**RocketMQ探讨**

**武汉研发中心-郭凯**

目录

[第一章 MQ介绍](#_Toc8689_WPSOffice_Level1) [4](#_Toc8689_WPSOffice_Level1)

[1.1 什么是MQ](#_Toc17653_WPSOffice_Level2) [4](#_Toc17653_WPSOffice_Level2)

[1.2 MQ使用场景](#_Toc21804_WPSOffice_Level2) [4](#_Toc21804_WPSOffice_Level2)

[1）应用解耦](#_Toc17653_WPSOffice_Level3) [4](#_Toc17653_WPSOffice_Level3)

[2）流量削锋](#_Toc21804_WPSOffice_Level3) [4](#_Toc21804_WPSOffice_Level3)

[4） 消息订阅发布](#_Toc3252_WPSOffice_Level3) [5](#_Toc3252_WPSOffice_Level3)

[5）保证分布式事物最终一致性](#_Toc29318_WPSOffice_Level3) [5](#_Toc29318_WPSOffice_Level3)

[1.3 一句话区分何时使用MQ：](#_Toc3252_WPSOffice_Level2) [5](#_Toc3252_WPSOffice_Level2)

[1）使用MQ的场景：](#_Toc3328_WPSOffice_Level3) [5](#_Toc3328_WPSOffice_Level3)

[2）不能使用MQ的场景：](#_Toc17969_WPSOffice_Level3) [5](#_Toc17969_WPSOffice_Level3)

[1.4 MQ的缺点：](#_Toc29318_WPSOffice_Level2) [5](#_Toc29318_WPSOffice_Level2)

[1） 系统更复杂，引入了MQ组件](#_Toc31601_WPSOffice_Level3) [5](#_Toc31601_WPSOffice_Level3)

[2） 消息传递路径更长，延时增加](#_Toc2345_WPSOffice_Level3) [5](#_Toc2345_WPSOffice_Level3)

[3） 消息可靠性和重复性互为矛盾，消息不重不丢难以同时保证](#_Toc10190_WPSOffice_Level3) [5](#_Toc10190_WPSOffice_Level3)

[4） 上游无法知道下游的执行结果，这一点是很致命的](#_Toc17383_WPSOffice_Level3) [5](#_Toc17383_WPSOffice_Level3)

[第二章 什么是RocketMQ](#_Toc17653_WPSOffice_Level1) [5](#_Toc17653_WPSOffice_Level1)

[2.1 官方简介](#_Toc3328_WPSOffice_Level2) [5](#_Toc3328_WPSOffice_Level2)

[2.2 工作流程（类比dubbo架构来看）](#_Toc17969_WPSOffice_Level2) [6](#_Toc17969_WPSOffice_Level2)

[2.2 四大模块功能特性](#_Toc31601_WPSOffice_Level2) [7](#_Toc31601_WPSOffice_Level2)

[1.  nameserver](#_Toc22834_WPSOffice_Level3) [7](#_Toc22834_WPSOffice_Level3)

[2. broker](#_Toc10111_WPSOffice_Level3) [7](#_Toc10111_WPSOffice_Level3)

[3. 消费者](#_Toc28988_WPSOffice_Level3) [9](#_Toc28988_WPSOffice_Level3)

[4. 生产者](#_Toc6277_WPSOffice_Level3) [10](#_Toc6277_WPSOffice_Level3)

[第三章 RocketMQ设计](#_Toc21804_WPSOffice_Level1) [11](#_Toc21804_WPSOffice_Level1)

[3.1消息存储和发送（零拷贝技术）](#_Toc2345_WPSOffice_Level2) [11](#_Toc2345_WPSOffice_Level2)

[3.2消息存储结构（数据和索引分离）](#_Toc10190_WPSOffice_Level2) [12](#_Toc10190_WPSOffice_Level2)

[3.3高可用机制](#_Toc17383_WPSOffice_Level2) [13](#_Toc17383_WPSOffice_Level2)

[3.4同步刷盘与异步刷盘（When-flushDisk）](#_Toc22834_WPSOffice_Level2) [13](#_Toc22834_WPSOffice_Level2)

[3.5同步复制与异步复制（M-S复制）](#_Toc10111_WPSOffice_Level2) [14](#_Toc10111_WPSOffice_Level2)

[3.6 消费模式](#_Toc28988_WPSOffice_Level2) [14](#_Toc28988_WPSOffice_Level2)

[3.6.1 集群消费模式（负载均衡机制）](#_Toc28126_WPSOffice_Level3) [15](#_Toc28126_WPSOffice_Level3)

[3.6.2 广播消费模式（发布订阅机制）](#_Toc16934_WPSOffice_Level3) [16](#_Toc16934_WPSOffice_Level3)

[3.7 顺序消息（生产者 - messagequeue - 消费者）](#_Toc6277_WPSOffice_Level2) [17](#_Toc6277_WPSOffice_Level2)

[3.8 事物消息（小事务+异步）](#_Toc28126_WPSOffice_Level2) [18](#_Toc28126_WPSOffice_Level2)

[3.9 消息去重](#_Toc16934_WPSOffice_Level2) [19](#_Toc16934_WPSOffice_Level2)

[3.10 消息重试](#_Toc26247_WPSOffice_Level2) [19](#_Toc26247_WPSOffice_Level2)

[3.10.1 概念](#_Toc26247_WPSOffice_Level3) [19](#_Toc26247_WPSOffice_Level3)

[3.10.2 Broker消息重试](#_Toc31524_WPSOffice_Level3) [20](#_Toc31524_WPSOffice_Level3)

[3.10.3 Producer端消息重试实现](#_Toc18366_WPSOffice_Level3) [20](#_Toc18366_WPSOffice_Level3)

[3.10.4 Consumer端消息重试实现](#_Toc21393_WPSOffice_Level3) [20](#_Toc21393_WPSOffice_Level3)

[3.11 消息的推送与拉取](#_Toc31524_WPSOffice_Level2) [22](#_Toc31524_WPSOffice_Level2)

[3.11.1 PushConsumer](#_Toc26722_WPSOffice_Level3) [22](#_Toc26722_WPSOffice_Level3)

[3.11.2 PullConsumer](#_Toc6526_WPSOffice_Level3) [22](#_Toc6526_WPSOffice_Level3)

[第四章 常见消息中间对比](#_Toc3252_WPSOffice_Level1) [23](#_Toc3252_WPSOffice_Level1)

[第五章 部署RocketMQ](#_Toc29318_WPSOffice_Level1) [24](#_Toc29318_WPSOffice_Level1)

[6.1 单台服务器部署](#_Toc18366_WPSOffice_Level2) [24](#_Toc18366_WPSOffice_Level2)

[6.1.1 下载源码](#_Toc11826_WPSOffice_Level3) [24](#_Toc11826_WPSOffice_Level3)

[6.1.2 构建](#_Toc21491_WPSOffice_Level3) [24](#_Toc21491_WPSOffice_Level3)

[6.1.3 启动名称服务器](#_Toc25414_WPSOffice_Level3) [24](#_Toc25414_WPSOffice_Level3)

[6.1.4 启动broker](#_Toc28405_WPSOffice_Level3) [24](#_Toc28405_WPSOffice_Level3)

[6.1.5 发送和接收消息](#_Toc11426_WPSOffice_Level3) [25](#_Toc11426_WPSOffice_Level3)

[6.1.6 关闭服务器](#_Toc26719_WPSOffice_Level3) [25](#_Toc26719_WPSOffice_Level3)

[6.2 集群服务器部署（第3种模式）](#_Toc21393_WPSOffice_Level2) [25](#_Toc21393_WPSOffice_Level2)

[1. 单个 Master](#_Toc21002_WPSOffice_Level3) [26](#_Toc21002_WPSOffice_Level3)

[2. 多 Master 模式](#_Toc3287_WPSOffice_Level3) [26](#_Toc3287_WPSOffice_Level3)

[3. 多 Master 多 Slave 模式，异步复制](#_Toc18872_WPSOffice_Level3) [26](#_Toc18872_WPSOffice_Level3)

[4. 多 Master 多 Slave 模式，同步双写](#_Toc3373_WPSOffice_Level3) [27](#_Toc3373_WPSOffice_Level3)

[第七章 管理控制台](#_Toc3328_WPSOffice_Level1) [29](#_Toc3328_WPSOffice_Level1)

[7.1 安装部署](#_Toc26722_WPSOffice_Level2) [29](#_Toc26722_WPSOffice_Level2)

[7.2 功能介绍](#_Toc6526_WPSOffice_Level2) [29](#_Toc6526_WPSOffice_Level2)

[第八章 Springcloud集成RocketMQ](#_Toc17969_WPSOffice_Level1) [30](#_Toc17969_WPSOffice_Level1)

[8.1 添加pom依赖](#_Toc11826_WPSOffice_Level2) [30](#_Toc11826_WPSOffice_Level2)

[8.2 配置文件application.yml](#_Toc21491_WPSOffice_Level2) [31](#_Toc21491_WPSOffice_Level2)

[8.3 消息生产者配置类](#_Toc25414_WPSOffice_Level2) [31](#_Toc25414_WPSOffice_Level2)

[8.4 消息生产者](#_Toc28405_WPSOffice_Level2) [32](#_Toc28405_WPSOffice_Level2)

[8.5 消息消费者配置类](#_Toc11426_WPSOffice_Level2) [33](#_Toc11426_WPSOffice_Level2)

[8.6 消息消费者](#_Toc26719_WPSOffice_Level2) [34](#_Toc26719_WPSOffice_Level2)

[8.7 具体业务类](#_Toc21002_WPSOffice_Level2) [35](#_Toc21002_WPSOffice_Level2)

[8.8 新建springboot启动类RocketMqApplication.java](#_Toc3287_WPSOffice_Level2) [36](#_Toc3287_WPSOffice_Level2)

[8.9 新建一个Controller，引入消息生产者](#_Toc18872_WPSOffice_Level2) [37](#_Toc18872_WPSOffice_Level2)

