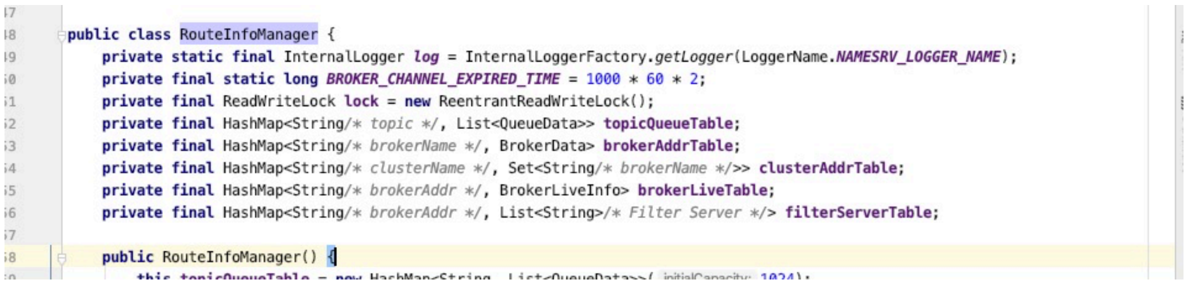
读书笔记 2 nameServer 消息队列协调

NameServer 的作用：

整个消息队列的状态服务器，各个组件通过 NameServer 来了解全局信息，各个角色的机器定时上报自己状态，超时认为不可用，其他组件会将此机器移出。

NameServer 可以多个， 互相独立，其他角色同时多个上报从而达到热备份。

集群状态的存储结构
org.apache.rocketmq.namesrv.routeinfo.RouteInfoManager

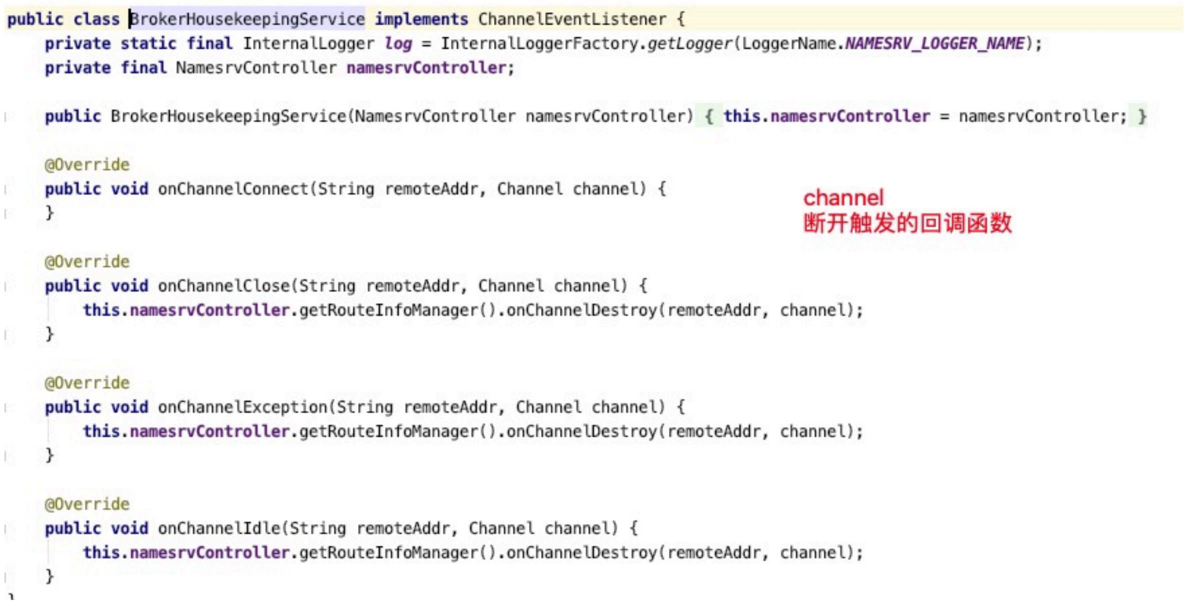

```java
public class RouteInfoManager {
    private static final InternalLogger log = InternalLoggerFactory.getLogger(LoggerName.NAMESRV_LOGGER_NAME);
    private final static long BROKER_CHANNEL_EXPIRED_TIME = 1000 * 60 * 2;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final HashMap<String/* topic */, List<QueueData>> topicQueueTable;
    private final HashMap<String/* brokerName */, BrokerData> brokerAddrTable;
    private final HashMap<String/* clusterName */, Set<String/* brokerName */>> clusterAddrTable;
    private final HashMap<String/* brokerAddr */, BrokerLiveInfo> brokerLiveTable;
    private final HashMap<String/* brokerAddr */, List<String>/* Filter Server */> filterServerTable;

    public RouteInfoManager() {
```

