

- 一、MQ简介
- 二、RocketMQ产品特点
  - 1、RocketMQ介绍
  - 2、RocketMQ特点
- 三、RocketMQ快速实战
  - 1、快速搭建RocketMQ服务
  - 2、快速实现消息收发
  - 3、搭建RocketMQ可视化管理服务
  - 4、升级分布式集群
  - 5、升级高可用集群
- 四、总结RocketMQ的运行架构
- 五、理解RocketMQ的消息模型
- 六、章节总结

图灵：楼兰  
RocketMQ快速实战以及核心概念详解  
你的神秘技术宝藏

笔记配合视频课程一起学习

## 一、MQ简介

MQ：MessageQueue，消息队列。是在互联网中使用非常广泛的一系列服务中间件。这个词可以分两个部分来看，一是Message：消息。消息是在不同进程之间传递的数据。这些进程可以部署在同一台机器上，也可以分布在不同机器上。二是Queue：队列。队列原意是指一种具有FIFO(先进先出)特性的数据结构，是用来缓存数据的。对于消息中间件产品来说，能不能保证FIFO特性，尚值得考量。但是，所有消息队列都是需要具备存储消息，让消息排队的能力。

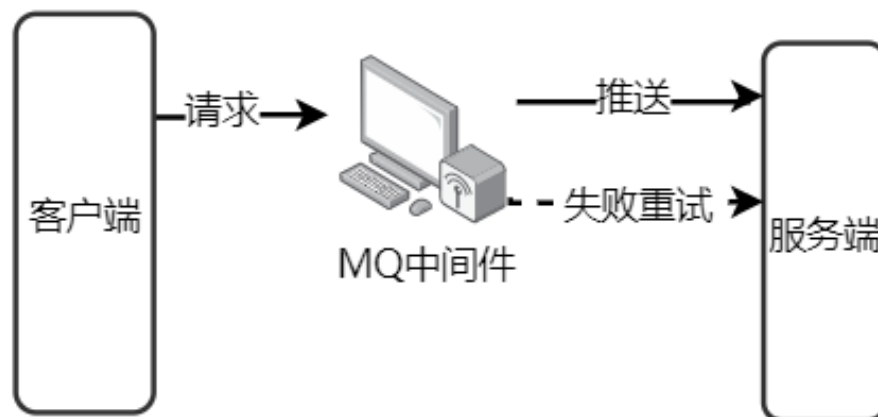
广义上来说，只要能够实现消息跨进程传输以及队列数据缓存，就可以称之为消息队列。例如我们常用的QQ、微信、阿里旺旺等就都具备了这样的功能。只不过他们对接的使用对象是人，而我们这里讨论的MQ产品需要对接的使用对象是应用程序。

MQ的作用主要有以下三个方面：

- 异步

例子：快递员发快递，直接到客户家效率会很低。引入菜鸟驿站后，快递员只需要把快递放到菜鸟驿站，就可以继续发其他快递去了。客户再按自己的时间安排去菜鸟驿站取快递。

作用：异步能提高系统的响应速度、吞吐量。

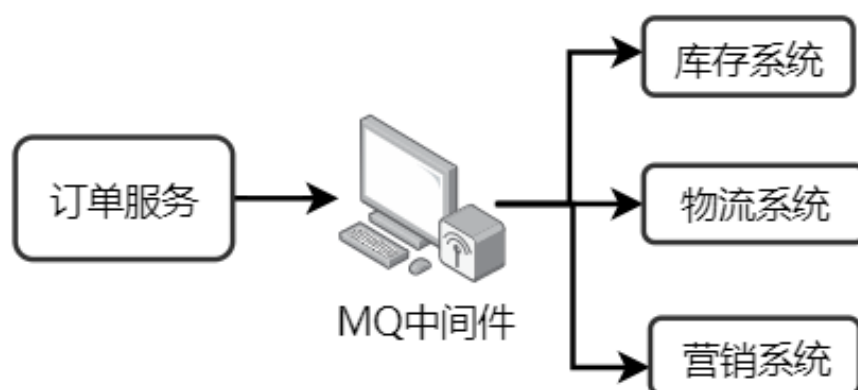


- 解耦

例子：《Thinking in JAVA》很经典，但是都是英文，我们看不懂，所以需要编辑社，将文章翻译成其他语言，这样就可以完成英语与其他语言的交流。

作用：

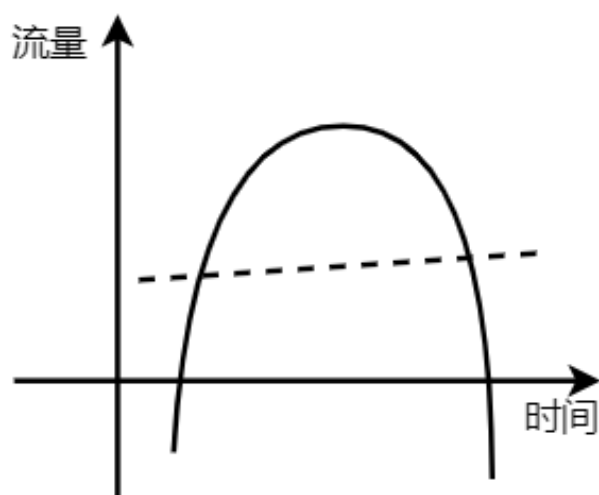
- 1、服务之间进行解耦，才可以减少服务之间的影响。提高系统整体的稳定性以及可扩展性。
- 2、另外，解耦后可以实现数据分发。生产者发送一个消息后，可以由一个或者多个消费者进行消费，并且消费者的增加或者减少对生产者没有影响。



- 削峰

例子：长江每年都会涨水，但是下游出水口的速度是基本稳定的，所以会涨水。引入三峡大坝后，可以把水储存起来，下游慢慢排水。

作用：以稳定的系统资源应对突发的流量冲击。



## 二、RocketMQ产品特点

### 1、RocketMQ介绍

RocketMQ是阿里巴巴开源的一个消息中间件，在阿里内部历经了双十一等很多高并发场景的考验，能够处理亿万级别的消息。2016年开源后捐赠给Apache，现在是Apache的一个顶级项目。

早期阿里使用ActiveMQ，但是，当消息开始逐渐增多后，ActiveMQ的IO性能很快达到了瓶颈。于是，阿里开始关注Kafka。但是Kafka是针对日志收集场景设计的，他的高级功能并不是很贴合阿里的业务场景。尤其当他的Topic过多时，由于Partition文件也会过多，这就会加大文件索引的耗时，会严重影响IO性能。于是阿里才决定自研中间件，最早叫做MetaQ，后来改名成为RocketMQ。最早他所希望解决的最大问题就是多Topic下的IO性能压力。但是产品在阿里内部的不断改进，RocketMQ开始体现出一些不一样的优势。

### 2、RocketMQ特点

当今互联网MQ产品众多，其中，影响力和使用范围最大的当数Apache Kafka、RabbitMQ、Apache RocketMQ以及Apache Plusar。这几大产品虽然都是典型的MQ产品，但是由于设计和实现上的一些差异，造成他们适合于不同的细分场景。

	优点	缺点	适合场景
Apache Kafka	吞吐量非常大，性能非常好，集群高可用。	会有丢数据的可能，功能比较单一	日志分析、大数据采集
RabbitMQ	消息可靠性高，功能全面。	erlang语言不好定制。吞吐量比较低。	企业内部小规模服务调用
Apache Pulsar	基于Bookeeper构建，消息可靠性非常高。	周边生态还有差距，目前使用的公司比较少。	企业内部大规模服务调用
Apache RocketMQ	高吞吐、高性能、高可用。功能全面。客户端协议丰富。使用java语言开发，方便定制。	服务加载比较慢。	几乎全场景，特别适合金融场景

其中RocketMQ，孵化自阿里巴巴。历经阿里多年双十一的严格考验，RocketMQ可以说是从全世界最严苛的高并发场景中摸爬滚打出来的过硬产品，也是少数几个在金融场景比较适用的MQ产品。从横向对比来看，RocketMQ与Kafka和RabbitMQ相比。RocketMQ的消息吞吐量虽然和Kafka相比还是稍有差距，但是却比RabbitMQ高很多。在阿里内部，RocketMQ集群每天处理的请求数超过5万亿次，支持的核心应用超过3000个。而RocketMQ最大的优势就是他天生就为金融互联网而生。他的消息可靠性相比Kafka也有了很大的提升，而消息吞吐量相比RabbitMQ也有很大的提升。另外，RocketMQ的高级功能也越来越全面，广播消费、延迟队列、死信队列等等高级功能一应俱全，甚至某些业务功能比如事务消息，已经呈现出领先潮流的趋势。

## 三、RocketMQ快速实战

### 1、快速搭建RocketMQ服务

RocketMQ的官网地址：<http://rocketmq.apache.org>。在下载页面可以获取RocketMQ的源码包以及运行包。下载页面地址：<https://rocketmq.apache.org/download>。

# RocketMQ

版本	文档	发布日期	Source 下载	Binary 下载	End of Support
5.1.0	<a href="#">Release Notes</a>	2022.9.22	<a href="#">download.zip</a>	<a href="#">download.zip</a>	
5.0.0	<a href="#">Release Notes</a>	2022.9.22	<a href="#">download.zip</a>	<a href="#">download.zip</a>	
4.9.5	<a href="#">Release Notes</a>	2023.4.1	<a href="#">download.zip</a>	<a href="#">download.zip</a>	2024/01
4.9.4	<a href="#">Release Notes</a>	2022.6.27	<a href="#">download.zip</a>	<a href="#">download.zip</a>	

当前最新的版本是5.x，这是一个着眼于云原生的新版本，给 RocketMQ 带来了非常多很亮眼的新特性。但是目前来看，企业中用得还比较少。因此，我们这里采用的还是更为稳定的4.9.5版本。

注：在2020年下半年，RocketMQ新推出了5.0的大版本，这对于RocketMQ来说，是一个里程碑式的大版本。在这个大版本中，RocketMQ对整体功能做了一次大的升级。增加了很多非常实用的新特性，也对已有功能重新做了升级。