[8.1 验证服务](#_Toc3373_WPSOffice_Level2) [38](#_Toc3373_WPSOffice_Level2)

[8.2 更多DEMO，请参考官方例子](#_Toc20147_WPSOffice_Level2) [38](#_Toc20147_WPSOffice_Level2)

# MQ介绍

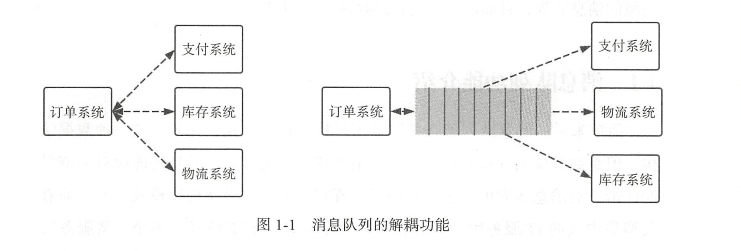
## 什么是MQ

消息队列（MessageQueue），是一种跨进程的通信机制，用于上下游传递消息。在互联网架构中，MQ是一种非常常见的上下游“逻辑解耦+物理解耦”的消息通信服务。使用MQ之后，消息发送上游只需要依赖MQ，逻辑上和物理上都不用依赖其他服务。

## MQ使用场景

1）应用解耦

以电商应用为例，用户创建订单后，如果调用库存系统、物流系统、支付系统，任何一个子系统出了故障或者因为升级等原因暂时不可用，都会造成下单操作异常，影响用户使用体验。



如图所示，当转变成基于消息队列的方式后，系统可用性就高多了，比如物流系统因为发生故障，需要几分钟的时间来修复，在这几分钟的时间里，物流系统要处理的内容被缓存在消息队列里，用户的下单操作可以正常完成。当物流系统恢复后，补充处理存储在消息队列里的订单信息即可，终端用户感知不到物流系统发生过几分钟的故障。

2）流量削锋

对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。高峰值流量是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术。

使用消息队列进行流量消峰，很多时候不是因为能力不够，而是出于经济性的考量。比如有的业务系统，流量最高峰也不会超过一万QPS，而平时只有一千左右的QPS。这种情况下我们就可以用个普通性能的服务器（只支持一千左右的QPS就可以），然后加个消息队列作为高峰期的缓冲，无须花大笔资金部署能处理上万QPS的服务器。

1. 消息订阅发布

数据的产生方只需要把各自的数据写人一个消息队列即可，数据使用方根据各自需求订阅感兴趣的数据，不同数据团队所订阅的数据可以重复也可以不重复，互不干扰，也不必和数据产生方关联。

1. 保证分布式事物最终一致性
2. 异步化

## 一句话区分何时使用MQ：

1）使用MQ的场景：

调用方非实时关注执行结果。

1. 数据驱动的任务依赖（比如授信系统，反欺诈通过后，再触发授信评分）
2. 上游不关注执行结果（比如用户信用卡还款后，奖励用户积分）
3. 异步执行时间较长

2）不能使用MQ的场景：

调用方实时关注执行结果。

比如登录过程中，调用鉴权服务，就是直接调用关系。

## MQ的缺点：

1. 系统更复杂，引入了MQ组件
2. 消息传递路径更长，延时增加
3. 消息可靠性和重复性互为矛盾，消息不重不丢难以同时保证
4. 上游无法知道下游的执行结果，这一点是很致命的

# 什么是RocketMQ

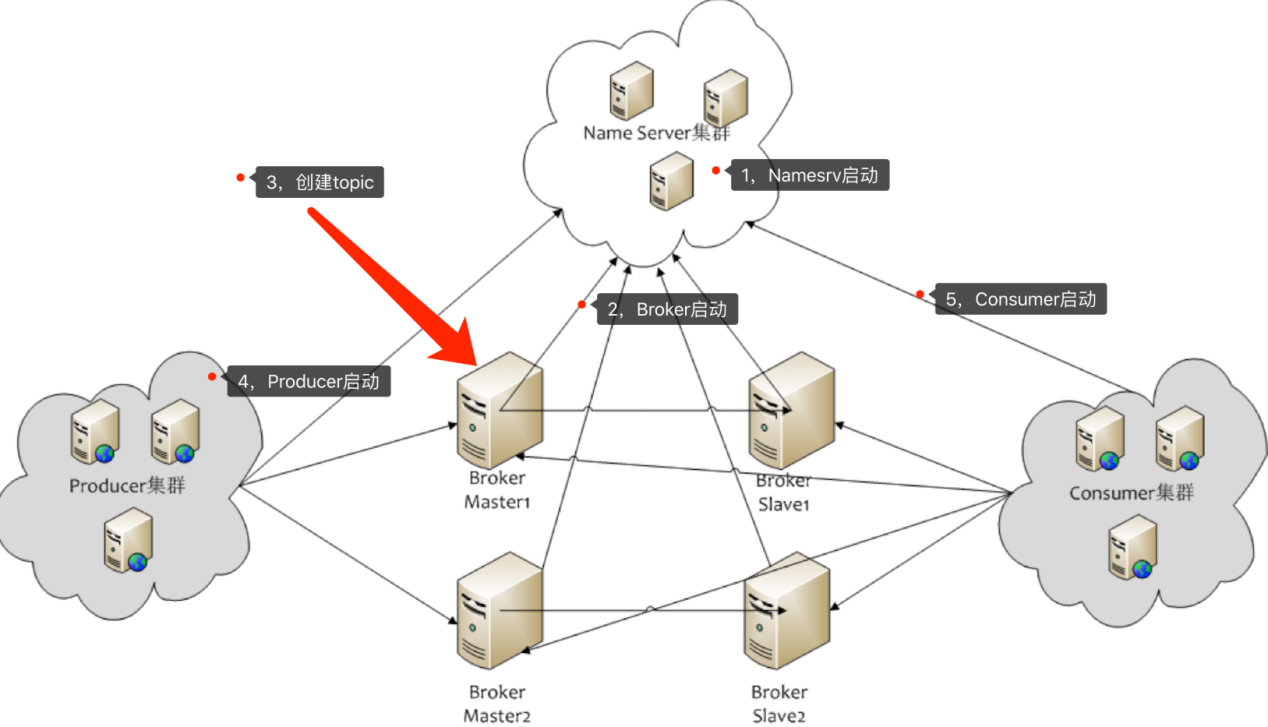
## 2.1 官方简介

RocketMQ是一个分布式开放消息中间件，底层基于队列模型来实现消息收发功能。

阿里的消息中间件有很长的历史，从2007年的Notify到2010年的Napoli,2011年升级后改为MetaQ，然后到2012年开始做RocketMQ,RocketMQ使用Java语言开发，于2016年开源。第一代的Notify主要使用了推模型，解决了事务消息；第二代的MetaQ主要使用了拉模型，解决了顺序消息和海量堆积的问题。RocketMQ基于长轮询的拉取方式，兼有两者的优点。

每一次产品迭代，都吸取了之前的经验教训，目前RocketMQ已经成为Apache顶级项目。在阿里内部，RocketMQ很好地服务了集团大大小小上千个应用，在每年的双十一当天，更有不可思议的万亿级消息通过RocketMQ流转（在2017年的双十一当天，整个阿里巴巴集团通过RocketMQ流转的线上消息达到了万亿级，峰值TPS达到5600万），在阿里大中台策略上发挥着举足轻重的作用。此外，RocketMQ是使用Java语言开发的，比起Kafka的Scala语言和RabbitMQ的Erlang语言，更容易找到技术人员进行定制开发。

## 2.2 工作流程（类比dubbo架构来看）



结合部署结构图，描述集群工作流程：

1，启动Namesrv，Namesrv起来后监听端口，等待Broker、Produer、Consumer连上来，相当于一个路由控制中心。

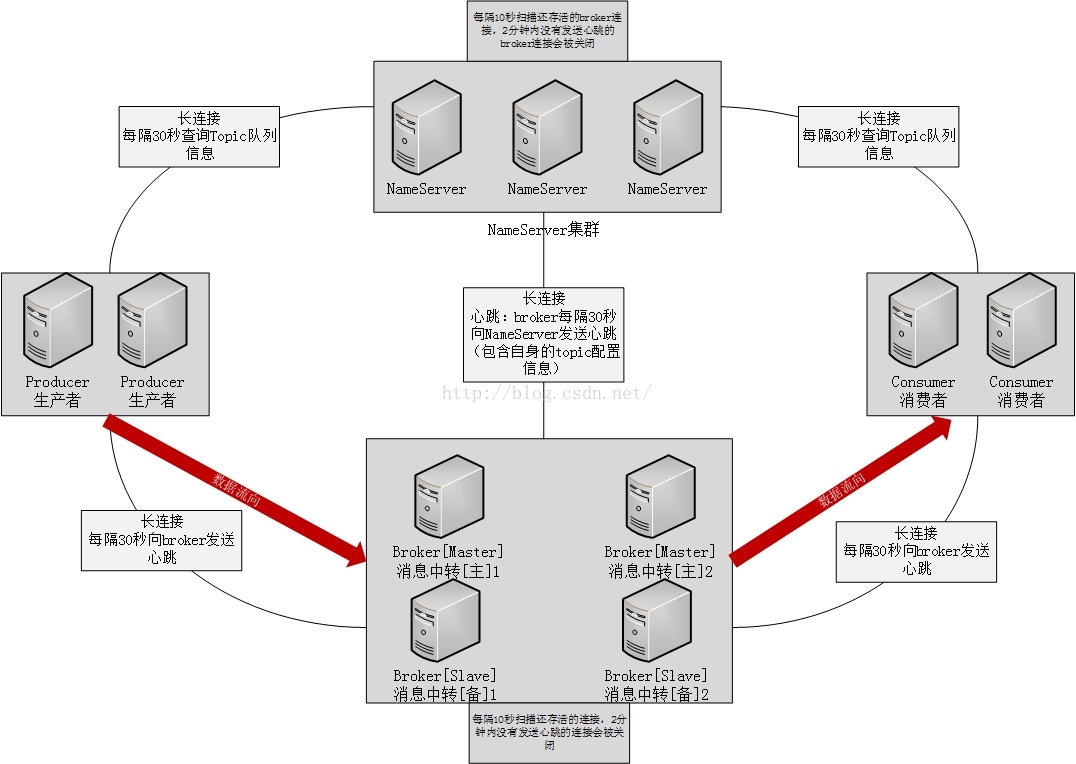
2，Broker启动，跟所有的Namesrv保持长连接，定时发送心跳包。心跳包中包含当前Broker信息(IP+端口等)以及存储所有topic信息。注册成功后，namesrv集群中就有Topic跟Broker的映射关系。

