分析选举超时定时器，重构节点选举流程，实现领导者选举(四)

完善选举超时定时器功能

大家好，新的一章又开始了，虽然距离过年还有一段时间，但是我不知道用什么话来打开这一章的局面，就提前给大家拜个早年吧。祝大家在新的一年里继续见多识广，和老板以及老板娘还有 HR 谈笑风生！很好，开场白说完了，接下来我们就接着上一章的内容，继续实现我们自己的 raft 框架。

在正式开始这一章的内容之前，我必须得抽出一点时间，为大家补充一个上一章遗漏的小知识，这是个非常简单的知识，本来我在上一章跟大家说了，会在上一章的最后一小节讲解，但是讲着讲着就给忘了。真是不好意思啦，但是我相信大家应该已经习惯了，就像你们每周五下午五点开始开会一样，本来大家都知道六点就下班了，但是可能你们开一会儿会，就把下班时间忘了。当然，就算没忘你们可能也会装作忘了。🐶

好了，这些都是玩笑话，大家千万别介意。下面我就为大家补充上一章遗漏的知识。在上一章我们实现了节点预投票功能，为了避免在发生网络分区时，某个触发了超时选举的节点的任期无线自增，我为程序引入了节点预投票功能。在出发了超时选举的节点开始正式进行领导者选举之前，先进行一次预投票操作，只有预投票操作成功了，该节点才能正式进入领导者选举阶段。**注意，我再次强调一下，这里说的是触发了超时选举，并不是选举超时。当一个节点在规定时间没有收到来自领导者发送过来的消息，就会触发超时选举。**这就意味着当前节点可能发生了网络分区，和集群中的其他节点失去了联系。只要先让当前节点执行预投票操作，就可以保证如果是当前节点自己发生了网络分区，它的任期不会无限自增。

这里面还有一个比较细节的问题，也就是我将要为大家补充的，本来应该在上一章讲解的知识。当一个节点发生了网络分区后，触发了超时选举，它就会将自己的状态更新为候选者，然后进行领导者选举(**这里这样说其实并不太准确，在本章的下一小节，实现了正式的领导者选举后，我会为大家修正，现在还没有实现正式领导者选举，所以无法展开讲解**)。因为现在是它自己和集群中的其他节点失去了联系，所

以它发送的索要投票的请求肯定无法收到响应。既然它一直无法收到响应，也就意味着不会收到足够的票数，这样一来，这个节点会怎么样呢?

这时候大家要理清楚，当节点将自己的状态更新为候选者之后，也会停止自己的超时选举定时器工作，这个知识在上一章已经为大家分析过了，**集群中的领导者和候选者是不需要启动超时选举定时器的。所以，在候选者节点没有收到足够的票数之后，会触发另一个定时器工作，那就是上一章我为大家提到的选举超时定时器。**所谓选举超时，就是一个节点开始进行领导者选举后，在规定时间内没有收到足够票数，无法顺利当选新的领导者，这就是选举超时概念的解释。那么选举超时之后要做什么呢？也就是选举超时定时器的工作是什么呢？这个很容易就想到了，而且上一章我也为大家分析过了，这个选举超时定时器就会让当前节点进入下一轮选举。而进入下一轮选举肯定就会将当前节点的任期自增 1，然后再次向集群中其他节点发送索要投票的请求。如果当前节点一直没有收到足够的票数，当前节点就会在选举超时定时器的作用下，无限自增自己的任期，这显然不是我们希望看到的。上一章我们引入预投票机制，就是为了避免出现这种情况。

那么这时候，其实我要为大家补充的知识点也差不多讲解完了。既然我们都不希望集群中某个节点出现任期无限自增的情况，那么就给这个节点的选举超时定时器也引入预投票机制不就好了？**当一个候选者节点的选举超时定时器持续工作时，就让这个选举超时定时器每一次进行新一轮领导者选举时，先让当前节点进行一次预投票操作。只要预投票操作没有成功，那么当前节点就不会进入领导者正式选举阶段。这样一来，当前候选者节点的任期就不会无限自增了。**这本来应该是上一章的知识点，但是忘记讲解了，现在补充在这里，这样上一章的内容就算完整了，大家也知道了这个预投票机制真正在哪里发挥作用。

在 sofajraft 框架中，这个选举超时定时器也是一个 RepeatedTimer 对象，定时器的名字为 voteTimer。这个定时器也是在 NodeImpl 对象初始化的过程中被创建的，所以这个选举超时定时器会在 NodeImpl 对象的 init 方法中被创建。具体实现请看下面代码块。

NodeImpl Java

```java
public class NodeImpl implements Node{

    private static final Logger LOG = LoggerFactory.getLogger(NodeImpl.class);


    //当前节点所在集群的Id
    private final String groupId;

    //当前节点的Id
    private NodeId nodeId;

    //当前节点的状态
    private volatile State state;

    //节点的当前任期
    private long currTerm;

    //当前节点投过票的节点的PeerId，这个成员变量可以记录下来，当前节点为哪个节点投过票
    private PeerId votedId;

    //当前节点记录的领导者的PeerId
    private PeerId leaderId = new PeerId();

    //当前节点自己的节点信息
    private final PeerId serverId;

    //集群当前生效的配置信息，包括旧的配置信息
    private ConfigurationEntry conf;

    //为jraft框架提供各种服务的工厂，在第一版本中，这个工厂只提供了元数据存储器服务
    private JRaftServiceFactory serviceFactory;

    //注意，这里多出来一个NodeOptions成员变量，节点需要的配置参数对象
    private NodeOptions options;

    //超时选举定时器
    private RepeatedTimer electionTimer;

    //选举超时定时器
    private RepeatedTimer voteTimer;


```

```
43      //当Nodeimpl调用它的init方法时，这时候方法参数NodeOptions
44      //已经创建了DefaultJRaftServiceFactory对象了
45      @Override
46      public boolean init(final NodeOptions opts) {
47
48
49          //省略部分内容
50
51          String name = "JRaft-VoteTimer-" + suffix;
52          //创建投票超时定时处理器
53          //也就是选举超时定时器，这里可以看到，选举超时时间和超时选举时间是一样的
54          //使用的都是this.options.getElectionTimeoutMs()这个值
55          this.voteTimer = new RepeatedTimer(name,this.options.getElectionT
    imeoutMs(),createTimer(name)) {
56
57              @Override
58              protected void onTrigger() {
59                  //处理选举超时的方法
60                  handleVoteTimeout();
61              }
62
63              @Override
64              protected int adjustTimeout(final int timeoutMs) {
65                  return randomTimeout(timeoutMs);
66              }
67          };
68          //下面我删减了很多代码，大家简单看看就行，在最终的源码中快照服务，状态机，日志
    组件都会初始化，但在第一版本中我全删除了
69          name = "JRaft-ElectionTimer-" + suffix;
70          //在这里创建了一个超时选举定时器，this.options.getElectionTimeoutMs()
    得到的就是超时选举时间
71          this.electionTimer = new RepeatedTimer(name,this.options.getElect
    ionTimeoutMs(),createTimer(name)) {
72              //这个方法实现的是RepeatedTimer类中的方法
73              @Override
74              protected void onTrigger() {
75                  //这是超时选举定时器中最核心的方法，就是在该方法中，开始进行超时选举
    任务了
76                  handleElectionTimeout();
77              }
78
79              @Override
80              protected int adjustTimeout(final int timeoutMs) {
81                  //在一定范围内返回一个随机的时间，这就意味着每个节点的超时选举时间是
    不同的
82                  //否则多个节点同时成为候选者，很可能选举失败
83                  return randomTimeout(timeoutMs);
```

```
            }
        };


        //初始化元数据存储器组件
        if (!initMetaStorage()) {
            LOG.error("Node {} initMetaStorage failed.", getNodeId());
            return false;
        }


        //省略部分内容

        //把当前节点添加到节点管理器中
        if (!NodeManager.getInstance().add(this)) {
            LOG.error("NodeManager add {} failed.", getNodeId());
            return false;
        }
    }




    //处理选举超时的方法，这个我写上来了，但是没有使用。
    //大家看看代码就行。
    private void handleVoteTimeout() {
        //判断当前节点状态是否为候选者，只有候选者才会出现投票超时的情况，也就是在一定时间内还没有成为主节点
        //也就是说一个跟随者节点触发了超时选举之后，先进行预投票操作，这个预投票操作通过了，然后进入正式的领导者选举
        //但这时候发生了网络分区(实际上经过预投票之后，发生网络分区的可能性很小)，或者是集群中出现了多个候选者，选举一直没有成功，就会一直让候选者节点的选举超时定时器工作
        if (this.state != State.STATE_CANDIDATE) {
            return;
        }
        //判断当前节点是否设置了选举超时的身份降级方法
        if (this.raftOptions.isStepDownWhenVoteTimedout()) {
            LOG.warn(
                    "Candidate node {} term {} steps down when election reaching vote timeout: fail to get quorum vote-granted.",
                    this.nodeId, this.currTerm);
            //进行身份降级，把当前节点降级为跟随者，然后重新开始预投票操作
            stepDown(this.currTerm);
            //进行预投票操作
            preVote();
        } else {

