

Zookeeper3.7源码剖析

能力目标

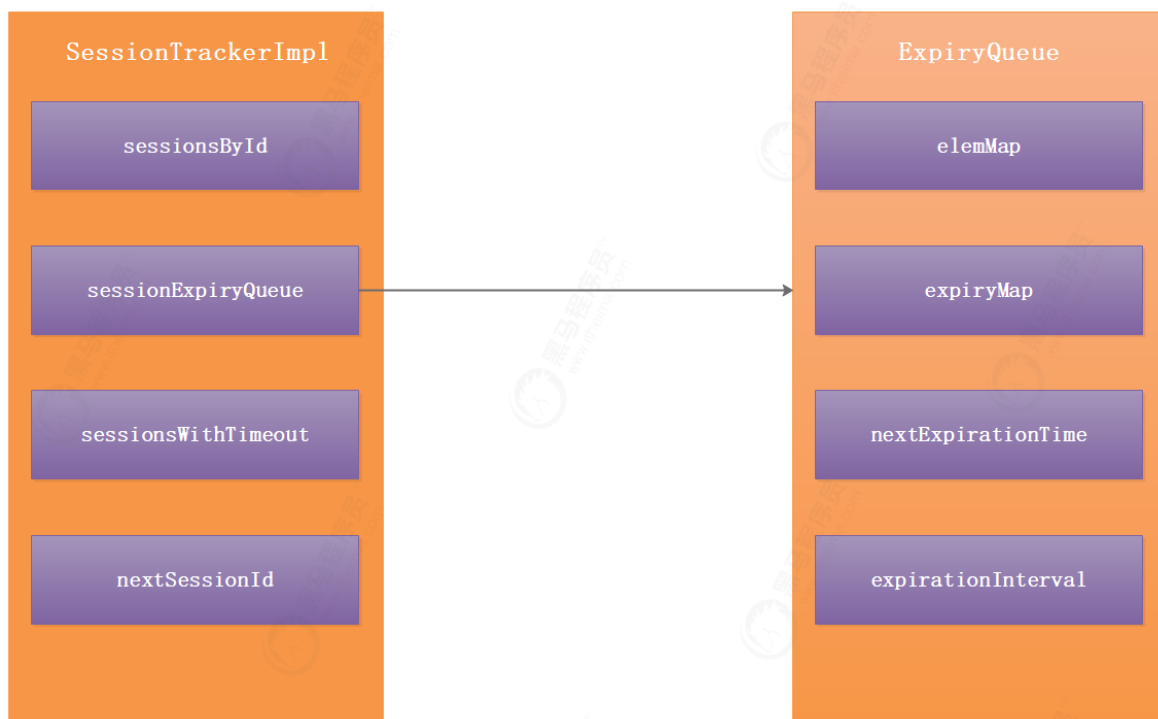
- 掌握Zookeeper中Session的管理机制
- 能基于Client进行Debug测试Session创建/刷新操作
- 能搭建Zookeeper集群源码配置
- 掌握集群环境下Leader选举启动过程
- 能说出Zookeeper选举过程中的概念
- 能说出Zookeeper选举投票规则
- 能画出Zookeeper集群数据同步流程

1 Session源码分析

客户端创建 socket 连接后，会尝试连接，如果连接成功成功会调用到 `primeConnection` 方法用来发送 `ConnectRequest` 连接请求，这里便是设置 session 会话，关于客户端创建会话我们就不在这里做讲解了，我们直接讲解服务端 Session 会话处理流程。

1.1 服务端Session属性分析

Zookeeper服务端会话操作如下图：



在服务端通过 `SessionTrackerImpl` 和 `ExpiryQueue` 来保存Session会话信息。

`SessionTrackerImpl` 有以下属性：

1:sessionsById 用来存储ConcurrentHashMap<Long, SessionImpl>
{sessionId:SessionImpl} 2:sessionExpiryQueue ExpiryQueue<SessionImpl>失效队列
3:sessionsWithTimeout ConcurrentMap<Long, Integer>存储的是{sessionId:
sessionTimeout}
4:nextSessionId 下一个sessionId

ExpiryQueue 失效队列有以下属性：

1:elemMap ConcurrentHashMap<E, Long> 存储的是{SessionImpl: newExpiryTime} Session
实例对象，失效时间。
2:expiryMap ConcurrentHashMap<Long, Set<E>>存储的是{time: set<SessionImpl>} 失效时
间，当前失效时间的Session对象集合。
3:nextExpirationTime 下一次失效时间 {(System.nanoTime() /
1000000)/expirationInterval+1}*expirationInterval 当前系统时间毫秒值
ms=System.nanoTime() / 1000000。 nextExpirationTime=当前系统时间毫秒值
+expirationInterval(失效间隔)。
4:expirationInterval 失效间隔，默认是10s，可以通过sessionlessCnxnTimeout修改。即是通过
配置文件的tickTime修改。

1.2 Session创建

我们接着上一章的案例继续分析，假如客户端发起请求后，后端如何识别是第一次创建请求？在之前的案例源码 NIOServerCnxn.readPayload() 中有所体现， NIOServerCnxn.readPayload() 部分关键源码如下：

```
if (incomingBuffer.remaining() == 0) { // have we read length bytes?  
    incomingBuffer.flip();  
    packetReceived( bytes: 4 + incomingBuffer.remaining());  
    //此时如果initialized=false，表示第一次连接 需要创建Session(createSession)  
    //调用readConnectRequest()后，在readConnectRequest()方法中会将initialized设置为true  
    if (!initialized) {  
        readConnectRequest();  
    } else {  
        readRequest();  
    }  
    lenBuffer.clear();  
    incomingBuffer = lenBuffer;  
}
```

此时如果 initialized=false，表示第一次连接 需要创建 Session(createSession)，此处调用 readConnectRequest()后，在readConnectRequest()方法中会将initialized设置为true，只有在处理完连接请求之后才会把initialized设置为true，才可以处理客户端其他命令。

```
private void readConnectRequest() throws IOException, InterruptedException, ClientCnxnLimitException {  
    if (!isZKServerRunning()) {  
        throw new IOException("ZooKeeperServer not running");  
    }  
    zkServer.processConnectRequest(cnxn: this, incomingBuffer);  
    initialized = true;  
}
```

上面方法还调用了 processConnectRequest 处理连接请求， processConnectRequest 第一次从请求中获取的 sessionId=0,此时会把创建Session作为一个业务，会调用 createSession() 方法， processConnectRequest 方法部分关键代码如下：

```
//sessionId=0, 所以需要创建Session
if (sessionId == 0) {
    //创建Session
    long id = createSession(cnxn, passwd, sessionTimeout); 创建Session, 第一次请求会作为一个业务提交
    LOG.debug(
        "Client attempting to establish new session: session = 0x{}", zxid = 0x{}, timeout = {}, address
        Long.toHexString(id),
        Long.toHexString(connReq.getLastZxidSeen()),
        connReq.getTimeout(),
        cnxn.getRemoteSocketAddress());
}
```

创建会话调用 `createSession()`，该方法会首先创建一个`sessionId`，并把该`sessionId`作为会话ID创建一个创建session会话的请求，并将该请求交给业务链作为一个业务处理，`createSession()` 源码如下：

```
/**
 * 创建Session
 */
long createSession(ServerCnxn cnxn, byte[] passwd, int timeout) {
    if (passwd == null) {
        // Possible since it's just deserialized from a packet on the wire.
        passwd = new byte[0];
    }
    //创建session
    long sessionId = sessionTracker.createSession(timeout); 创建SessionID
    Random r = new Random( seed: sessionId superSecret );
    r.nextBytes(passwd);
    ByteBuffer to = ByteBuffer.allocate(4);
    to.putInt(timeout);
    cnxn.setSessionId(sessionId); 根据创建的SessionID提交一个创建会话的业务
    //创建一个OpCode.createSession的请求（创建会话的业务）
    Request si = new Request(cnxn, sessionId, xid: 0, OpCode.createSession, to, authInfo: null);
    //提交业务
    submitRequest(si); 业务处理链和之前的一样
    return sessionId;
}
```

上面方法用到的 `sessionTracker.createSession(timeout)` 做了2个操作分别是创建`sessionId`和配置`sessionId`的跟踪信息，方法源码如下：

```
/**
 * 创建Session
 */
public long createSession(int sessionTimeout) {
    //获取下一个SessionID
    long sessionId = nextSessionId.getAndIncrement();
    //Session跟踪配置
    trackSession(sessionId, sessionTimeout);
    return sessionId;
}
```

会话信息的跟踪其实就是将会话信息添加到队列中，任何地方可以根据会话ID找到会话信息，`trackSession` 方法实现了Session创建、Session队列存储、Session过期队列存储，`trackSession` 方法源码如下：

```

/**
 * Session跟踪配置
 */
@Override
public synchronized boolean trackSession(long id, int sessionTimeout) {
    boolean added = false;

    //获取一个Session, 如果为空, 则以SessionID创建一个Session
    SessionImpl session = sessionsById.get(id);
    if (session == null) {
        session = new SessionImpl(id, sessionTimeout);
    }

    //session存入到sessionsById中, 可以根据ID获取到Session
    SessionImpl existedSession = sessionsById.putIfAbsent(id, session);

    if (existedSession != null) {
        session = existedSession;
    } else {
        added = true;
    }

    //将Session添加到失效队列中
    updateSessionExpiry(session, sessionTimeout);
    return added;
}

```

创建Session

添加到队列中

添加到过期队列中

在 `PreRequestProcessor` 的 `run` 方法中调用 `pRequest2Txn` , 关键代码如下 :

```

//create/close session don't require request record
case OpCode.createSession:
case OpCode.closeSession:
    if (!request.isLocalSession()) {
        pRequest2Txn(request.type, zks.getNextZxid(), request, record: null, deserialize: true);
    }
    break;

```

```

case OpCode.createSession:
    request.request.rewind();
    int to = request.request.getInt();
    request.setTxn(new CreateSessionTxn(to));
    request.request.rewind();
    // only add the global session tracker but not to ZKDb
    zks.sessionTracker.trackSession(request.sessionId, to);
    zks.setOwner(request.sessionId, request.getOwner());
    break;

```

在 `SyncRequestProcessor` 对 `txn`(创建 `session` 的操作)进行持久化, 在 `FinalRequestProcessor` 会对 `Session` 进行提交, 其实就是把 `Session` 的 `ID` 和 `Timeout` 存到 `sessionsWithTimeout` 中去。

由于 `FinalRequestProcessor` 中调用链路太复杂, 我们把调用链路写出来, 大家可以按照这个顺序跟踪 :

```

1: FinalRequestProcessor.applyRequest()
   方法代码: ProcessTxnResult rc = zks.processTxn(request);

2: ZooKeeperServer.processTxn(org.apache.zookeeper.server.Request)
   方法代码: processTxnForSessionEvents(request, hdr, request.getTxn());

```

上面调用链路中 `processTxnForSessionEvents(request, hdr, request.getTxn());` 方法代码如下：

```
private void processTxnForSessionEvents(Request request, TxnHeader hdr, Record txn) {
    int opCode = (request == null) ? hdr.getType() : request.type;
    long sessionId = (request == null) ? hdr.getClientId() : request.sessionId;

    //OpCode.createSession业务
    if (opCode == OpCode.createSession) {
        if (hdr != null && txn instanceof CreateSessionTxn) {
            CreateSessionTxn cst = (CreateSessionTxn) txn;
            //将sessionId、Timeout提交到sessionsWithTimeout中
            sessionTracker.commitSession(sessionId, cst.getTimeOut());
        } else if (request == null || !request.isLocalSession()) {
            LOG.warn("*****>>>> Got {} {}", txn.getClass(), txn.toString());
        }
    } else if (opCode == OpCode.closeSession) {
        sessionTracker.removeSession(sessionId);
    }
}
```

上面方法主要处理了 `OpCode.createSession` 并且将 `sessionId`、`Timeout` 提交到 `sessionsWithTimeout` 中，而提交到 `sessionsWithTimeout` 的方法 `SessionTrackerImpl.commitSession()` 代码如下：

```
public synchronized boolean commitSession(long id, int sessionTimeout) {
    return sessionsWithTimeout.put(id, sessionTimeout) == null;
}
```

1.3 Session刷新

服务端无论接受什么请求命令(增删或ping等请求)都会更新Session的过期时间。我们做增删或者ping命令的时候，都会经过 `RequestThrottler`，`RequestThrottler` 的 `run` 方法中调用 `zks.submitRequestNow()`，而 `zks.submitRequestNow(request)` 中调用了 `touch(si.cnxn)`；该方法源码如下：

```
void touch(ServerCnxn cnxn) throws MissingSessionException {
    if (cnxn == null) {
        return;
    }
    long id = cnxn.getSessionId();
    int to = cnxn.getSessionTimeout();
    if (!sessionTracker.touchSession(id, to)) {
        //更新会话过期时间
        throw new MissingSessionException("No session with sessionid 0x"
            + Long.toHexString(id)
            + " exists, probably expired and removed");
    }
}
```

`touchSession()` 方法更新 `sessionExpiryQueue` 失效队列中的失效时间，源码如下：

```

public synchronized boolean touchSession(long sessionId, int timeout) {
    //根据sessionId获取Session
    SessionImpl s = sessionsById.get(sessionId);

    if (s == null) {
        logTraceTouchInvalidSession(sessionId, timeout);
        return false;
    }

    if (s.isClosing()) {
        logTraceTouchClosingSession(sessionId, timeout);
        return false;
    }

    //更新session过期时间
    updateSessionExpiry(s, timeout);
    return true;
}

```

更新过期时间

update() 方法会在当前时间的基础上增加timeout，并更新失效时间为newExpiryTime，关键源码如下：

```

public Long update(E elem, int timeout) {
    Long prevExpiryTime = elemMap.get(elem);
    long now = Time.currentElapsedTime();
    //当前时间增加了timeout时间
    Long newExpiryTime = roundToNextInterval(time: now + timeout);

    if (newExpiryTime.equals(prevExpiryTime)) {
        // No change, so nothing to update
        return null;
    }

    // First add the elem to the new expiry time bucket in expiryMap.
    Set<E> set = expiryMap.get(newExpiryTime);
}

```

1.4 Session过期

SessionTrackerImpl 是一个线程类，继承了 zookeeperCriticalThread，我们可以看它的run方法，它首先获取了下一个会话过期时间，并休眠等待会话过期时间到期，然后获取过期的客户端会话集合并循环关闭，源码如下：

```

@Override
public void run() {
    try {
        while (running) {
            //获取下一个失效时间 waitTime
            long waitTime = sessionExpiryQueue.getWaitTime();
            if (waitTime > 0) {
                //休眠waitTime
                Thread.sleep(waitTime);
                continue;
            }

            //获取失效的客户端会话集合
            for (SessionImpl s : sessionExpiryQueue.poll()) {
                ServerMetrics.getMetrics().STALE_SESSIONS_EXPIRED.add(1);
                //关闭会话
                setSessionClosing(s.sessionId);
                //让客户端会话失效
                expirer.expire(s);
            }
        }
    }
}

```

获取下一个过期时间
并休眠等待时间到来

获取过期的客户端会话集合
并循环关闭会话

上面方法中调用了 `sessionExpiryQueue.poll()`，该方法代码主要是获取过期时间对应的客户端会话集合，源码如下：

```

public Set<E> poll() {
    long now = Time.currentElapsedTime();
    long expirationTime = nextExpirationTime.get();
    if (now < expirationTime) {
        return Collections.emptySet();
    }

    Set<E> set = null;
    long newExpirationTime = expirationTime + expirationInterval;
    if (nextExpirationTime.compareAndSet(expirationTime, newExpirationTime)) {
        //获取失效时间的客户端实现类的集合，这就是expiryMap集合类的意义。
        set = expiryMap.remove(expirationTime);
    }

    if (set == null) {
        return Collections.emptySet();
    }

    return set;
}

