rcmq讲稿

以rcmq为例，如何学习一个新的中间件

1.csdn和官网，搜搜简介，初步了解这玩意是干什么的，它的常见应用场景：

比如rcmq是一个 **队列模型**的高实时高可靠高并发的分布式消息队列，场景应用场景是：解耦，削峰，异步

2.了解基础概念和架构：比如rcmq的结构：nameserver，broker，生产者消费者，rcmq的四种消息模型以及其应用场景。

3.搜搜入门安装教程，先尝试在云服务器部署它

4.实际上手实操一下

5.看进阶应用的教程，常见问题的解决（消息堆积，顺序消息，重复消费，消息不丢失），思考在自己项目中的应用场景

6.深入了解原理，结合源码，总结笔记（其实就是八股了）

官网-初识rcmq

定时/延时消息 | RocketMQ (apache.org)

入门-了解rcmq

RocketMQ保姆级教程 - 掘金 (juejin.cn)

rocketmq详解(全)-CSDN博客

安装使用-上手rcmq

RocketMQ的下载与安装（全网最细保姆级别教学）*rocketmq下载安装*舒一笑的博客-CSDN博客

RocketMQ保姆级教程 - 掘金 (juejin.cn)

项目应用拓展

八股-深入了解

RocketMQ常见问题总结 | JavaGuide(Java面试 + 学习指南)

rocketmq/docs/cn/FAQ.md at master · apache/rocketmq · GitHub

RocketMQ经典高频面试题大全（附答案） rocketmq面试题-CSDN博客

RocketMQ消息短暂而又精彩的一生 - 掘金 (juejin.cn)

RocketMQ 如何实现高性能消息读写？ - 掘金 (juejin.cn)

> 5. 当前业内有哪些MQ？为什么选择RocketMQ？
> 6. RocketMQ为什么具有高安全性？怎么保证的？
> 7. 如何解决MQ消息丢失的问题？

深入应用-封装

RocketMQ与SpringBoot整合进行生产级二次封装 - 掘金 (juejin.cn)

1.简介：是什么

> **首先消息队列是什么**:

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削锋等问题。实现高性能，

高可用，可伸缩和最终一致性架构。是大型分布式系统不可缺少的中间件。

目前在生产环境，使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ等

消息队列是一种"先进先出"的数据结构

> **rcmq是什么**

`RocketMQ` 是一个 **队列模型** 的消息中间件，具有**高性能、高可靠、高实时、分布式** 的特点。它是一个采用 `Java` 语言开发的分布式的消息系统

> 应用场景呢?

应用解耦

问题描述

```
系统的耦合性越高，容错性就越低，以电商应用为例，用户创建订单后，如果耦合调用库存系统、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
```

解耦

```
使用消息队列解耦，系统的耦合性就会下降了，比如物流系统发生故障，需要几分钟才能修复，在这段时间内，物流系统要处理的数据被缓存到消息队列中，用户的下单操作正常完成。当物流系统恢复后，补充处理存在消息队列中的订单消息即可，终端系统感知不到物流系统发生过几分钟故障
```

流量削峰

问题描述

```
应用系统如果遇到系统请求流量的瞬间猛增，有可能将系统压垮，有了消
息队列可以将大量请求缓存起来，分散到很长一段时间处理，这样可以大
大提高系统的稳定性
```

削峰含义

```
一般情况，为了保证系统的稳定性，如果系统负载超过阈值，就会阻止用户请求，而如果使用消息队列将请求缓存起来，等待系统处理完毕后通知用户下单完毕，这方法虽然会耗时，但出现系统不能下单的情况
```

场景描述

秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为了解决这个问题，一般需要

在应用前端加入消息队列。这样做的好处有

1.可以控制活动的人数

2.可以缓解短时间内高流量压垮应用

3.用户请求，服务器接收后，首先写入消息队列，假如消息队列长度超过最大数量，则直接

抛弃用户请求或跳转到错误页面

4.秒杀业务根据消息队列中的请求信息，再做后续处理

数据分发

数据分发含义

通过消息队列可以让数据在多个系统之间更加方便流通。只需要将数据发
送到消息队列，数据使用方直接在消息队列中获取数据即可

A系统产生数据，发送到MQ
BCD哪个系统需要，自己去MQ消费即可
如果某个系统不需要数据，取消对MQ消息的消费即可
新系统要数据，直接从MQ消费即可

PS：数据异构也能通过消息队列实现

异步处理

场景描述

用户注册后，需要发注册邮件和注册短信。传统的做法有两种

1.串行方式　2. 并行方式

串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信，以上三个任务

完成后，返回给客户端

并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个

任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间

> 这里可以掺杂着看看优缺点，对比之类的

比如：

MQ对比

常见的MQ产品宏观对比

产品	开发语言	单机吞吐量	时效性	可用性	特性
ActiveMQ	java	万级	ms级	高(主从架构)	ActiveMQ 成熟的产品，在很多公司得到应用;有较多的文档;各种协议支持较好
RabbitMQ	erlang	万级	us级	高(主从架构)	RabbitMQ 基于erlang开发，所以并发能力强，性能极其好，延时很低，管理界面较丰富
RocketMQ	java	10万级	ms级	非常高(分布式架构)	RocketMQ MQ功能比较完备，扩展性佳
Kafka	scala	10万级	ms级以内	非常高(分布式架构)	Kafka 只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广

RocketMQ 优点：

1.单机吞吐量：十万级
2.可用性：非常高，分布式架构
3.消息可靠性：经过参数优化配置，消息可以做到 0 丢失
4.功能支持：MQ 功能较为完善，还是分布式的，扩展性好
5.支持 10 亿级别的消息堆积，不会因为堆积导致性能下降
6.源码是 Java，方便结合公司自己的业务进行二次开发
天生为金融互联网领域而生，对于可靠性要求很高的场景，尤其是电商里面的订单扣款，以及业务削峰，在大量交易涌入时，后端可能无法及时处理的情况
7.RoketMQ 在稳定性上可能更值得信赖，这些业务场景在阿里双11已经经历了多次考验

