# 为什么要使用消息中间件

在微服务架构下，一个业务服务会被拆分成多个微服务，各个服务之间相互通信完成整体的功能。系统间的通信协作通常有两种：

·Http/RPC通信：优点是通信实时，缺点是服务之间的耦合性高

·消息通信：优点是降低了服务之间的耦合度，提高了系统的处理能力，缺点是通

信非实时。

例如，用户交易完成后发送短信通知，假设交易耗时5ms，发短信耗时3ms。如果是实时通信，那么用户收到返回结果耗时8ms，但发短信是非核心步骤，可以从主流程中剥离出来异步处理，那么用户收到返回结果耗时就可以从8ms下降到5ms。

# 认识RocketMQ

## 什么是RocketMQ

RocketMQ是一个低延迟、高可靠、可伸缩、易于使用的分布式消息中间件（也称消息队列），经过阿里巴巴多年双11的验证，是阿里巴巴捐献给Apache的顶级项目。RocketMQ具有高吞吐、低延迟、海量消息堆积等优点，同时提供顺序消息、事务消息、定时消息、消息重试与追踪等功能，非常适合在电商、金融等领域广泛使用。

## RocketMQ的应用场景

RocketMQ的应用场景如下：

·削峰填谷：诸如秒杀、抢红包、企业开门红等大型活动皆会带来较高的流量脉冲，很

可能因每做好相应的保护而导致系统超负荷甚至崩溃，或因限制太过导致请求大量失

败而影响用户体验，RocketMQ可提供削峰填谷的服务来解决这些问题。

·异步解耦：交易系统作为淘宝/天猫主站最核心的系统，每笔交易订单数据的产生会

引起几百个下标业务系统的关注，包括物流、购物车、积分、流计算分析等，整体业

务系统庞大而且复杂，RocketMQ可实现异步通信和应用解耦，确保主站业务的连续

性。

·顺序收发：日常中需要保证顺序的应用场景非常多，例如证券交易过程中的时间优先

原则，交易系统中的订单创建、支付、退款等流程，航班中的旅客登机消息处理等。

与先进先出（First In First Out，缩写FIFO）原理类似，RocketMQ提供的顺序消息即保

证消息的FIFO。

·分布式事务一致性：交易系统、红包等场景需要确保数据的最终一致性，大量引入

RocketMQ的分布式事务，既可以实现系统之间的解耦，又可以保证最终的数据一致

性。

·大数据分析：数据在“流动”中产生价值，传统数据分析大都基于批量计算模型，无

法做到实时的数据分析，利用RocketMQ与流式计算引擎相结合，可以很方便地实现

对业务数据进行实时分析。

·分布式缓存同步：天猫双11大促，各个分会场的商品需要实时感知价格的变化，大

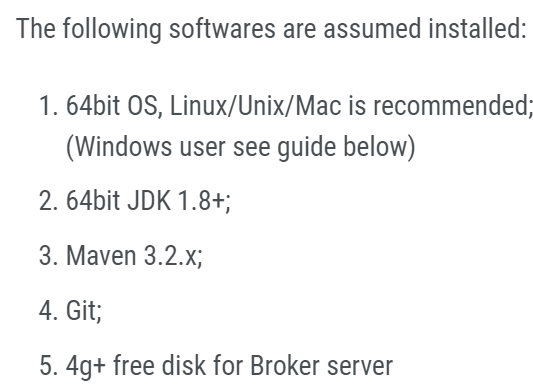
量并发访问会导致页面响应时间较长，集中式缓存因为带宽瓶颈限制商品变更的访问

流量，通过RocketMQ构件分布式缓存，可实时通知商品数据的变化。

## RocketMQ的安装与启动

一**、安装与启动**

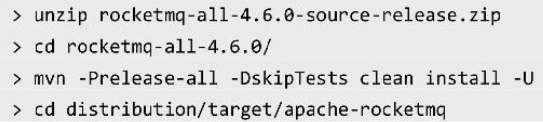
RocketMQ依赖Java环境，要求有JDK1.8及以上版本。官方给的建议如下：



RocketMQ支持三种集群部署模式，本章后面章节会进行详细说明，本书使用简单的单击部署模式进行演示。RocketMQ的安装方式有两种，一种是源码安装，另一种是使用已经编译好直接可用的安装包，这里以源码方式安装（确保安装了Java1.8+以及Maven3.2.x）。

·在官网下载RocketMQ的最新版本4.6.0

·解压编译并打包



·启动集群管理NameServer，RocketMQ默认端口是9876

**nohup sh bin/mqnamesrv &**

启动后，监控namesrv.log日志：

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

当打印出以下内容则表示启动成功：



·启动消息服务器Broker，指定NameServer的IP地址和端口，支持指定多个

NameServer

**nohup sh bin/mqbroker -n localhost:9876 &**

查看并监控日志：

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

打印出以下内容则表示启动成功：



**二、关闭**

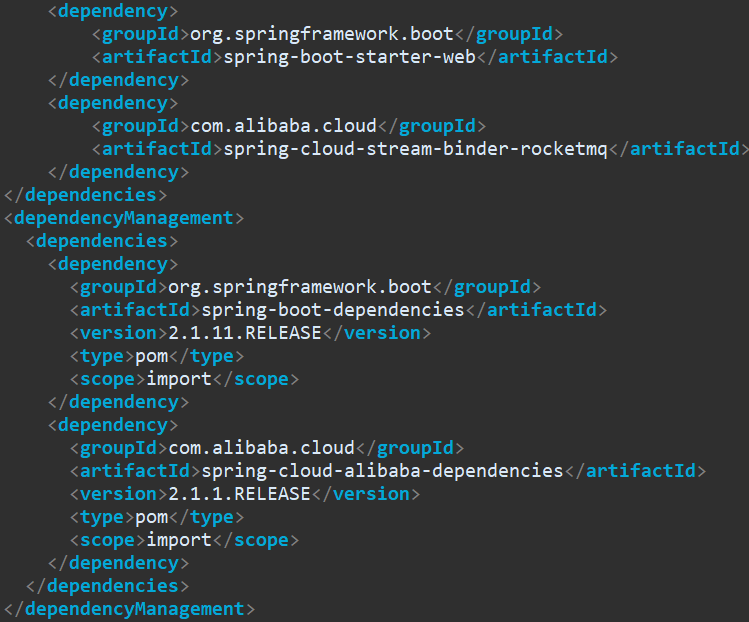
关闭NameServer：**sh bin/mqshutdown namesrv**

关闭Broker：**sh bin/mqshutdown broker**

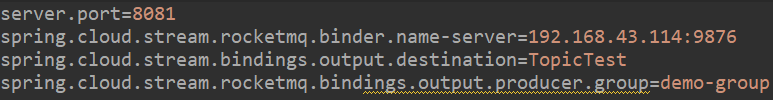
## RocketMQ如何发送消息

Spring Cloud Alibaba已集成了RocketMQ，使用Spring Cloud Stream对RocketMQ发送和接收消息。

1. 创建一个SpringBoot项目，在pom.xml引入依赖。

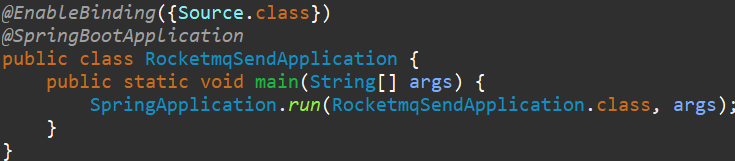


1. 配置application.properties



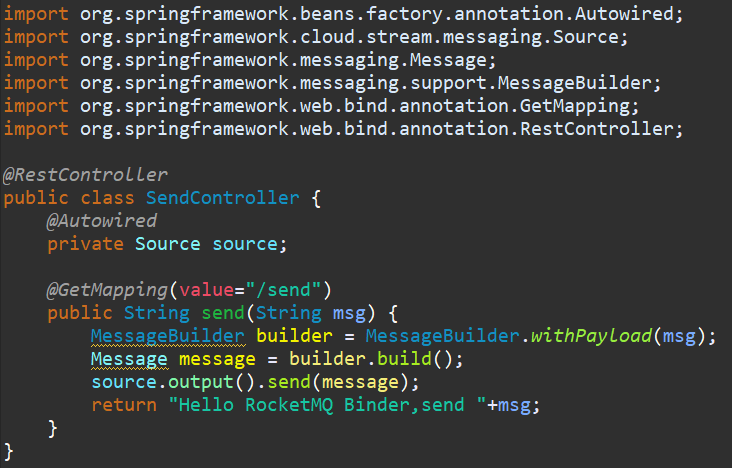
name-server用于指定RocketMQ的NamServer地址，将指定名称为output的Bingding消息发送到TopicTest。

1. 在启动类上使绑定消息通道



@EnableBinding({Source.class})表示绑定配置文件中名称为output的消息通道Binding，Source类中定义消息通道名称为output（在Source类的源码可以查看到）。

