图灵：楼兰

你的神秘技术宝藏

这一部分，我们开始深入RocketMQ的源码。源码的解读是个非常困难的过程，每个人的理解程度都会不一样，也不太可能通过讲解把其中的细节全部讲明白。我们今天在解读源码时，采取逐层抽取的模式，希望能够给大家形成一个源码解读的大框架，帮助大家对源码形成自己的理解。

我们分为几条主线来解读源码：

一、源码环境搭建

1、源码拉取：

RocketMQ的官方Git仓库地址：https://github.com/apache/rocketmq 可以用git把项目clone下来或者直接下载代码包。

也可以到RocketMQ的官方网站上下载指定版本的源码： http://rocketmq.apache.org/dowloading/releases/

4.9.1 release

- Released Aug 22, 2021
- Release Notes
- Source: rocketmq-all-4.9.1-source-release.zip [PGP] [SHA512]
- Binary: rocketmq-all-4.9.1-bin-release.zip [PGP] [SHA512]

> 下载后就可以解压导入到IDEA中进行解读了。我们只要注意下是下载的4.9.1版本就行了。

源码下很多的功能模块，很容易让人迷失方向，我们只关注下几个最为重要的模块：

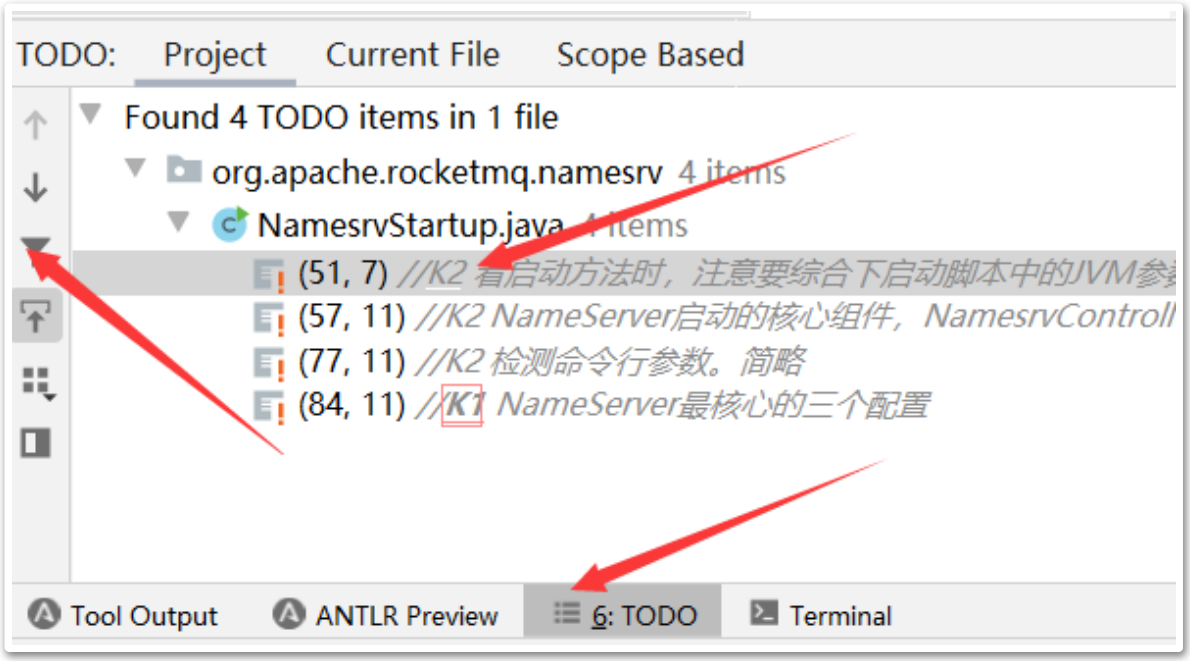
- broker: Broker 模块（broke 启动进程）
- client ： 消息客户端，包含消息生产者、消息消费者相关类
- example: RocketMQ 例代码
- namesrv：NameServer模块
- store：消息存储模块
- remoting：远程访问模块

> 各个模块的功能大都从名字上就能看懂。我们可以在有需要的时候再进去看源码。
>
> 但是这些模块有些东西还是要关注的。例如docs文件夹下的文档，以及各个模块下都有非常丰富的junit测试代码，这些都是非常有用的。

2、注解版源码引入

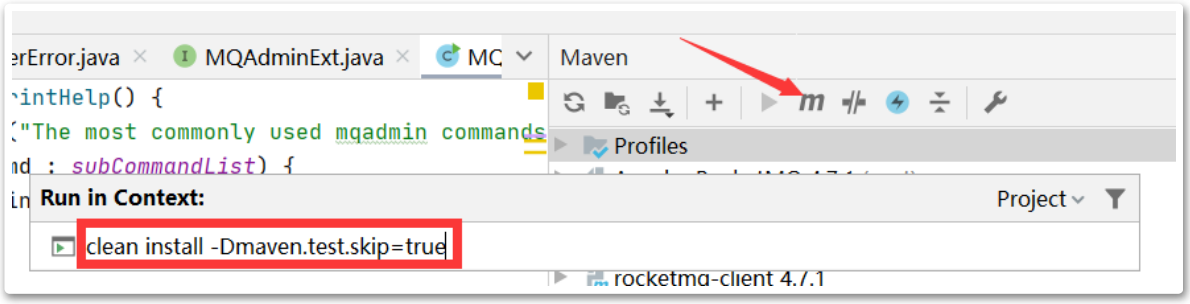
RocketMQ的源码中有个非常让人头疼的事情，就是他的代码注释几乎没有。为了帮助大家解读源码，我给大家准备了一个添加了自己注释的源码版本。 在配套资料当中。大家可以把这个版本导入IDEA来进行解读。

源码中对最为重要的注解设定了一个标记K1，相对不那么重要的注解设定了一个标记K2，而普通的注释就没有添加标记。大家可以在IDEA的TODO标签中配置这两个注解标记。



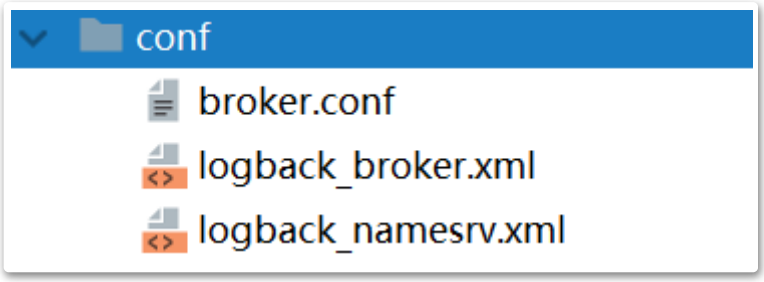
3、源码调试：

将源码导入IDEA后，需要先对源码进行编译。编译指令 clean install -Dmaven.test.skip=true



编译完成后就可以开始调试代码了。调试时需要按照以下步骤：

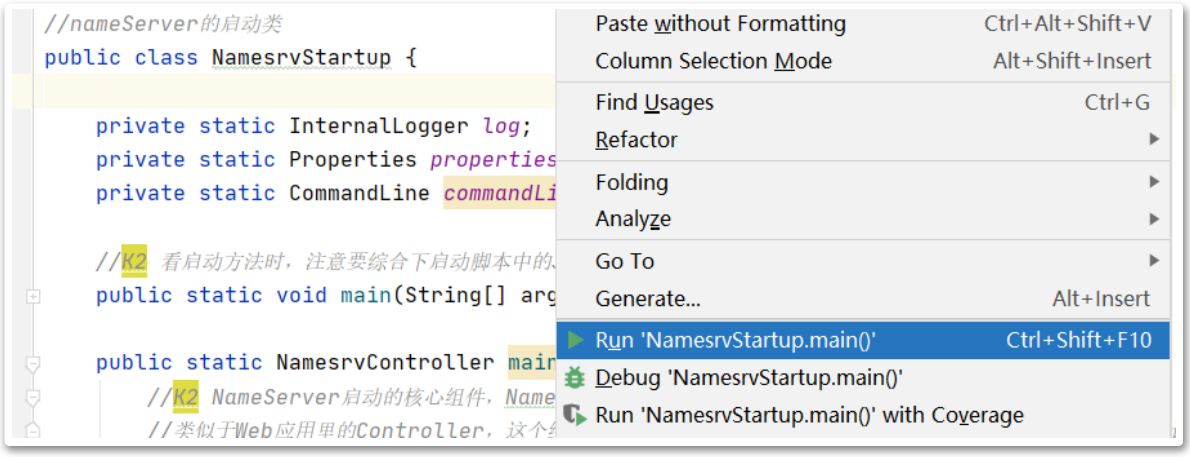
调试时，先在项目目录下创建一个conf目录，并从`distribution`拷贝`broker.conf`和`logback_broker.xml`和`logback_namesrv.xml`