RocketMQ 缺点：

1.没有在 MQ 核心中去实现 JMS 等接口，有些系统要迁移需要修改大量代码

2支持的客户端语言不多，目前是Java及c++，其中c++不成熟

为什么选择rocketmq

1 消息可靠性，稳定性，抗住过阿狸双十一，适合电商下单，支付（回调时发消息改变订单状态，取消订单时定时任务回滚）这种对消息丢失敏感，对可靠性要求高的场景

2.单机吞吐量高，支持大量消息堆积，适合高并发的电商场景进行业务削峰

3.分布式架构，可用性高

消息队列存在的问题

消息队列起到解耦、削峰、数据分发的作用，同时也存在着**系统可用性降低、系统复杂度提高、一致性问题** 这三个方面缺点。

系统可用性降低 : 系统引入的外部依赖越多，系统稳定性越差。一旦MQ宕机，就会对业务造成影响.

系统复杂度提高 : MQ的加入大大增加了系统的复杂度，以前系统间是同步的远程调用，现在是通过MQ进行异步调用。

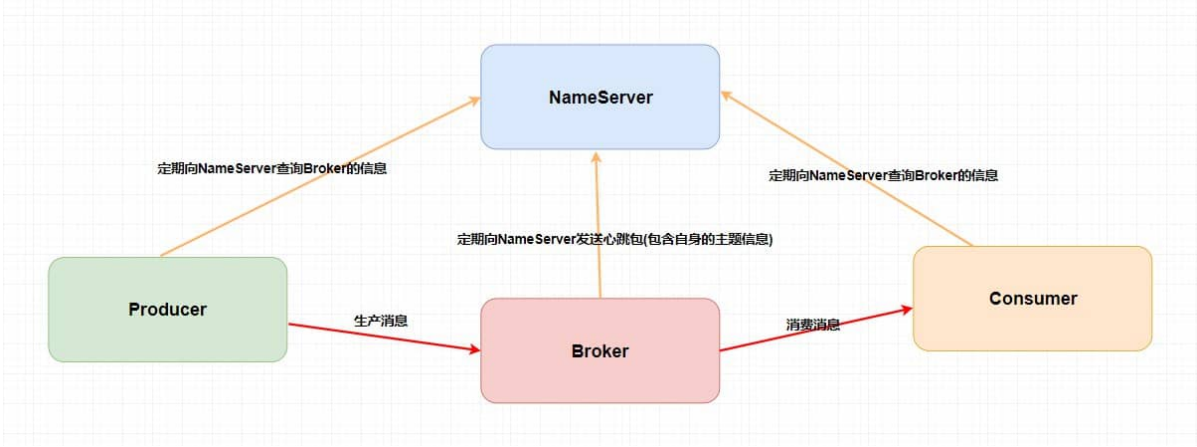
消息没有被重复消费、怎么处理消息丢失情况、如何保证消息传递的顺序性。

一致性问题 : A系统处理完业务，通过MQ给B、C、D三个系统发消息数据，如果B系统、C系统处理完成、D处理失败、如何保证消息数据处理的一致性。

2.入门-了解rcmq

时间有限，我们一般关注核心概念：

> **架构**：



RocketMQ 整体架构设计主要分为四大部分，分别是：Producer、Consumer、Broker、NameServer。

■ Producer：就是消息生产者，可以集群部署。它会先和 NameServer 集群中的**随机一台**建立长连接，得知当前要发送的 Topic 存在哪台 Broker Master上，然后再与其建立长连接，支持多种负载平衡模式发送消息，默认通过**轮询**去每个队列生产数据。

■ Consumer：消息消费者，也可以集群部署。它也会先和 NameServer 集群中的随机一台建立长连接，得知当前要消息的 Topic 存在哪台 Broker Master、Slave上，然后它们建立长连接

■ Broker：主要负责消息的存储、查询消费。**Broker 会向集群中的每一台 NameServer 注册自己的路由信息。**

支持主从部署，一个 Master 可以对应多个 Slave，Master 支持读写，Slave 只支持读，slave定时从master同步数据（同步/异步刷盘）。 即使master宕机了，slave还是只能被读（提供消费服务）

■ NameServer：

一个 **注册中心**，主要提供两个功能：**Broker 管理** 和 **路由信息管理***（*topic与broker之间的关联信息*）。 `Broker` 会将自己的信息注册到 `NameServer` 中，消费者和生产者就从 `NameServer` 中根据要查找的topic查询broker-topic路由表，和查找到的 `Broker` 进行通信(生产者和消费者定期会向 `NameServer` 去查询相关的 `Broker` 的信息)

通常也是集群部署，但是各 NameServer 之间不会互相通信（去中心化，没有主节点），每个broker都会跟所有nameserver保持长连接，并且每隔三十秒心跳发送心跳，心跳包含了自身的topic配置信息。

nameserver的作用：管理broker和路由信息，让broker和生产者消费组解耦，生产者和消费组不需要关注broker的添加删除变动，只需要与nameserver交流就能根据topic找到对应的broker进行交互，这样broker就可以比较灵活的进行变动了。

支持集群消费和广播消费消息。广播模式下，一条消息会被同一个消费组中的所有消费者消费，集群模式下消息只会被一个消费者消费。

> 三个概念：

■ Topic： 话题，主题，是一种消息的逻辑分类，比如说你有订单类的消息，也有库存类的消息，那么就需要进行分类，一个是订单 Topic 存放订单相关的消息，一个是库存 Topic 存储库存相关的消息。

■Message：消息的载体。一个 Message 必须指定 topic，相当于寄信的地址。Message 还有一个可选的 tag 设置，以便消费端可以基于 tag 进行过滤消息