```

```
                LOG.debug("Node {} term {} retry to vote self.", getNodeId
127 (), this.currTerm);
128             //如果没有设置身份降级，就直接再次进行选举操作，这里开启了新一轮选举操作
                //electSelf这个方法，大家在上一章已经见过了，那就是预投票成功之后，直接
129 进入领导者正式选举的方法
130             //在这个方法中，节点的任期会被真正改变，会自增1
                //这里大家肯定会有疑惑，为什么这里可以直接开始进行正式选举，难道不是先判
131 断一下预投票会不会成功吗?
                //当然，大家还没不知道正式选举的流程呢，所以还是跟着我的思路继续向下看，
132 看看我是怎么把上一章和这一章
133             //的知识连接起来的，也许最后大家会恍然大悟
134             electSelf();
135         }
136     }
137
138
139
140
141     //省略其他方法
142
```

可以看到，在上面的代码块中，我为大家展示了选举超时定时器的具体实现，voteTimer 超时选举定时器也是 NodeImpl 类中的成员变量，并且在该类的 init 方法中被创建了。并且，这个定时器也是一个 RepeatedTimer 对象，这就意味着它也是由时间轮来执行定时任务的。这个也是上一章的知识，所以我也就不再展开讲解了。现在我想对大家强调的是，**当选举超时定时器触发了之后，时间轮就会执行这个定时器中的 handleVoteTimeout 方法，这个才是真正处理选举超时的方法**。

在该方法中，会首先根据 this.raftOptions.isStepDownWhenVoteTimedout() 这行代码来判断，**当选举超时定时器工作的时候，也就是每一次选举超时然后要进行下一轮领导者选举的时候，是否要先让当前的候选者节点进行身份降级，降级为跟随者，然后再执行预投票操作，只有预投票成功之后，才能进入领导者正式选举阶段**。这段逻辑还是挺容易理解的，虽然我在前面为大家分析的时候没有提到让候选者节点身份降级，但是预投票操作肯定是为大家分析了。**而在 sofajraft 框架中，this.raftOptions.isStepDownWhenVoteTimedout() 默认为 true，所以选举超时定时器工作的时候，肯定每一次都会先让节点进行身份降级，然后再进行预投票操作**。但是，下面的 else 代码块中的代码，可能就会让大家感到疑惑了。虽然大家还没有真正学习领导者正式选举的流程，但是在上一章的 handlePreVoteResponse 方法，也就是处理预投票响应结果的方法中，如果发现当前节点预投票成功了，就会直接执行 electSelf 方法。就像下面代码块中展示的这样。请看下面代码块。

```java
//从响应中判断回复响应的节点是否给当前节点投票了
if (response.getGranted()) {
    //如果投票了，就在这里收集票数到计票器中
    this.prevVoteCtx.grant(peerId);
    //如果获得票数超过集群节点一半，就进入正式选举阶段
    if (this.prevVoteCtx.isGranted()) {
        //开始正式投票，下面这个方法还没有实现
        electSelf();
    }
}
```

在 electSelf 方法中，也就是在正式进入领导者选举后，肯定就会把当前节点的本地任期真正的自增 1 了。如果是这样的话，假如用户就把 this.raftOptions.isStepDownWhenVoteTimedout() 的值定义为 false，这样一来，如果当前节点的状态为候选者，并且本次领导者选举发生了选举超时，那么选举超时定时器工作的时候，就会直接进入 else 分支，执行 electSelf 正式选举领导者的方法了。假如这时候发生了网络分区，当前节点和集群中其他节点失去联系了，那该怎么办呢？可能会出现很多不确定的情况，但有一种情况一定可以确定，那就是本次领导者选举无法收到足够的票数，领导者选举失败，选举超时定时器会让当前候选者节点进入下一轮选举，但是 this.raftOptions.isStepDownWhenVoteTimedout() 的值定义为 false 了，这样程序在执行选举超时定时器中的 handleVoteTimeout 方法时，每次都会直接进入 else 代码块，直接进入领导者正式选举，将当前节点的任期自增 1，这样不就又会造成当前节点任期无限自增的情况吗？

好了，到此为止，我相信大家可能已经有些懵圈了，就算还没有懵圈的，可能也快懵圈了。请大家别急，因为这都在我预料之中，因为我还没有给大家正式讲解预投票之后的领导者选举流程，也就是没有为大家剖析 electSelf 方法的实现逻辑，一旦这个方法讲解完毕，到时候再把前面的知识串联一下，大家就会拨云见日，扫清迷雾了。

在正式开始讲解 electSelf 方法之前，让我再说几句，我知道这时候大家心里肯定会有一些疑问，脑子很乱，我也大概知道大家脑子为什么很乱，被哪些问题搅乱思路了。下面是我总结出的几个问题，请大家看看困扰你的问题是否是其中之一。

1 节点什么时候更改自己的状态为候选者？这个问题可能在上一章结束的时候就盘踞在大家脑海里了，因为在上一章我为大家分析 handleElectionTimeout 方法的操作流程时，讲到了在三个要进行的操作，那就是首先将当前节点状态更新为候选者；其次是停止超时选举定时器工作，之前已经解释过了，候选者和领导者是不用启动自己的超时选举定时器的；第三个要执行的操作就是向其他节点发送请求，等待其他节点投票。只不过我们把第三个操作更换为了预投票，在预投票成功之后，再正式进入领导者选举阶段。但是呢？在上一章实现的 handleElectionTimeout 方法中，当前节点已经触发了超时选举了，可在 handleElectionTimeout 方法中并没有将当前节点的状态更新为候选者状态，也没有终止超时选举定

时器工作，只是在方法的末尾调用了 preVote 方法，开始进行预投票操作。所以，现在我提出第一个问题，那就是触发了超时选举的节点什么时候会把自己的状态从跟随者更新为候选者呢?

2 第二个问题其实在第一个问题中已经提到了，那就是触发了超时选举的节点，在哪里将自己的超时选举定时器关闭呢？上一章我就为大家就分析了这个流程，但一直没有实现。

3 节点的选举超时定时器什么时候启动，注意，是选举超时定时器，并不是超时选举定时器。

4 为什么在选举超时定时器中可以直接进行正式选举？当然，这种情况有一个前提，那就是用户将 stepDownWhenVoteTimedout 设置为 false 了。这个 stepDownWhenVoteTimedout 是 RaftOptions 中的一个成员变量，设置为 true 意味着当节点选举超时之后，当前候选者节点需要进行身份降级，然后重新进行预投票操作，预投票成功后，才可以进入下一轮领导者正式选举。在我们的程序中，这个 stepDownWhenVoteTimedout 成员变量默认为 true。

5 节点的选举超时定时器什么时候应该终止？这也是一个很关键的问题，请大家想一想，选举超时定时器是不是只为候选者节点服务的？因为只有候选者节点在进行领导者选举，已经当选为领导者的节点或者状态为跟随者的节点，根本就不需要进行领导者选举，也就没有必要启动这个选举超时定时器。所以，我们需要理清楚这个选举超时定时器在什么时候终止工作。

只要解开以上 5 个谜团，我相信大家的困惑就全都会消除了。而要想解开这 5 个谜团，就要从领导者正式选举说起了。所以，接下来，就让我来为大家分析一下领导者正式选举的流程，实现 electSelf 方法。

实现领导者正式选举功能

在我们要实现 electSelf 方法之前，让我们先想一想，**当一个节点要正式进入领导者选举阶段了，应该先把这个节点的状态从跟随者更新为候选者呢？还是应该先启动选举超时定时器呢？还是应该先终止超时选举定时器工作呢**？对吧，首先应该分析清楚这些操作的执行顺序，在上一章的 handleElectionTimeout 方法中分析的几个操作，我们只实现了预投票操作，**还剩下将节点自己的状态更新为候选者状态，终止超时选举定时器工作这两个操作没有进行，还有本章第一小节讲解的选举超时定时器，我们也要弄清楚什么时候应该启动这个选举超时定时器**，所以，我们目前要明确的就是这三个操作的执行顺序。