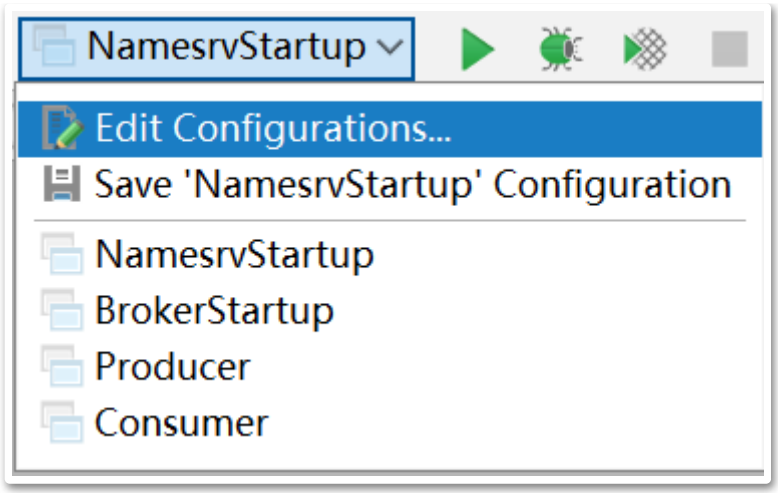
> 注解版源码中已经复制好了。

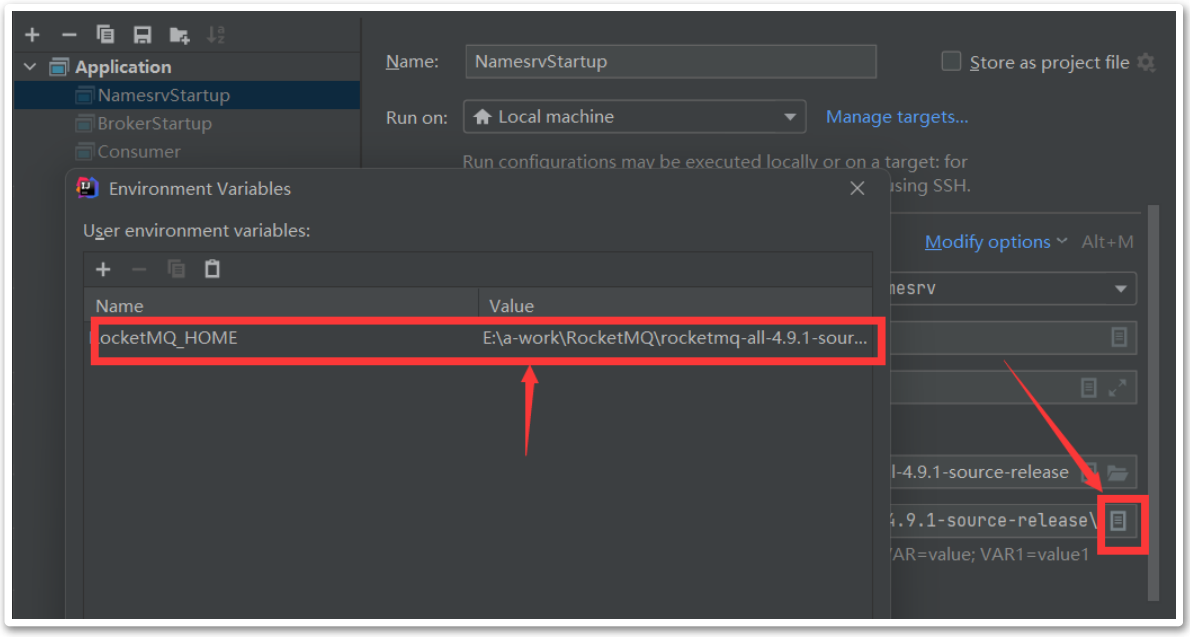
3.1 启动nameServer

展开namesrv模块，运行NamesrvStartup类即可启动NameServer



启动时，会报错，提示需要配置一个ROCKETMQ_HOME环境变量。这个环境变量我们可以在机器上配置，跟配置JAVA_HOME环境变量一样。也可以在IDEA的运行环境中配置。目录指向源码目录即可。





配置完成后，再次执行，看到以下日志内容，表示NameServer启动成功

```
The Name Server boot success. serializeType=JSON
```

3.2 启动Broker

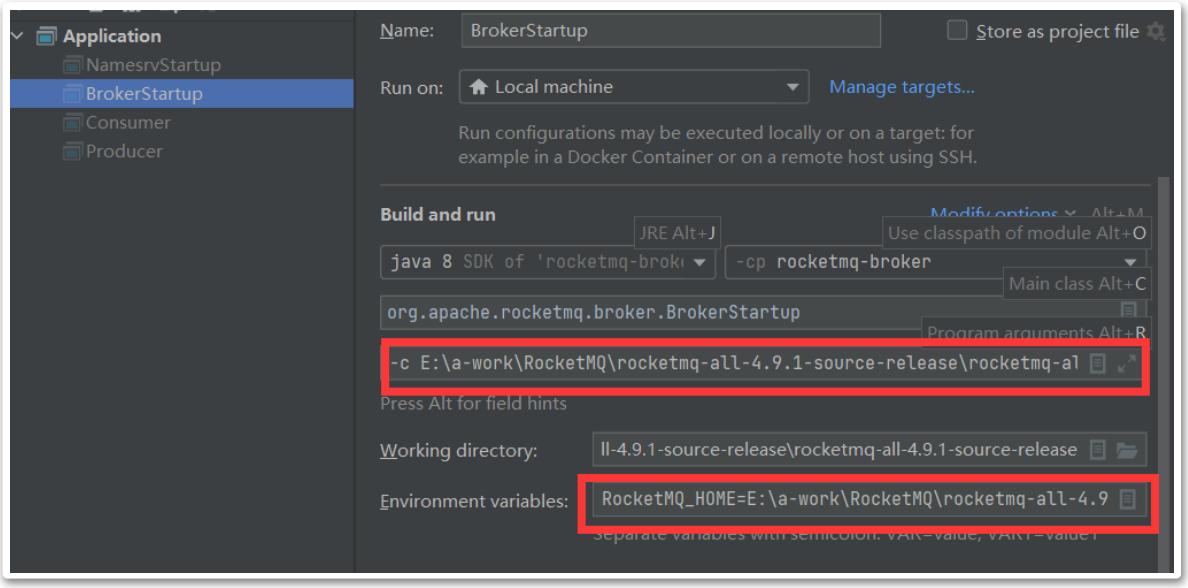
启动Broker之前，我们需要先修改之前复制的broker.conf文件

```
brokerClusterName = DefaultCluster
brokerName = broker-a
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH

# 自动创建Topic
autoCreateTopicEnable=true
# nameServ地址
```

```
namesrvAddr=127.0.0.1:9876
# 存储路径
storePathRootDir=E:\\RocketMQ\\data\\rocketmq\\dataDir
# commitLog路径
storePathCommitLog=E:\\RocketMQ\\data\\rocketmq\\dataDir\\commitlog
# 消息队列存储路径
storePathConsumeQueue=E:\\RocketMQ\\data\\rocketmq\\dataDir\\consumequeue
# 消息索引存储路径
storePathIndex=E:\\RocketMQ\\data\\rocketmq\\dataDir\\index
# checkpoint文件路径
storeCheckpoint=E:\\RocketMQ\\data\\rocketmq\\dataDir\\checkpoint
# abort文件存储路径
abortFile=E:\\RocketMQ\\data\\rocketmq\\dataDir\\abort
```

然后Broker的启动类是broker模块下的BrokerStartup。

启动Broker时，同样需要ROCETMQ_HOME环境变量，并且还需要配置一个-c 参数，指向broker.conf配置文件。



然后重新启动，即可启动Broker。

3.3 发送消息

在源码的example模块下，提供了非常详细的测试代码。例如我们启动example模块下的org.apache.rocketmq.example.quickstart.Producer类即可发送消息。

但是在测试源码中，需要指定NameServer地址。这个NameServer地址有两种指定方式，一种是配置一个NAMESRV_ADDR的环境变量。另一种是在源码中指定。我们可以在源码中加一行代码指定NameServer

```
producer.setNamesrvAddr("127.0.0.1:9876");
```

然后就可以发送消息了。

3.4 消费消息

我们可以使用同一模块下的org.apache.rocketmq.example.quickstart.Consumer类来消费消息。运行时同样需要指定NameServer地址

```
consumer.setNamesrvAddr("192.168.232.128:9876");
```

这样整个调试环境就搭建好了。

3.5 如何看源码

下面就可以一边调试一边看源码了。源码中大部分关键的地方都已经添加了注释，文档中就不做过多记录了。

我们在看源码的时候，要注意，不要一看源码就一行行代码都逐步看，更不要期望一遍就把代码给看明白。这样会陷入到代码的复杂细节中，瞬间打击到放弃。

建议的读源码的方式是以少数几个简单案例为基础，先把主线流程读清楚，对于源码的整体结构先形成一个整体的印象。然后接下来，是找准感兴趣的问题点，带着问题去读源码。例如上一章节中介绍了非常多RocketMQ的底层原理，这些原理都可以拿到源码中一一进行验证。在不断验证的过程中，加深对于源码的理解。

