day02

1、zookeeper源码解析

1）下载zookeeper源码，导入IDEA中

下载地址：https://github.com/apache/zookeeper

2）启动

根据bin目录下的启动脚本zkServer.sh中加载启动类QuorumPeerMain类

```
then
  # for some reason these two options are necessary on jdk6 on Ubuntu
  #   accord to the docs they are not necessary, but otw jconsole cannot
  #   do a local attach
  ZOOMAIN="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY org.apache.zookeeper.server.quorum.QuorumPeerMain"
else
  if [ "x$JMXAUTH" = "x" ]
  then
```

QuorumPeerMain中main方法执行initializeAndRun方法

```
public static void main(String[] args) {
    QuorumPeerMain main = new QuorumPeerMain();
    try {
        main.initializeAndRun(args);
    } catch (IllegalArgumentException e) {
        LOG.error("Invalid arguments, exiting abnormally", e);
        LOG.info(USAGE);
```

跟进initializeAndRun方法

```
 protected void initializeAndRun(String[] args)        throws ConfigException,
IOException, AdminServerException    {        QuorumPeerConfig config = new
QuorumPeerConfig();        if (args.length == 1) {            //解析配置文件
     config.parse(args[0]);        }        // Start and schedule the the purge
task        //启动清除任务        DatadirCleanupManager purgeMgr = new
DatadirCleanupManager(config                .getDataDir(),
config.getDataLogDir(), config                .getSnapRetainCount(),
config.getPurgeInterval());        purgeMgr.start();        //判断单机环境还是集群环
境        if (args.length == 1 && config.isDistributed()) {            //启动集群
          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                //启动单机            ZooKeeperServerMain.main(args);
     }    }
```

在initializeAndRun方法中主要做了三件事

- 加载解析配置文件

```
.build()).create(path);                                Properties cfg = new
Properties();            FileInputStream in = new FileInputStream(configFile);
         try {                cfg.load(in);                configFileStr = path;
           } finally {               in.close();           }            /*
Read entire config file as initial configuration */           initialConfig =
new String(Files.readAllBytes(configFile.toPath()));
parseProperties(cfg);
```

将配置文件加载到Properties cfg对象中，解析cfg对象。zookeeper所有配置信息封装到一个QuorumPeerConfig对象中

- 启动定时清除任务

  PurgeTask继承TimeTask，定时执行run方法中的purge方法

  ```
  public void run() {    LOG.info("Purge task started.");    try {
  PurgeTxnLog.purge(logsDir, snapsDir, snapRetainCount);    } catch (Exception
  e) {        LOG.error("Error occurred while purging.", e);    }
  LOG.info("Purge task completed.");}
  ```

purge方法主要清除旧的快照和日志文件

- 启动zk

  zookeeper启动方式分为两种：单机启动和集群启动

  ```
        if (args.length == 1 && config.isDistributed()) {              //集群启
  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);        }
  ```

  首先我们看看单机启动的源码 main方法调用initializeAndRun方法，initializeAndRun首先加载配置文件，然后执行runFromConfig(config)方法，我们看看runFromConfig具体执行了什么操作

```
   FileTxnSnapLog txnLog = null;        try {            try {
metricsProvider = MetricsProviderBootstrap
.startMetricsProvider(config.getMetricsProviderClassName(),
                              config.getMetricsProviderConfiguration());
       } catch (MetricsProviderLifeCycleException error) {
throw new IOException("Cannot boot MetricsProvider
"+config.getMetricsProviderClassName(),                      error);
}            ServerMetrics.metricsProviderInitialized(metricsProvider);
       // 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            txnLog = new FileTxnSnapLog(config.dataLogDir,
config.dataDir);            JvmPauseMonitor jvmPauseMonitor = null;
   if(config.jvmPauseMonitorToRun) {                jvmPauseMonitor = new
JvmPauseMonitor(config);            }            final ZooKeeperServer
zkServer = new ZooKeeperServer(jvmPauseMonitor, txnLog,
config.tickTime, config.minSessionTimeout, config.maxSessionTimeout,
           config.listenBacklog, null, config.initialConfig);
txnLog.setServerStats(zkServer.serverStats());            // Registers
shutdown handler which will be used to know the            // server error
or shutdown state changes.            final CountDownLatch shutdownLatch =
new CountDownLatch(1);            zkServer.registerServerShutdownHandler(
                new ZooKeeperServerShutdownHandler(shutdownLatch));
   // Start Admin server            adminServer =
AdminServerFactory.createAdminServer();
adminServer.setZooKeeperServer(zkServer);            adminServer.start();
        boolean needStartZKServer = true;            if
(config.getClientPortAddress() != null) {            cnxnFactory =
ServerCnxnFactory.createFactory();
cnxnFactory.configure(config.getClientPortAddress(),
config.getMaxClientCnxns(),
config.getClientPortListenBacklog(), false);                //
******************  启动zookeeper server  ******************
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(),
config.getClientPortListenBacklog(), 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);        }
```

```
                metricsProvider.stop();                } catch (Throwable
error) {                    LOG.warn("Error while stopping metrics", error);
            }            }        }    }
```

启动过程首先开启一下metrics监控,然后启动admin server,然后启动zk server，我们来看看启动过程

ServerCnxnFactory中startup方法调用NettyServerCnxnFactory实现类启动方法

```
  public void startup(ZooKeeperServer zks, boolean startServer)
throws IOException, InterruptedException {        start();
setZooKeeperServer(zks);        if (startServer) {
zks.startdata();            zks.startup();        }    }
```

启动方法执行操作

```
      if (sessionTracker == null) {            createSessionTracker();
}        startSessionTracker();        setupRequestProcessors();
registerJMX();        startJvmPauseMonitor();        registerMetrics();
   setState(State.RUNNING);        notifyAll();
```

