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/

4.7.1 release

- Released June 29, 2020
- Release Notes
- Source: rocketmq-all-4.7.1-source-release.zip [PGP] [SHA512]
- Binary: rocketmq-all-4.7.1-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4.7.1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：

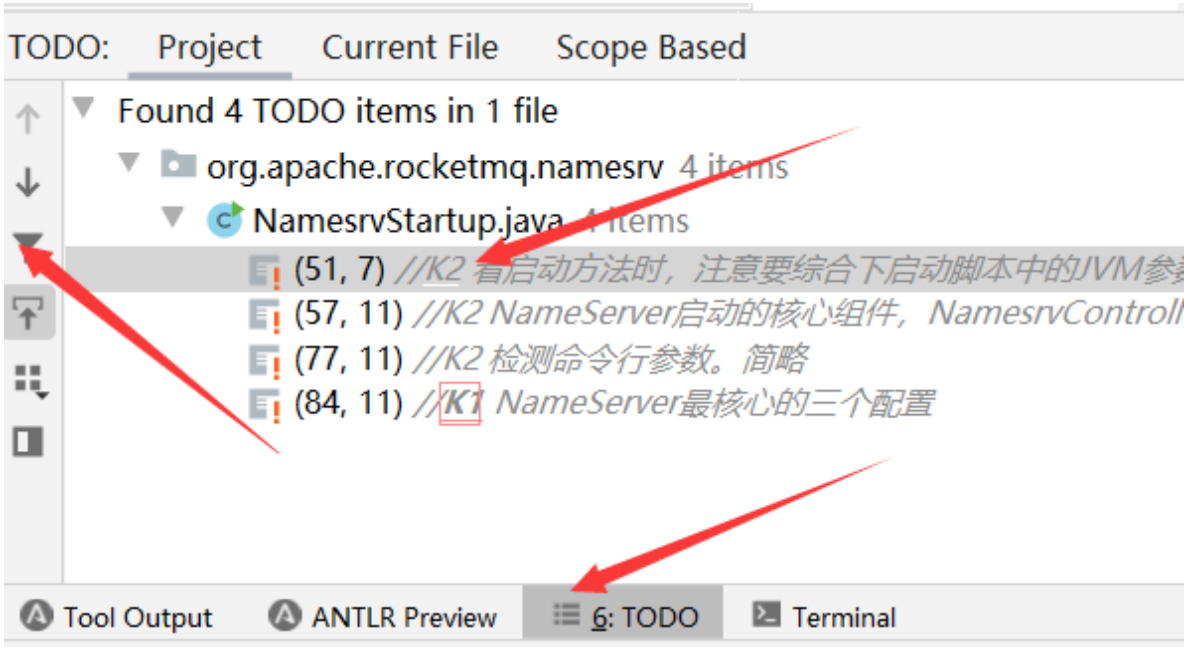
- broker: broker 模块（broke 启动进程）
- client ： 消息客户端，包含消息生产者、消息消费者相关类
- example: RocketMQ 例代码
- namesrv：NameServer实现相关类（NameServer启动进程）
- store：消息存储实现相关类

> 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

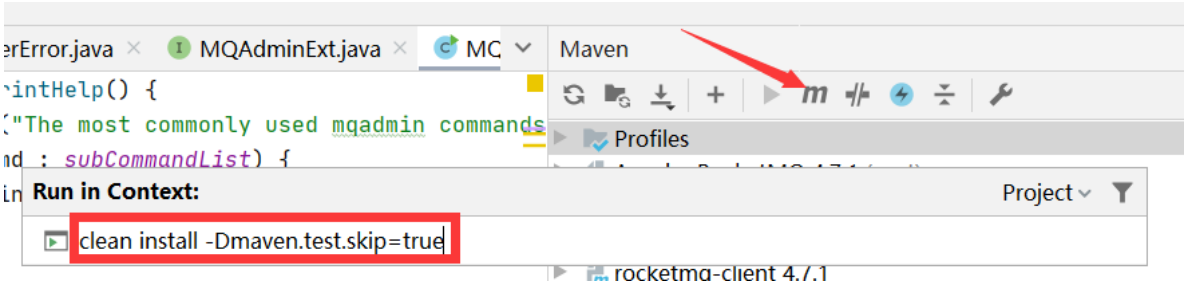
RocketMQ的源码中有个非常让人头疼的事情，就是他的代码注释几乎没有。为了帮助大家解读源码，我给大家准备了一个添加了自己注释的源码版本。 在配套资料当中。大家可以把这个版本导入IDEA来进行解读。

