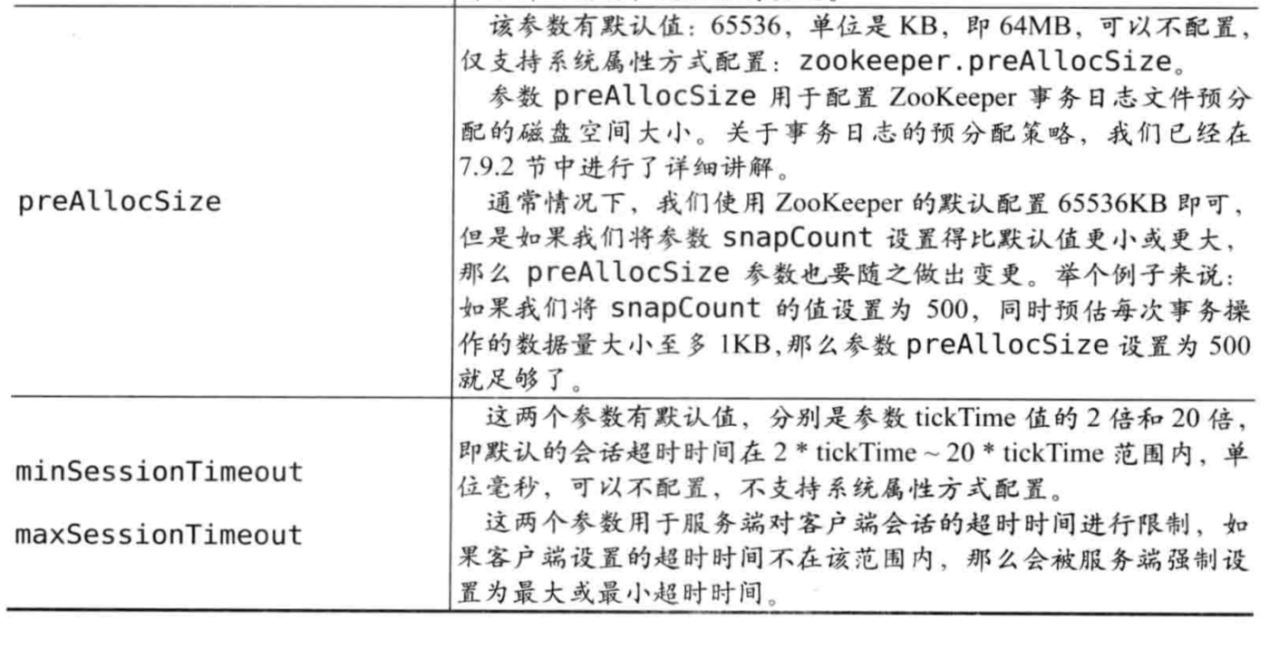
这个类组要为了完成解析zoo.cfg文件的

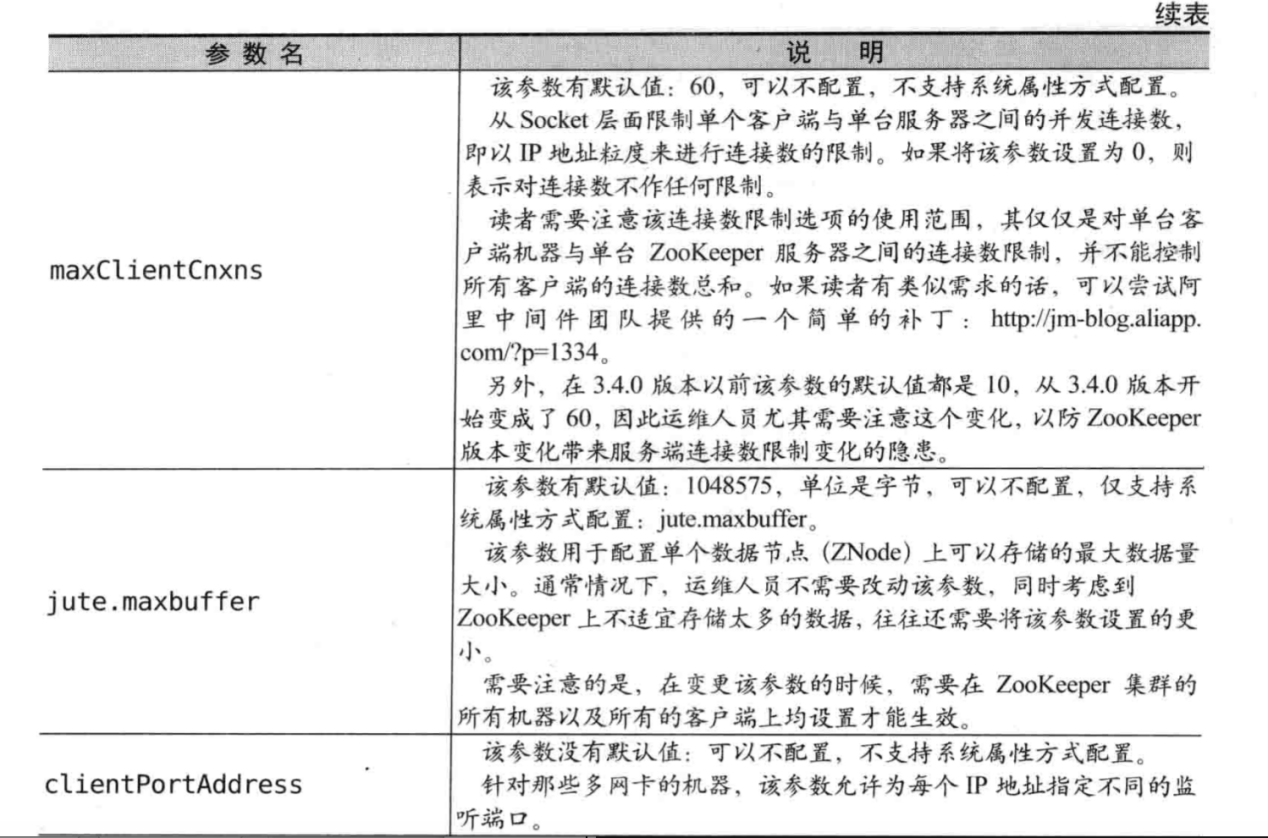
### 3.2.1 主要属性

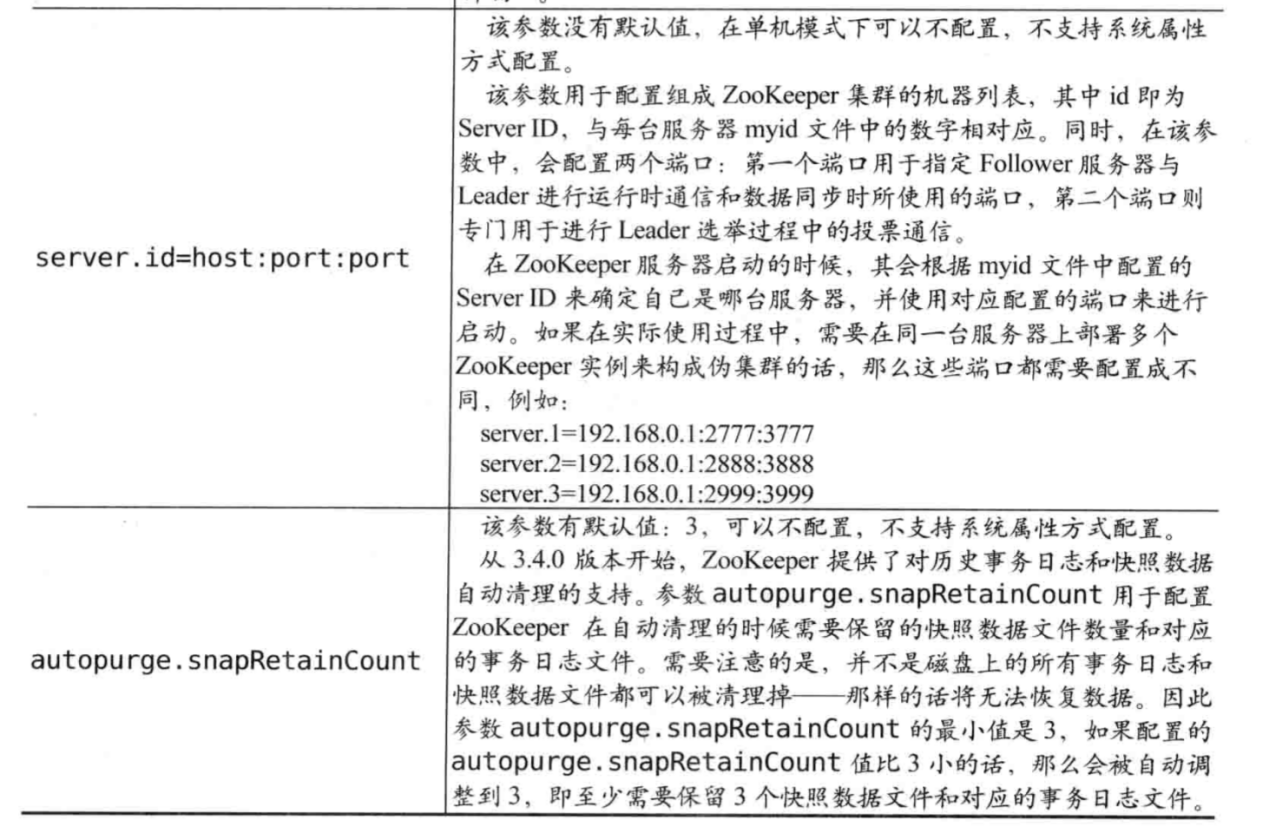






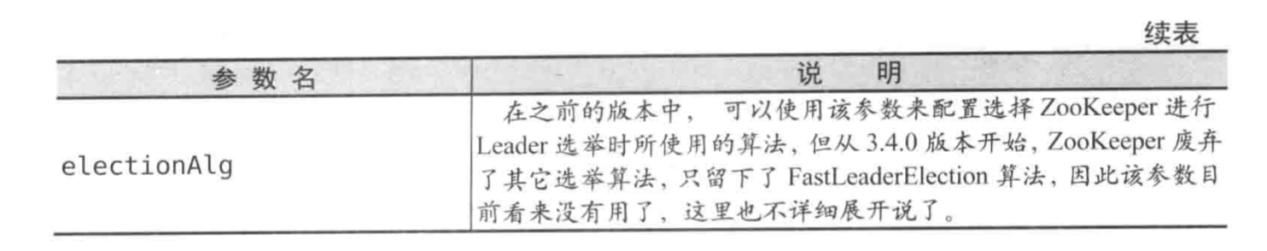












|  |
| --- |
| private static final int *UNSET\_SERVERID* = -1; public static final String *nextDynamicConfigFileSuffix* = ".dynamic.next";  private static boolean *standaloneEnabled* = true; private static boolean *reconfigEnabled* = false;  protected InetSocketAddress clientPortAddress; protected InetSocketAddress secureClientPortAddress; protected boolean sslQuorum = false; protected boolean shouldUsePortUnification = false; protected boolean sslQuorumReloadCertFiles = false;  // 必须配置，存储快照文件的目录。如果没有配置dataLogDir的话，事务日志夜壶存  // 储在这个目录中 protected File dataDir;  // 事务日志存储的目录 protected File dataLogDir;  protected String dynamicConfigFileStr = null; protected String configFileStr = null;  // 3000,单位是毫秒(ms)，可以不配置，不支持系统属性 protected int tickTime = ZooKeeperServer.*DEFAULT\_TICK\_TIME*; // 从Socket层面  protected int maxClientCnxns = 60; */\*\* defaults to -1 if not set explicitly \*/* protected int minSessionTimeout = -1; */\*\* defaults to -1 if not set explicitly \*/* protected int maxSessionTimeout = -1; protected boolean localSessionsEnabled = false; protected boolean localSessionsUpgradingEnabled = false; // 必须配置，Leader服务器等待Follower启动，并完成数据同步的时间。 protected int initLimit;  // 必须配置，Leader服务和Follower之间进行心跳检测的最大延时时间。 protected int syncLimit;  // 进行Leader选举时锁使用的算法。 protected int electionAlg = 3;  //  protected int electionPort = 2182; protected boolean quorumListenOnAllIPs = false;  protected long serverId = *UNSET\_SERVERID*;  protected QuorumVerifier quorumVerifier = null, lastSeenQuorumVerifier = null; protected int snapRetainCount = 3;  // 和snapRetainCount配套使用，用于配置ZooKeeper进行历史文件自动清理的频// 率,如果配置该值为0或负数，那么就表名不需要开启定时清理功能。 protected int purgeInterval = 0; protected boolean syncEnabled = true;  protected LearnerType peerType = LearnerType.*PARTICIPANT*;  */\*\*  \* Configurations for the quorumpeer-to-quorumpeer sasl authentication  \*/* protected boolean quorumServerRequireSasl = false; protected boolean quorumLearnerRequireSasl = false; protected boolean quorumEnableSasl = false; protected String quorumServicePrincipal = QuorumAuth.*QUORUM\_KERBEROS\_SERVICE\_PRINCIPAL\_DEFAULT\_VALUE*; protected String quorumLearnerLoginContext = QuorumAuth.*QUORUM\_LEARNER\_SASL\_LOGIN\_CONTEXT\_DFAULT\_VALUE*; protected String quorumServerLoginContext = QuorumAuth.*QUORUM\_SERVER\_SASL\_LOGIN\_CONTEXT\_DFAULT\_VALUE*; protected int quorumCnxnThreadsSize; |