比如在具体功能方面，在4.x版本中，对于定时消息，只能设定几个固定的延迟级别，而5.0版本中，已经可以指定具体的发送时间了。在客户端语言方面，4.x版本，RocketMQ原生只支持基于Netty框架的Java客户端。而在5.0版本中，增加了对Grpc协议的支持，这基本上就解除了对客户端语言的限制。在服务端架构方面，4.x版本只支持固定角色的普通集群和可以动态切换角色的Dledger集群，而在5.0版本中，增加了Dledger Controller混合集群模式，即可以混合使用Dledger的集群机制以及 Broker 本地的文件管理机制。

但是功能强大，同时也意味着问题会很多。所以目前来看，企业中直接用新版本的还比较少。小部分使用新版本的企业，也大都是使用内部的改造优化版本。

运行只需要下载Binary运行版本就可以了。当然，源码包也建议下载下来，后续会进行解读。运行包下载下来后，就可以直接解压，上传到服务器上。我们这里会上传到/app/rocketmq目录。解压后几个重要的目录如下：

/app/rocketmq/rocketmq-all-4.9.5-bin-release

✓ /

> afs

✓ app

✓ rocketmq

✓ rocketmq

benc1

bin

conf

lib

> boot

Name

..

benchmark

bin

conf

lib

NOTICE

README.md

LICENSE

压测脚本

执行脚本

配置文件

运行的jar包

接下来，RocketMQ建议的运行环境需要至少12G的内存，这是生产环境比较理想的资源配置。但是，学习阶段，如果你的服务器没有这么大的内存空间，那么就需要做一下调整。进入bin目录，对其中的runserver.sh和runbroker.sh两个脚本进行一下修改。

使用vi runserver.sh指令，编辑这个脚本，找到下面的一行配置，调整Java进程的内存大小。

```
JAVA_OPT="${JAVA_OPT} -server -Xms512m -Xmx512m -Xmn256m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m"
```

接下来，同样调整runbroker.sh中的内存大小。

```
JAVA_OPT="${JAVA_OPT} -server -Xms8g -Xmx8g"  
修改为：  
JAVA_OPT="${JAVA_OPT} -server -Xms1g -Xmx1g"
```

生产环境不建议调整。

调整完成后，就可以启动RocketMQ服务了。RocketMQ服务基于Java开发，所以需要提前安装JDK。JDK建议采用1.8版本即可。

Java环境安装略过。

RocketMQ的后端服务分为nameserver和broker两个服务，关于他们的作用，后面会给你分享。接下来我们先先将这两个服务启动起来。

### 第一步：启动nameserver服务。

```
cd /app/rocketmq/rocketmq-all-4.9.5-bin-release  
nohup bin/mqnamesrv &
```

指令执行后，会生成一个nohup.out的日志文件。在这个日志文件里如果看到下面这一条关键日志，就表示nameserver服务启动成功了。

```
Java HotSpot(TM) 64-Bit Server VM warning: Using the DefNew young collector  
with the CMS  
collector is deprecated and will likely be removed in a future release  
Java HotSpot(TM) 64-Bit Server VM warning: UseCMSCompactAtFullCollection is  
deprecated and  
will likely be removed in a future release.  
The Name Server boot success. serializeType=JSON
```

接下来，可以通过jps指令进行验证。使用jps指令后，可以看到有一个NamesrvStartup的进程运行，也表示nameserver服务启动完成。

### 第二步：启动broker服务。

启动broker服务之前，要做一个小小的配置。进入RocketMQ安装目录下的conf目录，修改broker.conf文件，在文件最后面加入一个配置：

```
autoCreateTopicEnable=true
```

这个选项是为了便于进行后续实验。他的作用是允许 broker 端自动创建新的 Topic。

另外，如果你的服务器配置了多张网卡，比如阿里云，腾讯云这样的云服务器，他们通常有内网网卡和外网网卡两张网卡，那么需要增加配置brokerIP1属性，指向服务器的外网IP 地址，这样才能确保从其他服务器上访问到RocketMQ 服务。

然后也可以用之前的方式启动broker服务。启动broker服务的指令是mqbroker

```
cd /app/rocketmq/rocketmq-all-4.9.5-bin-release
nohup bin/mqbroker &
```

启动完成后，同样检查nohup.out日志文件，有如下一条关键日志，就表示broker服务启动正常了。

```
The broker[xxxxxx] boot success. serializeType=JSON
```

注：1、在实际服务部署时，通常会将RocketMQ的部署地址添加到环境变量当中。例如使用vi ~/.bash\_profile指令，添加以下内容。

```
export ROCKETMQ_HOME=/app/rocketmq/rocketmq-all-4.9.5-bin-release
PATH=$ROCKETMQ_HOME/bin:$PATH
export PATH
```

这样就不必每次进入RocketMQ的安装目录了。直接可以使用mqnamesrv 和mqbroker指令。

2、停止RocketMQ服务可以通过mqshutdown指令进行

```
mqshutdown namesrv # 关闭nameserver服务
```

```
mqshutdown broker # 关闭broker服务
```

同样使用jps指令可以检查服务的启动状态。使用jps指令后，可以看到一个名为BrokerStartup的进程，则表示broker服务启动完成。

## 2、快速实现消息收发

RocketMQ后端服务启动完成后，就可以启动客户端的消息生产者和消息消费者进行消息转发了。接下来，我们会先通过RocketMQ提供的命令行工具快速体验一下RocketMQ消息收发的功能。然后，再动手搭建一个Maven项目，在项目中使用RocketMQ进行消息收发。

### 1、命令行快速实现消息收发

**第一步：**需要配置一个环境变量NAMESRV\_ADDR，只想我们之前启动的nameserver服务。

通过vi ~/.bash\_profile添加以下配置。然后使用source ~/.bash\_profile让配置生效。

```
export NAMESRV_ADDR='localhost:9876'
```

**第二步：**通过指令启动RocketMQ的消息生产者发送消息。

```
tools.sh org.apache.rocketmq.example.quickstart.Producer
```

这个指令会默认往RocketMQ中发送1000条消息。在命令行窗口可以看到发送消息的日志：

```
.....
SendResult [sendStatus=SEND_OK, msgId=C0A8E88007AC3764951D891CE9A003E7,
offsetMsgId=C0A8E88000002A9F000000000000317BF, messageQueue=MessageQueue
[topic=TopicTest, brokerName=worker1, queueId=1], queueOffset=249]

14:59:33.418 [NettyClientSelector_1] INFO RocketmqRemoting - closeChannel:
close the connection to remote address[127.0.0.1:9876] result: true

14:59:33.423 [NettyClientSelector_1] INFO RocketmqRemoting - closeChannel:
close the connection to remote address[192.168.232.128:10911] result: true
```

这部分日志中，并没有打印出发送了什么消息。上面SendResult开头部分是消息发送到Broker后的结果。最后两行日志表示消息生产者发完消息后，服务正常关闭了。

**第三步：**可以启动消息消费者接收之前发送的消息

```
tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

消费者启动完成后，可以看到消费到的消息

```
.....
ConsumeMessageThread_19 Receive New Messages: [MessageExt
[brokerName=worker1, queueId=2, storeSize=203, queueOffset=53, sysFlag=0,
bornTimestamp=1606460371999, bornHost=/192.168.232.128:43436,
storeTimestamp=1606460372000, storeHost=/192.168.232.128:10911,
msgId=C0A8E88000002A9F000000000000A7AE, commitLogOffset=42926,
bodyCRC=1968636794, reconsumeTimes=0, preparedTransactionOffset=0,
toString()=Message{topic='TopicTest', flag=0, properties={MIN_OFFSET=0,
MAX_OFFSET=250, CONSUME_START_TIME=1606460450150,
UNIQ_KEY=C0A8E88007AC3764951D891CE41F00D4, CLUSTER=DefaultCluster,
WAIT=true, TAGS=TagA}, body=[72, 101, 108, 108, 111, 32, 82, 111, 99, 107,
101, 116, 77, 81, 32, 50, 49, 50], transactionId='null'}]]
```

每一条这样的日志信息就表示消费者接收到了一条消息。

这个Consumer消费者的指令并不会主动结束，他会继续挂起，等待消费新的消息。我们可以使用CTRL+C停止该进程。

注：在RocketMQ提供的这个简单示例中并没有打印出传递的消息内容，而是打印出了消息相关的很多重要的属性。

其中有几个比较重要的属性：brokerId,brokerName,queueId,msgId,topic,cluster。这些属性的作用会在后续一起分享，这里你不妨先找一下这些属性是什么，消费者与生产者之间有什么样的对应关系。

## 2、搭建Maven客户端项目

之前的步骤实际上是在服务器上快速验证RocketMQ的服务状态，接下来我们动手搭建一个RocketMQ的客户端应用，在实际应用中集成使用RocketMQ。

**第一步：**创建一个标准的maven项目，在pom.xml中引入以下核心依赖



```
<dependency>
  <groupId>org.apache.rocketmq</groupId>
  <artifactId>rocketmq-client</artifactId>
  <version>4.9.5</version>
</dependency>
```

第二步：就可以直接创建一个简单的消息生产者

```
public class Producer {
    public static void main(String[] args) throws MQClientException,
    InterruptedException {
        //初始化一个消息生产者
        DefaultMQProducer producer = new
        DefaultMQProducer("please_rename_unique_group_name");
        // 指定nameserver地址
        producer.setNamesrvAddr("192.168.232.128:9876");
        // 启动消息生产者服务
        producer.start();
        for (int i = 0; i < 2; i++) {
            try {
                // 创建消息。消息由Topic,Tag和body三个属性组成，其中Body就是消息内容
                Message msg = new Message("TopicTest", "TagA", ("Hello RocketMQ "
                +i).getBytes(RemotingHelper.DEFAULT_CHARSET));
                //发送消息，获取发送结果
                SendResult sendResult = producer.send(msg);
                System.out.printf("%s\n", sendResult);
            } catch (Exception e) {
                e.printStackTrace();
                Thread.sleep(1000);
            }
        }
        //消息发送完后，停止消息生产者服务。
        producer.shutdown();
    }
}
```

运行其中的main方法，就会往RocketMQ中发送两条消息。在这个实现过程中，需要注意一下的是对于生产者，需要指定对应的nameserver服务的地址，这个地址需要指向你自己的服务器。

第三步：创建一个消息消费者接收RocketMQ中的消息。