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。



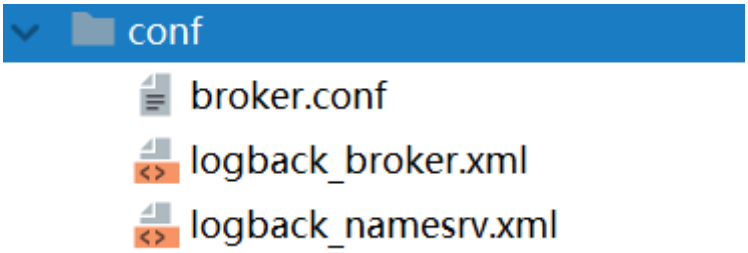
3、源码调试：

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：

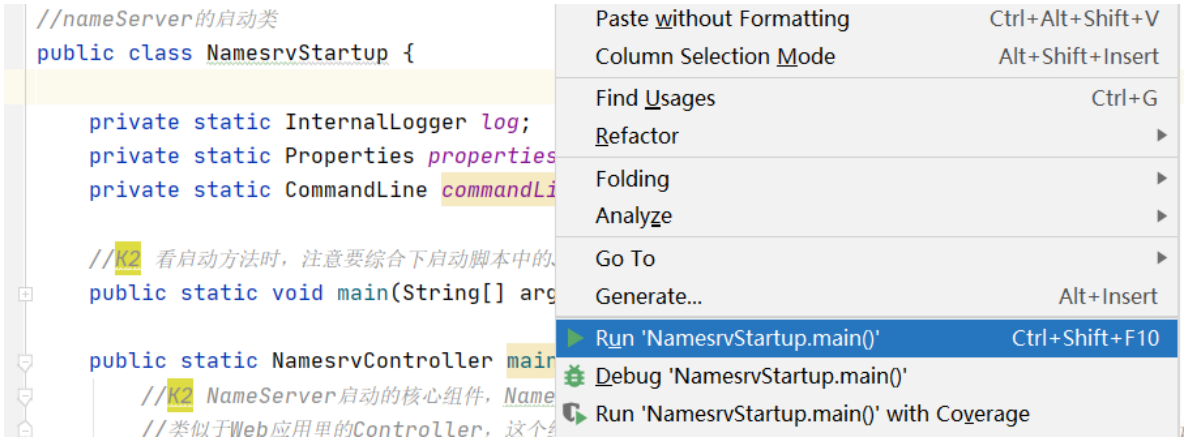
调试时，先在项目目录下创建一个conf目录，并从 `distribution` 拷贝 `broker.conf` 和 `logback_broker.xml` 和 `logback_namesrv.xml`