接下来我们看看集群启动过程

```
ManagedUtil.registerLog4jMBeans();      } catch (JMException e) {
LOG.warn("Unable to register log4j JMX control", e);      }
LOG.info("Starting quorum peer");      MetricsProvider metricsProvider;
try {        metricsProvider = MetricsProviderBootstrap
.startMetricsProvider(config.getMetricsProviderClassName(),
                          config.getMetricsProviderConfiguration());      }
catch (MetricsProviderLifeCycleException error) {        throw new
IOException("Cannot boot MetricsProvider " +
config.getMetricsProviderClassName(),                      error);      }
   try {          ServerMetrics.metricsProviderInitialized(metricsProvider);
         ServerCnxnFactory cnxnFactory = null;          ServerCnxnFactory
secureCnxnFactory = null;          if (config.getClientPortAddress() !=
null) {              cnxnFactory = ServerCnxnFactory.createFactory();
       cnxnFactory.configure(config.getClientPortAddress(),
     config.getMaxClientCnxns(),
config.getClientPortListenBacklog(), false);          }          if
(config.getSecureClientPortAddress() != null) {
secureCnxnFactory = ServerCnxnFactory.createFactory();
secureCnxnFactory.configure(config.getSecureClientPortAddress(),
         config.getMaxClientCnxns(),
config.getClientPortListenBacklog(), true);          }          quorumPeer =
getQuorumPeer();          quorumPeer.setTxnFactory(new FileTxnSnapLog(
              config.getDataLogDir(),
config.getDataDir()));
quorumPeer.enableLocalSessions(config.areLocalSessionsEnabled());
quorumPeer.enableLocalSessionsUpgrading(
config.isLocalSessionsUpgradingEnabled());
//quorumPeer.setQuorumPeers(config.getAllMembers());
quorumPeer.setElectionType(config.getElectionAlg());
quorumPeer.setMyid(config.getServerId());
quorumPeer.setTickTime(config.getTickTime());
quorumPeer.setMinSessionTimeout(config.getMinSessionTimeout());
quorumPeer.setMaxSessionTimeout(config.getMaxSessionTimeout());
quorumPeer.setInitLimit(config.getInitLimit());
quorumPeer.setSyncLimit(config.getSyncLimit());
quorumPeer.setObserverMasterPort(config.getObserverMasterPort());
quorumPeer.setConfigFileName(config.getConfigFilename());
quorumPeer.setClientPortListenBacklog(config.getClientPortListenBacklog());
         quorumPeer.setZKDatabase(new
ZKDatabase(quorumPeer.getTxnFactory()));
quorumPeer.setQuorumVerifier(config.getQuorumVerifier(), false);          if
(config.getLastSeenQuorumVerifier()!=null) {
quorumPeer.setLastSeenQuorumVerifier(config.getLastSeenQuorumVerifier(),
false);          }          quorumPeer.initConfigInZKDatabase();
quorumPeer.setCnxnFactory(cnxnFactory);
quorumPeer.setSecureCnxnFactory(secureCnxnFactory);
quorumPeer.setSslQuorum(config.isSslQuorum());
quorumPeer.setUsePortUnification(config.shouldUsePortUnification());
quorumPeer.setLearnerType(config.getPeerType());
quorumPeer.setSyncEnabled(config.getSyncEnabled());
quorumPeer.setQuorumListenOnAllIPs(config.getQuorumListenOnAllIPs());
   if (config.sslQuorumReloadCertFiles) {
quorumPeer.getX509Util().enableCertFileReloading();          }          //
sets quorum sasl authentication configurations
quorumPeer.setQuorumSaslEnabled(config.quorumEnableSasl);
if(quorumPeer.isQuorumSaslAuthEnabled()){
```

```
quorumPeer.setQuorumLearnerSaslRequired(config.quorumLearnerRequireSasl);

quorumPeer.setQuorumServicePrincipal(config.quorumServicePrincipal);

quorumPeer.setQuorumServerLoginContext(config.quorumServerLoginContext);

quorumPeer.setQuorumLearnerLoginContext(config.quorumLearnerLoginContext);
       }
quorumPeer.setQuorumCnxnThreadsSize(config.quorumCnxnThreadsSize);
quorumPeer.initialize();          if(config.jvmPauseMonitorToRun) {
     quorumPeer.setJvmPauseMonitor(new JvmPauseMonitor(config));          }
         quorumPeer.start();          quorumPeer.join();      } catch
(InterruptedException e) {          // warn, but generally this is ok
   LOG.warn("Quorum Peer interrupted", e);      } finally {          if
(metricsProvider != null) {              try {
metricsProvider.stop();             } catch (Throwable error) {
     LOG.warn("Error while stopping metrics", error);             }
}      }    }
```