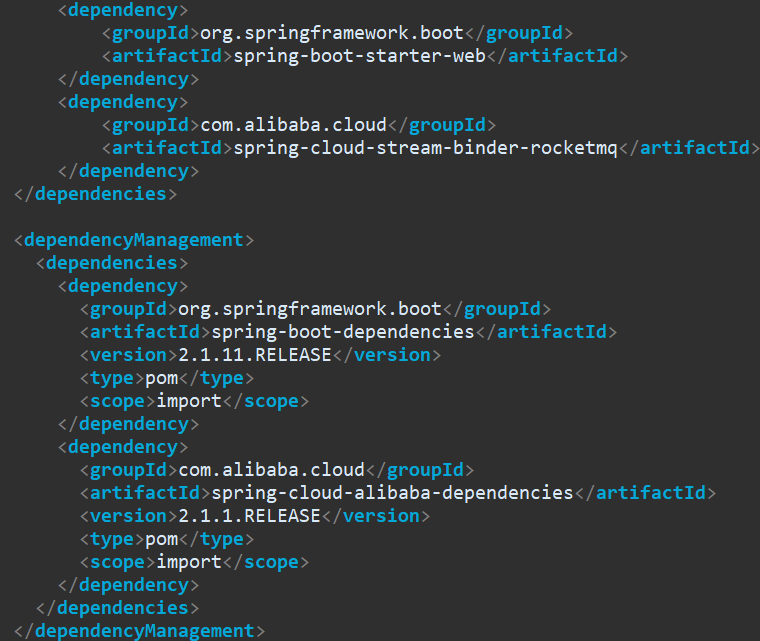
1. 定义控制器发送消息



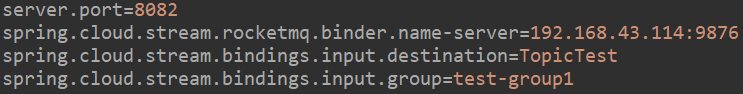
访问该接口时将参数msg作为消息发送到RocketMQ中。

## RocketMQ如何消费消息

1. 创建一个SpringBoot项目，在pom.xml中引入依赖。

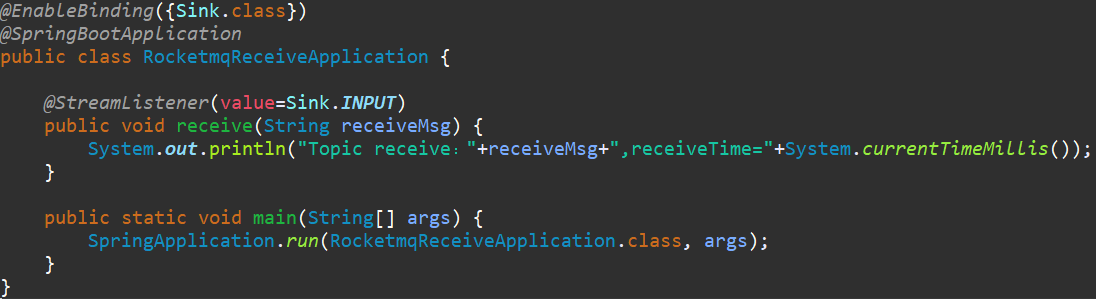


1. 配置application.properties



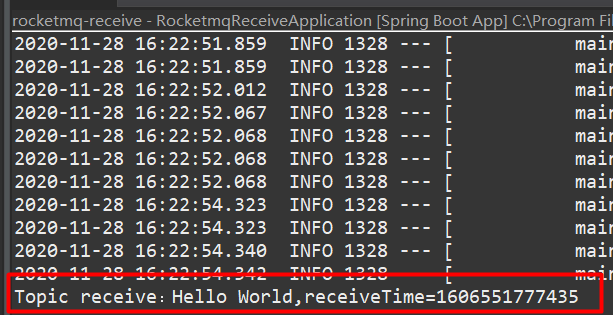
name-server指定RocketMQ的NameServer地址，destination指定Topic名称，指定名称为input的Binding接收TopicTest的消息。

1. 定义消息监听



@EnableBinding({Sink.class})表示绑定配置文件中名称为input的消息通道Binding，Sink类定义的消息通道的名称为input（查看Sink类源码可以发现），@StreamListener表示定义一个消息监听器，接收RocketMQ中的消息，并通过value属性指定通道。

1. 启动RocketMQ的NameServer和Broker（消息服务器）后，启动生产者和消费者，访问http://localhost:8081/send?msg=helloword，观察消费者是否消费生产者的消息：



消费成功。

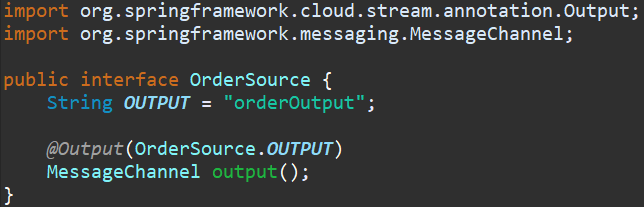
## 自定义通道

在实际开发场景中会存在多个发送消息通道和多个接收消息通道，此时我们可以自定义发送消息通道或接收消息通道。

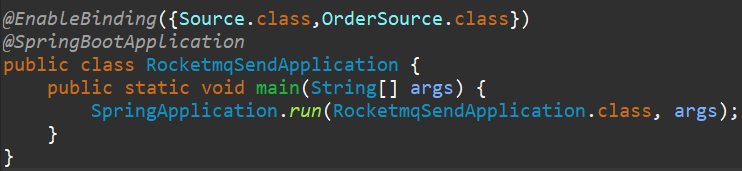
**自定义发送消息通道**

对于发送消息通道，可参考Source接口自定义一个接口，修改通道名称和相关配置即可。

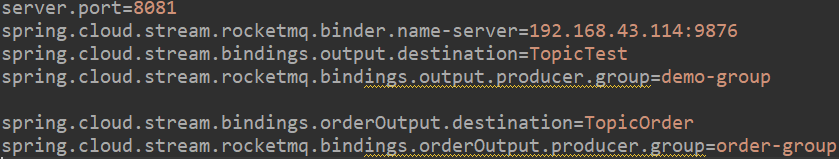
1. 参考Source接口，自定义一个名为orderOutput的发送消息通道



2）使用@EnableBinding注解进行绑定



1. 在application.properties中指定



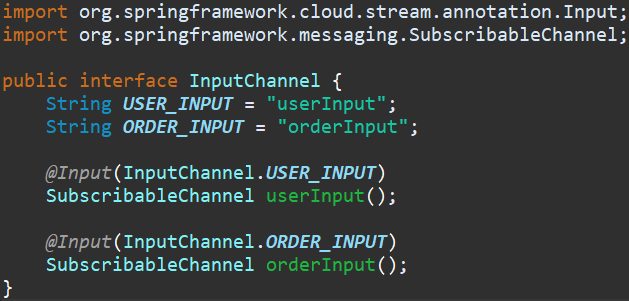
到此，就可以添加一个自定义发送消息通道，使用orderOutput消息发送到

TopticOrder中了。

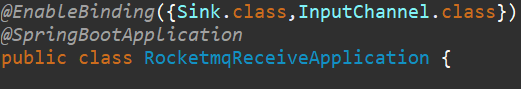
**自定义接收消息通道**

对于接收消息通道，可参考Sink接口自定义一个接口，修改通道名称和相关配置即可。

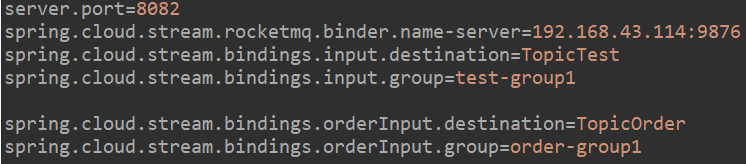
1. 参考Sink接口，自定义两个接收消息通道，分别名为userInput和orderInput



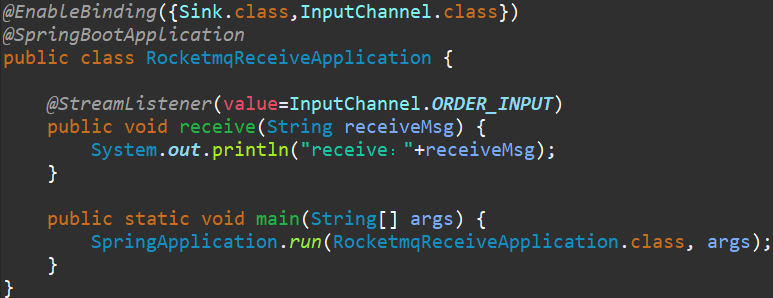
1. 使用@EnabeBinding注解进行绑定



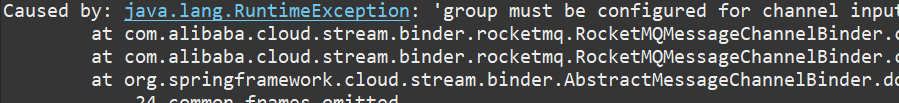
1. 在application.properties中指定



1. 使用@StreamListener定义消息监听器时指定



需要注意，@EnabledBinding注解中如果绑定的是接收消息通道类型的接口，则必须在application.properties中为该接口中定义的各个通道都分别指定Topic和Group，并且Topic和Group不能重复。否则在启动消费者时会提示错误，例如在消费者中，@EnabledBinding注解绑定了Sink接口时没有为其指定Topic和Group，启动项目报错如下：



## 问题

已在CSDN写博客进行了说明。

# Spring Cloud Alibaba RocketMQ

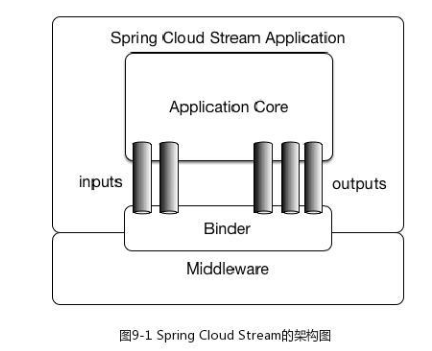
## 认识Spring Cloud Stream

**1、什么是Spring Cloud Stream**

Spring Cloud Stream是Spring Cloud体系内的一个框架，用于构建与共享消息传递系统连接高度可伸缩的事件驱动微服务，其目的是简化消息业务在Spring Cloud应用程序中的开发。

**2、Spring Cloud Stream架构图**

Spring Cloud Stream的架构图如下所示：



应用程序通过Spring Cloud Stream注入的输入通道inputs和输出通道outputs与消息中间件Middleware通信，消息通道通过特定的中间件绑定器Binder实现连接到外部代理。

**3、Spring Cloud Stream核心构成**

Spring Cloud Stream的实现基于发布/订阅机制，核心由四部分构成，Spring Framework中的Spring Message和Spring Integration，以及Spring Cloud Stream中的Binders和Bindings。