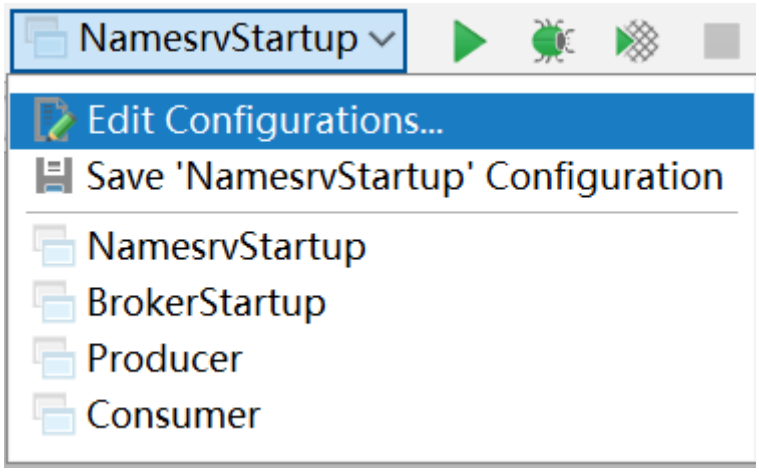
> 注解版源码中已经复制好了。

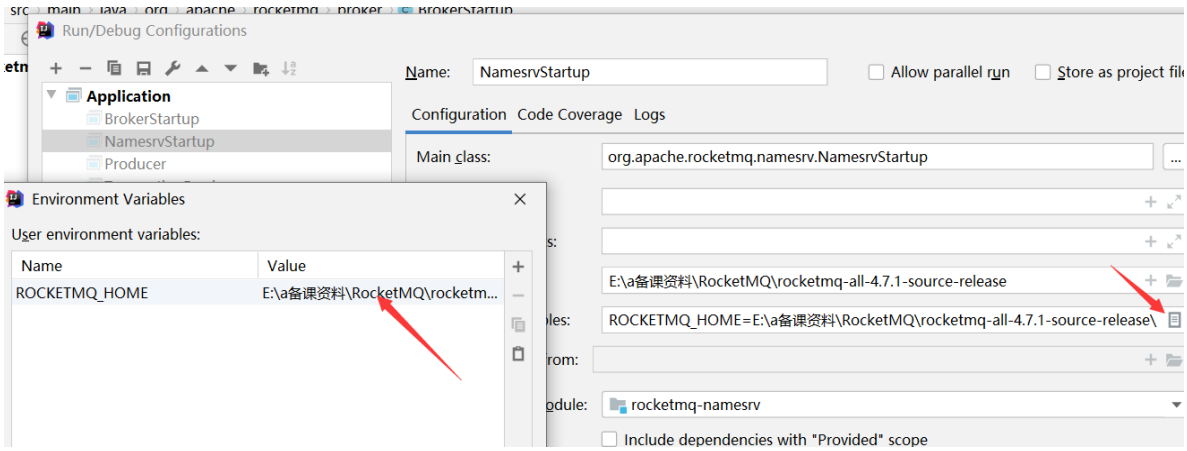
3.1 启动nameServer

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

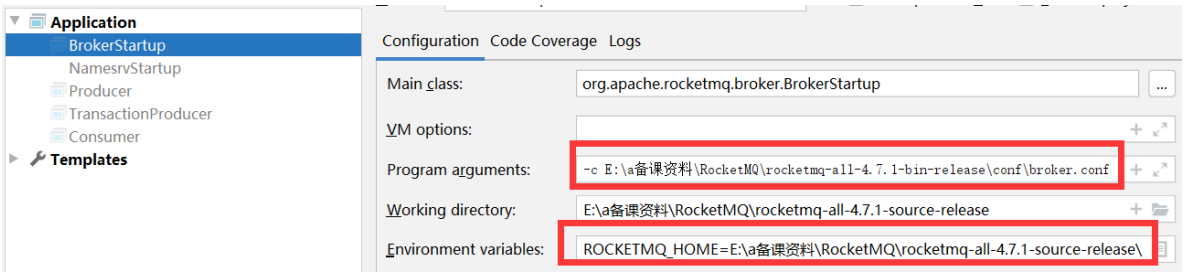
启动Broker之前，我们需要先修改之前复制的broker.conf文件

```
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

下面我们可以一边调试一边讲解源码了。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。

看源码时，需要用层层深入的方法。每一次阅读源码时，先了解程序执行的流程性代码，略过服务实现的细节性代码，形成大概的概念框架。然后再回头按同样的方法，逐步深入到之前略过的代码。这样才能从源码中看出一点门道来。所以我们这次看源码，对于Netty服务调用、文件读写等等这些细节问题不会太过关注，重要的是对RocketMQ形成一个整体的印象，以后如果想要再了解这些问题，可以知道如何下手。

二、NameServer启动

1、功能回顾

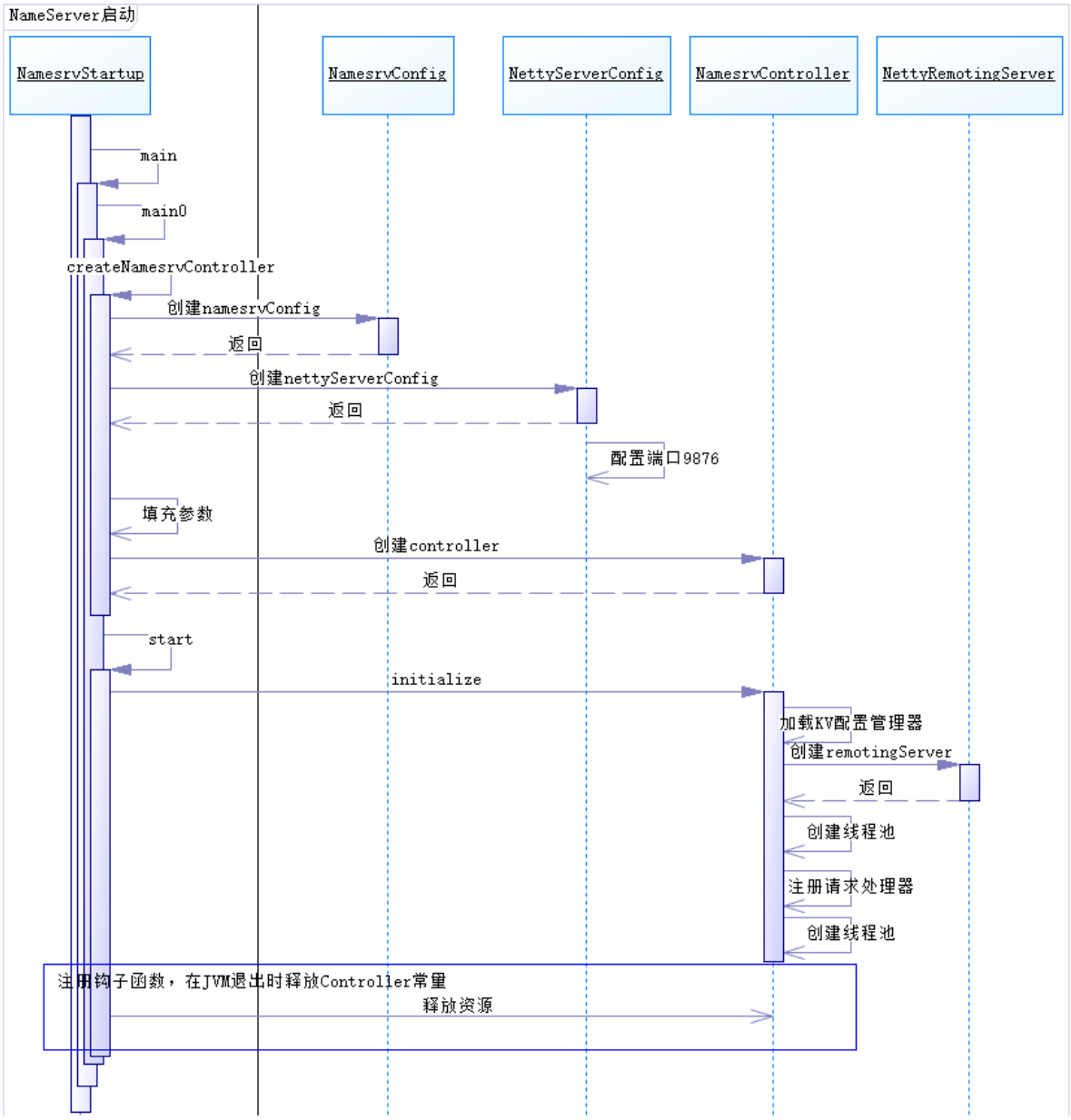
从之前的介绍中，我们已经了解到，在RocketMQ中，实际进行消息存储、推送等核心功能的是Broker。那NameServer具体做什么用呢？NameServer的核心作用其实就只有两个

- 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。

2、启动流程

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。

整体的流程：



3、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。