在runFromConfig执行过程中主要是QuorumPeer对象属性的赋值并执行start方法，通过查看QuorumPeer类的源码，发现QuorumPeer继承了ZooKeeperThread，而ZooKeeperThread继承了Thread,通过start方法启动了QuorumPeer线程，线程运行执行线程的run方法

```
     case LOOKING:                    LOG.info("LOOKING");
ServerMetrics.getMetrics().LOOKING_COUNT.add(1);                    if
(Boolean.getBoolean("readonlymode.enabled")) {
LOG.info("Attempting to start ReadOnlyZooKeeperServer");
   // Create read-only server but don't start it immediately
       final ReadOnlyZooKeeperServer roZk =                            new
ReadOnlyZooKeeperServer(logFactory, this, this.zkDb);
     // Instead of starting roZk immediately, wait some grace
         // period before we decide we're partitioned.
//                       // Thread is used here because otherwise it would
require                        // changes in each of election strategy
classes which is                        // unnecessary code coupling.
                 Thread roZkMgr = new Thread() {
public void run() {                                 try {
               // lower-bound grace period to 2 secs
            sleep(Math.max(2000, tickTime));
   if (ServerState.LOOKING.equals(getPeerState())) {
               roZk.startup();                                       }
                       } catch (InterruptedException e) {
                 LOG.info("Interrupted while attempting to start
ReadOnlyZooKeeperServer, not started");                               }
catch (Exception e) {                                    LOG.error("FAILED
to start ReadOnlyZooKeeperServer", e);                                }
                       }                          };
try {                           roZkMgr.start();
reconfigFlagClear();                               if (shuttingDownLE) {
                      shuttingDownLE = false;
   startLeaderElection();                                 }
     setCurrentVote(makeLEStrategy().lookForLeader());
} catch (Exception e) {                            LOG.warn("Unexpected
exception", e);
setPeerState(ServerState.LOOKING);                          } finally {
                    // If the thread is in the the grace period, interrupt
                         // to come out of waiting.
     roZkMgr.interrupt();                                roZk.shutdown();
                 }                    } else {                          try {
                          reconfigFlagClear();                               if
(shuttingDownLE) {                                 shuttingDownLE = false;
                          startLeaderElection();
  }
setCurrentVote(makeLEStrategy().lookForLeader());                          }
catch (Exception e) {                              LOG.warn("Unexpected
exception", e);
setPeerState(ServerState.LOOKING);                             }
                         }                    break;
//OBSERVING状态                  case OBSERVING:                     try {
                   LOG.info("OBSERVING");
setObserver(makeObserver(logFactory));
observer.observeLeader();                     } catch (Exception e) {
               LOG.warn("Unexpected exception",e );                     }
finally {                          observer.shutdown();
setObserver(null);                             updateServerState();
         // Add delay jitter before we switch to LOOKING
   // state to reduce the load of ObserverMaster                          if
(isRunning()) {
Observer.waitForObserverElectionDelay();                              }
```

```
LOG.info("FOLLOWING");
setFollower(makeFollower(logFactory));
follower.followLeader();                    } catch (Exception e) {
             LOG.warn("Unexpected exception",e);                    }
finally {                       follower.shutdown();
setFollower(null);                          updateServerState();
     }                    break;                //LEADING状态
case LEADING:                       LOG.info("LEADING");                    try
{                         setLeader(makeLeader(logFactory));
       leader.lead();                           setLeader(null);
     } catch (Exception e) {                        LOG.warn("Unexpected
exception",e);                     } finally {                         if
(leader != null) {                          leader.shutdown("Forcing
shutdown");                           setLeader(null);
   }                         updateServerState();                    }
             break;                }                start_fle =
Time.currentElapsedTime();            }
```

核心逻辑在while循环中，判断节点的状态，分为`LOOKING`、`OBSERVING`、`FOLLOWING`、`LEADING`，当某个QuorumPeerq刚启动时，状态为`LOOKING`，启动线程将zk节点启动，然后进行leader选举，这是zookeeper的选举算法的核心，leader的选举在org.apache.zookeeper.server.quorum.FastLeaderElection的lookForLeader方法中

3）leader选举

```
           //记录当前server接受其他server的本轮投票信息              Map<Long, Vote>
recvset = new HashMap<Long, Vote>();            //选举结束后法定server的投票信息
       Map<Long, Vote> outofelection = new HashMap<Long, Vote>();            //
选举超时时限             int notTimeout = minNotificationInterval;
synchronized(this){                 //逻辑时钟+1
logicalclock.incrementAndGet();                 //初始化选票，给自己投票
   updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
   }                        LOG.info("New election. My id =  " + self.getId() +
                  ", proposed zxid=0x" + Long.toHexString(proposedZxid));
     //向所有节点发送选票信息             sendNotifications();
```

此处两个变量，一个recvset，用来保存当前server的接受其他server的本轮投票信息，key为当前server的id，也即是我们在配置文件中配置的myid，而另外一个变量outofelection保存选举结束以后法定的server的投票信息，这里的法定指的是FOLLOWING和LEADING状态的server，不包活OBSERVING状态的server。

更新逻辑时钟，此处逻辑时钟是为了在选举leader时比较其他选票中的server中的epoch和本地谁最新，然后将自己的选票proposal发送给其他所有server。

```
self.getQuorumVerifier();            ToSend notmsg = new
ToSend(ToSend.mType.notification,                    proposedLeader,
       proposedZxid,                   logicalclock.get(),
QuorumPeer.ServerState.LOOKING,                    sid,
proposedEpoch, qv.toString().getBytes());            if(LOG.isDebugEnabled()){
            LOG.debug("Sending Notification: " + proposedLeader + " (n.leader),
0x"  +                      Long.toHexString(proposedZxid) + " (n.zxid), 0x" +
Long.toHexString(logicalclock.get())  +                      " (n.round), " +
sid + " (recipient), " + self.getId() +                      " (myid), 0x" +
Long.toHexString(proposedEpoch) + " (n.peerEpoch)");            }
sendqueue.offer(notmsg);        }    }
```

此方法遍历所有投票参与者集合，将选票信息构造成一个ToSend对象，分别发送消息放置到队列sendqueue中。同理集群中每一个server节点都会将自己的选票发送给其他server，那么既然有发送选票，肯定存在接受选票信息，并选出leader，接下来我们就来看看每一个server如何接受选票并处理的。

首先我们应该从队列出取出选票信息

```
    /*                * Remove next notification from queue, times out after 2
times                * the termination time                * 从队列中取出一个选票
信息                */                Notification n =
recvqueue.poll(notTimeout,                        TimeUnit.MILLISECONDS);
        if(n == null){                //判断是否投递过选票信息
   if(manager.haveDelivered()){                        //重新发送选票信息
            sendNotifications();                    } else {
     //重连所有server                        manager.connectAll();
     }                /*                        * Exponential backoff
         */                    int tmpTimeOut = notTimeout*2;
notTimeout = (tmpTimeOut < maxNotificationInterval?
tmpTimeOut : maxNotificationInterval);                    LOG.info("Notification
time out: " + notTimeout);                }
```