```

上面的 `setSessionClosing()` 方法其实是把Session会话的 `isClosing` 状态设置为了true,方法源码如下：

```

public synchronized void setSessionClosing(long sessionId) {
    if (LOG.isTraceEnabled()) {
        LOG.trace("Session closing: 0x{}", Long.toHexString(sessionId));
    }

    SessionImpl s = sessionsById.get(sessionId);
    if (s == null) {
        return;
    }

    s.isClosing = true;
}

```

而让客户端失效的方法 `expier.expire(s)`; 其实也是一个业务操作，主要调用了 `zooKeeperServer.expire()` 方法，而该方法获取 `SessionId` 后，又创建了一个 `OpCode.closeSession` 的请求，并交给业务链处理，我们查看 `zooKeeperServer.expire()` 方法源码如下：

```
/**
 * Session失效
 * @param session
 */
public void expire(Session session) {
    long sessionId = session.getSessionId();
    close(sessionId);
}

/**
 * 关闭Session业务
 */
private void close(long sessionId) {
    //创建一个OpCode.closeSession业务请求
    Request si = new Request(chxn: null, sessionId, xid: 0, OpCode.closeSession, bb: null, authInfo: null);
    //交给业务链处理
    submitRequest(si);
}
```

在 `PrepRequestProcessor.pRequest2Txn()` 方法中 `OpCode.closeSession` 操作里最后部分代理明确将会话 `Session` 的 `isClosing` 设置为了 `true`，源码如下：

```

case OpCode.closeSession:
    // We don't want to do this check since the session expiration thread
    // queues up this operation without being the session owner.
    // this request is the last of the session so it should be ok
    //zks.sessionTracker.checkSession(request.sessionId, request.getOwner());
    long startTime = Time.currentElapsedTime();
    synchronized (zks.outstandingChanges) {
        // need to move getEphemerals into zks.outstandingChanges
        // synchronized block, otherwise there will be a race
        // condition with the on flying deleteNode txn, and we'll
        // delete the node again here, which is not correct
        Set<String> es = zks.getZKDatabase().getEphemerals(request.sessionId);
        for (ChangeRecord c : zks.outstandingChanges) {
            if (c.stat == null) {
                // Doing a delete
                es.remove(c.path);
            } else if (c.stat.getEphemeralOwner() == request.sessionId) {
                es.add(c.path);
            }
        }
        for (String path2Delete : es) {
            if (digestEnabled) {
                parentPath = getParentPathAndValidate(path2Delete);
                parentRecord = getRecordForPath(parentPath);
                parentRecord = parentRecord.duplicate(request.getHdr().getZxid());
                parentRecord.stat.setPzxid(request.getHdr().getZxid());
                parentRecord.precalculatedDigest = precalculateDigest(
                    DigestOpCode.UPDATE, parentPath, parentRecord.data, parentRecord.stat);
                addChangeRecord(parentRecord);
            }
            nodeRecord = new ChangeRecord(
                request.getHdr().getZxid(), path2Delete, stat: null, childCount: 0, acl: null);
            nodeRecord.precalculatedDigest = precalculateDigest(
                DigestOpCode.REMOVE, path2Delete);
            addChangeRecord(nodeRecord);
        }
        if (ZooKeeperServer.isCloseSessionTxnEnabled()) {
            request.setTxn(new CloseSessionTxn(new ArrayList<String>(es)));
        }
        zks.sessionTracker.setSessionClosing(request.sessionId);
    }
    ServerMetrics.getMetrics().CLOSE_SESSION_PREP_TIME.add(Time.currentElapsedTime() - startTime);
    break;

```

业务链处理对象 `FinalRequestProcessor.processRequest()` 方法调用了

`zooKeeperServer.processTxn()`，并且在 `processTxn()` 方法中执行了

`processTxnForSessionEvents`，而 `processTxnForSessionEvents()` 方法正好移除了会话信息，方法源码如下：

```
private void processTxnForSessionEvents(Request request, TxnHeader hdr, Record txn) {
    int opCode = (request == null) ? hdr.getType() : request.type;
    long sessionId = (request == null) ? hdr.getClientId() : request.sessionId;

    //OpCode.createSession业务
    if (opCode == OpCode.createSession) {
        if (hdr != null && txn instanceof CreateSessionTxn) {
            CreateSessionTxn cst = (CreateSessionTxn) txn;
            //将sessionId、TimeOut提交到sessionsWithTimeout中
            sessionTracker.commitSession(sessionId, cst.getTimeOut());
        } else if (request == null || !request.isLocalSession()) {
            LOG.warn("*****>>>> Got {} {}", txn.getClass(), txn.toString());
        }
    } else if (opCode == OpCode.closeSession) {
        sessionTracker.removeSession(sessionId); 移除对应会话
    }
}
```

移除会话的方法 `SessionTrackerImpl.removeSession()` 会移除会话ID以及过期会话对象，源码如下：

```
public synchronized void removeSession(long sessionId) {
    LOG.debug("Removing session 0x{}", Long.toHexString(sessionId));
    SessionImpl s = sessionsById.remove(sessionId);
    sessionsWithTimeout.remove(sessionId); 会话ID移除
    if (LOG.isTraceEnabled()) {
        ZooTrace.logTraceMessage(
            LOG,
            ZooTrace.SESSION_TRACE_MASK,
            msg: "SessionTrackerImpl --- Removing session 0x" + Long.toHexString(sessionId));
    }
    if (s != null) {
        sessionExpiryQueue.remove(s); 超时会话移除
    }
}
```

1.5 Zookeeper会话测试

为了让Zookeeper的会话理解更深刻，我们对会话流程做一个测试，首先测试会话创建，再测试会话刷新。

1)会话创建测试

我们打开 `NIOserverCnxn.readPayload()` 方法，跟踪首次创建会话，调试情况如下：

```
if (incomingBuffer.remaining() == 0) { // have we read length bytes?
    incomingBuffer.flip();
    packetReceived( bytes: 4 + incomingBuffer.remaining()); incomingBuffer: "java.nio.HeapByteBuffer[p
    //此时如果initialized=false，表示第一次连接 需要创建Session(createSession)
    //调用readConnectRequest()后，在readConnectRequest()方法中会将initialized设置为true
    if (!initialized) { initialized: false
        System.out.println("1: 会话未连接，准备首次连接会话.....");
        readConnectRequest();
    } else {
        System.out.println("1-1: 会话已连接，非首次连接会话.....");
        readRequest();
    }
}
```

此时会建立远程连接并创建SessionID，我们调试到NIOServerCnxn.readConnectRequest()方法，此时建立链接，并且得到的sessionId=0。

```
@SuppressWarnings(value = "IS2_INCONSISTENT_SYNC", justification = "the value won't change after startup")
public void processConnectRequest(ServerCnxn cnxn, ByteBuffer incomingBuffer) throws IOException, ClientCnxnLimitException {

    BinaryInputArchive bia = BinaryInputArchive.getArchive(new ByteBufferInputStream(incomingBuffer));
    System.out.println("2: 建立远程连接.....");
    ConnectRequest connReq = new ConnectRequest();
    connReq.deserialize(bia, tag: "connect");
    LOG.debug(
        "Session establishment request from client {} client's lastZxid is 0x{}",
        cnxn.getRemoteSocketAddress(), cnxn: "ip: 127.0.0.1/127.0.0.1:24465 sessionId: 0x0"
        Long.toHexString(connReq.getLastZxidSeen()));
    //第1次链接，sessionId=0
    long sessionId = connReq.getSessionId();
    System.out.println("2: 第1次连接的sessionId="+sessionId);

    ZooKeeperMain.main(new String[]{});
    Variables vars = new Variables();
    vars.put("this", this);
    vars.put("cnxn", cnxn);
    vars.put("incomingBuffer", incomingBuffer);
    vars.put("bia", bia);
    vars.put("connReq", connReq);
}
```

当sessionId=0时，会执行Session创建，Session创建会调用SessionTrackerImpl.createSession()方法实现会话创建，并将会话存入跟踪队列，DEBUG测试如下：

```
//sessionId=0，所以需要创建Session
if (sessionId == 0) {
    //创建Session
    long id = createSession(cnxn, passwd, sessionTimeout);
    System.out.println("3:sessionId=0, 此时创建sessionId="+id);
    LOG.debug(
        "Client attempting to establish new session: session = 0x{}, zxid = 0x{}, timeout = {}, and address = {}",
        Long.toHexString(id),
        Long.toHexString(connReq.getLastZxidSeen()),
        connReq.getTimeOut(),
        cnxn.getRemoteSocketAddress());
}
```

会话创建代码如下：

```
public long createSession(int sessionTimeout) {
    //获取下一个SessionID
    long sessionId = nextSessionId.getAndIncrement();
    //Session跟踪配置
    trackSession(sessionId, sessionTimeout);
    System.out.println("使用SessionTrackerImpl创建会话，并将会话加入跟踪队列中");
    return sessionId;
}
```

跟踪测试后，控制台输出如下信息：

AcceptThread-----链接服务的IP: 127.0.0.1

1: 会话未连接, 准备首次连接会话.....

2: 建立远程连接.....

2: 第1次连接的sessionId=0

使用SessionTrackerImpl创建会话, 并将会话加入跟踪队列中

3: sessionId=0, 此时创建sessionId=72061099907219458

2) 会话刷新测试

我们执行 `get /zookeeper` 指令, 然后首先跟踪到 `RequestThrottler.run()` 方法, 执行如下:

```
final long elapsedTime = Time.currentElapsedTime() - request.requestThrottleQueueTime; elapsedTime: 0
ServerMetrics.getMetrics().REQUEST_THROTTLE_QUEUE_TIME.add(elapsedTime);
if (shouldThrottleOp(request, elapsedTime)) { elapsedTime: 0
    request.setIsThrottled(true); request: "sessionId:0x100034029430000 type:getData cxid:0x2 zxid:0xffffffff
    ServerMetrics.getMetrics().THROTTLED_OPS.add(1);
}
System.out.println("a. 当前请求并未过期, 不需要删除, 准备刷新会话");
zks.submitRequestNow(request);
```

执行程序到达 `zookeeperServer.touch()`, 即将开始准备刷新会话了, 我们测试效果如下:

```
void touch(ServerCnxn cnxn) throws MissingSessionException { cnxn: "ip: 127.0.0.1/127.0.0.1:25408 sessionId: 0x1
    if (cnxn == null) {
        return;
    }
    long id = cnxn.getSessionId(); id: 72061168142974976
    int to = cnxn.getSessionTimeout(); to: 30000 cnxn: "ip: 127.0.0.1/127.0.0.1:25408 sessionId: 0x100034029430000
    //更新会话
    System.out.println("b. 准备调用SessionTrackerImpl.touchSession()刷新会话");
    if (!sessionTracker.touchSession(id, to)) { sessionTracker: "Session Sets (2): (1): \r\n0 expire at Mon Dec 3
        throw new MissingSessionException("No session with sessionId 0x"
            + Long.toHexString(id)
            + " exists, probably expired and removed");
    }
}
```

调用 `SessionTrackerImpl.touchSession()` 的时候会先判断会话是否为空、会话是否已经关闭, 如果都没有, 才执行刷新会话操作, DEBUG跟踪如下:

```

public synchronized boolean touchSession(long sessionId, int timeout) {
    //根据sessionId获取Session
    SessionImpl s = sessionsById.get(sessionId);
    if (s == null) {
        logTraceTouchInvalidSession(sessionId, timeout);
        return false;
    }

    if (s.isClosing()) {
        logTraceTouchClosingSession(sessionId, timeout);
        return false;
    }

    //更新session过期时间
    System.out.println("c. 会话不为空，会话也未关闭，准备调用updateSessionExpiry()刷新会话");
    updateSessionExpiry(s, timeout);
    return true;
}

```

刷新会话其实就是会话时间增加，增加会话时间DEBUG跟踪如下：

```

public Long update(E elem, int timeout) {
    Long prevExpiryTime = elemMap.get(elem);
    long now = Time.currentElapsedTime();
    //当前时间增加了timeout时间
    Long newExpiryTime = roundToNextInterval(time:now + timeout);
    System.out.println("d. 剩余过期时间: "+now+", 增加过期时间: "+timeout+", 刷新会话后过期时间: "+newExpiryTime);

    if (newExpiryTime.equals(prevExpiryTime)) {
        // No change, so nothing to update
        return null;
    }
}

```

测试后效果如下：

- a. 当前请求并未过期，不需要删除，准备刷新会话
- b. 准备调用SessionTrackerImpl.touchSession()刷新会话
- c. 会话不为空，会话也未关闭，准备调用updateSessionExpiry()刷新会话
- d. 剩余过期时间：54572178, 增加过期时间：30000, 刷新会话后过期时间：54604000