在创建NamesrvController对象时，有两个关键的配置

- NamesrvConfig 这个是NameServer自己运行需要的配置信息。
- NettyServerConfig 包含Netty服务端的配置参数，默认占用了9876端口。可以在配置文件中覆盖。

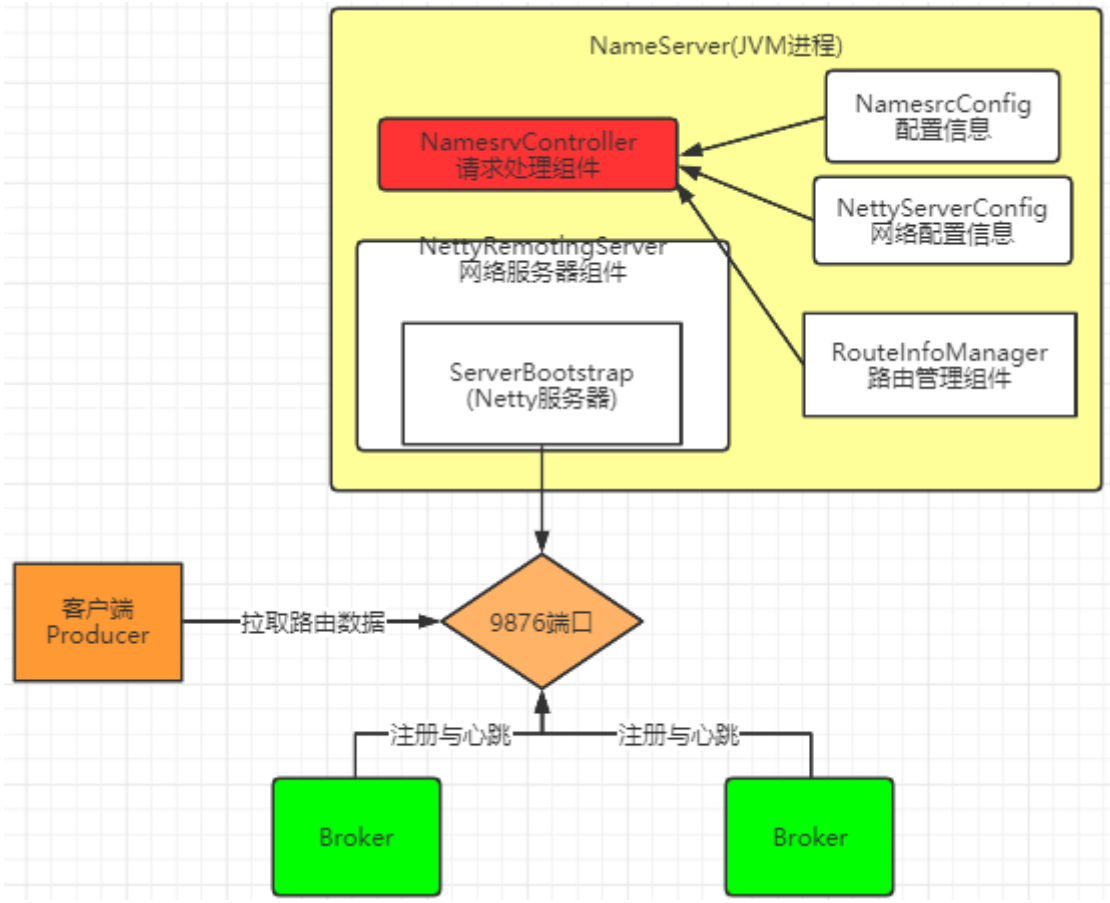
然后在启动服务时，启动几个重要组件：

- RemotingServer 这个就是用来响应请求的。
- 还有一个定时任务会定时扫描不活动的Broker。这个Broker管理是通过routeInfoManager这个功能组件。

在关闭服务时，关闭了四个东西

- RemotingServer
- remotingExecutor Netty服务线程池;
- scheduledExecutorService 定时任务;
- fileWatchService 这个是用来跟踪TLS配置的。这是跟权限相关的，我们暂不关注。

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：



三、Broker启动

1、功能回顾

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。

而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。

所以我们这一段的重点，是通过Broker的启动过程，观察总结出Broker的内部结构。

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

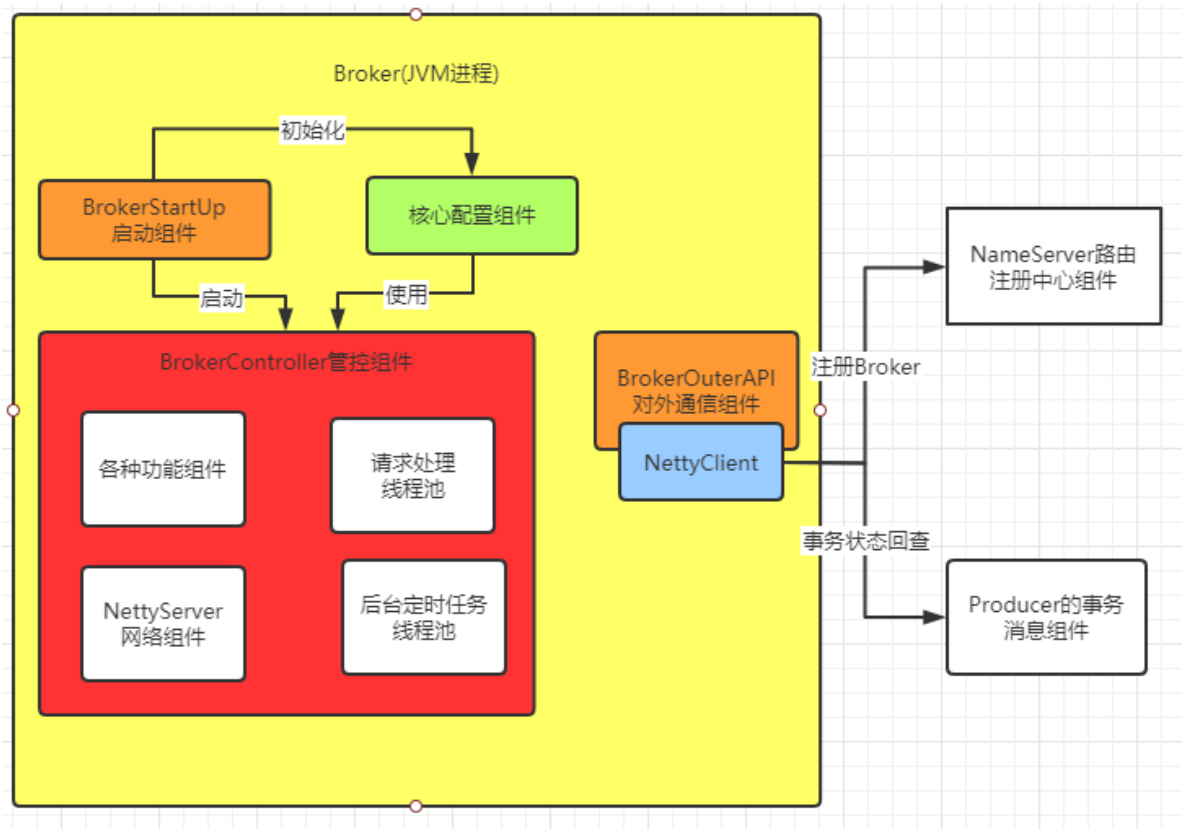
MessageStoreConfig

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：



四、Broker注册

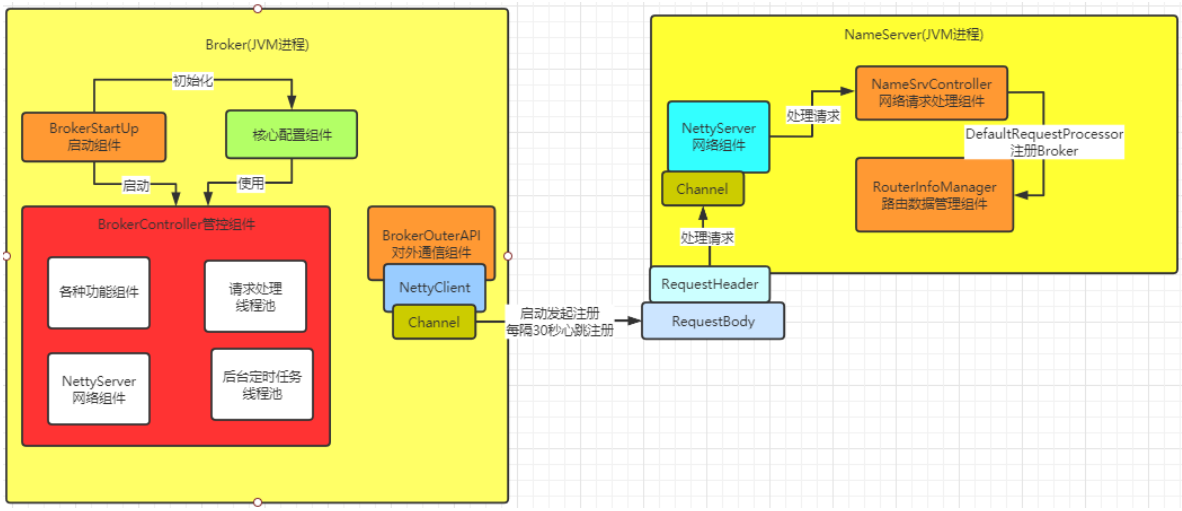
1、功能回顾

