分析 NodeImpl 初始化流程，实现节点预投票功能(二)

大家好，在上一章的结尾，我为大家展示了一个简单的测试类，并且初步启动了测试类。在启动了测试类之后，整个程序执行到 NodeImpl 对象的 init 方法之后，就执行不下去了，因为我还没有为自己的程序实现这个方法。这一章，我就为大家分析一下在 NodeImpl 初始化的过程中，需要初始化什么组件，然后为大家实现它的 init 方法。不过大家也不用心急，这一章才刚刚开始，先让我们做一个热身运动，对上一章的知识做一个简单的补充。在上一章的最后一个小节，我为大家展示了一个测试类，简单模拟了一下程序启动的流程。我把那个测试类搬运到这里了，请大家再回顾一下。请看下面代码块。

ServerTest Java

```java
//测试类，可以直接去看main方法，然后再回过头看看ServerTest的成员变量和构造方法
public class ServerTest {


    //集群服务类，这个类就是用来启动集群的，这个类的代码很快就会展示给大家
    private RaftGroupService raftGroupService;

    public ServerTest(final String dataPath, final String groupId, final PeerId serverId,
                      final NodeOptions nodeOptions) throws IOException {

        //根据用户配置的本地路径创建文件夹
        FileUtils.forceMkdir(new File(dataPath));
        //创建RPC服务器，这个服务器的具体实现先不必关心，后两章就会讲到
        final RpcServer rpcServer = RaftRpcServerFactory.createRaftRpcServer(serverId.getEndpoint());
        //设置日志存储文件的本地路径，File.separator就是路径分隔符
        nodeOptions.setLogUri(dataPath + File.separator + "log");
        //设置元数据存储文件的本地路径
        nodeOptions.setRaftMetaUri(dataPath + File.separator + "raft_meta");
        //创建集群服务对象并赋值
        this.raftGroupService = new RaftGroupService(groupId, serverId, nodeOptions, rpcServer);

    }


    public static void main(String[] args) throws InterruptedException, IOException {

        if (args.length != 4) {
            //如果用户没有传入正确的参数，或者传入的参数不足，就在控制台输出需要的参数形式
            System.out.println("Usage : java com.alipay.sofa.jraft.example.counter.CounterServer {dataPath} {groupId} {serverId} {initConf}");
            System.out.println("Example: java com.alipay.sofa.jraft.example.counter.CounterServer /tmp/server1 counter 127.0.0.1:8081 127.0.0.1:8081,127.0.0.1:8082,127.0.0.1:8083");
            //然后退出程序
            System.exit(1);
        }
        //解析命令行中传进来的字符串
        //解析出文件存放的本地路径
        final String dataPath = args[0];
```

```
        //要启动的集群服务的名字
        final String groupId = args[1];
        //当前节点的IP地址和端口号
        final String serverIdStr = args[2];
        //集群中所有节点的IP地址和端口号
        final String initConfStr = args[3];

        //创建的这个对象封装着当前节点需要的配置参数
        final NodeOptions nodeOptions = new NodeOptions();
        //设置超时选举时间，超过这个时间没有接收到领导者发送的信息，就会触发新一轮选举
        nodeOptions.setElectionTimeoutMs(1000);

        //创建的PeerId对象是用来封装代表当前服务器节点的信息的
        final PeerId serverId = new PeerId();

        //把当前节点的信息，比如IP地址，端口号等等都解析道PeerId对象中
        if (!serverId.parse(serverIdStr)) {
            throw new IllegalArgumentException("Fail to parse serverId:" + serverIdStr);
        }
        //创建集群配置类对象，这个对象中封装着整个集群中的节点信息
        final Configuration initConf = new Configuration();
        //把上面得到的字符串信息解析到该对象中，这时候配置类对象就拥有了集群中所有节点的信息
        if (!initConf.parse(initConfStr)) {
            throw new IllegalArgumentException("Fail to parse initConf:" + initConfStr);
        }
        //把集群初始配置信息设置到nodeOptions对象中
        nodeOptions.setInitialConf(initConf);
        //启动当前节点以及集群服务
        final ServerTest counterServer = new ServerTest(dataPath, groupId, serverId, nodeOptions);
        //通过集群服务类对象启动集群
        counterServer.raftGroupService.start();
    }
}
```

上面这个代码块，我相信大家都不陌生了，可能有的朋友已经去看了我提供的第一版本代码，发现我在第一版本代码中提供的测试类代码和这个代码有点小小的不一致。比如说，第一版本代码的测试类中，在 RaftGroupService 类的 start 方法中有这样一段代码。请看下面代码块。

RaftGroupService Java

```java
//框架的集群服务类，这个类中的start方法，就是启动集群的入口方法
public class RaftGroupService {

    //其他方法省略

     //启动当前节点以及节点所在集群的方法
    public synchronized Node start(final boolean startRpcServer) {
        //判断集群服务是否已经启动过了
        if (this.started) {
            return this.node;
        }
        //对节点ID和集群ID做非空检验
        if (this.serverId == null || this.serverId.getEndpoint() == null
                || this.serverId.getEndpoint().equals(new Endpoint(Utils.IP_ANY, 0))) {
            throw new IllegalArgumentException("Blank serverId:" + this.serverId);
        }
        if (StringUtils.isBlank(this.groupId)) {
            throw new IllegalArgumentException("Blank group id:" + this.groupId);
        }



        //把当前节点的端口号和IP地址信息交给节点管理器来管理
        NodeManager.getInstance().addAddress(this.serverId.getEndpoint());



        //创建当前服务器代表的节点，在这里节点被真正创建了！！！
        this.node = createAndInitRaftNode(this.groupId, this.serverId, this.nodeOptions);
        if (startRpcServer) {
            //启动RPC服务端
            this.rpcServer.init(null);
        } else {
            LOG.warn("RPC server is not started in RaftGroupService.");
        }
        this.started = true;
        LOG.info("Start the RaftGroupService successfully.");
        return this.no
    }


```

```
42      public Node createAndInitRaftNode(final String groupId, final PeerId s
    erverId, final NodeOptions opts) {
43          final Node ret = new NodeImpl(groupId, serverId);
44           //初始化节点,节点在这里初始化了！！！
45          if (!ret.init(opts)) {
46              throw new IllegalStateException("Fail to init node, please se
    e the logs to find the reason.");
47          }
48          return ret;
49      }
50  }
```