当然，三个操作的执行顺序也已经有定论了，在 sofajraft 框架中已经实现好了。但是我们也可以从现有的代码中找到蛛丝马迹。比如说在选举超时定时器 voteTimer 的 handleVoteTimeout 方法中，我们可以看到下面这样的代码，请看下面代码块。

```java
private void handleVoteTimeout() {
    //判断当前节点状态是否为候选者，只有候选者才会出现投票超时的情况，也就是在一定时间内还没有成为主节点
    //也就是说一个跟随者节点触发了超时选举之后，先进行预投票操作，这个预投票操作通过了，然后进入正式的领导者选举
    //但这时候发生了网络分区，或者是集群中出现了多个候选者，选举一直没有成功，就会一直让候选者节点的选举超时定时器工作
    if (this.state != State.STATE_CANDIDATE) {
        return;
    }


    //省略部分内容

}
```

在上面的代码块中，也就是在选举超时定时器执行选举超时操作时有一个判断，**如果当前节点的状态并不是候选者，那就直接退出当前方法，并不会执行接下来的选举超时操作。这就意味着选举超时定时器肯定是在候选者节点中工作的，也就意味着肯定是当前节点的状态更新为候选者之后，才启动了选举超时定时器。如果在跟随者状态下就启动选举超时定时器，这个选举超时定时器也不会正常发挥作用**。由此可以推断出，**肯定是先把节点的状态更新为候选者状态，然后再启动选举超时定时器的**。

那么超时选举定时器是什么时候终止工作的呢？这个就直接看代码就行了，毕竟我们也是按照 sofajraft 框架的编写思路来分析的。在 sofajraft 框架源码中是这么做的：**首先将超时选举定时器停止了，然后将节点状态更新为了候选者状态，最后启动了当前节点的选举超时定时器**。当然，这些逻辑都是在 electSelf 方法中实现的。electSelf 也就是正式进行领导者选举的方法，在该方法中，当前候选者节点会向其他节点发送索要投票的请求，然后等到响应，如果票数足够，就意味着当前节点可以成功当选领导者了。

接下来似乎就该给大家具体展示一下 electSelf 方法了，虽然向集群中其他节点发送请求的流程还没有讲解，但是在上一章我已经为大家分析了预投票的整个流程，而且也跟大家说了，预投票的流程和正式选举领导者的流程没什么区别。只不过在预投票中并不会真正改变节点当前的任期，只是在要发送的请求中把任期加 1。而在正式领导者选举流程中，则会真正将节点的任期加 1，然后封装到请求中发送给集群中的其他节点。这些知识没什么新意了，所以，是时候向大家直接展示 electSelf 方法本身了。

但是，我这个人并不怕麻烦，在具体展示该方法之前，我很乐意再为大家分析一下，这个方法的具体流程。**在 electSelf 方法中，首先会将当前节点的超时选举定时器停止了，然后将当前节点的状态更新为候选者状态，接着将当前节点的任期自增 1，之后启动当前候选者节点的选举超时定时器，接下来就和预投票的操作就差不多了，就可以初始化正式领导者选举的投票结果计数器了，然后就可以将当前节点自增过的任期，以及本地最新日志的任期和索引都封装到请求中，发送给集群中的其他节点**。最后，还有一

点要补充，也是非常重要的一点，之前我就给大家分析过了，**在 raft 集群中，节点在当前任期中只能有一次投票机会，如果在当前任期中，节点已经投过票了，那么就不能再给其他节点投票。这些投票信息又被我们定义为节点的元数据信息，元数据信息每次更新都要进行本地持久化。而候选者节点肯定会把自己的票数投给自己，所以在把索要投票的请求发送给集群中的其他节点之后，紧接着就给自己投一票，然后把投票信息和当前任期，也就是元数据信息进行本地持久化。**这就是 electSelf 方法的完整流程。接下来，就请大家看看下面的代码块。

electSelf 方法也定义在 NodeImpl 类中，当然，因为现在要进行正式领导者选举了，所以 NodeImpl 又引入了一个新的成员变量，那就是正式选举的投票计数器，也就是 voteCtx 成员变量。

NodeImpl Java

```java
public class NodeImpl implements Node{

    private static final Logger LOG = LoggerFactory.getLogger(NodeImpl.class);


    //当前节点所在集群的Id
    private final String groupId;

    //当前节点的Id
    private NodeId nodeId;

    //当前节点的状态
    private volatile State state;

    //节点的当前任期
    private long currTerm;

    //当前节点投过票的节点的PeerId，这个成员变量可以记录下来，当前节点为哪个节点投过票
    private PeerId votedId;

    //当前节点记录的领导者的PeerId
    private PeerId leaderId = new PeerId();

    //当前节点自己的节点信息
    private final PeerId serverId;

    //集群当前生效的配置信息，包括旧的配置信息
    private ConfigurationEntry conf;

    //为jraft框架提供各种服务的工厂，在第一版本中，这个工厂只提供了元数据存储器服务
    private JRaftServiceFactory serviceFactory;

    //注意，这里多出来一个NodeOptions成员变量，节点需要的配置参数对象
    private NodeOptions options;

    //超时选举定时器
    private RepeatedTimer electionTimer;

    //选举超时定时器
    private RepeatedTimer voteTimer;

    //元数据存储器
```

```
43      private RaftMetaStorage metaStorage;
44
45      //计算预投票结果的计数器
46      private final Ballot prevVoteCtx = new Ballot();
47
48      //计算正式选票结果的计数器
49      private final Ballot voteCtx = new Ballot();
50
51      //省略其他内容
52
53
54      //正式发起选取的方法，在正式发起选举的方法中，就有些限制了，最关键的就是集群中的每一个raft节点只允许投一票
55      //该限制的具体实现逻辑就在本方法中，并且该方法的大部分逻辑和预投票的逻辑差不多
56      private void electSelf() {
57
58          //记录开始正式投票的日志
59          LOG.info("Node {} start vote and grant vote self, term={}.", getNodeId(), this.currTerm);
60          //根据当前节点的ID去配置信息中查看当前节点是为集群中的节点，如果不是集群中的节点就直接退出
61          if (!this.conf.contains(this.serverId)) {
62              LOG.warn("Node {} can't do electSelf as it is not in {}.", getNodeId(), this.conf);
63              return;
64          }
65          //如果当前节点目前是跟随者，就停止超时选举定时器工作，超时选举定时器是用来进行选举的，但是当前的节点正在进行选举工作
66          //就没必要让这个定时器工作了，等选举工作结束之后再启动，如果选举迟迟没有结果，会有另一个定时器处理的，也就是选举超时定时器
67          if (this.state == State.STATE_FOLLOWER) {
68              LOG.debug("Node {} stop election timer, term={}.", getNodeId(), this.currTerm);
69              //在这里停止超时选举定时器工作
70              this.electionTimer.stop();
71          }
72          //既然正在选举，肯定是没有领导者了，所以在进行选举时把当前节点的领导者节点设置为空
73          //这里的PeerId.emptyPeer()就是一个无效的节点ID，会被赋值给当前节点的领导者ID
74          resetLeaderId(PeerId.emptyPeer(), new Status(RaftError.ERAFTTIMEDOUT,
75                  "A follower's leader_id is reset to NULL as it begins to request_vote."));
76
77          //将当前节点状态更改为候选者
78          this.state = State.STATE_CANDIDATE;
```

```
79
80          //递增当前节点的任期
81          this.currTerm++;
82
83          //当前节点要为自己投票，所以把为谁投票的ID设置为自己的服务器ID
84          this.votedId = this.serverId.copy();
85
86          //这里还会记录一条日志，表示要启动投票超时计时器了
87          LOG.debug("Node {} start vote timer, term={} .", getNodeId(), this.currTerm);
88
89          //启动选举超时定时器
90          this.voteTimer.start();
91
92          //初始投票计数器，这里初始化的是正式投票的计数器
93          this.voteCtx.init(this.conf.getConf(), this.conf.isStable() ? null : this.conf.getOldConf());
94
95          //这里仍然写死了，获得当前节点最后一个日志的ID
96          //源码是这样的：final LogId lastLogId = this.logManager.getLastLogId(true);使用日志管理器来获得最后一条日志信息
97          final LogId lastLogId = new LogId(0,0);
98
99          //遍历配置中的节点
100         for (final PeerId peer : this.conf.listPeers()) {
101             //如果发现这个节点就是当前节点，就跳过这次玄幻
102             if (peer.equals(this.serverId)) {
103                 continue;
104             }
105             //连接遍历到的这个节点，如果不能成功连接，就记录日志
106             if (!this.rpcService.connect(peer.getEndpoint())) {
107                 LOG.warn("Node {} channel init failed, address={}.", getNodeId(), peer.getEndpoint());
108                 continue;
109             }
110             //创建一个回调对象，这个对象中实现了一个回调函数，该回调函数会在接收到正式投票请求的响应后被回调
111             final OnRequestVoteRpcDone done = new OnRequestVoteRpcDone(peer, this.currTerm, this);
112             //创建正式投票请求
113             done.request = RpcRequests.RequestVoteRequest.newBuilder()
114                     //这里设置成false，意味着不是预投票请求
115                     .setPreVote(false)
116                     .setGroupId(this.groupId)
117                     .setServerId(this.serverId.toString())
118                     .setPeerId(peer.toString())
119                     .setTerm(this.currTerm)
```

```
120                         .setLastLogIndex(lastLogId.getIndex())
121                         .setLastLogTerm(lastLogId.getTerm())
122                         .build();
123                 //发送请求
124                 this.rpcService.requestVote(peer.getEndpoint(), done.reques
    t, done);
125             }
126             //因为当前节点正在申请成为领导者，任期和投票记录有变更了，所以需要持久化一下
127             //这里投票记录就是当前节点自己
128             //进行节点元数据信息持久化
129             this.metaStorage.setTermAndVotedFor(this.currTerm, this.serverI
    d);
130             //在这里给当前节点投票
131             this.voteCtx.grant(this.serverId);
132
133             //如果获取的投票已经超过半数，则让当前节点成为领导者
134             if (this.voteCtx.isGranted()) {
135                 //这个就是让当前节点成为领导者的方法
136                 //该方法暂时没有实现，可以先忽略
137                 becomeLeader();
138             }
139
140         }
141
142
143
144         //省略其他方法
145
146     }
```