```
public class Consumer {
    public static void main(String[] args) throws InterruptedException,
    MQClientException {
        //构建一个消息消费者
        DefaultMQPushConsumer consumer = new
        DefaultMQPushConsumer("please_rename_unique_group_name_4");
        //指定nameserver地址
```



```

consumer.setNamesrvAddr("192.168.232.128:9876");
consumer.setConsumeFromWhere(ConsumeFromWhere.CONSUME_FROM_LAST_OFFSET);
// 订阅一个感兴趣的话题，这个话题需要与消息的topic一致
consumer.subscribe("TopicTest", "*");
// 注册一个消息回调函数，消费到消息后就会触发回调。
consumer.registerMessageListener(new MessageListenerConcurrently() {
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs, ConsumeConcurrentlyContext context) {
        msgs.forEach(messageExt -> {
            try {
                System.out.println("收到消息:" + new String(messageExt.getBody(),
RemotingHelper.DEFAULT_CHARSET));
            } catch (UnsupportedEncodingException e) {}
        });
        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
});
//启动消费者服务
consumer.start();
System.out.print("Consumer Started");
}
}

```

运行其中的main方法后，就可以启动一个RocketMQ消费者，接收之前发到RocketMQ上的消息，并将消息内容打印出来。在这个实现过程中，需要重点关注的有两点。一是对于消费者，同样需要指定nameserver的地址。二是消费者需要在RocketMQ中订阅具体的Topic，只有发送到这个Topic上的消息才会被这个消费者接收到。

示例代码不用强行记忆，在RocketMQ的源码包中有个example模块，其中就有这些示例代码。

这样，通过几个简单的步骤，我们就完成了RocketMQ的应用集成。从这个过程中可以看到，RocketMQ的使用是比较简单的。但是这并不意味着这几个简单的步骤就足够搭建一个生产级别的RocketMQ服务。接下来，我们会一步步把我们这个简单的RocketMQ服务往一个生产级别的服务集群推进。

### 3、搭建RocketMQ可视化管理服务

在之前的简单实验中，RocketMQ都是以后台服务的方式在运行，我们并不很清楚RocketMQ是如何运行的。RocketMQ的社区就提供了一个图形化的管理控制台Dashboard，可以用可视化的方式直接观测并管理RocketMQ的运行过程。

Dashboard服务并不在RocketMQ的运行包中，需要到RocketMQ的官网下载页面单独下载。

# RocketMQ Dashboard

版本	文档	发布日期	Source 下载	Binary 下载	End of Support
1.0.0	<a href="#">Release Notes</a>	2021.9.23	<a href="#">download .zip</a>		

这里只提供了源码，并没有提供直接运行的jar包。将源码下载下来后，需要解压并进入对应的目录，使用maven进行编译。(需要提前安装maven客户端)

```
mvn clean package -Dmaven.test.skip=true
```

编译完成后，在源码的target目录下会生成可运行的jar包rocketmq-dashboard-1.0.1-SNAPSHOT.jar。接下来可以将这个jar包上传到服务器上。我们上传到/app/rocketmq/rocketmq-dashboard目录下

接下来我们需要在jar包所在的目录下创建一个application.yml配置文件，在配置文件中做如下配置：

```
rocketmq:
  config:
    namesrvAddrs:
      - 192.168.232.128:9876
```

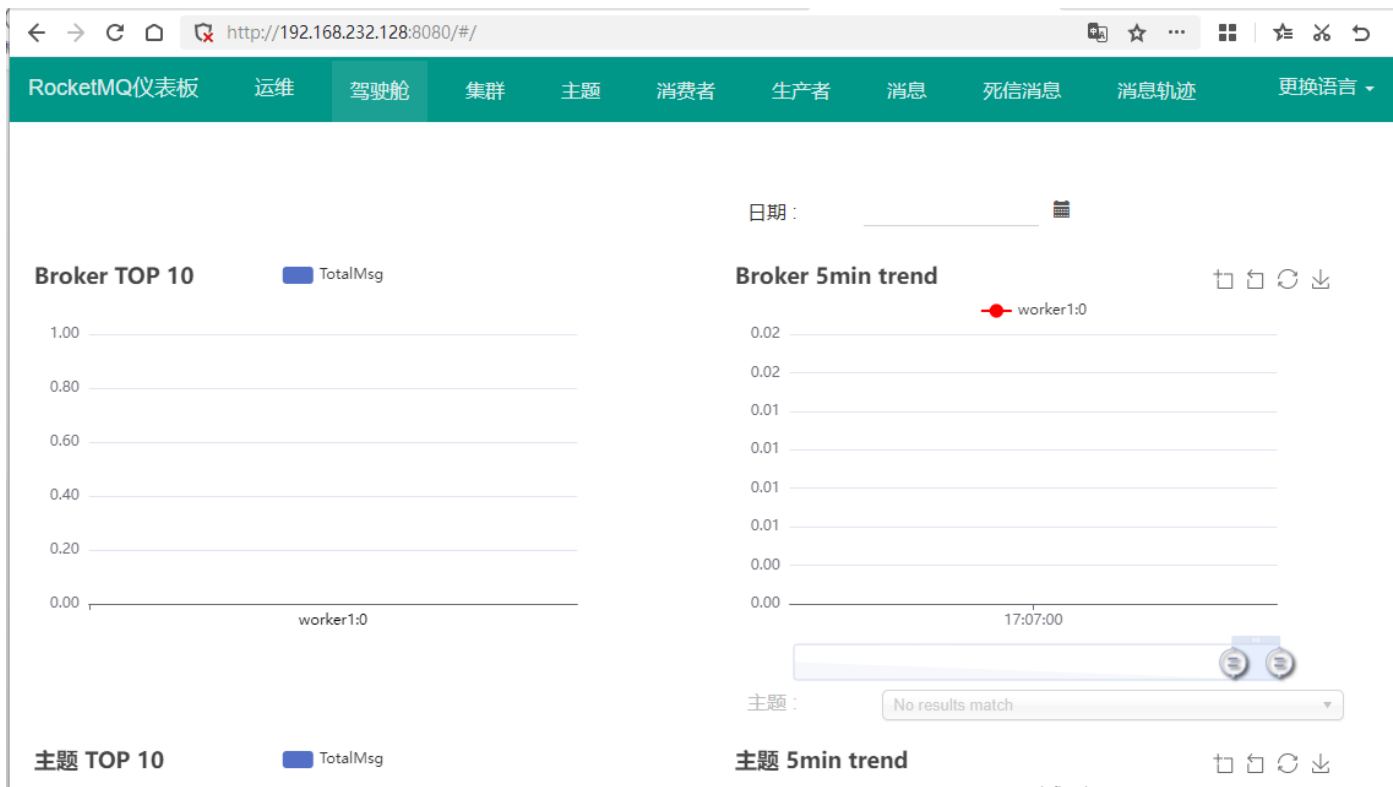
主要是要指定nameserver的地址。

注：关于这个配置文件中更多的配置选项，可以参考一下dashboard源码当中的application.yml配置文件。

接下来就可以通过java指令执行这个jar包，启动管理控制台服务。

```
java -jar rocketmq-dashboard-1.0.1-SNAPSHOT.jar
```

应用启动完成后，会在服务器上搭建起一个web服务，我们就可以通过访问<http://192.168.232.128:8080>查看到管理页面。



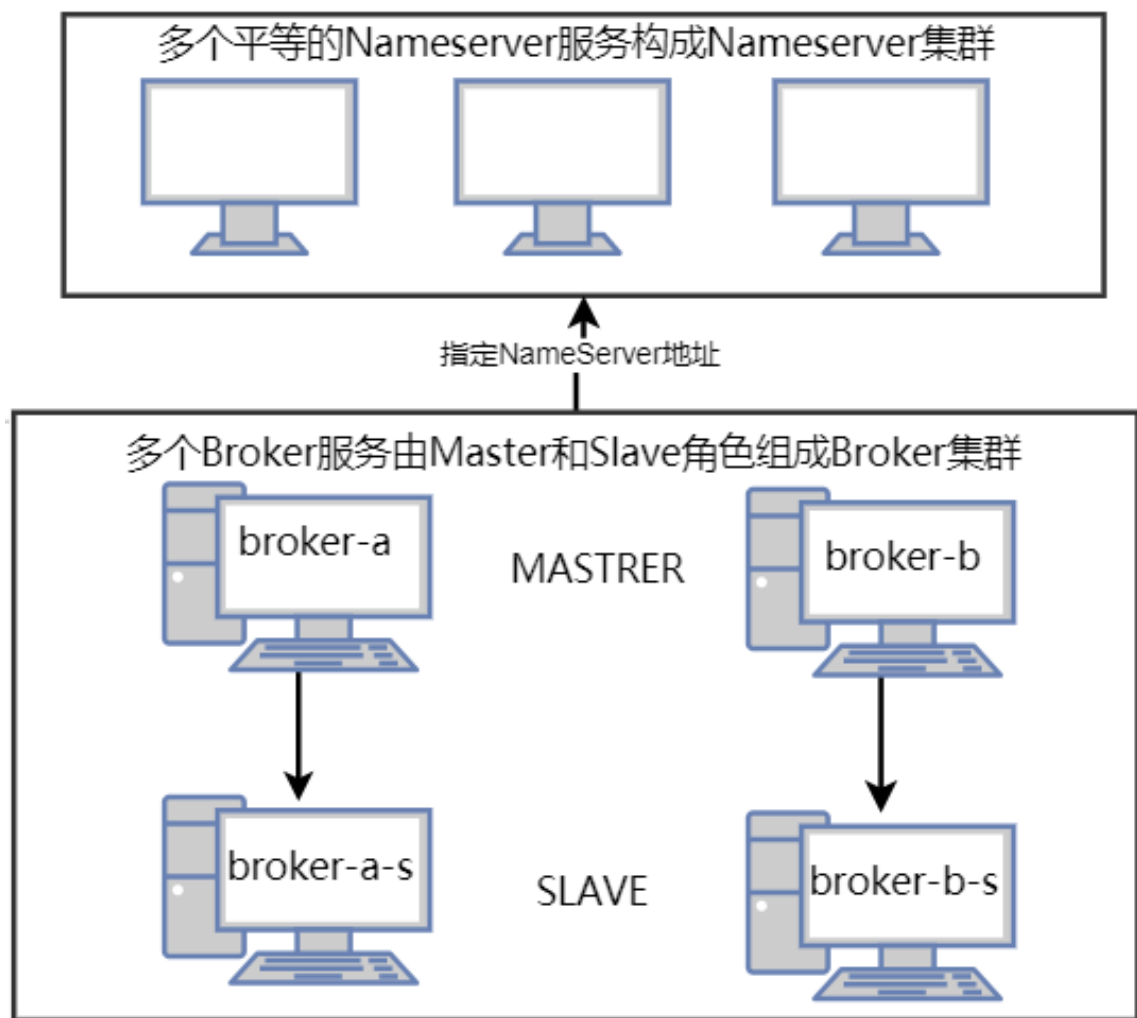
这个管理控制台的功能非常全面。驾驶舱页面展示RocketMQ近期的运行情况。运维页面主要是管理nameserver服务。集群页面主要管理RocketMQ的broker服务。很多信息都一目了然。在之后的过程中，我们也会逐渐了解DashBoard管理页面中更多的细节。

## 4、升级分布式集群

之前我们用一台Linux服务器，快速搭建起了一整套RocketMQ的服务。但是很明显，这样搭建的服务是无法放到生产环境上去用的。一旦nameserver服务或者broker服务出现了问题，整个RocketMQ就无法正常工作。而且更严重的是，如果服务器出现了问题，比如磁盘坏了，那么存储在磁盘上的数据就会丢失。这时RocketMQ暂存到磁盘上的消息也会跟着丢失，这个问题就非常严重了。因此，我们需要搭建一个分布式的RocketMQ服务集群，来防止单点故障问题。

RocketMQ的分布式集群基于主从架构搭建。在多个服务器组成的集群中，指定一部分节点作为Master节点，负责响应客户端的请求。指令另一部分节点作为Slave节点，负责备份Master节点上的数据，这样，当Master节点出现故障时，在Slave节点上可以保留有数据备份，至少保证数据不会丢失。

整个集群方案如下图所示：



接下来我们准备三台相同的Linux服务器，搭建一下RocketMQ的分布式集群。为了更清晰的描述这三台服务器上的操作，我们给每个服务器指定一个机器名。