选出的选票信息封装在一个`Notification`对象中，如果取出的选票为null，我们通过QuorumCnxManager检查发送队列中是否投递过选票，如果投递过说明连接并没有断开，则重新发送选票到其他sever，否则，说明连接断开，重连所有server即可。那么连接没有断开，为什么会收不到选票信息呢，有可能是选票超时时限导致没有收到选票，所有将选票时限延长了一倍。

```
            //校验选票中选举server和选举的leader sever是否合法                else if
(validVoter(n.sid) && validVoter(n.leader)) {                    /*
       * Only proceed if the vote comes from a replica in the current or next
                * voting view for a replica in the current or next voting
view.                    */                    switch (n.state) {
     case LOOKING:                    ......                        break;
            case OBSERVING:                        LOG.debug("Notification from
observer: " + n.sid);                        break;                    case
FOLLOWING:                    case LEADING:                    ......
                        }
```

如果选出的选票Notification不为null，校验投票server和选举leader是否合法，然后根据选票状态执行不同分支，选举过程走LOOKING分支，接下来比较选票epoch和当前逻辑时钟

发送新的投票给其他所有server。

```
  if (getInitLastLoggedZxid() == -1) {
LOG.debug("Ignoring notification as our zxid is -1");
break;                          }                          if (n.zxid == -1) {
                    LOG.debug("Ignoring notification from member with -1 zxid"
+ n.sid);                                 break;                          }
           // If notification > current, replace and send messages out
             if (n.electionEpoch > logicalclock.get()) {
   logicalclock.set(n.electionEpoch);
recvset.clear();                                 //比较选票和自己谁更适合做leader，比较规则
epoch>zxid>sid                                 if(totalOrderPredicate(n.leader,
n.zxid, n.peerEpoch,                                          getInitId(),
getInitLastLoggedZxid(), getPeerEpoch())) {
updateProposal(n.leader, n.zxid, n.peerEpoch);                                 } else
{                                 updateProposal(getInitId(),
                  getInitLastLoggedZxid(),
   getPeerEpoch());                                 }
sendNotifications();                              }
```

如果选票epoch<逻辑时钟,zk放弃此次选票，不做任何处理。

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

如果选票epoch=逻辑时钟,仍然是比较选票和当前自己server谁更适合当leader，并重新更新选票，发送给其他所有的server

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

接下来将收到的选票放入recvset的map中保存。

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

接下来是判断本轮选举是否结束，如果超过半数的，则leader预选举结束，注意此时还要比较其他少半选票中有没有谁更适合做leader？如果在选票找不到任何一个server比当前server更适合做leader，则更新更新server状态，清空recvqueue队列，确定最终选票并返回，否则将更适合做leader的Notification放回队列开始新一轮的选举。

```
(voteSet.hasAllQuorums()) {            // Verify if there is any change in the
proposed leader            //比较剩下少数的server是否更适合做leader
while((n = recvqueue.poll(finalizeWait,            TimeUnit.MILLISECONDS)) !=
null){               if(totalOrderPredicate(n.leader, n.zxid, n.peerEpoch,
          proposedLeader, proposedZxid, proposedEpoch)){
recvqueue.put(n);                     break;               }           }
       /*           * This predicate is true once we don't read any new
   * relevant message from the reception queue            如果全部比较都没有当前谁比
当前server更适合做leader，则更新server状态            */           if (n == null) {
              //更新状态              setPeerState(proposedLeader, voteSet);
            //构建最终选票，便于其他server同步                Vote endVote = new
Vote(proposedLeader,               proposedZxid, logicalclock.get(),
      proposedEpoch);              //清空队列
leaveInstance(endVote);              return endVote;           }     }
```

更新状态后，若选票中的服务器状态为FOLLOWING或者LEADING时，其大致步骤会再次判断选举epoch是否等于逻辑时钟.如果相等，再次盘检查选中的leader过半

```
 if(n.electionEpoch == logicalclock.get()){//
recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));
                        voteSet = getVoteTracker(recvset, new Vote(n.version,
                                n.leader, n.zxid, n.electionEpoch,
n.peerEpoch, n.state));                              if (voteSet.hasAllQuorums()
&&                           checkLeader(outofelection, n.leader,
n.electionEpoch)) {                              setPeerState(n.leader,
voteSet);                             Vote endVote = new Vote(n.leader,
                            n.zxid, n.electionEpoch, n.peerEpoch);
                    leaveInstance(endVote);
return endVote;                              }                        }
            /*                          * Before joining an established
ensemble, verify that                          * a majority are following the
same leader.                          */
outofelection.put(n.sid, new Vote(n.version, n.leader,
     n.zxid, n.electionEpoch, n.peerEpoch, n.state));
voteSet = getVoteTracker(outofelection, new Vote(n.version,
           n.leader, n.zxid, n.electionEpoch, n.peerEpoch, n.state));
            if (voteSet.hasAllQuorums() &&
checkLeader(outofelection, n.leader, n.electionEpoch)) {
   synchronized(this){
logicalclock.set(n.electionEpoch);
setPeerState(n.leader, voteSet);                              }
         Vote endVote = new Vote(n.leader, n.zxid,
      n.electionEpoch, n.peerEpoch);
leaveInstance(endVote);                              return endVote;
         }
```