以上就是 electSelf 方法的具体实现，代码中的注释非常详细，并且前面也给大家详细分析过该方法的具体实现了，所以就不再重复讲解了。到此为止，我相信在上一小节提出的 5 个问题里，前 3 个问题肯定已经解决了，这时候大家应该都弄清楚了。接下来，我们就应该看看第 4 个问题，**为什么在选举超时定时器中可以直接进行正式选举。**这个问题其实并不复杂，其实现在就可以为大家讲解了，但是，我想先把领导者正式选举的流程全部展示给大家了，然后再回过头为大家分析这个问题。

现在我们只是实现了领导者正式选举的第一步，那就是让候选者节点发送索要投票的请求给集群中的其他节点，当其他节点接收到请求之后，该怎么处理这个请求呢？其实整个流程和处理预投票请求的流程差不多：

1 **首先也要分析接收到请求的节点是否处于活跃状态。**

2 **其次就是判断发送请求过来的这个候选者节点的 IP 地址和端口号是否可以解析成功。**

3 **接下来判断请求中的任期是否大于等于当前节点的任期。如果大于，就意味着是正常情况，比如说集群中有一个候选者，将自己的任期加 1 发送给其他节点了，当前收到请求的节点为跟随者，当前节点的**

任期肯定小于请求中的任期，这时候，当前节点就要执行 stepDown 方法。这里可能会有朋友感到疑惑，明明当前节点已经是跟随者状态了，为什么还要执行 stepDown 方法进行身份降级呢？我想简单解释一下，这里我只是给大家把正常的情况列举出来了，假如当前集群中有两个候选者呢？经过了上一轮的选举没有选出领导者，其中一个候选者进入下一轮了，将自己的任期加 1 然后发送给集群中其他节点，当前节点为候选者节点，发现集群中有另一个候选者节点的任期比自己大，那就执行身份降级，重新成为跟随者吧。根据这种情况，也可以衍生出另一种情况，那就是请求中任期和当前收到请求节点的任期相等，这时候应该怎么办呢？如果出现任期相等的情况，那很可能也是集群中出现了两个候选者，候选者把请求投票的信息发送给彼此了，也有可能是其他的情况。当然，任期相等还会继续判断，先判断本地日志是否比请求中的日志新，再判断当前节点有没有投过票，如果在当前任期中已经投过票了，就不会再给发送请求的节点投票了，如果没有投过票，就会给发送请求的节点投票。

4 最后就是根据一系列判断，最后回复给发送请求过来的节点一个响应，当然，在回复请求之前，还要做一件事，那就是如果当前节点给发送请求的节点投票了，也要将投票信息，也就是节点元数据进行本地持久化。

以上就是处理正式投票请求的全部流程，这些操作也都定义在一个方法中了，在 sofajraft 框架中，这个方法的名字为 handleRequestVoteRequest，该方法也定义在了 NodeImpl 类中，具体实现请看下面代码块。

NodeImpl Java

```java
public class NodeImpl implements Node{

    private static final Logger LOG = LoggerFactory.getLogger(NodeImpl.class);


    //元数据存储器
    private RaftMetaStorage metaStorage;

    //计算预投票结果的计数器
    private final Ballot prevVoteCtx = new Ballot();

    //计算正式选票结果的计数器
    private final Ballot voteCtx = new Ballot();

    //省略其他内容


    //当接收到候选者节点发送过来的正式所要投票的请求时，就会调用这个方法处理，正式投票的请求
    //就是该方法中的参数对象
    @Override
    public Message handleRequestVoteRequest(final RpcRequests.RequestVoteRequest request) {


        //检查当前节点是否处于活跃状态
        if (!this.state.isActive()) {
            LOG.warn("Node {} is not in active state, currTerm={}.", getNodeId(), this.currTerm);
            return RpcFactoryHelper.responseFactory()
                    .newResponse(RpcRequests.RequestVoteResponse.getDefaultInstance(), RaftError.EINVAL,
                            "Node %s is not in active state, state %s.", getNodeId(), this.state.name());
        }
        //创建一个PeerId对象，这个对象是用来解析候选者节点信息的
        final PeerId candidateId = new PeerId();
        //解析候选者节点信息
        if (!candidateId.parse(request.getServerId())) {
            LOG.warn("Node {} received RequestVoteRequest from {} serverId bad format.", getNodeId(),
                    request.getServerId());
            return RpcFactoryHelper.responseFactory()
```

```
38                      .newResponse(RpcRequests.RequestVoteResponse.getDefau
    ltInstance(), RaftError.EINVAL,
39                              "Parse candidateId failed: %s.", request.getS
    erverId());
40          }
41
42          do {//进入伪循环
43              //首先判断候选者的任期是否大于等于当前节点的任期
44              if (request.getTerm() >= this.currTerm) {
45                  //记录日志
46                  LOG.info("Node {} received RequestVoteRequest from {}, te
    rm={}, currTerm={}.", getNodeId(),
47                          request.getServerId(), request.getTerm(), this.cu
    rrTerm);
48                  //如果候选者节点任期较大，责让当前节点身份降级，成为跟随者
49                  //这里也能进一步想到，如果候选者节点任期和当前节点相等，那就不会让当
    前节点身份降级
50                  //很可能一个集群中同时有两个候选者呀，任期也一样，谁也不必改变谁的，
    只等着领导者选出来
51                  //让领导者给候选者发送信息，让候选者身份降级就行
52                  //当然，当前节点假如是候选者的话，就不会给发送请求过来的候选者投票了
53                  //因为只能投一票，这一票已经投给自己了
54                  if (request.getTerm() > this.currTerm) {
55                      //这里调用了stepDown方法，stepDown方法的逻辑已经讲解过了，就
    不再细说了
56                      stepDown(request.getTerm());
57                  }
58              } else {
59                  //走到这里意味着候选者任期小于当前节点，直接忽略该请求即可
60                  LOG.info("Node {} ignore RequestVoteRequest from {}, term
    ={}, currTerm={}.", getNodeId(),
61                          request.getServerId(), request.getTerm(), this.cu
    rrTerm);
62                  break;
63              }
64
65
66              //获取当前节点最后一条日志
67              //final LogId lastLogId = this.logManager.getLastLogId(true);
68              //校验候选者节点的日志是否比当前节点最后一条日志新
69              //final boolean logIsOk = new LogId(request.getLastLogIndex
    (), request.getLastLogTerm())
70              //        .compareTo(lastLogId) >= 0;
71              //这里的true也是写死的，本来是上面的哪个logIsOk变量，如果为true，并且
    当前节点还没有投过票
72              //那就给候选者投票。这里写死的代码后面版本都会重构
73
```

```
                //这里还判断了当前节点是否已经给其他节点投过票了，这里大家可以看看上一章的内容，在stepDown方法中，更新当前节点的任期时，也会把
                //当前节点的投票记录重置，这就代表着当前节点在新的任期中还没有进行过投票
                //当然，这种情况必须是当前节点的任期小于请求中的任期才适用，如果当前节点的任期和请求中的任期相等，那就不会重置当前节点的votedId
                //直接看看当前节点是否在当前任期中投过票即可
                if (true && (this.votedId == null || this.votedId.isEmpty())) {
                    //在这里把当前节点的状态改为跟随者，这里之所以又调用stepDown方法，是因为假如当前节点的任期和请求中的任期相等，但是
                    //当前节点没投过票，并且日志没有请求中的心，程序就不会进入上面if分支中，执行stepDown方法，而是会执行到这里，再执行stepDown方法
                    stepDown(request.getTerm());
                    //记录当前节点的投票信息
                    this.votedId = candidateId.copy();
                    //元数据信息持久化
                    this.metaStorage.setVotedFor(candidateId);
                }
            } while (false);

            //这里构建响应对象的时候，会调用setGranted方法，这个方法的参数是一个bool，也就是是否投票的意思
            //如果经过上面的一系列操作，发现候选者节点的任期和当前节点任期相等了，并且候选者节点的信息和当前节点的投票记录
            //也相等了，给setGranted方法传递的参数就为true，也就代表着投票成功
            return RpcRequests.RequestVoteResponse.newBuilder()
                    .setTerm(this.currTerm)
                    //request.getTerm() == this.currTerm，这里大家可能会疑惑，为什么要再次判断一下请求中的任期是否和节点的当前任期是否相等
                    //可以先保留疑惑，再最后一小节，我就会为大家讲解
                    .setGranted(request.getTerm() == this.currTerm && candidateId.equals(this.votedId))
                    .build();

        }


        //省略其他方法

    }
```