2 Zookeeper集群启动流程

我们先搭建Zookeeper集群，再来分析选举算法。

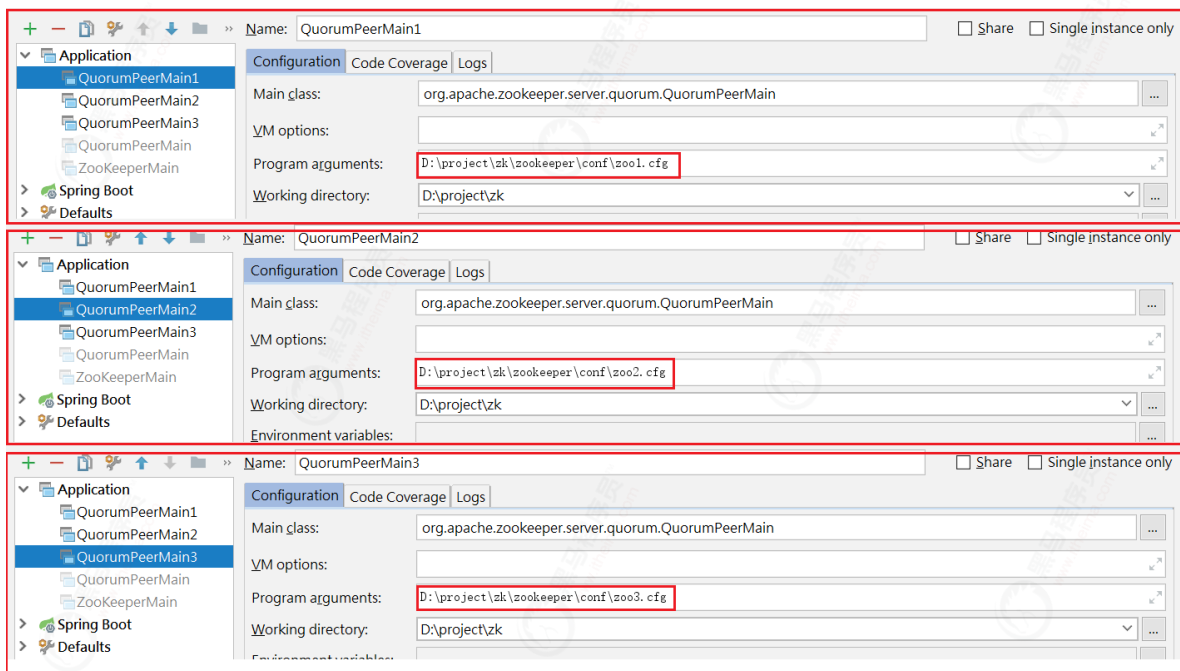
2.1 Zookeeper集群配置



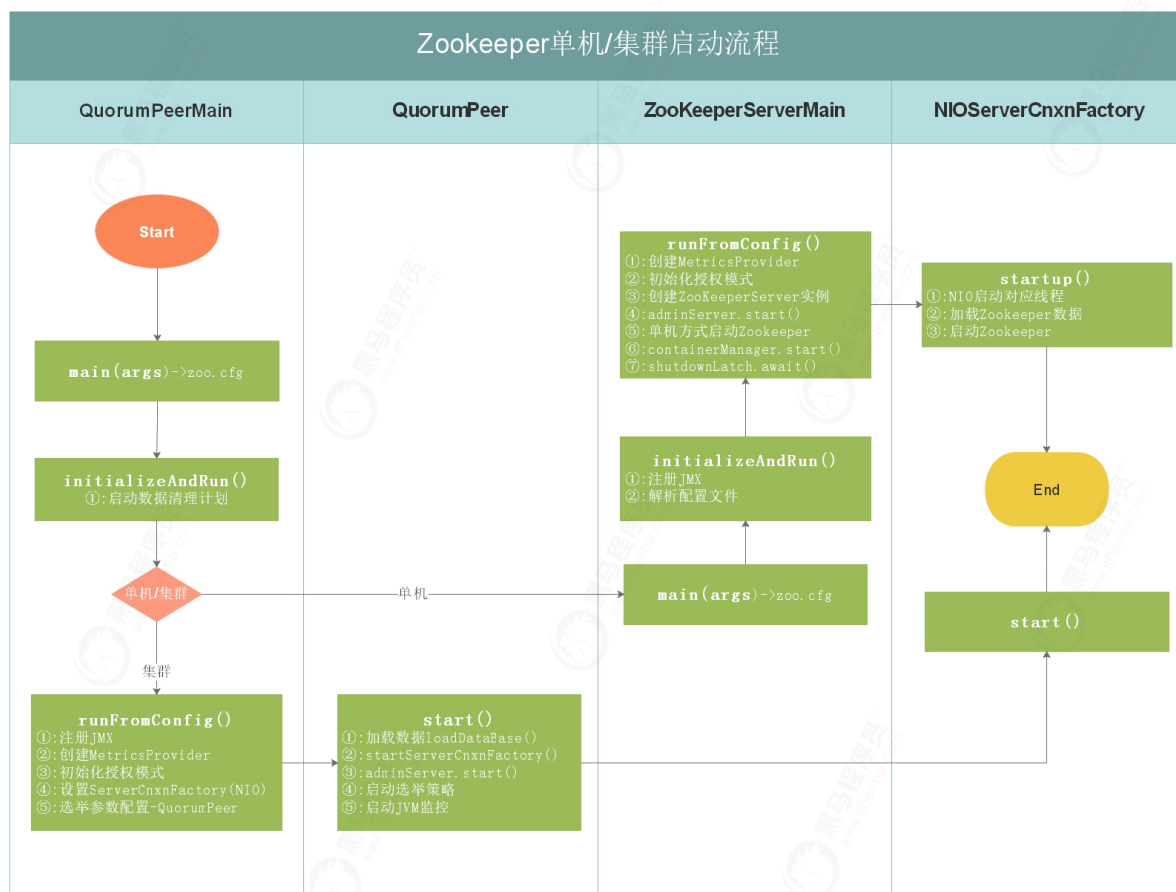
如上图：

- 1:创建zoo1.cfg、zoo2.cfg、zoo3.cfg
- 2:创建zkdata1、zkdata2、zkdata3
- 3:创建3个myid，值分别为1、2、3

配置3个启动类，如下图：

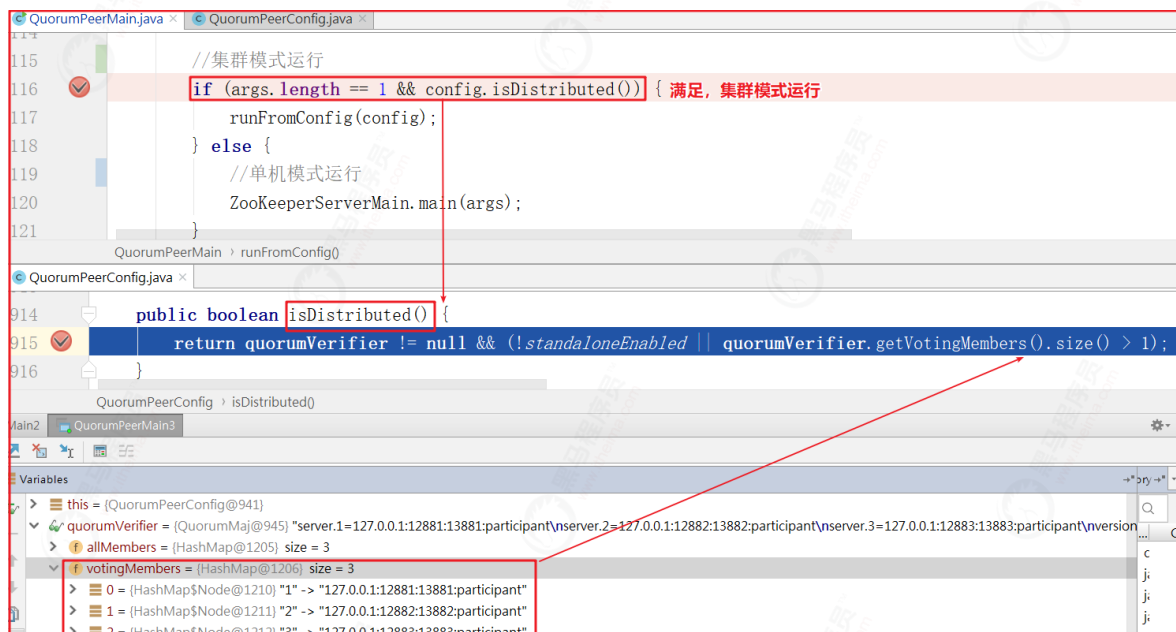


2.2 集群启动流程分析

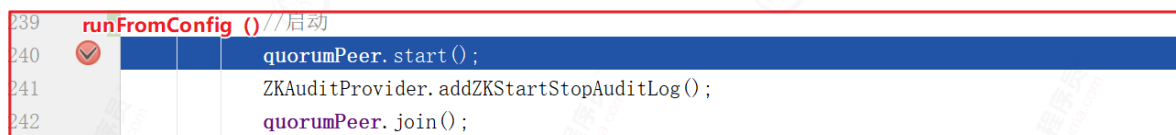


如上图，上图是Zookeeper单机/集群启动流程，每个细节所做的事情都在上图有说明，我们接下来按照流程图对源码进行分析。

程序启动，运行流程启动集群模式，如下图：



quorumPeer.start() 启动服务，如下代码：



quorumPeer.start() 方法代码如下：

```
public synchronized void start() {
    if (!getView().containsKey(myid)) {
        throw new RuntimeException("My id " + myid + " not in the peer list"); myid: 3
    }
    //加载数据库数据
    loadDataBase();
    //启动网络通信
    startServerCnxnFactory();
    try {
        //启动AdminServer
        adminServer.start(); adminServer: JettyAdminServer@1983
    } catch (AdminServerException e) {
        LOG.warn("Problem starting AdminServer", e);
        System.out.println(e);
    }
    //启动选举策略
    startLeaderElection();
    //启动JVM监听
    startJvmPauseMonitor();
    //启动线程
    super.start();
}
```

quorumPeer.start() 方法启动的主要步骤：

- 1:loadDataBase()加载数据。
- 2:startServerCnxnFactory 用来开启acceptThread、selectorThread和workerPool线程池。
- 3:开启Leader选举startLeaderElection。
- 4:开启JVM监控线程startJvmPauseMonitor。
- 5:调用父类super.start();进行Leader选举。

startLeaderElection() 开启Leader选举方法做了2件事，首先创建初始化选票选自己，接着创建选举投票方式，源码如下：

```
public synchronized void startLeaderElection() {
    try {
        //创建当前选票，启动的时候先投票投给自己
        if (getPeerState() == ServerState.LOOKING) {
            currentVote = new Vote(myid, getLastLoggedZxid(), getCurrentEpoch());
        }
    } catch (IOException e) {
        RuntimeException re = new RuntimeException(e.getMessage());
        re.setStackTrace(e.getStackTrace());
        throw re;
    }
    //创建选举方式createElectionAlgorithm
    this.electionAlg = createElectionAlgorithm(electionType);
}
```

createElectionAlgorithm() 创建选举算法只有第3种，其他2种均已废弃，方法源码如下：

```

protected Election createElectionAlgorithm(int electionAlgorithm) { electionAlgorithm: 3
    Election le = null;

    //TODO: use a factory rather than a switch
    switch (electionAlgorithm) {
    case 1:
        throw new UnsupportedOperationException("Election Algorithm 1 is not supported.");
    case 2:
        throw new UnsupportedOperationException("Election Algorithm 2 is not supported.");
    case 3: 选举方式只有electionAlgorithm=3, 其他的均已废弃
        QuorumCnxManager qcm = createCnxnManager();
        QuorumCnxManager oldQcm = qcmRef.getAndSet(qcm);
        if (oldQcm != null) {
            LOG.warn("Clobbering already-set QuorumCnxManager (restarting leader election?");
            oldQcm.halt();
        }
        QuorumCnxManager.Listener listener = qcm.listener;
        if (listener != null) {
            listener.start();
            FastLeaderElection fle = new FastLeaderElection(self, this, qcm);
            fle.start();
            le = fle;
        } else {
            LOG.error("Null listener when initializing cnx manager");
        }
        break;
    default:
        assert false;
    }
    return le;
}

```

这个方法创建了以下三个对象：

- ①、创建QuorumCnxManager对象
- ②、QuorumCnxManager.Listener
- ③、FastLeaderElection

3 Zookeeper集群Leader选举

3.1 Paxos算法介绍

Zookeeper选举主要依赖于FastLeaderElection算法，其他算法均已淘汰，但FastLeaderElection算法又是典型的Paxos算法，所以我们要先学习下Paxos算法，这样更有助于掌握FastLeaderElection算法。

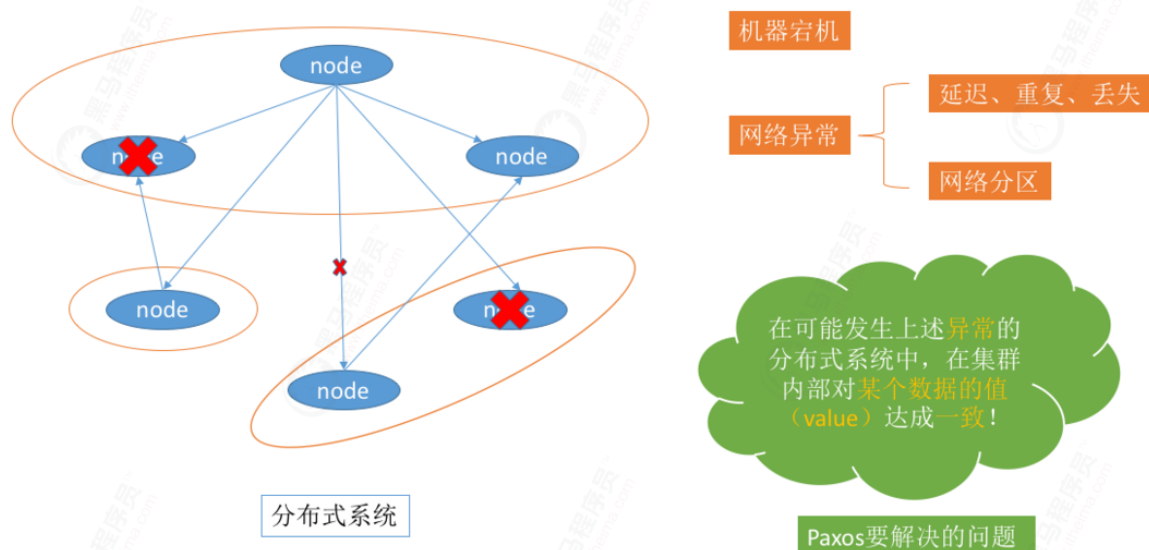
1) Paxos介绍

分布式事务中常见的事务模型有2PC和3PC，无论是2PC提交还是3PC提交都无法彻底解决分布式的一致性问题以及无法解决太过保守及容错性不好。Google Chubby的作者Mike Burrows说过，世上只有一种一致性算法，那就是Paxos，所有其他一致性算法都是Paxos算法的不完整版。Paxos算法是公认的晦涩，很难讲清楚，但是工程上也很难实现，所以有很多Paxos算法的工程实现，如Chubby，Raft，ZAB，微信的PhxPaxos等。这一篇会介绍这个公认为难于理解但是行之有效的Paxos算法。Paxos算法是莱斯利·兰伯特(Leslie Lamport)1990年提出的一种基于消息传递的一致性算法，它曾就此发表了《The Part-Time Parliament》，《Paxos Made Simple》，由于采用故事的方式来解释此算法，感觉还是很难理解。

2)Paxos算法背景 Paxos算法是基于消息传递且具有高度容错特性的一致性算法，是目前公认的解决分布式一致性问题最有效的算法之一，其解决的问题就是在分布式系统中如何就某个值（决议）达成一致。面试的时候：不要把这个Paxos算法达到的目的和分布式事务联系起来，而是针对Zookeeper这样的master-slave集群对某个决议达成一致，也就是副本之间写或者leader选举达成一致。我觉得这个算法和狭义的分布式事务是不一样的。在常见的分布式系统中，总会发生诸如机器宕机或网络异常（包括消息的延迟、丢失、重复、乱序，还有网络分区）(也就是会发生异常的分布式系统)等情况。Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中，快速且正确地在集群内部对某个数据的值达成一致。也可以理解成分布式系统中达成状态的一致性。

3)Paxos算法理解

Paxos 算法是分布式一致性算法用来解决一个分布式系统如何就某个值(决议)达成一致的问题。一个典型的场景是，在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。分布式系统中一般是通过多副本来保证可靠性，而多个副本之间会存在数据不一致的情况。所以必须有一个一致性算法来保证数据的一致，描述如下：假如在分布式系统中初始是各个节点的数据是一致的，每个节点都顺序执行系列操作，然后每个节点最终的数据还是一致的。Paxos算法就是解决这种分布式场景中的一致性问题。对于一般的开发人员来说，只需要知道paxos是一个分布式选举算法即可。多个节点之间存在两种通讯模型：共享内存（Shared memory）、消息传递（Messages passing），Paxos是基于消息传递的通讯模型的。