**Spring Messaging**：Spring Framework中的统一消息编程模型，其核心对象如下：

·Message：消息对象，包含消息头Header和消息体Payload

·MessageChannel：消息通道接口，用于接收消息，提供send方法将消息发

送至消息通道

·MessageHandler：消息处理器接口，用于处理消息逻辑

**Spring Integration**：Spring Framework中用于支持企业集成的一种扩展机制，作用

是提供一个简单的模型来构建企业集成解决方案，对Spring

Messaging进行了扩展

·MessageDispatcher：消息分发接口，用于分发消息和添加删除消息处理器

·MessageRouter：消息路由接口，定义默认的输出消息通道

·Filter：消息的过滤注解，用于配置消息过滤表达式

·Aggregator：消息的聚合注解，用于将多条消息聚合成一条

·Splitter：消息的分割，用于将一条消息拆分成多条

**Binders**：目标绑定器，负责与外部消息中间件系统集成的组件。

·doBindProducer：绑定消息中间件客户端发送消息模块

·doBindConsumer：绑定消息中间件客户端接收消息模块

**Bindings：**外部消息中间件系统与应用程序提供的消息生产者和消费者（由Binders

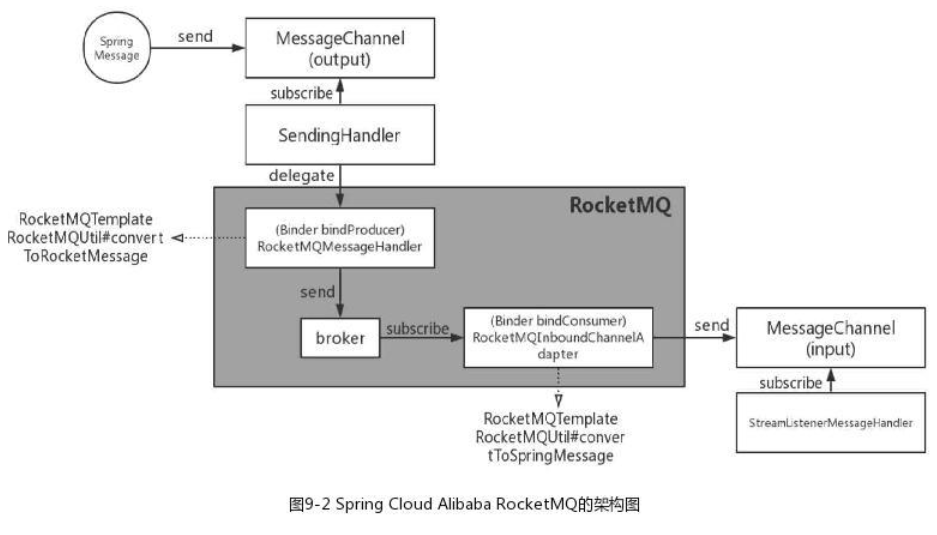
创建）之间的桥梁。

**4、Spring Cloud Stream与消息中间件**

Spring Cloud Stream官方提供了Kafka Binder和RabbitMQBinder，用于集成Kafka和RabbitMQ，Spring Cloud Alibaba中加入了RocketMQ Binder，用于将RocketMQ集成到Spring Cloud Stream。

## Spring Cloud Alibaba RocketMQ架构图

Spring Cloud Alibaba RocketMQ的架构图如下所示：



总体分为四部分：

·MessageChannel（output）：消息通道，用于发送消息，Spring Cloud Stream提供的标

准接口。

·MessageChannel（input）：消息通道，用于订阅消息，Spring Cloud Stream提供的标准

接口。

·Binder bindProducer：目标绑定器，将发送通道发过来的消息发送到RocketMQ消息

服务器，由Spring Cloud Alibaba团队按照Spring Cloud Stream的标准协议实现。

·Binder bindConsumer：目标绑定器，将接收到的RocketMQ消息服务器的消息推送给

订阅通道，由Spring Cloud Alibaba团队按照Spring Cloud Stream的标准协议实现。

## 源码分析

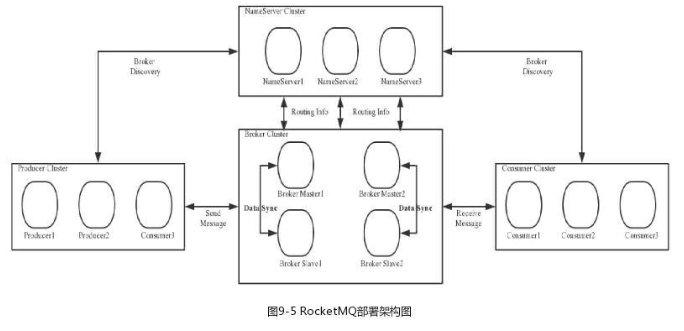
请参考教材书。

# RocketMQ集群管理

在分布式服务SOA架构中，任何中间件或者应用都不允许单点存在，服务发现机制是必备的，服务实例有多个，且数量是动态变化的。注册中心会提供服务管理能力，服务调用方（服务消费者）在注册中心获取服务提供者的信息，从而进行远程调用。

## 整体架构设计

说到RocketMQ的架构设计，不得不说一下它与Kafka的渊源。Kafka是一款高性能的消息中间件，在大数据场景中经常使用，但由于Kafka不支持消费失败重试、定时消息、事务消息，顺序消息也有明显缺陷，难以支撑淘宝交易、订单、充值的那个复杂业务场景。淘宝中间件团队参考Kafka重新设计并使用Java编写了RocketMQ，因此在RocketMQ中会有一些概念和Kafka类似。场景的消息中间件Kafka、RabbitMQ、RocketMQ等都基于发布/订阅机制，消息发送者（Producer）把消息发送到消息服务器，消息消费者（Consumer）从消息服务器订阅感兴趣的消息。这个过程中消息发送者和消息消费者是客户端，消息服务器时服务端，客户端与服务端双方都需要通过注册中心感知对方的存在。RocketMQ部署架构主要分为四部分，如下图所示：



·Producer：消息发布的角色，主要负责把消息发送到Broker，支持分布式集群方

式部署。

·Consumer：消息消费者的角色，主要负责从Broker订阅消息消费，支持分布式

集群方式部署。

·Broker：消息存储的角色，主要负责消息的存储、投递和查询，以及服务高可用

保证，支持分布式集群方式部署。

·NameServer：服务管理的角色，主要负责管理Broker集群的路由信息，支持分

布式集群方式部署。

其中，NameServer是一个非常简单的Topic路由注册中心，其角色类似于Dubbo中依赖的ZooKeeper，支持Broker的动态注册与发现。主要包括如下两个功能：

1. 服务注册：NameServer接收Broker集群的注册信息，保存下来作为路由信息的基本数据，并提供心跳检测机制，检查Broker是否还存活。
2. 路由信息管理：NameServer保存了Broker集群的路由信息，用于提供给客户端查询Broker的队列消息。Producer和Consumer通过NameServer可以知道Broker集群的路由信息，从而进行消息的投递和消息。

## RocketMQ相关概念

·Message：消息，系统所传输信息的物理载体，生产和消费数据的最小单位。每条消

息必须属于一个Topic，RocketMQ中每条消息拥有唯一的MessageID，且可以携带具

有业务标识的Key。

·Topic：主题，表示一类消息的集合，每个主题都包含若干条消息，每条消息只能属

于一个主题，Topic是RocketMQ进行消息订阅的基本单位。

·Queue：消息队列，组成Topic的最小单元。默认情况下一个Topic会对应多个Queue

，Topic是逻辑概念，Queue是物理存储，在Consumer消费Topic时底层实际则拉

Queue中的消息。

·Tag：为消息设置的标志，用于同一主题下区分不同类型的消息。来自同一业务单元

的消息，可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代

码的清晰度和连贯性，并优化RocketMQ提供的查询系统。消费者可以根据Tag实现

对不同子主题的不同消费的处理逻辑，实现更好的扩展性。

·UserProperties：用户自定义的属性集合，属于Message的一部分。

·ProducerGroup：同一类Producer的集合，这类Producer发送同一类消息逻辑一致。

如果发送的事务消息且原始生产者在发送之后崩溃，则Broker服务器会联系同一生

产者组的其他生产者实例已提交或回溯消息。

·ConsumerGroup：同一类Consumer的结合，这类Consumer通常消费同一类消息且消

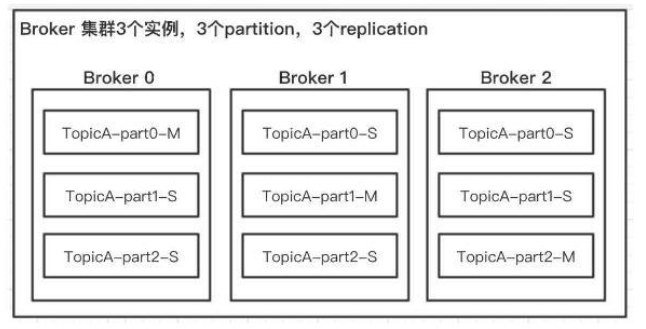
费逻辑一致。消费者组使得在消息消费方面，实现负载均衡和容错的目标变得非常容

易。要注意的是，消息者组的消费者实例必须订阅完全相同的Topic。

## 为什么放弃ZooKeeper而选择NameServer

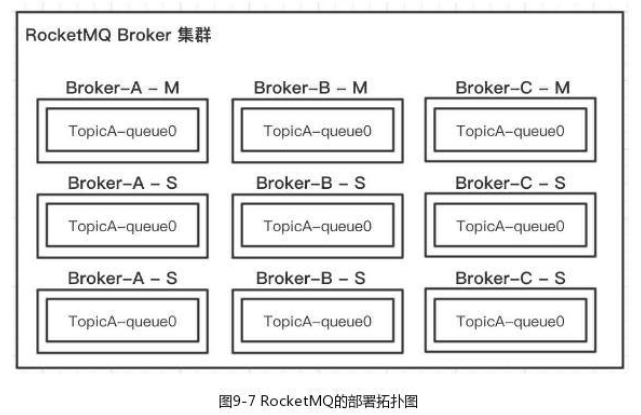
在Kafka的服务注册与发现通常是用ZooKeeper来完成的，RockeMQ早期也使用了ZooKeeper做集群的管理，但后来放弃了转而使用自己开发的NameServer。那为什么要放弃已有的ZooKeeper反而再写一个服务注册中心呢？

在Kafka中，Topic是逻辑概念，分区（Partition）是物理概念。1个Topic可以设置多个分区，每个分区可以设置多个副本（Replication），即有一个Master分区、多个Slave分区。Kafka的部署拓扑图如下所示：