另外，RocketMQ的源码不同于很多国外的开源软件，虽然注释比较少，但是整个设计思想其实是很符合国人思维的。所以看RocketMQ的源码，相对会比较顺畅。而且，对于Netty服务调用、文件读写等等很多比较通用的问题，源码当中的很多实用经验是非常有帮助价值的。

接下来，我们会从几条主要业务线入手，来逐步理解RocketMQ的源码。

二、服务启动过程

2.1、NameServer的启动过程

1、关注重点

从之前的介绍中，我们已经了解到，在RocketMQ中，实际进行消息存储、推送等核心功能的是Broker。那NameServer具体做什么用呢？NameServer的核心作用其实就只有两个

- 一是维护Broker的服务地址并进行及时的更新。
- 二是给Producer和Consumer提供服务获取Broker列表。

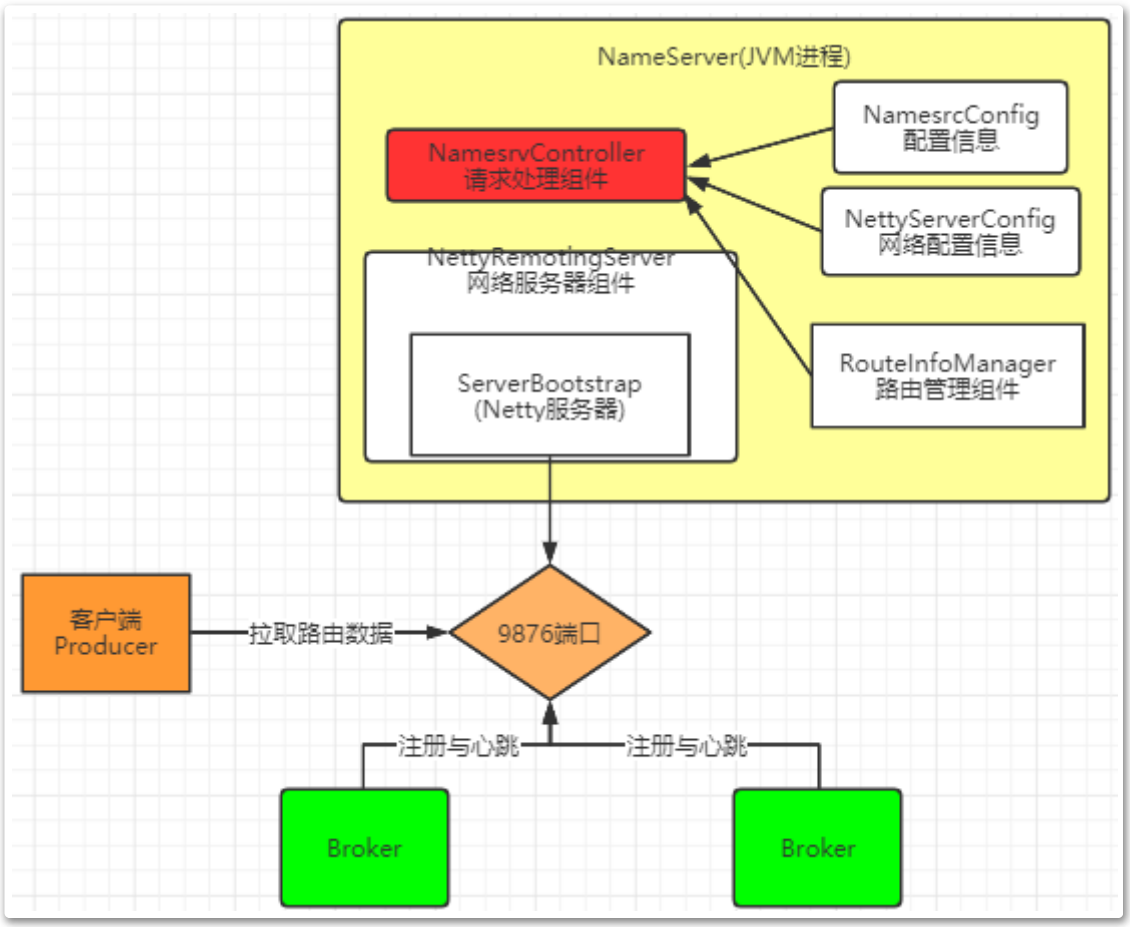
NameServer的启动入口为NamesrvStartup类的main方法，我们可以进行逐步调试。这次看源码，我们不要太过陷入其中的细节，我们的目的是先搞清楚NameServer的大体架构。

2、源码重点

整个NameServer的核心就是一个NamesrvController对象。这个controller对象就跟java Web开发中的Controller功能类似，都是响应客户端请求的。

在Controller的启动以及关闭过程中，会逐步启动RocketMQ的各种内部服务。要注意对这些关键服务的梳理。

从启动和关闭这两个关键步骤，我们可以总结出NameServer的组件其实并不是很多，整个NameServer的结构是这样：



2.2、Broker的启动过程

1、关注重点

Broker是整个RocketMQ的业务核心，所有消息存储、转发这些最为重要的业务都是在Broker中进行处理的。

Broker的内部架构，有点类似于JavaWeb开发的MVC架构。有Controller负责响应请求，各种Service组件负责具体业务，然后还有负责消息存盘的功能模块则类似于Dao。

第一轮看源码，重点依然是，是通过Broker的启动过程，观察总结出Broker的内部服务。

2、源码重点

Broker启动的入口在BrokerStartup这个类，可以从他的main方法开始调试。

启动过程关键点：重点也是围绕一个BrokerController对象，先创建，然后再启动。

首先：在BrokerStartup.createBrokerController方法中可以看到Broker的几个核心配置：

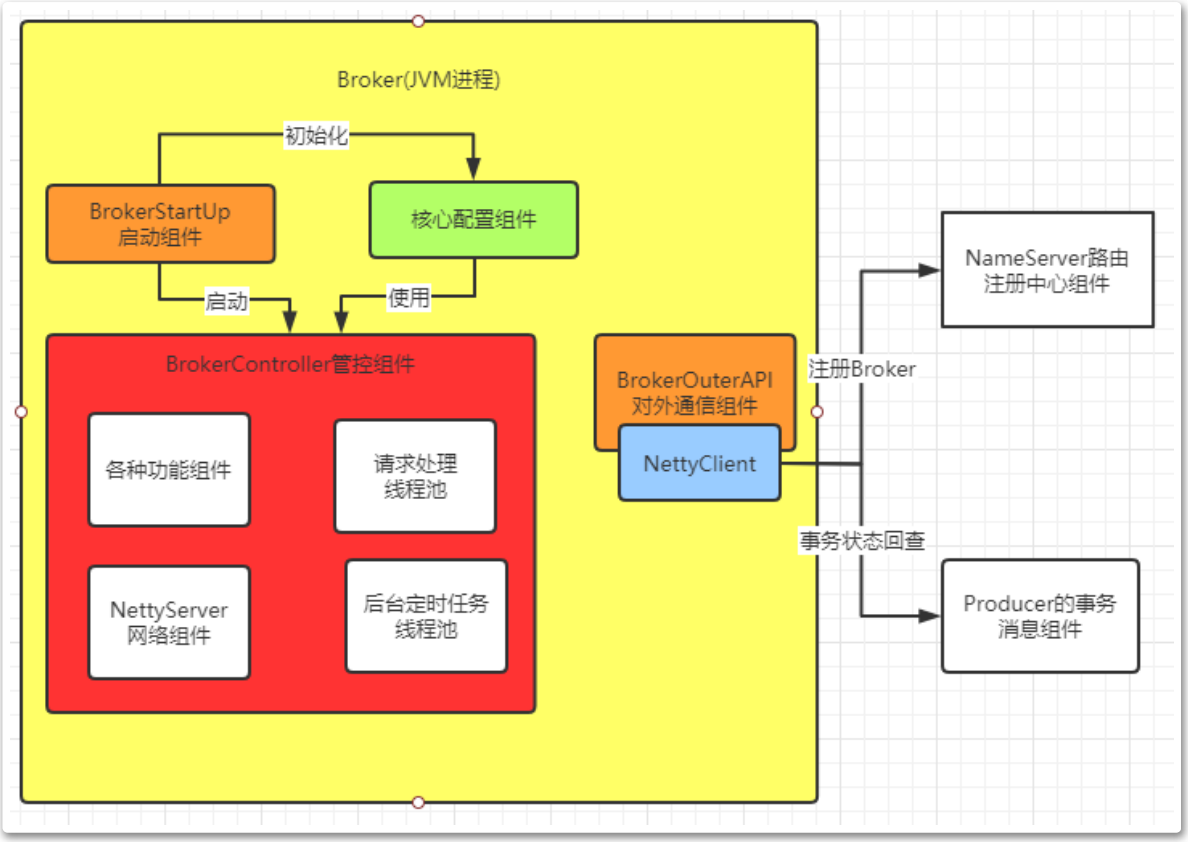
- BrokerConfig
- NettyServerConfig ：Netty服务端占用了10911端口。同样也可以在配置文件中覆盖。
- NettyClientConfig
- MessageStoreConfig