3，收发消息前，先创建topic，创建topic时需要指定该topic要存储在哪些Broker上。也可以在发送消息时自动创建Topic。

4，Producer发送消息，启动时先跟Namesrv集群中的其中一台建立长连接，并从Namesrv中获取当前发送的Topic存在哪些Broker上，然后跟对应的Broker建立长连接，直接向Broker发消息。

5，Consumer跟Producer类似。跟其中一台Namesrv建立长连接，获取当前订阅Topic存在哪些Broker上，然后直接跟Broker建立连接通道，开始消费消息。

## 2.2 四大模块功能特性



### **nameserver**

相对来说，nameserver的稳定性非常高。原因有二：

1 、nameserver互相独立，彼此没有通信关系，单台nameserver挂掉，不影响其他nameserver，即使全部挂掉，也不影响业务系统使用。

2 、nameserver不会有频繁的读写，所以性能开销非常小，稳定性很高。

### **broker**

#### **与nameserver关系**

l  连接

     单个broker和所有nameserver保持长连接

l  心跳

     心跳间隔：每隔30秒（此时间无法更改）向所有nameserver发送心跳，心跳包含了自身的topic配置信息。

     心跳超时：nameserver每隔10秒钟（此时间无法更改），扫描所有还存活的broker连接，若某个连接2分钟内（当前时间与最后更新时间差值超过2分钟，此时间无法更改）没有发送心跳数据，则断开连接。

l  断开

     时机：broker挂掉；心跳超时导致nameserver主动关闭连接

     动作：一旦连接断开，nameserver会立即感知，更新topc与队列的对应关系，但不会通知生产者和消费者

#### **负载均衡**

l  一个topic分布在多个broker上，一个broker可以配置多个topic，它们是多对多的关系。

l  如果某个topic消息量很大，应该给它多配置几个队列，并且尽量多分布在不同broker上，减轻某个broker的压力。

l  topic消息量都比较均匀的情况下，如果某个broker上的队列越多，则该broker压力越大。

#### **可用性**

   由于消息分布在各个broker上，一旦某个broker宕机，则该broker上的消息读写都会受到影响。所以rocketmq提供了master/slave的结构，salve定时从master同步数据，如果master宕机，则slave提供消费服务，但是不能写入消息，此过程对应用透明，由rocketmq内部解决。

这里有两个关键点：

l  一旦某个broker master宕机，生产者和消费者多久才能发现？受限于rocketmq的网络连接机制，默认情况下，最多需要30秒，但这个时间可由应用设定参数来缩短时间。这个时间段内，发往该broker的消息都是失败的，而且该broker的消息无法消费，因为此时消费者不知道该broker已经挂掉。

l  消费者得到master宕机通知后，转向slave消费，但是slave不能保证master的消息100%都同步过来了，因此会有少量的消息丢失。但是消息最终不会丢的，一旦master恢复，未同步过去的消息会被消费掉。

#### **可靠性**

l  所有发往broker的消息，有同步刷盘和异步刷盘机制，总的来说，可靠性非常高

l  同步刷盘时，消息写入物理文件才会返回成功，因此非常可靠

l  异步刷盘时，只有机器宕机，才会产生消息丢失，broker挂掉可能会发生，但是机器宕机崩溃是很少发生的，除非突然断电

#### **消息清理**

l  扫描间隔

     默认10秒，由broker配置参数cleanResourceInterval决定

l  空间阈值

     物理文件不能无限制的一直存储在磁盘，当磁盘空间达到阈值时，不再接受消息，broker打印出日志，消息发送失败，阈值为固定值85%

l  清理时机

     默认每天凌晨4点，由broker配置参数deleteWhen决定；或者磁盘空间达到阈值

l  文件保留时长

     默认72小时，由broker配置参数fileReservedTime决定

#### **读写性能**

l  文件内存映射方式操作文件，避免read/write系统调用和实时文件读写，性能非常高

l  永远一个文件在写，其他文件在读

l  顺序写，随机读

l  利用linux的sendfile机制，将消息内容直接输出到sokect管道，避免系统调用

### **消费者**

#### **与nameserver关系**

l  连接

     单个消费者和一台nameserver保持长连接，定时查询topic配置信息，如果该nameserver挂掉，消费者会自动连接下一个nameserver，直到有可用连接为止，并能自动重连。

l  心跳

与nameserver没有心跳

l  轮询时间

默认情况下，消费者每隔30秒从nameserver获取所有topic的最新队列情况，这意味着某个broker如果宕机，客户端最多要30秒才能感知。该时间由DefaultMQPushConsumer的pollNameServerInteval参数决定，可手动配置。

#### **与broker关系**

l  连接

单个消费者和该消费者关联的所有broker保持长连接。

l  心跳

默认情况下，消费者每隔30秒向所有broker发送心跳，该时间由DefaultMQPushConsumer的heartbeatBrokerInterval参数决定，可手动配置。broker每隔10秒钟（此时间无法更改），扫描所有还存活的连接，若某个连接2分钟内（当前时间与最后更新时间差值超过2分钟，此时间无法更改）没有发送心跳数据，则关闭连接，并向该消费者分组的所有消费者发出通知，分组内消费者重新分配队列继续消费

l  断开

时机：消费者挂掉；心跳超时导致broker主动关闭连接

动作：一旦连接断开，broker会立即感知到，并向该消费者分组的所有消费者发出通知，分组内消费者重新分配队列继续消费

#### **负载均衡**

集群消费模式下，一个消费者集群多台机器共同消费一个topic的多个队列，一个队列只会被一个消费者消费。如果某个消费者挂掉，分组内其它消费者会接替挂掉的消费者继续消费。

#### **消费机制**

l  本地队列

        消费者不间断的从broker拉取消息，消息拉取到本地队列，然后本地消费线程消费本地消息队列，只是一个异步过程，拉取线程不会等待本地消费线程，这种模式实时性非常高。对消费者对本地队列有一个保护，因此本地消息队列不能无限大，否则可能会占用大量内存，本地队列大小由DefaultMQPushConsumer的pullThresholdForQueue属性控制，默认1000，可手动设置。

l  轮询间隔

     消息拉取线程每隔多久拉取一次？间隔时间由DefaultMQPushConsumer的pullInterval属性控制，默认为0，可手动设置。

l  消息消费数量

     监听器每次接受本地队列的消息是多少条？这个参数由DefaultMQPushConsumer的consumeMessageBatchMaxSize属性控制，默认为1，可手动设置。

#### **消费进度存储**

     每隔一段时间将各个队列的消费进度存储到对应的broker上，该时间由DefaultMQPushConsumer的persistConsumerOffsetInterval属性控制，默认为5秒，可手动设置。

#### **如果一个topic在某broker上有3个队列，一个消费者消费这3个队列，那么该消费者和这个broker有几个连接？**

     一个连接，消费单位与队列相关，消费连接只跟broker相关，事实上，消费者将所有队列的消息拉取任务放到本地的队列，挨个拉取，拉取完毕后，又将拉取任务放到队尾，然后执行下一个拉取任务

### **4. 生产者**

#### **与nameserver关系**

l  连接

     单个生产者者和一台nameserver保持长连接，定时查询topic配置信息，如果该nameserver挂掉，生产者会自动连接下一个nameserver，直到有可用连接为止，并能自动重连。

l  轮询时间

默认情况下，生产者每隔30秒从nameserver获取所有topic的最新队列情况，这意味着某个broker如果宕机，生产者最多要30秒才能感知，在此期间，发往该broker的消息发送失败。该时间由DefaultMQProducer的pollNameServerInteval参数决定，可手动配置。

l  心跳

与nameserver没有心跳

#### **与broker关系**

l  连接

单个生产者和该生产者关联的所有broker保持长连接。

l  心跳

默认情况下，生产者每隔30秒向所有broker发送心跳，该时间由DefaultMQProducer的heartbeatBrokerInterval参数决定，可手动配置。broker每隔10秒钟（此时间无法更改），扫描所有还存活的连接，若某个连接2分钟内（当前时间与最后更新时间差值超过2分钟，此时间无法更改）没有发送心跳数据，则关闭连接。

l  连接断开

移除broker上的生产者信息

#### **负载均衡**

     生产者时间没有关系，每个生产者向队列轮流发送消息

这里需要注意一点：

假如某个Broker宕机，意味生产者最长需要30秒才能感知到。在这期间会向宕机的Broker发送消息。当一条消息发送到某个Broker失败后，会往该broker自动再重发2次，假如还是发送失败，则抛出发送失败异常。业务捕获异常，重新发送即可。客户端里会自动轮询另外一个Broker重新发送，这个对于用户是透明的。