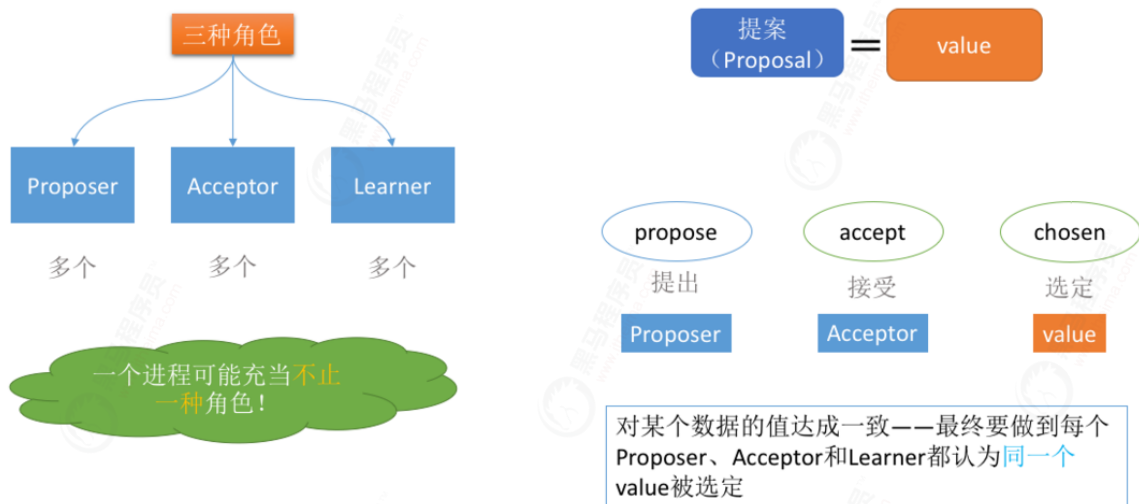
4)Paxos相关概念

在Paxos算法中，有三种角色：

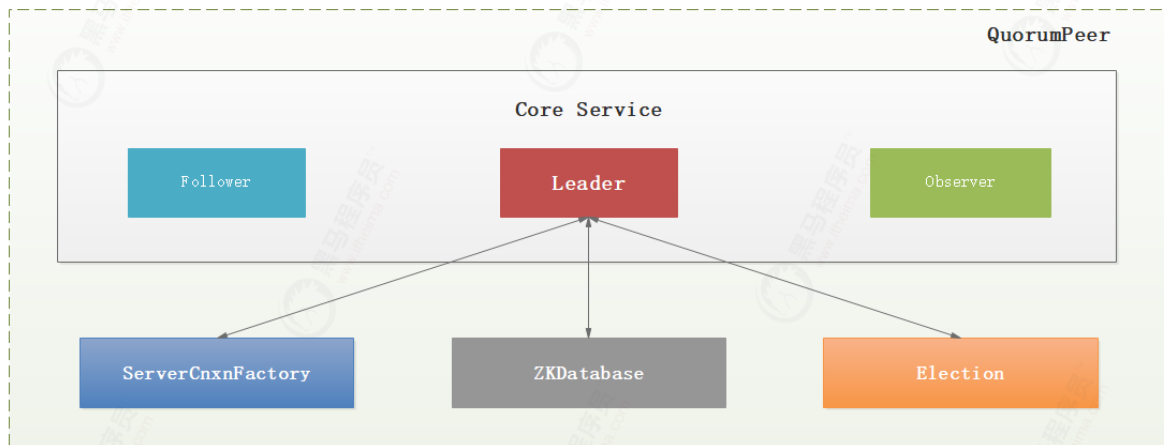
- Proposer
- Acceptor
- Learners

在具体的实现中，一个进程可能同时充当多种角色。比如一个进程可能既是Proposer又是Acceptor又是Learner。Proposer负责提出提案，Acceptor负责对提案作出裁决（accept与否），learner负责学习提案结果。还有一个很重要的概念叫提案（Proposal）。最终要达成一致的value就在提案里。只要Proposer发的提案被Acceptor接受（半数以上的Acceptor同意才行），Proposer就认为该提案里的

value被选定了。Acceptor告诉Learner哪个value被选定，Learner就认为那个value被选定。只要Acceptor接受了某个提案，Acceptor就认为该提案里的value被选定了。为了避免单点故障，会有一个Acceptor集合，Proposer向Acceptor集合发送提案，Acceptor集合中的每个成员都有可能同意该提案且每个Acceptor只能批准一个提案，只有当一半以上的成员同意了一个提案，就认为该提案被选定了。



3.2 QuorumPeer工作流程

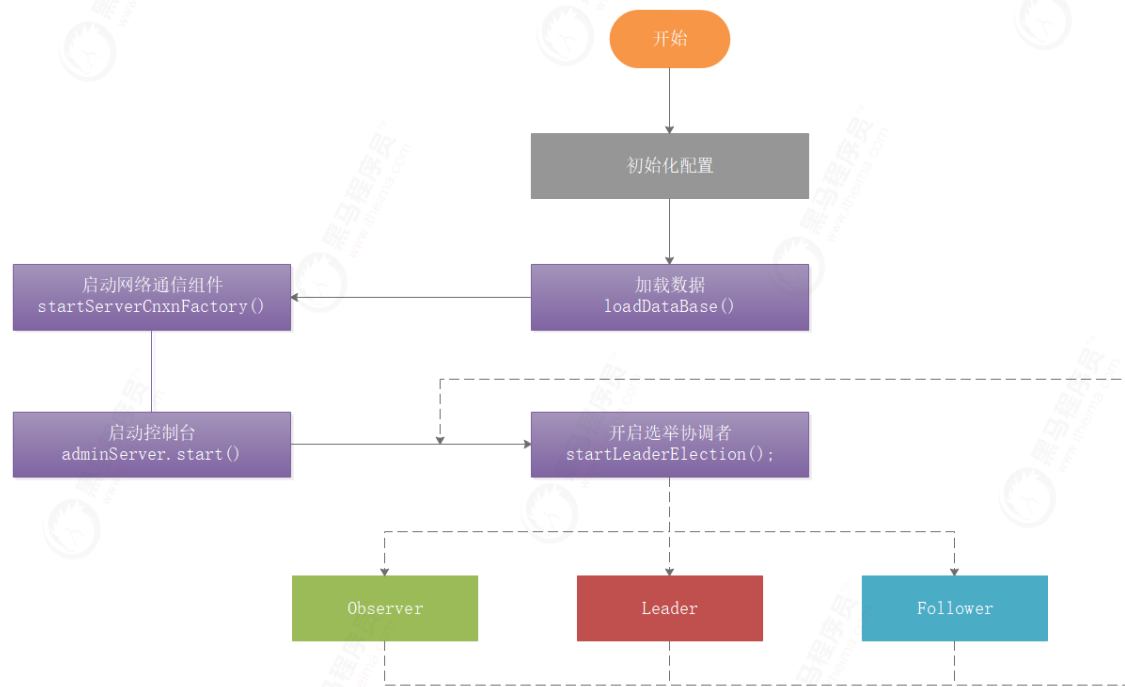


QuorumCnxManager：每台服务器在启动的过程中，会启动一个 QuorumPeer，负责各台服务器之间的底层Leader选举过程中的网络通信对应的类就是 QuorumCnxManager。

zookeeper 对于每个节点 QuorumPeer 的设计相当的灵活，QuorumPeer 主要包括四个组件：客户端请求接收器(ServerCnxnFactory)、数据引擎(ZKDatabase)、选举器(Election)、核心功能组件(Leader/Follower/Observer)。

- 1: ServerCnxnFactory负责维护与客户端的连接(接收客户端的请求并发送相应的响应); (1001行)
- 2: ZKDatabase负责存储/加载/查找数据(基于目录树结构的KV+操作日志+客户端Session); (129行)
- 3: Election负责选举集群的一个Leader节点; (998行)
- 4: Leader/Follower/Observer确认是QuorumPeer节点应该完成的核心职责; (1270行)

QuorumPeer 工作流程比较复杂，如下图：



QuorumPeer 工作流程：

- 1: 初始化配置
- 2: 加载当前存在的数据
- 3: 启动网络通信组件
- 4: 启动控制台
- 5: 开启选举协调者，并执行选举（这个过程是会持续，并不是一次操作就结束了）

3.3 QuorumCnxManager 源码分析

QuorumCnxManager 内部维护了一系列的队列，用来保存接收到的、待发送的消息以及消息的发送器，除接收队列以外，其他队列都按照SID分组形成队列集合，如一个集群中除了自身还有3台机器，那么就会为这3台机器分别创建一个发送队列，互不干扰。

```
public QuorumCnxManager(QuorumPeer self, final long mySid, Map<Long, QuorumPeer.QuorumServer> view,
    QuorumAuthServer authServer, QuorumAuthLearner authLearner, int socketTimeout, boolean listenOnAllIPs,
    int quorumCnxnThreadsSize, boolean quorumSaslAuthEnabled) {
    //消息接收队列，用于存放那些其他服务器接收到的消息。
    this.recvQueue = new CircularBlockingQueue<>(RECV_CAPACITY);
    //消息发送队列，用于保存那些待发送的消息，按照SID进行分组。
    this.queueSendMap = new ConcurrentHashMap<>();
    //发送器集合，每个SenderWorker消息发送器，都对应一台远程Zookeeper服务器，否则消息的发送，也按照SID进行分组。
    this.senderWorkerMap = new ConcurrentHashMap<>();
    //最近发送过的消息，为每个SID保留最近发送过的一个消息。
    this.lastMessageSent = new ConcurrentHashMap<>();

    String cnxToValue = System.getProperty("zookeeper.cnxTimeout");
    if (cnxToValue != null) {
        this.cnxTO = Integer.parseInt(cnxToValue);
    }
}
```

QuorumCnxManager.Listener：为了能够相互投票，Zookeeper集群中的所有机器都需要建立起网络连接。QuorumCnxManager在启动时会创建一个ServerSocket来监听Leader选举的通信端口。开启监听后，Zookeeper能够不断地接收到来自其他服务器地创建连接请求，在接收到其他服务器地TCP连接请求时，会进行处理。为了避免两台机器之间重复地创建TCP连接，Zookeeper只允许SID大的服务器主动和其他机器建立连接，否则断开连接。在接收到创建连接请求后，服务器通过对比自己和远程服务器的SID值来判断是否接收连接请求，如果当前服务器发现自己的SID更大，那么会断开当前连接，然后自己主动和远程服务器将连接（自己作为“客户端”）。一旦连接建立，就会根据远程服务器的SID来创建相应的消息发送器SendWorker和消息发送器RecvWorker，并启动。

QuorumCnxManager.Listener 监听启动可以查看 QuorumCnxManager.Listener 的 run 方法，源代码如下，可以断点调试看到此时监听的正是我们所说的投票端口：

```
@Override
public void run() {
    if (!shutdown) {
        LOG.debug("Listener thread started, myId: {}", self.getId());
        Set<InetSocketAddress> addresses; addresses: size = 1

        if (self.getQuorumListenOnAllIPs()) {
            addresses = self.getElectionAddress().getWildcardAddresses();
        } else {
            addresses = self.getElectionAddress().getAllAddresses();
        }

        CountDownLatch latch = new CountDownLatch(addresses.size()); latch: "java.util.concurrent.CountDownLatch@22e23
        listenerHandlers = addresses.stream().map(address -> listenerHandlers: size = 1

QuorumCnxManager$Listener.run()
QuorumPeerMain2
QuorumPeerMain3
Variables
latch = {CountDownLatch@2519} java.util.concurrent.CountDownLatch@22e230[Count = 1]
self = {QuorumPeer@1978} "Thread[QuorumPeer[myid=3](plain=[0:0:0:0:0:0:0:0:2183](secure=disabled),5,main]"
listenerHandlers = {ArrayList@2382} size = 1
0 = {QuorumCnxManager$Listener$ListenerHandler@2393}
serverSocket = null
address = {InetSocketAddress@2394} "/127.0.0.1:13883"
```

上面是监听器，各个服务之间进行通信我们需要开启 ListenerHandler 线程，在

QuorumCnxManager.Listener.ListenerHandler 的run方法中有一个方法 acceptConnections() 调用，该方法就是用于接受每次选举投票的信息，如果只有一个节点或者没有投票信息的时候，此时方法会阻塞，一旦执行选举，程序会往下执行，我们可以先启动1台服务，再启动第2台、第3台，此时会收到有客户端参与投票链接，程序会往下执行，源码如下：

```

private void acceptConnections() {
    int numRetries = 0;
    Socket client = null;

    while ((!shutdown) && (portBindMaxRetry == 0 || numRetries < portBindMaxRetry)) {
        try {
            //选举开启套接字
            serverSocket = createNewServerSocket();
            LOG.info("{} is accepting connections now, my election bind port: {}", QuorumCnxManager.this.mySid, address);
            while (!shutdown) {
                try {
                    //阻塞等待客户端链接
                    client = serverSocket.accept();
                    setSockOpts(client);
                    LOG.info("Received connection request from {}", client.getRemoteSocketAddress());
                    // Receive and handle the connection request
                    // asynchronously if the quorum sasl authentication is
                    // enabled. This is required because sasl server
                    // authentication process may take few seconds to finish,
                    // this may delay next peer connection requests.
                    if (quorumSaslAuthEnabled) {
                        receiveConnectionAsync(client);
                    } else {
                        receiveConnection(client);
                        System.out.println("有客户端链接, 参与投票---->" + client.getPort());
                    }
                }
            }
            numRetries = 0;
        }
    }
}

```

我们启动2台服务，效果如下：

有客户端链接，参与投票--->9915

启动两台服务后能收到其他服务被作为客户端链接参与选举

上面虽然能证明投票访问了当前监听的端口，但怎么知道是哪台服务呢？我们可以沿着 `receiveConnection()` 源码继续研究，源码如下：

```

public void receiveConnection(final Socket sock) {
    DataInputStream din = null;
    try {
        din = new DataInputStream(new BufferedInputStream(sock.getInputStream()));

        LOG.debug("Sync handling of connection request received from: {}", sock.getRemoteSocketAddress());
        handleConnection(sock, din);
    } catch (IOException e) {
        LOG.error("Exception handling connection, addr: {}, closing server connection", sock.getRemoteSocketAddress(), e);
        LOG.debug("Exception details: ", e);
        closeSocket(sock);
    }
}

```

`receiveConnection()` 方法只是获取了数据流，并没做特殊处理，并且调用了 `handleConnection()` 方法，该方法源码如下：

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

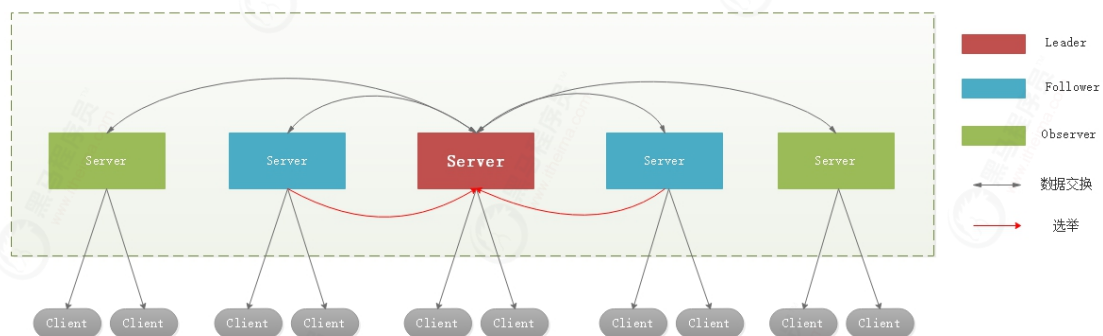
    try {
        protocolVersion = din.readLong();
        if (protocolVersion >= 0) { // this is a server id and not a protocol version
            sid = protocolVersion;
        } else {
            try {
                //获取sid, 也就是myid的值
                InitialMessage init = InitialMessage.parse(protocolVersion, din);
                sid = init.sid;
                if (!init.electionAddr.isEmpty()) {
                    electionAddr = new MultipleAddresses(init.electionAddr,
                        Duration.ofMillis(self.getMultiAddressReachabilityCheckTimeoutMs()));
                }
                LOG.debug("Initial message parsed by {}: {}", self.getId(), init.toString());
            } catch (InitialMessage.InitialMessageException ex) {
                LOG.error("Initial message parsing error!", ex);
                closeSocket(sock);
                return;
            }
        }
    }

    //输出SID
    System.out.println("参与投票的MyID="+sid);
}
```

通过网络连接获取数据sid，获取sid表示是哪一台连过来的，我们可以打印输出sid，测试输出如下数据：