```
cat /etc/hosts
192.168.232.128 worker1
192.168.232.129 worker2
192.168.232.130 worker3
```

为了便于观察，我们这次搭建一个2主2从的RocketMQ集群，并将主节点和节点都分别部署在不同的服务器上。预备的集群规划情况如下：

机器名	nameServer服务部署	broker服务部署
worker1	nameServer	
worker2	nameServer	broker-a,broker-b-s
worker3	nameServer	broker-a-s,broker-b

第一步：部署nameServer服务。

nameServer服务不需要做特别的配置，按照之前的步骤，在三台服务器上都分别部署nameServer服务即可。

## 第二步：对Broker服务进行集群配置。

这里需要修改RocketMQ的配置文件，对broker服务做一些集群相关的参数部署。这些配置文件并不需要我们手动进行创建，在RocketMQ运行包的conf目录下，提供了多种集群的部署配置文件模板。

- 2m-noslave: 2主无从的集群参考配置。这种集群存在单点故障。
- 2m-2s-async和2m-2s-sync: 2主2从的集群参考配置。其中async和sync表示主节点与从节点之间是同步同步还是异步同步。关于这两个概念，会在后续章节详细介绍
- dledger: 具备主从切换功能的高可用集群。集群中的节点会基于Raft协议随机选举出一个Leader，其作用类似于Master节点。其他的节点都是follower，其作用类似于Slave节点。

我们这次采用2m-2s-async的方式搭建集群，需要在worker2和worker3上修改这个文件夹下的配置文件。

### 1> 配置第一组broker-a服务

在worker2机器上配置broker-a的MASTER服务，需要修改conf/2m-2s-async/broker-a.properties。示例配置如下：

```
#所属集群名字，名字一样的节点就在同一个集群内
brokerClusterName=rocketmq-cluster
#broker名字，名字一样的节点就是一组主从节点。
brokerName=broker-a
#brokerid,0就表示是Master，>0的都是表示 Slave
brokerId=0
#nameServer地址，分号分割
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
#是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
deleteWhen=04
fileReservedTime=120
#存储路径
storePathRootDir=/app/rocketmq/store
storePathCommitLog=/app/rocketmq/store/commitlog
storePathConsumeQueue=/app/rocketmq/store/consumequeue
storePathIndex=/app/rocketmq/store/index
storeCheckpoint=/app/rocketmq/store/checkpoint
abortFile=/app/rocketmq/store/abort
#Broker 的角色
brokerRole=ASYNC_MASTER
flushDiskType=ASYNC_FLUSH
#Broker 对外服务的监听端口
listenPort=10911
```

这里对几个需要重点关注的属性，做下简单介绍：

- brokerClusterName: 集群名。RocketMQ会将同一个局域网下所有brokerClusterName相同的服务自动组成一个集群，这个集群可以作为一个整体对外提供服务
- brokerName: Broker服务名。同一个RocketMQ集群当中，brokerName相同的多个服务会有一套相同的数据副本。同一个RocketMQ集群中，是可以将消息分散存储到多个不同的brokerName服务上的。

- brokerId: RocketMQ中对每个服务的唯一标识。RocketMQ对brokerId定义了一套简单的规则，master节点需要固定配置为0，负责响应客户端的请求。slave节点配置成其他任意数字，负责备份master上的消息。
- brokerRole: 服务的角色。这个属性有三个可选项：ASYNC\_MASTER，SYNC\_MASTER和SLAVE。其中，ASYNC\_MASTER和SYNC\_MASTER表示当前节点是master节点，目前暂时不用关心他们的区别。SLAVE则表示从节点。
- namesrvAddr: nameserver服务的地址。nameserver服务默认占用9876端口。多个nameserver地址用；隔开。

接下来在worker3上配置broker-a的SLAVE服务。需要修改conf/2m-2s-async/broker-a-s.properties。示例配置如下：

```
#所属集群名字，名字一样的节点就在同一个集群内
brokerClusterName=rocketmq-cluster
#broker名字，名字一样的节点就是一组主从节点。
brokerName=broker-a
#brokerid,0就表示是Master，>0的都是表示 Slave
brokerId=1
#nameServer地址，分号分割
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
#是否允许 Broker 自动创建Topic，建议线下开启，线上关闭
autoCreateTopicEnable=true
deleteWhen=04
fileReservedTime=120
#存储路径
storePathRootDir=/app/rocketmq/storeSlave
storePathCommitLog=/app/rocketmq/storeSlave/commitlog
storePathConsumeQueue=/app/rocketmq/storeSlave/consumequeue
storePathIndex=/app/rocketmq/storeSlave/index
storeCheckpoint=/app/rocketmq/storeSlave/checkpoint
abortFile=/app/rocketmq/storeSlave/abort
#Broker 的角色
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
#Broker 对外服务的监听端口
listenPort=11011
```

其中关键是brokerClusterName和brokerName两个参数需要与worker2上对应的broker-a.properties配置匹配。brokerId配置0以外的数字。然后brokerRole配置为SLAVE。

这样，第一组broker服务就配置好了。

## 2> 配置第二组broker-b服务

与第一组broker-a服务的配置方式类似，在worker3上配置broker-b的MASTER服务。需要修改conf/2m-2s-async/broker-b.properties文件，配置示例如下：

```
#所属集群名字，名字一样的节点就在同一个集群内
brokerClusterName=rocketmq-cluster
#broker名字，名字一样的节点就是一组主从节点。
```

```

brokerName=broker-b
#brokerid,0就表示是Master, >0的都是表示 Slave
brokerId=0
#nameServer地址, 分号分割
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
#是否允许 Broker 自动创建Topic, 建议线下开启, 线上关闭
autoCreateTopicEnable=true
deleteWhen=04
fileReservedTime=120
#存储路径
storePathRootDir=/app/rocketmq/store
storePathCommitLog=/app/rocketmq/store/commitlog
storePathConsumeQueue=/app/rocketmq/store/consumequeue
storePathIndex=/app/rocketmq/store/index
storeCheckpoint=/app/rocketmq/store/checkpoint
abortFile=/app/rocketmq/store/abort
#Broker 的角色
brokerRole=ASYNC_MASTER
flushDiskType=ASYNC_FLUSH
#Broker 对外服务的监听端口
listenPort=10911

```

在worker2上配置broker-b的SLAVE服务。需要修改conf/2m-2s-async/broker-b-s.properties文件，配置示例如下：

```

#所属集群名字, 名字一样的节点就在同一个集群内
brokerClusterName=rocketmq-cluster
#broker名字, 名字一样的节点就是一组主从节点。
brokerName=broker-b
#brokerid,0就表示是Master, >0的都是表示 Slave
brokerId=1
#nameServer地址, 分号分割
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
#是否允许 Broker 自动创建Topic, 建议线下开启, 线上关闭
autoCreateTopicEnable=true
deleteWhen=04
fileReservedTime=120
#存储路径
storePathRootDir=/app/rocketmq/storeSlave
storePathCommitLog=/app/rocketmq/storeSlave/commitlog
storePathConsumeQueue=/app/rocketmq/storeSlave/consumequeue
storePathIndex=/app/rocketmq/storeSlave/index
storeCheckpoint=/app/rocketmq/storeSlave/checkpoint
abortFile=/app/rocketmq/storeSlave/abort
#Broker 的角色
brokerRole=SLAVE
flushDiskType=ASYNC_FLUSH
#Broker 对外服务的监听端口
listenPort=11011

```



这样就完成了2主2从集群的配置。配置过程汇总有几个需要注意的配置项：

- store开头的一系列配置：表示RocketMQ的存盘文件地址。在同一个机器上需要部署多个Broker服务时，不同服务的存储目录不能相同。
- listenPort：表示Broker对外提供服务的端口。这个端口默认是10911。在同一个机器上部署多个Broker服务时，不同服务占用的端口也不能相同。
- 如果你使用的是多网卡的服务器，比如阿里云上的云服务器，那么就需要在配置文件中增加配置一个brokerIP1属性，指向所在机器的外网网卡地址。

### 第三步：启动Broker服务

集群配置完成后，需要启动Broker服务。与之前启动broker服务稍有不同，启动时需要增加-c参数，指向我们修改的配置文件。

在worker2上启动broker-a的master服务和broker-b的slave服务：