# RocketMQ设计

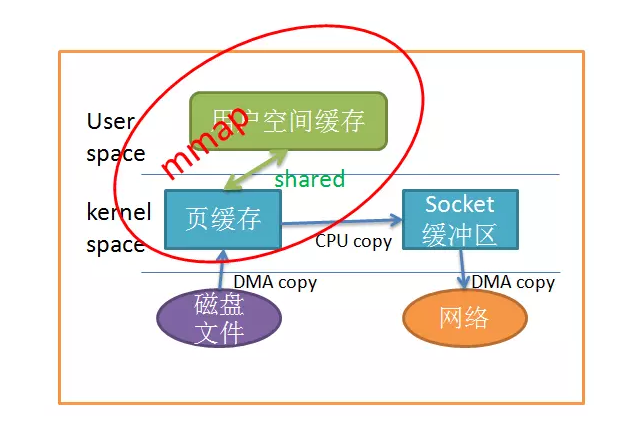
## 3.1消息存储和发送（零拷贝技术）

分布式队列因为有高可靠性的要求，所以数据要通过磁盘进行持久化存储。

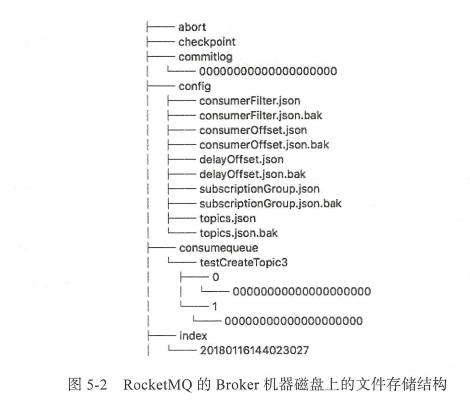
基于“零拷贝”技术，提高消息存盘和网络发送的速度。

我们减少拷贝次数的一种方法是调用mmap()来代替read调用：buf = mmap(diskfd, len);write(sockfd, buf, len);

应用程序调用mmap()，磁盘上的数据会通过DMA被拷贝的内核缓冲区，接着操作系统会把这段内核缓冲区与应用程序共享，这样就不需要把内核缓冲区的内容往用户空间拷贝。应用程序再调用write(),操作系统直接将内核缓冲区的内容拷贝到socket缓冲区中，这一切都发生在内核态，最后，socket缓冲区再把数据发到网卡去。同样的，看图很简单：



## 3.2消息存储结构（数据和索引分离）



RocketMQ消息的存储是由ConsumeQueue和CommitLog配合完成的，消息真正的物理存储文件是CommitLog,ConsumeQueue是消息的逻辑队列，类似数据库的索引文件，存储的是指向物理存储的地址。

存储机制这样设计有以下几个好处：

1 ) CommitLog顺序写，可以大大提高写人效率。

2）虽然是随机读，但是利用操作系统的pagecache机制，可以批量地从磁盘读取，作为cache存到内存中，加速后续的读取速度。

## 3.3高可用机制

RocketMQ分布式集群是通过Master和Slave的配合达到高可用性的，首先说一下Master和Slave的区别：在Broker的配置文件中，参数brokerId

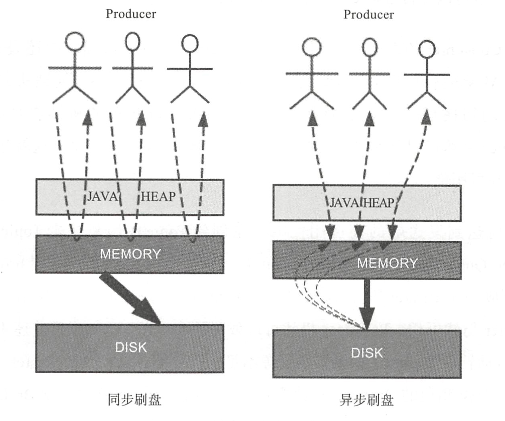
仅限非商业用途或学习研究使用第5章消息队列的核心机制令61的值为0表明这个Broker是Master，大于0表明这个Broker是Slave，同时brokerRole参数也会说明这个Broker是Master还是Slave。Master角色的Broker支持读和写，Slave角色的Broker仅支持读，也就是Producer只能和Master角色的Broker连接写人消息；Consumer可以连接Master角色的Broker，也可以连接Slave角色的Broker来读取消息。

## 3.4同步刷盘与异步刷盘（When-flushDisk）

RocketMQ的消息是存储到磁盘上的，这样既能保证断电后恢复，又可以让存储的消息量超出内存的限制。RocketMQ为了提高性能，会尽可能地保证磁盘的顺序写。消息在通过Producer写人RocketMQ的时候，有两种写磁盘方式，下面逐一介绍。

异步刷盘方式：在返回写成功状态时，消息可能只是被写人了内存的PAGECACHE，写操作的返回快，吞吐量大；当内存里的消息量积累到一定程度时，统一触发写磁盘动作，快速写人。

同步刷盘方式：在返回写成功状态时，消息已经被写人磁盘。具体流程是，消息、写入内存的PAGECACHE后，立刻通知刷盘线程刷盘，然后等待刷盘完成，刷盘线程执行完成后唤醒等待的线程，返回消息写成功的状态。



同步刷盘还是异步刷盘，是通过Broker配置文件里的flushDiskType参数设置的，这个参数被配置成SYNCFLUSH、ASYNCFLUSH中的一个。

## 3.5同步复制与异步复制（M-S复制）

如果一个Broker组有Master和Slave,消息需要从Master复制到Slave上，有同步和异步两种复制方式。同步复制方式是等Master和Slave均写成功后才反馈给客户端写成功状态；异步复制方式是只要Master写成功即可反馈给客户端写成功状态。

这两种复制方式各有优劣，在异步复制方式下，系统拥有较低的延迟和较高的吞吐量，但是如果Master出了故障，有些数据因为没有被写人Slave，有可能会丢失；在同步复制方式下，如果Master出故障，Slave上有全部的备份数据，容易恢复，但是同步复制会增大数据写人延迟，降低系统吞吐量。

## 3.6 消费模式

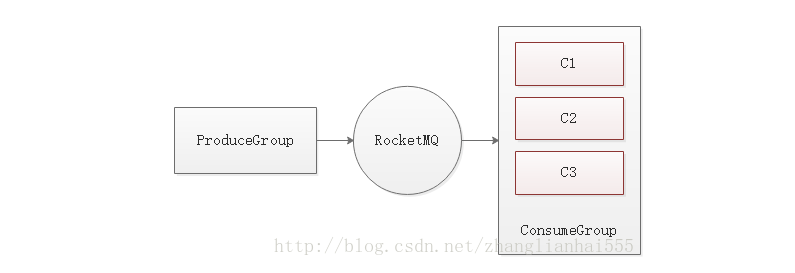
在前面学习ActiveMQ时，看到ActiveMQ可以是队列消息模式，也可以是订阅发布模式。

同样，在RocketMQ中，也存在两种消息模式，即是集群消费模式和广播消费模式。

### 3.6.1 集群消费模式（负载均衡机制）

跟AciiveMQ一样，当存在多个消费者时，消息通过一定负载均衡策略，将消息分发到多个consumer中。

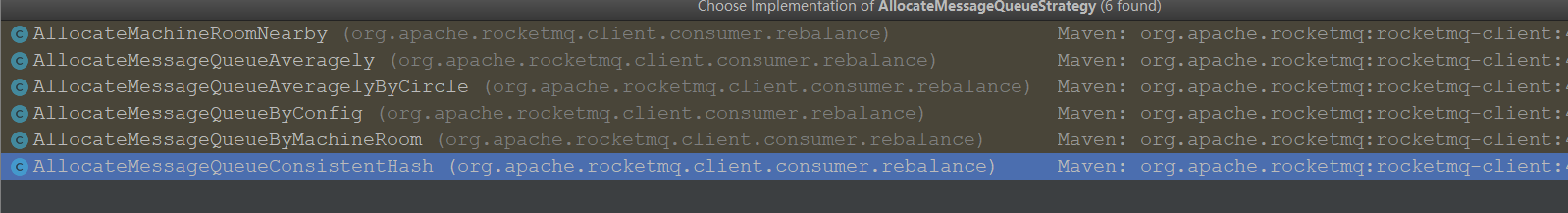
如图：



在RockeMQ中，通过ConsumeGroup的机制，实现了天然的消息负载均衡！通俗点来说，RocketMQ中的消息通过ConsumeGroup实现了将消息分发到C1/C2/C3/……的机制，这意味着我们将非常方便的通过加机器来实现水平扩展！

消息分发也是有多个策略可以配置的，配置方式如下：

可以使用setAllocateMessageQueueStrategy（）方法传入如下参数实现不同的负载均衡策略，默认AllocateMessageQueueAveragely，轮询算法策略。



RocketMQ默认的消息模式就是集群模式。

开启两个不同的consumer，控制台打印结果如下：

consumer1  
收到来自topic: MyTopic,的消息：2  
收到来自topic: MyTopic,的消息：3  
收到来自topic: MyTopic,的消息：4  
收到来自topic: MyTopic,的消息：5  
consumer2：  
收到来自topic: MyTopic,的消息：0  
收到来自topic: MyTopic,的消息：1  
收到来自topic: MyTopic,的消息：6  
收到来自topic: MyTopic,的消息：7  
收到来自topic: MyTopic,的消息：8  
收到来自topic: MyTopic,的消息：9

可以看出消息是被分发给两个消费者的，可以通过consumer.setMessageModel(MessageModel.CLUSTERING);设置集群消费策略。