■Tag：可以被认为是对 Topic 进一步细化。一般在相同业务模块中通过引入标签来标记不同用途的消息。

broker与Topic的关系：

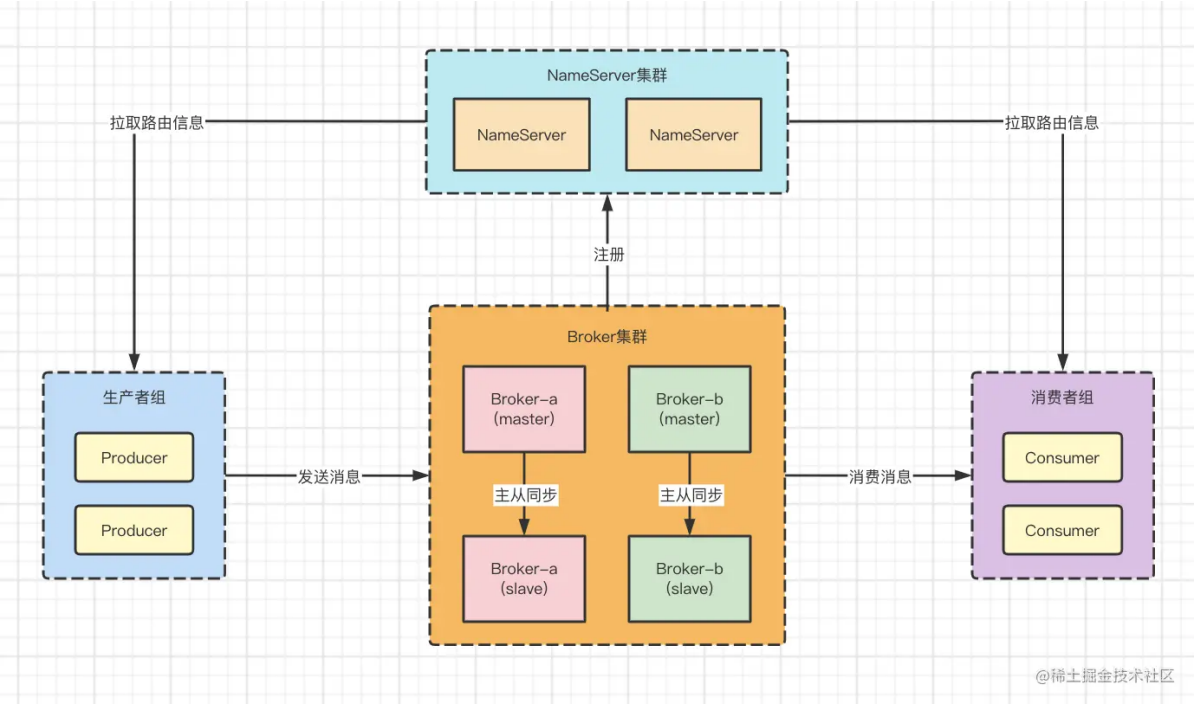
Broker 在实际部署过程中对应一台服务器（RocketMQ服务端），每个 Broker 可以存储多个Topic的消息，每个Topic的消息也可以分片存储于不同的 Broker，Broker和Topic是多对多的关系。

Topic和MessageQueue的关系：

Topic是一个逻辑上的概念。图中的MessageQueue，即消息队列，是用于存储消息的物理地址，一个队列只能归属一个topic，一个topic可以有多个队列

每个Topic中的消息地址可以存储于多个MessageQueue 中，MessageQueue又可以分布在不同的Broker上。

> 工作流程



通过这张图就可以很清楚的知道，RocketMQ大致的工作流程：

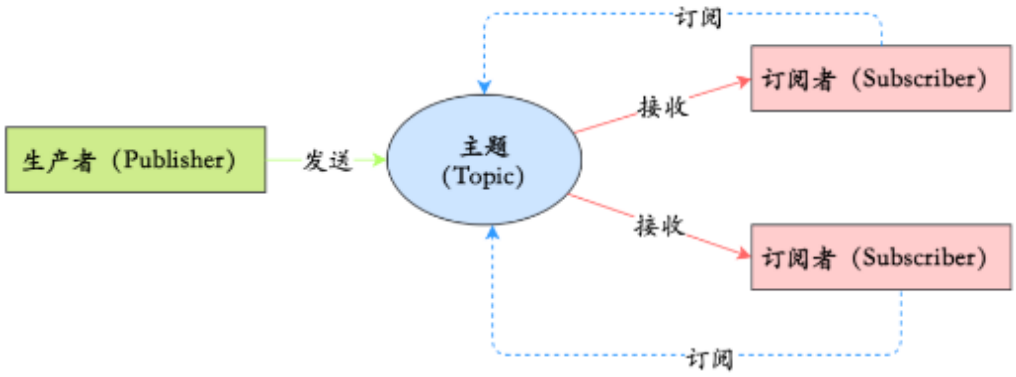
- 先启动 NameServer 集群，各 NameServer 之间无任何数据交互，Broker 启动之后会向所有 NameServer 定期（每 30s）发送心跳包，包括：IP、Port、TopicInfo，NameServer 会定期扫描 Broker 存活列表，如果超过 120s 没有心跳则移除此 Broker 相关信息，代表下线。

  这样每个 NameServer 就知道集群所有 Broker 的相关信息
- Producer在启动之后会跟会NameServer建立长连接，定期从NameServer中获取Broker的信息，当发送消息的时候，会根据消息发送需要的topic去找对应的Broker地址，如果有的话，就向这台Broker发送请求；没有找到的话，就看根据是否允许自动创建topic来决定是否发送消息。

- Broker在接收到Producer的消息之后，会将消息存起来，持久化，如果有从节点的话，也会主动同步给从节点，实现数据的备份
- Consumer启动之后也会跟会NameServer建立长连接，定期从NameServer中获取Broker和对应topic的信息，然后根据自己需要订阅的topic信息找到对应的Broker的地址，然后跟Broker建立连接，获取消息，进行消费