```
cd /app/rocketmq/rocketmq-all-4.9.5-bin-release
nohup bin/mqbroker -c ./conf/2m-2s-async/broker-a.properties &
nohup bin/mqbroker -c ./conf/2m-2s-async/broker-b-s.properties &
```

在worker3上启动broker-b的master服务和broker-a的slave服务：

```
cd /app/rocketmq/rocketmq-all-4.9.5-bin-release
nohup bin/mqbroker -c ./conf/2m-2s-async/broker-b.properties &
nohup bin/mqbroker -c ./conf/2m-2s-async/broker-a-s.properties &
```

### 第四步：检查集群服务状态

对于服务的启动状态，我们依然可以用之前介绍的jps指令以及nohup.out日志文件进行跟踪。不过，在RocketMQ的bin目录下，也提供了mqadmin指令，可以通过命令行的方式管理RocketMQ集群。

例如下面的指令可以查看集群broker集群状态。通过这个指令可以及时了解集群的运行状态。

```
[oper@worker1 bin]$ cd /app/rocketmq/rocketmq-all-4.9.5-bin-release/bin
[oper@worker1 bin]$ mqadmin clusterList
RocketMQLog:WARN No appenders could be found for logger
(io.netty.util.internal.InternalThreadLocalMap).
RocketMQLog:WARN Please initialize the logger system properly.
#Cluster Name      #Broker Name      #BID #Addr      #Version
#InTPS(LOAD)      #OutTPS(LOAD) #PCWait(ms) #Hour #SPACE
rocketmq-cluster  broker-a          0    192.168.232.129:10911  V4_9_1
0.00(0,0ms)       0.00(0,0ms)      0  3425.28 0.3594
rocketmq-cluster  broker-a          1    192.168.232.130:11011  V4_9_1
0.00(0,0ms)       0.00(0,0ms)      0  3425.28 0.3607
rocketmq-cluster  broker-b          0    192.168.232.130:10911  V4_9_1
0.00(0,0ms)       0.00(0,0ms)      0  3425.27 0.3607
rocketmq-cluster  broker-b          1    192.168.232.129:11011  V4_9_1
0.00(0,0ms)       0.00(0,0ms)      0  3425.27 0.3594
```

注：执行这个指令需要在机器上配置了NAMESRV环境变量

mqadmin指令还提供了非常丰富的管理功能。你可以尝试直接使用mqadmin指令，就会列出mqadmin支持的所有管理指令。如果对某一个指令不会使用，还可以使用mqadmin help 指令查看帮助。

另外，之前搭建的dashboard也是集群服务状态的很好的工具。只需要在之前搭建Dashboard时创建的配置文件中增加指定nameserver地址即可。

```
rocketmq:
  config:
    namesrvAddr:
      - worker1:9876
      - worker2:9876
      - worker3:9876
```

启动完成后，在集群菜单页就可以看到集群的运行情况

RocketMQ仪表板 运维 驾驶舱 集群 主题 消费者 生产者 消息 死信消息 消息轨迹 更换语言

集群: rocketmq-cluster

分片	编号	地址	版本	生产消息TPS	消费消息TPS	昨日生产总数	昨日消费总数	今天生产总数	今天消费总数	操作
broker-b	0(master)	192.168.65.205:10911	V4_9_5	0.00	0.00	0	0	0	0	状态 配置
broker-b	1(slave)	192.168.65.159:11011	V4_9_5	0.00	0.00	0	0	0	0	状态 配置
broker-a	0(master)	192.168.65.159:10911	V4_9_5	0.00	0.00	0	0	0	0	状态 配置
broker-a	1(slave)	192.168.65.205:11011	V4_9_5	0.00	0.00	0	0	0	0	状态 配置

在RocketMQ的这种主从架构的集群下，客户端发送的消息会分散保存到broker-a和broker-b两个服务上，然后每个服务都配有slave服务，可以备份对应master服务上的消息，这样就可以防止单点故障造成的消息丢失问题。

## 5、升级高可用集群

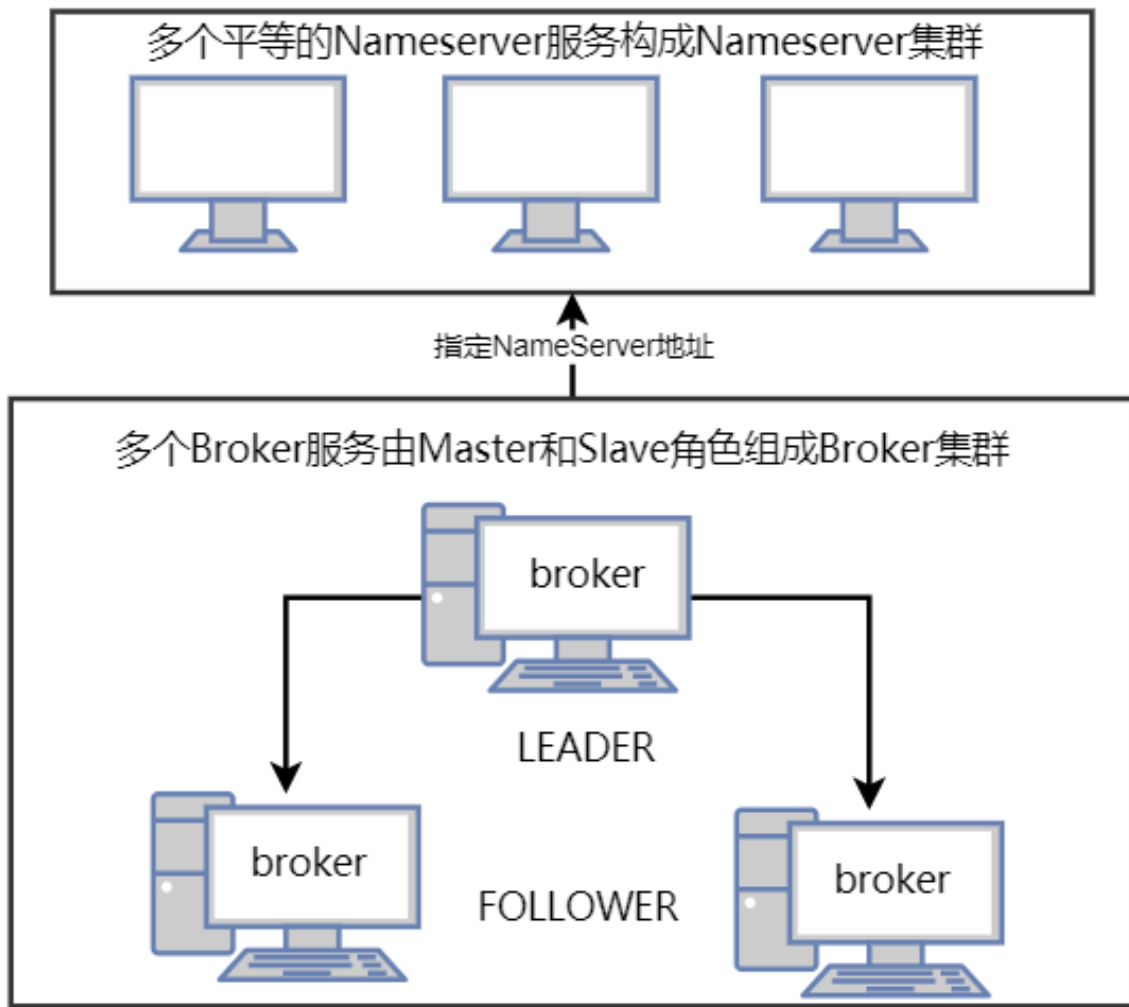
主从架构的RocketMQ集群，由于给每个broker服务配置了一个或多个slave备份服务，可以保证当broker服务出现问题时，broker上的消息不会丢失。但是，这种主从架构的集群却也有一个不足的地方，那就是不具备服务高可用。

这里所说的服务高可用，并不是并不是指整个RocketMQ集群就不能对外提供服务了，而是指集群中的消息就不完整了。实际上，当RocketMQ集群中的broker宕机后，整个集群会自动进行broker状态感知。后续客户端的各种请求，依然可以转发到其他正常的broker上。只不过，原本保存在当前broker上的消息，就无法正常读取了，需要等到当前broker服务重启后，才能重新被消息消费者读取。

当一个broker上的服务宕机后，我们可以从对应的slave服务上找到broker上所有的消息。但是很可惜，主从架构中各个服务的角色都是固定了的，slave服务虽然拥有全部的数据，但是它没办法升级成为master服务去响应客户端的请求，依然只是傻傻等待master服务重启后，继续做它的数据备份工作。

这时，我们自然就希望这个slave服务可以升级成为master服务，继续响应客户端的各种请求，这样整个集群的消息服务就不会有任何中断。而RocketMQ提供的Dledger集群，就是具备角色自动转换功能的高可用集群。

整个集群结构如下图所示：



在Dledger集群中，就不再单独指定各个broker的服务，而是由这些broker服务自行进行选举，产生一个Leader角色的服务，响应客户端的各种请求。而其他的broker服务，就作为Follower角色，负责对Leader上的数据进行备份。当然，Follower所要负责的事情，比主从架构中的SLAVE角色会要复杂一点，因为这种节点选举是在后端不断进行的，他们需要随时做好升级成Leader的准备。

Dledger集群的选举是通过Raft协议进行的，Raft协议是一种多数同意机制。也就是每次选举需要有集群中超过半数的节点确认，才能形成整个集群的共同决定。同时，这也意味着在Dledger集群中，只要有超过半数的节点能够正常工作，那么整个集群就能正常工作。因此，在部署Dledger集群时，通常都是部署奇数台服务，这样可以让集群的容错性达到最大。

接下来，我们就用之前准备的3台服务器，搭建一个3个节点的Dledger集群。在这个集群中，只需要有2台Broker服务正常运行，这个集群就能正常工作。

### 第一步：部署nameserver

这一步和之前部署主从集群没有区别，不需要做过多的配置，直接在三台服务器上启动nameserver服务即可。

实际上，如果你是从上一个主从架构开始搭建起来的话，那么nameserver集群都不需要重新启动，nameserver会自动感知到broker的变化。

### 第二步：对Broker服务进行集群配置。

对于Dledger集群的配置，RocketMQ依然贴心的给出了完整的示例，不需要强行记忆。

在conf/dledger目录下，RocketMQ默认给出了三个配置文件，这三个配置文件可以在单机情况下直接部署成一个具有三个broker服务的Dledger集群，我们只需要按照这个配置进行修改即可。

注：在RocketMQ运行包的bin/dledger目录下，RocketMQ还提供了一个fast-try.sh脚本。这个脚本会指定conf/dledger目录下的配置文件，直接启动有三个broker服务的Dledger集群。每个集群指定的内存大小占用1G。

接下来我们可以在三台机器的conf/dledger目录下，都创建一个broker.conf文件，对每个broker服务进行配置。

worker1的broker.conf配置示例