然后：在BrokerController.start方法可以看到启动了一大堆Broker的核心服务，我们挑一些重要的

```
this.messageStore.start();启动核心的消息存储组件

this.remotingServer.start();
this.fastRemotingServer.start(); 启动两个Netty服务

this.brokerOuterAPI.start();启动客户端，往外发请求

BrokerController.this.registerBrokerAll:  向NameServer注册心跳。

this.brokerStatsManager.start();
this.brokerFastFailure.start();这也是一些负责具体业务的功能组件
```

我们现在不需要了解这些核心组件的具体功能，只要有个大概，Broker中有一大堆的功能组件负责具体的业务。后面等到分析具体业务时再去深入每个服务的细节。

我们需要抽象出Broker的一个整体结构：



三、客户端主要业务

3.1、Netty服务注册框架

1、功能回顾

网络通信服务是构建分布式应用的基础，也是我们去理解RocketMQ底层业务的基础。

RocketMQ使用Netty框架提供了一套基于服务码的服务注册机制，让各种不同的组件都可以按照自己的需求，注册自己的服务方法。RocketMQ的这一套服务注册机制，是非常简洁使用的。在使用Netty进行其他相关应用开发时，都可以借鉴他的这一套服务注册机制。例如要开发一个大型的IM项目，要加减好友、发送文本，图片，甚至红包、维护群聊信息等等各种各样的请求，这些请求如何封装，就可以很好的参考这个框架。

Netty的所有远程通信功能都由remoting模块实现。RemotingServer模块里包含了RPC的服务端RemotingServer以及客户端RemotingClient。在RocketMQ中，涉及到的远程服务非常多，在RocketMQ中，NameServer主要是RPC的服务端RemotingServer，Broker对于客户端来说，是RPC的服务端RemotingServer，而对于NameServer来说，又是RPC的客户端。各种Client是RPC的客户端RemotingClient。

需要理解的是，RocketMQ基于Netty保持客户端与服务端的长连接Channel。只要Channel是稳定的，那么即可以从客户端发请求到服务端，同样服务端也可以发请求到客户端。例如在事务消息场景中，就需要Broker多次主动向Producer发送请求确认事务的状态。所以，RemotingServer和RemotingClient都需要注册自己的服务。

2、源码重点：

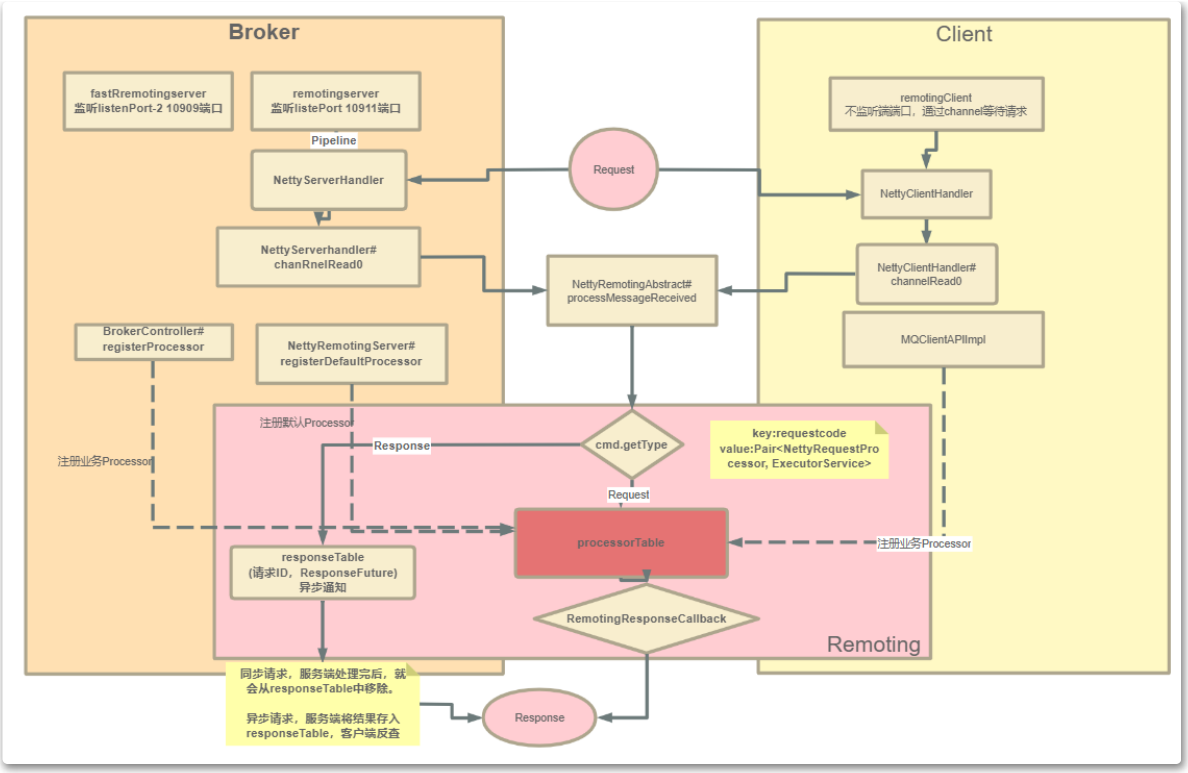
1、所有的请求都封装成RemotingCommand对象。而每个处理消息的服务逻辑，会封装成一个NettyRequestProcessor对象。

2、服务端和客户端都维护了一个processorTable，这是个HashMap，key是服务码requestCode，value是对应的运行单元。Pair<NettyRequestProcessor, ExecutorService> 类型，包含了处理逻辑Prcessor和执行线程池ExecutorService

3、服务端的注册BrokerController.registerProcessor()，客户端的服务注册见MQClientAPIImpl。NameServer则会注册一个大的DefaultRequestProcessor，统一处理所有的服务。

4、理解服务注册流程

整体服务加载流程如下图：



> 深度解析：尝试去理解一下RemotingCommand的序列化协议。NettyEncoder和NettyDecoder。
>
> RocketMQ的序列化协议还是比较复杂的，你可以尝试简化一下序列化方法，比如使用JSON字符串来做序列化。

补充：关于RocketMQ的同步结果推送与异步结果推送

RocketMQ的RemotingServer服务端，会维护一个responseTable，这是一个线程同步的Map结构。 key为请求的ID，value是异步的消息结果。 ConcurrentMap<Integer /* opaque */, ResponseFuture> 。

处理同步请求(NettyRemotingAbstract#invokeSyncImpl)时，处理的结果会存入responseTable，通过ResponseFuture提供一定的服务端异步处理支持，提升服务端的吞吐量。 请求返回后，立即从responseTable中移除请求记录。

处理异步请求(NettyRemotingAbstract#invokeAsyncImpl)时，处理的结果依然会存入responsTable，等待客户端后续再来请求结果。但是他保存的依然是一个ResponseFuture，也就是在客户端请求结果时再去获取真正的结果。 另外，在RemotingServer启动时，会启动一个定时的线程任务，不断扫描responseTable，将其中过期的response清除掉。

3.2、Broker心跳注册过程

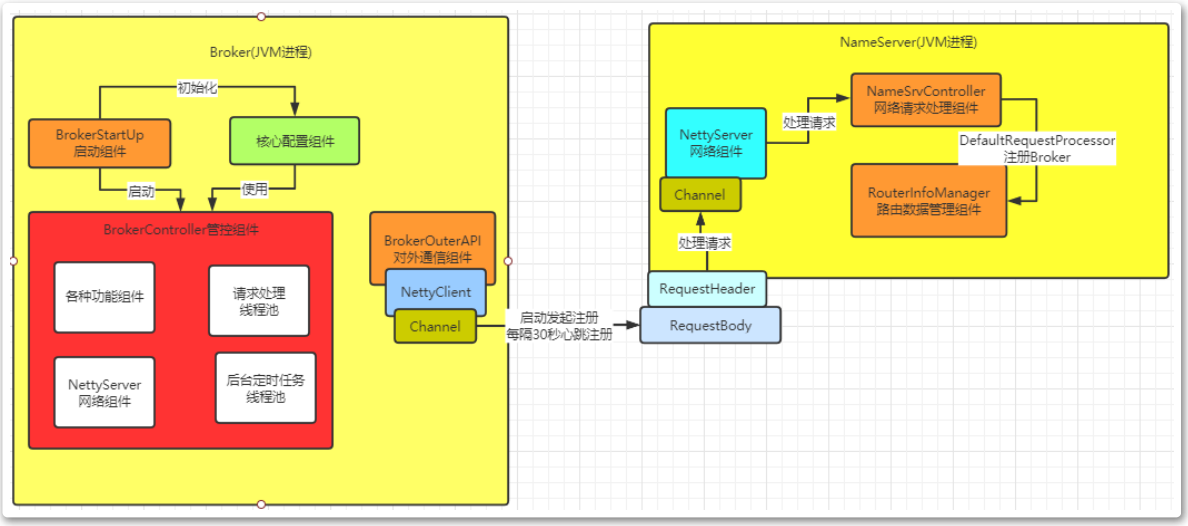
1、关注重点

在之前我们已经介绍到了。Broker会在启动时向NameServer注册自己的服务信息，并且会定时的往NameServer发送心跳信息。而NameServer会维护Broker的路由列表，并对路由列表进行实时更新。这一轮就重点梳理这个问题。