### 3.2.2 parse方法

|  |
| --- |
| public void parse(String path) throws ConfigException {  *LOG*.info("Reading configuration from: " + path);    try {  // 获得到配置文件对象  File configFile = (new VerifyingFileFactory.Builder(*LOG*)  .warnForRelativePath()  .failForNonExistingPath()  .build()).create(path);    Properties cfg = new Properties();  FileInputStream in = new FileInputStream(configFile);  try {  // 加载  cfg.load(in);  configFileStr = path;  } finally {  in.close();  }  // 解析属性，这个方法中纯粹是解析赋值操作,可以查看parse  parseProperties(cfg);  } catch (IOException e) {  throw new ConfigException("Error processing " + path, e);  } catch (IllegalArgumentException e) {  throw new ConfigException("Error processing " + path, e);  }  // 动态配置文件解析  if (dynamicConfigFileStr!=null) {  try {   Properties dynamicCfg = new Properties();  FileInputStream inConfig = new FileInputStream(dynamicConfigFileStr);  try {  dynamicCfg.load(inConfig);  if (dynamicCfg.getProperty("version") != null) {  throw new ConfigException("dynamic file shouldn't have version inside");  }   String version = *getVersionFromFilename*(dynamicConfigFileStr);  // If there isn't any version associated with the filename,  // the default version is 0.  if (version != null) {  dynamicCfg.setProperty("version", version);  }  } finally {  inConfig.close();  }  setupQuorumPeerConfig(dynamicCfg, false);   } catch (IOException e) {  throw new ConfigException("Error processing " + dynamicConfigFileStr, e);  } catch (IllegalArgumentException e) {  throw new ConfigException("Error processing " + dynamicConfigFileStr, e);  }   File nextDynamicConfigFile = new File(configFileStr + *nextDynamicConfigFileSuffix*);  if (nextDynamicConfigFile.exists()) {  try {   Properties dynamicConfigNextCfg = new Properties();  FileInputStream inConfigNext = new FileInputStream(nextDynamicConfigFile);   try {  dynamicConfigNextCfg.load(inConfigNext);  } finally {  inConfigNext.close();  }  boolean isHierarchical = false;  for (Entry<Object, Object> entry : dynamicConfigNextCfg.entrySet()) {  String key = entry.getKey().toString().trim();   if (key.startsWith("group") || key.startsWith("weight")) {  isHierarchical = true;  break;  }  }  lastSeenQuorumVerifier = *createQuorumVerifier*(dynamicConfigNextCfg, isHierarchical);  } catch (IOException e) {  *LOG*.warn("NextQuorumVerifier is initiated to null");  }  }  } } |