nameServer 的作用就是维护这 5 个变量中存储的信息

状态维护逻辑：

NameServer 的主要逻辑再 DefaultRequestProcessor 中

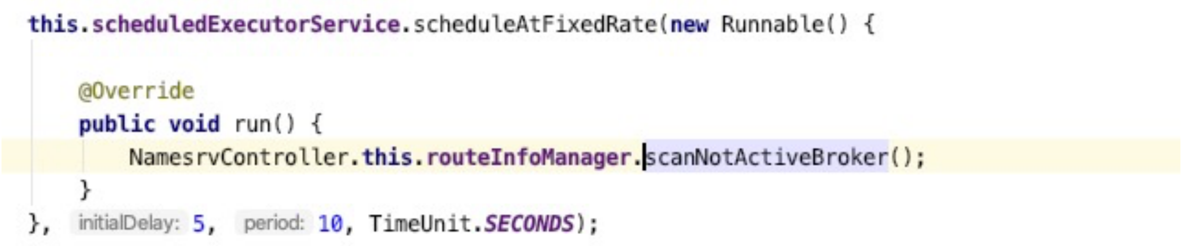
连接断开也会触发更新，如下：


```java
public class BrokerHousekeepingService implements ChannelEventListener {
    private static final InternalLogger log = InternalLoggerFactory.getLogger(LoggerName.NAMESRV_LOGGER_NAME);
    private final NamesrvController namesrvController;

    public BrokerHousekeepingService(NamesrvController namesrvController) { this.namesrvController = namesrvController; }

    @Override
    public void onChannelConnect(String remoteAddr, Channel channel) {
    }

    @Override
    public void onChannelClose(String remoteAddr, Channel channel) {
        this.namesrvController.getRouteInfoManager().onChannelDestroy(remoteAddr, channel);
    }

    @Override
    public void onChannelException(String remoteAddr, Channel channel) {
        this.namesrvController.getRouteInfoManager().onChannelDestroy(remoteAddr, channel);
    }

    @Override
    public void onChannelIdle(String remoteAddr, Channel channel) {
        this.namesrvController.getRouteInfoManager().onChannelDestroy(remoteAddr, channel);
    }
}
```

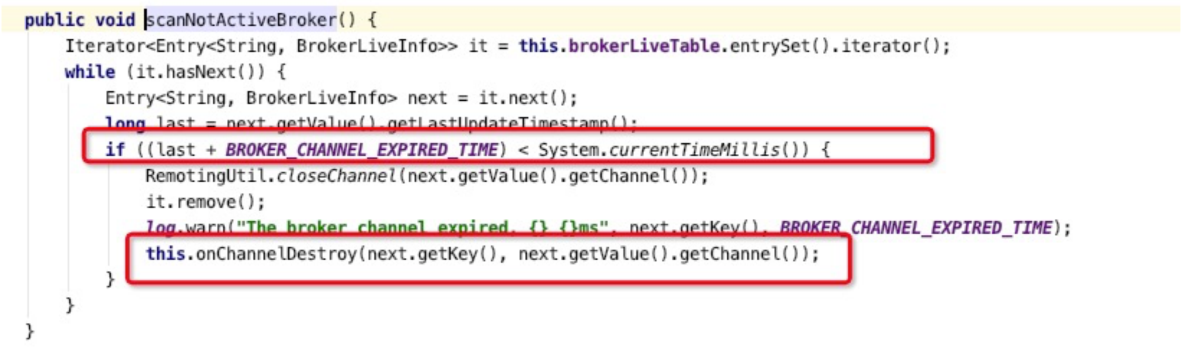
org.apache.rocketmq.namesrv.routeinfo.RouteInfoManager#onChannelDestroy