上面代码块的第 24 行，就是测试类中多出来的代码，也正是我要给大家补充的一点知识。NodeManager 这个类大家肯定还没见过，也没听我讲解过，这里我想简单解释一下。请大家想一想，我们构建的完整的集群，集群中会有多个节点，假如说集群中有 3 个节点，那么集群中每一个节点启动的时候，是不是都会执行上面那个 ServerTest 测试类中的代码？这个是肯定的吧？因为每个节点的启动流程都是相同的。每个服务器上部署了一个 raft 节点，执行程序的过程中，就会创建 Node 节点本身了。这时候也就意味着在服务器的这一个进程中产生了一个 raft 节点。但是上一章我跟大家简单解释了一下 MULTI-RAFT-GROUP 模式，大家应该也了解了，在构建多个集群的时候，很有可能一台服务器上部署了多个 Node 节点，并且这些节点都是在一个进程中创建的。这种情况下，一个进程中存在多个 raft 节点，肯定就要对这些节点做一些管理了。因为这些节点属于不同的集群，有各自的 groupId，而且每个节点的 NodeId 都是唯一的。所以，就引入了这个 NodeManager 类，从名字上就可以看出这个类的作用，就是节点管理器。这个类的具体实现也很简单，请看下面代码块。

NodeManager Java

```java
//节点管理器，这个管理器管理着当前服务器，或者说当前进程内所有的raft节点
public class NodeManager {

    //单例模式，把当前类的对象暴露出去
    private static final NodeManager INSTANCE = new NodeManager();

    //存储节点ID和节点的MaP
    private final ConcurrentMap<NodeId, Node> nodeMap = new ConcurrentHashMap<>();

    //存储集群ID和集群节点的Map
    private final ConcurrentMap<String, List<Node>> groupMap = new ConcurrentHashMap<>();

    //存放程序中所有节点的地址
    private final ConcurrentHashSet<Endpoint> addrSet = new ConcurrentHashSet<>();

    //把当前类对象暴露出去
    public static NodeManager getInstance() {
        return INSTANCE;
    }


    private NodeManager() {
    }

    //判断传进来的这个RPC地址是否已经存在了
    public boolean serverExists(final Endpoint addr) {
        if (addr.getIp().equals(Utils.IP_ANY)) {
            return this.addrSet.contains(new Endpoint(Utils.IP_ANY, addr.getPort()));
        }//如果集合中包含这个对象，就说明这个地址添加到集合中了
        return this.addrSet.contains(addr);
    }

    //从addrSet集合移除一个地址
    public boolean removeAddress(final Endpoint addr) {
        return this.addrSet.remove(addr);
    }

    //向addrSet集合中添加节点的地址
    public void addAddress(final Endpoint addr) {
        this.addrSet.add(addr);
    }
```

```
42
43
44      //添加节点的方法
45      public boolean add(final Node node) {
46          //判断当前节点的IP地址和端口号是否已经添加到集合中
47          //没有的话直接返回false
48          if (!serverExists(node.getNodeId().getPeerId().getEndpoint())) {
49              return false;
50          }
51          //得到当前节点的ID
52          final NodeId nodeId = node.getNodeId();
53          //将当前节点添加到nodeMap中，如果nodeId已经存在Map中了，该方法就会返回nodeId，否则返回null
54          if (this.nodeMap.putIfAbsent(nodeId, node) == null) {
55              //走到这里说明Map中还没有存放当前节点呢，获得当前节点所在的组ID
56              //我习惯把这个groupId当作集群ID，因为不管这个JVM进程中启动了多少节点
57              //这些raft节点又分为几个不同的组，组和组之间的数据并没有交集呀，一个组就相当于一个小集群
58              //所以我就在注释中也就写为集群ID了，大家知道就行
59              final String groupId = node.getGroupId();
60              //根据groupId获得存放该集群节点的List
61              List<Node> nodes = this.groupMap.get(groupId);
62              //如果list为null
63              if (nodes == null) {
64                  //就创建一个线程安全的ArrayList来存放集群中的节点
65                  nodes = Collections.synchronizedList(new ArrayList<>());
66                  //把集合添加到Map中
67                  List<Node> existsNode = this.groupMap.putIfAbsent(groupId, nodes);
68                  //如果返回null，说明groupMap中还没有对应的value，如果不反回null，说明已经有了value
69                  if (existsNode != null) {
70                      //这里直接获得Map中的value即可
71                      nodes = existsNode;
72                  }
73              }
74              //把当前节点添加到集合中
75              nodes.add(node);
76              return true;
77          }
78          //走到这里意味着nodeMap中已经存在对应的节点了，直接返回即可
79          return false;
80      }
81
82
83      //下面这些方法都很简单，就不再一一注释了
84      public boolean remove(final Node node) {
85          if (this.nodeMap.remove(node.getNodeId(), node)) {
```

```
                final List<Node> nodes = this.groupMap.get(node.getGroupId())
;
                if (nodes != null) {
                    return nodes.remove(node);
                }
            }
            return false;
        }


        public Node get(final String groupId, final PeerId peerId) {
            return this.nodeMap.get(new NodeId(groupId, peerId));
        }


        public List<Node> getNodesByGroupId(final String groupId) {
            return this.groupMap.get(groupId);
        }


        public List<Node> getAllNodes() {
            return this.groupMap.values().stream().flatMap(Collection::stream
).collect(Collectors.toList());
        }


    }
```

上面代码块中的注释非常详细，所以我就不再重复讲解其中的具体逻辑了。当然，在 RaftGroupService 类的 start 方法中，程序调用的只是 NodeManager 类的 addAddress 方法，只是把当前节点的 IP 地址和端口号信息存放到了 NodeManager 类的 addrSet 成员变量中，但是我可以提前告诉大家，在 NodeImpl 初始化的时候，会调用 NodeManager 节点管理器的 add 方法，把当前节点交给节点管理来保管，因为这个时候节点已经被创建成功了。总之，说来说去，很多操作都会在节点的初始化方法中进行，所以，接下来就让我们一起来分析分析 NodeImpl 的 init 方法究竟该怎么实现吧。

分析 NodeImpl 的初始化流程

还是老规矩，既然现在注意力又回到了 NodeImpl 类上，就让我们把上一章定义好的 NodeImpl 类搬运到这里，请大家简单回顾一下。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;


    //构造方法
    public NodeImpl() {
        this(null, null);
    }

    //构造方法
    public NodeImpl(final String groupId, final PeerId serverId) {
        //对集群Id判空
        if (groupId != null) {
            Utils.verifyGroupId(groupId);
        }//给集群id成员变量赋值
        this.groupId = groupId;
        //给当前节点的serverId赋值
        this.serverId = serverId != null ? serverId.copy() : null;
```

```
        //节点刚刚创建的时候，是未初始化状态
        this.state = State.STATE_UNINITIALIZED;
        //节点刚创建的时候，任期为0
        this.currTerm = 0;
        LOG.info("The number of active nodes increment to {}.", num);
    }


    //该方法暂时不做实现，这里也不用关心这个NodeOptions
    @Override
    public boolean init(final NodeOptions opts) {