2、zookeeper应用场景

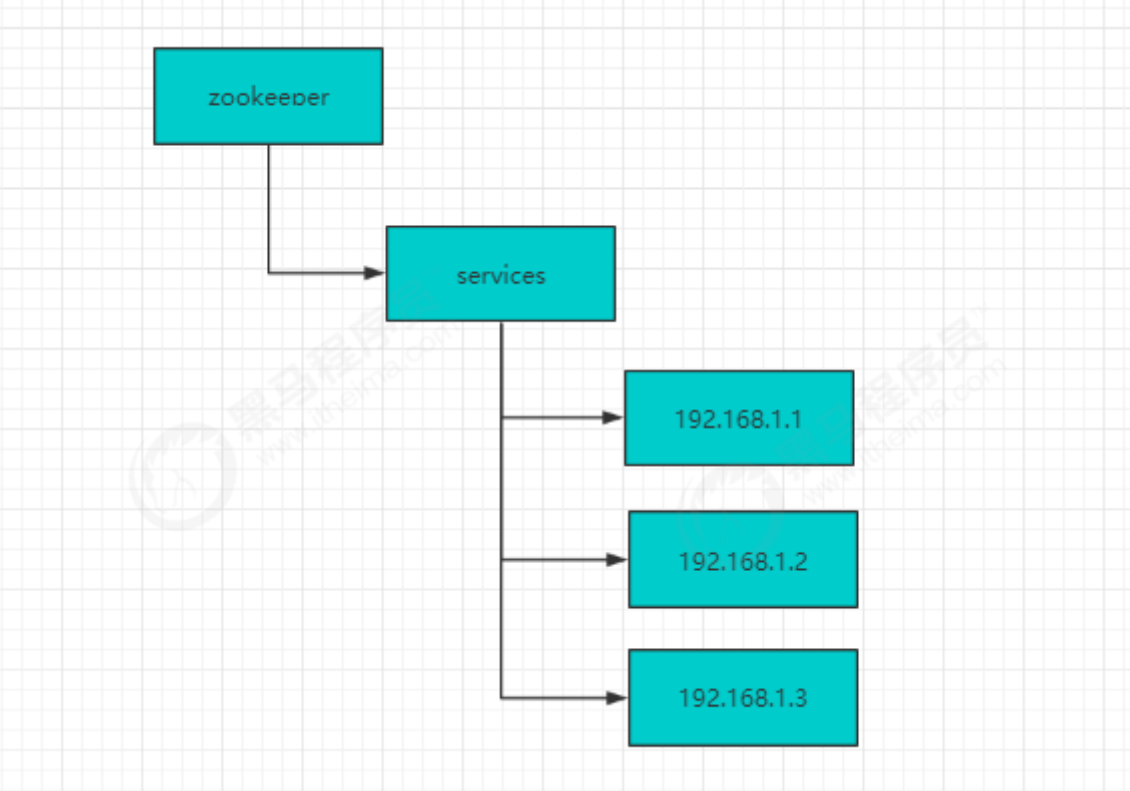
配置中心

心读取配置信息，进行初始化。传统的实现方式将配置存储在本地文件和内存中，一旦机器规模更大，配置变更频繁情况下，本地文件和内存方式的配置维护成本较高，使用zookeeper作为分布式的配置中心就可以解决这个问题。

我们将配置信息存在zk中的一个节点中，同时给该节点注册一个数据节点变更的watcher监听，一旦节点数据发生变更，所有的订阅该节点的客户端都可以获取数据变更通知。

负载均衡

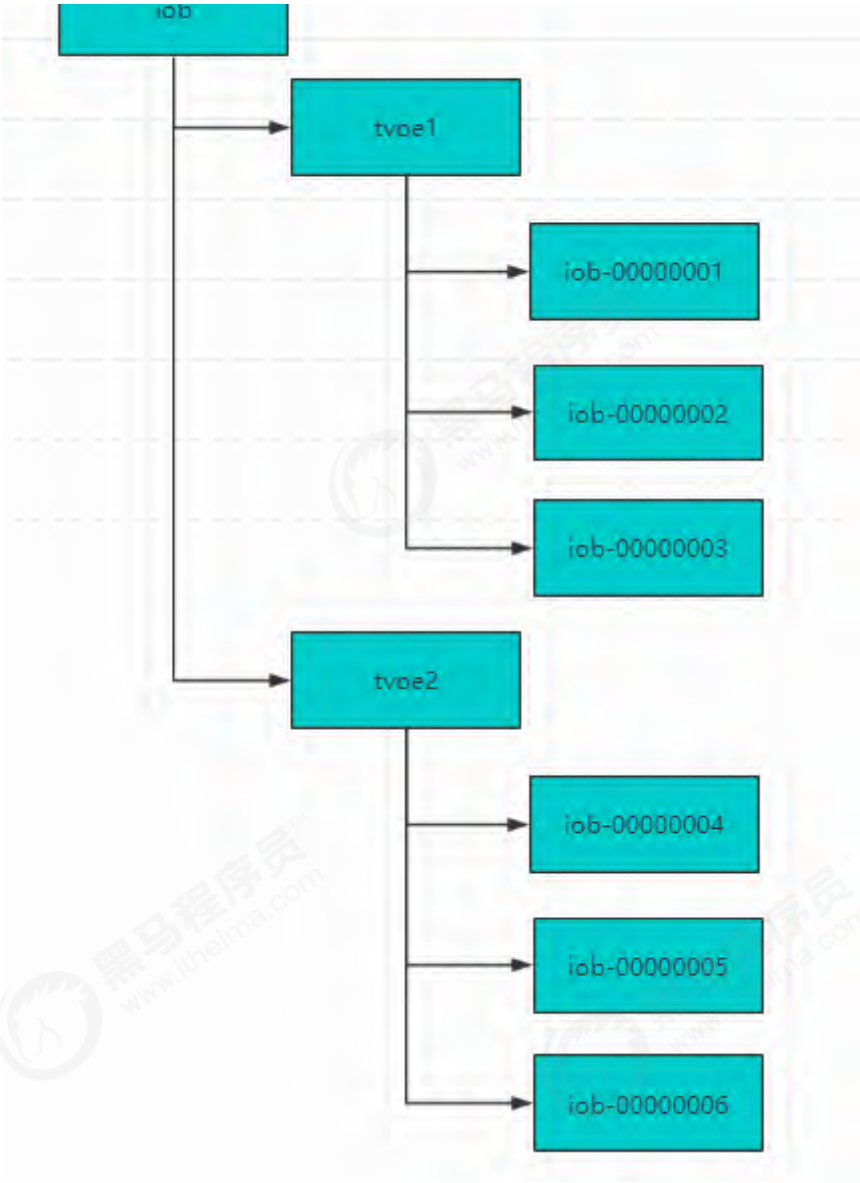
建立server节点，并建立监听器监视servers子节点的状态（用于在服务器增添时及时同步当前集群中服务器列表）。在每个服务器启动时，在servers节点下建立具体服务器地址的子节点,并在对应的字节点下存入服务器的相关信息。这样，我们在zookeeper服务器上可以获取当前集群中的服务器列表及相关信息，可以自定义一个负载均衡算法，在每个请求过来时从zookeeper服务器中获取当前集群服务器列表，根据算法选出其中一个服务器来处理请求。