例如，搭建3个Broker构成一个集群，创建一个Topic取名为TopicA，分区是3个，副本数是2个、在上图中part表示分区，M表示Master、S表示Slave。在Kafka中消息只能发送到Master分区中，消息发送给Topic时会发送到具体某个分区。如果发送给part0就只会发送到Broker0这个实例，再由这个实例同步到Broker21Broker2的part0副本中；如果发送给part1就只会发送到Broker1这个实例，再由Broker1同步到Broker0和Broker2的part1副本中

在RocketMQ中，Topic也是逻辑概念，队列（Queue）是物理概念（对应Kafka中的分区）。一个Topic可以设置多个队列，每个队列也可以有多个副本，即有一个Master队列、多个Slave队列。RocketMQ的部署拓扑图如下所示：



为了方便与Kafka对比，同样创建了一个Topic取名为TopicA，队列是3个，副本数也是2个，但构成Broker集群的实例有9个。

Kafka与RocketMQ两者在概念上相似，但又有明显的差异。如下：

·在Kafka中，Master和Slave在同一台Broker机器上，Broker机器上有多个分区

，每个分区的Master/Slave身份是在运行过程中选举出来的。Broker机器具有双

重身份，既有Master分区，也有Slave分区。

·在RocketMQ中，Master和Slave不在同一台Broker机器上，每台Broker机器不

是Master就是Slave，Brokr的Master/Slave身份是在Broker的配置文件中预先

定义好的，在Broker启动之前就已经决定了。

这个差异的影响在哪呢？Kafka的Master/Slave需要通过ZooKeeper选举出来，而

RocketMQ不需要。问题就在这个选举上，ZooKeeper具备选举功能，选举机制的原理就是少数服从多数，那么ZooKeeper的选举机制必须由ZooKeeper集群中的多个实例共同完成。ZooKeeper集群中的多个实例必须相互通信，如果实例数很多，网络通信就会变得非常复杂且低效。NameServer的设计目标是让网络通信变简单，从而使性能得到极大的提升。为了避免单点故障，NameServer也必须以集群的方式部署，但集群中各实例间相互不进行网络通信。NameServer是无状态的，可以任意部署多个实例。Broker向每一台NameServer注册自己的路由信息，因此每一个NameServer实例都保存一份完整的路由信息。NameServer与每台Broker机器保持长连接，间隔30s从路由注册表中将故障机器移除。NameServer为了降低实现的复杂度，并不会立刻通知客户端的Producer和Consumer。

## 集群环境中需要考虑的场景

集群环境下实例很多，偶尔会出现各种各样的问题，以下几个场景需要大家思考：

·当某个NameServer因宕机或网络问题下线了，Broker如何同步路由信息？由于

Broker会连接NameServer集群的每一个实例，Broker仍然可以向其他NameServer

同步其路由信息，Produce和Consumer仍然可以动态感知Broker的路由信息。

·NameServer如果检测到了Broker宕机怎么办？Consumer从Broker订阅消息失

败怎么办？RocketMQ作者为了简化NameServer的设计，这两个问题都是在客户

端中解决的，具体将在本章后续“高可用设计”章节中解答。

·由于NameServer集群中的实例相互不通信，在某个时间点不同NameServer实例

保存的路由注册信息可能不一致。这对发送消息和消费消息并不会有什么影响，

原理和上一个问题是一样的，从这里能看到NameServer是CAP中的AP架构。

# 如何实现顺序消息

## 顺序消息的使用场景

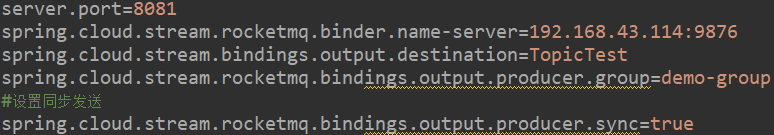
日常中需要保证顺序的应用场景非常多，也就是顺序消息和使用场景非常多。例如交易系统中的订单创建、支付、退款等流程，先创建订单才能支付，支付完成才能退款，这需要保证先进先出（First In First Out，缩写FIFO）。又例如数据库的BinLog消息（BinLog是记录所有数据库表结构变更（例如CREATE、ALTER TABLE…）以及表数据修改（INSERT、UPDATE、DELETE…）的二进制日志。），数据执行新增语句，修改语句，BinLog消息的顺序也必须保证是新增消息、修改消息。

## 如何发送和消费顺序消息

使用RocketMQ顺序消息来模拟一下订单的场景，顺序消息分为两部分：顺序发送（发送消息）、顺序消费（接收消息）。基于前面的案例进行改造。

**消息生产者**

1. 配置application.properties文件



指定producer.sync=true，默认是异步发送，将值改为false后表示同步发送。

1. 生产消息

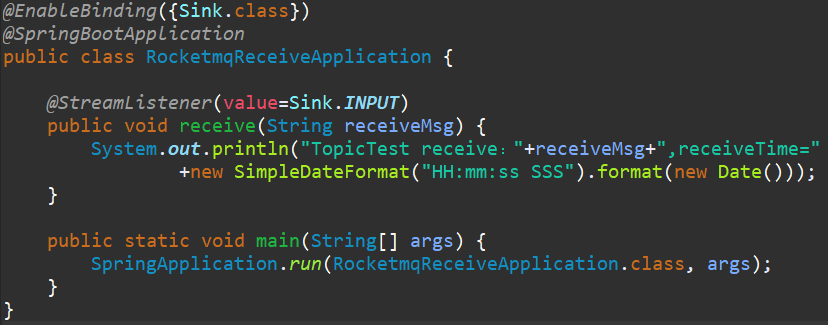


为了简化代码，模拟按顺序依次发送创建、支付、退款消息到TopicTest。

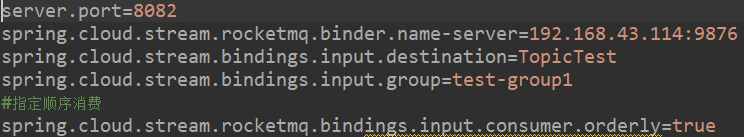
MessageBuilder设置Header消息头，表示这是一条顺序消息，将消息固定地发送到第0个消息队列。

**消息消费者**

1. 定义监听器接收消息

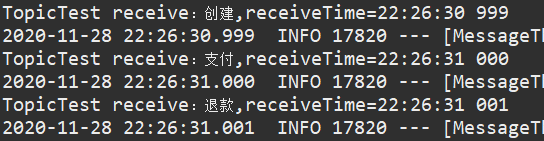


1. 指定顺序消费



consumer.orderly=true表示顺序消费，默认是并发消费。

1. 运行程序之后和生产消息后查看消费者控制台的日志输出，也是按顺序打印出来的：



## 顺序消息的两种情况

RocketMQ的顺序消息分为2种情况：局部有序和全局有序。上节中实现的顺序消息属于局部有序场景。

·局部有序：指发送到同一个队列的消息有序，可以在发送消息时指定队列，在消

费消息时也按顺序消费。例如同一个订单ID的消息要保证有序，不同订单ID的

消息没有约束，相互不影响，不同订单ID之间的消息时并行的。

·全局有序：设置Topic只有1个队列可以实现全局有序，创建Topic时手动设置。

此类场景极少，且性能差，通常不推荐使用。

## 发送消息的方式

RocketMQ中发送消息有三种方式：同步、异步、单向。

·同步：发送网络请求后同步等待Broker服务器的返回结果（会阻塞当前线程），

支持发送失败重试，适用于较重要的消息通知场景。

·异步：异步发送网络请求，不会阻塞当前线程，不支持失败重试，适用于对响应

时间要求更高的场景。

·单向：单向发送原理和异步一致，但不支持回调。适用于响应时间非常短、对可

靠性要求不高的场景，例如日志收集。

## 顺序发送的技术原理

顺序消息发送的原理很简单，同一类消息发送到相同的队列即可。为了保证先发送的消

息先存储到消息队列，必须使用同步发送的方式，否则可能出现先发的消息后到消息队列，导致消息乱序。在队列列表的获取过程中，由Procucer从NameServer根据Topic查询Broker列表，缓存在本地内存中，以便下次从缓存中读取。（关于消息发送的技术原理，教程书对源码进行了分析）

## 普通发送的技术原理

RocketMQ中除了顺序消息外，还支持事务消息和延迟消息，除了这三种特殊的消息外，

其他消息我们称为普通消息，以便区分。日常开发过程中最常用的是普通消息，这是因

为最常用的场景是系统间的异步解耦和流量的削峰填谷，这些场景下尽量保证消息高性

能收发即可。

从普通消息与顺序消息的对比来看，普通消息在发送时选择消息队列的策略不同。普通

消息发送选择队列有两种机制：轮询机制和故障规避机制（也称故障延迟机制）。默认

使用轮询机制，即一个Topic有多个队列，轮询选择其中一个队列。轮询机制的原理是由信息TopicPublishInfo中维护一个计数器sendWhichQueue，每发送一次消息需要查询一次路由，计数器就进行“+1”，通过计数器的值index与队列的数量取模来实现轮询算法（书上有源码解释）。轮询算法简单好用，但有个弊端。如果轮询选择的队列是在宕机的Broker上，会导致消息发送失败，即使消息发送重试的时候重新选择队列，也可能还是在宕机的Broker上，无法规避发送失败的情况，因此就有了故障规避机制的消息发送策略（后续章节会纤细讲解）。

## 消息模式

RocketMQ支持两种消息模式：集群消费（Clustering）和广播消息（Broadcasting）。两者的区别是：在广播消费模式下每条消息会被ConsumerGroup的每个Consumer消费，在集群消费模式下每条消息只会被ConsumerGroup的一个Consumer消费。

多数场景下都使用集群消费，消息每消费一次代表一次业务处理，集群消费表示每条消息由业务应用集群中任意一个服务实例来处理。少数场景下使用广播消费，例如数据发送变化，更新业务应用集群中每个服务的本地缓冲，这就需要一条消息整个集群中的每个服务都消费一次。默认情况下是集群消费。