2、源码重点

BrokerController.this.registerBrokerAll方法会发起向NameServer注册心跳。启动时会立即注册，同时也会启动一个线程池，以10秒延迟，默认30秒的间隔 持续向NameServer发送心跳。

BrokerController.this.registerBrokerAll这个方法就是注册心跳的入口。



然后，在NameServer中也会启动一个定时任务，扫描不活动的Broker。具体观察NamesrvController.initialize方法

3.3、Producer发送消息过程

1、关注重点

首先回顾下我们之前的Producer使用案例。

Producer有两种：

- 一种是普通发送者：DefaultMQProducer。只负责发送消息，发送完消息，就可以停止了。
- 另一种是事务消息发送者： TransactionMQProducer。支持事务消息机制。需要在事务消息过程中提供事务状态确认的服务，这就要求事务消息发送者虽然是一个客户端，但是也要完成整个事务消息的确认机制后才能退出。

整个Producer的流程，大致分两个步骤

- start方法，进行一大堆的准备工作.
- 各种各样的send方法，进行消息发送。

那我们重点关注以下几个问题：

1、 我们关注下Producer的核心启动流程以及两种消息发送者的区别。

在mQClientFactory的start方法中，启动了生产者的一大堆重要服务。

然后在DefaultMQProducerImpl的start方法中，又回到了生产者的mqClientFactory的启动过程，这中间有服务状态的管理。这里RocketMQ的所有客户端实例，包括生产者和消费者，都是统一交由mQClientFactory组件来启动，也就是说，所有客户端的启动流程是固定的，不同客户端的区别只是在于他们在启动前注册的一些信息不同。例如生产者注册到producerTable，消费者注册到consumerTable，管理控制端注册到adminExtTable

2、Producer如何管理Borker路由信息： Producer需要拉取Broker列表，然后跟Broker建立连接等等很多核心的流程，其实都是在发送消息时建立的。因为在启动时，还不知道要拉取哪个Topic的Broker列表呢。所以对于这个问题，我们关注的重点，不应该是start方法，而是send方法。

3、 关于Producer的负载均衡。

在之前介绍RocketMQ的顺序消息时，讲到了Producer的负载均衡策略，默认会把消息平均的发送到所有MessageQueue里的。那到底是怎么进行负载均衡的呢？

4、 在发送Netty请求时，如何制定Broker？实际上是指定的MessageQueue，而不是Topic。Topic只是用来找MessageQueue。

2、源码重点

1、 Producer的核心启动流程以及两种消息发送者的区别。

所有Producer的启动过程，最终都会调用DefaultMQProducerImpl#start方法。在start方法中的通过一个mQClientFactory对象，启动生产者的一大堆重要服务。

这里其实就是一种设计模式，虽然有很多种不同的客户端，但是这些客户端的启动流程最终都是统一的，全是交由mQClientFactory对象来启动。而不同之处在于这些客户端在启动过程中，按照服务端的要求注册不同的信息。例如生产者注册到producerTable，消费者注册到consumerTable，管理控制端注册到adminExtTable

然后关于两种消息发送者：

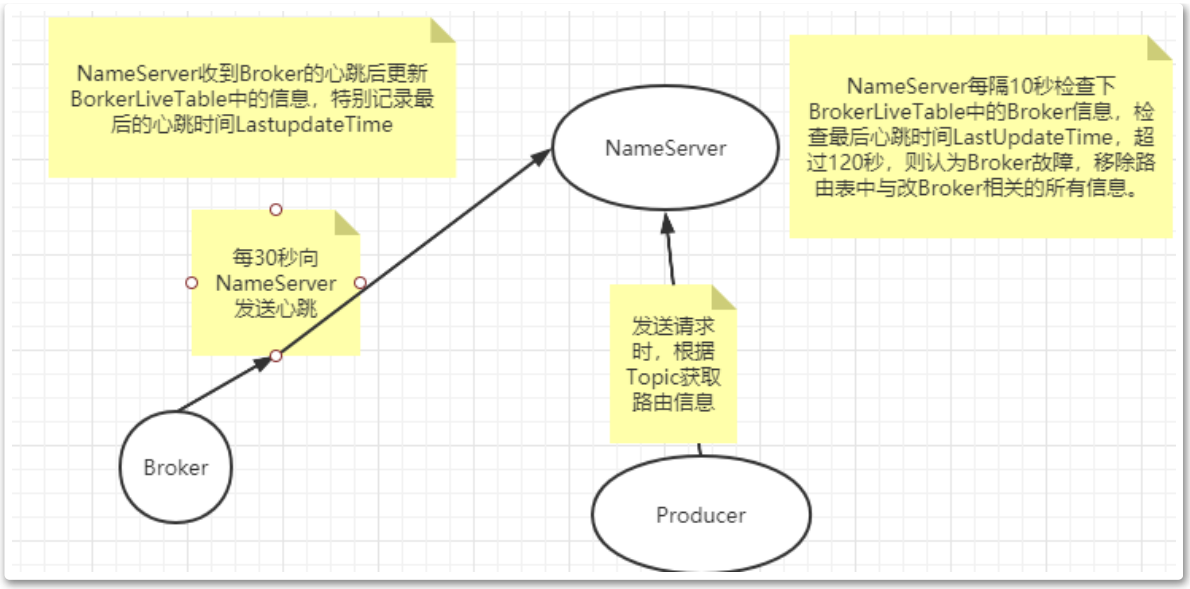
- DefaultMQProducer只需要构建一个Netty客户端，往Broker发送消息就行了。注意，异步回调只是在Producer接收到Broker的响应后自行调整流程，不需要提供Netty服务。
- TransactionMQProducer由于需要在事务消息机制中给Broker提供状态确认服务，所以在发送消息的同时，还需要保持连接，提供服务。在TransactionMQProducer的启动过程中，会往RemotingClient中注册相应的Processor，这样RemotingServer和RemotingClient之间就可以通过channel进行双向的服务请求了。

2、Producer如何管理Borker路由信息?

Producer需要拉取Broker列表，然后跟Broker建立连接等等很多核心的流程，其实都是在发送消息时建立的。因为在启动时，还不知道要拉取哪个Topic的Broker列表呢。所以对于这个问题，我们关注的重点，不应该是start方法，而是send方法。而对NameServer的地址管理，则是散布在启动和发送的多个过程当中，并且NameServer地址可以通过一个Http服务来获取。Send方法中，首先需要获得Topic的路由信息。这会从本地缓存中获取，如果本地缓存中没有，就从NameServer中去申请。

核心在org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl#tryToFindTopicPublishInfo方法

路由信息大致的管理流程：



3、 关于Producer的负载均衡。

Producer在获取路由信息后，会选出一个MessageQueue去发送消息。这个选MessageQueue的方法就是一个索引自增然后取模的方式。

```
erImpl.java × MQFaultStrategy.java × TopicPublishInfo.java × LatencyFaultTolerance.java × LatencyFaultToleranceImp
//Producer选择MessageQueue的方法
public MessageQueue selectOneMessageQueue(final TopicPublishInfo tpInfo, final String lastBrokerName) {
    if (this.sendLatencyFaultEnable) {
        try {
            //K2 这里可以看到，Producer选择MessageQueue的方法就是自增，然后取模。并且只有这一种方法。
            int index = tpInfo.getSendWhichQueue().getAndIncrement();
            for (int i = 0; i < tpInfo.getMessageQueueList().size(); i++) {
                int pos = Math.abs(index++) % tpInfo.getMessageQueueList().size();
                if (pos < 0)
                    pos = 0;
                MessageQueue mq = tpInfo.getMessageQueueList().get(pos);
                if (latencyFaultTolerance.isAvailable(mq.getBrokerName())) {
                    if (null == lastBrokerName || mq.getBrokerName().equals(lastBrokerName))
                        return mq;
                }
            }

            final String notBestBroker = latencyFaultTolerance.pickOneAtLeast();
            int writeQueueNums = tpInfo.getQueueIdByBroker(notBestBroker);
            if (writeQueueNums > 0) {
                //这里计算也还是自增取模
                final MessageQueue mq = tpInfo.selectOneMessageQueue();
                if (notBestBroker != null) {
                    mq.setBrokerName(notBestBroker);
                    mq.setQueueId(tpInfo.getSendWhichQueue().getAndIncrement() % writeQueueNums);
                }
                return mq;
```