NameServer还有定时清除，长时间时间戳不更新进行清除

org.apache.rocketmq.namesrv.NamesrvController#initialize



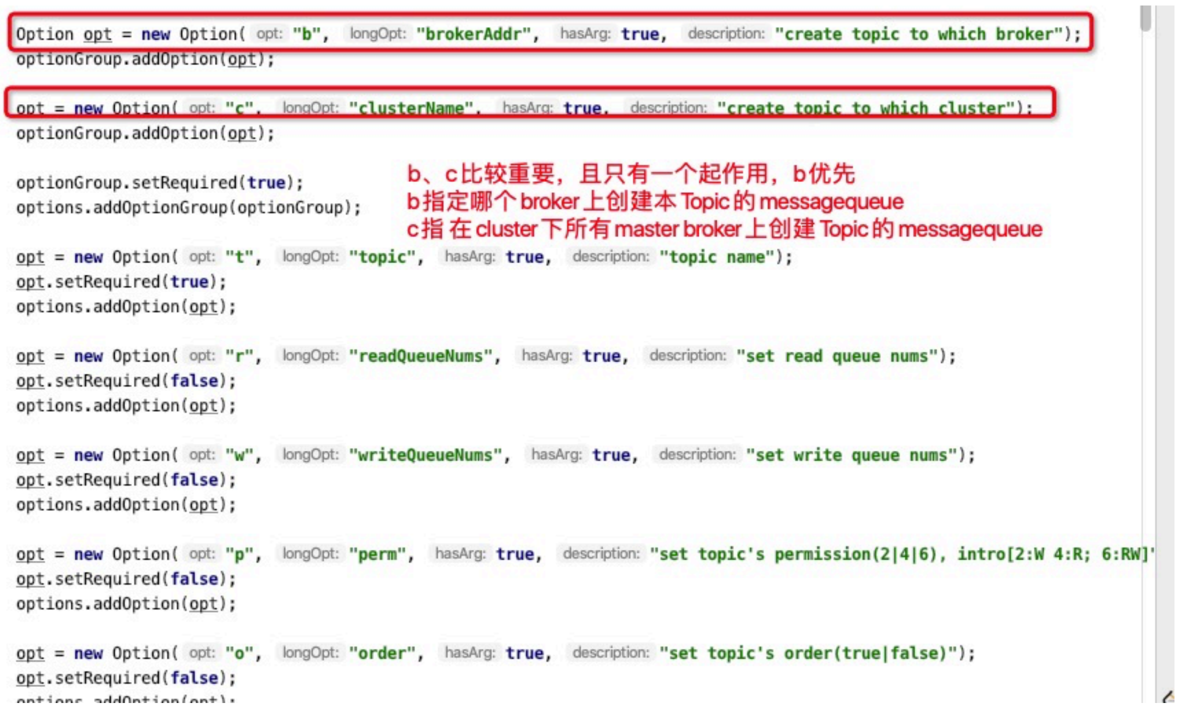
```java
this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

    @Override
    public void run() {
        NamesrvController.this.routeInfoManager.scanNotActiveBroker();
    }
}, initialDelay: 5, period: 10, TimeUnit.SECONDS);
```



```java
public void scanNotActiveBroker() {
    Iterator<Entry<String, BrokerLiveInfo>> it = this.brokerLiveTable.entrySet().iterator();
    while (it.hasNext()) {
        Entry<String, BrokerLiveInfo> next = it.next();
        long last = next.getValue().getLastUpdateTimestamp();
        if ((last + BROKER_CHANNEL_EXPIRED_TIME) < System.currentTimeMillis()) {
            RemotingUtil.closeChannel(next.getValue().getChannel());
            it.remove();
            log.warn("The broker channel expired, {} {}ms", next.getKey(), BROKER_CHANNEL_EXPIRED_TIME);
            this.onChannelDestroy(next.getKey(), next.getValue().getChannel());
        }
    }
}
```