    }

    //该方法暂时不做实现
    @Override
    public void shutdown() {

    }


}
```

可以看到，现在 Nodeimpl 类中已经定义了一些成员变量了，可能在之后的某个时刻，NodeImpl 类的某些成员变量会被其他类的对象使用，也许应该给某些成员变量定义 get 方法。当然，我在这里忽然提起这个也是为了稍微完善一下 Node 接口。比如说我们可以直接在 Node 接口中定义一些 get 方法。具体实现请看下面代码块。

Node Java

```
public interface Node extends Lifecycle<NodeOptions> {


    PeerId getLeaderId();


    NodeId getNodeId();

    String getGroupId();

}

```

当 NodeImpl 实现了上面的几个方法之后，其他对象就有机会在需要的时候得到 Node 节点的集群 Id 或者是 NodeId 了。当然，这个并不是什么重要的知识。我只是突然想起来了，所以对 Node 接口进行了一点重构。接下来，就让我们一起分析分析 init 方法该怎么实现吧。

一开始大家可能都没什么头绪，这很正常，在没有参照对象的情况下分析一个将要实现的框架的启动流程并不容易，这得要求你见多识广，毕竟这可不是什么微小的工作。不过，我们也不是一点头绪都没有，在上一章我已经为大家引入了节点的元数据信息这个概念，**也就是节点的任期以及在这个人任期给哪个节点投过票，这些信息都应该被持久化到本地，否则在节点宕机重启后可能会出现重复投票的情况。**这就告诉我们，节点在启动的过程中需要加载持久化到本地的元数据信息，用元数据文件中记录的信息给节点启动后的任期和投票纪录赋值，也就是下面代码块中的这两个成员变量。请看下面代码块。

NodeImpl Java

```
public class NodeImpl implements Node{

    //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;

    //其他内容省略
}
```

如果大家都理解了这个操作，那么接下来我想再多讲几句。**程序中有各种各样的功能要实现，实现起来需要定义各种各样的类，通常情况下都是很多类协同工作，才能实现一个完整的功能。按照不同的功能，我们也可以把程序中的所有类划分为不同的体系，通常情况下，我喜欢称一个体系为组件，或者模块。给大家讲解的时候，通常也都是说讲解某个组件的知识，这就意味着会引入很多类。**现在要实现加载元数据文件的功能了，这个功能当然也需要好几个类协同工作才能实现，所以，这时候我就可以告诉大家了，接下来，我就要为我们的程序引入第一个组件，**那就是节点元数据存储器，**也许有的朋友会有疑问，元数据是什么意思大家都清楚，但为什么叫存储器呢？原因很简单，**节点元数据信息肯定是会更新的吧？整个集群在工作期间可能会进行多次领导者选举，每一次选举集群节点的任期都会改变，为谁投票的记录也会改变，而每一次改变之后，都要及时把元数据信息持久化到本地。**所以，我把它称为元数据存储器，当然，这个存储器也负责在节点启动时将本地源数据信息加载到内存中，也就是说，这个节点元数据信息存储器中机肯定会定义 save 方法，也会定义 load 方法。讲到这里，这个元数据存储器的功能和作用，我就为大家大概讲解完毕了，毕竟这个组件的功能比较单一，实现起来非常简单，所以，这个并不是我要讲解的重点知识。

接下来，请大家再想一想，除了这个节点元数据存储器，我们的节点可能还需要什么组件呢？有的组件可能很容就被大家想到了，比如说日志组件，因为整个集群就是通过日志复制来保持数据统一，状态统一的，所以肯定需要有一个组件管理日志。有的组件大家目前可能还想不到，比如后面要引入的日志压缩组件，实际上就是快照组件。我现在跟大家讲解这个，并不是想在第五章就为大家拓展后面的知识，而是想为当前程序引入另一个类，**那就是 DefaultJRaftServiceFactory 这个类。**从名称上来说，这个类就是一个默认的 raft 服务工厂，刚才我已经给大家简单介绍了程序中需要实现的各种组件，其实也就说

明了，我们要实现的这个 raft 框架最终会包含一些组件，而这个 DefaultJRaftServiceFactory 类就是用来提供这些组件的。当然，现在我们还在实现第一版本代码，所以这个 DefaultJRaftServiceFactory 类的内容非常简单，只提供一个节点元数据存储器组件。具体实现请看下面代码块。

DefaultJRaftServiceFactory Java

```java
//默认的JRaft服务工厂类，这个类上有SPI注解，在程序启动的时候，显然是要被SPI机制加载到内存中的
@SPI
public class DefaultJRaftServiceFactory implements JRaftServiceFactory {

    public static DefaultJRaftServiceFactory newInstance() {
        return new DefaultJRaftServiceFactory();
    }

    //真正创建元数据存储器的方法
    //url是元数据文件的路径
    //raftOptions中封装着jRaft需要用到的一些配置参数，这些信息从RaftOptions对象创建完成的那一刻就初始化好的
    //因为这些信息都是RaftOptions对象的成员变量
    //这里的这个RaftOptions就不再展开讲解了，之后大家可以去看第一版本代码，封装的都是一些配置参数
    //方法也都是简单的get、set方法，等我在文章中为大家把第一版本代码实现了，大家就可以去代码中学习了
    @Override
    public RaftMetaStorage createRaftMetaStorage(final String uri, final RaftOptions raftOptions) {
        Requires.requireTrue(!StringUtils.isBlank(uri), "Blank raft meta storage uri.");
        //在这里创建元数据存储器
        return new LocalRaftMetaStorage(uri, raftOptions);
    }

}
```

如果大家理解了这个 DefaultJRaftServiceFactory 类，现在来看另外一个问题，那就是这个类的对象在什么时候创建呢？我相信这肯定是大家最关心的问题，因为在节点初始化的过程中肯定要用到这个类的对象，也就是要用到这个 raft 服务工厂，然后从这个工厂中获得元数据存储器。这个也很容易解释，还记得 NodeOptions 这个类的对象是什么时候创建的吧？**NodeOptions 类中其实就定义了 DefaultJRaftServiceFactory 这个成员变量，随着 NodeOptions 对象的创建，DefaultJRaftServiceFactory 也就被创建成功了。当 NodeImpl 初始化的时候，也就是执行 init 方法的时候，会根据方法参数 NodeOptions 中配置的信息进行初始化操作，这个时候，NodeOptions 就会把自己的成员变量 DefaultJRaftServiceFactory 交给 NodeImpl 使用。到这里大家应该也能意识到了，那就是在 NodeImpl 类中，也应该定义一个 DefaultJRaftServiceFactory 成员变量。当然，节点元数**

据存储器也应该定义在 NodeImpl 类中，不然 NodeImpl 怎么使用这个元数据存储器呢？所以，接下来我要对我们的程序进行一次大的重构。

首先我要为大家展示一下 LocalRaftMetaStorage 节点元数据存储器本身，为 raft 节点提供服务的默认工厂 DefaultJRaftServiceFactory 已经展示了，所以接下来直接展示重构之后的 NodeOptions 类，最后展示重构之后的 NodeImpl 类，在最后展示的 NodeImpl 类中，我会先实现 init 方法的部分逻辑。请看下面的代码块。

首先是 LocalRaftMetaStorage 类。

LocalRaftMetaStorage Java