命名服务

命名服务是分布式系统中的基本功能之一。被命名的实体通常可以是集群中的机器、提供的服务地址或者远程对象，这些都可以称作为名字。常见的就是一些分布式服务框架（RPC、RMI）中的服务地址列表，通过使用名称服务客户端可以获取资源的实体、服务地址和提供者信息。命名服务就是通过一个资源引用的方式来实现对资源的定位和使用。在分布式环境中，上层应用仅仅需要一个全局唯一名称，就像数据库中的主键。

在单库单表系统中可以通过自增ID来标识每一条记录，但是随着规模变大分库分表很常见，那么自增ID有仅能针对单一表生成ID，所以在这种情况下无法依靠这个来标识唯一ID。UUID就是一种全局唯一标识符。但是长度过长不易识别。

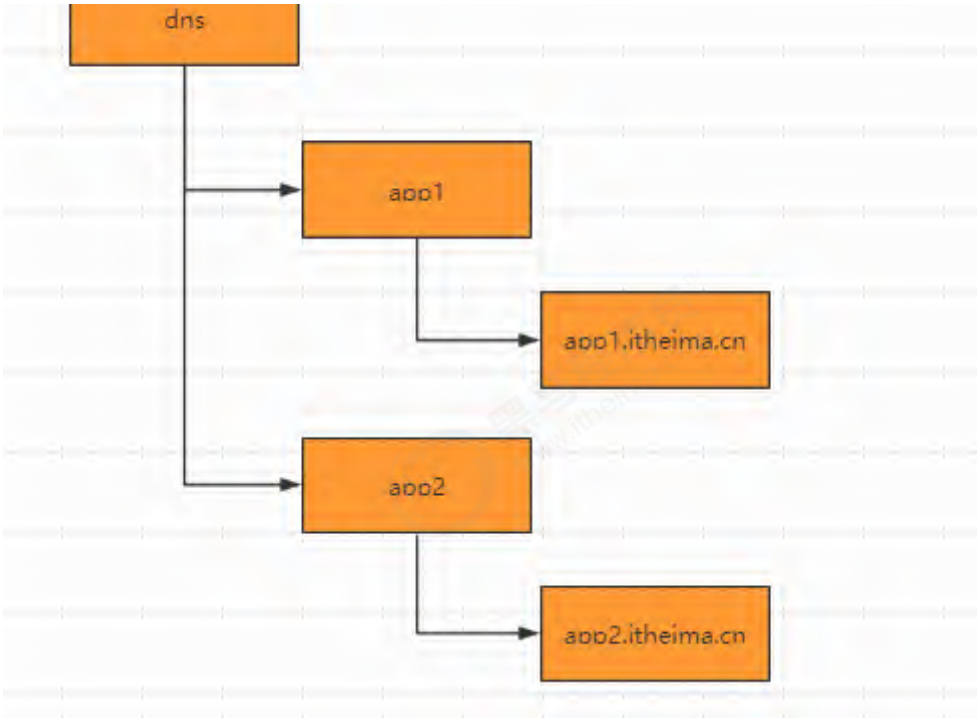


- 在Zookeeper中通过创建顺序节点就可以实现，所有客户端都会根据自己的任务类型来创建一个顺序节点，例如 job-00000001
- 节点创建完毕后，create()接口会返回一个完整的节点名，例如：job-00000002
  - 拼接type类型和完整节点名作为全局唯一的ID

DNS服务

- 域名配置

在分布式系统应用中，每一个应用都需要分配一个域名，日常开发中，往往使用本地HOST绑定域名解析，开发阶段可以随时修改域名和IP的映射，大大提高开发的调试效率。如果应用的机器规模达到一定程度后，需要频繁更新域名时，需要在规模的集群中变更，无法保证实时性。所有我们在zk上创建一个节点来进行域名配置