### 3.6.2 广播消费模式（发布订阅机制）

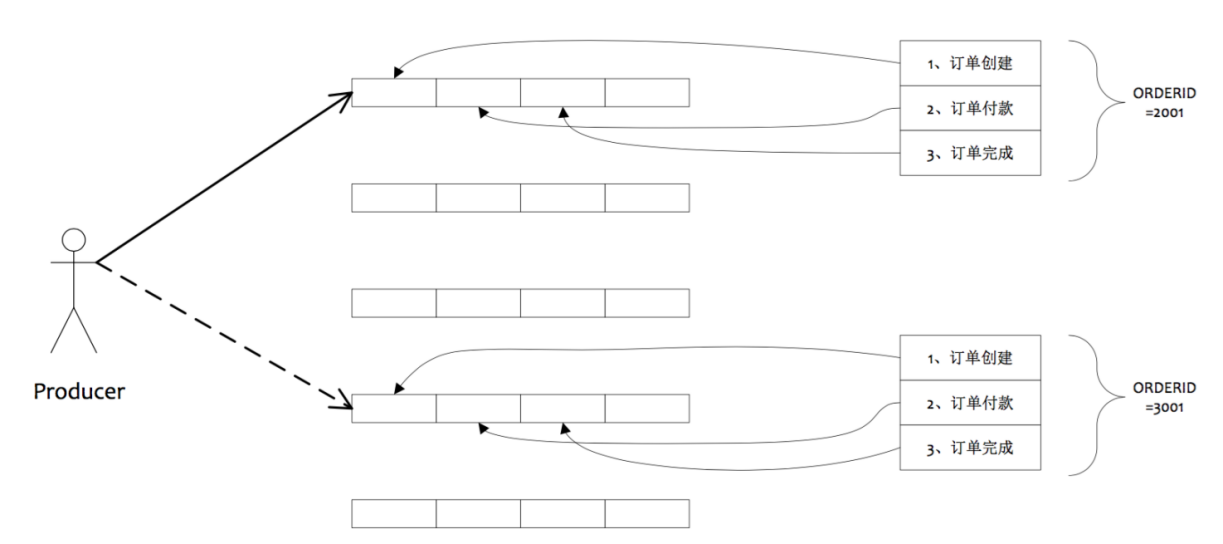
广播模式跟ActiveMQ的发布订阅一样，即是将所有消息分发给Consume Group中每个消费者消费。

public class Consumer {  
 public static void main(String[] args) throws MQClientException {  
 final DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("MyConsumerGroup");  
 consumer.setNamesrvAddr("10.104.112.222:9876");  
 consumer.setConsumeFromWhere(ConsumeFromWhere.*CONSUME\_FROM\_LAST\_OFFSET*);  
 consumer.subscribe("MyTopic", "\*");  
 //设置消费模式为广播模式  
 consumer.setMessageModel(MessageModel.*BROADCASTING*);  
 consumer.registerMessageListener(new MessageListenerConcurrently() {  
 @Override  
 public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  
 try {  
 MessageExt ext = msgs.get(0);  
 String topic = ext.getTopic();  
 String body = new String(ext.getBody(),"utf-8");  
 System.*out*.println("收到来自topic: " + topic + ",的消息：" + body);  
 } catch (Exception e) {  
 return ConsumeConcurrentlyStatus.*RECONSUME\_LATER*;  
 }  
 return ConsumeConcurrentlyStatus.*CONSUME\_SUCCESS*;  
 }  
 });  
 consumer.start();  
 }  
}

启动两个消费者，每个消费者都能接收到所有消息，控制台打印如下：

收到来自topic: MyTopic,的消息：0  
收到来自topic: MyTopic,的消息：1  
收到来自topic: MyTopic,的消息：2  
收到来自topic: MyTopic,的消息：3  
收到来自topic: MyTopic,的消息：4  
收到来自topic: MyTopic,的消息：5  
收到来自topic: MyTopic,的消息：6  
收到来自topic: MyTopic,的消息：7  
收到来自topic: MyTopic,的消息：8  
收到来自topic: MyTopic,的消息：9

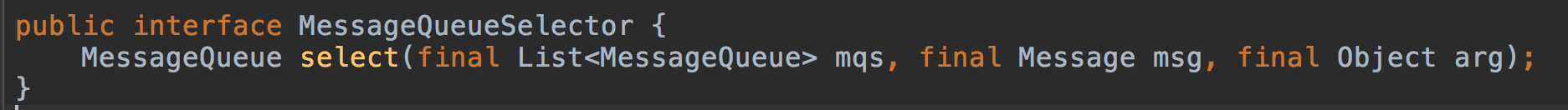
## 3.7 顺序消息（生产者 - messagequeue - 消费者）



上图是RocketMQ顺序消息原理的介绍，将不同订单的消息路由到不同的分区中。文档只是给出了Producer顺序的处理，Consumer消费时通过一个分区只能有一个线程消费的方式来保证消息顺序，具体实现如下。

Producer端

Producer端确保消息顺序唯一要做的事情就是将消息路由到特定的分区，在RocketMQ中，通过MessageQueueSelector来实现分区的选择。



List<MessageQueue> mqs：消息要发送的Topic下所有的分区

Message msg：消息对象

额外的参数：用户可以传递自己的参数

比如如下实现就可以保证相同的订单的消息被路由到相同的分区：

long orderId = ((Order) arg).getOrderId();

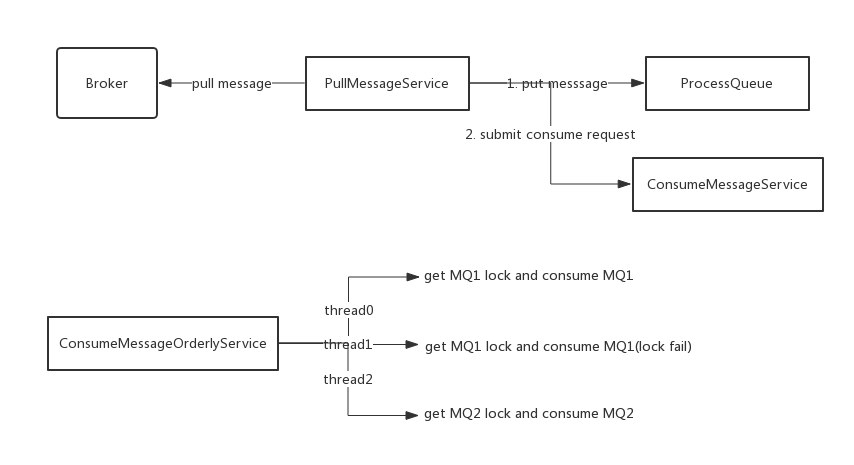
return mqs.get(orderId % mqs.size());

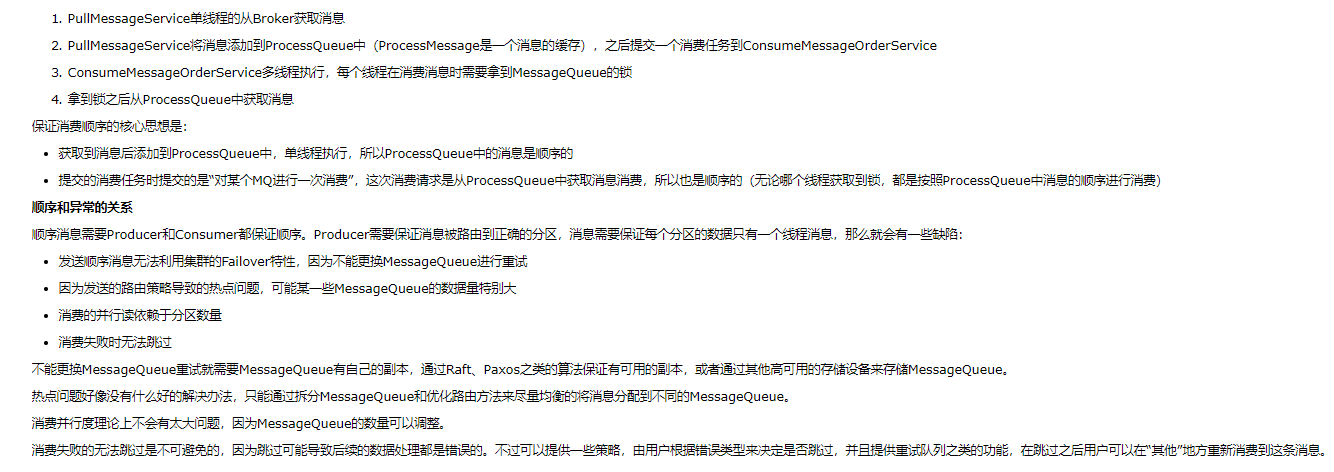
Consumer端

RocketMQ消费端有两种类型：MQPullConsumer和MQPushConsumer。

MQPullConsumer由用户控制线程，主动从服务端获取消息，每次获取到的是一个MessageQueue中的消息。PullResult中的List msgFoundList自然和存储顺序一致，用户需要再拿到这批消息后自己保证消费的顺序。

对于PushConsumer，由用户注册MessageListener来消费消息，在客户端中需要保证调用MessageListener时消息的顺序性。RocketMQ中的实现如下：



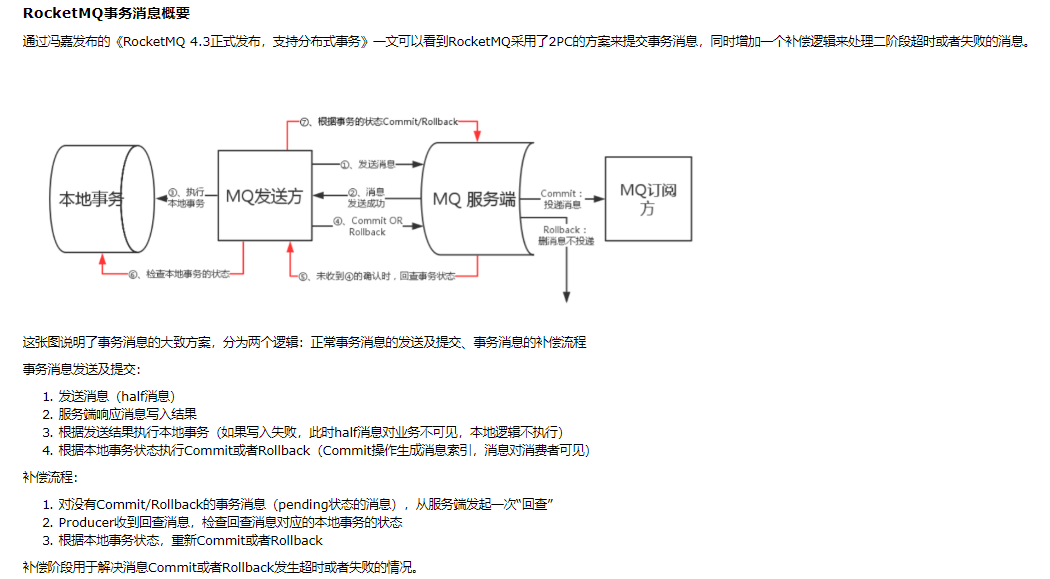


## 3.8 事物消息（小事务+异步）

支持事务方式对消息进行提交处理，在rocket里事务分两个阶段。

第一个阶段把消息传给MQ，只不过消费端不可见，但数据其实已经在Broker上了。

第二个阶段为本地消息回调处理，如果都成功返回COMMIT\_MESSAGE,则在broker上的数据对消费端可见，失败则为ROLLBACK\_MESSAGE，消费端不可见。