```java
//元数据存储器，这个类实现的接口RaftMetaStorage我就不再展开了，接口很简单，看我提供的第一版本代码就行
public class LocalRaftMetaStorage implements RaftMetaStorage {

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

    //元数据文件的名称
    private static final String RAFT_META = "raft_meta";

    //判断元数据是否已经初始化了没有，所谓初始化，就是当前节点在启动的时候
    //是否已经存本地元数据文件中把之前的元数据加载到内存中了
    private boolean isInited;

    //元数据文件的路径，这个就是从main方法参数中传递过来的用户自己定义的
    //元数据文件夹的路径
    private final String path;

    //初始化好的任期，也就是当前节点之前的任期
    private long term;

    //最后一次给哪个节点投票了，PeerId封装的就是一个节点的信息，就可以代表raft集群中的一个节点
    //该成员变量初始化的时候为空，PeerId.emptyPeer()方法返回一个PeerId对象，但这个对象中的信息为空
    //所以我就直接称为空节点了
    private PeerId votedFor = PeerId.emptyPeer();

    //raft协议用到的配置参数对象，这个RaftOptions我就不再展开了
    //在上一个代码块已经跟大家解释过了，逻辑很简单，到时候直接去看我提供的代码就行
    private final RaftOptions raftOptions;

    //当前节点的实现类，当前节点的所有功能几乎全集中在这个nodeImpl类里了
    //这个NodeImpl类的内容非常多
    private NodeImpl node;

    //构造方法
    public LocalRaftMetaStorage(final String path, final RaftOptions raftOptions) {
        super();
        this.path = path;
        this.raftOptions = raftOptions;
    }


```

```
41
42      //当前组件初始化方法，这个方法会在NodeImpl类的init方法中被调用，相当于节点初始化
    的时候
43      //这个节点的元数据就已经从本地元数据文件中加载到程序中了，因为在下面这个方法中，会
    加载本地元数据文件
44      //到内存中
45      //这里的这个RaftMetaStorageOptions类我也不再展开讲解了，否则光是一些封装配置参
    数的类
46      //就讲起来没完了。在jraft中定义了很多组件，每个组件都有各自的XXXOptopns类，每个
    类的
47      //作用就是纯粹的封装一些配置参数，让组件初始化时使用的，方法也都是get、set方法
48      @Override
49      public boolean init(final RaftMetaStorageOptions opts) {
50          //如果元数据已经初始化了，就直接返回
51          if (this.isInited) {
52              LOG.warn("Raft meta storage is already inited.");
53              return true;
54          }
55          //从RaftMetaStorageOptions对象中获得当前正在初始化的NodeImpl节点
56          this.node = opts.getNode();
57          //这里就要从NodeImpl对象中获得nodeMetrics对象了，但是我在第一版本中
58          //把NodeImpl中很多方法都删掉了，所以下面这行代码就注释掉了，因为现在的NodeI
    mpl
59          //对象中没有getNodeMetrics方法
60          //this.nodeMetrics = this.node.getNodeMetrics();
61          try {
62              //根据用户配置的路径创建用来存储元数据的元文件夹，如果元文件已经存在了，
    就不再重复创建
63              FileUtils.forceMkdir(new File(this.path));
64          } catch (final IOException e) {
65              //创建过程中发生异常就打印日志
66              LOG.error("Fail to mkdir {}", this.path, e);
67              return false;
68          }
69          //在这里执行加载元数据文件到内存的操作
70          if (load()) {
71              this.isInited = true;
72              return true;
73          } else {
74              //加载失败则直接返回false
75              return false;
76          }
77      }
78
79
80
81      //根据元数据文件的本地路径和元数据文件名称创建一个用来加载元数据文件的
    //ProtoBufFile对象
82      private ProtoBufFile newPbFile() {
```

```
        //注意，这里才是真正对文件本身进行操作，并不是对文件夹进行操作，一定要区分清楚
        return new ProtoBufFile(this.path + File.separator + RAFT_META);
    }



    //加载本地的元数据文件到内存的方法
    private boolean load() {
        //创建一个ProtoBufFile对象，该对象用来加载本地的元数据文件
        final ProtoBufFile pbFile = newPbFile();
        try {
            //加载元数据文件，这里加载完毕之后，会根据Protobuf协议把元数据文件中的数据解析成一个StablePBMeta
            //元数据文件中的数据都会封装带这个StablePBMeta中
            final LocalStorageOutter.StablePBMeta meta = pbFile.load();
            //假如当前节点在集群中是第一次启动，这时候肯定还没有什么元数据文件，所以会返回null
            //判空，然后根据meta中的数据对成员变量term和votedFor赋值
            if (meta != null) {
                //将当前节点之前的任期赋值
                this.term = meta.getTerm();
                //获得当前节点最后一次给哪个节点投过票
                return this.votedFor.parse(meta.getVotedfor());
            }
            //为null直接返回true，由此可见，在这个jraft框架中，程序加载本地元信息文件失败了
            //或者集群中根本就没有元数据文件呢，也不会报错，而是直接返回true
            //大不了节点初始化任期就是0，反正可以和其他节点通信，修改自己的任期值
            return true;
        } catch (final FileNotFoundException e) {
            //这里跑出的是找不到文件的异常，就算抛这个异常，也返回true
            return true;
        } catch (final IOException e) {
            //报出其他异常才返回false
            LOG.error("Fail to load raft meta storage", e);
            return false;
        }
    }



    //这个方法的作用是持久化元数据，也就是元数据落盘
    //当程序在执行的时候，当前节点的任期可能会发生变化，也可能会给其他节点投票，只要出现了这些动作，那么这些数据就要持久化到硬盘中
    //在NodeImpl这个类中，就会看到下面这些方法被调用了
    private boolean save() {
        //创建序列化数据，还是通过Protobuf协议进行落盘，所以要创建一个StablePBMeta对象
```