以上就是 handleRequestVoteRequest 方法的整体流程。注释已经非常详细了，所以我不会再重复讲解方法的执行流程，我唯一想补充的就是在上面代码块的第 95 行，可以看到在 setGranted 方法的时候，执行了 request.getTerm() == this.currTerm 这样一行代码。 setGranted 方法的参数是一个 bool，如果设置为 true 就代表着当前节点回复个发送请求的节点一个成功投票的响应，如果设置为 false，就意味着没有成功投票。**但是为什么在设置是否成功投票的时候，需要判断请求中的任期和当前节点的任期**

是否相等呢？如果当前节点的任期小于请求中的任期，也会在 stepDown 方法中将自己的任期更新为请求中的任期，如果当前节点的任期等于请求中的任期，那么当前节点的任期也不会变化，到最后肯定也和请求中的任期相等，为什么在设置投票是否成功时，还要再进行一次判断呢？大家可以暂时保留这个疑问，在本篇文章的最后一小节，我将为大家分析这个问题。

好了，发送正式投票请求的方法和处理正式投票请求的方法都讲解完毕了，接下来该讲解处理正式投票响应的方法了。这个方法没什么可讲的，和处理预投票请求的方法没什么两样，无法就是收到足够票数之后，就可以成为领导者了。我把这个方法定义为了 handleRequestVoteResponse，当然，也定义在 NodeImpl 类中了。具体实现请看下面代码块。

NodeImpl Java

```java
public class NodeImpl implements Node{

    private static final Logger LOG = LoggerFactory.getLogger(NodeImpl.class);


    //元数据存储器
    private RaftMetaStorage metaStorage;

    //计算预投票结果的计数器
    private final Ballot prevVoteCtx = new Ballot();

    //计算正式选票结果的计数器
    private final Ballot voteCtx = new Ballot();

    //省略其他内容



    //如果当前节点的身份是候选者，发送了索要投票的请求给其他节点后，收到其他节点的响应后就会调用这个方法
    //处理响应，这个方法中term参数代表的是当前候选者节点发送索要投票请求之前的任期，这个逻辑可以从OnRequestVoteRpcDone类的run方法中查看
    public void handleRequestVoteResponse(final PeerId peerId, final long term, final RpcRequests.RequestVoteResponse response) {



        //当前候选者节点接收到响应之后，发现自己的身份不是候选者了，就直接退出该方法
        //也许在接收到投票响应之前已经收到了集群中选举出来的领导者的信息，这样就会把自己的身份降级为跟随者了
        if (this.state != State.STATE_CANDIDATE) {
            LOG.warn("Node {} received invalid RequestVoteResponse from {}, state not in STATE_CANDIDATE but {}.",
                    getNodeId(), peerId, this.state);
            return;
        }

        //判断当前候选者节点的任期是否发生了变化，这里的这个term代表的是当前候选者节点发送索要投票请求之前的任期
        //而在发送索要投票请求之后，收到响应之前，很可能任期就被新的领导者改变了，如果前后任期不一致，也直接退出该方法
        if (term != this.currTerm) {

```

```
            LOG.warn("Node {} received stale RequestVoteResponse from {},
term={}, currTerm={}.", getNodeId(),
                    peerId, term, this.currTerm);
            return;
        }

        //如果接收到响应后发现请求中的任期，也就是回复响应的节点的任期比自己大
        //也直接退出该方法，并且让自己成为跟随者
        if (response.getTerm() > this.currTerm) {
            LOG.warn("Node {} received invalid RequestVoteResponse from
{}, term={}, expect={}.", getNodeId(),
                    peerId, response.getTerm(), this.currTerm);
            //身份降级
            stepDown(response.getTerm(), false, new Status(RaftError.EHIGH
ERTERMRESPONSE,
                    "Raft node receives higher term request_vote_respons
e."));
            return;
        }

        //从请求中判断是否收到了投票
        if (response.getGranted()) {
            //如果收到了投票就把来自peerId节点的投票收集到计票器中
            this.voteCtx.grant(peerId);
            //使用投票计算器计算当前节点收到的投票是否超过了集群中半数节点
            if (this.voteCtx.isGranted()) {
                //超过了就让当前节点成为领导者
                //该方法暂时没有实现
                becomeLeader();
            }
        }

    }


    //省略其他方法

}
```

这个 handleRequestVoteResponse 方法同样被包装在 NodeImpl 的一个内部类中，这个内部类的方法会在发送请求的节点收到响应后被回调。具体实现请看下面代码块。

NodeImpl Java

```java
public class NodeImpl implements Node{

    private static final Logger LOG = LoggerFactory.getLogger(NodeImpl.class);


    //元数据存储器
    private RaftMetaStorage metaStorage;

    //计算预投票结果的计数器
    private final Ballot prevVoteCtx = new Ballot();

    //计算正式选票结果的计数器
    private final Ballot voteCtx = new Ballot();

    //省略其他内容


    //这个类的对象中封装着一个回调方法，该方法会在候选者节点接收到正式投票的响应之后被回调
    private class OnRequestVoteRpcDone extends RpcResponseClosureAdapter<RpcRequests.RequestVoteResponse> {

        final long         startMs;
        final PeerId       peer;
        final long         term;
        final NodeImpl     node;
        RpcRequests.RequestVoteRequest request;

        public OnRequestVoteRpcDone(final PeerId peer, final long term, final NodeImpl node) {
            super();
            this.startMs = Utils.monotonicMs();
            this.peer = peer;
            this.term = term;
            this.node = node;
        }

        @Override
        public void run(final Status status) {
            NodeImpl.this.metrics.recordLatency("request-vote", Utils.monotonicMs() - this.startMs);
            if (!status.isOk()) {

```

```
                LOG.warn("Node {} RequestVote to {} error: {}.", this.node.getNodeId(), this.peer, status);
            } else {
                this.node.handleRequestVoteResponse(this.peer, this.term, getResponse());
            }
        }
    }



    //如果当前节点的身份是候选者，发送了索要投票的请求给其他节点后，收到其他节点的响应后就会调用这个方法
    //处理响应，这个方法中term参数代表的是当前候选者节点发送索要投票请求之前的任期，这个逻辑可以从OnRequestVoteRpcDone类的run方法中查看
    public void handleRequestVoteResponse(final PeerId peerId, final long term, final RpcRequests.RequestVoteResponse response) {

        //内容省略

    }


    //省略其他方法

}
```

到此为止，我就为大家把领导者正式选举的流程全部实现了。大家可以自己再反复研究研究，争取把每一个细节都理清楚。接下来，我想为大家分析之前提出来的那个问题，也许大家还有印象，那就是**为什么在选举超时定时器中可以直接进行正式选举**，没错，是时候讲解这个问题了。