### 3.2.3 parseProperties方法

|  |
| --- |
| public void parseProperties(Properties zkProp) throws IOException, ConfigException {  int clientPort = 0;  int secureClientPort = 0;  String clientPortAddress = null;  String secureClientPortAddress = null;  VerifyingFileFactory vff = new VerifyingFileFactory.Builder(*LOG*).warnForRelativePath().build();  for (Entry<Object, Object> entry : zkProp.entrySet()) {  String key = entry.getKey().toString().trim();  String value = entry.getValue().toString().trim();  if (key.equals("dataDir")) {  dataDir = vff.create(value);  } else if (key.equals("dataLogDir")) {  dataLogDir = vff.create(value);  } else if (key.equals("clientPort")) {  clientPort = Integer.*parseInt*(value);  } else if (key.equals("localSessionsEnabled")) {  localSessionsEnabled = Boolean.*parseBoolean*(value);  } else if (key.equals("localSessionsUpgradingEnabled")) {  localSessionsUpgradingEnabled = Boolean.*parseBoolean*(value);  } else if (key.equals("clientPortAddress")) {  clientPortAddress = value.trim();  } else if (key.equals("secureClientPort")) {  secureClientPort = Integer.*parseInt*(value);  } else if (key.equals("secureClientPortAddress")){  secureClientPortAddress = value.trim();  } else if (key.equals("tickTime")) {  tickTime = Integer.*parseInt*(value);  } else if (key.equals("maxClientCnxns")) {  maxClientCnxns = Integer.*parseInt*(value);  } else if (key.equals("minSessionTimeout")) {  minSessionTimeout = Integer.*parseInt*(value);  } else if (key.equals("maxSessionTimeout")) {  maxSessionTimeout = Integer.*parseInt*(value);  } else if (key.equals("initLimit")) {  initLimit = Integer.*parseInt*(value);  } else if (key.equals("syncLimit")) {  syncLimit = Integer.*parseInt*(value);  } else if (key.equals("electionAlg")) {  electionAlg = Integer.*parseInt*(value);  } else if (key.equals("quorumListenOnAllIPs")) {  quorumListenOnAllIPs = Boolean.*parseBoolean*(value);  } else if (key.equals("peerType")) {  if (value.toLowerCase().equals("observer")) {  peerType = LearnerType.*OBSERVER*;  } else if (value.toLowerCase().equals("participant")) {  peerType = LearnerType.*PARTICIPANT*;  } else  {  throw new ConfigException("Unrecognised peertype: " + value);  }  } else if (key.equals( "syncEnabled" )) {  syncEnabled = Boolean.*parseBoolean*(value);  } else if (key.equals("dynamicConfigFile")){  dynamicConfigFileStr = value;  } else if (key.equals("autopurge.snapRetainCount")) {  snapRetainCount = Integer.*parseInt*(value);  } else if (key.equals("autopurge.purgeInterval")) {  purgeInterval = Integer.*parseInt*(value);  } else if (key.equals("standaloneEnabled")) {  if (value.toLowerCase().equals("true")) {  *setStandaloneEnabled*(true);  } else if (value.toLowerCase().equals("false")) {  *setStandaloneEnabled*(false);  } else {  throw new ConfigException("Invalid option " + value + " for standalone mode. Choose 'true' or 'false.'");  }  } else if (key.equals("reconfigEnabled")) {  if (value.toLowerCase().equals("true")) {  *setReconfigEnabled*(true);  } else if (value.toLowerCase().equals("false")) {  *setReconfigEnabled*(false);  } else {  throw new ConfigException("Invalid option " + value + " for reconfigEnabled flag. Choose 'true' or 'false.'");  }  } else if (key.equals("sslQuorum")){  sslQuorum = Boolean.*parseBoolean*(value);  } else if (key.equals("portUnification")){  shouldUsePortUnification = Boolean.*parseBoolean*(value);  } else if (key.equals("sslQuorumReloadCertFiles")) {  sslQuorumReloadCertFiles = Boolean.*parseBoolean*(value);  } else if ((key.startsWith("server.") || key.startsWith("group") || key.startsWith("weight")) && zkProp.containsKey("dynamicConfigFile")) {  throw new ConfigException("parameter: " + key + " must be in a separate dynamic config file");  } else if (key.equals(QuorumAuth.*QUORUM\_SASL\_AUTH\_ENABLED*)) {  quorumEnableSasl = Boolean.*parseBoolean*(value);  } else if (key.equals(QuorumAuth.*QUORUM\_SERVER\_SASL\_AUTH\_REQUIRED*)) {  quorumServerRequireSasl = Boolean.*parseBoolean*(value);  } else if (key.equals(QuorumAuth.*QUORUM\_LEARNER\_SASL\_AUTH\_REQUIRED*)) {  quorumLearnerRequireSasl = Boolean.*parseBoolean*(value);  } else if (key.equals(QuorumAuth.*QUORUM\_LEARNER\_SASL\_LOGIN\_CONTEXT*)) {  quorumLearnerLoginContext = value;  } else if (key.equals(QuorumAuth.*QUORUM\_SERVER\_SASL\_LOGIN\_CONTEXT*)) {  quorumServerLoginContext = value;  } else if (key.equals(QuorumAuth.*QUORUM\_KERBEROS\_SERVICE\_PRINCIPAL*)) {  quorumServicePrincipal = value;  } else if (key.equals("quorum.cnxn.threads.size")) {  quorumCnxnThreadsSize = Integer.*parseInt*(value);  } else {  System.*setProperty*("zookeeper." + key, value);  }  }   if (!quorumEnableSasl && quorumServerRequireSasl) {  throw new IllegalArgumentException(  QuorumAuth.*QUORUM\_SASL\_AUTH\_ENABLED* + " is disabled, so cannot enable "  + QuorumAuth.*QUORUM\_SERVER\_SASL\_AUTH\_REQUIRED*);  }  if (!quorumEnableSasl && quorumLearnerRequireSasl) {  throw new IllegalArgumentException(  QuorumAuth.*QUORUM\_SASL\_AUTH\_ENABLED* + " is disabled, so cannot enable "  + QuorumAuth.*QUORUM\_LEARNER\_SASL\_AUTH\_REQUIRED*);  }  // If quorumpeer learner is not auth enabled then self won't be able to  // join quorum. So this condition is ensuring that the quorumpeer learner  // is also auth enabled while enabling quorum server require sasl.  if (!quorumLearnerRequireSasl && quorumServerRequireSasl) {  throw new IllegalArgumentException(  QuorumAuth.*QUORUM\_LEARNER\_SASL\_AUTH\_REQUIRED* + " is disabled, so cannot enable "  + QuorumAuth.*QUORUM\_SERVER\_SASL\_AUTH\_REQUIRED*);  }   // Reset to MIN\_SNAP\_RETAIN\_COUNT if invalid (less than 3)  // PurgeTxnLog.purge(File, File, int) will not allow to purge less  // than 3.  if (snapRetainCount < MIN\_SNAP\_RETAIN\_COUNT) {  *LOG*.warn("Invalid autopurge.snapRetainCount: " + snapRetainCount  + ". Defaulting to " + MIN\_SNAP\_RETAIN\_COUNT);  snapRetainCount = MIN\_SNAP\_RETAIN\_COUNT;  }   if (dataDir == null) {  throw new IllegalArgumentException("dataDir is not set");  }  if (dataLogDir == null) {  dataLogDir = dataDir;  }   if (clientPort == 0) {  *LOG*.info("clientPort is not set");  if (clientPortAddress != null) {  throw new IllegalArgumentException("clientPortAddress is set but clientPort is not set");  }  } else if (clientPortAddress != null) {  this.clientPortAddress = new InetSocketAddress(  InetAddress.*getByName*(clientPortAddress), clientPort);  *LOG*.info("clientPortAddress is {}", this.clientPortAddress.toString());  } else {  this.clientPortAddress = new InetSocketAddress(clientPort);  *LOG*.info("clientPortAddress is {}", this.clientPortAddress.toString());  }   if (secureClientPort == 0) {  *LOG*.info("secureClientPort is not set");  if (secureClientPortAddress != null) {  throw new IllegalArgumentException("secureClientPortAddress is set but secureClientPort is not set");  }  } else if (secureClientPortAddress != null) {  this.secureClientPortAddress = new InetSocketAddress(  InetAddress.*getByName*(secureClientPortAddress), secureClientPort);  *LOG*.info("secureClientPortAddress is {}", this.secureClientPortAddress.toString());  } else {  this.secureClientPortAddress = new InetSocketAddress(secureClientPort);  *LOG*.info("secureClientPortAddress is {}", this.secureClientPortAddress.toString());  }  if (this.secureClientPortAddress != null) {  configureSSLAuth();  }   if (tickTime == 0) {  throw new IllegalArgumentException("tickTime is not set");  }   minSessionTimeout = minSessionTimeout == -1 ? tickTime \* 2 : minSessionTimeout;  maxSessionTimeout = maxSessionTimeout == -1 ? tickTime \* 20 : maxSessionTimeout;   if (minSessionTimeout > maxSessionTimeout) {  throw new IllegalArgumentException(  "minSessionTimeout must not be larger than maxSessionTimeout");  }    // backward compatibility - dynamic configuration in the same file as  // static configuration params see writeDynamicConfig()  if (dynamicConfigFileStr == null) {  setupQuorumPeerConfig(zkProp, true);  if (isDistributed() && *isReconfigEnabled*()) {  // we don't backup static config for standalone mode.  // we also don't backup if reconfig feature is disabled.  backupOldConfig();  }  } } |

## 3.3 DatadirCleanupManager（快照数据清理任务管理器）

### 3.3.1 主要属性

|  |
| --- |
| // 清理状态，未启动，启动，完成  public enum PurgeTaskStatus {  *NOT\_STARTED*, *STARTED*, *COMPLETED*; }  // 默认状态是为启动 private PurgeTaskStatus purgeTaskStatus = PurgeTaskStatus.*NOT\_STARTED*;  // 快照数据目录 private final File snapDir;  // 事务日志目录 private final File dataLogDir;  // 清理之后保留的快照文件的数量  // 即快照数据保持一定的数量，超过这个数量，就会被清理掉 private final int snapRetainCount;  // 定时清理的时间 private final int purgeInterval;  // 定时器 private Timer timer; |

### 3.3.2 start方法

启动定时清理的任务

|  |
| --- |
| public void start() {  // 判断状态是否已经启动  if (PurgeTaskStatus.*STARTED* == purgeTaskStatus) {  *LOG*.warn("Purge task is already running.");  return;  }  // Don't schedule the purge task with zero or negative purge interval.  // 定时清理的时间要大于0  if (purgeInterval <= 0) {  *LOG*.info("Purge task is not scheduled.");  return;  }  // 创建定时器  timer = new Timer("PurgeTask", true);  // 创建任务  // PurgeTask是TimeTask的子类，实际是个Runnable  // 最终执行PurgeTask的run方法  TimerTask task = new PurgeTask(dataLogDir, snapDir, snapRetainCount);  // 提交任务，执行定时清理的  timer.scheduleAtFixedRate(task, 0, TimeUnit.*HOURS*.toMillis(purgeInterval));   purgeTaskStatus = PurgeTaskStatus.*STARTED*; } |

## 3.4 ZooKeeperServerMain（单机启动）

其实，单机启动这个类就可以启动

### 3.4.1 主要属性

|  |
| --- |
| // 服务端连接工厂，有两个实现，基于NIO和基于netty的实现  // 可通过zookeeper.serverCnxnFactory来设置类名  // 没有设置默认是使用NIOServerCnxnFactory  private ServerCnxnFactory cnxnFactory;  // 安全连接工厂 private ServerCnxnFactory secureCnxnFactory;  // 数据节点的清理容器：仅运行在leader服务器，followers/observers运行产生// 没必要的额外开销,也就是followers/observers可以运行，只是没必要。 private ContainerManager containerManager; // 猜测是命令行操作的连接服务器：有两个实现，一个是个“空”server，另一个是个  // jetty实现的服务器 private AdminServer adminServer; |

### 3.4.2 main方法

调用初始化和运行的方法

|  |
| --- |
| public static void main(String[] args) {  ZooKeeperServerMain main = new ZooKeeperServerMain();  try {  main.initializeAndRun(args);  } catch (IllegalArgumentException e) {  *LOG*.error("Invalid arguments, exiting abnormally", e);  *LOG*.info(*USAGE*);  System.*err*.println(*USAGE*);  System.*exit*(2);  } catch (ConfigException e) {  *LOG*.error("Invalid config, exiting abnormally", e);  System.*err*.println("Invalid config, exiting abnormally");  System.*exit*(2);  } catch (DatadirException e) {  *LOG*.error("Unable to access datadir, exiting abnormally", e);  System.*err*.println("Unable to access datadir, exiting abnormally");  System.*exit*(3);  } catch (AdminServerException e) {  *LOG*.error("Unable to start AdminServer, exiting abnormally", e);  System.*err*.println("Unable to start AdminServer, exiting abnormally");  System.*exit*(4);  } catch (Exception e) {  *LOG*.error("Unexpected exception, exiting abnormally", e);  System.*exit*(1);  }  *LOG*.info("Exiting normally");  System.*exit*(0); } |

### 3.4.3 initializeAndRun

|  |
| --- |
| protected void initializeAndRun(String[] args)  throws ConfigException, IOException, AdminServerException {  try {  // 注册log4j的bean，先不用管，猜测是设置日志输出用的，和日志配置有关  ManagedUtil.*registerLog4jMBeans*();  } catch (JMException e) {  *LOG*.warn("Unable to register log4j JMX control", e);  }  // 配置文件解析类，这个和QuorumPeerConfig类似，不过比它少了很多属性  // 只有几个重要的属性，也有解析的方法  ServerConfig config = new ServerConfig();   if (args.length == 1) {  config.parse(args[0]);  } else {  config.parse(args);  }  // 根据配置文件开始启动相关组件  runFromConfig(config); } |