```
126             final LocalStorageOutter.StablePBMeta meta = LocalStorageOutter.S
    tablePBMeta.newBuilder()
127                     //在这里把当前节点任期穿进去
128                     .setTerm(this.term)
129                     //为哪个节点投票也传进去
130                     .setVotedfor(this.votedFor.toString())
131                     .build();
132             //根据数据要存放的本地文件的路径和文件名获得ProtoBufFile对象
133             final ProtoBufFile pbFile = newPbFile();
134             try {//开始持久化操作
135                 if (!pbFile.save(meta, this.raftOptions.isSyncMeta())) {
136                     reportIOError();
137                     return false;
138                 }
139                 return true;
140             } catch (final Exception e) {
141                 LOG.error("Fail to save raft meta", e);
142                 reportIOError();
143                 return false;
144             } finally {//记录日志
145                 LOG.info("Save raft meta, path={}, term={}, votedFor={}, cos
    t time={} ms", this.path, this.term,
146                         this.votedFor, cost);
147             }
148         }
149 
150 
151         //持久化当前节点任期和投票记录的方法
152         @Override
153         public boolean setTermAndVotedFor(final long term, final PeerId peerI
    d) {
154             checkState();
155             this.votedFor = peerId;
156             this.term = term;
157             return save();
158         }
159 
160         //剩下的get、set方法就全省略了
161 
162 
163 
164 }
```

可以看到，在 LocalRaftMetaStorage 初始化的过程中，也就是 init 方法中，会执行 load 方法，把本地的元数据文件加载到内存中。当然，假如当前节点在集群中是第一次启动，这时候肯定还没有什么元数据文件，所以 load 方法 也不会加载到什么文件。

接下来就是 NodeOptions 类。

NodeOptions Java

```java
//这个类封装的是Node节点初始化时需要用到的配置参数
public class NodeOptions{

    //该类加载的时候就会执行这行代码，然后就会通过SPI机制加载DefaultJRaftServiceFactory类到内存中，DefaultJRaftServiceFactory类是JRaftServiceFactory接口的实现类
    //DefaultJRaftServiceFactory的作用我写在该类的注释中了，去该类中看一看
    //JRaftServiceFactory这个接口我也不再展开讲解了，定义的方法很简单
    public static final JRaftServiceFactory defaultServiceFactory = JRaftServiceLoader.load(JRaftServiceFactory.class).first();

    //超时选举时间，默认1000ms
    private int electionTimeoutMs = 1000;

    //集群配置信息对象在这里初始化，刚初始化时配置信息是空的
    private Configuration initialConf = new Configuration();

    //用户配置的存储日志文件的路径
    private String logUri;

    //元数据文件存储路径
    private String raftMetaUri;

    //get、set方法就省略了

}


```

最后展示一下重构之后的 NodeImpl 类。

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;

    //元数据存储器
    private RaftMetaStorage metaStorage;


    //构造方法
    public NodeImpl() {
        this(null, null);
    }
```

```
44
45
46      //构造方法
47      public NodeImpl(final String groupId, final PeerId serverId) {
48          //对集群Id判空
49          if (groupId != null) {
50              Utils.verifyGroupId(groupId);
51          }//给集群id成员变量赋值
52          this.groupId = groupId;
53          //给当前节点的serverId赋值
54          this.serverId = serverId != null ? serverId.copy() : null;
55          //节点刚刚创建的时候，是未初始化状态
56          this.state = State.STATE_UNINITIALIZED;
57          //节点刚创建的时候，任期为0
58          this.currTerm = 0;
59          LOG.info("The number of active nodes increment to {}.", num);
60      }
61
62
63      //当Nodeimpl调用它的init方法时，这时候方法参数NodeOptions
64      //已经创建了DefaultJRaftServiceFactory对象了
65      @Override
66      public boolean init(final NodeOptions opts) {
67          //对Node节点需要的数据进行非空校验
68          Requires.requireNonNull(opts, "Null node options");
            Requires.requireNonNull(opts.getRaftOptions(), "Null raft option
    s");
69          Requires.requireNonNull(opts.getServiceFactory(), "Null jraft ser
    vice factory");
70
71          //得到了DefaultJRaftServiceFactory对象，这个就是提供组件服务的工厂，在第
    一版本中只提供了元数据存储器组件服务
72          //日志和快照服务都没有提供
73          this.serviceFactory = opts.getServiceFactory();
74          //给options赋值
75          this.options = opts;
76
77          //校验一下IP地址不为空，注意，这个serverId在构造方法中就被赋值了
78          //也就是说，在RaftGroupService类中创建NodeImpl对象的时候，就已经给serve
    rId赋值了
79          //IP_ANY其实就是0.0.0.0
80          if (this.serverId.getIp().equals(Utils.IP_ANY)) {
81              LOG.error("Node can't started from IP_ANY.");
82              return false;
83          }
84
85          //在这里校验当前节点的IP地址和端口号是否已经添加到节点管理器中了
86          //如果没有添加进去则记录错误日志，这里大家可以去看一下测试类，当测试类启动时
87
```

```
88          //节点的网络地址已经在RaftGroupService对象的start方法中被添加到节点管理器中了
89          //所以程序执行到这里时，肯定会返回true，也就不会进入下面的分支了
90          if (!NodeManager.getInstance().serverExists(this.serverId.getEndpoint())) {
91              LOG.error("No RPC server attached to, did you forget to call addService?");
92              return false;
93          }
94
95
96          //接下来就是初始化节点元数据存储器组件
97          //初始化元数据存储器组件，在初始化的过程中，会把本地元数据文件加载到内存中
98          if (!initMetaStorage()) {
99              //初始化失败则记录错误信息
100             LOG.error("Node {} initMetaStorage failed.", getNodeId());
101             return false;
102         }
103
104         //接下来还有一个操作，那就是把集群刚启动时的初始配置信息存放到conf成员变量中
105         //创建一个配置对象，该对象会封装一个集群的配置信息
106         this.conf = new ConfigurationEntry();
107
108         //这里其实还有这样一行代码，因为在sofajraft框架中，配置信息也是日志，也是需要传递给
109         //跟随者的，加入集群中配置发生变更时，最终会在领导者产生一条配置变更日志，会被封装在
110         //ConfigurationEntry对象中，然后赋值给跟随者，只要是日志就会有日志索引和任期，而LogId
111         //就是封装日志索引和任期的对象，但现在我们还没有引入日志组件，所以就先把这条代码注释了
112         //this.conf.setId(new LogId());
113         //把配置信息设置到conf成员变量中
114         this.conf.setConf(this.options.getInitialConf());
115         if (!this.conf.isEmpty()) {
116             Requires.requireTrue(this.conf.isValid(), "Invalid conf: %s", this.conf);
117         } else {
118             LOG.info("Init node {} with empty conf.", this.serverId);
119         }
120
121
122
123         //把当前节点添加到节点管理器中
124         if (!NodeManager.getInstance().add(this)) {
125             LOG.error("NodeManager add {} failed.", getNodeId());
126             return false;
127         }
```

```
    }
    //初始化元数据存储器的方法，这个方法很重要
    private boolean initMetaStorage() {
        //通过服务工厂创建元数据存储器，这里把用户配置的元数据文件的本地路径传递给元数据存储器了
        //this.options.getRaftMetaUri()这个就是用户自己定义的存放元数据文件的文件夹路径
        this.metaStorage = this.serviceFactory.createRaftMetaStorage(this.options.getRaftMetaUri(), this.raftOptions);
        //创建RaftMetaStorageOptions对象，这个对象是专门给元数据存储器使用的，从名字上就可以看出来
        //封装了元数据存储器需要的配置参数
        RaftMetaStorageOptions opts = new RaftMetaStorageOptions();
        //把当前节点设置到opts中
        opts.setNode(this);
        //在这里真正初始化了元数据存储器，初始化的过程中，会把元数据本地文件中的任期和为谁投票这些数据加载到内存中
        //就赋值给元数据存储器对象中的两个对应成员变量，初始化失败则记录日志
        if (!this.metaStorage.init(opts)) {
            LOG.error("Node {} init meta storage failed, uri={}.", this.serverId, this.options.getRaftMetaUri());
            return false;
        }
        //给当前节点的任期赋值
        this.currTerm = this.metaStorage.getTerm();
        //得到当前节点的投票记录，这里的copy在功能上就相当于一个深拷贝方法
        this.votedId = this.metaStorage.getVotedFor().copy();
        return true;
    }


    //该方法暂时不做实现
    @Override
    public void shutdown() {

    }


}
```