```
brokerClusterName = RaftCluster
brokerName=RaftNode00
listenPort=30911
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
storePathRootDir=/app/rocketmq/storeDledger/
storePathCommitLog=/app/rocketmq/storeDledger/commitlog
storePathConsumeQueue=/app/rocketmq/storeDledger/consumequeue
storePathIndex=/app/rocketmq/storeDledger/index
storeCheckpoint=/app/rocketmq/storeDledger/checkpoint
abortFile=/app/rocketmq/storeDledger/abort
enableDLegerCommitLog=true
dLegerGroup=RaftNode00
dLegerPeers=n0-worker1:40911;n1-worker2:40911;n2-worker3:40911
## must be unique
dLegerSelfId=n0
sendMessageThreadPoolNums=16
```

worker2的broker.conf配置示例：

```
brokerClusterName = RaftCluster
brokerName=RaftNode00
listenPort=30911
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
storePathRootDir=/app/rocketmq/storeDledger/
storePathCommitLog=/app/rocketmq/storeDledger/commitlog
storePathConsumeQueue=/app/rocketmq/storeDledger/consumequeue
storePathIndex=/app/rocketmq/storeDledger/index
storeCheckpoint=/app/rocketmq/storeDledger/checkpoint
abortFile=/app/rocketmq/storeDledger/abort
enableDLegerCommitLog=true
dLegerGroup=RaftNode00
dLegerPeers=n0-worker1:40911;n1-worker2:40911;n2-worker3:40911
## must be unique
dLegerSelfId=n1
sendMessageThreadPoolNums=16
```

worker3的broker.conf配置示例：

```
brokerClusterName = RaftCluster
brokerName=RaftNode00
```

```
listenPort=30911
namesrvAddr=worker1:9876;worker2:9876;worker3:9876
storePathRootDir=/app/rocketmq/storeDledger/
storePathCommitLog=/app/rocketmq/storeDledger/commitlog
storePathConsumeQueue=/app/rocketmq/storeDledger/consumequeue
storePathIndex=/app/rocketmq/storeDledger/index
storeCheckpoint=/app/rocketmq/storeDledger/checkpoint
abortFile=/app/rocketmq/storeDledger/abort
enableDLegerCommitLog=true
dLegerGroup=RaftNode00
dLegerPeers=n0-worker1:40911;n1-worker2:40911;n2-worker3:40911
## must be unique
dLegerSelfId=n2
sendMessageThreadPoolNums=16
```

这里对几个需要重点关注的配置项，做下介绍：

- enableDLegerCommitLog: 是否启动Dledger。true表示启动
- namesrvAddr: 指定nameserver地址
- dLegerGroup: Dledger Raft Group的名字，建议跟brokerName保持一致。
- dLegerPeers: Dledger Group内各个服务节点的地址及端口信息。同一个Group内的各个节点配置必须要保持一致。
- dLegerSelfId: Dledger节点ID，必须属于dLegerPeers中的一个。同一个Group内的各个节点必须不能重复。
- sendMessageThreadPoolNums: dLeger内部发送消息的线程数，建议配置成cpu核心数。
- store开头的一系列配置：这些是配置dLeger集群的消息存盘目录。如果你是从主从架构升级成为dLeger架构，那么这个地址可以指向之前搭建住主从架构的地址。dLeger集群会兼容主从架构集群的消息格式，只不过主从架构的消息无法享受dLeger集群的两阶段同步功能。

### 第三步：启动broker服务

和启动主从架构的broker服务一样，我们只需要在启动broker服务时，指定配置文件即可。在三台服务器上分别执行以下指令，启动broker服务。