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。

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

五、Producer

1、功能回顾

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。这个只需要构建一个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。
- 另一种是事务消息发送者： TransactionMQProducer。这个需要构建一个Netty客户端，往Broker发送消息。同时也要构建Netty服务端，供Broker回查本地事务状态。

由于整个Producer的流程，其实还是挺复杂的，我们这里只关注DefaultMQProducer的整个过程。TransactionMQProducer就不在课上带大家看了。

2、源码重点

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：

首先 我们关注下Broker的核心启动流程：

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。

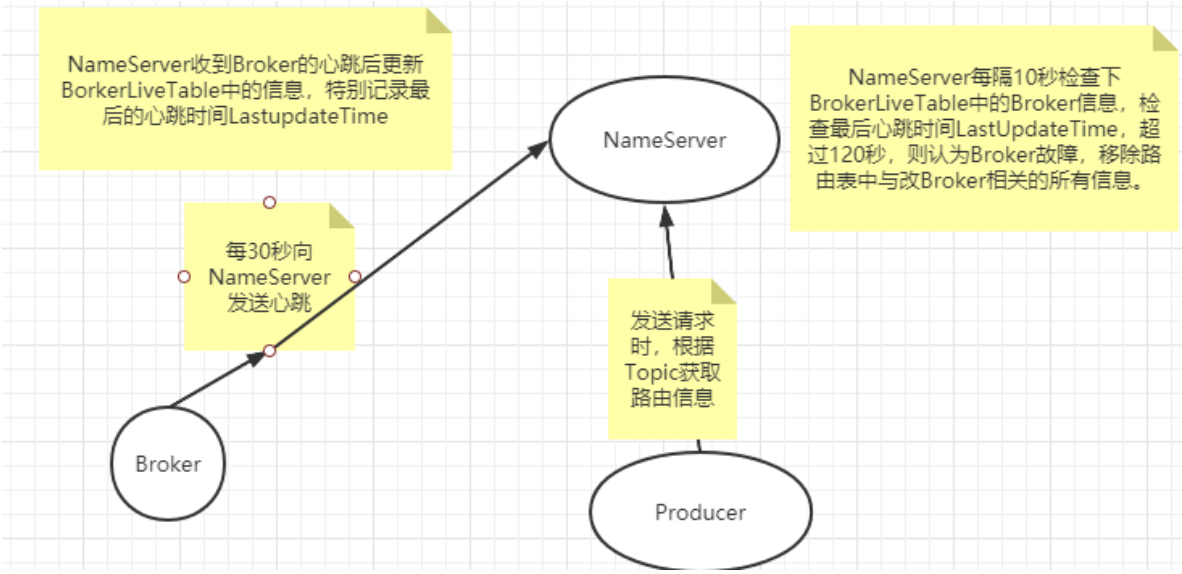
其次：关于Borker路由信息的管理： Producer需要拉取Broker列表，然后跟Broker建立连接等等很多核心的流程，其实都是在发送消息时建立的。因为在启动时，还不知道要拉取哪个Topic的Broker列表呢。所以对于这个问题，我们关注的重点，不应该是start方法，而是send方法。

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：



然后 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?

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