各个角色间交互流程：

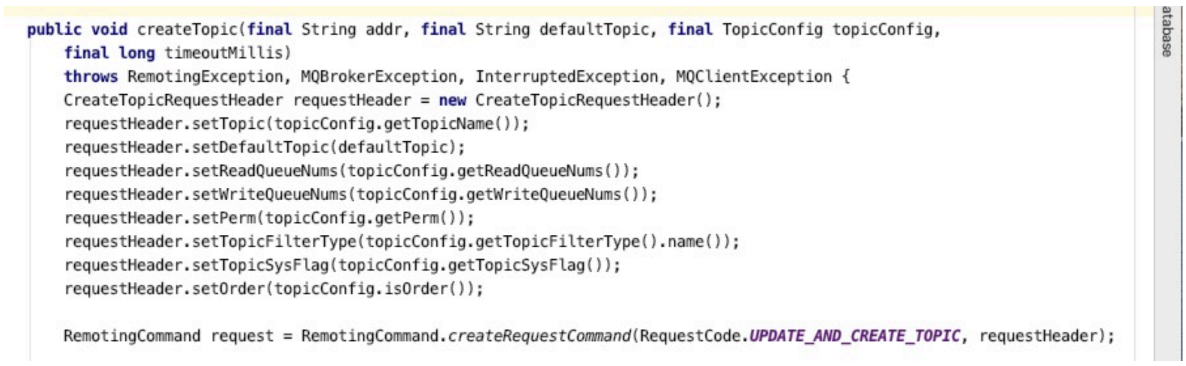
创建 Topic：
org.apache.rocketmq.tools.command.topic.UpdateTopicSubCommand



```
Option opt = new Option( opt: "b",  longOpt: "brokerAddr",  hasArg: true,  description: "create topic to which broker");
optionGroup.addOption(opt);

opt = new Option( opt: "c",  longOpt: "clusterName",  hasArg: true,  description: "create topic to which cluster");
optionGroup.addOption(opt);

optionGroup.setRequired(true);
options.addOptionGroup(optionGroup);

opt = new Option( opt: "t",  longOpt: "topic",  hasArg: true,  description: "topic name");
opt.setRequired(true);
options.addOption(opt);

opt = new Option( opt: "r",  longOpt: "readQueueNums",  hasArg: true,  description: "set read queue nums");
opt.setRequired(false);
options.addOption(opt);

opt = new Option( opt: "w",  longOpt: "writeQueueNums",  hasArg: true,  description: "set write queue nums");
opt.setRequired(false);
options.addOption(opt);

opt = new Option( opt: "p",  longOpt: "perm",  hasArg: true,  description: "set topic's permission(2|4|6), intro[2:W 4:R; 6:RW]"
opt.setRequired(false);
options.addOption(opt);

opt = new Option( opt: "o",  longOpt: "order",  hasArg: true,  description: "set topic's order(true|false)");
opt.setRequired(false);
options.addOption(opt);
```