好了，现在我可以为大家初步总结一下目前的 NodeImpl 在调用 init 方法的过程中，会执行什么操作。因为这个流程比较简短，也很清晰，所以就先不画流程图了。**在 NodeImpl 执行初始化方法的时候，首**

先会对一些重要数据进行判空操作，然后给自己的 serviceFactory 和 options 成员变量赋值，接着就是初始化节点元数据存储器组件，在初始化的过程中，元数据存储器会加载本地的元数据文件，加载完毕后会用元数据文件中的数据给当前节点的 currTerm 和 votedId 赋值，然后将集群配置文件信息封装到 conf 成员变量中，最后把节点添加到节点管理器中。这就是 NodeImpl 的 init 方法现阶段的流程。

引入超时选举定时器

既然我们刚才实现的 init 只拥有现阶段的流程逻辑，这就意味着它肯定还会继续迭代和重构。这是肯定的，因为现在我就意识到目前的程序中有个地方可以进一步改造了，那就是应该让节点触发超时选举。这肯容易理解吧，我的节点都已经启动了，如果集群中有 3 个节点，3 个节点部署在 3 台不同的服务器上，都启动了程序，NodeImpl 也都初始化完毕了，然后呢？就没动作了？那我启动这 3 个节点干什么呀，对吧？既然我构建了一个 raft 集群，肯定要选出一个领导者，然后处理客户端写操作，生产日志，再把日志赋值给跟随者。但我们的程序目前只能做到一个启动，启动之后就尴尬地停在那里了，根本不会选举出领导者，没有领导者，集群就无法工作，所以接下来，我就就要为我的程序实现领导者选举功能。

那请大家想一下，领导者选举的流程是什么？这个我们在前几章已经分析过了：**就是当前节点在规定的时间内没有接收到来自领导者的心跳消息，就可以进入超时选举阶段了，当然，为了避免选举风暴，应该给每个节点的超时选举时间增加随机性。所以，按照这个逻辑，其实还应该在 NodeImpl 这个类中定义一个成员变量，这个成员变量的作用就是记录上一次接收到领导者心跳消息的时间，然后用现在时间和记录的上一次接收到领导消息的时间记录做对比，如果超过超时选举时间了，就意味着当前节点可以进行超时选举了。比如说，就把这个成员变量定义成 lastLeaderTimestamp，意思就是最后一次收到领导者信息的时间戳。**

开始深入分析之后，我现在又意识到了一个问题，或者说是一个细节，要和大家探讨一下，当节点启动的时候，这个 lastLeaderTimestamp 肯定要被赋值，就用当前时间来赋值吧。赋值完毕之后，因为是集群刚刚启动，所以还没有领导者，每个节点都不会收到来自领导者的信息，过一段时间之后，触发了超时选举，肯定有某个节点会率先进入领导者选举阶段，如果选举成功了，它就当选为领导者了。这时候集群中有 1 个领导者，还有两个跟随者，领导者会持续向跟随者发送心跳消息。我的问题是，这两个跟随者怎么判断自己有没有在规定时间内接收到了领导者的心跳消息呢？我的思路很明确，**因为领导者给跟随者发送心跳消息是一个周期性的操作，跟随者每接收到一次心跳消息，都会更新自己的 lastLeaderTimestamp 成员变量，所以，跟随者自然也需要定义一个具有周期性功能的组件，来周期性地判断当前时间和 lastLeaderTimestamp 的差值是不是大于超时选举时间了，如果超过了，那么跟随者就要进入领导者选举阶段。**经过这一番分析，最终我决定给 NodeImpl 引入一个超时选举定时器，其

实就是一个具有定时任务功能的组件，我决定使用时间轮来实现。并且把这个超时选举定时器交给 NodeImpl 来使用，其实就是定义为 NodeImpl 的成员变量，名字我都想好了，就叫做 electionTimer。

很好，现在我们已经引入了 lastLeaderTimestamp 这个成员变量，还有 electionTimer 超时选举定时器，接下来就让我来为大家一一展示它们。首先是 lastLeaderTimestamp 成员变量，这个成员变量没什么可多说的，就是一个时间戳而已，在 NodeImpl 被创建的时候就应该被赋值了，也就是在构造方法中被赋值了，同时我也为它定义了两个方法，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;


    //构造方法
    public NodeImpl() {
        this(null, null);
    }

```

```java
    //构造方法
    public NodeImpl(final String groupId, final PeerId serverId) {
        //对集群Id判空
        if (groupId != null) {
            Utils.verifyGroupId(groupId);
        }//给集群id成员变量赋值
        this.groupId = groupId;
        //给当前节点的serverId赋值
        this.serverId = serverId != null ? serverId.copy() : null;
        //节点刚刚创建的时候，是未初始化状态
        this.state = State.STATE_UNINITIALIZED;
        //节点刚创建的时候，任期为0
        this.currTerm = 0;
        //初始化lastLeaderTimestamp的值
        //Utils.monotonicMs()得到的就是当前时间
        updateLastLeaderTimestamp(Utils.monotonicMs());
        LOG.info("The number of active nodes increment to {}.", num);
    }




    //当接收到领导者发送过来的信息时，就调用这个方法，更新当前节点最后一次接收到领导者信息的时间
    private void updateLastLeaderTimestamp(final long lastLeaderTimestamp) {
        this.lastLeaderTimestamp = lastLeaderTimestamp;
    }