### 3.4.4 runFromConfig

整个方法均是初始化相关组件和启动组件

|  |
| --- |
| public void runFromConfig(ServerConfig config)  throws IOException, AdminServerException {  *LOG*.info("Starting server");  FileTxnSnapLog txnLog = null;  try {  // Note that this thread isn't going to be doing anything else,  // so rather than spawning another thread, we will just call  // run() in this thread.  // create a file logger url from the command line args  // 创建事务日志和快照的管理的类  txnLog = new FileTxnSnapLog(config.dataLogDir, config.dataDir);  // 创建zookeeperServer  final ZooKeeperServer zkServer = new ZooKeeperServer(txnLog,  config.tickTime, config.minSessionTimeout, config.maxSessionTimeout, null);  // 把初始化的zkServer状态记录到日志和快照中  txnLog.setServerStats(zkServer.serverStats());   // Registers shutdown handler which will be used to know the  // server error or shutdown state changes.  // 打开一个闭锁，用于阻塞当前线程，保持一致运行状态，  final CountDownLatch shutdownLatch = new CountDownLatch(1);  zkServer.registerServerShutdownHandler(  new ZooKeeperServerShutdownHandler(shutdownLatch));   // Start Admin server  // 创建一个管理服务器，应该就是用于zkCli连接和接收命令用的  adminServer = AdminServerFactory.*createAdminServer*();  adminServer.setZooKeeperServer(zkServer);  adminServer.start();   boolean needStartZKServer = true;  if (config.getClientPortAddress() != null) {  // 初始化连接工厂  cnxnFactory = ServerCnxnFactory.*createFactory*();  cnxnFactory.configure(config.getClientPortAddress(), config.getMaxClientCnxns(), false);  // 这里启动其实就是zkServer的启动  // 也就是启动操作是由cnxnFactory来完成的。  cnxnFactory.startup(zkServer);  // zkServer has been started. So we don't need to start it again in secureCnxnFactory.  needStartZKServer = false;  }  if (config.getSecureClientPortAddress() != null) {  secureCnxnFactory = ServerCnxnFactory.*createFactory*();  secureCnxnFactory.configure(config.getSecureClientPortAddress(), config.getMaxClientCnxns(), true);  secureCnxnFactory.startup(zkServer, needStartZKServer);  }   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();   // Watch status of ZooKeeper server. It will do a graceful shutdown  // if the server is not running or hits an internal error.  shutdownLatch.await();   shutdown();   if (cnxnFactory != null) {  cnxnFactory.join();  }  if (secureCnxnFactory != null) {  secureCnxnFactory.join();  }  if (zkServer.canShutdown()) {  zkServer.shutdown(true);  }  } catch (InterruptedException e) {  // warn, but generally this is ok  *LOG*.warn("Server interrupted", e);  } finally {  if (txnLog != null) {  txnLog.close();  }  } } |

### 3.4.5 shutdown

|  |
| --- |
| protected void shutdown() {  if (containerManager != null) {  containerManager.stop();  }  if (cnxnFactory != null) {  cnxnFactory.shutdown();  }  if (secureCnxnFactory != null) {  secureCnxnFactory.shutdown();  }  try {  if (adminServer != null) {  adminServer.shutdown();  }  } catch (AdminServerException e) {  *LOG*.warn("Problem stopping AdminServer", e);  } } |

## 3.5 ZooKeeperServer （zk服务器）

### 3.5.1 属性

|  |
| --- |
| //  protected ZooKeeperServerBean jmxServerBean;  //  protected DataTreeBean jmxDataTreeBean;  public static final int *DEFAULT\_TICK\_TIME* = 3000; protected int tickTime = *DEFAULT\_TICK\_TIME*; */\*\* value of -1 indicates unset, use default \*/* protected int minSessionTimeout = -1; */\*\* value of -1 indicates unset, use default \*/* protected int maxSessionTimeout = -1;  // 会话管理 protected SessionTracker sessionTracker;  // 事务日志和快照管理类 private FileTxnSnapLog txnLogFactory = null;  // zk数据节点的内存数据库 private ZKDatabase zkDb;  //  private final AtomicLong hzxid = new AtomicLong(0); public final static Exception *ok* = new Exception("No prob");  // 请求处理器 protected RequestProcessor firstProcessor;  // 服务器状态 protected volatile State state = State.*INITIAL*;  protected enum State {  *INITIAL*, *RUNNING*, *SHUTDOWN*, *ERROR* }  */\*\*  \* This is the secret that we use to generate passwords. For the moment,  \* it's more of a checksum that's used in reconnection, which carries no  \* security weight, and is treated internally as if it carries no  \* security weight.  \*/*  static final private long *superSecret* = 0XB3415C00L;  private final AtomicInteger requestsInProcess = new AtomicInteger(0); //  final Deque<ChangeRecord> outstandingChanges = new ArrayDeque<>(); // this data structure must be accessed under the outstandingChanges lock final HashMap<String, ChangeRecord> outstandingChangesForPath =  new HashMap<String, ChangeRecord>();  //  protected ServerCnxnFactory serverCnxnFactory; protected ServerCnxnFactory secureServerCnxnFactory;  private final ServerStats serverStats;  // 监听器：主要监听服务器线程异常捕捉后，通知服务器，进行状态调整，已经和服务器// 关闭操作等 private final ZooKeeperServerListener listener;  //  private ZooKeeperServerShutdownHandler zkShutdownHandler; private volatile int createSessionTrackerServerId = 1; |

### 3.5.2 startdata

// 从事务日志中获取ZKData

|  |
| --- |
| public void startdata() throws IOException, InterruptedException {  //check to see if zkDb is not null  if (zkDb == null) {  zkDb = new ZKDatabase(this.txnLogFactory);  }  // 没有初始化，进行加载数据  if (!zkDb.isInitialized()) {  loadData();  } } |

### 3.5.3 loadData

|  |
| --- |
| public void loadData() throws IOException, InterruptedException {  /\*  \* When a new leader starts executing Leader#lead, it   \* invokes this method. The database, however, has been  \* initialized before running leader election so that  \* the server could pick its zxid for its initial vote.  \* It does it by invoking QuorumPeer#getLastLoggedZxid.  \* Consequently, we don't need to initialize it once more  \* and avoid the penalty of loading it a second time. Not   \* reloading it is particularly important for applications  \* that host a large database.  \*   \* The following if block checks whether the database has  \* been initialized or not. Note that this method is  \* invoked by at least one other method:   \* ZooKeeperServer#startdata.  \*   \* See ZOOKEEPER-1642 for more detail.  \*/  if(zkDb.isInitialized()){  // 设置事务id  setZxid(zkDb.getDataTreeLastProcessedZxid());  }  else {  setZxid(zkDb.loadDataBase());  }    // Clean up dead sessions  // 清除死掉的会话  LinkedList<Long> deadSessions = new LinkedList<Long>();  for (Long session : zkDb.getSessions()) {  if (zkDb.getSessionWithTimeOuts().get(session) == null) {  deadSessions.add(session);  }  }   for (long session : deadSessions) {  // XXX: Is lastProcessedZxid really the best thing to use?  killSession(session, zkDb.getDataTreeLastProcessedZxid());  }   // Make a clean snapshot  // 生成一个快照文件，在配置dataDir或者dataLogDir下面  takeSnapshot(); } |

### 3.5.4 startup 启动

启动session和设置

|  |
| --- |
| public synchronized void startup() {  if (sessionTracker == null) {  // 创建session管理器  createSessionTracker();  }  // 启动session  startSessionTracker();  // 设置请求处理链  setupRequestProcessors();  // 注册JMX  registerJMX();  // 修改状态为RUNNING  setState(State.*RUNNING*);  // 通知  notifyAll(); } |

### 3.5.5 createSessionTracker

|  |
| --- |
| protected void createSessionTracker() {  // 直接创建实例，  sessionTracker = new SessionTrackerImpl(this, zkDb.getSessionWithTimeOuts(),  tickTime, createSessionTrackerServerId, getZooKeeperServerListener()); } |

### 3.5.6 startSessionTracker

|  |
| --- |
| protected void startSessionTracker() {  // 直接启动sessionTracker线程  ((SessionTrackerImpl)sessionTracker).start(); } |

### 3.5.7 expire

// 在SessionTrackerImpl.run方法中被调用

// 将过期的会话进行关闭操作

|  |
| --- |
| public void expire(Session session) {  long sessionId = session.getSessionId();  *LOG*.info("Expiring session 0x" + Long.*toHexString*(sessionId)  + ", timeout of " + session.getTimeout() + "ms exceeded");  close(sessionId); }  private void close(long sessionId) {  // 构造一个请求  Request si = new Request(null, sessionId, 0, OpCode.*closeSession*, null, null);  // 空方法  setLocalSessionFlag(si);  submitRequest(si); }  protected void setLocalSessionFlag(Request si) { } |

### 3.5.8 submitRequest(提交请求)

|  |
| --- |
| public void submitRequest(Request si) {  // 请求处理链的第一个请求处理器firstProcessor  if (firstProcessor == null) {  synchronized (this) {  try {  // Since all requests are passed to the request  // processor it should wait for setting up the request  // processor chain. The state will be updated to RUNNING  // after the setup.  while (state == State.*INITIAL*) {  // 如果zookeeperServer是初始换状态，就一直等待  wait(1000);  }  } catch (InterruptedException e) {  *LOG*.warn("Unexpected interruption", e);  }  // 处理链不能空，状态必须是运行状态  if (firstProcessor == null || state != State.*RUNNING*) {  throw new RuntimeException("Not started");  }  }  }  try {  touch(si.cnxn);  // 判断是否是有效的数据包，这里除了通知  // OpCode.notification的类型，其余都是true  boolean validpacket = Request.*isValid*(si.type);  if (validpacket) {  // 开始经过处理链对请求进行处理  // 这个过程比较复杂  firstProcessor.processRequest(si);  if (si.cnxn != null) {  incInProcess();  }  } else {  *LOG*.warn("Received packet at server of unknown type " + si.type);  new UnimplementedRequestProcessor().processRequest(si);  }  } catch (MissingSessionException e) {  if (*LOG*.isDebugEnabled()) {  *LOG*.debug("Dropping request: " + e.getMessage());  }  } catch (RequestProcessorException e) {  *LOG*.error("Unable to process request:" + e.getMessage(), e);  } }  void touch(ServerCnxn cnxn) throws MissingSessionException {  // 关闭操作，cnxn==null  if (cnxn == null) {  return;  }  long id = cnxn.getSessionId();  int to = cnxn.getSessionTimeout();  if (!sessionTracker.touchSession(id, to)) {  throw new MissingSessionException(  "No session with sessionid 0x" + Long.*toHexString*(id)  + " exists, probably expired and removed");  } } |