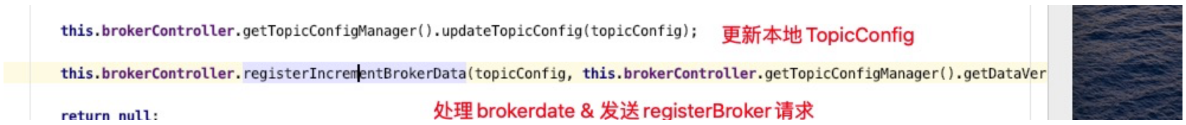
b、c比较重要，且只有一个起作用，b优先
b指定哪个 broker 上创建本 Topic 的 messagequeue
c指 在 cluster 下所有 master broker 上创建 Topic 的 messagequeue

org.apache.rocketmq.client.impl.MQClientAPIImpl#createTopic



```
public void createTopic(final String addr, final String defaultTopic, final TopicConfig topicConfig,
    final long timeoutMillis)
    throws RemotingException, MQBrokerException, InterruptedException, MQClientException {
    CreateTopicRequestHeader requestHeader = new CreateTopicRequestHeader();
    requestHeader.setTopic(topicConfig.getTopicName());
    requestHeader.setDefaultTopic(defaultTopic);
    requestHeader.setReadQueueNums(topicConfig.getReadQueueNums());
    requestHeader.setWriteQueueNums(topicConfig.getWriteQueueNums());
    requestHeader.setPerm(topicConfig.getPerm());
    requestHeader.setTopicFilterType(topicConfig.getTopicFilterType().name());
    requestHeader.setTopicSysFlag(topicConfig.getTopicSysFlag());
    requestHeader.setOrder(topicConfig.isOrder());

    RemotingCommand request = RemotingCommand.createRequestCommand(RequestCode.UPDATE_AND_CREATE_TOPIC, requestHeader);
```

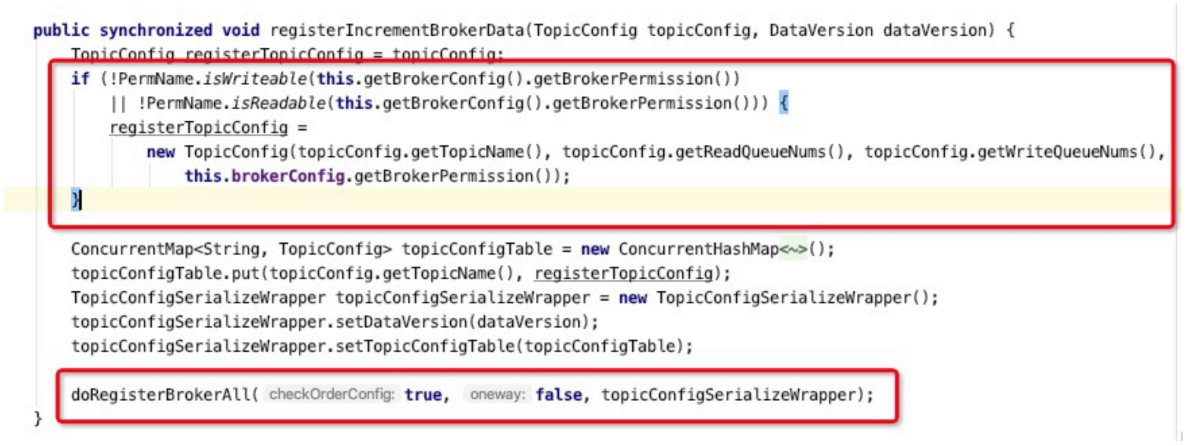
创建命令被发到对应 broker，org.apache.rocketmq.broker.processor.AdminBrokerProcessor#updateAndCreateTopic



```
this.brokerController.getTopicConfigManager().updateTopicConfig(topicConfig);

this.brokerController.registerIncrementBrokerData(topicConfig, this.brokerController.getTopicConfigManager().getDataVer

return null;
```