    //判断是否触发选举超时了，如果触发了就可以让选举定时器进行工作了，在选举定时器中就可以
    //预投票，然后正式投票选举出新的领导者了，该方法返回false，就意味着当前领导者无效了
    //也就可以让跟随者进行领导者选举了
    private boolean isCurrentLeaderValid() {
        //下面这行代码的意思就是用当前时间减去当前节点最后一次收到领导者信息的时间
        //如果这个时间差超过了用户设置的超时选举的时间，那就可以在超时选举定时器中进行投票选举领导者的活动了
        return Utils.monotonicMs() - this.lastLeaderTimestamp < this.options.getElectionTimeoutMs();
    }


    //其他方法省略

```

```
88    }
```

然后是 electionTimer 超时选举定时器。这个定时器的代码我也已经实现了，是一个新的类型，我定义为了 RepeatedTimer。我们可以先看一下该类的具体实现，然后再展示一下又一次重构后的 NodeImpl 类。在阅读下面代码块的时候，大家可以思考一下，节点的超时选举时间的随机性是怎么保障的。在测试类中，我们为每个节点定义的超时时间为 1000 毫秒，看看 RepeatedTimer 是怎么实现这个随机性的。

RepeatedTimer Java

```java
//定时任务管理器，学习这个类的逻辑可以直接从这个类的start方法作为入口
//这里我还要再扩展一下，这个RepeatedTimer类并不是只用来创建超时选举定时器的
//它可以用来创建很多定时器，比如后面要讲解的日志压缩定时器，也就是快照定时器，选举超时定时器等等
//都是这个类的对象
public abstract class RepeatedTimer {


    public static final Logger LOG = LoggerFactory.getLogger(RepeatedTimer.class);

    //timer是HashedWheelTimer
    private final Timer timer;

    //实例是HashedWheelTimeout
    private Timeout timeout;

    //超时选举时间
    private volatile int timeoutMs;
    //定时器名称
    private final String name;

    //下面是当前定时器的几种状态
    //定时器是否停止工作
    private boolean stopped;
    //是否正在运行
    private volatile boolean running;
    //是否被销毁
    private volatile boolean destroyed;
    //是否正在调用超时选举方法
    private volatile boolean invoking;
    //为了防止出现并发问题，还定义了一个锁
    private final Lock lock = new ReentrantLock();


    //该构造方法会在NodeImpl类的init方法中被调用
    public RepeatedTimer(final String name, final int timeoutMs, final Timer timer) {
        this.name = name;
        this.timeoutMs = timeoutMs;
        this.stopped = true;
        this.timer = Requires.requireNonNull(timer, "timer");
    }


```

```
    //该方法在NodeImpl类的init中实现了，就是调用handleElectionTimeout方法，开始进行预投票活动
    protected abstract void onTrigger();


    //该方法也在NodeImpl类的init中实现了，返回一个超时选举时间，这个时间是随机的
    protected int adjustTimeout(final int timeoutMs) {
        return timeoutMs;
    }


    //启动定时器的方法
    public void start() {
        this.lock.lock();
        try {
            if (this.destroyed) {
                return;
            }
            if (!this.stopped) {
                return;
            }
            this.stopped = false;
            if (this.running) {
                return;
            }
            this.running = true;
            //开始调度定时器管理的定时任务
            schedule();
        } finally {
            this.lock.unlock();
        }
    }


    //重新启动该定时器
    public void restart() {
        this.lock.lock();
        try {
            if (this.destroyed) {
                return;
            }
            this.stopped = false;
            this.running = true;
            schedule();
        } finally {
            this.lock.unlock();
        }
    }
```

```
90
91
92
        //启动该类对象代表的定时器之后，就会在下面这个方法中创建一个TimerTask任务，并且把这个任务提交到
93
        //时间轮中，设置好多少时间之后执行该定时任务
94      private void schedule() {
95          if (this.timeout != null) {
96              this.timeout.cancel();
97          }//在这里创建了一个时间轮定时任务
98          final TimerTask timerTask = timeout -> {
99              try {//任务逻辑就是执行当前类的run方法
100                 RepeatedTimer.this.run();
101             } catch (final Throwable t) {
102                 LOG.error("Run timer task failed, taskName={}.", RepeatedTimer.this.name, t);
103             }
104         };//在这里提交到时间轮中，adjustTimeout(this.timeoutMs)得到的就是随即选举超时时间
105         //时间轮就会在这个时间之后执行定时任务
106         this.timeout = this.timer.newTimeout(timerTask, adjustTimeout(this.timeoutMs), TimeUnit.MILLISECONDS);
107     }
108
109
110     //该类对象的核心方法，时间轮线程中执行的实际上就是该类对象中的run方法
111     public void run() {
112         this.invoking = true;
113         try {//调用下面这个方法后，会在该方法内部进一步调用handleElectionTimeout方法方法进行预投票活动
114             onTrigger();
115         } catch (final Throwable t) {
116             LOG.error("Run timer failed.", t);
117         }
118         boolean invokeDestroyed = false;
119         this.lock.lock();
120         try {
121             this.invoking = false;
122             //判断该定时器有没有终止
123             if (this.stopped) {
124                 this.running = false;
125                 invokeDestroyed = this.destroyed;
126             } else {//走到这里说明定时器还在正常活动
127                 this.timeout = null;
128                 //再次调用schedule方法，其实就是向时间轮提交了下一个选举超时任务
129                 //到时间执行即可
130                 schedule();
131             }
132         } finally {
```