```
参与投票的MyID=2
参与投票的MyID=3
```

3.4 FastLeaderElection算法源码分析



在 zookeeper 集群中，主要分为三者角色，而每一个节点同时只能扮演一种角色，这三种角色分别是：

- (1) **Leader** 接受所有Follower的提案请求并统一协调发起提案的投票，负责与所有的Follower进行内部的数据交换(同步);
- (2) **Follower** 直接为客户端提供服务并参与提案的投票，同时与 **Leader** 进行数据交换(同步);
- (3) **observer** 直接为客户端服务但并不参与提案的投票，同时也与 **Leader** 进行数据交换(同步);

`FastLeaderElection` 选举算法是标准的 `Fast Paxos` 算法实现，可解决 `LeaderElection` 选举算法收敛速度慢的问题。

创建 `FastLeaderElection` 只需要 `new FastLeaderElection()` 即可，如下代码：

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

创建 `FastLeaderElection` 会调用 `starter()` 方法，该方法会创建 `sendqueue`、`recvqueue` 队列、`Messenger` 对象，其中 `Messenger` 对象的作用非常关键，方法源码如下：

```
private void starter(QuorumPeer self, QuorumCnxManager manager) {  
    this.self = self;  
    proposedLeader = -1;  
    proposedZxid = -1;  
    //创建sendqueue、recvqueue队列。Messenger对象  
    sendqueue = new LinkedBlockingQueue<ToSend>();  
    recvqueue = new LinkedBlockingQueue<Notification>();  
    this.messenger = new Messenger(manager);  
}
```

创建 `Messenger` 的时候，会创建 `workerSender` 并封装成 `wsThread` 线程，创建 `workerReceiver` 并封装成 `wrThread` 线程，看名字就很容易理解，`wsThread` 用于发送数据，`wrThread` 用于接收数据，`Messenger` 创建源码如下：

```
Messenger(QuorumCnxManager manager) {  
  
    this.ws = new WorkerSender(manager);  
  
    this.wsThread = new Thread(this.ws, name: "WorkerSender[myid=" + self.getId() + "]");  
    this.wsThread.setDaemon(true);  
  
    this.wr = new WorkerReceiver(manager);  
  
    this.wrThread = new Thread(this.wr, name: "WorkerReceiver[myid=" + self.getId() + "]");  
    this.wrThread.setDaemon(true);  
}
```

创建完 `FastLeaderElection` 后接着会调用它的 `start()` 方法启动选举算法，代码如下：

```
if (listener != null) {  
    listener.start();  
    FastLeaderElection fle = new FastLeaderElection(self: this, qcm);  
    fle.start(); 启动选举算法  
    le = fle;  
} else {  
    LOG.error("Null listener when initializing cnx manager");  
}
```

启动选举算法会调用 `start()` 方法，`start()` 方法如下：

```
public void start() {
    this.messenger.start();
}
```

上面会执行 `messenger.start()`，也就是如下方法，也就意味着 `wsThread` 和 `wrThread` 线程都将启动，源码如下：

```
void start() {
    this.wsThread.start();
    this.wrThread.start();
}
```

`wsThread` 由 `workerSender` 封装而来，此时会调用 `workerSender` 的 `run` 方法，`run` 方法会调用 `process()` 方法，源码如下：

```
public void run() {
    while (!stop) {
        try {
            ToSend m = sendqueue.poll(timeout: 3000, TimeUnit.MILLISECONDS);
            if (m == null) {
                continue;
            }
            process(m);
        } catch (InterruptedException e) {
            break;
        }
    }
    LOG.info("WorkerSender is down");
}
```

`process` 方法调用了 `manager` 的 `toSend` 方法，此时是把对应的 `sid` 作为消息发送出去，这里其实是发送投票信息，源码如下：

```
void process(ToSend m) {
    ByteBuffer requestBuffer = buildMsg(m.state.ordinal(), m.leader, m.zxid,
        m.electionEpoch, m.peerEpoch, m.configData);
    manager.toSend(m.sid, requestBuffer);
}
```

投票可以投自己，也可以投别人，如果是选票选自己，只需要把投票信息添加到 `recvQueue` 中即可，源码如下：

```

public void toSend(Long sid, ByteBuffer b) {
    /*
     * If sending message to myself, then simply enqueue it (loopback).
     */
    if (this.mySid == sid) {
        b.position(0);
        //自己发送给自己的选票
        addToRecvQueue(new Message(b.duplicate(), sid));
        /*
         * Otherwise send to the corresponding thread to send.
         */
    } else {
        /*
         * 发送投票信息给其他节点
         */
        BlockingQueue<ByteBuffer> bq = queueSendMap.computeIfAbsent(sid, serverId -> new CircularBlockingQueue<>(SEND_CAPACITY));
        addToSendQueue(bq, b);
        connectOne(sid);
    }
}
}

```

在 `workerReceiver.run` 方法中会从 `recvQueue` 中获取 `Message`，并把发送给其他服务的投票封装到 `sendqueue` 队列中，交给 `workerSender` 发送处理，源码如下：

```

public void run() {

    Message response;
    while (!stop) {
        // Sleeps on receive
        try {
            // 取出Message
            // 从recvQueue中获取Message对象
            response = manager.pollRecvQueue(timeout: 3000, TimeUnit.MILLISECONDS);
            if (response == null) {
                continue;
            }

            final int capacity = response.buffer.capacity();

            // The current protocol and two previous generations all send at least 28 bytes
            if (capacity < 28) {
                LOG.error("Got a short response from server {}: {}", response.sid, capacity);
                continue;
            }

            // this is the backwardCompatibility mode in place before ZK-107
            // It is for a version of the protocol in which we didn't send peer epoch
            // With peer epoch and version the message became 40 bytes
            boolean backCompatibility28 = (capacity == 28);

            // this is the backwardCompatibility mode for no version information
            boolean backCompatibility40 = (capacity == 40);

            response.buffer.clear();

            // Instantiate Notification and set its attributes
            Notification n = new Notification();

            int rstate = response.buffer.getInt();
            long rleader = response.buffer.getLong();
            long rxid = response.buffer.getLong();
            long relectionEpoch = response.buffer.getLong();
            long rpeerepoch;

            int version = 0x0;
            QuorumVerifier rqv = null;

            try {
                if (!backCompatibility28) {
                    rpeerepoch = response.buffer.getLong();
                    if (!backCompatibility40) {
                        /*
                         * Version added in 3.4.6
                         */
                        version = response.buffer.getInt();
                    } else {
                        LOG.info("Backward compatibility mode (36 bits), server id: {}", response.sid);
                    }
                } else {
                    LOG.info("Backward compatibility mode (28 bits), server id: {}", response.sid);
                    rpeerepoch = ZxidUtils.getEpochFromZxid(rxid);
                }
            }

            // check if we have a version that includes config. If so extract config info from message.
            if (version > 0x1) {
                int configLength = response.buffer.getInt();

                // we want to avoid errors caused by the allocation of a byte array with negative length
                // (causing NegativeArraySizeException) or huge length (causing e.g. OutOfMemoryError)
                if (configLength < 0 || configLength > capacity) {
                    throw new IOException(String.format("Invalid configLength in notification message! s

```

```

        response.sid, capacity, version, configLength));
    }

    byte[] b = new byte[configLength];
    response.buffer.get(b);

    synchronized (self) {
        try {
            rqv = self.configFromString(new String(b, UTF_8));
            QuorumVerifier curQV = self.getQuorumVerifier();
            if (rqv.getVersion() > curQV.getVersion()) {
                LOG.info("{} Received version: {} my version: {}",
                    self.getId(),
                    Long.toHexString(rqv.getVersion()),
                    Long.toHexString(self.getQuorumVerifier().getVersion()));
                if (self.getPeerState() == ServerState.LOOKING) {
                    LOG.debug("Invoking processReconfig(), state: {}", self.getServerState());
                    self.processReconfig(rqv, suggestedLeaderId: null, zxid: null, restartLE: false);
                    if (!rqv.equals(curQV)) {
                        LOG.info("restarting leader election");
                        self.shuttingDownLE = true;
                        self.getElectionAlg().shutdown();

                        break;
                    }
                } else {
                    LOG.debug("Skip processReconfig(), state: {}", self.getServerState());
                }
            }
        } catch (IOException | ConfigException e) {
            LOG.error("Something went wrong while processing config received from {}", response.sid);
        }
    }
} else {
    LOG.info("Backward compatibility mode (before reconfig), server id: {}", response.sid);
}
} catch (BufferUnderflowException | IOException e) {
    LOG.warn("Skipping the processing of a partial / malformed response message sent by sid={}",
        response.sid, capacity, e);
    continue;
}
/*
 * 发送给其他服务的投票封装成ToSend，并存入到sendqueue中
 */
if (!validVoter(response.sid)) {
    Vote current = self.getCurrentVote();
    QuorumVerifier qv = self.getQuorumVerifier();
    ToSend notmsg = new ToSend(
        ToSend.mType.notification,
        current.getId(),
        current.getZxid(),
        logicalclock.get(),
        self.getPeerState(),
        response.sid,
        current.getPeerEpoch(),
        qv.toString().getBytes(UTF_8));

    sendqueue.offer(notmsg);
}

```

封装ToSend

存入队列中

3.5 Zookeeper选举投票剖析

选举是个很复杂的过程，要考虑很多场景，而且选举过程中有很多概念需要理解。

3.5.1 选举概念

1)ZK服务状态：

```
public enum ServerState {  
    //代表没有当前集群中没有Leader,此时是投票选举状态  
    LOOKING,  
    //代表已经是伴随者状态  
    FOLLOWING,  
    //代表已经是领导者状态  
    LEADING,  
    //代表已经是观察者状态（观察者不参与投票过程）  
    OBSERVING  
}
```

2)服务角色:

```
//Learner 是随从服务和观察者的统称  
public enum LearnerType {  
    //随从者角色  
    PARTICIPANT,  
    //观察者角色  
    OBSERVER  
}
```

3)投票消息广播:

```
public static class Notification {  
    int version;  
  
    //被推荐leader的ID  
    long leader;  
  
    //被推荐leader的zxid  
    long zxid;  
  
    //投票轮次  
    long electionEpoch;  
  
    //当前投票者的服务状态（LOOKING）  
    QuorumPeer.ServerState state;  
    //当前投票者的ID  
    long sid;  
    //QuorumVerifier作为集群验证器，主要完成判断一组server在  
    //已给定的配置的server列表中，是否能够构成集群  
    QuorumVerifier qv;  
  
    //被推荐leader的投票轮次  
    long peerEpoch;
```

```
}
```

4)选票模型:

```
public class Vote {  
    //投票版本号, 作为一个标识  
    private final int version;  
    //当前服务的ID  
    private final long id;  
    //当前服务事务ID  
    private final long zxid;  
    //当前服务投票的轮次  
    private final long electionEpoch;  
    //被推举服务器的投票轮次  
    private final long peerEpoch;  
    //当前服务器所处的状态  
    private final ServerState state;  
}
```

5)消息发送对象:

```
public static class ToSend {  
    //支持的消息类型  
    enum mType {  
        crequest, //请求  
        challenge, //确认  
        notification, //通知  
        ack //确认回执  
    }  
  
    ToSend(mType type, long leader, long zxid, long electionEpoch, ServerState  
state, long sid, long peerEpoch, byte[] configData) {  
  
        this.leader = leader;  
        this.zxid = zxid;  
        this.electionEpoch = electionEpoch;  
        this.state = state;  
        this.sid = sid;  
        this.peerEpoch = peerEpoch;  
        this.configData = configData;  
    }  
  
    /*  
     * Proposed leader in the case of notification  
     * 被投票推举为leader的服务ID  
     */ long leader;  
  
    /*  
     * id contains the tag for acks, and zxid for notifications  
     */  
    /* long zxid;
```

```

/*
 * Epoch
 * 投票轮次
 */ long electionEpoch;

/*
 * Current state;
 * 服务状态
 */ QuorumPeer.ServerState state;

/*
 * Address of recipient
 * 消息接收方服务ID
 */ long sid;

/*
 * Used to send a QuorumVerifier (configuration info)
 */ byte[] configData = dummyData;

/*
 * Leader epoch
 */ long peerEpoch;
}

```

3.5.2 选举过程

QuorumPeer本身是个线程，在集群启动的时候会执行 `quorumPeer.start()`；此时会调用它重写的 `start()` 方法，最后会调用父类的 `start()` 方法，所以该线程会启动执行，因此会执行它的 `run` 方法，而 `run` 方法正是选举流程的入口，我们看 `run` 方法关键源码如下：

```

try {
    reconfigFlagClear();
    if (shuttingDownLE) {
        shuttingDownLE = false;
        startLeaderElection();
    }
}
// 选举操作 makeLEStrategy():获取选举算法    lookForLeader(): 选举过程
setCurrentVote(makeLEStrategy().lookForLeader());

```

所有节点初始状态都为 `LOOKING`，会进入到选举流程，选举流程首先要获取算法，获取算法的方法是 `makeLEStrategy()`，该方法返回的是 `FastLeaderElection` 实例，核心选举流程是 `FastLeaderElection` 中的 `lookForLeader()` 方法。

```

/****
 * 获取选举算法
 */
@SuppressWarnings("deprecation")
protected Election makeLEStrategy() {
    return electionAlg;
}

```

`lookForLeader()` 是选举过程的关键流程，源码分析如下：

```

public Vote lookForLeader() throws InterruptedException {
    try {
        self.jmxLeaderElectionBean = new LeaderElectionBean();
        MBeanRegistry.getInstance().register(self.jmxLeaderElectionBean, self.jmxLocalPeerBean);
    } catch (Exception e) {
        LOG.warn("Failed to register with JMX", e);
        self.jmxLeaderElectionBean = null;
    }

    self.start_fle = Time.currentElapsedTime();
    try {
        /*
         * 接收的投票票池
         */
        Map<Long, Vote> recvset = new HashMap<Long, Vote>();

        /*
         * 对外的投票记录
         */
        Map<Long, Vote> outofelection = new HashMap<Long, Vote>();

        int notTimeout = minNotificationInterval;

        synchronized (this) {
            //投票轮次自增
            logicalclock.incrementAndGet();
            //首次推举自己为leader
            updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
        }

        //广播发出投票
        sendNotifications();

        SyncedLearnerTracker voteSet;

        //如果当前server状态依然是LOOKING状态，且未选出leader则直到找到为止
        while ((self.getPeerState() == ServerState.LOOKING) && (!stop)) {
            /*
             * 准备接收发来的投票，该过程属于阻塞过程，直到本次阻塞超时，一次取一个
             */
            Notification n = recvqueue.poll(notTimeout, TimeUnit.MILLISECONDS);

            /*
             * 如果没有收到足够的投票，就继续发出广播进行投票，否则处理投票信息。
             */
            if (n == null) {
                //如果集群中有其他节点信息，就开始广播
                if (manager.haveDelivered()) {
                    //开始广播自己的投票信息
                    sendNotifications();
                } else {
                    //如果服务器不存在，尝试与每个服务器建立连接。
                    manager.connectAll();
                }
            }

            /*
             * 延长从队列获取选票时长
             */
            int tmpTimeOut = notTimeout * 2;
            notTimeout = Math.min(tmpTimeOut, maxNotificationInterval);
        } 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:
                    //如果当前选举人是LOOKING状态
                    if (getInitLastLoggedZxid() == -1) {
                        break;
                    }
                    if (n.zxid == -1) {
                        break;
                    }
                    // 如果收到服务器轮次的大于自己的，则将自己的轮次设置成最新的，将自己的投票池清空
                    if (n.electionEpoch > logicalclock.get()) {
                        logicalclock.set(n.electionEpoch);
                        recvset.clear();
                        //进行选票PK，如果自己的票没有PK过其他投递的票，则将自己的票变更为其
                        if (totalOrderPredicate(n.leader, n.zxid, n.peerEpoch, getInitId(), getInitLastLoggedZxid(), getPeerEpoch())) {
                            updateProposal(n.leader, n.zxid, n.peerEpoch);
                        } else {

```