```
cd /app/rocketmq/rocketmq-all-4.9.5-bin-release/
nohup bin/mqbroker -c conf/dledger/broker.conf &
```

### 第四步：检查集群服务状态

我们可以在Dashboard控制台的集群菜单页看到Dledger集群的运行状况。

集群：

RaftCluster

分片	编号	地址	版本	生产消息TPS	消费消息TPS	昨日生产总数	昨日消费总数	今天生产总数	今天消费总数	操作
RaftNode00	0(master)	192.168.65.112:30911	V4_9_5	0.00	0.00	0	0	0	0	<div>状态配置</div>
RaftNode00	2(slave)	192.168.65.159:30911	V4_9_5	0.00	0.00	0	0	0	0	<div>状态配置</div>
RaftNode00	3(slave)	192.168.65.205:30911	V4_9_5	0.00	0.00	0	0	0	0	<div>状态配置</div>

从整个配置过程中可以看到，我们并没有指定每个节点的角色，而Dledger集群就自动将192.168.232.129也就是worker2上的broker服务选举成了master。

接下来如果你有兴趣，可以自己尝试下停止woker2上的broker服务，再重新观察集群的运行状况。RocketMQ会在发现worker2服务宕机后，很快的选举产生新的master节点。但具体选举出worker1还是worker3作为master，则是随机的。

但是，如果你尝试继续停止worker1或worker3上的broker服务，那么集群中宕机的broker服务就超过了半数，也就是两台。这时这个Dledger集群就选举不出master节点，也就无法正常工作了。

关于Dledger集群的一些补充

Dledger集群机制是RocketMQ自4.5版本开始支持的一个重要特性。他其实是由OpenMessage组织带入RocketMQ的一个系列框架。他是一个为高可用、高性能、高可靠的分布式存储系统提供基础支持的组件。他做的事情主要有两个，一是在集群中选举产生master节点。RocketMQ集群需要用这个master节点响应客户端的各种请求。二是在各种复杂的分布式场景下，保证CommitLog日志文件在集群中的强一致性。

以下是ChatGPT对于Dledger的功能描述

RocketMQ是一款分布式消息队列系统，主要用于处理大量数据的实时传输和处理。在RocketMQ中，DLedger是一个为高可用、高性能、高可靠的分布式存储系统提供基础支持的组件。DLedger集群主要具有以下功能：

- 数据复制：DLedger集群通过raft协议来保证数据的一致性。在集群中，每个节点都维护一个相同的数据副本，以确保当某个节点出现故障时，数据不会丢失。
- 容错性：DLedger集群具有很高的容错性。即使集群中的部分节点发生故障，只要集群中有大多数节点（即超过半数）仍在正常工作，整个集群将继续提供服务。
- 高可用性：DLedger集群通过负载均衡和热备份等机制，确保在节点故障时能够快速切换到其他正常节点，提高整个系统的可用性。
- 分布式锁：DLedger集群提供分布式锁功能，可以解决分布式系统中的资源争用问题，实现跨节点的资源同步。
- 强一致性：DLedger集群通过使用Raft一致性协议，确保在多个副本节点之间同步数据，保证数据的强一致性。
- 高性能：DLedger集群支持水平扩展，可以通过增加节点来提高系统的吞吐量和存储能力，以满足不断增长的业务需求。
- 易于管理：DLedger集群提供了一系列管理和监控功能，使运维人员可以更方便地掌握系统的运行状况，及时发现和解决问题。

总之，RocketMQ的DLedger集群具有高可用、高性能、高可靠等特点，为分布式消息队列系统提供了坚实的基础。



其背后的核心就是Raft协议。这是一种强大的分布式选举算法，其核心是只要集群中超过半数的节点作出的共同决议，就认为是集群最终的共同决议。

Raft协议通过投票机制保持数据一致性。详细的细节，我们这里不做过多讨论，只是给你介绍一下Raft协议一个很强大的地方，就是他解决了分布式集群中的脑裂问题。

关于脑裂问题，这是在集群选举过程中一个出现概率不高，但是让很多人头疼的问题。在分布式集群内，有可能会由于网络波动或者其他一些不稳定因素，造成集群内节点之间短时间通信不畅通。这时就容易在集群内形成多个包含多个节点的小集合。这些集合就会独立进行选举，各自产生新的Master节点。当网络恢复畅通后，集群中就有了多个Master节点。当集群中出现多个Master节点后，其他节点就不知道要听从谁的指令了，从而造成集群整体工作瘫痪。也就是俗话说的“一山不容二虎”。脑裂问题在以Zookeeper为代表的早一代分布式一致性产品中，是一个非常头疼的问题。而Raft协议对于脑裂问题，会采用随机休眠的机制，彻底解决脑裂问题。RocketMQ是Raft协议的一个重要的成功示例。Kafka也在之后基于Raft协议，自行实现了Kraft集群机制。

同样，附上ChatGPT对于脑裂问题的介绍，供你参考：

问题：Dledger集群如何防止集群脑裂问题？

Dledger集群通过使用Raft协议来防止集群脑裂（split-brain）问题。脑裂问题是指在分布式系统中，由于网络分区或其他原因导致集群被分割成两个或多个子集群，各自独立运行且无法感知到其他子集群的存在。这可能导致数据不一致和错误决策。Raft协议采用了一系列措施来避免脑裂问题的发生：

1. 选举机制：Raft协议的基础是选举出一个领导者（Leader），其他节点（Follower）都从领导者获取数据。选举过程要求候选人必须获得集群中大多数节点的支持才能成为领导者。这确保了集群中只能有一个领导者，从而避免了脑裂问题。

2. 任期（Term）：Raft协议为每个选举周期设置了一个递增的任期编号。任期编号用于标识当前的领导者，确保旧的领导者不会再次被选为领导者。如果一个节点发现自己的任期小于其他节点，那么它会停止当前的工作并更新自己的任期。

3. 心跳机制：领导者会定期向其他节点发送心跳消息，以保持与Follower节点的连接。当一个节点长时间未收到领导者的心跳时，它会认为当前领导者失效，并启动新一轮选举。这确保了当领导者出现故障时，系统能够快速选出新的领导者。

4. 日志复制：领导者负责将数据更新（日志条目）复制到其他节点。Follower节点只有在收到领导者的日志条目并将其写入本地日志后，才会响应客户端的请求。这确保了在发生脑裂情况下，不会出现多个节点试图同时修改同一份数据的情况。

通过以上措施，Dledger集群利用Raft协议避免了脑裂问题的发生，保证了系统的高可用性和数据一致性。

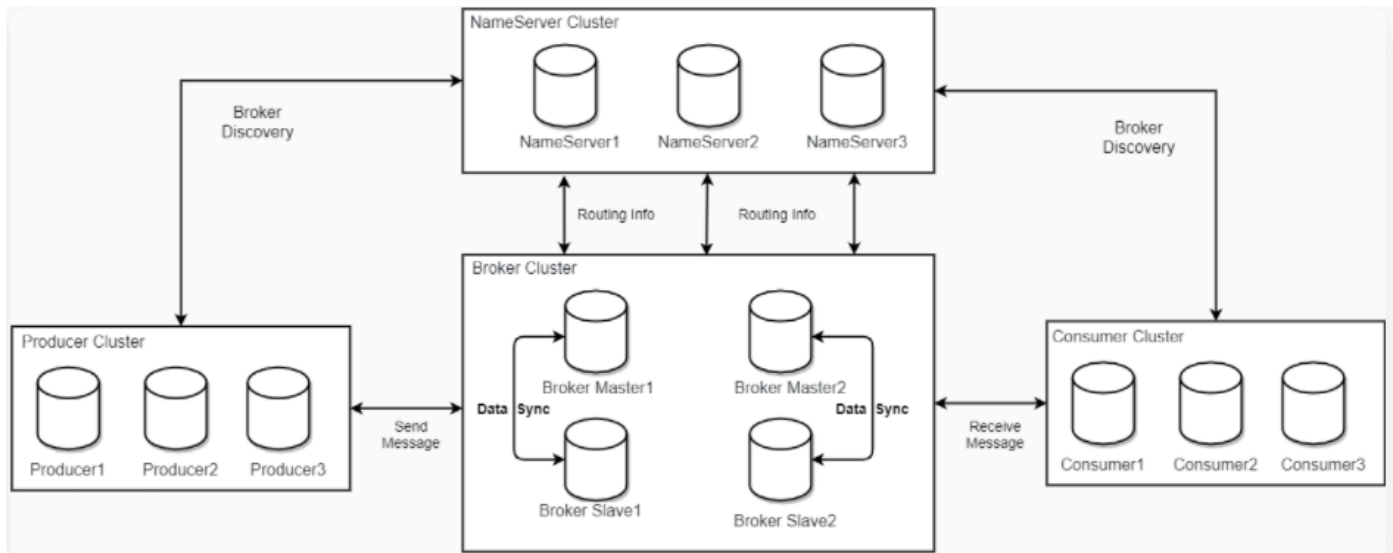
注：Dledger集群由于会接管RocketMQ原生的文件写入，所以，Dledger集群的文件写入速度比RocketMQ的原生写入方式是要慢一点的。这会对RocketMQ的性能产生一些影响。所以，当前版本的Dledger集群在企业中用得并不是太多。5.0版本对Dledger集群抽出了一种Dledger Controller模式，也就是只用Dledger集群的选举功能，而不用他的Commit文件写入功能，这样性能可以得到一定的提升。

## 四、总结RocketMQ的运行架构

通过之前的一系列实验，相信你对RocketMQ的运行机制有了一个大致的了解。接下来我们结合一下之前实验的过程，来理解一下RocketMQ的运行架构。

下图是RocketMQ运行时的整体架构：





接下来，我们就完整梳理一下RocketMQ中各个组件的作用：

### 1、nameServer 命名服务

在我们之前的实验过程中，你会发现，nameServer不依赖于任何其他的服务，自己独立就能启动。并且，不管是broker还是客户端，都需要明确指定nameServer的服务地址。以一台电脑为例，nameServer可以理解是整个RocketMQ的CPU，整个RocketMQ集群都要在CPU的协调下才能正常工作。

### 2、broker 核心服务

从之前的实验过程中你会发现，broker是RocketMQ中最为娇贵的一个组件。RocketMQ提供了各种各样的重要设计来保护broker的安全。同时broker也是RocketMQ中配置最为繁琐的部分。同样以电脑为例，broker就是整个RocketMQ中的硬盘、显卡这一类的核心硬件。RocketMQ最核心的消息存储、传递、查询等功能都要由broker提供。

### 3、client 客户端

Client包括消息生产者和消息消费者。同样以电脑为例，Client可以认为是RocketMQ中的键盘、鼠标、显示器这类的输入输出设备。鼠标、键盘输入的数据需要传输到硬盘、显卡等硬件才能进行处理。但是键盘、鼠标是不能直接将数据输入到硬盘、显卡的，这就需要CPU进行协调。通过CPU，鼠标、键盘就可以将输入的数据最终传输到核心的硬件设备中。经过硬件设备处理完成后，再通过CPU协调，显示器这样的输出设备就能最终从核心硬件设备中获取到输出的数据。

## 五、理解RocketMQ的消息模型

首先：我们先来尝试往RocketMQ中发送一批消息。

在上一章提到，RocketMQ提供了一个测试脚本tools.sh，用于快速测试RocketMQ的客户端。

在服务器上配置了一个NAMESRV\_ADDR环境变量后，就可以直接使用RocketMQ提供的tools.sh脚本，调用RocketMQ提供的Producer示例。

```
tools.sh org.apache.rocketmq.example.quickstart.Producer
```

这里调用的Producer示例实际上是在RocketMQ安装目录下的lib/rocketmq-example-4.9.5.jar中包含的一个测试类。tools.sh脚本则是提供Producer类的运行环境。

Producer这个测试类，会往RocketMQ中发送一千条测试消息。发送消息后，我们可以在控制台看到很多如下的日志信息。

```
SendResult [sendStatus=SEND_OK, msgId=7F000001426E28A418FC6545DFD803E7,
offsetMsgId=C0A8E88100002A9F0000000000B4F6E5, messageQueue=MessageQueue
[topic=TopicTest, brokerName=broker-a, queueId=2], queueOffset=124]
```

这是RocketMQ的Broker服务端给消息生产者的响应。这个响应信息代表的是Broker服务端已经正常接收并保存了消息生产者发送的消息。这里面提到了很多topic、messageQueue等概念，这些是什么意思呢？我们不妨先去RocketMQ的DashBoard控制台看一下RocketMQ的Broker是如何保存这些消息的。

访问DashBoard上的“主题”菜单，可以看到多了一个名为TopicTest的主题。

RocketMQ仪表盘

运维

驾驶舱

集群

主题

消费者

生产者

消息

死信消息

消息轨迹

主题:

☒ 普通 ☐ 重试 ☐ 死信 ☐ 系统

新增/更新

刷新

主题	操作
TopicTest	<div><div>状态</div><div>路由</div><div>CONSUMER 管理</div><div>TOPIC 配置</div><div>发送消息</div><div>重置消费位点</div><div>跳过堆积</div><div>删除</div></div>

«

1

»

这个TopicTest就是我们之前运行的Producer创建的主题。点击“状态”按钮，可以看到TopicTest上的消息分布。

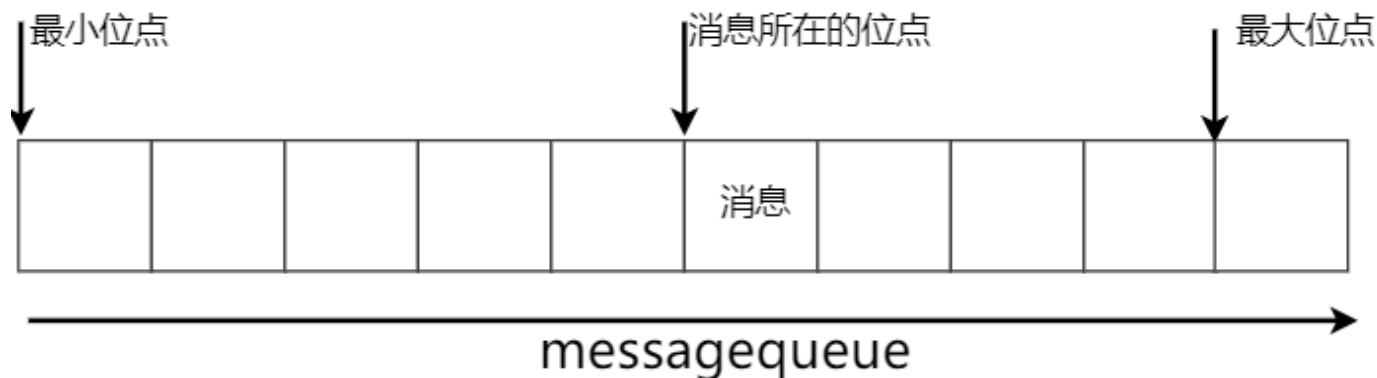
[TopicTest]状态

队列	最小位点	最大位点	上次更新时间
MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=3]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-b, queueId=3]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=2]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-b, queueId=0]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-b, queueId=2]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=1]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-b, queueId=1]	0	125	
MessageQueue [topic=TopicTest, brokerName=broker-a, queueId=0]	0	125	

从这里可以看到，TopicTest这个话题下，分配了八个MessageQueue。这里的MessageQueue就是一个典型的具有FIFO（先进先出）特性的消息集合。这八个MessageQueue均匀的分布在了集群中的两个Broker服务上。每个MesasgeQueue都记录了一个最小位点和最大位点。这里的位点代表每个MessageQueue上存储的消息的索引，也称为offset(偏移量)。每一条新记录的消息，都按照当前最大位点往后分配一个新的位点。这个位点就记录了这一条消息的存储位置。

从Dashboard就能看到，每一个MessageQueue，当前都记录了125条消息。也就是说，我们之前使用Producer示例往RocketMQ中发送的一千条消息，就被均匀的分配到了这八个MessageQueue上。

这是，再回头来看之前日志中打印的SendResult的信息。日志中的MessageQueue就代表这一条消息存在哪个队列上了。而queueOffset就表示这条消息记录在MessageQueue的哪个位置。



然后：我们尝试启动一个消费者来消费消息

我们同样可以使用tools.sh来启动一个消费者示例。