分析预投票 preVote 方法的作用

请大家跟我一起回顾一下，当一个节点触发了超时选举之后，会进行什么操作？肯定会让当前节点进行预投票操作，预投票操作通过之后，再进行领导者正式选举操作，这个应该是公认的流程了，对吧？那么预投票操作都做了什么呢？其实也没有什么，也没有改变当前节点的状态，也没有停止超时选举定时器工作，也没有启动选举超时定时器。这些工作都是在领导者正式选举中进行的，也就是在 electSelf 方法之中进行的。那么，不知道大家有没有想过，为什么这些操作要定义在 electSelf 方法中，而不是在预投票 preVote 方法中实现？理清楚了这个问题其实也就明白了 preVote 方法的真正作用，也就明白了为什么在选举超时定时器中可以直接进行正式选举。

让我们先来想一想，预投票操作是在哪里进行的？显然是在超时选举定时器中进行的，准确地说，是当前节点触发了超时选举之后，时间轮开始执行超时选举定时器中提交的定时任务，这个定时任务中有一个 handleElectionTimeout 方法，在该方法中进行了 preVote 预投票操作。就像下面代码块展示的这样。

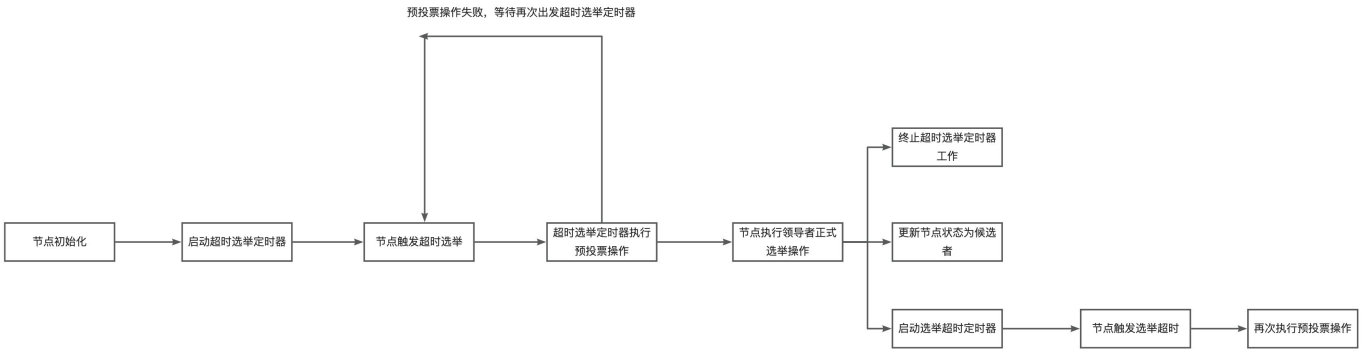
```java
//超时选举处理处理方法，这个方法就是超时选举定时器要调用的方法
private void handleElectionTimeout() {

    //判断当前节点的状态，如果当前节点不是跟随者的状态，就不能进行投票选举
    //因为当节点正式开始进行选举领导者操作时，会将自己的状态从跟随者更新为候选者
    //也就是说候选者是从跟随着转换过去的，如果连跟随者都不是，怎么转换成候选者呢?
    if (this.state != State.STATE_FOLLOWER) {
        return;
    }
    //这里的这个判断非常重要，超时选举定时器启动之后，只要节点状态是跟随着，就会一直工作
    //每过一段时间就会触发一次，但是触发了并不意味着就要真的去进行选举，会在这里进行判断
    //看看距离当前节点最后一次收到领导者信息过去的多少时间，如果这个时间超过了超时选举时间，才会进入预投票阶段
    //否则直接退出，这里返回true就代表还没超时呢
    if (isCurrentLeaderValid()) {
        return;
    }
    //既然是要超时选举了，说明领导者已经没有了，在这里对当前节点存放的领导者信息进行重置
    //这里我想再为大家解释一下，可能有朋友觉得已经在stepDown方法中执行了下面这个方法了
    //为什么在这里还要再执行一次呢？我想说的是，大家不要主观代入，不要很主观地就认为stepDown方法
    //和handleElectionTimeout方法存在强因果关系
    //请大家思考一下，假如一个跟随者节点一直在集群中正常工作，超时选举定时器也一直在启动着
    //过了一段时间没有收到领导者的信息，触发了超时选举，这时候超时选举定时器就会直接执行这个
    //handleElectionTimeout方法了，所以就要在这里重置当前节点记录的领导者信息
    resetLeaderId(PeerId.emptyPeer());
    //开始预投票
    preVote();

}
```

这时候，我们其实也可以意识到，说到底，在超时任务定时器中执行的这个 handleElectionTimeout 方法，其内部其实也只是进一步调用了 preVote 方法，之后就并没有什么直接操作了。只有当预投票操作成功之后，才能进入领导者正式选举阶段。而当节点触发了超时选举定时任务，执行 preVote 方法时，当前节点仍然为跟随者状态，假如当前节点是自己发生了网络分区，也就是跟集群中其他节点都失去了

联系，那么当前跟随者节点的预投票操作就一直不会成功，当前跟随者节点就会一直触发自己的超时选举定时器，也就是说，假如当前节点发生了网络分区，从而触发了超时选举，这时候，这个节点在执行 preVote 预投票操作时就会失败，当前节点的状态仍然为跟随者状态，超时选举定时器仍然在按时工作。只有当前节点预投票操作成功了，才能进入 electSelf 方法， 开始正式进行领导者选举，而在 electSelf 方法中，当前节点的状态会更新为候选者，超时选举定时器会被关闭，选举超时定时器会启动。这才是一个节点在触发超时选举定时器后完整的工作流程，上一章和本上前半部分的讲解并不严谨，现在我才为大家把完整的流程展示出来了。

而这个 preVote 预投票方法的作用也很明显了，其实就是用来判断当前节点是不是和集群中其他节点发生了严重的网络分区，所谓严重的网络分区，表现结果就是不能收到超过集群过半节点的票数。换句话说，只要 preVote 操作成功了，在 sofajraft 框架中其实就可以默认当前节点并没有发生严重的网络分区，集群重新选举领导者，可能是因为旧的领导者自己网络分区，或者是故障宕机了。所以，在选举超时定时器中，可以在某些条件下直接执行 electSelf 方法，而不必再次进行预投票操作，因为选举超时定时器是在 electSelf 方法中启动的，既然可以执行到 electSelf 方法，就意味着当前节点肯定成功执行了预投票操作，这也就意味着并没有发生严重的网络分区。至于为什么本轮领导者选举没有成功，触发了选举超时定时器，很可能是因为集群中出现了多个候选者。**所以，在选举超时定时器中，确实可以在某些条件下直接执行 electSelf 方法，而不必再次进行预投票操作**。

当然，如果为了追求严谨，我们确实可以认为就算预投票操作通过，也只能表明当前节点在那一刻没有发生严重的网络分区，并不能保障在之后的操作中不会发生网络分区，也许之后进入领导者正式选举时就发生严重网络分区了，和集群中其他节点失去联系了。如果是这种情况，肯定会触发选举超时定时器工作，这个定时器每一次都直接执行 electSelf 方法，不就又会让当前节点的任期无限自增了吗？为了避免这种情况，在 sofajraft 框架中才定义了一个 stepDownWhenVoteTimedout 配置参数，这个配置参数默认为 true，这样一来，超时选举定时器每一次执行任务时，仍然会先执行预投票操作，等待预投票成功之后再正式开始领导者选举。到此为止，我相信大家才会真正理解选举超时定时器中的细节操作。我为大家准备了一张简图，大家可以看一看，加深对节点选举的完整流程的理解。



好了，流程图也展示完毕了，到此为止，本章的重要内容其实就讲解完了。当然，之前提出来的 5 个问题中的第 5 个问题还没有讲解，就是**节点的选举超时定时器什么时候应该终止**，这个问题其实在前面也分析过了，领导者和跟随者根本不需要进行领导者选举，所以也就不必启动这个选举超时定时器。所以，当一个节点身份降级为跟随者的时候，就需要终止这个选举超时定时器；而当一个节点成为领导者

时，也需要终止这个超时选举定时器。我这么一分析，大家可能都有思路了，其实就是在 stepDown 方法和 becomeLeader 方法中终止选举超时定时器工作即可，当然，这个 becomeLeader 方法还没有为大家实现，下一章我就会为大家实现这个方法了。更具体的细节我就不再展开了，说实话，这个选举超时定时器本来并没有被我引入到第一版本代码中，我是在写文章的过程中发现这个知识绕不过去，临时添加上的。在我为大家提供的第一版本代码中，虽然展示出了选举超时定时器的代码实现，但是并没有真正启动或者终止它。因为我不想一开始就把代码实现得这么复杂。实际上，在 sofajraft 框架中还有一些定时器我都没讲解，我的计划是在我为大家完整实现了 sofajraft 框架之后，在最后一个版本，也就是第 13 版本代码中，把各种定时器补充完善。那个时候，大家理解起来就会毫无压力了。所以这个选举超时定时器也会在第 13 版本代码中真正展示给大家。现在大家在文章中了解这个定时器即可。