```

        updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
    }
    //重新发出投票
    sendNotifications();
} else if (n.electionEpoch < logicalclock.get()) {
    //如果收到的轮次小于自己的轮次，不做处理，n的投票无效
    break;
} else if (totalOrderPredicate(n.leader, n.zxid, n.peerEpoch, proposedLeader, proposedZxid, proposedEpoch)) {
    //如果收到的轮次等于自己的轮次，则进行选票PK，如果自己的票没有PK过其他投送的票，则将自己的票变更为其他
    updateProposal(n.leader, n.zxid, n.peerEpoch);
    //重新发出投票
    sendNotifications();
}

//将获取到的投票数据放入自己的票池中
recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));

//统计选票
voteSet = getVoteTracker(recvset, new Vote(proposedLeader, proposedZxid, logicalclock.get(), proposedEpoch));

//判断是否有超过半数的票是指向同一个服务ID
if (voteSet.hasAllQuorums()) {
    //如果此刻在票池汇总还有未取出的投票，则和选举出的投票PK，如果取出的票优于当前推举的投票，则重新投票
    while ((n = recvqueue.poll(finalizeWait, TimeUnit.MILLISECONDS)) != null) {
        if (totalOrderPredicate(n.leader, n.zxid, n.peerEpoch, proposedLeader, proposedZxid, proposedEpoch)) {
            recvqueue.put(n);
            break;
        }
    }

    //如果票池中没有可PK的投票，则认为选举出来的服务为leader
    if (n == null) {
        //修改各个服务的状态，
        setPeerState(proposedLeader, voteSet);
        Vote endVote = new Vote(proposedLeader, proposedZxid, logicalclock.get(), proposedEpoch);
        //清除投票池
        leaveInstance(endVote);
        return endVote;
    }
}
break;
case OBSERVING:
    LOG.debug("Notification from observer: {}", n.sid);
    break;
case FOLLOWING:
case LEADING:
    //如果新收到的选票发送者角色是leader角色状态且选票轮次和自己的选票轮次一样
    if (n.electionEpoch == logicalclock.get()) {
        //则将leader角色投送的这张选票放入自己的选票池中
        recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch, n.state));
        //判断是否有超过半数的票是推荐了n推荐的leader且n.leader也确实是LEADING状态
        voteSet = getVoteTracker(recvset, new Vote(n.version, n.leader, n.zxid, n.electionEpoch, n.peerEpoch, n.state));
        if (voteSet.hasAllQuorums() && checkLeader(recvset, n.leader, n.electionEpoch)) {
            //则指定n推荐的为真正的leader同时修改其他服务对应的状态
            setPeerState(n.leader, voteSet);
            Vote endVote = new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch);
            //清空票池
            leaveInstance(endVote);
            return endVote;
        }
    }

    //如果轮次不一致，则将N的投票记录到outofelection中
    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));
    //判断是否有超过半数的票是推荐了n推荐的leader且n.leader也确实是LEADING状态
    if (voteSet.hasAllQuorums() && checkLeader(outofelection, n.leader, n.electionEpoch)) {
        synchronized (this) {
            //更新当前服务选举轮次
            logicalclock.set(n.electionEpoch);
            //则指定n推荐的为真正的leader同时修改其他服务对应的状态
            setPeerState(n.leader, voteSet);
        }
        Vote endVote = new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch);
        //清空票池
        leaveInstance(endVote);
        return endVote;
    }
}
break;
default:
    LOG.warn("Notification state unrecognized: {} (n.state), {} (n.sid)", n.state, n.sid);
    break;

```

```

    }
    } else {
        if (!validVoter(n.leader)) {
            LOG.warn("Ignoring notification for non-cluster member sid {} from sid {}", n.leader, n.sid);
        }
        if (!validVoter(n.sid)) {
            LOG.warn("Ignoring notification for sid {} from non-quorum member sid {}", n.leader, n.sid);
        }
    }
}
return null;
} finally {
    try {
        if (self.jmxLeaderElectionBean != null) {
            MBeanRegistry.getInstance().unregister(self.jmxLeaderElectionBean);
        }
    } catch (Exception e) {
        LOG.warn("Failed to unregister with JMX", e);
    }
    self.jmxLeaderElectionBean = null;
    LOG.debug("Number of connection processing threads: {}", manager.getConnectionThreadCount());
}
}
}

```

上面多个地方都用到了过半数以上的方法 hasAllQuorums() 该方法用到了 QuorumMaj 类，代码如下：

```

//投票过半的判断函数
public boolean hasAllQuorums() {
    for (QuorumVerifierAcksetPair qvAckset : qvAcksetPairs) {
        //过半数以上算法过程 用到了QuorumMaj类
        if (!qvAckset.getQuorumVerifier().containsQuorum(qvAckset.getAckset())) {
            return false;
        }
    }
    return true;
}

```

QuorumMaj 构造函数中体现了过半数以上的操作，代码如下：

```

public QuorumMaj(Map<Long, QuorumServer> allMembers) {
    this.allMembers = allMembers;
    for (QuorumServer qs : allMembers.values()) {
        //获取所有角色为PARTICIPANT的成员，Observer不参与投票，所以在过半计算中不计入
        if (qs.type == LearnerType.PARTICIPANT) {
            votingMembers.put(Long.valueOf(qs.id), qs);
        } else {
            observingMembers.put(Long.valueOf(qs.id), qs);
        }
    }
    //半数计算
    half = votingMembers.size() / 2;
}

```

3.5.3 投票规则

我们来看一下选票PK的方法 totalOrderPredicate()，该方法其实就是Leader选举规则，规则有如下三个：

- 1:比较 `epoche(zxid高32bit)`，如果其他节点的`epoche`比自己的大，选举 `epoch`大的节点（理由：`epoch` 表示年代，`epoch`越大表示数据越新）代码：`(newEpoch > curEpoch)`;
- 2:比较 `zxid`， 如果`epoche`相同，就比较两个节点的`zxid`的大小，选举 `zxid`大的节点（理由：`zxid` 表示节点所提交事务最大的id，`zxid`越大代表该节点的数据越完整）代码：`(newEpoch == curEpoch) && (newZxid > curZxid)`;
- 3:比较 `serviceId`，如果 `epoch`和`zxid`都相等，就比较服务的`serverId`，选举 `serviceId`大的节点（理由：`serviceId` 表示机器性能，他是在配置`zookeeper`集群时确定的，所以我们配置`zookeeper`集群的时候可以把服务性能更高的集群的`serverId`设置大些，让性能好的机器担任`leader`角色）代码：`(newEpoch == curEpoch) && ((newZxid == curZxid) && (newId > curId))`。

源码如下：

```
protected boolean totalOrderPredicate(long newId, long newZxid, long newEpoch, long curId, long curZxid, long curEpoch) {
    if (self.getQuorumVerifier().getWeight(newId) == 0) {
        return false;
    }

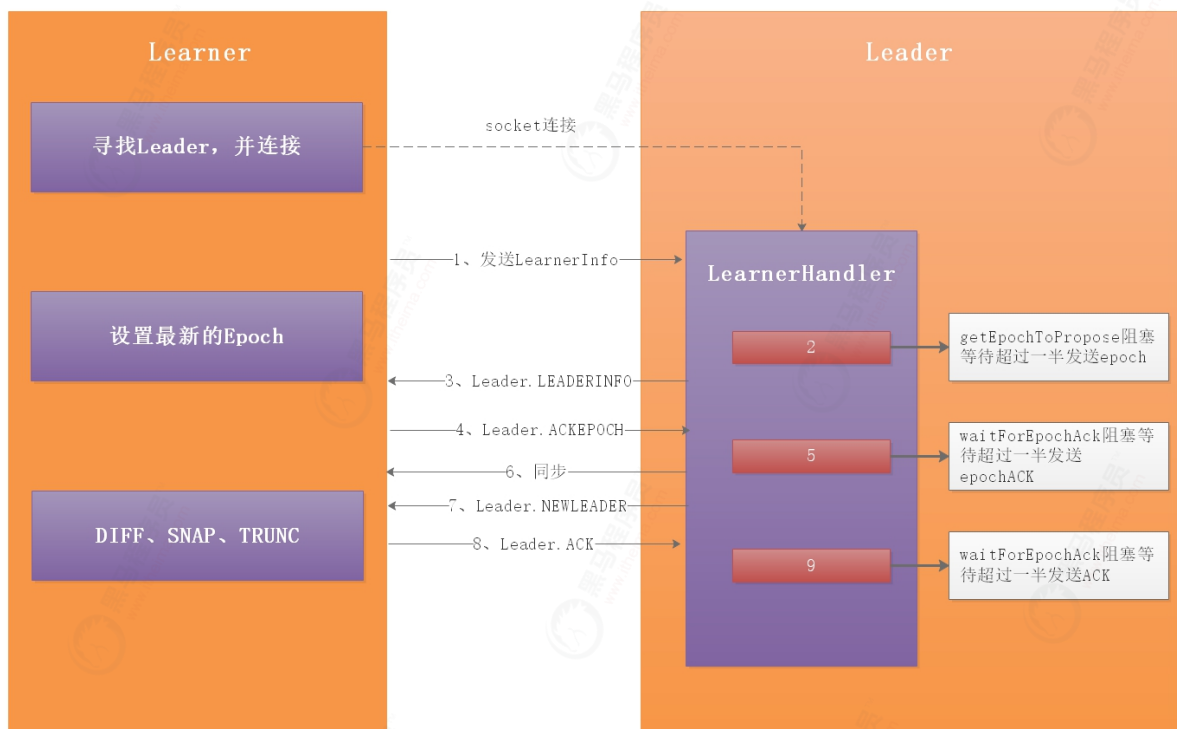
    /*
     * 对应上面代码的解释（两个节点之间使用比较的方法来选举，三种比较规则）
     * 1- 比较 epoche(zxid高32bit)，如果其他节点的epoche比自己的大，选举 epoch大的节点（理由：epoch 表示年代，epoch越大表示数据越新）
     * 2- 比较 zxid， 如果纪元相同，就比较两个节点的zxid的大小，选举 zxid大的节点（理由：zxid 表示节点所提交事务最大的id，zxid越大代表该节点的数据越完整）
     * 3- 比较 serviceId，如果 epoch和zxid都相等，就比较服务的serverId，选举 serviceId大的节点（理由：serviceId 表示机器性能）
     */

    return ((newEpoch > curEpoch)
        || ((newEpoch == curEpoch)
            && ((newZxid > curZxid)
                || ((newZxid == curZxid)
                    && (newId > curId)))));
}
```

4 Zookeeper集群数据同步

所有事务操作都将由`leader`执行，并且会把数据同步到其他节点，比如`follower`、`observer`，我们可以分析`leader`和`follower`的操作行为即可分析出数据同步流程。

4.1 Zookeeper同步流程说明



整体流程：

- 1: 当角色确立之后，**leader**调用**leader.lead()**；方法运行，创建一个接收连接的**LearnerCnxAcceptor**线程，在**LearnerCnxAcceptor**线程内部又建立一个阻塞的**LearnerCnxAcceptorHandler**线程等待**Learner**端的连接。**Learner**端以**follower**为例，**follower**调用**follower.followLeader()**；方法首先查找**leader**的**Socket**服务端，然后建立连接。当**follower**建立连接后，**leader**端会建立一个**LearnerHandler**线程相对应，用来处理**follower**与**leader**的数据包传输。
- 2: **follower**端封装当前zk服务器的**Zxid**和**Leader.FOLLOWERINFO**的**LearnerInfo**数据包发送给**leader**
- 3: **leader**端这时处于**getEpochToPropose**方法的阻塞时期，需要得到**Learner**端超过一半的服务器发送**Epoch**
- 4: **getEpochToPropose**解阻塞之后，**LearnerHandler**线程会把超过一半的**Epoch**与**leader**比较得到最新的**newLeaderZxid**，并封装成**Leader.LEADERINFO**包发送给**Learner**端
- 5: **Learner**端得到最新的**Epoch**，会更新当前服务器的**Epoch**。并把当前服务器所处的**lastLoggedZxid**位置封装成**Leader.ACKEPOCH**发送给**leader**
- 6: 此时**leader**端处于**waitForEpochAck**方法的阻塞时期，需要得到**Learner**端超过一半的服务器发送**EpochACK**
- 7: 当**waitForEpochAck**阻塞之后便可以在**LearnerHandler**线程内决定用那种方式进行同步。如果**Learner**端的**lastLoggedZxid > leader**端的，**Learner**端将会被删除多余的部分。如果小于**leader**端的，将会以不同方式进行同步
- 8: **leader**端发送**Leader.NEWLEADER**数据包给**Learner**端（6、7步骤都是另开一个线程来发送这些数据包）
- 9: **Learner**端同步之后，会在一个**while**循环内处理各种**leader**端发送数据包，包括两阶段提交的**Leader.PROPOSAL**、**Leader.COMMIT**、**Leader.INFORM**等。在同步数据后会处理**Leader.NEWLEADER**数据包，然后发送**Leader.ACK**给**leader**端
- 10: 此时**leader**端处于**waitForNewLeaderAck**阻塞等待超过一半节点发送**ACK**。

我们回到 `QuorumPeer.run()` 方法，根据确认的不同角色执行不同操作展开分析。

4.2 Zookeeper Follower同步流程

Follower主要连接Leader实现数据同步，我们看看Follower做的事，我们仍然沿着`QuorumPeer.run()`展开学习，关键代码如下：

```
case FOLLOWING:
    try {
        // 创建Follower对象，并且设置Follower
        setFollower(makeFollower(logFactory));
        /**
         * 这里会进行一个死循环，主要做的逻辑
         * 1、连接Leader
         * 2、和Leader进行数据同步
         * 3、同步完毕后，正常接收Leader的请求，并且执行对应的逻辑，包括Request、Propose、Commit等请求
         */
        follower.followLeader();
    } catch (Exception e) {
        LOG.warn("Unexpected exception", e);
    } finally {
        // 如果集群超过半数服务宕机或者Leader宕机，那么首先设置调用shutdown()，然后设置状态为LOOKING，会重新触发选举
        follower.shutdown();
        setFollower(null);
        updateServerState();
    }
    break;
```

创建Follower的方法比较简单，代码如下：

```
/**
 * 把自己作为Follower
 * @param logFactory
 * @return
 * @throws IOException
 */
protected Follower makeFollower(FileTxnSnapLog logFactory) throws IOException {
    return new Follower(self: this, new FollowerZooKeeperServer(logFactory, self: this, this.zkDb));
}
```

我们看一下整个Follower在数据同步中做的所有操作 `follower.followLeader()`；，源码如下图：