```java
//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 在发送Netty请求时，实际上是指定的MessageQueue，而不是Topic。Topic只是用来找MessageQueue。

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

六、消息存储

1、功能回顾

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.putMessage -> CommitLog.handleDiskFlush

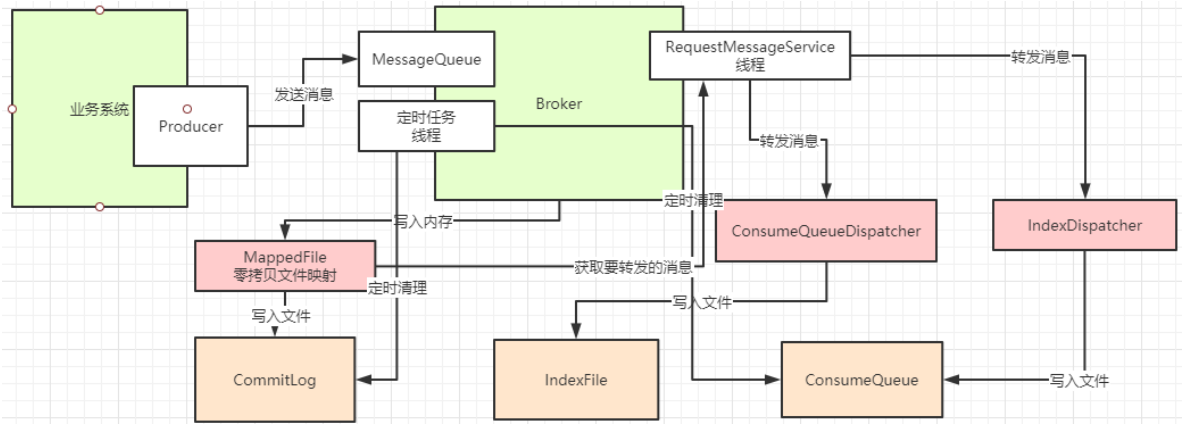
其中主要涉及到是否开启了对外内存。TransientStorePoolEnable。如果开启了堆外内存，会在启动时申请一个跟CommitLog文件大小一致的堆外内存，这部分内存就可以确保不会被交换到虚拟内存中。

4、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。



5文件存储部分的总结:

RocketMQ的存储文件包括消息文件（Commitlog）、消息消费队列文件（ConsumerQueue）、Hash索引文件（IndexFile）、监测点文件（checkPoint）、abort（关闭异常文件）。单个消息存储文件、消息消费队列文件、Hash索引文件长度固定以便使用内存映射机制进行文件的读写操作。RocketMQ组织文件以文件的起始偏移量来命令文件，这样根据偏移量能快速定位到真实的物理文件。RocketMQ基于内存映射文件机制提供了同步刷盘和异步刷盘两种机制，异步刷盘是指在消息存储时先追加到内存映射文件，然后启动专门的刷盘线程定时将内存中的文件数据刷写到磁盘。

CommitLog，消息存储文件，RocketMQ为了保证消息发送的高吞吐量，采用单一文件存储所有主题消息，保证消息存储是完全的顺序写，但这样给文件读取带来了不便，为此RocketMQ为了方便消息消费构建了消息消费队列文件，基于主题与队列进行组织，同时RocketMQ为消息实现了Hash索引，可以为消息设置索引键，根据所以能够快速从CommitLog文件中检索消息。