然后根据MessageQueue再找所在的Broker，往Broker发送请求。

3.4、Consumer拉取消息过程

1、关注重点

结合我们之前的示例，回顾下消费者这一块的几个重点问题：

- 消费者也是有两种，推模式消费者和拉模式消费者。优秀的MQ产品都会有一个高级的目标，就是要提升整个消息处理的性能。而要提升性能，服务端的优化手段往往不够直接，最为直接的优化手段就是对消费者进行优化。所以在RocketMQ中，整个消费者的业务逻辑是非常复杂的，甚至某种程度上来说，比服务端更复杂，所以，在这里我们重点关注用得最多的推模式的消费者。
- 消费者组之间有集群模式和广播模式两种消费模式。我们就要了解下这两种集群模式是如何做的逻辑封装。
- 然后我们关注下消费者端的负载均衡的原理。即消费者是如何绑定消费队列的，哪些消费策略到底是如何落地的。
- 最后我们来关注下在推模式的消费者中，MessageListenerConcurrently 和MessageListenerOrderly这两种消息监听器的处理逻辑到底有什么不同，为什么后者能保持消息顺序。

我们接下来就通过这几个问题来把RocketMQ的消费者部分源码串起来。

2、源码重点：

1、启动

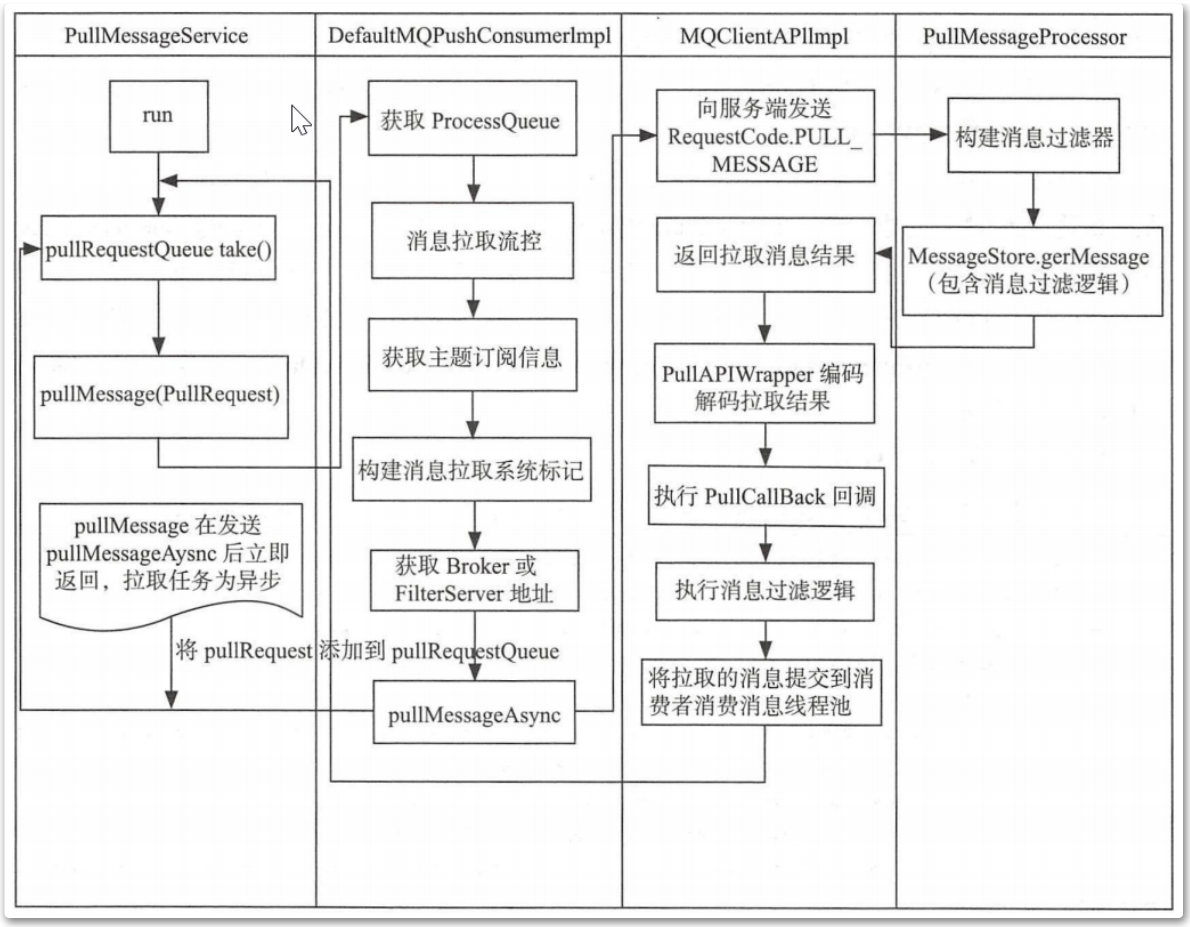
DefaultMQPushConsumer.start作为入口。最终消费者的启动过程，跟生产者一样，也交由了mQClientFactory。

通过mQClientFactory，消费者实例也启动了一大堆的服务。这些服务可以结合具体场景再进行深入。例如pullMessageService主要处理拉取消息服务，rebalanceService主要处理客户端的负载均衡。

2、消息拉取：

拉模式核心服务类： PullMessageService

PullRequest里有messageQueue和processQueue，其中messageQueue负责拉取消息，拉取到后，将消息存入processQueue，进行处理。 存入后就可以清空messageQueue，继续拉取了。



3 客户端负载均衡策略

在消费者示例的start方法中，启动RebalanceService，这个是客户端进行负载均衡策略的启动服务。他只负责根据负载均衡策略获取当前客户端分配到的MessageQueue示例。

五种负载策略，可以由Consumer的allocateMessageQueueStrategy属性来选择。这个属性可以在DefaultMQPushConsumer的构造方法当中指定。默认是AllocateMessageQueueAveragely策略

最常用的是AllocateMessageQueueAveragely平均分配和AllocateMessageQueueAveragelyByCircle平均轮询分配。

平均分配是把MessageQueue按组内的消费者个数平均分配。

而平均轮询分配就是把MessageQueue按组内的消费者一个一个轮询分配。

> 例如，六个队列q1,q2,q3,q4,q5,q6，分配给三个消费者c1,c2,c3
>
> 平均分配的结果就是：c1:{q1,q2},c2:{q3,q4},c3{q5,q6}

> 平均轮询分配的结果就是： c1:{q1,q4},c2:{q2,q5},c3:{q3,q6}

4 并发消费与顺序消费的过程

消费的过程依然是在DefaultMQPushConsumerImpl的consumeMessageService中。他有两个子类ConsumeMessageConcurrentlyService和ConsumeMessageOrderlyService。其中最主要的差别是ConsumeMessageOrderlyService会在消费前把队列锁起来，优先保证拉取同一个队列里的消息。

消费过程的入口在DefaultMQPushConsumerImpl的pullMessage中定义的PullCallback中。

3、 消费者部分小结：

RocketMQ消息消费方式分别为集群模式、广播模式。

消息队列负载由RebalanceService线程默认每隔20s进行一次消息队列负载，根据当前消费者组内消费者个数与主题队列数量按照某一种负载算法进行队列分配，分配原则为同一个消费者可以分配多个消息消费队列，同一个消息消费队列同一个时间只会分配给一个消费者。

消息拉取由PullMessageService线程根据RebalanceService线程创建的拉取任务进行拉取，默认每次拉取一批消息(可以由业务指定，默认是1)，提交给消费者消费线程后继续下一次消息拉取。如果消息消费过慢产生消息堆积会触发消息消费拉取流控。

并发消息消费指消费线程池中的线程可以并发对同一个消息队列的消息进行消费，消费成功后，取出消息队列中最小的消息偏移量作为消息消费进度偏移量存储在于消息消费进度存储文件中，集群模式消息消费进度存储在Broker（消息服务器），广播模式消息消费进度存储在消费者端。

RocketMQ不支持任意精度的定时调度消息，只支持自定义的消息延迟级别，例如1s、2s、5s等，可通过在broker配置文件中设置messageDelayLevel。

顺序消息一般使用集群模式，是指对消息消费者内的线程池中的线程对消息消费队列只能串行消费。与并发消息消费最本质的区别是消息消费时必须成功锁定消息消费队列，在Broker端会存储消息消费队列的锁占用情况。

四、重点业务机制

4.1 文件存储

1、关注重点

我们接着上面的流程，Producer把消息发到了Broker，接下来就关注下Broker接收到消息后是如何把消息进行存储的。最终存储的文件有哪些?