```

/**
 * Follower调用Leader的方法
 * @throws InterruptedException
 */
void followLeader() throws InterruptedException {
    self.end_file = Time.currentElapsedTime();
    long electionTimeTaken = self.end_file - self.start_file;
    self.setElectionTimeTaken(electionTimeTaken);
    ServerMetrics.getMetrics().ELECTION_TIME.add(electionTimeTaken);
    LOG.info("FOLLOWING - LEADER ELECTION TOOK - {} {}", electionTimeTaken, QuorumPeer.FLE_TIME_UNIT);
    self.start_file = 0;
    self.end_file = 0;
    fzk.registerJMX(new FollowerBean(follower: this, zk), self.jmxLocalPeerBean);

    long connectionTime = 0;
    boolean completedSync = false;

    try {
        self.setZabState(QuorumPeer.ZabState.DISCOVERY);
        //寻找Leader
        QuorumServer leaderServer = findLeader();
        try {
            //创建链接
            connectToLeader(leaderServer.addr, leaderServer.hostname);
            connectionTime = System.currentTimeMillis();
            //注册Follower, 会将当前Follower节点信息发送给Leader节点
            long newEpochZxid = registerWithLeader(Leader.FOLLOWERINFO);
            if (self.isReconfigStateChange()) {
                throw new Exception("learned about role change");
            }
            //check to see if the leader zxid is lower than ours
            //this should never happen but is just a safety check
            long newEpoch = ZxidUtils.getEpochFromZxid(newEpochZxid);
            if (newEpoch < self.getAcceptedEpoch()) {
                LOG.error("Proposed leader epoch "
                    + ZxidUtils.zxidToString(newEpochZxid)
                    + " is less than our accepted epoch "
                    + ZxidUtils.zxidToString(self.getAcceptedEpoch()));
                throw new IOException("Error: Epoch of leader is lower");
            }
            long startTime = Time.currentElapsedTime();
            try {
                self.setLeaderAddressAndId(leaderServer.addr, leaderServer.getId());
                self.setZabState(QuorumPeer.ZabState.SYNCHRONIZATION);
                //和Leader同步历史数据
                syncWithLeader(newEpochZxid);
                self.setZabState(QuorumPeer.ZabState.BROADCAST);
                completedSync = true;
            } finally {
                long syncTime = Time.currentElapsedTime() - startTime;
                ServerMetrics.getMetrics().FOLLOWER_SYNC_TIME.add(syncTime);
            }
            if (self.getObserverMasterPort() > 0) {
                LOG.info("Starting ObserverMaster");

                om = new ObserverMaster(self, fzk, self.getObserverMasterPort());
                om.start();
            } else {
                om = null;
            }
        }
        // 读取Leader发送的数据包
        QuorumPacket qp = new QuorumPacket();
        while (this.isRunning()) {
            //读取数据包
            readPacket(qp);
            //处理数据包
        }
    }
}

```

寻找Leader

创建链接

注册Follower信息

和Leader同步历史数据

读取数据包并解析数据包

```

        processPacket(qp);
    }
} catch (Exception e) {
    LOG.warn("Exception when following the leader", e);
    closeSocket();

    // clear pending revalidations
    pendingRevalidations.clear();
}
} finally {
    if (om != null) {
        om.stop();
    }

    zk.unregisterJMX(peer: this);

    if (connectionTime != 0) {
        long connectionDuration = System.currentTimeMillis() - connectionTime;
        LOG.info(
            "Disconnected from leader (with address: {}). Was connected for {}ms. Sync state: {}",
            leaderAddr,
            connectionDuration,
            completedSync);
        messageTracker.dumpToLog(leaderAddr.toString());
    }
}
}

```

上面源码中的 `follower.followLeader()` 方法主要做了如下几件事：

- 1: 寻找Leader
- 2: 和Leader创建链接
- 3: 向Leader注册Follower，会将当前Follower节点信息发送给Leader节点
- 4: 和Leader同步历史数据
- 5: 读取Leader发送的数据包
- 6: 同步Leader数据包

我们对 `follower.followLeader()` 调用的其他方法进行剖析，其中 `findLeader()` 是寻找当前Leader节点的，源代码如下：

```

protected QuorumServer findLeader() {
    QuorumServer leaderServer = null;
    // Leader的信息
    Vote current = self.getCurrentVote(); // 获取当前投票选出的Leader
    // 循环匹配Leader
    for (QuorumServer s : self.getView().values()) {
        if (s.id == current.getId()) {
            // 获取Leader的地址信息
            s.recreateSocketAddresses();
            leaderServer = s;
            break; // 循环查找Leader
        }
    }

    if (leaderServer == null) {
        LOG.warn("Couldn't find the leader with id = {}", current.getId());
    }

    return leaderServer;
}

```

followLeader() 中调用了 registerWithLeader(Leader.FOLLOWERINFO); 该方法是向Leader注册Follower，会将当前Follower节点信息发送给Leader节点，Follower节点信息发给Leader是必须的，是Leader同步数据个基础，源码如下：

```
protected long registerWithLeader(int pktType) throws IOException {
    /*
     * Send follower info, including last zxid and sid
     */
    long lastLoggedZxid = self.getLastLoggedZxid();
    QuorumPacket qp = new QuorumPacket();
    qp.setType(pktType);
    qp.setZxid(ZxidUtils.makeZxid(self.getAcceptedEpoch(), counter: 0));

    /*
     * Add sid to payload
     */
    LearnerInfo li = new LearnerInfo(self.getId(), protocolVersion: 0x10000, self.getQuorumVerifier().getVersion());
    ByteArrayOutputStream bsid = new ByteArrayOutputStream();
    BinaryOutputArchive boa = BinaryOutputArchive.getArchive(bsid);
    //向Leader写payload
    boa.writeRecord(li, tag: "LearnerInfo");
    qp.setData(bsid.toByteArray());

    //向Leader写数据包
    writePacket(qp, flush: true);
    readPacket(qp);
    final long newEpoch = ZxidUtils.getEpochFromZxid(qp.getZxid());
    if (qp.getType() == Leader.LEADERINFO) {
        // we are connected to a 1.0 server so accept the new epoch and read the next packet
        leaderProtocolVersion = ByteBuffer.wrap(qp.getData()).getInt();
        byte[] epochBytes = new byte[4];
        final ByteBuffer wrappedEpochBytes = ByteBuffer.wrap(epochBytes);
        if (newEpoch > self.getAcceptedEpoch()) {
            wrappedEpochBytes.putInt((int) self.getCurrentEpoch());
            self.setAcceptedEpoch(newEpoch);
        } else if (newEpoch == self.getAcceptedEpoch()) {
            // since we have already acked an epoch equal to the leaders, we cannot ack
            // again, but we still need to send our lastZxid to the leader so that we can
            // sync with it if it does assume leadership of the epoch.
            // the -1 indicates that this reply should not count as an ack for the new epoch
            wrappedEpochBytes.putInt(-1);
        } else {
            throw new IOException("Leaders epoch, "
                + newEpoch
                + " is less than accepted epoch, "
                + self.getAcceptedEpoch());
        }
    }
    //向Leader写ackNewEpoch
    QuorumPacket ackNewEpoch = new QuorumPacket(Leader.ACKEPOCH, lastLoggedZxid, epochBytes, authinfo: null);
    writePacket(ackNewEpoch, flush: true);
    return ZxidUtils.makeZxid(newEpoch, counter: 0);
} else {
    if (newEpoch > self.getAcceptedEpoch()) {
        self.setAcceptedEpoch(newEpoch);
    }
    if (qp.getType() != Leader.NEWLEADER) {
        LOG.error("First packet should have been NEWLEADER");
        throw new IOException("First packet should have been NEWLEADER");
    }
    return qp.getZxid();
}
```

followLeader() 中最后读取数据包执行同步的方法中调用了 readPacket(qp);，这个方法就是读取Leader的数据包的封装，源码如下：

```

/**
 * 从Leader读取数据包
 */
void readPacket(QuorumPacket pp) throws IOException {
    synchronized (leaderIs) {
        leaderIs.readRecord(pp, tag: "packet");
        messageTracker.trackReceived(pp.getType());
    }
    if (LOG.isTraceEnabled()) {
        final long traceMask =
            (pp.getType() == Leader.PING) ? ZooTrace.SERVER_PING_TRACE_MASK
            : ZooTrace.SERVER_PACKET_TRACE_MASK;

        ZooTrace.logQuorumPacket(LOG, traceMask, direction: 'i', pp);
    }
}

```

4.3 Zookeeper Leader同步流程

我们查看 `QuorumPeer.run()` 方法的LEADING部分，可以看到先创建了Leader对象，并设置了Leader，然后调用了 `leader.lead()`，`leader.lead()` 是执行的核心业务流程，源码如下：

```

case LEADING:
    // Leader需要执行的逻辑
    try {
        // 创建Leader对象，并且设置Leader
        setLeader(makeLeader(logFactory)); // 设置Leader
        /*
         * 这里会进行一个死循环，主要做的逻辑有
         * 1、接受Follower的连接
         * 2、进行Follower进行数据同步
         * 3、同步完成后，正常接收请求（主要包括客户端发来的请求、
         * 集群Follower转发的事务请求等等）
         */
        leader.lead();
        // 如果集群超过半数服务宕机，那么首先设置Leader对象为null
        setLeader(null); // 执行业务操作
    } catch (Exception e) {
        LOG.warn("Unexpected exception", e);
    } finally {
        if (leader != null) {
            leader.shutdown(reason: "Forcing shutdown");
            setLeader(null);
        }
        // 设置zk状态为LOOKING，需要重新触发选举
        updateServerState();
    }
    break;
}

```

`leader.lead()` 方法是Leader执行的核心业务流程，源码如下：

```

/**
 * Leader主要功能的方法
 */
void lead() throws IOException, InterruptedException {
    self.end_fle = Time.currentElapsedTime();
    long electionTimeTaken = self.end_fle - self.start_fle;
    self.setElectionTimeTaken(electionTimeTaken);
    ServerMetrics.getMetrics().ELECTION_TIME.add(electionTimeTaken);
    LOG.info("LEADING - LEADER ELECTION TOOK - {} {}", electionTimeTaken, QuorumPeer.FLE_TIME_UNIT);
    self.start_fle = 0;
    self.end_fle = 0;

    zk.registerJMX(new LeaderBean(leader: this, zk), self.jmxLocalPeerBean);

    try {
        self.setZabState(QuorumPeer.ZabState.DISCOVERY);
        self.tick.set(0);
        //从快照和事务日志中加载数据
        zk.loadData();

        leaderStateSummary = new StateSummary(self.getCurrentEpoch(), zk.getLastProcessedZxid());

        // 创建一个线程，接收Follower/Observer的连接
        cnxAcceptor = new LearnerCnxAcceptor();
        // 开启线程
        cnxAcceptor.start();

        // 等待超过一半的(Follower和Observer)连接，这里才会往下执行，返回新的朝代epoch
        // 反之，阻塞在这里
        long epoch = getEpochToPropose(self.getId(), self.getAcceptedEpoch());

        // 根据新的epoch，设置新的起始zxid
        zk.setZxid(ZxidUtils.makeZxid(epoch, counter: 0));

        synchronized (this) {
            lastProposed = zk.getZxid();
        }

        newLeaderProposal.packet = new QuorumPacket(NEWLEADER, zk.getZxid(), data: null, authinfo: null);

        if ((newLeaderProposal.packet.getZxid() & 0xffffffffL) != 0) {
            LOG.info("NEWLEADER proposal has Zxid of {}", Long.toHexString(newLeaderProposal.packet.getZxid()));
        }

        QuorumVerifier lastSeenQV = self.getLastSeenQuorumVerifier();
        QuorumVerifier curQV = self.getQuorumVerifier();
        if (curQV.getVersion() == 0 && curQV.getVersion() == lastSeenQV.getVersion()) {
            // This was added in ZOOKEEPER-1783. The initial config has version 0 (not explicitly
            // specified by the user; the lack of version in a config file is interpreted as version=0).
            // As soon as a config is established we would like to increase its version so that it
            // takes precedence over other initial configs that were not established (such as a config
            // of a server trying to join the ensemble, which may be a partial view of the system, not the full config).
            // We chose to set the new version to the one of the NEWLEADER message. However, before we can do that
            // there must be agreement on the new version, so we can only change the version when sending/receiving UPTODATE,
            // not when sending/receiving NEWLEADER. In other words, we can't change curQV here since its the committed quorum
            // and there's still no agreement on the new version that we'd like to use. Instead, we use
            // lastSeenQuorumVerifier which is being sent with NEWLEADER message
            // so its a good way to let followers know about the new version. (The original reason for sending
            // lastSeenQuorumVerifier with NEWLEADER is so that the leader completes any potentially uncommitted reconfs
            // that it finds before starting to propose operations. Here we're reusing the same code path for
            // reaching consensus on the new version number.)

            // It is important that this is done before the leader executes waitForEpochAck,
            // so before LearnerHandlers return from their waitForEpochAck
            // hence before they construct the NEWLEADER message containing
            // the last-seen-quorumverifier of the leader, which we change below
            try {
                LOG.debug(String.format("set lastSeenQuorumVerifier to currentQuorumVerifier (%s)", curQV.toString()));
                QuorumVerifier newQV = self.configFromQuorumVerifier(curQV.toString());
                newQV.setVersion(zk.getZxid());
                self.setLastSeenQuorumVerifier(newQV, writeToDisk: true);
            } catch (Exception e) {
                throw new IOException(e);
            }
        }
    }
}

```

```

newLeaderProposal.addQuorumVerifier(self.getQuorumVerifier());
if (self.getLastSeenQuorumVerifier().getVersion() > self.getQuorumVerifier().getVersion()) {
    newLeaderProposal.addQuorumVerifier(self.getLastSeenQuorumVerifier());
}

// 等待超过一半的(Follower和Observer)获取了新的epoch, 并且返回了Leader.ACKEPOCH
// 这里才会往下执行。 反之, 阻塞在这里
waitForEpochAck(self.getId(), leaderStateSummary);
// 设置当前新的朝代epoch
self.setCurrentEpoch(epoch);
self.setLeaderAddressAndId(self.getQuorumAddress(), self.getId());
self.setZabState(QuorumPeer.ZabState.SYNCHRONIZATION);

try {
    // 等待超过一半的(Follower和Observer)进行数据同步成功, 并且返回了Leader.ACK
    // 这里才会往下执行。 反之, 阻塞在这里
    waitForNewLeaderAck(self.getId(), zk.getZxid());
} catch (InterruptedException e) {
    shutdown(reason: "Waiting for a quorum of followers, only synced with sids: [ "
        + newLeaderProposal.ackSetsToString()
        + " ]");
    HashSet<Long> followerSet = new HashSet<>();

    for (LearnerHandler f : getLearners()) {
        if (self.getQuorumVerifier().getVotingMembers().containsKey(f.getSid())) {
            followerSet.add(f.getSid());
        }
    }

    boolean initTicksShouldBeIncreased = true;
    for (Proposal.QuorumVerifierAcksetPair qvAckset : newLeaderProposal.qvAcksetPairs) {
        if (!qvAckset.getQuorumVerifier().containsQuorum(followerSet)) {
            initTicksShouldBeIncreased = false;
            break;
        }
    }

    if (initTicksShouldBeIncreased) {
        LOG.warn("Enough followers present. Perhaps the initTicks need to be increased.");
    }

    return;
}

// 走到这里说明集群中数据已经同步完成, 可以正常运行
// 开启zkServer, 并且同时开启请求调用链接接收请求执行
startZkServer();

/**
 * WARNING: do not use this for anything other than QA testing
 * on a real cluster. Specifically to enable verification that quorum
 * can handle the lower 32bit roll-over issue identified in
 * ZOOKEEPER-1277. Without this option it would take a very long
 * time (on order of a month say) to see the 4 billion writes
 * necessary to cause the roll-over to occur.
 *
 * This field allows you to override the zxid of the server. Typically
 * you'll want to set it to something like 0xffffffff0 and then
 * start the quorum, run some operations and see the re-election.
 */
String initialZxid = System.getProperty("zookeeper.testonly.initialZxid");
if (initialZxid != null) {
    long zxid = Long.parseLong(initialZxid);
    zk.setZxid((zk.getZxid() & 0xffffffff00000000L) | zxid);
}

if (!System.getProperty(key: "zookeeper.leaderServes", def: "yes").equals("no")) {
    self.setZooKeeperServer(zk);
}

self.setZabState(QuorumPeer.ZabState.BROADCAST);
self.adminServer.setZooKeeperServer(zk);

// We ping twice a tick, so we only update the tick every other
// iteration
boolean tickSkip = true;
// If not null then shutdown this leader
String shutdownMessage = null;

```