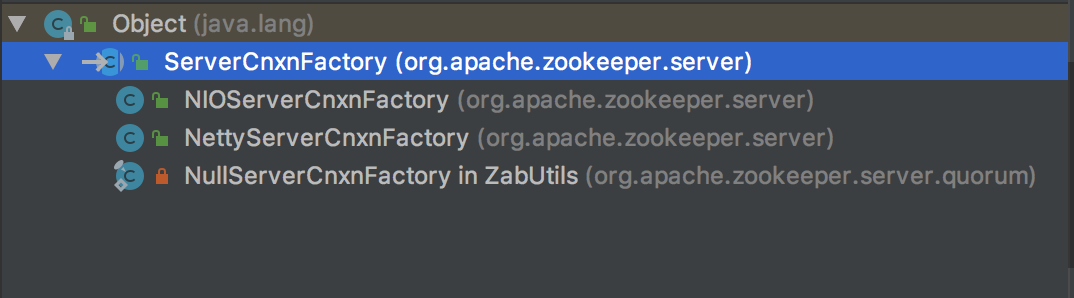
### 3.5.9 setupRequestProcessors

启动请求处理链线程

|  |
| --- |
| protected void setupRequestProcessors() {  // 处理器链上的最后一个处理器  RequestProcessor finalProcessor = new FinalRequestProcessor(this);  // 异步处理器  RequestProcessor syncProcessor = new SyncRequestProcessor(this,  finalProcessor);  ((SyncRequestProcessor)syncProcessor).start();  // 链上的第一个处理器  firstProcessor = new PrepRequestProcessor(this, syncProcessor);  // 以上三个pre->sync->final构成了一个长度为3的链  // 即是pre处理完后，sync处理，最后final处理  // 启动处理器链线程  ((PrepRequestProcessor)firstProcessor).start(); } |

## 3.6 ServerCnxnFactory

三个子类：



### 3.6.1 startup

有两个启动方法，一个两参的抽象方法，最终都是调用两参的抽象方法

|  |
| --- |
| //  public void startup(ZooKeeperServer zkServer) throws IOException, InterruptedException {  // 调用两参的方法，  startup(zkServer, true); }  // This method is to maintain compatiblity of startup(zks) and enable sharing of zks // when we add secureCnxnFactory. public abstract void startup(ZooKeeperServer zkServer, boolean startServer)  throws IOException, InterruptedException; |

## 3.7 NIOServerCnxnFactory（NIO）

### 3.7.1 startup(启动类)

|  |
| --- |
| // 启动类  public void startup(ZooKeeperServer zks, boolean startServer)  throws IOException, InterruptedException {  start();  setZooKeeperServer(zks);  if (startServer) {  zks.startdata();  zks.startup();  } } |

### 3.7.2 start

启动线程池、接收线程、连接过去监控线程

|  |
| --- |
| public void start() {  stopped = false;  if (workerPool == null) {  // 初始化线程池  // numWorkerThreads工作线程的数量，可配置，没有配置，  // 是cpu核数的两倍  workerPool = new WorkerService(  "NIOWorker", numWorkerThreads, false);  }  // 循环启动selector线程  for(SelectorThread thread : selectorThreads) {  if (thread.getState() == Thread.State.*NEW*) {  thread.start();  }  }  // ensure thread is started once and only once  // 确保线程仅且启动一次  // 接收线程启动  if (acceptThread.getState() == Thread.State.*NEW*) {  acceptThread.start();  }  // 连接过期的监控线程启动  if (expirerThread.getState() == Thread.State.*NEW*) {  expirerThread.start();  } } |

## 3.8 NettyServerCnxnFactory

## 3.9 ZooKeeperThread (zk线程)

继承至Thread，

主要方法就是个处理异常的方法。

|  |
| --- |
| protected void handleException(String thName, Throwable e) {  *LOG*.warn("Exception occurred from thread {}", thName, e); } |

## 3.10 WorkerService(zk线程池)

## 3.11 SessionTrackerImpl(会话管理器，是个线程)

### 3.11.1 类结构

ZooKeeperCriticalThread是ZooKeeperThread的子类，说明这个会话是个线程

|  |
| --- |
| public class SessionTrackerImpl extends ZooKeeperCriticalThread implements  SessionTracker {  } |

### 3.11.2 内部类

SessionTracker中的会话类

|  |
| --- |
| public static interface Session {  // 会话id  long getSessionId();  // 超时时间  int getTimeout();  // 会话是否关闭  boolean isClosing(); } |

子类中的Session实现

|  |
| --- |
| public static class SessionImpl implements Session {  SessionImpl(long sessionId, int timeout) {  this.sessionId = sessionId;  this.timeout = timeout;  isClosing = false;  }   final long sessionId;  final int timeout;  boolean isClosing;   Object owner;   public long getSessionId() { return sessionId; }  public int getTimeout() { return timeout; }  public boolean isClosing() { return isClosing; }   public String toString() {  return "0x" + Long.*toHexString*(sessionId);  } } |

### 3.11.3 主要属性

|  |
| --- |
| // 维护sessionId和session实体映射关系的  protected final ConcurrentHashMap<Long, SessionImpl> sessionsById =  new ConcurrentHashMap<Long, SessionImpl>();  // 过期的会话对象实例对列 private final ExpiryQueue<SessionImpl> sessionExpiryQueue;  // 维护sessionId和过期时间的集合，这个会被定期持久化到快照文件中 private final ConcurrentMap<Long, Integer> sessionsWithTimeout;  // 会话的ID private final AtomicLong nextSessionId = new AtomicLong(); |

### 3.11.4 SessionTrackerImpl的构造函数

|  |
| --- |
| public SessionTrackerImpl(SessionExpirer expirer,  ConcurrentMap<Long, Integer> sessionsWithTimeout, int tickTime,  long serverId, ZooKeeperServerListener listener) {  super("SessionTracker", listener);  this.expirer = expirer;  // tickTime是下次会话超时的时间  // 根据分桶策略维护的会话过期的对列  this.sessionExpiryQueue = new ExpiryQueue<SessionImpl>(tickTime);  this.sessionsWithTimeout = sessionsWithTimeout;  // 初始话SessionId  this.nextSessionId.set(*initializeNextSession*(serverId));  for (Entry<Long, Integer> e : sessionsWithTimeout.entrySet()) {  addSession(e.getKey(), e.getValue());  }   EphemeralType.*validateServerId*(serverId); }  搞8位是服务器serverId，紧接着55位是时间戳，低两位是0  这个sessionId是全局唯一的  public static long initializeNextSession(long id) {  long nextSid;  nextSid = (Time.*currentElapsedTime*() << 24) >>> 8;  nextSid = nextSid | (id << 56);  if (nextSid == EphemeralType.*CONTAINER\_EPHEMERAL\_OWNER*) {  ++nextSid; // this is an unlikely edge case, but check it just in case  }  return nextSid; }  public synchronized boolean addSession(long id, int sessionTimeout) {  sessionsWithTimeout.put(id, sessionTimeout);   boolean added = false;   SessionImpl session = sessionsById.get(id);  if (session == null) {  实例化session  session = new SessionImpl(id, sessionTimeout);  }   // findbugs2.0.3 complains about get after put.  // long term strategy would be use computeIfAbsent after JDK 1.8  SessionImpl existedSession = sessionsById.putIfAbsent(id, session);   if (existedSession != null) {  session = existedSession;  } else {  added = true;  *LOG*.debug("Adding session 0x" + Long.*toHexString*(id));  }   if (*LOG*.isTraceEnabled()) {  String actionStr = added ? "Adding" : "Existing";  ZooTrace.*logTraceMessage*(*LOG*, ZooTrace.*SESSION\_TRACE\_MASK*,  "SessionTrackerImpl --- " + actionStr + " session 0x"  + Long.*toHexString*(id) + " " + sessionTimeout);  }   updateSessionExpiry(session, sessionTimeout);  return added; }  // 修改每个会话的过期时间  private void updateSessionExpiry(SessionImpl s, int timeout) {  logTraceTouchSession(s.sessionId, timeout, "");  sessionExpiryQueue.update(s, timeout); } |

### 3.11.5 run方法

|  |
| --- |
| public void run() {  try {  while (running) {  // 计算会话等待时间  long waitTime = sessionExpiryQueue.getWaitTime();  if (waitTime > 0) {  Thread.*sleep*(waitTime);  continue;  }  // 如果会话超时，进行对会话的清除工作  for (SessionImpl s : sessionExpiryQueue.poll()) {  setSessionClosing(s.sessionId);  // expirer就是ZooKeeperServer实例  expirer.expire(s);  }  }  } catch (InterruptedException e) {  handleException(this.getName(), e);  }  *LOG*.info("SessionTrackerImpl exited loop!"); }  synchronized public void setSessionClosing(long sessionId) {  if (*LOG*.isTraceEnabled()) {  *LOG*.trace("Session closing: 0x" + Long.*toHexString*(sessionId));  }  SessionImpl s = sessionsById.get(sessionId);  if (s == null) {  return;  }  // 将session的关闭标志设置成true  s.isClosing = true; } |

## 3.12 ExpiryQueue(过期会话对列)

这个主要是根据“分桶策略”进行对 同一时间段内过期的会话进行管理

### 3.12.1 主要属性

|  |
| --- |
| */\*\*  \* The maximum number of buckets is equal to max timeout/expirationInterval,  \* so the expirationInterval should not be too small compared to the  \* max timeout that this expiry queue needs to maintain.  \*/*  *// 时间戳为key，会话实体的set集合为value，* private final ConcurrentHashMap<Long, Set<E>> expiryMap =  new ConcurrentHashMap<Long, Set<E>>();  private final AtomicLong nextExpirationTime = new AtomicLong();  // 这个值就是tickTime，也就是会话的下次连接时间 private final int expirationInterval; |

## 3.13 ServerCnxn(服务端连接管理类)

代表一个从客户端到服务端连接

### 3.13.1 属性

|  |
| --- |
| // 认证信息  protected ArrayList<Id> authInfo = new ArrayList<Id>();  private static final byte[] *fourBytes* = new byte[4];  */\*\*  \* If the client is of old version, we don't send r-o mode info to it.  \* The reason is that if we would, old C client doesn't read it, which  \* results in TCP RST packet, i.e. "connection reset by peer".  \*/*  *// 是否是旧客户端* boolean isOldClient = true;  // 一下主要是描述连接的统计信息  // 连接创建时间  protected final Date established = new Date(); // 接收的数据包数量 protected final AtomicLong packetsReceived = new AtomicLong();  // 发送的数据包数量 protected final AtomicLong packetsSent = new AtomicLong();  // 最小延迟 protected long minLatency;  // 最大延迟 protected long maxLatency;  // 最后的操作 protected String lastOp;  // 最后的cxid protected long lastCxid;  // 最后的zxid protected long lastZxid;  // 最后的响应时间 protected long lastResponseTime;  // 最后的延迟 protected long lastLatency; // 数量 protected long count;  // 总的延迟 protected long totalLatency; |