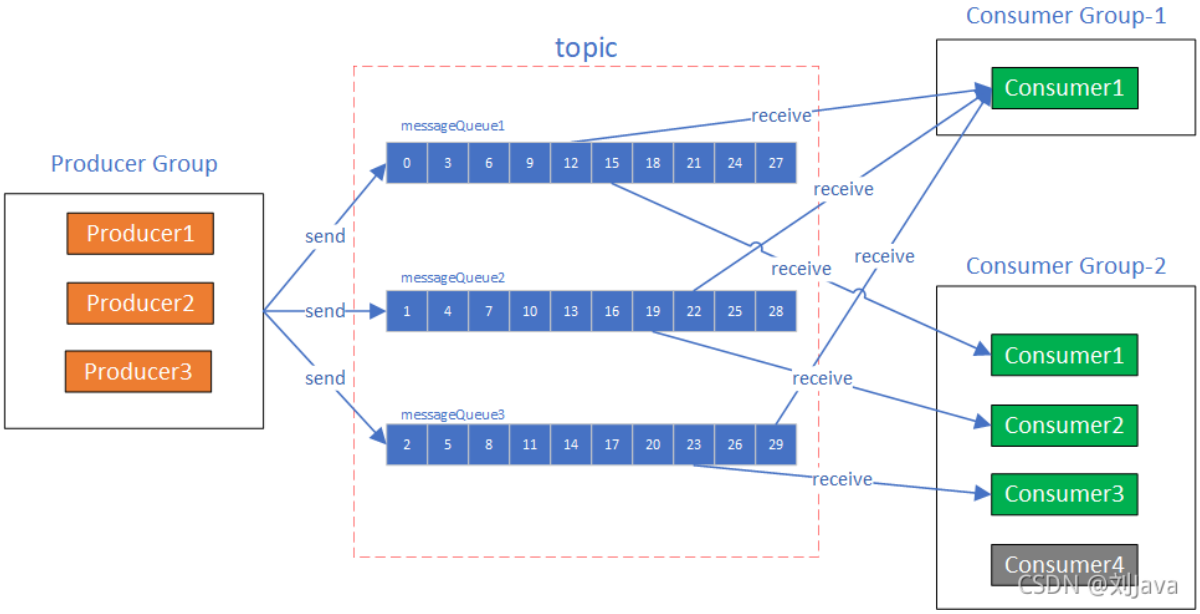
> rocketmq的消息模型

RocketMQ的消息模型（Message Model）主要由 Producer、Broker、Consumer 三部分组成，其中 Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息，**典型的发布-订阅模式。**



发布-订阅模型　　牛客@、烟雨

> 消息生成与消息消费



> 消息类型

1.普通消息

2.定时/延时消息

3.顺序消息

4.事务消息

其他还有概念，消费位移，集群与广播模式，push和pull的消费方式，rcmq如何保证高性能读写以及rcmq的刷盘，集群架构，不过这些都是八股，属于后续深入学习的内容，这里入门我们了解上述概念就足够了

3.安装使用-上手rcmq

应用

安装与启动

RocketMQ的下载与安装（全网最细保姆级别教学）_rocketmq 下载安装舒一笑的博客-CSDN博客

RocketMQ保姆级教程 - 掘金 (juejin.cn)

安装rcmq

环境要求：

- Linux64位系统
- JDK1.8(64位)

安装

```
$ wget https://dist.apache.org/repos/dist/release/rocketmq/5.1.1/rocketmq-all-5.1.1-bin-release.zip

$ unzip rocketmq-all-5.1.1-bin-release.zip
```

目录介绍

- bin：启动脚本，包括shell脚本和CMD脚本
- conf：实例配置文件，包括broker配置文件、logback配置文件等
- lib：依赖jar包，包括Netty、commons-lang、FastJSON等

启动rcmq

0.开放服务器端口

我们在安装rocketmq后，要开放的端口一般有4个：9876，10911，10912，10909

RocketMQ服务中各端口号说明_rocketmq 端口-CSDN博客

1.配置修改

修改jvm参数

在启动NameServer，broker之前，修改一下启动时的jvm参数，因为默认的参数都比较大，为了避免内存不够，建议修改小，否则无法启动

修改runbroker.sh runserver.sh

```
JAVA_OPT="${JAVA_OPT} -server -Xms256m -Xmx256m"
```

将启动jvm内存参数调小

修改conf/broker.conf

这里需要改一下Broker配置文件，需要指定NameServer的地址，因为需要Broker需要往NameServer注册

在文件末尾追加namesrv地址

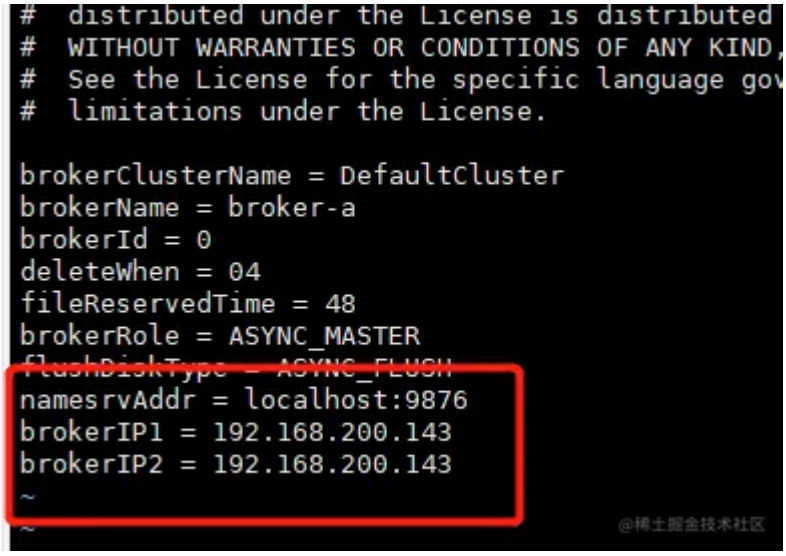
```
namesrvAddr = localhost:9876
```

因为NameServer跟Broker在同一台机器，所以是localhost，NameServer端口默认的是9876。

文件末尾继续追加brokerIp，IP值是当前部署broker的服务器外网IP

```
brokerIP1 = 192.168.200.143
brokerIP2 = 192.168.200.143
```