接下来，还有最后一个小问题需要和大家探讨一下，那就是程序中可能会出现的并发问题。

分析程序中的并发问题，引入读写锁

请大家想一想，在我们的程序中会不会出现可以预见的并发问题呢？当然，要分析并发问题，肯定就要先明确哪些地方会发生并发问题。很简单，多个线程同时操作一些成员变量，就有可能发生并发问题。那在我们的程序中，什么时候会出现多线程同时操作成员变量的情况呢？在我们文章目前展示的代码中，似乎还没有发现这种情况。但是，我必须得解释一下，没有发生这种情况是因为我没有讲解这种情况，并且把加锁的代码都删去了，实际上，在第一版本代码中就引入了同步锁。

请大家想一想，现在文章中展示的这些代码，涉及到了什么操作，一个 Node 节点，其实现类是 NodeImpl ，在这个 NodeImpl 类中定义了很多和节点相关的方法，这个节点既可以向集群中的其他节点发送消息，同时也可以接收集群中其他节点发送过来的消息。因为这一个节点中既存在客户端又存在服务器。**一个程序中既部署了客户端同时又部署了服务器，我想这一点应该也是容易理解的，而且也并不矛盾**。

这样一来，就导致程序中会出现这样一种局面：**当前节点假如刚要触发超时选举的时候，也就是执行 handleElectionTimeout 方法的时候，当前节点即将在 handleElectionTimeout 方法中进一步调用预投票方法，然后是 electSelf 方法，将自己状态更新为候选者，然后将自己的任期加 1，向集群中其他节点发送索要投票的请求，可是就在要执行 handleElectionTimeout 方法的时候，在此之前集群中已经有另一个节点触发超时选举了，因为当前节点也部署了服务器程序，所以当前节点此时接收到了另一个节点发送过来的索要投票的请求。这样一来，当前节点很可能就要同时执行两个方法，一个是 handleElectionTimeout 方法，一个是服务器中的业务线程池要执行的 handleRequestVoteRequest 方法。在这两个方法内部，都会修改本地的任期，以及节点的状态，并且在 handleElectionTimeout 方法内部的 electSelf 方法中，当前节点还会给自己投一票，然后更新节点元数据信息。而在**

handleRequestVoteRequest 方法中也会记录当前节点为谁投了票，也会更新节点的元数据信息。这不就是一个典型的多个线程同时操作成员变量的局面吗?

我们要做的，而且必须要做到的就是避免这种并发情况，原因很简单，如果当前节点在执行 handleRequestVoteRequest 方法的同时，也同时执行 handleElectionTimeout 方法，很可能出现当前节点已经在 handleRequestVoteRequest 方法中被更新为跟随者状态了，而在 handleElectionTimeout 的 electSelf 方法中又进行了领导者选举，这不就出错了吗？所以，我们必须保证 handleElectionTimeout 和 handleRequestVoteRequest 方法一先一后执行，可以让 handleRequestVoteRequest 先执行，也可以让 handleElectionTimeout 先执行，就是不能让两个方法同时执行。如果 handleElectionTimeout 先执行，那么当前节点就会在 handleElectionTimeout 的 electSelf 方法中把票投给自己，更改自己的任期，之后执行 handleRequestVoteRequest 方法的时候就不会投票了，节点元数据信息也不会再更新；如果先执行 handleRequestVoteRequest 方法，在该方法中会将当前节点的 lastLeaderTimestamp 时间戳更新，之后再执行 handleElectionTimeout 方法时，就会判断当前节点是不是触发了超时选举，这时候显然没有触发，所以当前节点也就不会再执行超时选举的操作了。这样分析下来，为我们自己实现的框架引入同步锁就是必须的了。

实际上，在 sofajraft 框架中并不是直接粗暴地使用了 synchronized 来解决并发问题，而是引入了读写锁，因为在程序运行中也有大量的读操作，比如可能出现多个线程同时读取同一个成员变量的情况，读锁是可以共享的，所以只需要获得读锁即可。但是读锁和写锁是互斥的，当有线程正在读取资源，这时候是不允许修改资源的。当然，写锁本身就是个独占锁，当修改某些资源时，这些资源不能被其他线程修改或读取。总之，引入读写锁可以降级锁力度，提高程序并发性能，但又不至于出现并发问题。就拿我刚才分析的两个方法来举例吧，当引入读写锁后，这两个方法会被重构成下面这样。具体实现请看下面代码块。