## 3.14 Watcher

定义了一事件(事件类型和zookeeper状态)和watcherType

## 3.15 RequestProcessor（请求处理器）

该类比较多，zookeeperServer服务启动的时候，请求处理链的第一个类是PreRequestProcessor

### 3.15.1 PreRequestProcessor(第一个处理器)

### 3.15.2 SyncRequestProcessor

### 3.15.3 FinalRequestProcessor(最后一个处理器)

所有的请求或查询服务都会经过这个处理器，这个处理器会对zookeeperServer的outstandingRequests进行计数统计

## 3.16 ZKDatabase

zookeeper服务器状态的内存数据库，包括session、datatree和提交的事务日志。

当重启时，会重日志和快照中读取数据，加载恢复

|  |
| --- |
| */\*\*  \* This class maintains the in memory database of zookeeper  \* server states that includes the sessions, datatree and the  \* committed logs. It is booted up after reading the logs  \* and snapshots from the disk.  \*/* public class ZKDatabase {  } |

### 3.16.1 主要属性

|  |
| --- |
| // 节点数据对象  protected DataTree dataTree;  // 持久化到快照中的session对应的过期时间的映射关系集合 protected ConcurrentHashMap<Long, Integer> sessionsWithTimeouts;  // 快照和事务日志对象 protected FileTxnSnapLog snapLog;  // 最小提交的日志，最大提交的日志 protected long minCommittedLog, maxCommittedLog;  */\*\*  \* Default value is to use snapshot if txnlog size exceeds 1/3 the size of snapshot  \*/*  *//* public static final String *SNAPSHOT\_SIZE\_FACTOR* = "zookeeper.snapshotSizeFactor"; public static final double *DEFAULT\_SNAPSHOT\_SIZE\_FACTOR* = 0.33; private double snapshotSizeFactor;  public static final int *commitLogCount* = 500; protected static int *commitLogBuffer* = 700; protected LinkedList<Proposal> committedLog = new LinkedList<Proposal>();  // 日志文件的读写锁 protected ReentrantReadWriteLock logLock = new ReentrantReadWriteLock(); volatile private boolean initialized = false; |

## 3.20 FileTxnSnapSnapLog(辅助类：事务日志和快照)

主要作用将最新的事务日志和快照文件再重启动的过程中，重新加载，并且反序列化出文件的内容。用于获取session等信息

## 3.21 FileTxnLog(事务日志的文件类)

## 3.22 FileSnap(快照的文件类)

## 3.23 ZooKeeper (客户端连接启动类）

### 3.23.1 主要字段

|  |
| --- |
| // 负责解析服务器地址配置的类  protected final HostProvider hostProvider;  // 管理注册到服务器的Watcher  protected final ZKWatchManager watchManager; // 客户端的配置文件类 private final ZKClientConfig clientConfig; |

### 3.23.2 内部类

|  |
| --- |
| // 管理watchers、处理ClientCnxn(客户端连接类)产生的事件  static class ZKWatchManager implements ClientWatchManager {  // 数据监听器  private final Map<String, Set<Watcher>> dataWatches =  new HashMap<String, Set<Watcher>>();  // 特殊监听器 private final Map<String, Set<Watcher>> existWatches =  new HashMap<String, Set<Watcher>>();  // 子结点监听器 private final Map<String, Set<Watcher>> childWatches =  new HashMap<String, Set<Watcher>>();  }  // Watcher的注册器：将一个Watcher注册到一个特殊的路径下  public abstract class WatchRegistration {  }  // 处理已存在的特殊的watches  // 这种情形一般是无节点结果码值返回  class ExistsWatchRegistration extends WatchRegistration {  public ExistsWatchRegistration(Watcher watcher, String clientPath) {  super(watcher, clientPath);  }   @Override  protected Map<String, Set<Watcher>> getWatches(int rc) {  return rc == 0 ? watchManager.dataWatches : watchManager.existWatches;  }   @Override  protected boolean shouldAddWatch(int rc) {  return rc == 0 || rc == KeeperException.Code.*NONODE*.intValue();  } }  @Override protected boolean shouldAddWatch(int rc) {  return rc == 0 || rc == KeeperException.Code.*NONODE*.intValue(); }  }  //和上面类似的注册器：这是个节点数据的注册器  class DataWatchRegistration extends WatchRegistration {  public DataWatchRegistration(Watcher watcher, String clientPath) {  super(watcher, clientPath);  }   @Override  protected Map<String, Set<Watcher>> getWatches(int rc) {  return watchManager.dataWatches;  } }  // 子节点的Watches注册器  class ChildWatchRegistration extends WatchRegistration {  public ChildWatchRegistration(Watcher watcher, String clientPath) {  super(watcher, clientPath);  }   @Override  protected Map<String, Set<Watcher>> getWatches(int rc) {  return watchManager.childWatches;  } } |

### 3.23.3 构造函数

有好些构造函数，但最终都是调用的下面这个

connectString:服务端连接地址

sessionTimeout：会话超时时间

watcher：监听器watcher

canBeReadOnly：是否仅读模式

aHostProvider：ip地址解析的提供者

clientConfig：客户端配置

|  |
| --- |
| public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,  boolean canBeReadOnly, HostProvider aHostProvider,  ZKClientConfig clientConfig) throws IOException {  *LOG*.info("Initiating client connection, connectString=" + connectString  + " sessionTimeout=" + sessionTimeout + " watcher=" + watcher);   if (clientConfig == null) {  // 创建配置类  clientConfig = new ZKClientConfig();  }  this.clientConfig = clientConfig;  // 实例化watch管理器，  watchManager = defaultWatchManager();  // 管理器设置默认的Watcher  watchManager.defaultWatcher = watcher;  // 连接解析器  ConnectStringParser connectStringParser = new ConnectStringParser(  connectString);  // 这个实在上一个构造器中传过来的  // 通过调用createDefaultHostProvider方法进行创建的  // StaticHostProvider  hostProvider = aHostProvider;  // 连接管理：发送请求，接收响应等等  // getClientCnxnSocket()是获得一个底层处理socket IO的实现类  // 分别是ClientCnxnSocketNIO和ClientCnxnSocketNetty  // 默认是使用ClientCnxnSocketNIO，最终这个实例会传给SendThread  cnxn = createConnection(connectStringParser.getChrootPath(),  hostProvider, sessionTimeout, this, watchManager,  getClientCnxnSocket(), canBeReadOnly);  // 启动连接线程  cnxn.start(); }  protected ZKWatchManager defaultWatchManager() {  return new ZKWatchManager(getClientConfig().getBoolean(ZKClientConfig.*DISABLE\_AUTO\_WATCH\_RESET*)); }  private static HostProvider createDefaultHostProvider(String connectString) {  return new StaticHostProvider(  new ConnectStringParser(connectString).getServerAddresses()); } |

## 3.24 ZKClientConfig (客户端配置)

继承至ZKConfig，在这个父类中有properties属性HashMap字段，解析的配置值都放在这个map中了。就不详细展开了，这个类其实和服务端的QuorumPeerConfig作用类似，有些区别，这个是用map装了很多属性，QuorumPeerConfig是针对每个配置定义了一个字段。

## 3.25 ClientCnxn (客户端连接器)

ZooKeeper启动类，启动的就是这个类。这个类是管理客户端的socket IO。

### 3.25.1 主要属性

|  |
| --- |
| /\* ZOOKEEPER-706: If a session has a large number of watches set then  \* attempting to re-establish those watches after a connection loss may  \* fail due to the SetWatches request exceeding the server's configured  \* jute.maxBuffer value. To avoid this we instead split the watch  \* re-establishement across multiple SetWatches calls. This constant  \* controls the size of each call. It is set to 128kB to be conservative  \* with respect to the server's 1MB default for jute.maxBuffer.  \*/ private static final int *SET\_WATCHES\_MAX\_LENGTH* = 128 \* 1024;  // 权限验证信息包装类 static class AuthData {  AuthData(String scheme, byte data[]) {  this.scheme = scheme;  this.data = data;  }   String scheme;   byte data[]; }  // 认证信息集合 private final CopyOnWriteArraySet<AuthData> authInfo = new CopyOnWriteArraySet<AuthData>();  */\*\*  \* These are the packets that have been sent and are waiting for a response.  \*/*  *// 已经发送的请求数据包的集合，处于等待响应中* private final LinkedList<Packet> pendingQueue = new LinkedList<Packet>();  */\*\*  \* These are the packets that need to be sent.  \*/*  *// 待发送的数据包集合* private final LinkedBlockingDeque<Packet> outgoingQueue = new LinkedBlockingDeque<Packet>();  // 连接超时时间 private int connectTimeout;  */\*\*  \* The timeout in ms the client negotiated with the server. This is the  \* "real" timeout, not the timeout request by the client (which may have  \* been increased/decreased by the server which applies bounds to this  \* value.  \*/* private volatile int negotiatedSessionTimeout;  // 读超时 private int readTimeout;  // 会话超时 private final int sessionTimeout;  private final ZooKeeper zooKeeper;  private final ClientWatchManager watcher; // 会话id private long sessionId;  // 会话的密码 private byte sessionPasswd[] = new byte[16];  */\*\*  \* If true, the connection is allowed to go to r-o mode. This field's value  \* is sent, besides other data, during session creation handshake. If the  \* server on the other side of the wire is partitioned it'll accept  \* read-only clients only.  \*/* private boolean readOnly;  final String chrootPath;  // 发送线程 final SendThread sendThread;  // 时间监听的响应线程 final EventThread eventThread;  */\*\*  \* Set to true when close is called. Latches the connection such that we  \* don't attempt to re-connect to the server if in the middle of closing the  \* connection (client sends session disconnect to server as part of close  \* operation)  \*/* private volatile boolean closing = false;  *// 服务器集群地址提供者* private final HostProvider hostProvider;  // volatile boolean seenRwServerBefore = false;  //  public ZooKeeperSaslClient zooKeeperSaslClient;  // 配置 private final ZKClientConfig clientConfig; */\*\*  \* If any request's response in not received in configured requestTimeout  \* then it is assumed that the response packet is lost.  \*/*  *// 请求超时时间* private long requestTimeout; |