```

// 进行一个死循环，每次休眠self.tickTime / 2，和对所有的(Observer/Follower)发起心跳检测
while (true) {
    synchronized (this) {
        long start = Time.currentTimeMillis();
        long cur = start;
        long end = start + self.tickTime / 2;
        while (cur < end) {
            wait((timeout: end - cur);
            cur = Time.currentTimeMillis();
        }

        if (!tickSkip) {
            self.tick.incrementAndGet();
        }

        // We use an instance of SyncedReaderTracker to
        // track synced learners to make sure we still have a
        // quorum of current (and potentially next pending) view.
        SyncedReaderTracker syncedAckSet = new SyncedReaderTracker();
        syncedAckSet.addQuorumVerifier(self.getQuorumVerifier());
        if (self.getLastSeenQuorumVerifier() != null
            && self.getLastSeenQuorumVerifier().getVersion() > self.getQuorumVerifier().getVersion()) {
            syncedAckSet.addQuorumVerifier(self.getLastSeenQuorumVerifier());
        }

        // 将Follower加入该容器
        syncedAckSet.addAck(self.getId());

        for (LearnerHandler f : getLearners()) {
            if (f.synced()) {
                syncedAckSet.addAck(f.getSid());
            }
        }

        // check leader running status
        if (!this.isRunning()) {
            // set shutdown flag
            shutdownMessage = "Unexpected internal error";
            break;
        }

        // 判断是否有超过一半Follower在集群中
        if (!tickSkip && !syncedReaderAckSet.hasAllQuorums()) {
            // Lost quorum of last committed and/or last proposed
            // config, set shutdown flag
            // 如果没有，那么调用shutdown关闭一些对象，然后return，重新选举
            shutdownMessage = "Not sufficient followers synced, only synced with sids: [ "
                + syncedAckSet.ackSetsToString()
                + " ]";
            break;
        }

        tickSkip = !tickSkip;
    }

    for (LearnerHandler f : getLearners()) {
        f.ping();
    }

    if (shutdownMessage != null) {
        // 没有过半Follower在集群中，调用shutdown关闭一些对象，重新选举
        shutdown(shutdownMessage);
        // leader goes in looking state
    }
} finally {
    zk.unregisterJMX(leader, this);
}
}

```

Leader.lead() 方法会执行如下几个操作：

- 1:从快照和事务日志中加载数据
- 2:创建一个线程,接收Follower/Observer的连接
- 3:等待超过一半的(Follower和Observer)连接,再继续往下执行程序
- 4:等待超过一半的(Follower和Observer)获取了新的epoch,并且返回了Leader.ACKEPOCH,再继续往下执行程序
- 5:等待超过一半的(Follower和Observer)进行数据同步成功,并且返回了Leader.ACK,再继续往下执行程序
- 6:数据同步完成,开启zkServer,并且同时开启请求调用链接接收请求执行
- 7:进行一个死循环,每次休眠 $\text{self.tickTime} / 2$,和对所有的(Observer/Follower)发起心跳检测
- 8:集群中没有过半Follower在集群中,调用shutdown关闭一些对象,重新选举

lead()方法中会创建LearnerCnxAcceptor,该对象是一个线程,主要用于接收followers的连接,这里加了CountDownLatch根据配置的同步的地址的数量(例如:server.2=127.0.0.1:12881:13881配置同步的端口是12881只有一个),LearnerCnxAcceptor的run方法源码如下:

```
@Override
public void run() {
    if (!stop.get() && !serverSockets.isEmpty()) {
        ExecutorService executor = Executors.newFixedThreadPool(serverSockets.size());
        CountDownLatch latch = new CountDownLatch(serverSockets.size());

        //创建socket链接,并等待Follower节点链接
        serverSockets.forEach(serverSocket ->
            executor.submit(new LearnerCnxAcceptorHandler(serverSocket, latch)));

        try {
            latch.await(); //阻塞等待
        } catch (InterruptedException ie) {
            LOG.error("Interrupted while sleeping in LearnerCnxAcceptor.", ie);
        } finally {
            closeSockets();
            executor.shutdown();
            try {
                if (!executor.awaitTermination(timeout: 1, TimeUnit.SECONDS)) {
                    LOG.error("not all the LearnerCnxAcceptorHandler terminated properly");
                }
            }
        }
    }
}
```

LearnerCnxAcceptor的run方法中创建了LearnerCnxAcceptorHandler对象,在接收到链接后,就会调用LearnerCnxAcceptorHandler,而LearnerCnxAcceptorHandler是一个线程,它的run方法中调用了acceptConnections()方法,源码如下:

```
@Override
public void run() {
    try {
        Thread.currentThread().setName("LearnerCnxAcceptorHandler-" + serverSocket.getLocalSocketAddress());

        while (!stop.get()) {
            acceptConnections();
        }

        catch (Exception e) {
            LOG.warn("Exception while accepting follower", e);
            if (fail.compareAndSet(expect: false, update: true)) {
                handleException(getName(), e);
                halt();
            }
        } finally {
            latch.countDown(); //此处可以解除阻塞
        }
    }
}
```

`acceptConnections()` 方法会在这里阻塞接收followers的连接，当有连接过来会生成一个socket对象。然后根据当前socket生成一个LearnerHandler线程，每个Learner者都会开启一个LearnerHandler线程，方法源码如下：

```
private void acceptConnections() throws IOException {
    Socket socket = null;
    boolean error = false;
    try {
        //接收请求
        socket = serverSocket.accept();

        // start with the initLimit, once the ack is processed
        // in LearnerHandler switch to the syncLimit
        socket.setSoTimeout(self.tickTime * self.initLimit);
        socket.setTcpNoDelay(nodeLAY);
        //获取数据流
        BufferedInputStream is = new BufferedInputStream(socket.getInputStream());
        //创建LearnerHandler线程
        LearnerHandler fh = new LearnerHandler(socket, is, learnerMaster: Leader.this);
        fh.start();
    } catch (SocketException e) {
```

`LearnerHandler.run` 这里就是读取或写数据包与Learner交换数据包。如果没有数据包读取，则会阻塞当前方法 `ia.readRecord(qp, "packet");`，源码如下：

```
@Override
public void run() {
    try {
        learnerMaster.addLearnerHandler(this);
        tickOfNextAckDeadline = learnerMaster.getTickOfInitialAckDeadline();

        ia = BinaryInputArchive.getArchive(bufferedInput);
        bufferedOutput = new BufferedOutputStream(sock.getOutputStream());
        oa = BinaryOutputArchive.getArchive(bufferedOutput);

        // 等待读取Follower/Observer发出的请求 请求包的类型Leader.FOLLOWERINFO或者Leader.OBSERVERINFO
        QuorumPacket qp = new QuorumPacket();
        //阻塞读取数据
        ia.readRecord(qp, tag: "packet");

        messageTracker.trackReceived(qp.getType());
        if (qp.getType() != Leader.FOLLOWERINFO && qp.getType() != Leader.OBSERVERINFO) {
            LOG.error("First packet {} is not FOLLOWERINFO or OBSERVERINFO!", qp.toString());
        }

        return;
    }
}
```

我们再回到 `Leader.lead()` 方法，其中调用了 `getEpochToPropose()` 方法，该方法是判断 `connectingFollowers` 发给leader端的Epoch是否过半，如果过半则会解阻塞，不过半会一直阻塞着，直到Follower把自己的Epoch数据包发送过来并符合过半机制，源码如下：

```

@Override
public long getEpochToPropose(long sid, long lastAcceptedEpoch) throws InterruptedException, IOException {
    synchronized (connectingFollowers) {
        if (!waitingForNewEpoch) {
            return epoch;
        }
        if (lastAcceptedEpoch >= epoch) {
            epoch = lastAcceptedEpoch + 1;
        }
        if (isParticipant(sid)) {
            connectingFollowers.add(sid);
        }
        QuorumVerifier verifier = self.getQuorumVerifier();
        if (connectingFollowers.contains(self.getId()) && verifier.containsQuorum(connectingFollowers)) {
            waitingForNewEpoch = false;
            self.setAcceptedEpoch(epoch);
            connectingFollowers.notifyAll();
        } else {
            long start = Time.currentElapsedTime();
            if (sid == self.getId()) {
                timeStartWaitForEpoch = start;
            }
            long cur = start;
            long end = start + self.getInitLimit() * self.getTickTime();
            while (waitingForNewEpoch && cur < end && !quitWaitForEpoch) {
                connectingFollowers.wait(timeout: end - cur);
                cur = Time.currentElapsedTime();
            }
            if (waitingForNewEpoch) {
                throw new InterruptedException("Timeout while waiting for epoch from quorum");
            }
        }
        return epoch;
    }
}

```

过半算法

不过半，会一直在这里阻塞着

在 `lead()` 方法中，当发送的Epoch过半之后，把当前zxid设置到zk，并等待EpochAck，关键源码如下：

```

// 等待超过一半的(Follower和Observer)获取了新的epoch，并且返回了Leader.ACKEPOCH
// 这里才会往下执行。 反之，阻塞在这里
waitForEpochAck(self.getId(), leaderStateSummary);
// 设置当前新的朝代epoch
self.setCurrentEpoch(epoch);
self.setLeaderAddressAndId(self.getQuorumAddress(), self.getId());
self.setZabState(QuorumPeer.ZabState.SYNCHRONIZATION);

```

`waitForEpochAck()` 方法也会等待超过一半的(Follower和Observer)获取了新的epoch，并且返回了Leader.ACKEPOCH，才会解除阻塞，否则会一直阻塞。等待EpochAck解除阻塞后，把得到最新的epoch更新到当前服务，设置当前leader节点的zab状态是 SYNCHRONIZATION，方法源码如下：

```

@Override
public void waitForEpochAck(long id, StateSummary ss) throws IOException, InterruptedException {
    synchronized (electingFollowers) {
        if (electionFinished) {
            return;
        }
        if (ss.getCurrentEpoch() != -1) {
            if (ss.isMoreRecentThan(leaderStateSummary)) {
                throw new IOException("Follower is ahead of the leader, leader summary: "
                    + leaderStateSummary.getCurrentEpoch()
                    + " (current epoch), "
                    + leaderStateSummary.getLastZxid()
                    + " (last zxid)");
            }
            if (ss.getLastZxid() != -1 && isParticipant(id)) {
                electingFollowers.add(id);
            }
        }
        //过半才能解除阻塞 等待超过一半的(Follower和Observer)获取了新的epoch, 并且返回了Leader.ACKEPOCH
        QuorumVerifier verifier = self.getQuorumVerifier();
        if (electingFollowers.contains(self.getId()) && verifier.containsQuorum(electingFollowers)) {
            electionFinished = true;
            electingFollowers.notifyAll();
        } else {
            long start = Time.currentElapsedTime();
            long cur = start;
            long end = start + self.getInitLimit() * self.getTickTime();
            while (!electionFinished && cur < end) {
                //否则一直等待
                electingFollowers.wait(timeout: end - cur);
                cur = Time.currentElapsedTime();
            }
            if (!electionFinished) {
                throw new InterruptedException("Timeout while waiting for epoch to be acked by quorum");
            }
        }
    }
}

```

lead() 方法中还需要等待超过一半的(Follower和Observer)进行数据同步成功, 并且返回了Leader.ACK, 程序才会解除阻塞, 如下代码:

```

try {
    // 等待超过一半的(Follower和Observer)进行数据同步成功, 并且返回了Leader.ACK
    // 这里才会往下执行。反之, 阻塞在这里
    waitForNewLeaderAck(self.getId(), zk.getZxid());
} catch (InterruptedException e) {
}

```

上面所有流程都走完之后, 就证明数据已经同步成功了, 会执行startZkServer();

4.4 LearnerHandler数据同步操作

LearnerHandler 线程是对应于 Learner 连接 Leader 端后, 建立的一个与 Learner 端交换数据的线程。每一个 Learner 端都会创建一个 LearnerHandler 线程。

我们详细讲解 LearnerHandler.run() 方法。

```
// 等待读取Follower/Observer发出的请求 请求包的类型Leader.FOLLOWERINFO或者Leader.OBSERVERINFO
QuorumPacket qp = new QuorumPacket();
//阻塞读取数据
ia.readRecord(qp, tag: "packet");
```

readRecord 读取数据包 不断从 learner 节点读数据，如果没读到将会阻塞 readRecord。

```
messageTracker.trackReceived(qp.getType());
if (qp.getType() != Leader.FOLLOWERINFO && qp.getType() != Leader.OBSERVERINFO) {
    LOG.error("First packet {} is not FOLLOWERINFO or OBSERVERINFO!", qp.toString());

    return;
}
```

如果数据包类型不是Leader.FOLLOWERINFO或Leader.OBSERVERINFO将会返回，因为咱们这里本身就是Leader节点，读数据肯定是读非Leader节点数据。

```
// 读取请求的字节数据
byte[] learnerInfoData = qp.getData() 获取learnerInfoData，用它来获取版本信息以及sid
if (learnerInfoData != null) {
    ByteBuffer bbsid = ByteBuffer.wrap(learnerInfoData);
    // 根据字节数据的长度判断
    if (learnerInfoData.length >= 8) {
        // 获取myid
        this.sid = bbsid.getLong();
    }
    if (learnerInfoData.length >= 12) {
        // 获取协议版本号，默认为 0x10000
        this.version = bbsid.getInt(); // protocolVersion
    }
    if (learnerInfoData.length >= 20) {
        long configVersion = bbsid.getLong();
        if (configVersion > learnerMaster.getQuorumVerifierVersion()) {
            throw new IOException("Follower is ahead of the leader (has a later activated configuration)");
        }
    }
} else {
    // 获取myid
    this.sid = learnerMaster.getAndDecrementFollowerCounter();
}
```

获取 learnerInfoData 来获取sid和版本信息。

```
String followerInfo = learnerMaster.getPeerInfo(this.sid); 获取followerInfo
if (followerInfo.isEmpty()) {
    LOG.info(
        "Follower sid: {} not in the current config {}",
        this.sid,
        Long.toHexString(learnerMaster.getQuorumVerifierVersion()));
} else {
    LOG.info("Follower sid: {} : info : {}", this.sid, followerInfo);
}

if (qp.getType() == Leader.OBSERVERINFO) {
    learnerType = LearnerType.OBSERVER;
}

learnerMaster.registerLearnerHandlerBean(learnerHandler: this, sock);

long lastAcceptedEpoch = ZxidUtils.getEpochFromZxid(qp.getZxid());
```

获取followerInfo和lastAcceptedEpoch，信息如下：

```
> followerInfo = "127.0.0.1:12881:13881:participant"
lastAcceptedEpoch = 8
```

```
if (getVersion() < 0x10000) {
    QuorumPacket newLeaderQP = new QuorumPacket(Leader.NEWLEADER, newLeaderZxid, data: null, authinfo: null);
    oa.writeRecord(newLeaderQP, tag: "packet");
} else {
    //把Leader.NEWLEADER数据包放入到queuedPackets
    QuorumPacket newLeaderQP = new QuorumPacket(Leader.NEWLEADER, newLeaderZxid, learnerMaster.getQuorumVerifierBytes(),
    queuedPackets.add(newLeaderQP);
}
bufferedOutput.flush();

// 启动数据包发送
startSendingPackets();
```

把Leader.NEWLEADER数据包放入到queuedPackets，并向其他节点发送，源码如下：

```
protected void startSendingPackets() {
    if (!sendingThreadStarted) {
        // Start sending packets
        new Thread() {
            public void run() {
                Thread.currentThread().setName("Sender-" + sock.getRemoteSocketAddress());
                try {
                    sendPackets();
                } catch (InterruptedException e) {
                    LOG.warn("Unexpected interruption", e);
                }
            }
        }.start();
        sendingThreadStarted = true;
    } else {
        LOG.error("Attempting to start sending thread after it already started");
    }
}
```