## 3.9 消息去重

　　幂等性：接口幂等性。

业务去重：添加消息操作记录表（比如唯一流水号），唯一索引。甚至高并发场景要配合分布式锁来实施。

## 3.10 消息重试

### 3.10.1 概念

Producer端重试

生产者端的消息失败，也就是Producer往MQ上发消息没有发送成功，比如网络抖动导致生产者发送消息到MQ失败。

这种消息失败重试我们可以手动设置发送失败重试的次数。

Consumer端重试

Consumer消费消息失败后，要提供一种重试机制，令消息再消费一次，Consumer消费消息失败通常可以认为有以下几种情况：

　1. 由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机被注销，无法充值）等。这种错误通常需要跳过这条消息，再消费其他消息，而且这条失败消息即使立刻重试消费，99%也不成功，所以最后提供一种定时的重试机制，即过10s再重试。

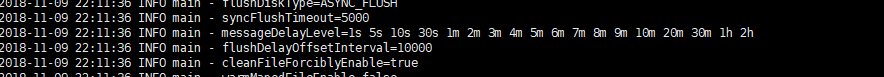
　　2. 由于依赖下游应用服务不可用，例如db连接不可用，外系统网络不可达等。遇到这种错误，即使跳过当前失败的消息，消费其他消息也会报错，这种情况下建议应用sleep 30s，再消费下一条消息，这样可以减轻Broker重试消息的压力。

### 3.10.2 Broker消息重试

正常情况下消费者在消息消费失败时会马上重新消费消息，正常情况下再次消费应该会将消息消费完成的，但是当一次消费消费不成功时我们可以设置让Broker延时一段时间再重新投递消息，是不是看到这就发现和延时消息类似了。

对于失败重试消息RocketMQ将其处理为延时消息进行操作，进行延时投递就可以了，设计还是比较牛的

查看broker.log文件，可以看到启动有很多的启动参数，其中有一条如下：



这里就表示的是消息重试的时间，1s,5s....的间隔时间后再进行消息的重试，这里是消息消费的消息重试。

### 3.10.3 Producer端消息重试实现

public class Producer {  
 public static void main(String[] args) throws MQClientException, RemotingException, InterruptedException, MQBrokerException {  
 DefaultMQProducer producer = new DefaultMQProducer("myGroup");  
 producer.setNamesrvAddr("localhost:9876");  
 producer.start();  
　　　　　//重试三秒  
 producer.setRetryTimesWhenSendFailed(3);  
 for (int i = 0; i < 10; i++) {  
 Message message = new Message("MyTopic", "tabA", ("Hello World" + i).getBytes());  
　　　　　　　//超时时间  
 SendResult result = producer.send(message,100);  
 System.*out*.println(result);  
 }  
 producer.shutdown();  
 }  
}

如果消息在100ms之内发送失败，就重试三次

### 3.10.4 Consumer端消息重试实现

在Consumer中，当消费消息的处理过程中，出现异常时，我们通常返回的是RECONSUME\_LATER，表示一会儿之后再重试，当返回了这个状态之后，broker就会按照2中的时间间隔来重试消息。当然，最大也只能重试到2h.

　　在实际的运用场景中，我们并不想要消息无止境的一直重试下去，可能我们回想要消息重试几次之后，还是不能成功的情况下就将这条消息存储到db或log文件中，所以此时我们可以这样实现：

public class Consumer {  
 public static void main(String[] args) throws MQClientException {  
 final DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("MyConsumerGroup");  
 consumer.setNamesrvAddr("47.105.145.123:9876;47.105.149.61:9876");  
 consumer.setConsumeFromWhere(ConsumeFromWhere.*CONSUME\_FROM\_LAST\_OFFSET*);  
 consumer.subscribe("MyTopic", "\*");  
 consumer.registerMessageListener(new MessageListenerConcurrently() {  
 @Override  
 public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  
 try {  
 MessageExt ext = msgs.get(0);  
 int x = 0;  
 String topic = ext.getTopic();  
 String body = new String(ext.getBody(),"utf-8");  
 if (Integer.*parseInt*(body) % 2 == 0) {  
 //产生异常  
 x = Integer.*parseInt*(body) / 0;  
 }  
 System.*out*.println("收到来自topic: " + topic + ",的消息：" + body);  
 } catch (Exception e) {  
 try {  
 MessageExt ext = msgs.get(0);  
 String topic = ext.getTopic();  
 String body = new String(ext.getBody(),"utf-8");  
 if (ext.getReconsumeTimes() == 3) {  
  
 //模拟将消息保存到db或日志文件中，返回成功状态，使消息不再重试  
 System.*out*.println("保存成功消息：" + body + "成功！！！");  
 return ConsumeConcurrentlyStatus.*CONSUME\_SUCCESS*;  
 }  
 System.*err*.println("err:收到来自topic: " + topic + ",的消息：" + body);  
 } catch (Exception e1) {  
 e1.printStackTrace();  
 }  
 return ConsumeConcurrentlyStatus.*RECONSUME\_LATER*;  
 }  
 return ConsumeConcurrentlyStatus.*CONSUME\_SUCCESS*;  
 }  
  
 });  
 consumer.start();  
 }  
}

这样就可以保证消息出现异常时我们可以做适当的操作避免消息一直重试或对于消息无法消费情况做一些补偿操作。

## 3.11 消息的推送与拉取

### 3.11.1 PushConsumer

推，Broker主动向Consumer推消息，它Consumer的一种，应用通常向对象注册一个Listener接口，一旦接收到消息，Consumer对象立刻回调Linstener接口方法。Push方式里,consumer把轮询过程封装了,并注册MessageListener监听器,取到消息后,唤醒MessageListener的consumeMessage()来消费,对用户而言,感觉消息是被推送过来的。

　　缺点：

　　慢消费无疑是Push模式最大的致命伤,如果消费者的速度比发送者的速度慢很多,势必造成消息在broker的堆积。假设这些消息都是有用的无法丢弃的,消息就要一直在broker端保存。当然这还不是最致命的,最致命的是broker给consumer推送一堆consumer无法处理的消息,consumer不是reject就是error,然后来回踢皮球。所以push适合于没有慢消费情况的场景下

### 3.11.2 PullConsumer

拉，Consumer主动的从Broker拉取消息，主动权由应用控制，可以实现批量的消费消息。Pull方式里,取消息的过程需要用户自己写,首先通过打算消费的Topic拿到MessageQueue的集合,遍历MessageQueue集合,然后针对每个MessageQueue批量取消息,一次取完后,记录该队列下一次要取的开始offset,直到取完了,再换另一个MessageQueue。

　　反观Pull模式,consumer可以按需消费,不用担心自己处理不了的消息来骚扰自己,而broker堆积消息也会相对简单,无需记录每一个要发送消息的状态,只需要维护所有消息的队列和偏移量就可以了。所以对于慢消费,消息量有限且到来的速度不均匀的情况,pull模式比较合适消息延迟与忙等。

　　这是Pull模式最大的短板。由于主动权在消费方,消费方无法准确地决定何时去拉取最新的消息。如果一次Pull取到消息了还可以继续去Pull,如果没有Pull取到则需要等待一段时间重新Pull，在阿里中的解决是长轮询 Pull,消费者如果尝试拉取失败,不是直接return,而是把连接挂在那里wait,服务端如果有新的消息到来,把连接notify起来,这也是不错的思路。但海量的长连接block对系统的开销还是不容小觑的,还是要合理的评估时间间隔,给wait加一个时间上限比较好。

## 3.12 消息延迟消息实现

（1）生产者将延时消息发送到Broker，Broker是如何区分普通消息和延时消息（消息类型）

1. 消息如何保证不被消费者拉取到的（RocketMQ将消息以SCHEDULE\_TOPIC\_XXXX为topic将延时消息持久化，等到达延时消息之后再以原有的topic重新保存）

延时消息的实现还是挺精巧的，首先将延时消息换了一个topic名称进行持久化，这样消费者就无法获取消息，然后有定时任务，会将消息还原到原有的topic信息，这样消费者又可以重新拉取消息了。

# 常见消息中间对比



# 部署RocketMQ

## 6.1 单台服务器部署

### 6.1.1 下载源码

wget http://mirrors.hust.edu.cn/apache/rocketmq/4.3.0/rocketmq-all-4.3.0-source-release.zip

### 6.1.2 构建

1. > unzip rocketmq-all-4.3.0-source-release.zip
2. > cd rocketmq-all-4.3.0/
3. > mvn -Prelease-all -DskipTests clean install -U
4. > cd distribution/target/apache-rocketmq

### 6.1.3 启动名称服务器

> nohup sh bin/mqnamesrv &

> tail -f ~/logs/rocketmqlogs/namesrv.log

The Name Server boot success...

如果在启动服务器的时候报：

Java HotSpot(TM) 64-Bit Server VM warning: INFO: os::commit\_memory(0x00000006c0000000, 2147483648, 0) failed; error='Cannot allocate memory' (errno=12)

那么需要调整rocketMQ的内存

> vim bin/runserver.sh

JAVA\_OPT="${JAVA\_OPT} -server -Xms128m -Xmx256m -Xmn256m -XX:PermSize=128m -XX:MaxPermSize=320m"

-Xms 的值一定要比 -Xmx 要小不让，也会报错：如下

Initial heap size set to a larger value than the maximum heap size那么需要调整rocketMQ的内存

### 6.1.4 启动broker

# autoCreateTopicEnable=true 参数可以在启动Broker时指定自动创建Topic:

> nohup sh bin/mqbroker -n localhost:9876 autoCreateTopicEnable=true &

> tail -f ~/logs/rocketmqlogs/broker.log

The broker[%s, ip:port] boot success...