- 域名解析

应用解析时，首先从zk域名节点中获取域名映射的IP和端口。

- 域名变更

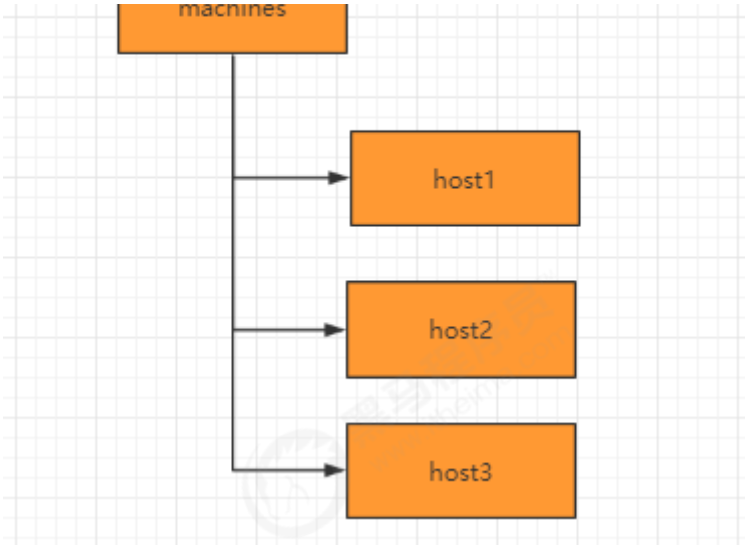
每个应用都会在在对应的域名节点注册一个数据变更的watcher监听，一旦监听的域名节点数据变更，zk会向所有订阅的客户端发送域名变更通知。

集群管理

随着分布式系统规模日益扩大，集群中机器的数量越来越多。有效的集群管理越来越重要了，zookeeper集群管理主要利用了watcher机制和创建临时节点来实现。以机器上下线和机器监控为例：

- 机器上下线

  新增机器的时候，将Agent部署到新增的机器上，当Agent部署启动时，会向zookeeper指定的节点下创建一个临时子节点，当Agent在zk上创建完这个临时节点后，当关注的节点zookeeper/machines下的子节点新加入新的节点时或删除都会发送通知，这样就对机器的上下线进行监控。



- 机器监控

  在机器运行过程中，Agent会定时将主机的的运行状态信息写入到/machines/hostn主机节点，监控中心通过订阅这些节点的数据变化来获取主机的运行信息。

分布式锁

- 数据库实现分布式锁

首先我们创建一张锁表，锁表中字段设置唯一约束

```
CREATE TABLE `lock_record` (  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT
'主键',  `lock_name` varchar(50) DEFAULT NULL COMMENT '锁名称',  PRIMARY KEY
(`id`),  UNIQUE KEY `lock_name` (`lock_name`)) ENGINE=InnoDB AUTO_INCREMENT=38
DEFAULT CHARSET=utf8
```

定义锁，实现Lock接口，tryLock()尝试获取锁，从锁表中查询指定的锁记 录，如果查询到记录，说明已经上锁，不能再上锁

```
 @Override    public  boolean tryLock() {        Example example = new
Example(LockRecord.class);
example.createCriteria().andEqualTo("lockName",LOCK_NAME);        LockRecord
lockRecord = lockRecordMapper.selectOneByExample(example);
if(lockRecord==null){            return true;        }        return false;    }
```

在lock方法获取锁之前先调用tryLock()方法尝试获取锁，如果未加锁则向锁表中插入一条锁记录来获取锁，这里我们通过循环，如果上锁我们一致等待锁的释放

```
    public void lock() {        while(true){            if(tryLock()){
     LockRecord lockRecord = new LockRecord();
lockRecord.setLockName(LOCK_NAME);
lockRecordMapper.insert(lockRecord);                return;            }else{
            System.out.println("等待锁......");            }        }    }
```

释放锁，即是将数据库中对应的锁表记录删除

```
example.createCriteria().andEqualTo("lockName",LOCK_NAME);
lockRecordMapper.deleteByExample(example);    }
```

注意在尝试获取锁的方法tryLock中，存在多个线程同时获取锁的情况，可以简单通过synchronized解决

- redis实现分布式锁

redis分布式锁的实现基于setnx（set if not exists），设置成功，返回1；设置失败，返回0，释放锁的操作通过del指令来完成

如果设置锁后在执行中间过程时，程序抛出异常，导致del指令没有调用，锁永远无法释放，这样就会陷入死锁。所以我们拿到锁之后会给锁加上一个过期时间，这样即使中间出现异常，过期时间到后会自动释放锁。

同时在setnx 和 expire 如果进程挂掉，expire不能执行也会死锁。所以要保证setnx和expire是一个原子性操作即可。redis 2.8之后推出了setnx和expire的组合指令

```
> set key value ex 5 nx
```

redis实现分布式锁注意的事项:

redis如何避免死锁

lock获取锁方法

```
        while(true){            //注意版本            Boolean f =
redisTemplate.opsForValue().setIfAbsent(LOCK_KEY_NAME,  LOCK_NAME, 10,
TimeUnit.SECONDS);            if(f){                return;            }else{
            System.out.println("等待锁......");            }        }
```

释放锁

```
 redisTemplate.delete(LOCK_KEY_NAME);
```

redis实现分布式锁存在的问题，为了解决redis单点问题，我们会部署redis集群，在 Sentinel 集群中，主节点突然挂掉了。同时主节点中有把锁还没有来得及同步到从节点。这样就会导致系统中同样一把锁被两个客户端同时持有，不安全性由此产生。redis官方为了解决这个问题，推出了Redlock 算法解决这个问题。但是带来的网络消耗较大。

分布式锁的redisson实现:

```
<dependency>    <groupId>org.redisson</groupId>
<artifactId>redisson</artifactId>    <version>3.6.5</version></dependency>
```

获取锁释放锁

```
Config config = new
Config();config.useSingleServer().setAddress("redis://127.0.0.1:6379").setDataba
se(0);Redisson redisson = (Redisson) Redisson.create(config);RLock mylock =
redisson.getLock(key); //获取锁mylock.lock();.....资源操作//释放锁mylock.unlock();
```

- zookeeper实现分布式锁

来监控节点的变化，从剩下的节点的找到最小的序列节点，获取分布式锁，执行相应处理，依次类推.....