更新本地 TopicConfig

处理 brokerdate & 发送 registerBroker 请求



```
public synchronized void registerIncrementBrokerData(TopicConfig topicConfig, DataVersion dataVersion) {
    TopicConfig registerTopicConfig = topicConfig;
    if (!PermName.isWriteable(this.getBrokerConfig().getBrokerPermission())
        || !PermName.isReadable(this.getBrokerConfig().getBrokerPermission())) {
        registerTopicConfig =
            new TopicConfig(topicConfig.getTopicName(), topicConfig.getReadQueueNums(), topicConfig.getWriteQueueNums(),
                this.brokerConfig.getBrokerPermission());
    }

    ConcurrentMap<String, TopicConfig> topicConfigTable = new ConcurrentHashMap<~>();
    topicConfigTable.put(topicConfig.getTopicName(), registerTopicConfig);
    TopicConfigSerializeWrapper topicConfigSerializeWrapper = new TopicConfigSerializeWrapper();
    topicConfigSerializeWrapper.setDataVersion(dataVersion);
    topicConfigSerializeWrapper.setTopicConfigTable(topicConfigTable);

    doRegisterBrokerAll( checkOrderConfig: true,  oneway: false, topicConfigSerializeWrapper);
}
```

最终 在 org.apache.rocketmq.namesrv.routeinfo.RouteInfoManager#registerBroker 中注册 Broker 信息
增加 / 更新 QueueData

为什么不用 ZooKeeper?

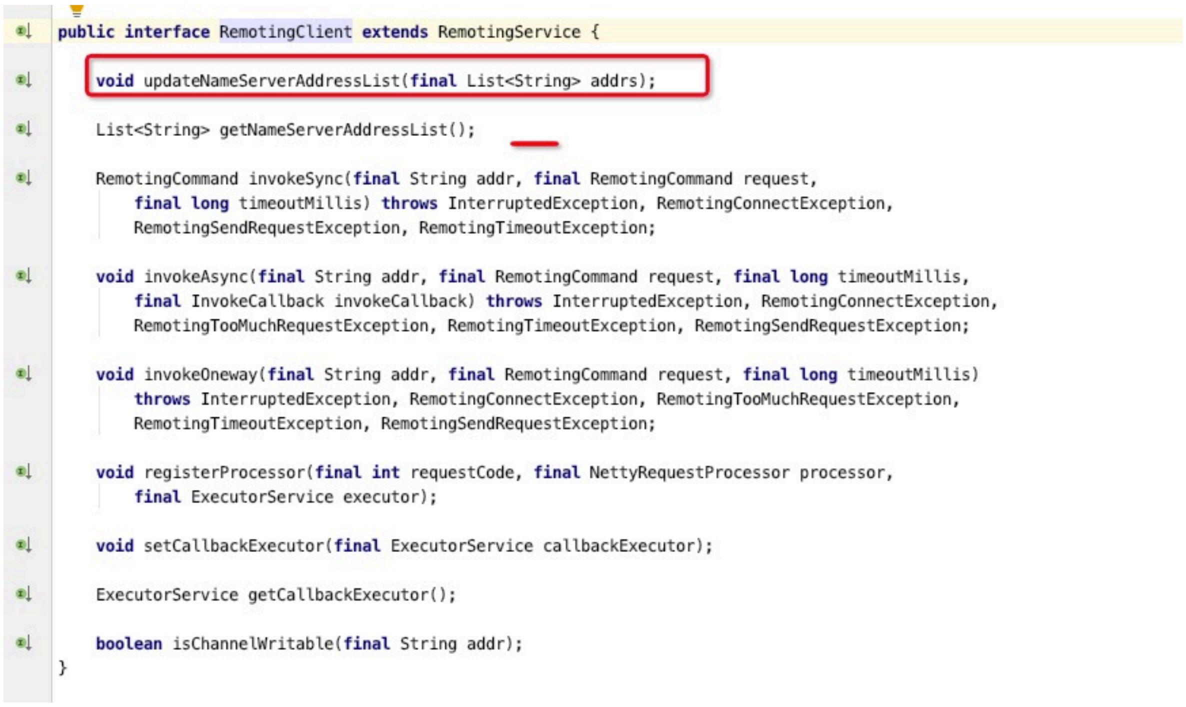
rocketMq 架构不需要 master 选举等 zookeeper 提供的复杂功能，只是需要一个轻量级的元数据服务器
不需要再依赖另一个中间件，减少维护成本