如果在启动服务器的时候报：

Java HotSpot(TM) 64-Bit Server VM warning: INFO: os::commit\_memory(0x00000005c0000000, 8589934592, 0) failed; error='Cannot allocate memory' (errno=12)

那么需要调整broker的内存

> vim bin/runbroker.sh

JAVA\_OPT="${JAVA\_OPT} -server -Xms256m -Xmx256m -Xmn128m"

设置好后启动，

### 6.1.5 发送和接收消息

> export NAMESRV\_ADDR=localhost:9876

> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer

SendResult [sendStatus=SEND\_OK, msgId= ...

> sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer

ConsumeMessageThread\_%d Receive New Messages: [MessageExt...

### 6.1.6 关闭服务器

> sh bin/mqshutdown broker

The mqbroker(36695) is running...

Send shutdown request to mqbroker(36695) OK

> sh bin/mqshutdown namesrv

The mqnamesrv(36664) is running...

Send shutdown request to mqnamesrv(36664) OK

## 6.2 集群服务器部署（第3种模式）

[http://10.104.116.139:8080/#/cluster](http://10.104.116.139:8080/" \l "/cluster)



### **1. 单个 Master**

   这种方式风险较大，一旦Broker 重启或者宕机时，会导致整个服务不可用，不建议线上环境使用。

### **2. 多 Master 模式**

   一个集群无 Slave，全是 Master，例如 2 个 Master 或者 3 个 Master

   优点：配置简单，单个Master 宕机或重启维护对应用无影响，在磁盘配置为 RAID10 时，即使机器宕机不可恢复情况下，由与 RAID10 磁盘非常可靠，消息也不会丢（异步刷盘丢失少量消息，同步刷盘一条不丢）。性能最高。

   缺点：单台机器宕机期间，这台机器上未被消费的消息在机器恢复之前不可订阅，消息实时性会受到受到影响。

   ###  先启动 NameServer，例如机器 IP 为：10.104.112.222:9876

|  |  |
| --- | --- |
| 1 | nohup sh mqnamesrv & |

   ###  在机器 A，启动第一个 Master

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-noslave/broker-a.properties & |

   ###  在机器 B，启动第二个 Master

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-noslave/broker-b.properties & |

### **3. 多 Master 多 Slave 模式，异步复制**

   每个 Master 配置一个 Slave，有多对Master-Slave，HA 采用异步复制方式，主备有短暂消息延迟，毫秒级。

   优点：即使磁盘损坏，消息丢失的非常少，且消息实时性不会受影响，因为 Master 宕机后，消费者仍然可以从 Slave 消费，此过程对应用透明。不需要人工干预。性能同多 Master 模式几乎一样。

   缺点：Master 宕机，磁盘损坏情况，会丢失少量消息。

   ###  先启动 NameServer，例如机器 IP 为：10.104.112.222:9876

|  |  |
| --- | --- |
| 1 | nohup sh mqnamesrv & |

   ###  在机器 A，启动第一个 Master

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-async/broker-a.properties & |

   ###  在机器 B，启动第二个 Master

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-async/broker-b.properties & |

   ###  在机器 C，启动第一个 Slave

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-async/broker-a-s.properties & |

   ###  在机器 D，启动第二个 Slave

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-async/broker-b-s.properties & |

### **4. 多 Master 多 Slave 模式，同步双写**

   每个 Master 配置一个 Slave，有多对Master-Slave，HA 采用同步双写方式，主备都写成功，向应用返回成功。

   优点：数据与服务都无单点，Master宕机情况下，消息无延迟，服务可用性与数据可用性都非常高

   缺点：性能比异步复制模式略低，大约低 10%左右，发送单个消息的 RT 会略高。目前主宕机后，备机不能自动切换为主机，后续会支持自动切换功能。

   ###  先启动 NameServer，例如机器 IP 为：10.104.112.222:9876

|  |  |
| --- | --- |
| 1 | nohup sh mqnamesrv & |

   ###  在机器 A，启动第一个 Master

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-sync/broker-a.properties & |

   ###  在机器 B，启动第二个 Master

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-sync/broker-b.properties & |

   ###  在机器 C，启动第一个 Slave

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-sync/broker-a-s.properties & |

   ###  在机器 D，启动第二个 Slave

|  |  |
| --- | --- |
| 1 | nohup sh mqbroker -n 10.104.112.222:9876 -c ../conf/2m-2s-sync/broker-b-s.properties & |

   以上 Broker 与 Slave 配对是通过指定相同的brokerName 参数来配对，Master 的 BrokerId 必须是 0，Slave的BrokerId 必须是大与 0 的数。另外一个 Master 下面可以挂载多个 Slave，同一 Master 下的多个 Slave 通过指定不同的 BrokerId 来区分。

# 第七章 管理控制台

## 7.1 安装部署

1）.下载git clone <https://github.com/apache/rocketmq-externals.git>

1. .修改namesrv信息

>rocketmq-externals>rocketmq-console>src>main>resources>application.properties

rocketmq.config.namesrvAddr=10.104.116.139:9876;10.104.116.140:9876;10.104.130.243:9876;10.104.130.244:9876

2）.打包

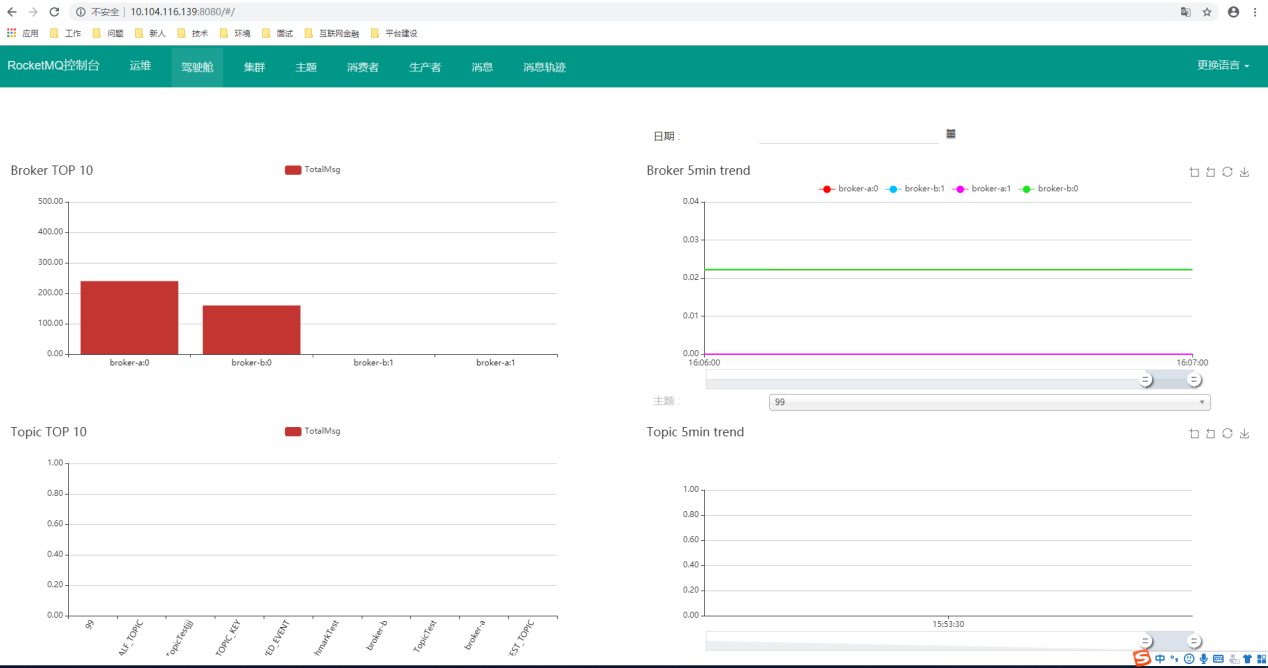
> cd rocketmq-externals/rocketmq-console

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

3）.启动 java -jar target/rocketmq-console-ng-1.0.0.jar,访问 [http://10.104.116.139:8080](http://localhost:8080)

## 7.2 功能介绍

运维页面打开后，从左至右有7个Tab,分别是：配置、驾驶舱、集群信息、Topic信息、Consumer信息、Producer信息和消息查询。如图：



1）配置页面，设置好NaveServer的地址。修改这个服务是否使用VIPChannel，取决于你的RocketMQ版本，如果版本小于3.5.8，请设置不使用，否则保持默认值（VIPChannel用于实现读写分离，是3.5.8以后的版本才增加的功能）

2）在驾驶舱中可以查看Broker的消息量（总量5分钟图），还可以查看单一主题的消息量（总量／趋势图）。

3）在集群信息页面，可以查看集群数量、地址、主从的分布情况，还可以查看Broker的运行状态信息和配置信息。

4）Topic页面展示所有的主题，可以通过搜索框进行过滤，筛选普通／重试／死信类型的主题；还可以添加／更新主题，修改主题的配置参数。每个参数的含义和MQAdmin命令中updateTope命令的参数对应。还可以查看每个主题的消息投递状态，消息的路由信息（这个主题的消息会发往哪些Broker，对应Broker的MessageQueue信息）。还可以向某个主题发送测试消息和重置消费位点（Offset）。

5）Consumer信息页面展示所有的消费组，还可以通过搜索框进行搜索，手动刷新页面或每隔五秒定时刷新页面，按照订阅组／数量／TPS／延迟进行排序，添加／更新消费组等。