因为Broker向NameServer进行注册的时候，带过去的ip如果不指定就会自动获取，但是自动获取的有个坑，就是有可能客户端无法访问到这个自动获取的ip，所以我建议手动指定客户端可以访问到的服务器ip。



2.启动NameServer

sh ./mqnamesrv

```
cd /root/rocketmq/rocketmq-all/bin

nohup sh `./mqnamesrv` &
```

在bin目录下执行

jps查看当前已启动的java进程

```
[root@VM-8-12-centos bin]# jps
21233 Jps
20051 NamesrvStartup
9206 BrokerStartup
```

出现了namesrvstartup即为成功

3.启动Broker

进入bin目录执行

```
nohup sh ./mqbroker -c ../conf/broker.conf -n localhost:9876
autoCreateTopicEnable=true &
```

jps查看当前已启动的java进程

```
[root@VM-8-12-centos bin]# jps
21233 Jps
20051 NamesrvStartup
9206 BrokerStartup
```

出现brokerstartup即为成功

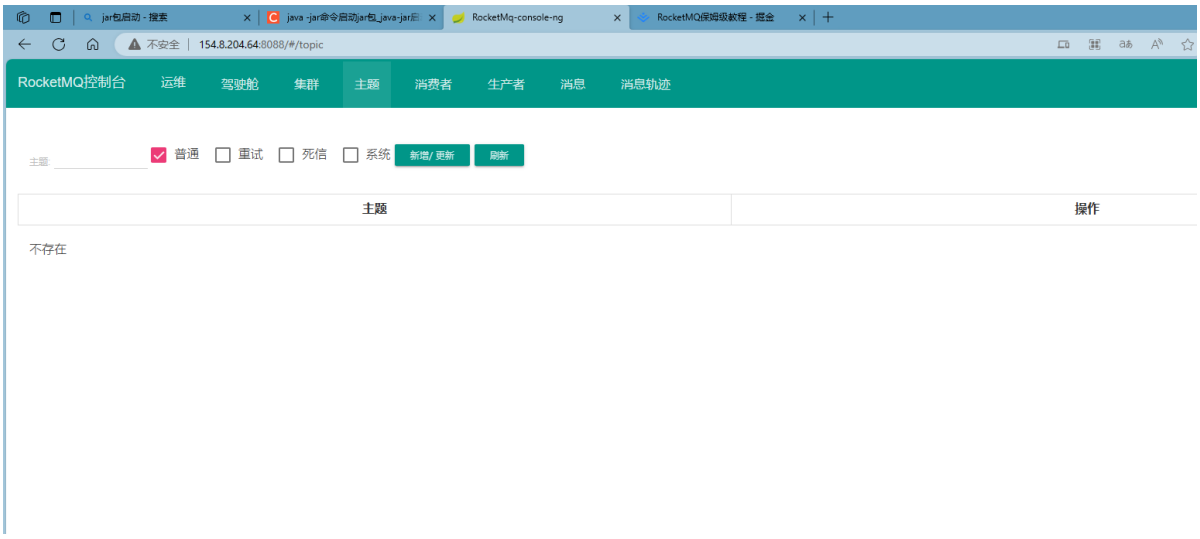
nohup sh ./mqbroker -c ../conf/broker.conf -n 154.8.204.64:9876 autoCreateTopicEnable=true &

4.搭建可视化控制台

[linux - nohup 命令 &后一按回车就exit_nohup回车后就退出-CSDN博客]

[RocketMQ保姆级教程 - 掘金 (juejin.cn)]

```
cd /root/rocketmq/rcmq-console


nohup `/root/java/jdk1.8.0_151/bin/java -jar -jar -server -Xms256m -Xmx256m -
Drocketmq.config.namesrvAddr=localhost:9876 -Dserver.port=8088 ./rcmq-
console.jar` &
```



5.**创建主题**

Apache RocketMQ 5.0版本下创建主题操作，推荐使用mqadmin工具，需要注意的是，对于消息类型需要通过属性参数添加。示例如下：

```
/bin/mqadmin updateTopic -c DefaultCluster -t DelayTopic -n 127.0.0.1:9876 -a
+message.type=DELAY
```

虽然我们可以在发消息时创建主题，但是最好手动来创建

springboot整合发送普通消息

原生使用

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>5.1.1</version>
</dependency>
```

生产者

```
public class Producer {
    public static void main(String[] args) throws Exception {
        //创建一个生产者，指定生产者组为StarGeo
        DefaultMQProducer producer = new DefaultMQProducer("StarGeo");

        // 指定NameServer的地址
        producer.setNamesrvAddr("154.8.204.64:9876");
        // 第一次发送可能会超时，我设置的比较大
        producer.setSendMsgTimeout(1000000);

        // 启动生产者
        producer.start();

        // 创建一条消息
        // topic为HomuraAkime
        // 消息内容为homura daisuki
        // tags 为 homura
        Message msg = new Message("HomuraAkime", "homura", "homura daisuki
".getBytes(RemotingHelper.DEFAULT_CHARSET));

        // 发送消息并得到消息的发送结果，然后打印
        SendResult sendResult = producer.send(msg);
        System.out.printf("%s%n", sendResult);

        // 关闭生产者
        producer.shutdown();
    }

}
```

构建一个消息生产者实例，然后指定生产者组

指定NameServer的地址：服务器的ip:9876，因为需要从NameServer拉取Broker的信息

producer.start() 启动生产者

构建一个消息，指定这个消息往目标topic发送

producer.send(msg)：发送消息，打印结果

消费者

```
public class Consumer {
    public static void main(String[] args) throws InterruptedException,
MQClientException {

        // 通过push模式消费消息，指定消费者组
        DefaultMQPushConsumer consumer = new
DefaultMQPushConsumer("StarGeoConsumer");