```
tools.sh org.apache.rocketmq.example.quickstart.Consumer
```

这个Consumer同样是RocketMQ下的lib/rocketmq-example-4.9.5.jar中提供的消费者示例。Consumer启动完成后，我们可以在控制台看到很多类似这样的日志：

```
ConsumeMessageThread_3 Receive New Messages: [MessageExt [brokerName=broker-b,
queueId=0, storeSize=194, queueOffset=95, sysFlag=0, bornTimestamp=1666252677571,
bornHost=/192.168.232.128:38414, storeTimestamp=1666252678510,
storeHost=/192.168.232.130:10911, msgId=C0A8E88200002A9F0000000000B4ADD2,
commitLogOffset=11840978, bodyCRC=634652396, reconsumeTimes=0,
preparedTransactionOffset=0, toString()=Message{topic='TopicTest', flag=0, properties=
{MIN_OFFSET=0, MAX_OFFSET=125, CONSUME_START_TIME=1666257428525,
UNIQ_KEY=7F000001426E28A418FC6545DDC302F9, CLUSTER=rocketmq-cluster, TAGS=TagA}, body=
[72, 101, 108, 108, 111, 32, 82, 111, 99, 107, 101, 116, 77, 81, 32, 55, 54, 49],
transactionId='null'}]]
```

这里面也打印出了一些我们刚刚熟悉的brokerName, queueId, queueOffset这些属性。其中queueOffset属性就表示这一条消息在MessageQueue上的存储位点。通过记录每一个消息的Offset偏移量，RocketMQ就可以快速的定位到这一条消息具体的存储位置，继而正确读取到消息的内容。

接下来，我们还是可以到DashBoard上印证一下消息消费的情况。

在DashBoard的“主题”页面，选择对应主题后的“CONSUMER管理”功能，就能看到消费者的消费情况。

TopicTest订阅组

订阅组	please_rename_unique_group_name_4	延迟	0	最后消费时间	
-----	-----------------------------------	----	---	--------	--

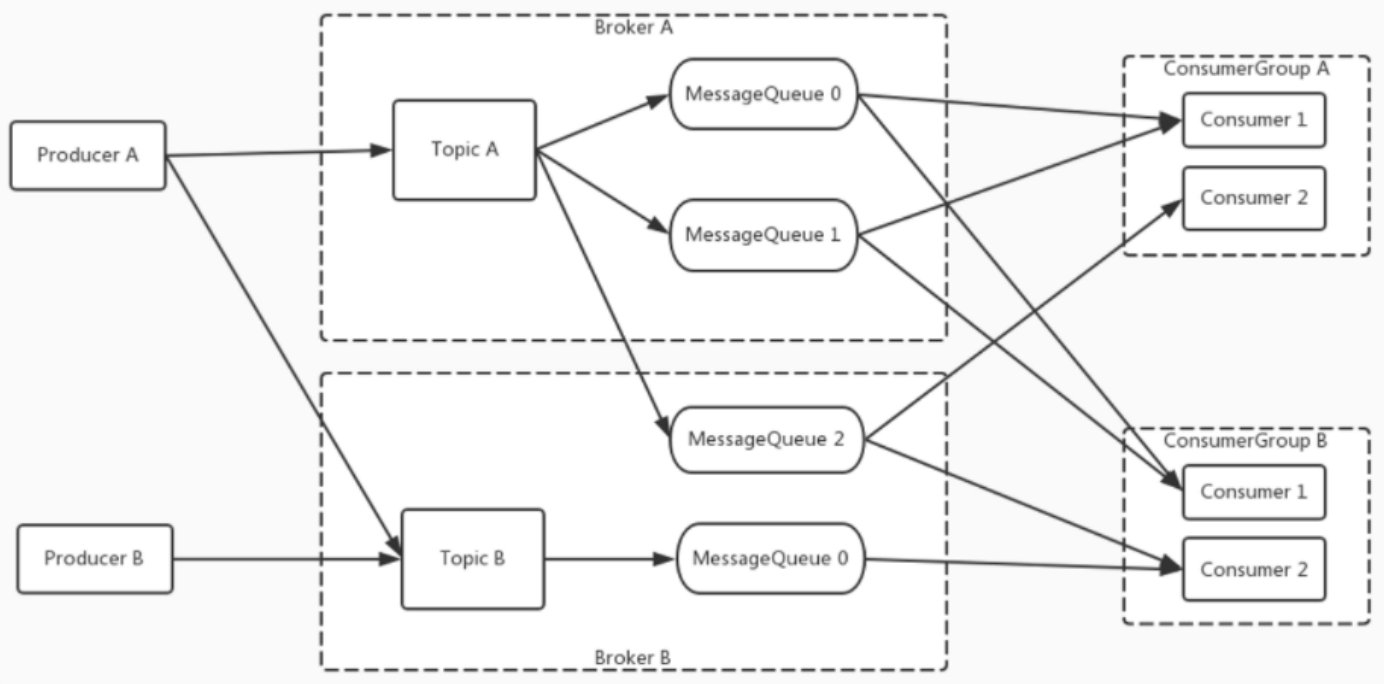
Broker	队列	消费者终端	代理者位点	消费者位点	差值	上次时间
broker-a	0	192.168.232.128@36807#8337054516585	125	125	0	
broker-a	1	192.168.232.128@36807#8337054516585	125	125	0	
broker-a	2	192.168.232.128@36807#8337054516585	125	125	0	
broker-a	3	192.168.232.128@36807#8337054516585	125	125	0	
broker-b	0	192.168.232.128@36807#8337054516585	125	125	0	
broker-b	1	192.168.232.128@36807#8337054516585	125	125	0	
broker-b	2	192.168.232.128@36807#8337054516585	125	125	0	
broker-b	3	192.168.232.128@36807#8337054516585	125	125	0	

从这里可以看到，刚才的Comsumer示例启动了一个叫做please\_rename\_unique\_group\_name\_4的消费者组。然后这个消费者从八个队列中都消费了数据。后面的代理者位点记录的是当前MessageQueue上记录的最大消息偏移量。而消费者位点记录的是当前消费者组在MessageQueue上消费的最大消息偏移量。其中的差值就表示当前消费者组没有处理完的消息。

并且，从这里还可以看出，RocketMQ记录消费者的消费进度时，都是以“订阅组”为单位的。我们也可以使用上一章节的示例，自己另外定义一个新的消费者组来消费TopicTest上的消息。这时，RocketMQ就会单独记录新消费者组的消费进度。而新的消费者组，也能消费到TopicTest下的所有消息。

接下来：我们就可以梳理出RocketMQ的消息记录方式

对之前的实验过程进行梳理，我们就能抽象出RocketMQ的消息模型。如下图所示：



生产者和消费者都可以指定一个Topic发送消息或者拉取消息。而Topic是一个逻辑概念。Topic中的消息会分布在后面多个MessageQueue当中。这些MessageQueue会分布到一个或者多个broker中。

在RocketMQ的这个消息模型当中，最为核心的就是Topic。对于客户端，Topic代表了一类有相同业务规则的消息。对于Broker，Topic则代表了系统中一系列存储消息的资源。所以，RocketMQ对于Topic是需要做严格管理的。如果任由客户端随意创建Topic，那么服务端的资源管理压力就会非常大。默认情况下，Topic都需要由管理员在RocketMQ的服务端手动进行创建，然后才能给客户端使用的。而我们之前在broker.conf中手动添加的autoCreateTopic=true，就是表示可以由客户端自行创建Topic。这种配置方式显然只适用于测试环境，在生产环境不建议打开这个配置项。如果需要创建Topic，可以交由运维人员提前创建Topic。

而对于业务来说，最为重要的就是消息Message了。生产者发送到某一个Topic下的消息，最终会保存在Topic下的某一个MessageQueue中。而消费者来消费消息时，RocketMQ会在Broker端给每个消费者组记录一个消息的消费位点Offset。通过Offset控制每个消费者组的消息处理进度。这样，每一条消息，在一个消费者组当中只被处理一次。

从逻辑层面来看，RocketMQ的消息模型和Kafka的消息模型是很相似的。没错，早期RocketMQ就是借鉴Kafka设计出来的。但是，在后续的发展过程中，RocketMQ在Kafka的基础上，做了非常大的调整。所以，对于RocketMQ，你也不妨回顾下Kafka，与Kafka对比着进行学习。

例如，在Kafka当中，如果Topic过多，会造成消息吞吐量下降。但是在RocketMQ中，对Topic的支持已经得到很大的加强。Topic过多几乎不会影响整体性能。RocketMQ是怎么设计的？另外，之前Kafka课程中也分析过，Leader选举的过程中，Kafka优先保证服务可用性，而一定程度上牺牲了消息的安全性，那么RocketMQ是怎么做的呢？保留这些问题，后续我们一一解决。

## 六、章节总结

这一章节，主要是快速熟悉RocketMQ产品，并通过操作，理解总结RocketMQ的运行架构以及消息模型。这些抽象的模型和架构，实际上就体现了MQ产品最为核心的设计思想。如果可以的话，最好对比之间介绍过的RabbitMQ和Kafka，将这几个产品进行横向对比，这样就能更好地理解设计的精髓。

当然，这只是一个基础的消息模型。在面对具体业务时，RocketMQ在这个消息模型的基础上，进行了大量的业务封装。下一章节就会着重了解RocketMQ针对各种业务场景设计的消息功能。

有道云笔记链接：<https://note.youdao.com/s/ToneNeza>