Producer信息页面，可以通过Topic和Group查询在线的消息生产者信息，信息包含客户端的主机、版本等。

6）消息查询页面，可以根据Topic的时间、Key和消息ID进行消息查询。消息详情可以展示这条消息的详细内容。消息详情可以查看消息对应的具体消费组的消费情况（如果异常，可以查看具体的异常信息）。可以向指定的消费组重发消息。

NOT\_ONLINE 订阅端不在线

CONSUMED 消息已经被投递。订阅端返回ReconsumerLater，或者返回NULL，或者抛出异常，消息都会走重试流程，消息投递状态都是CONSUMED。

CONSUMED\_BUT\_FILTERED 消息已经被投递且被过滤。比如，发布端发布消息topicA，tagA，订阅端订阅topicA，tagB

NOT\_CONSUME\_YET 消息未被投递。有可能消息发生了堆积，还未被消费；也有可能消费线程hang住了，导致消费线程迟迟没有返回。

# 第八章 Springcloud集成RocketMQ

## 8.1 添加pom依赖

<dependency>  
 <groupId>org.apache.rocketmq</groupId>  
 <artifactId>rocketmq-client</artifactId>  
 <version>4.2.0</version>  
</dependency>

## 8.2 配置文件application.yml

**rocketmq:** *# 生产者配置* **producer:  
 groupName:** rocketmq-group  
 **namesrvAddr:** 10.104.112.222:9876  
 **default:** false  
 *# 消费者配置* **consumer:  
 groupName:** rocketmq-group  
 **namesrvAddr:** 10.104.112.222:9876

## 8.3 消息生产者配置类

package com.ucarinc.blink.ota.rocketmq.config;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.context.annotation.Configuration;  
  
@ConfigurationProperties(prefix = "rocketmq.producer")  
@Configuration  
public class ProducerConfig {  
   
 private String namesrvAddr;  
   
 private String groupName;  
  
 public String getNamesrvAddr() {  
 return namesrvAddr;  
 }  
  
 public void setNamesrvAddr(String namesrvAddr) {  
 this.namesrvAddr = namesrvAddr;  
 }  
  
 public String getGroupName() {  
 return groupName;  
 }  
  
 public void setGroupName(String groupName) {  
 this.groupName = groupName;  
 }  
   
 @Override  
 public String toString() {  
 return "ProducerConfig [namesrvAddr=" + namesrvAddr + ", groupName=" + groupName + "]";  
 }  
}

## 8.4 消息生产者

package com.ucarinc.blink.ota.rocketmq.config;  
  
import org.apache.rocketmq.client.exception.MQClientException;  
import org.apache.rocketmq.client.producer.DefaultMQProducer;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class ProducerConfigure {  
  
 Logger log = LoggerFactory.*getLogger*(ProducerConfigure.class);  
  
 @Autowired  
 private ProducerConfig producerConfigure;  
  
 */\*\*  
 \* 创建普通消息发送者实例  
 \*   
 \** ***@return*** *\** ***@throws*** *MQClientException  
 \*/* @Bean  
 public DefaultMQProducer defaultProducer() throws MQClientException {  
 log.info(producerConfigure.toString());  
 log.info("defaultProducer 正在创建---------------------------------------");  
 DefaultMQProducer producer = new DefaultMQProducer(producerConfigure.getGroupName());  
 producer.setNamesrvAddr(producerConfigure.getNamesrvAddr());  
 producer.setVipChannelEnabled(false);  
 producer.setRetryTimesWhenSendAsyncFailed(10);  
 producer.start();  
 log.info("rocketmq producer server开启成功---------------------------------.");  
 return producer;  
 }  
}

## 8.5 消息消费者配置类

package com.ucarinc.blink.ota.rocketmq.config;  
  
import org.springframework.boot.context.properties.ConfigurationProperties;  
import org.springframework.context.annotation.Configuration;  
  
@ConfigurationProperties(prefix = "rocketmq.consumer")  
@Configuration  
public class ConsumerConfig {  
   
 private String groupName;  
   
 private String namesrvAddr;  
  
 public String getGroupName() {  
 return groupName;  
 }  
  
 public void setGroupName(String groupName) {  
 this.groupName = groupName;  
 }  
  
 public String getNamesrvAddr() {  
 return namesrvAddr;  
 }  
  
 public void setNamesrvAddr(String namesrvAddr) {  
 this.namesrvAddr = namesrvAddr;  
 }  
  
 @Override  
 public String toString() {  
 return "ConsumerConfig [groupName=" + groupName + ", namesrvAddr=" + namesrvAddr + "]";  
 }  
   
}

## 8.6 消息消费者

package com.ucarinc.blink.ota.rocketmq.config;  
  
import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;  
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;  
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;  
import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;  
import org.apache.rocketmq.client.exception.MQClientException;  
import org.apache.rocketmq.common.message.MessageExt;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Configuration;  
  
import java.util.List;  
  
@Configuration  
public abstract class DefaultConsumerConfigure {  
  
 Logger log = LoggerFactory.*getLogger*(DefaultConsumerConfigure.class);  
   
 @Autowired  
 private ConsumerConfig consumerConfig;  
  
 // 开启消费者监听服务  
 public void listener(String topic, String tag) throws MQClientException {  
 log.info("开启" + topic + ":" + tag + "消费者-------------------");  
 log.info(consumerConfig.toString());  
  
 DefaultMQPushConsumer consumer = new DefaultMQPushConsumer(consumerConfig.getGroupName());  
  
 consumer.setNamesrvAddr(consumerConfig.getNamesrvAddr());  
  
 consumer.subscribe(topic, tag);  
  
 // 开启内部类实现监听  
 consumer.registerMessageListener(new MessageListenerConcurrently() {  
 @Override  
 public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  
 return DefaultConsumerConfigure.this.dealBody(msgs);  
 }  
 });  
  
 consumer.start();  
  
 log.info("rocketmq启动成功---------------------------------------");  
  
 }  
   
 // 处理body的业务  
 public abstract ConsumeConcurrentlyStatus dealBody(List<MessageExt> msgs);  
  
}

## 8.7 具体业务类

package com.ucarinc.blink.ota.rocketmq.service;  
  
import com.ucarinc.blink.ota.rocketmq.config.DefaultConsumerConfigure;  
import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;  
import org.apache.rocketmq.client.exception.MQClientException;  
import org.apache.rocketmq.common.message.MessageExt;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.context.ApplicationListener;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.event.ContextRefreshedEvent;  
  
import java.io.UnsupportedEncodingException;  
import java.util.List;  
  
@Configuration  
public class CustomConsumer extends DefaultConsumerConfigure implements ApplicationListener<ContextRefreshedEvent> {  
  
 Logger log = LoggerFactory.*getLogger*(CustomConsumer.class);  
  
 @Override  
 public void onApplicationEvent(ContextRefreshedEvent arg0) {  
 try {  
 super.listener("TopicTest", "Tag1");  
 } catch (MQClientException e) {  
 log.error("消费者监听器启动失败", e);  
 }  
  
 }  
  
 @Override  
 public ConsumeConcurrentlyStatus dealBody(List<MessageExt> msgs) {  
 int num = 1;  
 log.info("进入");  
 for (MessageExt msg : msgs) {  
 log.info("第" + num + "次消息");  
 try {  
 String msgStr = new String(msg.getBody(), "utf-8");  
 log.info(msgStr);  
 } catch (UnsupportedEncodingException e) {  
 log.error("body转字符串解析失败");  
 }  
 }  
 return ConsumeConcurrentlyStatus.*CONSUME\_SUCCESS*;  
 }  
}

这个CustomConsumer类实现了ApplicationListener，让他在启动的时候就开启执行DefaultConsumerConfigure的listener方法

## 8.8 新建springboot启动类RocketMqApplication.java

package com.ucarinc.blink.ota.rocketmq;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class RocketMqApplication {  
   
 public static void main(String[] args) {  
 SpringApplication.*run*(RocketMqApplication.class, args);  
 }  
  
}

## 8.9 新建一个Controller，引入消息生产者

package com.ucarinc.blink.ota.rocketmq.controller;  
  
import com.alibaba.fastjson.JSON;  
import org.apache.rocketmq.client.producer.DefaultMQProducer;  
import org.apache.rocketmq.client.producer.SendCallback;  
import org.apache.rocketmq.client.producer.SendResult;  
import org.apache.rocketmq.common.message.Message;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import javax.servlet.http.HttpServletResponse;  
  
@RestController  
public class ProducerController {  
  
 Logger log = LoggerFactory.*getLogger*(ProducerController.class);  
  
 @Autowired  
 private DefaultMQProducer producer;  
  
 @GetMapping("/msg/product")  
 public void test(String info, HttpServletResponse response) throws Exception {  
 Message message = new Message("TopicTest", "Tag1", "12345", "rocketmq测试成功".getBytes());  
 // 这里用到了这个mq的异步处理，类似ajax，可以得到发送到mq的情况，并做相应的处理  
 // 不过要注意的是这个是异步的

/\*\*一般一条消息主要有四个属性Topic，tags，key，msgBody。

Topic

消息主题，一个Producer实例只能对于一个topic，一条消息也必须属于一个topic。

tags

消息标签，可用来做服务端消息过滤。一个topic下可以有很多tags，一般都通过topic+tags来消费自己想要的结果。

keys

消息关键词，查询消息使用。

msgBody

消息体，所需要发的内容。

\*\*/  
 producer.setVipChannelEnabled(false);  
 producer.send(message, new SendCallback() {  
 @Override  
 public void onSuccess(SendResult sendResult) {  
 log.info("传输成功");  
 log.info(JSON.*toJSONString*(sendResult));  
 }  
  
 @Override  
 public void onException(Throwable e) {  
 log.error("传输失败", e);  
 }  
 });  
 response.getWriter().write("send msg success");  
 }  
  
}

8.1 验证服务

访问http://127.0.0.1:8080/msg/product

可以看到controller产生消息，然后CustomConsumer类的dealBody方法消息消息

8.2 更多DEMO，请参考官方例子