原生实现

首先在ZkLock的构造方法中，连接zk,创建lock根节点

```
    //zk客户端    private ZooKeeper zk;    //zk是一个目录结构，root为最外层目录
private String root = "/locks";    //锁的名称    private String lockName;    //当
前线程创建的序列node    private ThreadLocal<String> nodeId = new ThreadLocal<>();
//用来同步等待zkclient链接到了服务端    private CountDownLatch connectedSignal = new
CountDownLatch(1);    private final static int sessionTimeout = 3000;    private
final static byte[] data= new byte[0];    public ZkLock(String config, String
lockName) {        this.lockName = lockName;        try {            zk = new
ZooKeeper(config, sessionTimeout, new Watcher() {                @Override
        public void process(WatchedEvent event) {                    // 建立连接
                if (event.getState() == Event.KeeperState.SyncConnected) {
                  connectedSignal.countDown();                    }
    }            });            connectedSignal.await();            Stat stat =
zk.exists(root, false);            if (null == stat) {                // 创建根节
点                zk.create(root, data, ZooDefs.Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT);            }        } catch (Exception e) {
throw new RuntimeException(e);        }    }
```

添加watch监听临时顺序节点的删除

```
class LockWatcher implements Watcher {    private CountDownLatch latch = null;
public LockWatcher(CountDownLatch latch) {        this.latch = latch;    }
@Override    public void process(WatchedEvent event) {        if
(event.getType() == Event.EventType.NodeDeleted)            latch.countDown();
}}
```

获取锁操作

```
            CreateMode.EPHEMERAL_SEQUENTIAL);
System.out.println(Thread.currentThread().getName()+myNode+ "created");
       // 取出所有子节点                List<String> subNodes =
zk.getChildren(root, false);                TreeSet<String> sortedNodes = new
TreeSet<>();                for(String node :subNodes) {
sortedNodes.add(root +"/" +node);                }                String
smallNode = sortedNodes.first();                String preNode =
sortedNodes.lower(myNode);                if (myNode.equals( smallNode)) {
             // 如果是最小的节点,则表示取得锁
System.out.println(Thread.currentThread().getName()+ myNode+"get lock");
           this.nodeId.set(myNode);                return;                }
               CountDownLatch latch = new CountDownLatch(1);                Stat
stat = zk.exists(preNode, new LockWatcher(latch));// 同时注册监听。
// 判断比自己小一个数的节点是否存在,如果不存在则无需等待锁,同时注册监听                if
(stat != null) {
System.out.println(Thread.currentThread().getName()+myNode+
       " waiting for " + root + "/" + preNode + " released lock");
     latch.await();// 等待，这里应该一直等待其他线程释放锁
nodeId.set(myNode);                latch = null;                }
} catch (Exception e) {                throw new RuntimeException(e);
}
```

释放锁

```
   public void unlock() {        try {
System.out.println(Thread.currentThread().getName()+ "unlock ");            if
(null != nodeId) {                zk.delete(nodeId.get(), -1);            }
       nodeId.remove();        } catch (InterruptedException e) {
e.printStackTrace();        } catch (KeeperException e) {
e.printStackTrace();        }    }
```

基于curator实现分布式锁:

maven依赖

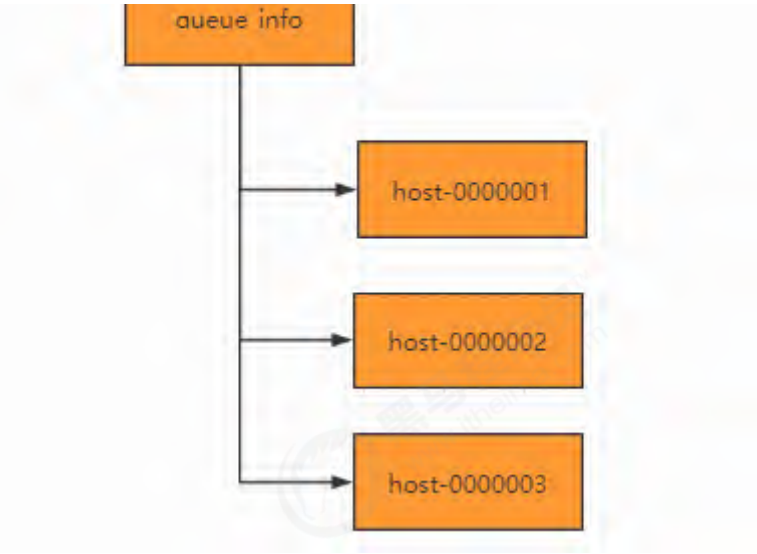
```
<dependency>    <groupId>org.apache.curator</groupId>    <artifactId>curator-
recipes</artifactId>    <version>4.0.0</version></dependency>
```

锁操作

```
    //创建zookeeper的客户端    RetryPolicy retryPolicy = new
ExponentialBackoffRetry(1000, 3);    //集群通过，分割    CuratorFramework client =
CuratorFrameworkFactory.newClient("127.0.0.1:2181", retryPolicy);
client.start();    //创建分布式锁，锁空间的根节点路径为/curator/lock
InterProcessMutex mutex = new InterProcessMutex(client, "/curator/lock");
mutex.acquire();    //获得了锁，进行业务流程    ......    //完成业务流程，释放锁
mutex.release();        //关闭客户端    client.close();
```

分布式队列

队列特性：FIFO（先入先出），zookeeper实现分布式队列的步骤：



- 在队列节点下创建临时顺序节点 例如/queue_info/192.168.1.1-0000001
- 调用getChildren()接口来获取/queue_info节点下所有子节点，获取队列中所有元素
- 比较自己节点是否是序号最小的节点，如果不是，则等待其他节点出队列，在序号最小的节点注册watcher
- 获取watcher通知后，重复步骤