## 顺序消费的技术原理

顺序消费也称为有序消费，原理是同一个消息队列只允许Consumer中的一个消费线程拉取消费。Consumer中有多个消费线程池，多个线程会同时消费消息。在顺序消费的场景下消费线程请求到Broker时会申请独占锁，获得锁的请求则允许消费（书上有源码分析）。消息消费成功后，会向Broker提交消费进度，更新消费位点信息，避免下次拉取到已消费的消息。顺序消费中如果消费线程在监听器中进行业务处理时抛出异常，则不会提交消费进度，消费进度会阻塞在当前这条消息，如果重试也失败，会一直阻塞在当前消息，直到超出最大重试次数，从而在很长一段时间内无法消费后续消息导致队列消息堆积。

## 并发消费的技术原理

RocketMQ支持两种消费模式：顺序消费和并发消费。其中并发消费是默认的消费方式，日常开发过程中最常用的方式。并发消费也称为乱序消费，其原理是同一个消息队列提供给Consumer中的多个消费线程拉取消费。Consumer中会维护一个消费线程池，多个消费线程池可以并发去同一个消费队列中拉取消息进行消费。如果某个消费线程在监听中进行业务处理时抛出异常，当前消费线程拉取的消息会进行重试，不影响其他消费线程和消费队列的消费进度，消费成功的线程正常提交消费进度。并发消费相比顺序消息没有资源抢上锁的过程，消费消息的速度比顺序消费要快的多。

## 消息的幂等性

说到消息消费就不得不提到消息的幂等性，业务代码中通常收到一条消息进行一次业务逻辑处理，如果一条相同消息被重复收到几次，是否导致业务重复处理？Consumer能够不重复接收消息？RocketMQ并不保证消息不被重复消费，如果业务对消费消息非常敏感，必须要在业务层面进行幂等性处理，具体实现可以通过分布式锁来完成。

在所有消息系统中消息消费有三种模式：at-most-once（最多一次）、at-least-once（最少一次）和exactly-only-once（精确仅一次），分布式消息系统都是在三者间取平衡，前两者是可行的并且被广泛使用的。

·at-most-once：消息投递后不论消费是否成功，不会再重复投递，有可能会导致

消息未被消费，RocketMQ未使用该方式。

·at-least-once：消息投递后，消费完成后，向服务器返回ACK（消费确认机制），

没有消费则一定不会返回ACK信息，服务器则会再次投递，可能会导致重复消

费，RocketMQ通过ACK来却表消息至少被消费一次。

·exactly-only-once：必须下面两个条件都满足，才能认为消息是“ExactlyOnly”

1. 发送消息阶段，不允许发送重复的消息
2. 消息消费阶段，不允许消费重复的消息

在分布式系统环境下，如果要实现该模式，巨大的开销不可避免。RocketMQ为

了追求高性能，并不保证此特性，无法避免消息重复，所有需要由业务上进行幂

等性处理。

# 如何实现事务消息

## 事务消息的使用场景

事务消息的使用场景很多，例如，在电商系统中用户下单后新增了订单记录，对应的商品库存需要减少，怎么保证新增订单后商品库存减少呢？又例如红包业务，张三给李四发红包，张三的账户余额需要扣减，李四的账户余额需要增加，怎么保证张三账户扣钱后李四账户价钱呢？此类问题都属是事务问题，可以简单理解为：一个表的数据更新后，如果保证另外一个表的数据也更新成功。如果使用的同一个数据库实例，那么问题很简单，可以使用本地事务来解决，Spring的@Transaction注解就支持。但实际场景并不这么简单，互联网应用的流量大，系统规模通常也比较大，会存在许多数据库实例、分库分表等。我们需要修改的表往往不在同一个数据库实例或同一个数据库中，此时就不能使用本地事务来解决了，需要用到分布式事务。RocketMQ的一大特点就是支持事务消息，支持一些分布式事务场景。

## 如何发送事务消息

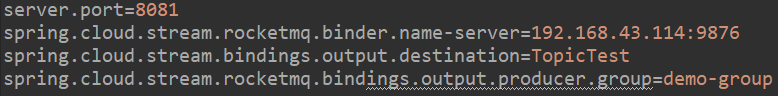
使用RocketMQ事务消息模拟下单减库存的场景，为了简化代码，部分非核心的实现仅用注释说明。

实例基于前面的项目进行改造。

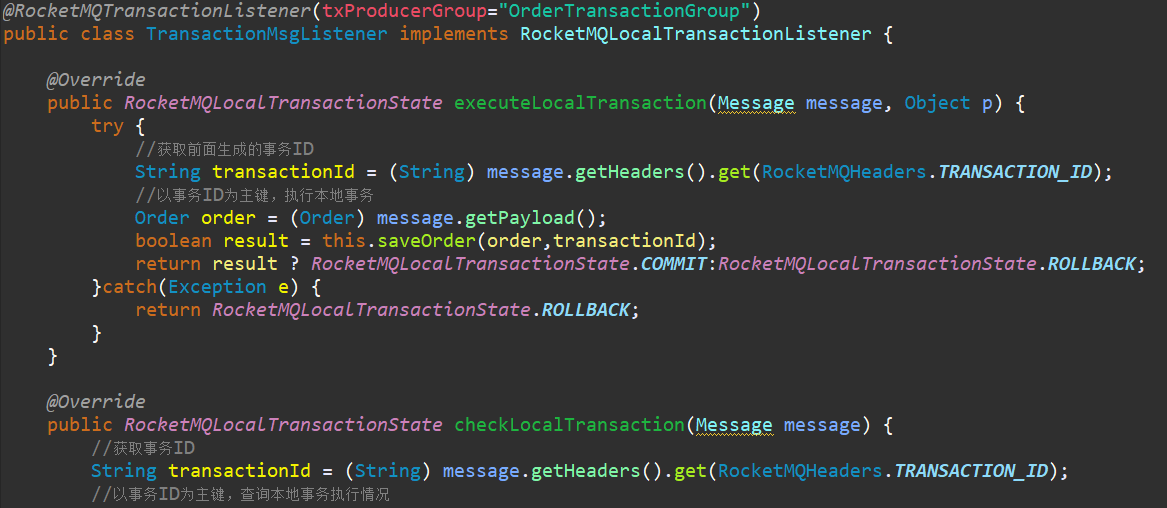
1. 发送订单的事务消息，预提交

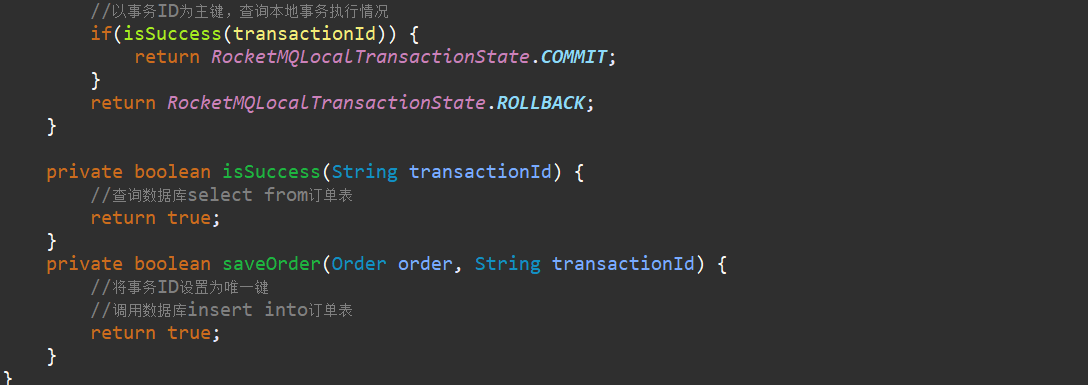


Order对象保存了订单信息，随机生成一个ID作为消息的事务ID，定义了一个名为OrderTransactionGroup的事务组，用于下一步接收本地事务的监听。



1. 执行订单信息入库的事务操作，提交或回滚事务消息。





实现RocketMQLocalTransactionListener接口，使用@RocketMQTransactionListener注解用于接收本地事务的监听，txProducerGroup是事务组名称，和前面定义的OrderTransationGroup保持一致。RocketMQLocalTransactionListener有两个实现方法。

·executLocalTransaction：执行本地事务，在第1步中消息发送成功会回调执行，

一旦事务提交成功，下游应用的Consumer能收到该消息，在这里demo的本地

事务就是保存订单信息入库。

·checkLocalTransaction：检查本地事务执行状态，如果executeLocalTransaction方

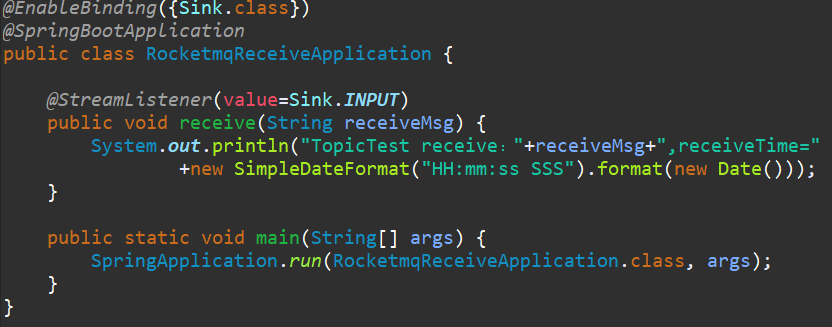
法中返回的状态是未知UNKNOWN或者未返回状态，默认会在预处理发送的1

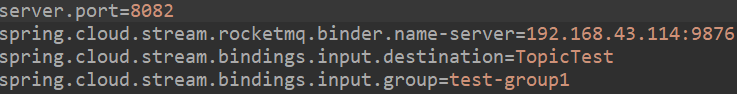
分钟后由Broker通知Producer检查本地事务，在Producer中回调本地事务监听