### 3.25.2 start方法

启动，实际上是是启动了发送线程和事件响应线程

这两个线程均是内部类

|  |
| --- |
| public void start() {  sendThread.start();  eventThread.start(); } |

### 3.25.3 queuePacket方法

该方法有几个重载，但都是要走下面的方法,

组装一个Packet对象，加入到outgoingQueue中

|  |
| --- |
| public Packet queuePacket(RequestHeader h, ReplyHeader r, Record request,  Record response, AsyncCallback cb, String clientPath,  String serverPath, Object ctx, WatchRegistration watchRegistration,  WatchDeregistration watchDeregistration) {  Packet packet = null;   // Note that we do not generate the Xid for the packet yet. It is  // generated later at send-time, by an implementation of ClientCnxnSocket::doIO(),  // where the packet is actually sent.  //   packet = new Packet(h, r, request, response, watchRegistration);  packet.cb = cb;  packet.ctx = ctx;  packet.clientPath = clientPath;  packet.serverPath = serverPath;  packet.watchDeregistration = watchDeregistration;  // The synchronized block here is for two purpose:  // 1. synchronize with the final cleanup() in SendThread.run() to avoid race  // 2. synchronized against each packet. So if a closeSession packet is added,  // later packet will be notified.  synchronized (state) {  if (!state.isAlive() || closing) {  // 针对关闭session或者正要关闭的session进行做一些处理  // 方法还比较长，懒得看  conLossPacket(packet);  } else {  // If the client is asking to close the session then  // mark as closing  if (h.getType() == OpCode.*closeSession*) {  closing = true;  }  // 加入到这个对列的packet，会被SendThread发送出去  outgoingQueue.add(packet);  }  }  sendThread.getClientCnxnSocket().packetAdded();  return packet; } |

### 3.25.3 内部类SendThead(发送线程)

继承至ZookeeperThread，服务于待发送的请求，已经发送心跳检测

**3.25.3.1 字段**

|  |
| --- |
| // 心跳  private long lastPingSentNs;  // Socket连接 private final ClientCnxnSocket clientCnxnSocket; private Random r = new Random();  // 是否首次连接 private boolean isFirstConnect = true; |

**3.25.3.2 run方法**

|  |
| --- |
| public void run() {  // 为sendTread，sessionId，outgoingQueue三字段赋值  clientCnxnSocket.introduce(this, sessionId, outgoingQueue);  // Time.*currentElapsedTime*(),就是当前时间，毫秒  clientCnxnSocket.updateNow();  // 更新上次发送的时间戳和心跳的时间戳  clientCnxnSocket.updateLastSendAndHeard();  int to;  // 最后一次ping服务器的时间  long lastPingRwServer = Time.*currentElapsedTime*();  // 最大发送ping的时间间隔  final int MAX\_SEND\_PING\_INTERVAL = 10000; //10 seconds  InetSocketAddress serverAddress = null;  // 连接还存活，就一直会运行下去，存活的条件是：只要不是连接关闭且认证失败的  // 状态  while (state.isAlive()) {  try {  // socket 是否还在连接着，  if (!clientCnxnSocket.isConnected()) {  // 这里执行不连接的逻辑  // don't re-establish connection if we are closing  // 如果正在关闭客户端，则跳出循环  if (closing) {  break;  }  // zookeeper服务端的地址  if (rwServerAddress != null) {  serverAddress = rwServerAddress;  rwServerAddress = null;  } else {  serverAddress = hostProvider.next(1000);  }  // 没有连接开始启动连接  startConnect(serverAddress);  // socket更新最后一次发送和心跳时间  clientCnxnSocket.updateLastSendAndHeard();  }  // 客户端还处在连接状态中  if (state.isConnected()) {  // determine whether we need to send an AuthFailed event.  // 判断是否还需要认证  if (zooKeeperSaslClient != null) {  boolean sendAuthEvent = false;  if (zooKeeperSaslClient.getSaslState() == ZooKeeperSaslClient.SaslState.*INITIAL*) {  try {  zooKeeperSaslClient.initialize(ClientCnxn.this);  } catch (SaslException e) {  *LOG*.error("SASL authentication with Zookeeper Quorum member failed: " + e);  state = States.*AUTH\_FAILED*;  sendAuthEvent = true;  }  }  KeeperState authState = zooKeeperSaslClient.getKeeperState();  if (authState != null) {  if (authState == KeeperState.*AuthFailed*) {  // An authentication error occurred during authentication with the Zookeeper Server.  // 认证失败，把状态修改为认证失败  state = States.*AUTH\_FAILED*;  sendAuthEvent = true;  } else {  if (authState == KeeperState.*SaslAuthenticated*) {  sendAuthEvent = true;  }  }  }  // 认证成功后，  if (sendAuthEvent) {  // 在事件处理线程的等待对列中添加一个需要被触发的  // 事件类型  eventThread.queueEvent(new WatchedEvent(  Watcher.Event.EventType.*None*,  authState,null));  if (state == States.*AUTH\_FAILED*) {  eventThread.queueEventOfDeath();  }  }  }  // 读超时时间减去(上次心跳到这次连接的时间差)  to = readTimeout - clientCnxnSocket.getIdleRecv();  } else {  // 连接超时时间减去(上次心跳到这次连接的时间差)  to = connectTimeout - clientCnxnSocket.getIdleRecv();  }  // to目的是为了判断连接是否超时了  if (to <= 0) {  // 说明会话超时了  String warnInfo;  warnInfo = "Client session timed out, have not heard from server in "  + clientCnxnSocket.getIdleRecv()  + "ms"  + " for sessionid 0x"  + Long.*toHexString*(sessionId);  *LOG*.warn(warnInfo);  throw new SessionTimeoutException(warnInfo);  }  //   if (state.isConnected()) {  //1000(1 second) is to prevent race condition missing to send the second ping  //also make sure not to send too many pings when readTimeout is small  // 计算下次Ping时间  int timeToNextPing = readTimeout / 2 - clientCnxnSocket.getIdleSend() -   ((clientCnxnSocket.getIdleSend() > 1000) ? 1000 : 0);  //send a ping request either time is due or no packet sent out within MAX\_SEND\_PING\_INTERVAL  if (timeToNextPing <= 0 || clientCnxnSocket.getIdleSend() > MAX\_SEND\_PING\_INTERVAL) {  // 进行心跳Ping  sendPing();  clientCnxnSocket.updateLastSend();  } else {  if (timeToNextPing < to) {  to = timeToNextPing;  }  }  }   // If we are in read-only mode, seek for read/write server  // 如果状态是仅读状态连接  if (state == States.*CONNECTEDREADONLY*) {  long now = Time.*currentElapsedTime*();  int idlePingRwServer = (int) (now - lastPingRwServer);  if (idlePingRwServer >= pingRwTimeout) {  lastPingRwServer = now;  idlePingRwServer = 0;  pingRwTimeout =  Math.*min*(2\*pingRwTimeout, *maxPingRwTimeout*);  pingRwServer();  }  to = Math.*min*(to, pingRwTimeout - idlePingRwServer);  }  // 发送请求  clientCnxnSocket.doTransport(to, pendingQueue, ClientCnxn.this);  } catch (Throwable e) {  if (closing) {  if (*LOG*.isDebugEnabled()) {  // closing so this is expected  *LOG*.debug("An exception was thrown while closing send thread for session 0x"  + Long.*toHexString*(getSessionId())  + " : " + e.getMessage());  }  break;  } else {  // this is ugly, you have a better way speak up  if (e instanceof SessionExpiredException) {  *LOG*.info(e.getMessage() + ", closing socket connection");  } else if (e instanceof SessionTimeoutException) {  *LOG*.info(e.getMessage() + *RETRY\_CONN\_MSG*);  } else if (e instanceof EndOfStreamException) {  *LOG*.info(e.getMessage() + *RETRY\_CONN\_MSG*);  } else if (e instanceof RWServerFoundException) {  *LOG*.info(e.getMessage());  } else if (e instanceof SocketException) {  *LOG*.info("Socket error occurred: {}: {}", serverAddress, e.getMessage());  } else {  *LOG*.warn("Session 0x{} for server {}, unexpected error{}",  Long.*toHexString*(getSessionId()),  serverAddress,  *RETRY\_CONN\_MSG*,  e);  }  // At this point, there might still be new packets appended to outgoingQueue.  // they will be handled in next connection or cleared up if closed.  cleanAndNotifyState();  }  }  }  synchronized (state) {  // When it comes to this point, it guarantees that later queued  // packet to outgoingQueue will be notified of death.  cleanup();  }  clientCnxnSocket.close();  if (state.isAlive()) {  eventThread.queueEvent(new WatchedEvent(Event.EventType.*None*,  Event.KeeperState.*Disconnected*, null));  }  eventThread.queueEvent(new WatchedEvent(Event.EventType.*None*,  Event.KeeperState.*Closed*, null));  ZooTrace.*logTraceMessage*(*LOG*, ZooTrace.*getTextTraceLevel*(),  "SendThread exited loop for session: 0x"  + Long.*toHexString*(getSessionId())); }  // 发送Ping  private void sendPing() {  lastPingSentNs = System.*nanoTime*();  RequestHeader h = new RequestHeader(-2, OpCode.*ping*);  queuePacket(h, null, null, null, null, null, null, null, null); } |

**3.25.3.3 startConnect方法**

启动连接

|  |
| --- |
| private void startConnect(InetSocketAddress addr) throws IOException {  // initializing it for new connection  // sasl登陆失败标志  saslLoginFailed = false;  if(!isFirstConnect){  // 不是首次连接，则线程沉睡0-1s之间的随机时间  try {  Thread.*sleep*(r.nextInt(1000));  } catch (InterruptedException e) {  *LOG*.warn("Unexpected exception", e);  }  }  //ZooKeeper将状态修改为正在连接中  state = States.*CONNECTING*;  // 服务器的地址端口  String hostPort = addr.getHostString() + ":" + addr.getPort();  MDC.*put*("myid", hostPort);  // 设置线程的名称，设置成服务器地址端口  setName(getName().replaceAll("\\(.\*\\)", "(" + hostPort + ")"));  if (clientConfig.isSaslClientEnabled()) {  try {  if (zooKeeperSaslClient != null) {  zooKeeperSaslClient.shutdown();  }  zooKeeperSaslClient = new ZooKeeperSaslClient(SaslServerPrincipal.*getServerPrincipal*(addr, clientConfig),  clientConfig);  } catch (LoginException e) {  // An authentication error occurred when the SASL client tried to initialize:  // for Kerberos this means that the client failed to authenticate with the KDC.  // This is different from an authentication error that occurs during communication  // with the Zookeeper server, which is handled below.  *LOG*.warn("SASL configuration failed: " + e + " Will continue connection to Zookeeper server without "  + "SASL authentication, if Zookeeper server allows it.");  eventThread.queueEvent(new WatchedEvent(  Watcher.Event.EventType.*None*,  Watcher.Event.KeeperState.*AuthFailed*, null));  saslLoginFailed = true;  }  }  // 将连接信息输出到日志  logStartConnect(addr);  // socket 连接  clientCnxnSocket.connect(addr); } |