当消息达到CommitLog后，会通过ReputMessageService线程接近实时地将消息转发给消息消费队列文件与索引文件。为了安全起见，RocketMQ引入abort文件，记录Broker的停机是否是正常关闭还是异常关闭，在重启Broker时为了保证CommitLog文件，消息消费队列文件与Hash索引文件的正确性，分别采用不同策略来恢复文件。

RocketMQ不会永久存储消息文件、消息消费队列文件，而是启动文件过期机制并在磁盘空间不足或者默认凌晨4点删除过期文件，文件保存72小时并且在删除文件时并不会判断该消息文件上的消息是否被消费。

七、消费者

1、功能回顾

结合我们之前的示例，回顾下消费者这一块的几个重点：

- 消费者也是有两种，推模式消费者和拉模式消费者。消费者的使用过程也跟生产者差不多，都是先start()然后再开始消费。

- 消费者以消费者组的模式开展。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。
- 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方法

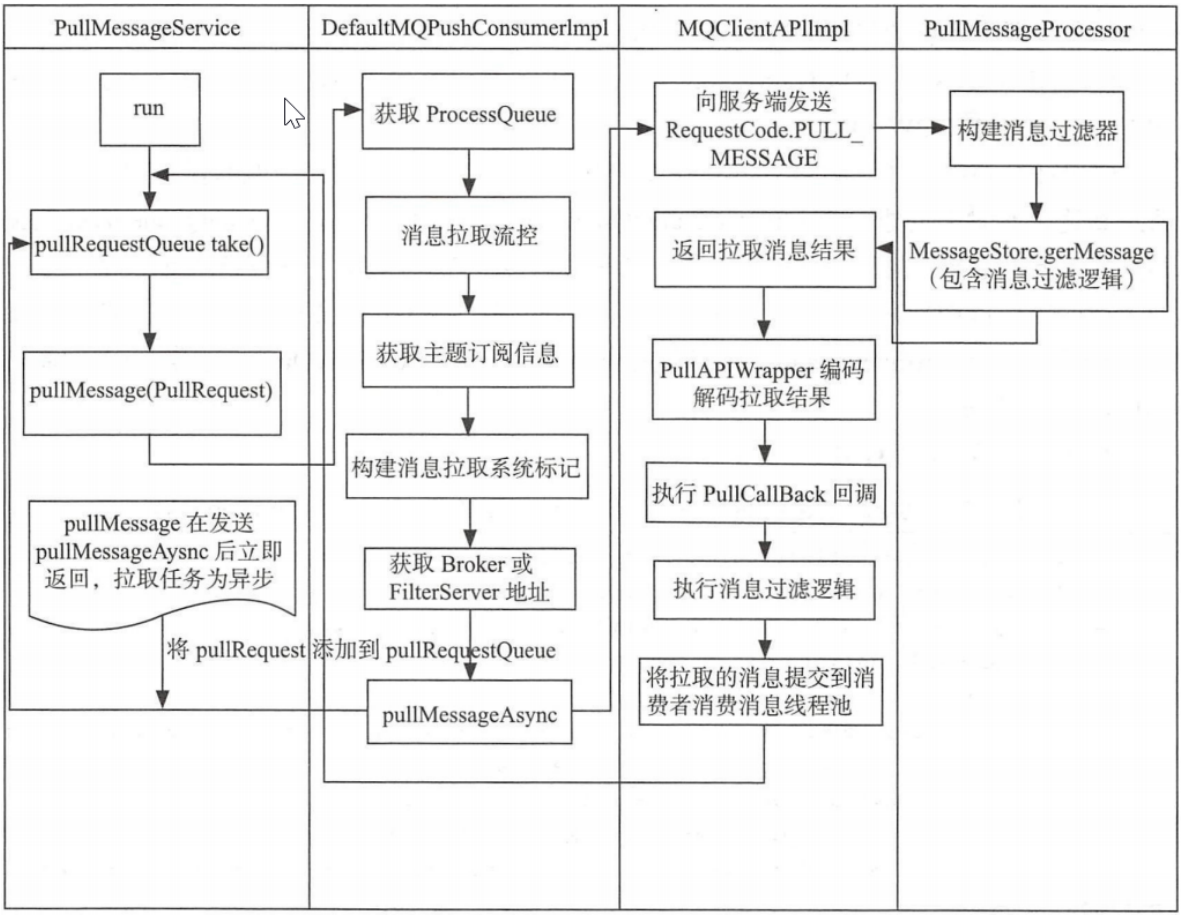
启动过程不用太过关注，有个概念就行，然后客户端启动的核心是mQClientFactory 主要是启动了一大堆的服务。

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： 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。