器中的checkLocalTransaction方法。检查本地事务时，可以根据事务ID查询本地

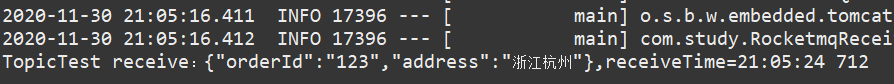
事务的状态，再返回具体事务状态给Broker。

1. 消费订单消息





1. 启动NameServer和Broker，生产消息后查看消费者后台打印：



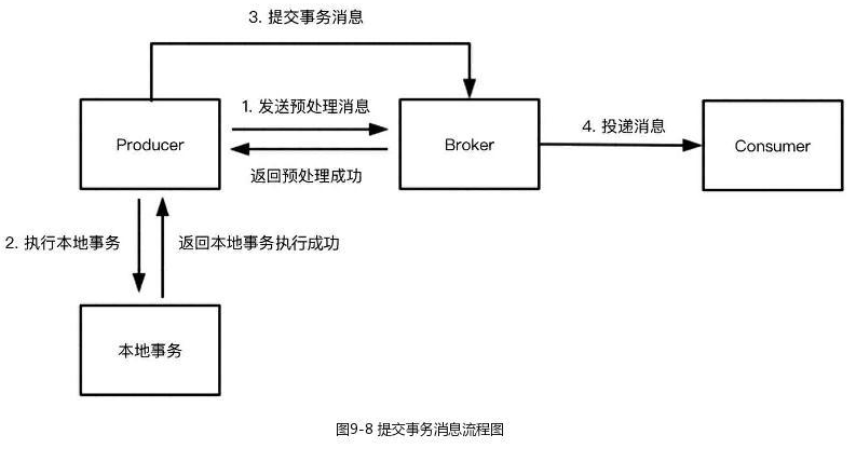
## 事务消息的技术原理

RocketMQ采用了2PC的方案来提交事务信息。第一阶段Producer向Broker发送预处理信息（也称半信息），此时消息还未被投递出去，Consumer不能消费；第二阶段Producer向Broker发送提交或回滚信息。具体流程如下：

·发送预处理消息成功后，开始执行本地事务

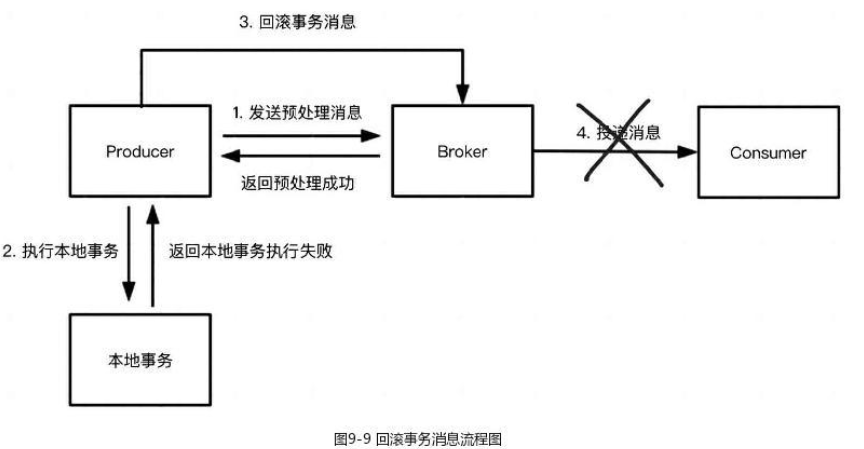
·如果本地事务执行成功后，发送提交请求提交事务消息，消息会投递给Consum

er，如下图所示：



·如果本地事务执行失败，发送回滚请求回滚事务消息，消息不会投递给Consum

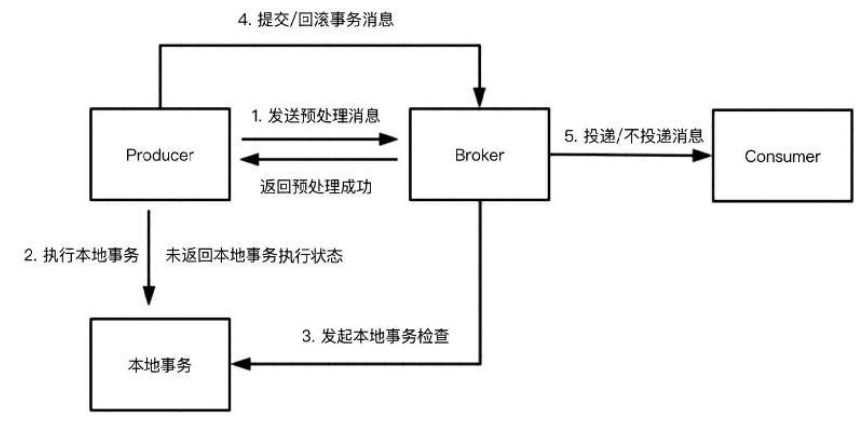
er，如下图所示：



·如果本地事务状态未知，网络故障或Producer宕机，Broker未收到二次确认的

消息。由Broker端发送请求给Producer进行消息回查，确认提交或回滚。如果

消息状态一致未被确认，需要人工介入处理，如下：



## 疑问

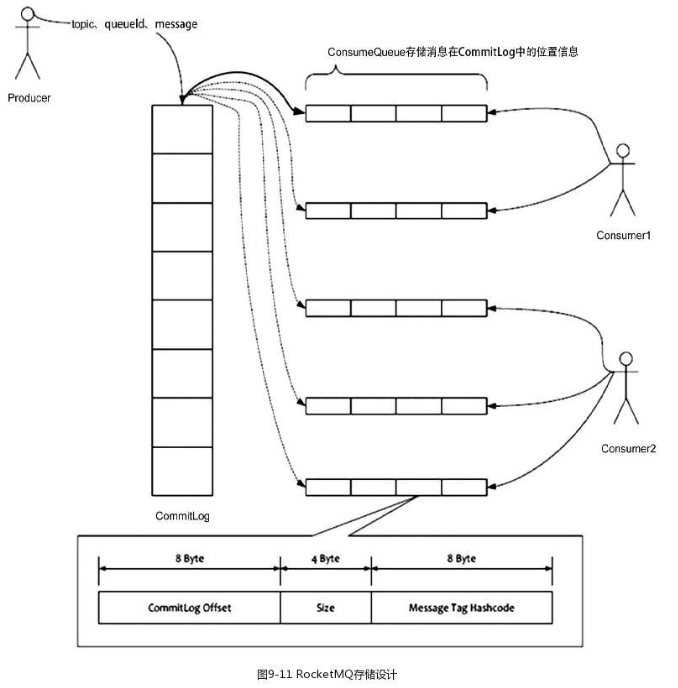
学习完RocketMQ的事务消息原理，心中存在一个疑问：当Producer本地事务提交成功后，Broker会通知Consumer消费消息，但如果此时Consumer消费消息失败，Producer中执行的DB操作如何进行回滚呢？

# 高性能设计

## RocketMQ高性能的体现

经过阿里巴巴多年双11的验证，RocketMQ在稳定的基础上一直保持着非常高的性能，这是诸多企业在消息中间件方面选择使用RocketMQ的重要原因。RocketMQ的高性能设计体现在三个方面：数据存储设计、动态伸缩的能力、消息实时投递。其中，数据存储设计包括顺序写盘、消费队列设计、消息跳跃读取、数据零拷贝。动态伸缩的能力包括消息队列扩容、Broker集群扩容。

RocketMQ以高吞吐量著称，主要得益于其数据存储方式的设计。数据存储的核心由两部分组成：CommitLog数据存储文件和ConsumeQueue消费队列文件。Producer将消息发送到Broker服务器，Broker会把所有消息都存储在CommitLog文件中，再由CommitLog转发消息到ConsumeQueue文件提供给各个Consumer消费。RocketMQ存储设计如下图所示：



## 顺序写盘

顺序写盘指写磁盘上的文件采用顺序写的方式，在解释为什么要采用顺序写盘之前，我们先简单了解一下磁盘读写的过程。一次磁盘请求（读或写）完成过程由三个动作组成：寻道、旋转延迟、数据传输。

·寻道：磁头移动定位到指定磁道，时间很长，找到数据在哪个地方

·旋转延迟：等待指定扇区旋转至磁头下，机械硬盘和每分钟多少转有关系，时间

很短

·数据传输：数据通过系统总线从磁盘传送到内存中，时间很短

磁盘读写最慢的动作是寻道，缩短寻道时间就能有效提升磁盘的读写速度，最优的方式就是不用寻道。随机写入会导致磁头不停的更换磁道，时间都花在寻道上了，而顺序写几乎不用换磁道，或者寻道时间很短。

CommitLog文件是负责存储消息数据的文件，所有Topic的消息都会先存在${ROCKETMQ\_HOME}/store/commitlog文件夹下的文件中（可以查看验证，注意store目录可能不在RocketMQ安装目录而是在用户家目录），消息数据写入CommitLog文件是加锁串行追加写入。RocketMQ为了保证消息发送的吞吐量，使用单个文件存储所有Topic的消息，从而保证消息存储是完全的磁盘顺序写，但这样给文件读取（消费消息）带来了困难，当消息到达CommitLog文件后，会通过线程异步几乎实时地将消息转发给消费队列文件。每个CommitLog文件的默认大小是1GB，写满1GB再写新的文件，大量数据I/O都是在顺序写同一个CommitLog文件。文件名按照该文件起始的总的字节偏移量offset命名，文件名固定长度为20位，不足20为前面补0。如下：

·第一个文件起始偏移量是0，即文件名是00000000000000000000

·第二个文件起始偏移量是1024×1024×1024=1073741824（1GB=1073741824B）

，即文件名是00000000001073741824



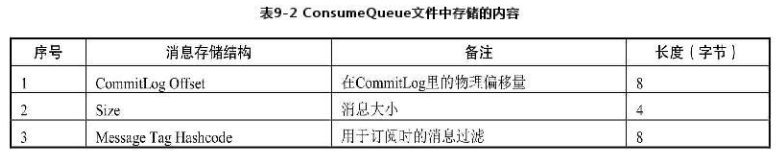
文件名这样设计的目的是在消费消息时能够根据偏移量offset快速定位到消息存储在哪个CommitLog文件，从而加快消息的检索速度。消息数据文件中每条消息数据的具体格式如下表所示：



## 消息队列设计

消费Borker中存储的消息实际上就是读取文件，但消息数据文件中所有Topic的消息数据是混合在一起的，而消费消息时是区分Topic消费的，这就导致如果消费时也读取CommitLog文件会使得消费消息的性能差、吞吐量低。为了解决消息数据文件顺序写难以读取的问题，RocketMQ中设计了消费队列ConsumeQueue。

ConsumeQueue负责存储消费队列文件，在消息写入CommitLog文件时，会异步转发给ConsumeQueue文件（在RocketMQ安装目录或用户家目录下的store/consumequeue文件夹中），然后提供给Consumer消费。ConsumeQueue文件中并不存储具体的消息数据，只存CommitLog的偏移量、消息大小size、消息Tag Hashcode，如下表所示：



每个Topic在某个Broker下对应多个队列Queue，默认是4个消费队列Queue。每一条记录的大小是20B，默认一个文件存储30万个记录，文件名同样也按照字节偏移量offset命名（上节讲过），文件名固定长度为20位，不足20位前面补0.