- commitLog：消息存储目录
- config：运行期间一些配置信息
- consumerqueue：消息消费队列存储目录
- index：消息索引文件存储目录
- abort：如果存在改文件寿命Broker非正常关闭
- checkpoint：文件检查点，存储CommitLog文件最后一次刷盘时间戳、consumerquueue最后一次刷盘时间，index索引文件最后一次刷盘时间戳。

还记得我们之前看到的Broker的核心组件吗？其中messageStore就是负责消息存储的核心组件。

2、源码重点：

消息存储的入口在：DefaultMessageStore.putMessage

1-commitLog写入

CommitLog的doAppend方法就是Broker写入消息的实际入口。这个方法最终会把消息追加到MappedFile映射的一块内存里，并没有直接写入磁盘。写入消息的过程是串行的，一次只会允许一个线程写入。

2-分发ConsumeQueue和IndexFile

当CommitLog写入一条消息后，在DefaultMessageStore的start方法中，会启动一个后台线程reputMessageService每隔1毫秒就会去拉取CommitLog中最新更新的一批消息，然后分别转发到ComsumeQueue和IndexFile里去，这就是他底层的实现逻辑。

并且，如果服务异常宕机，会造成CommitLog和ConsumeQueue、IndexFile文件不一致，有消息写入CommitLog后，没有分发到索引文件，这样消息就丢失了。DefaultMappedStore的load方法提供了恢复索引文件的方法，入口在load方法。

3、文件同步刷盘与异步刷盘

入口：CommitLog.submitFlushRequest

这里涉及到了对于同步刷盘与异步刷盘的不同处理机制。这里有很多极致提高性能的设计，对于我们理解和设计高并发应用场景有非常大的借鉴意义。

- 同步刷盘也是使用异步机制实现的。刷盘是一个很重的操作，所以，RocketMQ即便是同步刷盘，也要对刷盘次数精打细算。对于单条消息，那么直接将commitlog刷盘即可。但是对于批量消息，RockeMQ会先收集这一批次消息的刷盘请求，再进行一次统一的刷盘操作。并且一批消息有可能会跨两个commitlog文件，所以在刷盘时，要严格计算commitlog文件的刷盘次数。
- 异步刷盘是通过RocketMQ自己实现的一个CountDownLatch2提供了线程阻塞，使用CAS来驱动CountDownLatch2的countDown操作。每来一个消息就启动一次CAS，成功后，调用一次countDown。 **这个CountDownLatch2在java.util.concurrent.CountDownLatch的基础上，增加实现了reset功能**，实现了对象的重用。

其中主要涉及到是否开启了对外内存。TransientStorePoolEnable。如果开启了堆外内存，会在启动时申请一个跟CommitLog文件大小一致的堆外内存，这部分内存就可以确保不会被交换到虚拟内存中。

4、CommigLog主从复制

入口：CommitLog.submitReplicaRequest

主从同步时，也体现到了RocketMQ对于性能的极致追求。最为明显的，RocketMQ整体是基于Netty实现的网络请求，而在主从复制这一块，却放弃了Netty框架，转而使用更轻量级的Java的NIO来构建。

在主要的HAService中，会在启动过程中启动三个守护进程。

```
//HAService#start
public void start() throws Exception {
    this.acceptSocketService.beginAccept();
    this.acceptSocketService.start();
    this.groupTransferService.start();
    this.haClient.start();
}
```

这其中与Master相关的是acceptSocketService和groupTransferService。其中acceptSocketService主要负责维护Master与Slave之间的TCP连接。groupTransferService主要与主从同步复制有关。而slave相关的则是haClient。

至于其中关于主从的同步复制与异步复制的实现流程，还是比较复杂的，有兴趣的同学可以深入去研究一下。

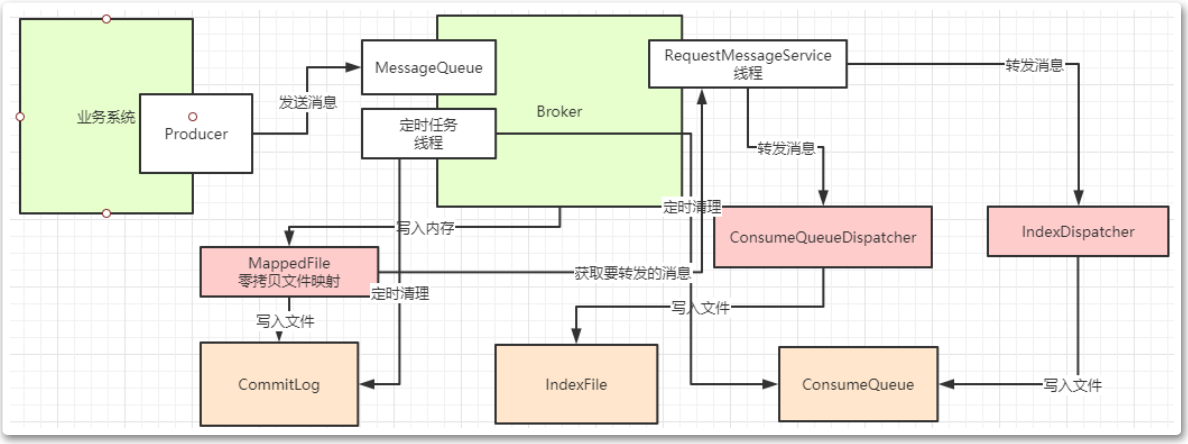
> 推荐一篇可供参考的博客 https://blog.csdn.net/qq_25145759/article/details/115865245

4、过期文件删除

入口： DefaultMessageStore.addScheduleTask -> DefaultMessageStore.this.cleanFilesPeriodically()

默认情况下， Broker会启动后台线程，每60秒，检查CommitLog、ConsumeQueue文件。然后对超过72小时的数据进行删除。也就是说，默认情况下， RocketMQ只会保存3天内的数据。这个时间可以通过fileReservedTime来配置。注意他删除时，并不会检查消息是否被消费了。

整个文件存储的核心入口入口在DefaultMessageStore的start方法中。



4.2、延迟消息

1、关注重点

延迟消息是RocketMQ非常有特色的一个功能，其他MQ产品中，往往需要开发者使用一些特殊方法来变相实现延迟消息功能。而RocketMQ直接在产品中实现了这个功能，开发者只需要设定一个属性就可以快速实现。

延迟消息的核心使用方法就是在Message中设定一个MessageDelayLevel参数，对应18个延迟级别。然后Broker中会创建一个默认的Schedule_Topic主题，这个主题下有18个队列，对应18个延迟级别。消息发过来之后，会先把消息存入Schedule_Topic主题中对应的队列。然后等延迟时间到了，再转发到目标队列，推送给消费者进行消费。

2、源码重点

延迟消息的处理入口在scheduleMessageService这个组件中。 他会在broker启动时也一起加载。

1、消息写入：

代码见CommitLog.putMessage方法。

在CommitLog写入消息时，会判断消息的延迟级别，然后修改Message的Topic和Queue，达到转储Message的目的。

2、消息转储到目标Topic

这个转储的核心服务是scheduleMessageService，他也是Broker启动过程中的一个功能组件。随DefaultMessageStore组件一起构建。这个服务只在master节点上启动，而在slave节点上会主动关闭这个服务。

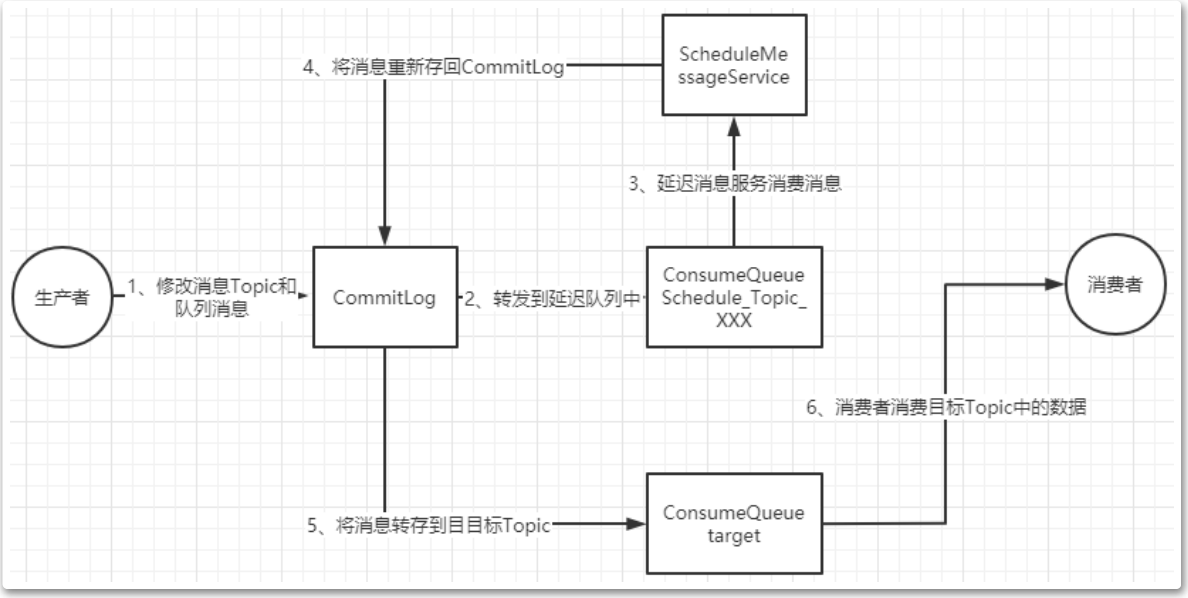
由于RocketMQ的主从节点支持切换，所以就需要考虑这个服务的幂等性。在节点切换为slave时就要关闭服务，切换为master时就要启动服务。并且，即便节点多次切换为master，服务也只启动一次。所以在ScheduleMessageService的start方法中，就通过一个CAS操作来保证服务的启动状态。

```
1  if (started.compareAndSet(false, true)) {
```