**3.25.3.4 primeConnect方法**

设置session、watches、authentication

|  |
| --- |
| void primeConnection() throws IOException {  *LOG*.info("Socket connection established, initiating session, client: {}, server: {}",  clientCnxnSocket.getLocalSocketAddress(),  clientCnxnSocket.getRemoteSocketAddress());  isFirstConnect = false;  long sessId = (seenRwServerBefore) ? sessionId : 0;  // 创建连接请求  ConnectRequest conReq = new ConnectRequest(0, lastZxid,  sessionTimeout, sessId, sessionPasswd);  // We add backwards since we are pushing into the front  // Only send if there's a pending watch  // *TODO: here we have the only remaining use of zooKeeper in* // this class. It's to be eliminated!  if (!clientConfig.getBoolean(ZKClientConfig.*DISABLE\_AUTO\_WATCH\_RESET*)) {  List<String> dataWatches = zooKeeper.getDataWatches();  List<String> existWatches = zooKeeper.getExistWatches();  List<String> childWatches = zooKeeper.getChildWatches();  if (!dataWatches.isEmpty()  || !existWatches.isEmpty() || !childWatches.isEmpty()) {  Iterator<String> dataWatchesIter = prependChroot(dataWatches).iterator();  Iterator<String> existWatchesIter = prependChroot(existWatches).iterator();  Iterator<String> childWatchesIter = prependChroot(childWatches).iterator();  long setWatchesLastZxid = lastZxid;   while (dataWatchesIter.hasNext()  || existWatchesIter.hasNext() || childWatchesIter.hasNext()) {  List<String> dataWatchesBatch = new ArrayList<String>();  List<String> existWatchesBatch = new ArrayList<String>();  List<String> childWatchesBatch = new ArrayList<String>();  int batchLength = 0;   // Note, we may exceed our max length by a bit when we add the last  // watch in the batch. This isn't ideal, but it makes the code simpler.  while (batchLength < *SET\_WATCHES\_MAX\_LENGTH*) {  final String watch;  if (dataWatchesIter.hasNext()) {  watch = dataWatchesIter.next();  dataWatchesBatch.add(watch);  } else if (existWatchesIter.hasNext()) {  watch = existWatchesIter.next();  existWatchesBatch.add(watch);  } else if (childWatchesIter.hasNext()) {  watch = childWatchesIter.next();  childWatchesBatch.add(watch);  } else {  break;  }  batchLength += watch.length();  }   SetWatches sw = new SetWatches(setWatchesLastZxid,  dataWatchesBatch,  existWatchesBatch,  childWatchesBatch);  RequestHeader header = new RequestHeader(-8, OpCode.*setWatches*);  Packet packet = new Packet(header, new ReplyHeader(), sw, null, null);  outgoingQueue.addFirst(packet);  }  }  }   for (AuthData id : authInfo) {  outgoingQueue.addFirst(new Packet(new RequestHeader(-4,  OpCode.*auth*), null, new AuthPacket(0, id.scheme,  id.data), null, null));  }  outgoingQueue.addFirst(new Packet(null, null, conReq,  null, null, readOnly));  clientCnxnSocket.connectionPrimed();  if (*LOG*.isDebugEnabled()) {  *LOG*.debug("Session establishment request sent on "  + clientCnxnSocket.getRemoteSocketAddress());  } } |

### 3.25.4 内部类EventThead(事件线程)

负责客户端的事件处理，并触发客户端注册的Watcher监听。EventThread中有一个waitingEvents对列，用于临时存放那些需要被触发的Object。包括哪些客户端注册的Watcher和异步接口中注册的回调器AsyncCallback。同时，EventThread会不断地从waitingEvents这个对列中取出Object，识别出其具体类型，并分别调用process和processResult接口方法来实现对事件的触发和回调。

**3.25.4.1 字段**

|  |
| --- |
| // 等待被触发的Object，Watcher或者回调器  private final LinkedBlockingQueue<Object> waitingEvents =  new LinkedBlockingQueue<Object>();  private volatile KeeperState sessionState = KeeperState.*Disconnected*;  private volatile boolean wasKilled = false; private volatile boolean isRunning = false; |

## 3.26 ClientCnxnSocketNIO(客户端处理Socket IO)

### 3.26.1 doTransport

|  |
| --- |
| void doTransport(int waitTimeOut, List<Packet> pendingQueue, ClientCnxn cnxn)  throws IOException, InterruptedException {  // 查看是否有准备就绪的通道，是一个阻塞的操作  selector.select(waitTimeOut);  // 获取准备就绪的Selector的key集合  Set<SelectionKey> selected;  synchronized (this) {  selected = selector.selectedKeys();  }  // Everything below and until we get back to the select is  // non blocking, so time is effectively a constant. That is  // Why we just have to do this once, here  updateNow();  for (SelectionKey k : selected) {  SocketChannel sc = ((SocketChannel) k.channel());  if ((k.readyOps() & SelectionKey.*OP\_CONNECT*) != 0) {  // 若是连接事件，  if (sc.finishConnect()) {  updateLastSendAndHeard();  updateSocketAddresses();  sendThread.primeConnection();  }  } else if ((k.readyOps() & (SelectionKey.*OP\_READ* | SelectionKey.*OP\_WRITE*)) != 0) {  // 读和写事件  doIO(pendingQueue, cnxn);  }  }  if (sendThread.getZkState().isConnected()) {  if (findSendablePacket(outgoingQueue,  sendThread.tunnelAuthInProgress()) != null) {  enableWrite();  }  }  selected.clear(); } |

### 3.26.2 doIO

处理读和写事件

|  |
| --- |
| void doIO(List<Packet> pendingQueue, ClientCnxn cnxn)  throws InterruptedException, IOException {  SocketChannel sock = (SocketChannel) sockKey.channel();  if (sock == null) {  throw new IOException("Socket is null!");  }  // 处理读  if (sockKey.isReadable()) {  int rc = sock.read(incomingBuffer);  if (rc < 0) {  throw new EndOfStreamException(  "Unable to read additional data from server sessionid 0x"  + Long.*toHexString*(sessionId)  + ", likely server has closed socket");  }  if (!incomingBuffer.hasRemaining()) {  incomingBuffer.flip();  if (incomingBuffer == lenBuffer) {  recvCount.getAndIncrement();  readLength();  } else if (!initialized) {  readConnectResult();  enableRead();  if (findSendablePacket(outgoingQueue,  sendThread.tunnelAuthInProgress()) != null) {  // Since SASL authentication has completed (if client is configured to do so),  // outgoing packets waiting in the outgoingQueue can now be sent.  enableWrite();  }  lenBuffer.clear();  incomingBuffer = lenBuffer;  updateLastHeard();  initialized = true;  } else {  sendThread.readResponse(incomingBuffer);  lenBuffer.clear();  incomingBuffer = lenBuffer;  updateLastHeard();  }  }  }  // 处理写  if (sockKey.isWritable()) {  Packet p = findSendablePacket(outgoingQueue,  sendThread.tunnelAuthInProgress());   if (p != null) {  updateLastSend();  // If we already started writing p, p.bb will already exist  if (p.bb == null) {  if ((p.requestHeader != null) &&  (p.requestHeader.getType() != OpCode.*ping*) &&  (p.requestHeader.getType() != OpCode.*auth*)) {  p.requestHeader.setXid(cnxn.getXid());  }  p.createBB();  }  sock.write(p.bb);  if (!p.bb.hasRemaining()) {  sentCount.getAndIncrement();  outgoingQueue.removeFirstOccurrence(p);  if (p.requestHeader != null  && p.requestHeader.getType() != OpCode.*ping* && p.requestHeader.getType() != OpCode.*auth*) {  synchronized (pendingQueue) {  pendingQueue.add(p);  }  }  }  }  if (outgoingQueue.isEmpty()) {  // No more packets to send: turn off write interest flag.  // Will be turned on later by a later call to enableWrite(),  // from within ZooKeeperSaslClient (if client is configured  // to attempt SASL authentication), or in either doIO() or  // in doTransport() if not.  disableWrite();  } else if (!initialized && p != null && !p.bb.hasRemaining()) {  // On initial connection, write the complete connect request  // packet, but then disable further writes until after  // receiving a successful connection response. If the  // session is expired, then the server sends the expiration  // response and immediately closes its end of the socket. If  // the client is simultaneously writing on its end, then the  // TCP stack may choose to abort with RST, in which case the  // client would never receive the session expired event. See  // http://docs.oracle.com/javase/6/docs/technotes/guides/net/articles/connection\_release.html  disableWrite();  } else {  // Just in case  enableWrite();  }  } } |

# debug-demo

## 4.1 会话建立的demo

|  |
| --- |
| package org.apache.myzookeeper;  import org.apache.zookeeper.WatchedEvent; import org.apache.zookeeper.Watcher; import org.apache.zookeeper.ZooKeeper;  import java.util.concurrent.CountDownLatch;  public class ZookeeperTest1 implements Watcher {   private static CountDownLatch *latch* = new CountDownLatch(1);   public static void main(String[] args) throws Exception {   ZooKeeper zooKeeper = new ZooKeeper("127.0.0.1:2181", 5000, new ZookeeperTest1());  System.*err*.println("client getState = " + zooKeeper.getState());  *latch*.await();  System.*out*.println("zookeeper session established");   }    @Override  public void process(WatchedEvent event) {  System.*err*.println("Receive watched event: " + event);  if (Event.KeeperState.*SyncConnected* == event.getState()) {  *latch*.countDown();  }  } } |