·第一个文件起始偏移量是0，文件名是000000000000000000，与CommitLog一

致

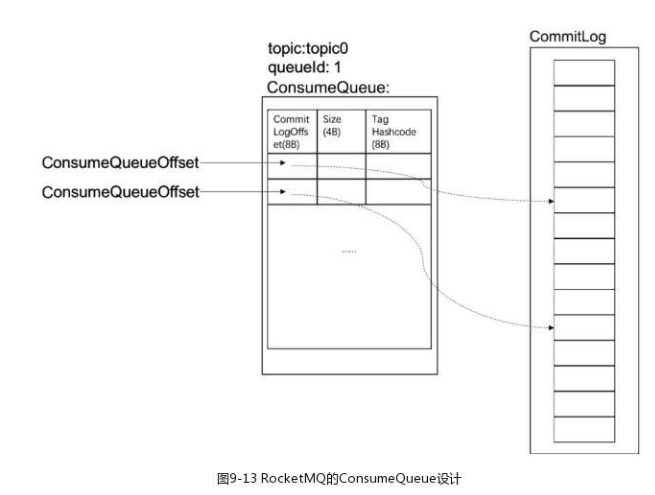
·第二个文件起始偏移量是20×30w=6000000，第二个文件名是00000000000600

0000

在集群模式下，Broker会记录客户端对每个消费队列的消费偏移量，定位到CommitLog

里相应的记录，并通过CommitLog的offset定位到CommitLog文件里的该条消息，如

下图所示：



## 消息跳跃读取

消费Broker中存储的消息实际上就是读取文件，消息队列文件是一种数据结构上的设计。前面讲了磁盘顺序读写和消息队列文件的设计，为了高性能读数据，除此之外，还使用了操作系统中的PageCache机制。RocketMQ读取消息依赖操作系统的PageCache，PageCache命中率越高则性能越高，操作系统会尽量预读数据，使应用直接访问磁盘的概率降低。消息队列文件的读取流程如下：

·检查要读的数据是否在上次预读的Cache中。

·如果没有命中Cache，操作系统从磁盘中读取对应的数据页，并将该数据页之后

的连续页一起读入Cache中，再将应用需要的数据返回给应用，这种方式称为跳

跃读取

·如果命中Cache，上次缓存的数据有效，操作系统认为在顺序读盘，则继续扩大

缓存的数据范围，将之前缓存的数据页的几页再读取到Cache中。

在计算机系统中，CPU、RAM、DISK的速度不相同，按速度高低排列为：CPU>RAM>DISK。CPU与RAM之间、RAM与DISK之间的速度和容量差异是指数级的。为了在速度和容量上折中，在CPU和RAM之间使用CPU Cache以提高访存速度，在RAM与磁盘之间，操作系统使用Page Cache提高系统对文件的访问速度。

## 数据零拷贝

在网络通信中，通常情况下对文件的读写要多经历一次数据拷贝，例如写文件数据要从用户态拷贝到内核态，再由内核态写入物理文件。所谓零拷贝，指的是用户态与内核态之间不存在拷贝。RocketMQ的文件读写主要通过Java NIO中的MapperByteBuffer来进行文件映射。利用Java NIO中的FileChannel模型，可以直接将物理文件映射到缓冲区Page Cache，少了一次数据拷贝过程，提高了读写速度。

## 动态伸缩能力

随着业务的增长，线上流量会出现快速增长，经常出现的情况是已有的服务器集群能力不足以支撑现有的流量，此时就需要增加服务器（扩容）。例如大促、秒杀等活动，流量上涨持续一段时间又会回归到正常情况，为了避免服务器资源（成本）浪费，此时就需要减少服务器（缩容），这些场景都会用到RocketMQ中的动态伸缩能力。

动态伸缩（水平扩容）能力是分布式应用很重要的能力，RocketMQ中的动态伸缩能力主要体现在消息扩容和集群扩容两个方面，需要根据实际场景进行选择。

·消息队列扩容/缩容：一个Consumer实例可以同时消费多个消息队列的消息。如

果一个Topic的消息量特别大，但Broker集群水位压力还是很低，就可以对该Topic

的消息队列进行扩容，Topic的消息队列数跟消费速度成正比。消息队列数在创

建Topic时可以指定，也可以在运行中修改。相反，如果一个Topic的消息量特

别小，但该Topic的消息队列数很多，则可以对该Topic消息队列缩容。

·Broker集群扩容/缩容：同样如果一个Topic的消息量特别大，但Broker集群水

位很高，此时就需要对Broker机器扩容，扩容方式很简单，直接加机器部署Broker

即可。新Broker启动后会向NameServer注册，Producer和Consumer通过

NameServer发现新Broker并更新路由信息。相反，如果Broker集群水位很低，

则可以适当减少Broker服务器来节约成本。

## 消息实时传递

消息实时传递指的是消息发送到Broker之后能够立即被消费，而Consumer消费消息的实时性与获取消息的方式有很大关系。任何一款消息中间件都会有两种获取消息的方式：Push推模式和Pull拉模式。这两种模式各有优缺点，适用于不同的场景。

1. Push推模式：当消息发送到服务器端时，由服务器主动推送给客户端Consumer。优点是客户端Consumer能实时地接收到新的消息数据。但也有两个缺点，缺点1是如果Consumer消费一条消息耗时很长，服务端推送速度大于消费速度时，Consumer消费不过来就会出现缓冲区溢出；缺点2是一个Topic往往会对应多个ConsumerGroup，服务端一条消息会产生多次推送，可能会对服务端造成压力。
2. Pull拉模式：由客户端Consumer主动发送请求，每间隔一段时间轮询去服务端拉取信息。优点是Consumer可以根据当前消费速度选择合适的时机触发拉取。缺点是拉取的间隔时间不好控制，间隔时间如果很长，会导致消息消费不及时，服务端容易积压消息；间隔如果时间很短，服务端收到的消息少，会导致Consumer可能多数拉取请求都是无效的（拿不到消息），从而浪费网络资源和服务端资源。

这两种获取信息方式的缺点都很明显，单一的方式难以应对复杂的消费场景，所以RocketMQ中提供了一种推/拉结合的长轮询机制来平衡推/拉模式各自的缺点。RocketMQ中的长轮询本质上是对普通pull模式的优化，即还是以客户端Consumer轮序的方式主动发送拉取请求到服务端Broker，Broker如果检测到有新的消息就立即返回Consumer，但如果没有新消息则暂时不返回任何消息，挂起当前请求缓存到本地，Broker后台有个线程去检查挂起请求，等到消息产生时再返回Consumer。平常使用的DefaultMQPushConsumer的实现就是推、拉结合的，既能解决资源浪费问题，也能解决消费不及时问题。

# 高可用设计

## RocketMQ高可用的体现

计算机系统的可用性用平均无故障时间来度量，系统的可用性越高，则平均无故障时间越长。高可用性也是分布式中间件的重要特性，RocketMQ的高可用设计主要有四个方面的体现：

·消息发送的可用性：消息发送时可能会遇到网络问题、Broker宕机等情况，而

NameServer检测Broker是有延迟的，虽然NameServer每间隔10秒会扫描所有Broker信息，但要Broker的最后心跳时间超过120秒以上才会认为该Broker不可用，所以Producer不能及时感知Broker下线。如果在这期间消息一直发送失败，那么消息发送失败率会很高，这在业务上是无法接受的。那为什么NameServer不及时检查Broker和通知Procucer呢？这是因为那样做会使网络通信和架构设计变得非常复杂，而NameServer的设计初衷就是尽可能简单，所以这块高可用方案是Producer中来实现。RocketMQ采用了一些发送端的高可用方案，来解决发送失败的问题，其中最重要的两个设计是重试机制与故障延迟机制。

·消息存储的高可用：在RocketMQ中消息存储的高可用体现在发送成功的消息不能丢

、Broker不能发生单点故障，出现Broker异常宕机、操作系统Crash、机房断电或断网等情况保证数据不丢。RocketMQ主要通过消息持久化（也称刷盘）、主从复制、读写分离机制来保证。

·消息消费的高可用：实际业务场景中无法避免消费消息失败的情况，可能由于网络原

因导致，也可能由于业务逻辑错误导致。但无论发生任何情况，即使消息一直消费失

败，也不能丢失消息数据。RocketMQ主要通过消费重试机制和ACK机制来保证。

·集群管理的高可用：集群管理的高可用主要体现在NameServer的设计上，当部分

NameServer节点宕机时不会有什么糟糕的影响，只剩一个NameServer节点RocketMQ

集群也能正常运行，即使NameServer全部宕机，也不影响已经允许的Broker、Producer

和Consumer。

## 消息发送重试机制

在消息发送出现异常时会尝试再次发送，默认最多重试三次。重试机制仅支持同步发送方式，不支持异步和单向发送方式。根据发送失败的异常类型处理策略也有不同，如果是网络异常RemotingException和客户端异常MQClientException会重试，而Broker服务端异常MQBrokerException和线程中断异常InterruptedException则不会重试，且抛出异常。（书上有源码分析）