```
                this.lock.unlock();
            }
            if (invokeDestroyed) {
                onDestroy();
            }
        }
    }
```

不好意思，刚才给大家开了个玩笑，其实随机性根本没体现在上面的代码块，所以大家根本找不到结果。RepeatedTimer 是个抽象类，它的实现类在 NodeImpl 节点中，所以随机性方法也在 NodeImpl 中，所以，请大家先看看重构后的 NodeImpl 类，然后我再来为大家梳理这个超时选举定时器的执行流程吧。

RepeatedTimer 类中有两个非常重要的方法没有被实现，那就是 onTrigger 和 adjustTimeout 方法，其中 onTrigger 方法就是用来具体处理超时选举的逻辑的，也就是执行领导者选举的操作，而 adjustTimeout 就是给当前定时器返回一个随机超时选举时间，当过了这个时间之后，定时器开始执行超时选举方法，也就是 onTrigger 方法。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;


    //当Nodeimpl调用它的init方法时，这时候方法参数NodeOptions
    //已经创建了DefaultJRaftServiceFactory对象了
    @Override
    public boolean init(final NodeOptions opts) {
```

```
        //省略部分内容

        if (!NodeManager.getInstance().serverExists(this.serverId.getEndpoint())) {
            LOG.error("No RPC server attached to, did you forget to call addService?");
            return false;
        }



        //下面我删减了很多代码，大家简单看看就行，在源码中快照服务，状态机，日志组件都会初始化
        //但在第一版本中我全删除了
        //定义超时选举定时器名称
        String name = "JRaft-ElectionTimer-" + suffix;
        //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
        //createTimer方法就是创建了一个时间轮对象，时间轮的刻度是1秒
        //new HashedWheelTimer(new NamedThreadFactory(name, true), 1, TimeUnit.MILLISECONDS, 2048);
        //这里就不具体展开了，实际上在第一版本代码中这里并不是这样写的
        //大家看源码的时候就会发现差别了，这里简化了一些，只是为了讲解方便，当然，在第一版本代码中逻辑也很简单
        this.electionTimer = new RepeatedTimer(name,this.options.getElectionTimeoutMs(),createTimer(name)) {
            //这个方法实现的是RepeatedTimer类中的方法
            @Override
            protected void onTrigger() {
                //这是超时选举定时器中最核心的方法，就是在该方法中，开始执行超时选举任务了
                handleElectionTimeout();
            }

            @Override
            protected int adjustTimeout(final int timeoutMs) {
                //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
                //否则多个节点同时成为候选者，很可能选举失败
                return randomTimeout(timeoutMs);
            }
        };



        //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;
        }
    }


    //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() {
        //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;
        }

        //其他逻辑暂时不做实现
    }


    //返回一个随机超时时间的方法，超时选举定时器会调用该方法。这里大家应该能想到，在raft协议中，为了避免多个节点同时成为候选者
    //所以会让超时选举时间都错开一些，也就是每个节点的超时选举时间都不相同，就是通过下面的方法实现的
    private int randomTimeout(final int timeoutMs) {
        return ThreadLocalRandom.current().nextInt(timeoutMs, timeoutMs + this.raftOptions.getMaxElectionDelayMs());
    }


    //判断是否触发选举超时了，如果触发了就可以让选举定时器进行工作了，在选举定时器中就可以
    //预投票，然后正式投票选举出新的领导者了，该方法返回false，就意味着当前领导者无效了
    //也就可以让跟随者进行领导者选举了
    private boolean isCurrentLeaderValid() {
        //下面这行代码的意思就是用当前时间减去当前节点最后一次收到领导者信息的时间
        //如果这个时间差超过了用户设置的超时选举的时间，那就可以在超时选举定时器中进行投票选举领导者的活动了
        return Utils.monotonicMs() - this.lastLeaderTimestamp < this.options.getElectionTimeoutMs();
    }


```

```
121         //其他方法省略
122
123
124     }
```

到此为止，我就把超时选举的基本流程为大家展示完毕了，当然，具体的细节还没有铺展开。但我们确实可以先停下脚步，仔细梳理一下超时选举的流程，先把这一块的逻辑完全掌握了，再去深入研究下一个问题。

首先，超时任务定时器是在 NodeImpl 对象的 init 方法中创建的，也就是说，在节点初始化的时候，节点的超时选举定时器就被创建了。这个定时器内部，也就是 RepeatedTimer 对象内部，持有者一个时间轮工具，用这个时间轮可以按照用户设定的时间执行定时任务。到这里，其实流程和之前的也差不多，无非就是在 init 方法中多创建了一个 RepeatedTimer 对象，如果这个 RepeatedTimer 对象不启动，那么当前节点仍然会停在这里，不会继续运行了。这样一来，集群中仍然选不出一个领导者。所以，现在要做的就是启动这个 RepeatedTimer 超时选举定时器。当它被启动的时候，程序就会来来到 RepeatedTimer 对象的 strart 方法中，在方法内部调用 schedule 方法。就是下面代码块展示的这样。

Java

```java
//启动定时器的方法
public void start() {
    this.lock.lock();
    try {
        if (this.destroyed) {
            return;
        }
        if (!this.stopped) {
            return;
        }
        this.stopped = false;
        if (this.running) {
            return;
        }
        this.running = true;
        //开始调度定时器管理的定时任务
        schedule();
    } finally {
        this.lock.unlock();
    }
}
```

而在 schedule 方法中，会创建一个定时任务，任务逻辑就是执行 RepeatedTimer 对象中的 run 方法，然后通过 adjustTimeout(this.timeoutMs) 获得随机超时选举时间，也就是定时任务的触发时间，接着把定时任务提交给时间轮。

```java
//启动该类对象代表的定时器之后，就会在下面这个方法中创建一个TimerTask任务，并且把这个任务提交到
//时间轮中，设置好多少时间之后执行该定时任务
private void schedule() {
    if (this.timeout != null) {
        this.timeout.cancel();
    }//在这里创建了一个时间轮定时任务
    final TimerTask timerTask = timeout -> {
        try {//任务逻辑就是执行当前类的run方法
            RepeatedTimer.this.run();
        } catch (final Throwable t) {
            LOG.error("Run timer task failed, taskName={}.", RepeatedTimer.this.name, t);
        }
    };//在这里提交到时间轮中，adjustTimeout(this.timeoutMs)得到的就是随即选举超时时间
    //时间轮就会在这个时间之后执行定时任务
    this.timeout = this.timer.newTimeout(timerTask, adjustTimeout(this.timeoutMs), TimeUnit.MILLISECONDS);
}
```

接下来，当时间轮执行了该定时任务，程序就会来到 RepeatedTimer 类的 run 方法中，在 run 方法中，会执行超时选举的方法，也就是 onTrigger 方法，当该方法执行完毕后，会再次调用 schedule 方法，也就意味着会再次向时间轮中提交下一个超时选举定时任务。之后，程序就这样开始循环提交超时选举定时任务了。

```java
//该类对象的核心方法，时间轮线程中执行的实际上就是该类对象中的run方法
public void run() {
    this.invoking = true;
    try {//调用下面这个方法后，会在该方法内部进一步调用handleElectionTimeout方法方法进行预投票活动
        onTrigger();
    } catch (final Throwable t) {
        LOG.error("Run timer failed.", t);
    }
    boolean invokeDestroyed = false;
    this.lock.lock();
    try {
        this.invoking = false;
        //判断该定时器有没有终止
        if (this.stopped) {
            this.running = false;
            invokeDestroyed = this.destroyed;
        } else {//走到这里说明定时器还在正常活动
            this.timeout = null;
            //再次调用schedule方法，其实就是向时间轮提交了下一个选举超时任务
            //到时间执行即可
            schedule();
        }
    } finally {
        this.lock.unlock();
    }
    if (invokeDestroyed) {
        onDestroy();
    }
}
```

以上，就是超时选举定时器的大概工作流程，大家可以先仔细品味品味这个过程，如果理解了，那么接下来，就让我们一起看看真正的超时选举操作该怎么实现。是的，接下来，我就要为大家具体实现一下 handleElectionTimeout 方法了。这个方法就是触发超时选举时，真正要调用的方法。当然，这一章显然是没时间对这个方法详细展开了，所以就在下一章再讲解吧。

好了各位，我们下一章见！