这个CAS操作还保证了在后面，同一时间只有一个DeliverDelayedMessageTimerTask执行。这种方式，给整个延迟消息服务提供了一个基础保证。

ScheduleMessageService会每隔1秒钟执行一个executeOnTimeup任务，将消息从延迟队列中写入正常Topic中。 代码见ScheduleMessageService中的DeliverDelayedMessageTimerTask.executeOnTimeup方法。

整个延迟消息的实现方式是这样的：



4.3、长轮询机制

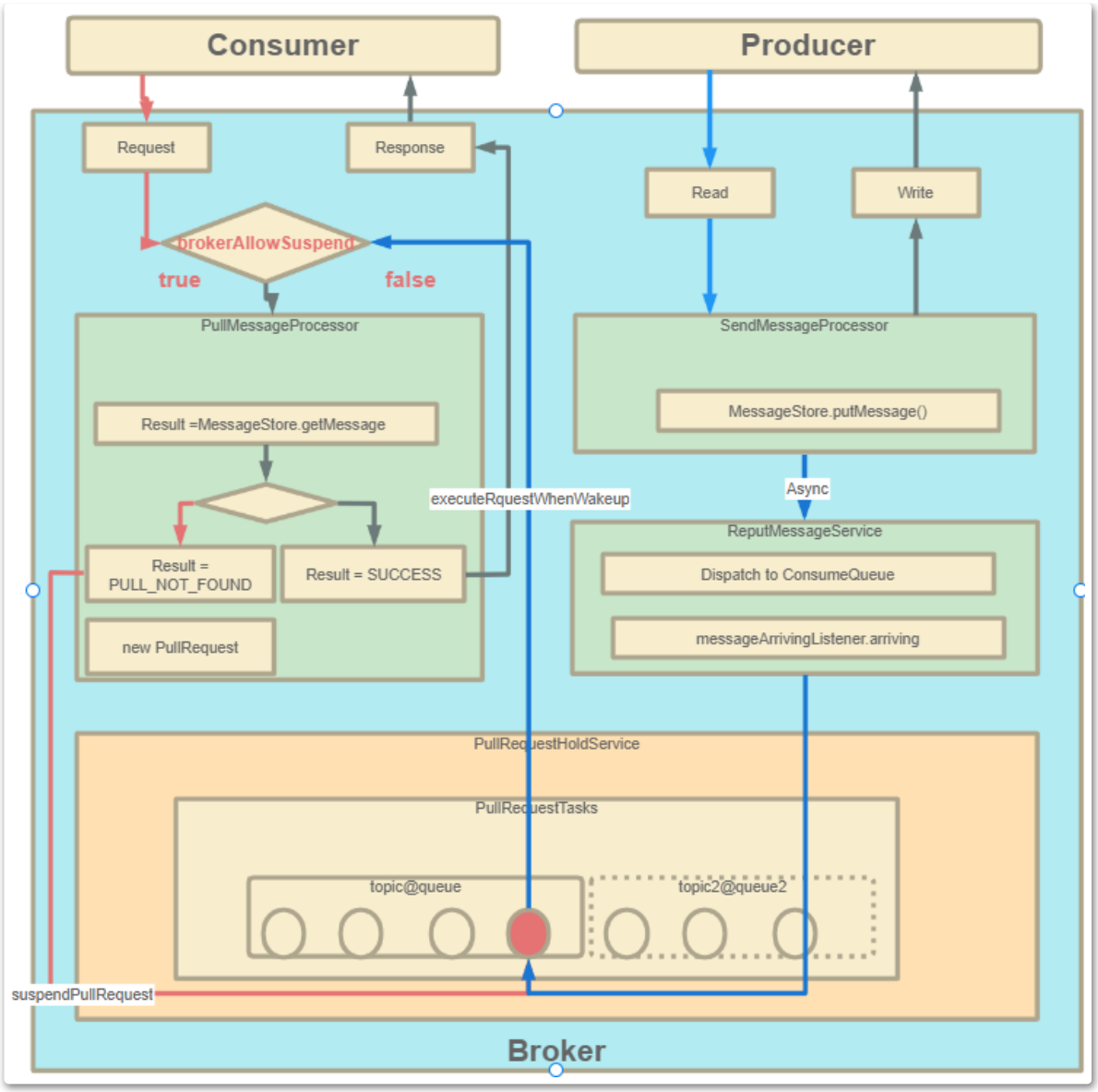
1、功能回顾

RocketMQ对消息消费者提供了Push推模式和Pull拉模式两种消费模式。但是这两种消费模式的本质其实都是Pull拉模式，Push模式可以认为是一种定时的Pull机制。但是这时有一个问题，当使用Push模式时，如果RocketMQ中没有对应的数据，那难道一直进行空轮询吗？如果是这样的话，那显然会极大的浪费网络带宽以及服务器的性能，并且，当有新的消息进来时，RocketMQ也没有办法尽快通知客户端，而只能等客户端下一次来拉取消息了。针对这个问题，RocketMQ实现了一种长轮询机制 long polling。

长轮询机制简单来说，就是当Broker接收到Consumer的Pull请求时，判断如果没有对应的消息，不用直接给Consumer响应(给响应也是个空的，没意义)，而是就将这个Pull请求给缓存起来。当Producer发送消息过来时，增加一个步骤去检查是否有对应的已缓存的Pull请求，如果有，就及时将请求从缓存中拉取出来，并将消息通知给Consumer。

2、源码重点

整个流程以及源码重点如下图所示：

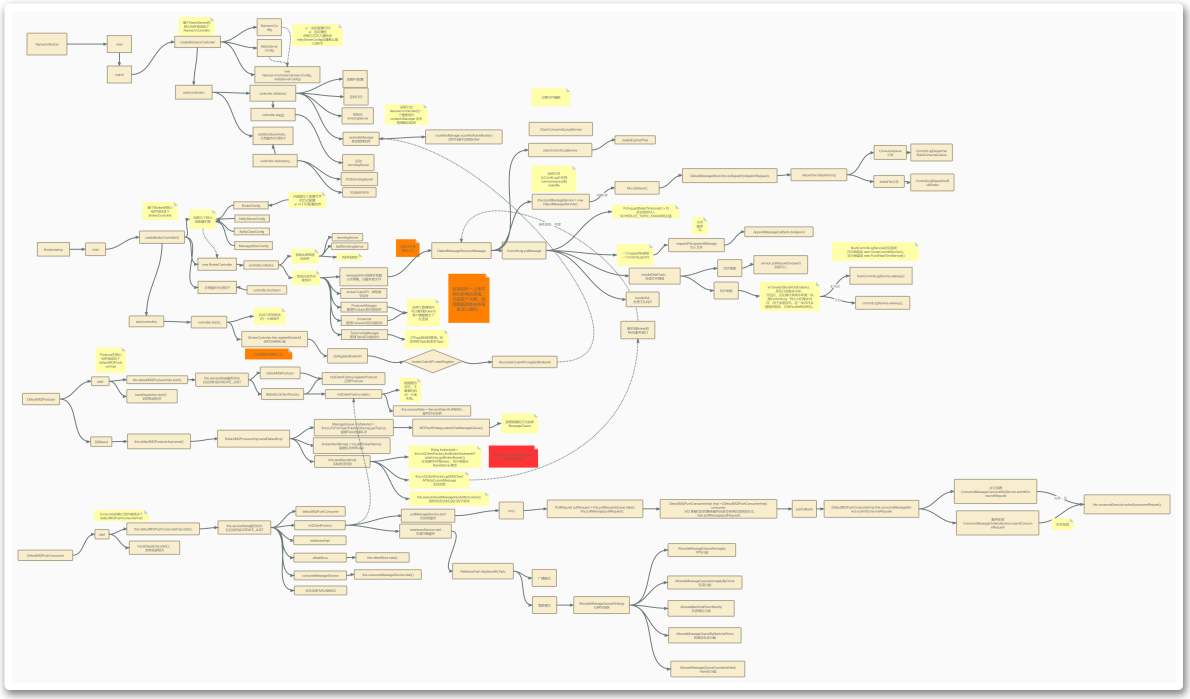


源码解读小结

关于RocketMQ的源码部分，我们就带大家解读到这里。到目前为止，几个核心的流程我们已经解读完成了，我们按照由大到小，由粗到细的方式对几条主线进行了解读。通过解读源码，我们可以对之前提到的各种高级特性有更深入的理解。对有些有争议的问题，带着问题来源码中找答案是最好的。例如我们经常有人讨论NameServer全部挂了之后，生产者和消费者是否能够用他本地的缓存继续工作一段时间？ 这样的一些问题，看过源码之后是不是有更清晰的了解？

至于其他的代码，大家也可以按照自己的关注点，以业务线的方式来逐步解读。

最后将我们今天解读的部分源码整理成了一个大图，可供参考



> 有道云笔记链接：
>
> 文档：五期VIP04-RocketMQ源码解读以及设计思...
> 链接：http://note.youdao.com/noteshare?id=b567733dfd842a5583a47829cab837c3&sub=A54BCD334B8C415FB9171A665BF14661