## 故障规避机制

前面讲过，NameServer为了简化和客户端通信，发现Broker故障时并不会立即通知客户端，会有延迟。故障规避机制用来解决Broker故障，Producer不能及时感知而导致消息发送失败的问题。故障规避机制默认是不开启的，如果在开启的情况下，消息发送失败的时候会将失败的Broker暂时排除在队列选择列表外。规避时间是衰减的，如果Broker一直不可用，会被NameServer检测到并在Procucer更新路由信息时进行剔除。（书上有源码分析）

## 同步刷盘和异步刷盘

刷盘是指消息数据发送到Broker之后，写入磁盘中做持久化，保障在Broker出现故障重启时数据不会丢失。RocketMQ提供了两种刷盘机制：同步刷盘和异步刷盘。两者在性能和使用场景上有明显区别。

**同步刷盘**

在同步刷盘的模式下，当消息写到内存后，会等待数据写到磁盘的CommitLog文件。具体实现代码在CommmitLog#handleDiskFlush（即CommitLog类中的handleDiskFlush方法）中。GroupCommitRequest是刷盘任务，提交刷盘任务后，会在刷盘队列中等待刷盘，而刷盘线程GroupCommitService每间隔10ms写一批数据到磁盘。至于为什么不直接写，主要原因是磁盘I/O压力大、写入性能低，每间隔10毫秒写一次可以提升磁盘I/O效率和写入性能。（书上有源码分析）

**异步刷盘**

RocketMQ默认采用异步刷盘，异步刷盘又有两种策略：开启缓冲池和不开启缓冲池。

1. 不开启缓冲池：默认为不开启缓存池，刷盘线程会间隔500ms尝试去刷盘，这间隔500ms仅仅是尝试，实际去刷盘还得满足一些前提条件，即距离上次刷盘时间超过10秒或者写入内存的数据超过4页（16KB），这样即使服务器宕机，丢失的数据也是在10秒内的或大小在16KB以内的（书上有源码分析）。
2. 开启缓冲池：RocketMQ会申请一块和CommitLog文件相同大小的堆外内存用来做缓冲池，数据会先写入缓冲池，提交线程也每间隔500ms尝试提交到文件通道等待刷盘，刷盘最终还由FlushRealTimeService来完成，和不开启缓冲池的处理一致。使用缓冲池的目的是多条消息合并写入，从而提高I/O性能（书上有源码分析）。

在实际应用中需要结合业务场景，合理设置刷盘方式，不建议使用SYNC\_FLUSH（同步刷盘），因为它会频繁地触发写操作，性能下降很明显。通常使用ASYNC\_FLUSH异步刷盘。

## 主从复制

RocketMQ为了提高消息消费的可用性，避免Broker发生单点故障引起存储在Broker上的消息无法及时消费，同时避免单个机器上硬盘损坏出现消息数据丢失。RocketMQ采用Broker数据主从复制机制，当消息发送到Master服务器后会将消息同步到Slave服务器，如果Master服务器宕机，消息消费者还可以继续从Slave拉取消息。

消息从Master服务器复制到Slave服务器上，有两种复制方式：同步复制SYNC\_MASTER和异步复制ASYNC\_MASTER。通过配置文件${ROCKETMQ\_HOME}/conf/broker.conf文件中的brokerRole参数设置。

·同步复制：Master服务器和Slave服务器都写成功后才返回给客户端写成功的状

态。优点是如果Master服务器出现故障，Slave服务器上有全部备份的数据，很容易恢复到Master服务器。确定是由于多了一个同步等待的步骤，会增加数据写入延迟，并且降低系统的吞吐量。

·异步复制：仅Master服务器写成功即可返回给客户端写成功的状态。优点刚好

是同步复制的缺点，由于没有一次同步等待的步骤，服务器的延迟较低且吞吐

量高。缺点是如果Master服务器出现故障，有些数据因为没有被写入Slave服务

器，未同步的数据有可能会丢失。

在实际应用中需要结合业务场景，合理主从复制方式，通常可以把Master和Slave设置成SYNC\_MASTER同步复制，这样即使有一台服务器出现故障，仍然可以保证数据不丢失。

## 读写分离

读写分离机制也是高性能、高可用架构中常见的设计。例如MySQL也实现了读写分离机制，Client只能从Master服务器写数据，可以从Master服务器和Slave服务器读数据。RocketMQ设计也是如此，但在实现方式上有一些区别：RocketMQ的Consumer在拉取消息时，Broker会判断Master服务器的消息堆积量以决定Consumer是否从Slave服务器拉取消息消费。默认一开始从Master服务器上拉取消息，如果Master服务器的消息堆积超过了物理内存的40%，则会在返回给Consumer的消息结果中告知Consumer，下次拉取消息需要去其他Slave服务器上拉取消息。

## 消息重试机制

实际业务场景中无法避免消息消费失败的情况，消费失败可能是因为业务处理中调用远程服务网络问题失败，不代表消息一定不能被消费，通过重试可以解决。在介绍RocketMQ的消息重试机制之前，有必要先了解一下“重试队列”和“死信队列”的概念。

重试队列：Consumer消费消息失败时，将消费失败的消息重新发送给Broker

保存在重试队列。这样的设计的原因是不能影响整体消费进度又必须防止消费失败的消息丢失。重试队列的消息存在一个单独的Topic中，不在原消息的Topic中，Consumer自定订阅该Topic。重试队列的Topic名称格式为“%RETRY%+consumerGroup”，每个业务Topic都会有多个ConsumerGroup，每个ConsumerGroup消费失败的情况都不一样，因此各对应一个重试队列的Topic。

死信队列：由于逻辑Bug等原因，导致Consumer对部分消息长时间消费重试一直都是失败，为了保证者部分消息不丢失，同时不能阻塞其他能重试消费成功的消息，超过最大重试次数之后的消息会进入死信队列。消息进入了死信队列之后就不再自动消费，需要人工干预处理。死信队列也存在一个单独的Topic中，名称格式为“%DLQ%+consumerGroup”，原理和重试队列一致。

通常故障恢复需要一定的时间，如果不间断地重试，重试又失败的情况会占用并浪费资源，所以RocketMQ的消费重试机制采用时间衰减的方式，使用了定时消费的能力。首次在10秒后重试消费，如果消费成功则不再重试，如果消费失败则继续重试消费，在第二次30秒后重试消费，依此类推，每次重试的间隔时间都会加长，直到超出最大重试次数（默认为16次），则进入死信队列不再重试。重试消费过程中的间隔时间使用了定时消息，重试的消息数据并非直接写入重试队列，而是先写入定时消息队列，再通过定时消息的功能转发到重试队列。

## 定时消息

RocketMQ支持定时消息（也称延迟消息），延迟消息是指消息发送之后，等待指定的延迟时间后再进行消费。除了在消息重试机制中适用，延迟消息也适用于一些处理异步任务的场景。例如调用某个服务，调用结果需要异步在1分钟内返回，此时就可以发送一个延迟消息，延迟时间为1分钟，等1分钟后收到该消息去查询上次的调用结果是否返回。

RocketMQ不支持任意时刻精确的延迟消息，仅支持1s、5s、10s、30s、1min、2min、3min、4min、5min、6min、7min、8min、9min、10min、20min、30min、1h、2h。

## ACK机制

在实际业务场景中，业务应用在消费消息的过程中偶尔会出现一些异常情况，例如程序发布导致的重启，或网络突然出现问题，此时正在进行业务处理的消息可能消费完了，也可能业务逻辑执行到一半没有消费完，此时就需要依靠消息的ACK机制来识别这些情况。

广播消费的消费进度保存在客户端本地，集群模式的消费进度保存在Broker上。集群模式中RocketMQ中采用ACK机制确保消息一定被消费。在消息投递过程中，不是消息从Broker发送到Consumer就算消费成功了，而是需要Consumer明确给Broker返回消费成功状态才算成功。如果从Broker发送到Consumer后，已经完成了业务处理，但在给Broker返回消费成功状态之前，Consumer发送宕机或断点、断网情况，Broker未收到反馈，则不会保存消费进度。Consumer重启之后，消息会重新投递，此时也会出现重复消费的场景，前面的学习中讲过消息幂等性需要业务自行保证。

## Broker集群部署

Broker集群部署是消息存储高可用的基本保障，最直接的表现是Broker出现单机故障或重启时，不会影响RocketMQ整体的服务能力。RocketMQ中Broker一种非集群搭建方式和三种不同的集群搭建方式。

**单Master模式**

单Master模式仅部署一台Broker机器，属于非集群模式。这种方式存在单点故障的风险，一旦Broker重启或宕机，会导致整个服务不可用。不建议线上环境使用，仅可以用于本地测试。

**多Master模式**

一个集群全部都是Master机器，没有Slave机器，属于不配置主从复制的场景，例

如2个Master或者3个Master。也不建议线上环境使用，这种模式的优缺点如下：

·优点：配置简单，单个Master宕机或重启维护对应用没有影响，在磁盘配置为

RAID10时，即使机器宕机不可恢复，由于RAID10磁盘非常可靠，消息也不会丢

失（异步刷盘丢失少量信息，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

，消息的实时性受到影响。这个缺点是致命的，消息实时性受到影响意味着一段

时间内部分消费不可用，违背系统的可用性原则。

**异步复制的多Master多Slave模式**

每个Master配置一个Slave，有多对Master-Slave，主从复制采用异步复制方式，主备有短暂消息延迟（毫秒级），这种模式的优缺点如下：

·优点：即使磁盘损坏，消息丢失非常少，且消息实时性不会受影响，同时Master

宕机后，消费者仍然可以从Slave消费，而且此过程对应用透明，不需要人工干

预，性能同多Master模式几乎一样。

·缺点：在Master宕机且磁盘损坏的情况下可能丢失少量消息。出现这种场景的

概率很小，但风险还是有的。

**同步复制的多Master多Slave模式**

每个Master配置一个Slave，有多对Master-Slave，主从复制采用同步复制方式，即只有主备写成功，才向应用返回成功。线上推荐使用异步刷盘+同步复制的多Master多Slave模式，这种模式的优缺点如下：

·优点：数据与服务都无单点故障，在Master宕机的情况下，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（低10%左右），发送单个消息的RT（Response

Time，事务从请求到系统处理完成后返回结果的时间，即响应时间）会略高。

线上环境推荐使用异步刷盘+同步复制的多Master多Slave模式。