        // 指定NameServer的地址
        consumer.setNamesrvAddr("154.8.204.64:9876");

        // 订阅这个topic下的所有的消息
        consumer.subscribe("HomuraAkime", "*");

        // 注册一个消费的监听器，当有消息的时候，会回调这个监听器来消费消息
        consumer.registerMessageListener(new MessageListenerConcurrently() {

            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt>
msgs,

 ConsumeConcurrentlyContext context) {
                for (MessageExt msg : msgs) {
                    System.out.printf("消费消息:%s", new String(msg.getBody()) +
"\n");
                }

                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });

        // 启动消费者
        consumer.start();

        System.out.printf("Consumer Started.%n");
    }

}
```

- 创建一个消费者实例对象，指定消费者组为
- 指定NameServer的地址：服务器的ip:9876
- 订阅xxx 这个topic的所有信息

- consumer.registerMessageListener ，这个很重要，是注册一个监听器，这个监听器是当有消息的时候就会**回调**这个监听器，处理消息，所以需要用户实现这个接口，然后处理消息。（异步的还是同步阻塞的？）
- 启动消费者

启动之后，消费者就会消费刚才生产者发送的消息

集成SpringBoot

[常见问题 · apache/rocketmq-spring Wiki (github.com)](https://github.com/apache/rocketmq-spring/wiki/%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98)

官方文档

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-spring-boot-starter</artifactId>
    <version>2.1.1</version>
</dependency>
```

缺点：相比于官方最新版本相对滞后，有些特性不支持

教程较少，用起来不是特别舒服

yml配置

```
rocketmq:
  producer:
    group: homura
  name-server: 154.8.204.64:9876
```

创建消费者

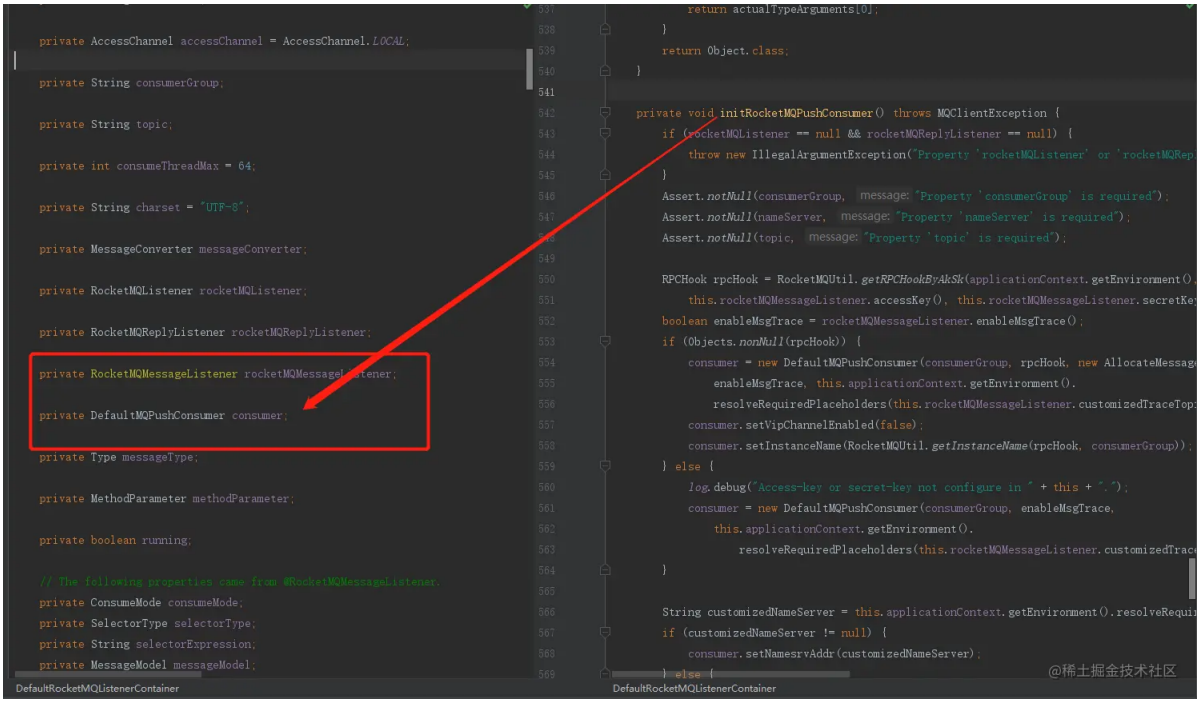
SpringBoot底下只需要实现RocketMQListener接口，然后加上@RocketMQMessageListener注解即可

```
@Component
@RocketMQMessageListener(consumerGroup = "madoka", topic = "love")
public class MadokaConsumer implements RocketMQListener<String> {
    @Override
    public void onMessage(String msg) {
        System.out.println( msg);
    }
}
```

@RocketMQMessageListener需要指定消费者属于哪个消费者组，消费哪个topic，NameServer的地址已经通过yml配置文件配置类获取

消费者消费消息时会执行onMessage里的自定义逻辑

@RocketMQMessageListener 注解处理



从这可以看出，会为每一个加了@RocketMQMessageListener注解的对象创建一个DefaultMQPushConsumer，所以最终也是通过DefaultMQPushConsumer消费消息的。

至于监听器，是在这

```
public class DefaultMessageListenerConcurrently implements MessageListenerConcurrently {

    /unchecked/
    @Override
    public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {
        for (MessageExt messageExt : msgs) {
            log.debug("received msg: {}", messageExt);
            try {
                long now = System.currentTimeMillis();
                handleMessage(messageExt);
                long costTime = System.currentTimeMillis() - now;
                log.debug("consume {} cost: {} ms", messageExt.getMsgId(), costTime);
            } catch (Exception e) {
                log.warn("consume message failed. messageExt:{}, error:{}", messageExt, e);
                context.setDelayLevelWhenNextConsume(delayLevelWhenNextConsume);
                return ConsumeConcurrentlyStatus.RECONSUME_LATER;
            }
        }

        return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
    }
}
```