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。

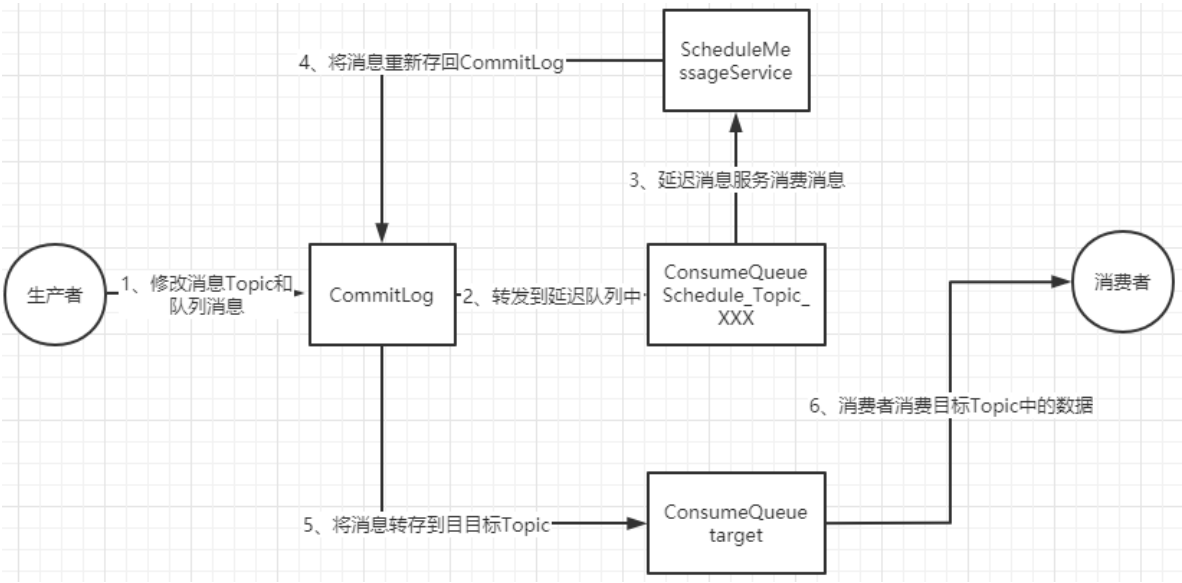
八、延迟消息

1、功能回顾

我们这里，就用一个典型的延迟消息的流程，来把上面看到的各个组件，结合一下。

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。

整个延迟消息的实现方式是这样的：



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、

然后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。

这个其中有个需要注意的点就是在ScheduleMessageService的start方法中。有一个很关键的CAS操作：

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

这个CAS操作保证了同一时间只会有一个DeliverDelayedMessageTimerTask执行。保证了消息安全的同时也限制了消息进行回传的效率。所以，这也是很多互联网公司在使用RocketMQ时，对源码进行定制的一个重点。

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32条消息，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。

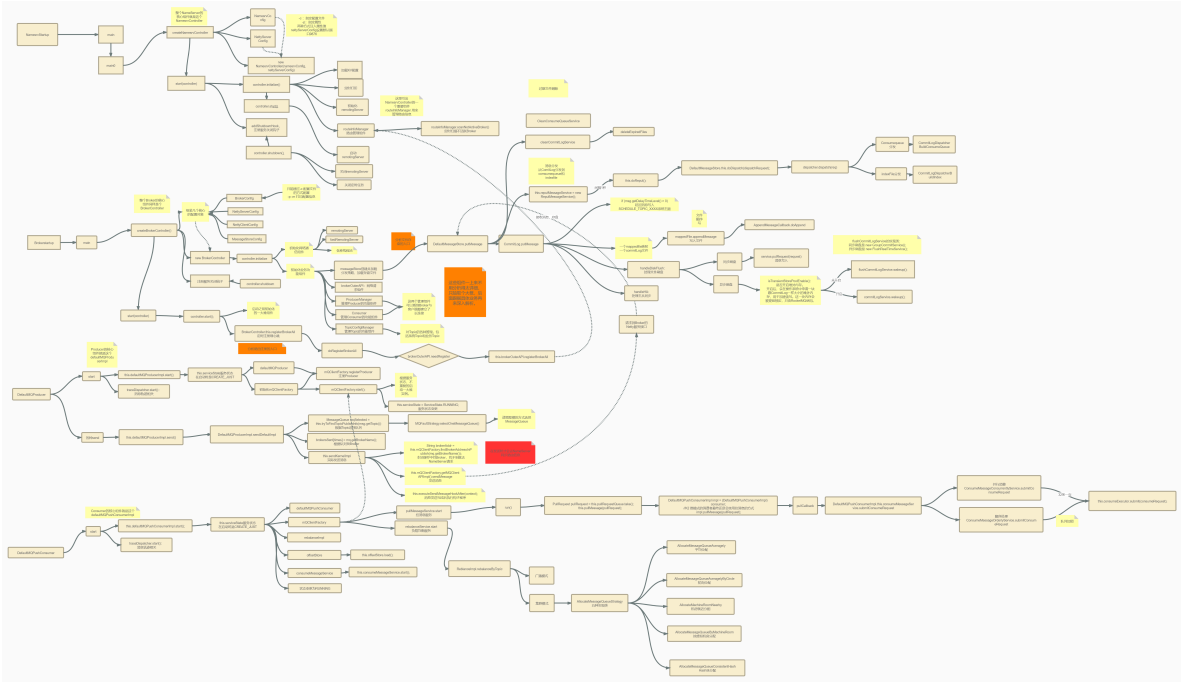
顺序消息一般使用集群模式，是指对消息消费者内的线程池中的线程对消息消费队列只能串行消费。与并发消息消费最本质的区别是消息消费时必须成功锁定消息消费队列，在Broker端会存储消息消费队列的锁占用情况。

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



> 有道云分享链接：
>
> 文档：VIP04-RocketMQ源码解读.md
> 链接：http://note.youdao.com/noteshare?id=c91c232cb5052cc9917c35ff2aa62e49&sub=06CA69844E4F43A6814FFE273ED0A9FE