底层通信机制：

RocketMq 通信主要是在 Remoting 模块， 类结构图如下

RemotingService 为最上层借口， 主要有 3 个方法
start
shutdwon
registerRPCHook(RPCHook rpcHook);

RemotingClient 和 RemotingServer 分别继承 RemotingService 接口， 并增加了特有方法
例如 RemotingClient

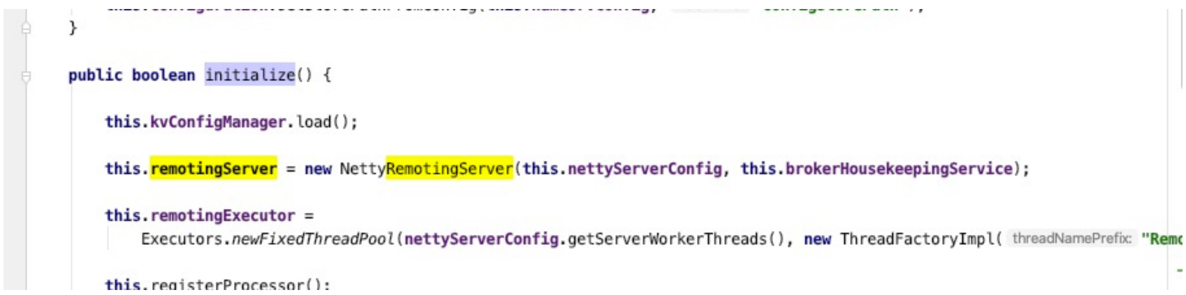


```java
public interface RemotingClient extends RemotingService {

    void updateNameServerAddressList(final List<String> addrs);

    List<String> getNameServerAddressList();

    RemotingCommand invokeSync(final String addr, final RemotingCommand request,
        final long timeoutMillis) throws InterruptedException, RemotingConnectException,
        RemotingSendRequestException, RemotingTimeoutException;

    void invokeAsync(final String addr, final RemotingCommand request, final long timeoutMillis,
        final InvokeCallback invokeCallback) throws InterruptedException, RemotingConnectException,
        RemotingTooMuchRequestException, RemotingTimeoutException, RemotingSendRequestException;

    void invokeOneway(final String addr, final RemotingCommand request, final long timeoutMillis)
        throws InterruptedException, RemotingConnectException, RemotingTooMuchRequestException,
        RemotingTimeoutException, RemotingSendRequestException;

    void registerProcessor(final int requestCode, final NettyRequestProcessor processor,
        final ExecutorService executor);

    void setCallbackExecutor(final ExecutorService callbackExecutor);

    ExecutorService getCallbackExecutor();

    boolean isChannelWritable(final String addr);
}
```

再到具体的实现类， NettyRemotingClient 和 NettyRemotingServer 分别实现了上面两个类，继承了 NettyRemotingAbstract 类

通过这样的封装，rocketMq 各个模块的通信，可以通过统一格式（RemotingCommand）完成，

例如： NameServerController 有一个 remotingServer 变量，NameServer 启动时初始化各个变量，然后启动 remotingServer 即可，NameServer 只需要专心实现 RemotingCommand 的逻辑



```java
}

public boolean initialize() {

    this.kvConfigManager.load();

    this.remotingServer = new NettyRemotingServer(this.nettyServerConfig, this.brokerHousekeepingService);

    this.remotingExecutor =
        Executors.newFixedThreadPool(nettyServerConfig.getServerWorkerThreads(), new ThreadFactoryImpl( threadNamePrefix: "Remo

    this.registerProcessor();
```