NodeImpl Java

```java
public class NodeImpl implements Node{

    private static final Logger LOG = LoggerFactory.getLogger(NodeImpl.class);

    //其他内容省略

    //提供读写锁功能的成员变量，这个读写锁虽然是jraft框架自己定义的，但本质是继承了jdk中的ReentrantReadWriteLock类
    private final ReadWriteLock readWriteLock = new NodeReadWriteLock(this);
    //写锁
    protected final Lock writeLock = this.readWriteLock.writeLock();
    //读锁
    protected final Lock readLock = this.readWriteLock.readLock();









    //超时选举处理处理方法，这个方法就是超时选举定时器要调用的方法
    private void handleElectionTimeout() {
        //设计一个是否释放锁的标志
        boolean doUnlock = true;
        //获得写锁，这里获得写锁是因为下面进行的一系列操作中对某些属性进行了状态或者值的变更
        //比如节点状态，也就是state，还有领导者节点是谁，也就是this.leaderId，这些成员变量的值都会在投票完成后进行变更，还有任期的值也会改变
        //但是在整个框架中，不止有超时选举定时器的线程会对这两个成员变量进行操作，还有另外的线程会对这些属性进行读取和变更
        //比如当前节点刚刚成为候选者，但是在整个集群中有另外的节点已经成为候选者，并且发送过来索要投票的消息了
        //如果这时候当前节点正在把票投给自己，因为当前节点也正在发起选举，肯定会先投自己一票，但是另外的节点发送索要投票的消息过来了
        //另外的线程在处理索要投票的消息时，肯定会判断当前节点是否已经投过票了，这时候一个线程正在修改投票记录，一个线程正在读取投票记录
        //很容易发生并发问题，所以需要使用写锁来保证并发安全，谁获得到了写锁，谁才能访问对应的属性
        //这里我还要再解释一下，jraft内部的通信也是依靠Netty来实现的，集群内部各个节点都是使用Netty的功能进行远程通信的
        //只不过是在Netty之上封装了一层，变成了一个bolt框架，关于这个框架的知识大家可以自己去查一查，简单看一看，说实话，我之前也没接触过这个框架
        //但是看懂代码还是不成问题的，既然底层使用的是Netty，IO逻辑使用的是Netty自己的单线程执行器，那么业务逻辑就要使用另外定义的线程池了
```

```
32          //就是因为有这个业务线程池，所以才会出现上面我为大家分析的情况
33          //从我刚才的分析中也能看到，为什么要把这个读锁在这里就加上
34          this.writeLock.lock();
35          try {
36              //判断当前节点的状态，如果当前节点不是跟随着的状态，就不能进行投票选举
37              //因为候选者是从跟随着转换过去的，连跟随者都不是，怎么转换成候选者呢?
38              if (this.state != State.STATE_FOLLOWER) {
39                  return;
40              }
41              //这里的这个判断非常重要，超时选举定时器启动之后，只要节点状态是跟随着，
    就会一直工作
42              //每过一段时间就会触发一次，但是触发了并不意味着就要真的去进行选举，会在
    这里进行判断
43              //看看距离当前节点最后一次收到领导者信息过去的多少时间，如果这个时间超过
    了超时选举时间，才会进入预投票阶段
44              //否则直接退出，这里返回true就代表还没超时呢
45              if (isCurrentLeaderValid()) {
46                  return;
47              }
48              //既然是要超时选举了，说明领导者已经没有了，在这里对当前节点存放的领导者
    信息进行充值
49              resetLeaderId(PeerId.emptyPeer(), new Status(RaftError.ERAFTT
    IMEDOUT, "Lost connection from leader %s.",
50                      this.leaderId));
51              //设置释放锁标识为false，这里设置成false是为了不在finally块中释放写锁
52              //因为写锁一旦走到这里，就意味着要进行预投票了，会进入下面的preVote方法
53              //写锁会在下面这个方法内释放
54              doUnlock = false;
55              //开始预投票，注意，只要进行预投票，就意味着这时候还持有者写锁
56              //这一点要记清楚
57              preVote();
58          } finally {
59              if (doUnlock) {
60                  this.writeLock.unlock();
61              }
62          }
63      }
64
65
66
67      //接下来是handleRequestVoteRequest方法
68      //当接收到候选者节点发送过来的正式所要投票的请求时，就会调用这个方法处理，正式投票
    的请求
69      //就是该方法中的参数对象
70      @Override
71      public Message handleRequestVoteRequest(final RpcRequests.RequestVote
    Request request) {
```

```
72          //定义一个是否释放写锁的标志
73          boolean doUnlock = true;
74          //获取写锁
75          this.writeLock.lock();
76          try {
77              //检查当前节点是否处于活跃状态
78              if (!this.state.isActive()) {
79                  LOG.warn("Node {} is not in active state, currTerm={}.", getNodeId(), this.currTerm);
80                  return RpcFactoryHelper.responseFactory()
81                          .newResponse(RpcRequests.RequestVoteResponse.getDefaultInstance(), RaftError.EINVAL,
82                                  "Node %s is not in active state, state %s.", getNodeId(), this.state.name());
83              }
84              //创建一个PeerId对象，这个对象是用来解析候选者节点信息的
85              final PeerId candidateId = new PeerId();
86              //解析候选者节点信息
87              if (!candidateId.parse(request.getServerId())) {
88                  LOG.warn("Node {} received RequestVoteRequest from {} serverId bad format.", getNodeId(),
89                          request.getServerId());
90                  return RpcFactoryHelper.responseFactory()
91                          .newResponse(RpcRequests.RequestVoteResponse.getDefaultInstance(), RaftError.EINVAL,
92                                  "Parse candidateId failed: %s.", request.getServerId());
93              }
94              do {//进入伪循环
95                  //首先判断候选者的任期是否大于等于当前节点的任期
96                  if (request.getTerm() >= this.currTerm) {
97                      //记录日志
98                      LOG.info("Node {} received RequestVoteRequest from {}, term={}, currTerm={}.", getNodeId(),
99                              request.getServerId(), request.getTerm(), this.currTerm);
100                     //如果候选者节点任期较大，责让当前节点身份降级，成为跟随者
101                     //这里也能进一步想到，如果候选者节点任期和当前节点相等，那就不会让当前节点身份降级
102                     //很可能一个集群中同时有两个候选者呀，任期也一样，谁也不必改变谁的，只等着领导者选出来
103                     //让领导者给候选者发送信息，让候选者身份降级就行
104                     //当然，当前节点假如是候选者的话，就不会给发送请求过来的候选者投票了
105                     //因为只能投一票，这一票已经投给自己了
106                     if (request.getTerm() > this.currTerm) {
107
```

```
                            //这里调用了stepDown方法，stepDown方法的逻辑已经讲解过了，就不再细说了
                            stepDown(request.getTerm());
                        }
                    } else {
                        //走到这里意味着候选者任期小于当前节点，直接忽略该请求即可
                        LOG.info("Node {} ignore RequestVoteRequest from {}, term={}, currTerm={}.", getNodeId(),
                                request.getServerId(), request.getTerm(), this.currTerm);
                        break;
                    }
                    doUnlock = false;
                    //释放写锁，这里之所以释放写锁，是因为源码中this.logManager.getLastLogId(true)这行
                    //代码可能有IO操作，阻塞线程，所以暂时把锁释放了，让其他线程可以获得写锁
                    this.writeLock.unlock();
                    //获取当前节点最后一条日志
                    //final LogId lastLogId = this.logManager.getLastLogId(true);
                    doUnlock = true;
                    //获取写锁
                    this.writeLock.lock();
                    //判断当前节点的任期是否和之前相等，防止在释放锁的期间，已经有领导者产生了，并且修改了当前节点的任期
                    if (request.getTerm() != this.currTerm) {
                        //不一致则直接退出循环
                        LOG.warn("Node {} raise term {} when get lastLogId.", getNodeId(), this.currTerm);
                        break;
                    }
                    //校验候选者节点的日志是否比当前节点最后一条日志新
                    //final boolean logIsOk = new LogId(request.getLastLogIndex(), request.getLastLogTerm())
                    //        .compareTo(lastLogId) >= 0;
                    //这里的true也是写死的，本来是上面的哪个logIsOk变量，如果为true，并且当前节点还没有投过票
                    //那就给候选者投票。这里写死的代码后面版本都会重构
                    //这里还判断了当前节点是否已经给其他节点投过票了，这里大家可以看看上一章的内容，在stepDown方法中，更新当前节点的任期时，也会把
                    //当前节点的投票记录重置，这就代表着当前节点在新的任期中还没有进行过投票
                    //当然，这种情况必须是当前节点的任期小于请求中的任期才适用，如果当前节点的任期和请求中的任期相等，那就不会重置当前节点的votedId
                    //直接看看当前节点是否在当前任期中投过票即可
```

```
141                     if (true && (this.votedId == null || this.votedId.isEmpty
142     ())) {
143                         //在这里把当前节点的状态改为跟随者
144                         stepDown(request.getTerm());
145                         //记录当前节点的投票信息
146                         this.votedId = candidateId.copy();
147                         //元数据信息持久化
148                         this.metaStorage.setVotedFor(candidateId);
149                     }
                    } while (false);
150                 //这里构建响应对象的时候，会调用setGranted方法，这个方法的参数是一个bo
        ol，也就是是否投票的意思
151                 //如果经过上面的一系列操作，发现候选者节点的任期和当前节点任期相等了，并
152     且候选者节点的信息和当前节点的投票记录
153                 //也相等了，给setGranted方法传递的参数就为true，也就代表着投票成功
154                 return RpcRequests.RequestVoteResponse.newBuilder()
                            .setTerm(this.currTerm)
155                         .setGranted(request.getTerm() == this.currTerm && can
156     didateId.equals(this.votedId))
157                         .build();
158             } finally {
159                 //释放写锁的操作
160                 if (doUnlock) {
161                     this.writeLock.unlock();
162                 }
163             }
164         }
165
166
167
168         //省略其他方法
```

在阅读了这两个方法的完整实现之后，现在大家应该知道了，在上面代码块的第 153 行，为什么要最后再判断一下 request.getTerm() == this.currTerm 了吧？就是因为在 handleRequestVoteRequest 中间会释放写锁，程序运行中可能会产生并发问题，所以要最后判断一下。至于为什么 handleRequestVoteRequest 方法在执行过程中会释放写锁，我把原因写在注释中了，请看上面代码块的第 117 行即可。好了，具体的读写锁的分析我就不展开讲解了，代码中添加的注释非常详细，大家可以自己去看一看。

到此为止，本章的内容就真正讲解完毕了。讲到现在，我想大家应该也能明白了，实际上我们实现的这个 raft 共识算法框架的所有内容在前三章就讲的差不多了，现在实现的这个 sofajraft 框架说白了只不过是一个严格遵循 raft 共识算法模式的 RPC 通信框架。实际上就是在简单的 RPC 的一问一答中，完成了领导者的选举。当然，这个框架底层的 RPC 机制我还没有在文章中为大家讲解，后面会专门录制视频讲解。

最后，我也应该承认，**上一章和这一章前半部分讲解确实不太严谨，有些逻辑在讲解的时候很模糊，甚至并不正确，但确实是没有办法， 因为之前还没有完全实现正式领导者选举，各种方法和操作都没有展示完全，所以我无法详细展开。比如说，我不可能在讲解预投票的时候，就告诉大家会在正式选举领导者的操作中关闭超时选举定时器，然后启动选举超时定时器，然后更改节点状态为候选者。那个时候大家连正式选举的操作是什么还没见到呢，我要是真这样讲了，大家肯定会更蒙圈。所以我只能一点点讲解，先绕个弯子，然后渐渐修正，最后给大家呈现正确的知识**。如果还有其他的不足之处，还请大家多多包涵。

当然，这一章仍然有一个方法没有实现，那就是节点成功当选举领导者后，应该执行的 becomeLeader 方法，该方法还没有实现。因为这是下一章的内容，在下一章中，我会为大家把这个方法实现了，然后引入一个新的概念，那就是复制器。一个节点一旦成为领导者之后，这个节点需要同时和集群中的其他所有节点通信，这么多节点该怎么管理呢？这时候就轮到复制器对象登场了。我们下一章见！