遍历每条消息，然后调用handleMessage，最终会调用实现了RocketMQListener的对象处理消息。

生产者

```java
@Tag(name = "单品api")
@RestController
@RequestMapping("/sku")
@RequiredArgsConstructor(onConstructor_ = { @Autowired})
public class SkuController {
    //所有按销量排序的接口保留但弃用
    private final SkuService skuService;
    private final RocketMQTemplate template;
    /*根据商品id新增单品*/
```

先注入RocketMQTemplate

```java
@PostMapping("/testMQ")
public BaseResponse<String> testMQ() throws InterruptedException {
//      Order order= Order.builder().orderNum("hhhahaha").createTime(new Timestamp(System.currentTimeMillis())).build();
//
//      rocketMQDelayProducer.orderDelaySend("order-delay",order,3000,3);

    rocketMQTemplate.convertAndSend( destination: "love", payload: "2333");
    return null;
}
```

直接发送消息，绑定对应的topic和消息内容

NameServer的地址和生产者组名已经通过yml配置文件配置类获取

4.进阶应用

事务消息实战

基于RocketMQ分布式事务 - 完整示例 - 知乎 (zhihu.com)

SpringCloud 集成RocketMQ实现分布式事务 - 掘金 (juejin.cn)

常见问题 · apache/rocketmq-spring Wiki (github.com)

针对不同的分布式场景业界常见的解决方案有2PC、TCC、可靠消息最终一致性、最大努力通知这几种。

rocketmq属于可靠消息最终一致性，但是借鉴了2PC的思想并且实现了自己的补偿机制（回查）

rocketmq把消息的发送分为准备和提交两个阶段，本地事务执行成功后才提交消息（2PC）

普通的分布式事务实现

1.定义半消息的发送者

```java
import javax.annotation.PostConstruct;
import java.nio.charset.StandardCharsets;
@Component
@RequiredArgsConstructor(onConstructor_ = {@Autowired})
public class TransactionProducer {

   private final RocketMQTemplate rocketMQTemplate;
    public void sendTransactionMessage(String topic,String business,String businessId,String transId,Object o)
    {


        //发送半消息
        //第一个参数是topic，第二个参数是要发送给消费者消费的信息，第三个是只有回调执行本地事务时会传入的额外参数，不会传给消费者的,没有就null

        TransactionSendResult transactionSendResult = rocketMQTemplate.sendMessageInTransaction(topic,
                MessageBuilder.withPayload(o)
                        .setHeader(RocketMQHeaders.TRANSACTION_ID, transId)
                        .setHeader( headerName: "business_id",businessId)
                        .setHeader( headerName: "business",business)
                        .build(),  arg: null);

        //发送事务消息采用的是sendMessageInTransaction方法，返回结果为TransactionSendResult对象，该对象中包含了事务发送的状态、本地事务执行的状态等
        //发送状态
        String sendStatus = transactionSendResult.getSendStatus().name();
        //本地事务执行状态
        String localState = transactionSendResult.getLocalTransactionState().name();
        System.out.println("发送状态:"+sendStatus+";本地事务执行状态"+localState);

    }

}
```

如图，通过rocketMQTemplate.sendMessageInTransaction发送半消息，用MessageBuider.withPayload构建messaging消息

在方法内，org.springframework.messaging消息（spring封装的）会被转成rocketmq的消息

```java
org.apache.rocketmq.common.message.Message rocketMsg = this.createRocketMqMessage(destination, message);
```

通过TransactionSendResult获取发送事务消息的结果，在本地事务执行完返回给broker状态信息（UNKNOWN/COMMIT/ROLLBACK）时，返回给消息发送者，消息发送者处理返回消息，可以返回（业务中不需要的情况下也可以不等待返回值直接返回）

```
发送状态:SEND_OK;本地事务执行状态ROLLBACK_MESSAGE
```

```
发送状态:SEND_OK;本地事务执行状态UNKNOW
```

```
发送状态:SEND_OK;本地事务执行状态COMMIT_MESSAGE
```

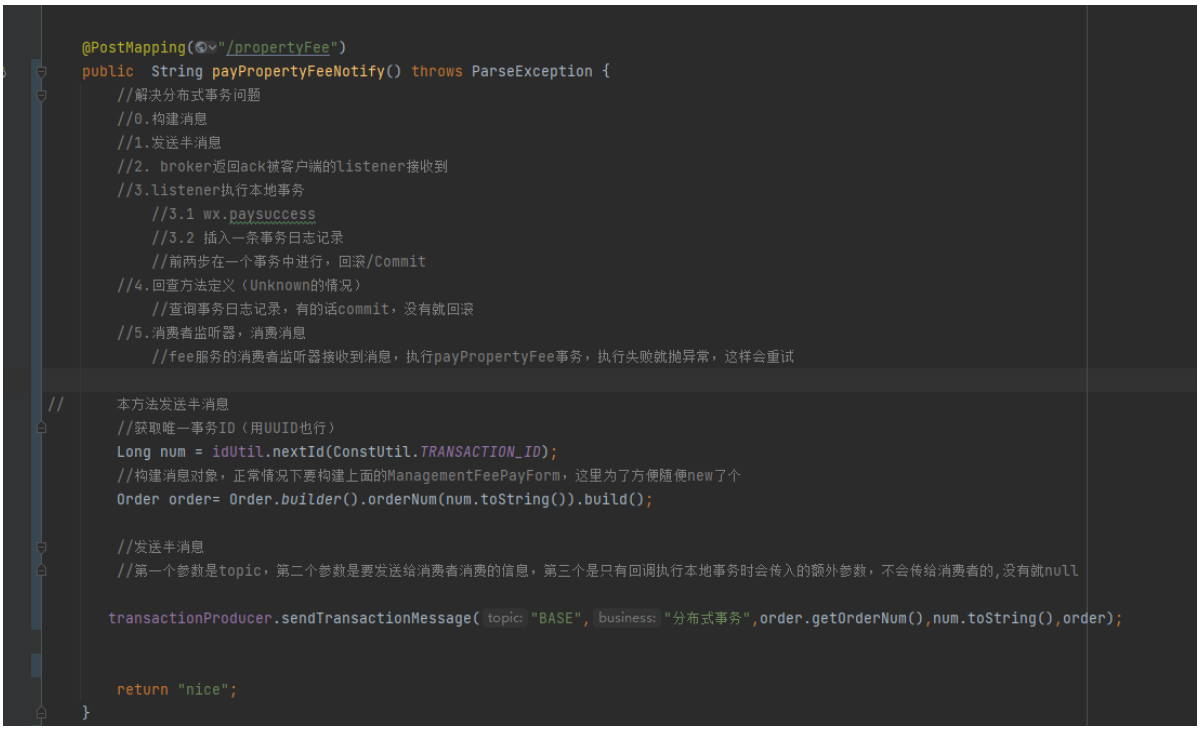
2.定义生产者本地事务监听器

```java
@RequiredArgsConstructor(onConstructor_ = {@Autowired})
@RocketMQTransactionListener(rocketMQTemplateBeanName = "rocketMQTemplate")
public class DefaultTransactionListener implements RocketMQLocalTransactionListener {
    private final RocketMqTransactionLogMapper rocketMqTransactionLogMapper;
    private final TransactionService transactionService;
    @Override
    public RocketMQLocalTransactionState executeLocalTransaction(Message message, Object o) {
        MessageHeaders headers=message.getHeaders();
        String id= (String)headers.get(RocketMQHeaders.TRANSACTION_ID);
        System.out.println(id+"?????");
        String businessId=(String) headers.get("business_id");
        try {
            //信息放headers的问题就是，不看消息发送者，消费者就不知道接受的消息的含义
            //所以一般都会直接包装成实体类发送消息，这样可以直接点进实体类看
            //执行本地事务，如果执行失败会回滚异常会抛给catch，然后返回rollback，brokerOP标记删除已发送的消息
            transactionService.changeStatuswithRocketMqLog(businessId, business: "fee分布式事务",Long.parseLong(id));
            return RocketMQLocalTransactionState.ROLLBACK;
        }catch (Exception e){
            e.printStackTrace();
            return RocketMQLocalTransactionState.ROLLBACK;
        }
    }

    @Override
    public RocketMQLocalTransactionState checkLocalTransaction(Message message) {
        MessageHeaders headers = message.getHeaders();
        //获取事务ID
        String transactionId = (String) headers.get(RocketMQHeaders.TRANSACTION_ID);
        //根据事务id从日志表检索
        QueryWrapper<TransactionLog> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq( column: "id",transactionId);
        TransactionLog transactionLog = rocketMqTransactionLogMapper.selectOne(queryWrapper);
        if(null != transactionLog){
            return RocketMQLocalTransactionState.COMMIT;
        }
        return RocketMQLocalTransactionState.ROLLBACK;
    }

}
```

如图，要带上@RocketMQTransactionListener，实现RocketMQLocalTransactionListener，复写本地事务执行方法和回查方法

根据事务执行情况返回COMMIT/ROLLBACK/UNKNOWN

3.在接口内调用生产者发送半消息



```
@PostMapping("/propertyFee")
public  String payPropertyFeeNotify() throws ParseException {
    //解决分布式事务问题
    //0.构建消息
    //1.发送半消息
    //2. broker返回ack被客户端的listener接收到
    //3.listener执行本地事务
        //3.1 wx.paysuccess
        //3.2 插入一条事务日志记录
        //前两步在一个事务中进行，回滚/Commit
    //4.回查方法定义（Unknown的情况）
        //查询事务日志记录，有的话commit，没有就回滚
    //5.消费者监听器，消费消息
        //fee服务的消费者监听器接收到消息，执行payPropertyFee事务，执行失败就抛异常，这样会重试

//      本方法发送半消息
    //获取唯一事务ID（用UUID也行）
    Long num = idUtil.nextId(ConstUtil.TRANSACTION_ID);
    //构建消息对象，正常情况下要构建上面的ManagementFeePayForm，这里为了方便随便new了个
    Order order= Order.builder().orderNum(num.toString()).build();

    //发送半消息
    //第一个参数是topic，第二个参数是要发送给消费者消费的信息，第三个是只有回调执行本地事务时会传入的额外参数，不会传给消费者的,没有就null

   transactionProducer.sendTransactionMessage( topic: "BASE", business: "分布式事务",order.getOrderNum(),num.toString(),order);


    return "nice";
}
```

如图，这里调用了注入的生产者中的方法，传递topic，business,businessid，事务唯一ID，要发送的对象

PS:我们这里的业务规范是，分布式事务的实现采取统一实现，部分特殊的另外再写

统一分布式事务实现规范都调用统一的defaultlistener和produce中的不同方法来处理，在生产者事务监听器中，执行本地事务时根据不同的business来选择执行不同的本地事务方法实现

我们有统一的三个业务字段，用来写日志表，business,businessid，事务唯一ID，事务唯一ID通过UUID或redis实现

4.消费者接受消息开始消费

这步和普通消息的实现一样，没啥特殊的

延时消息实战

使用RocketMQTemplate发送各种消息 - 掘金 (juejin.cn)

springboot+rocketmq（4）:实现延时消息*rocketmqtemplate发送延迟消息*12程序猿的博客-CSDN博客

1.构建生产者工具类

```java
@Component
public class RocketMQDelayProducer {
    @Autowired
    private RocketMQTemplate rocketMQTemplate;
    /**
     * 同步发送延时消息
     *
     * @param topic      topic
     * @param message    消息体
     * @param timeout    超时
     * @param delayLevel 延时等级：现在RocketMq并不支持任意时间的延时，需要设置几个固定的延时等级，
     *                   从1s到2h分别对应着等级 1 到 18，消息消费失败会进入延时消息队列
     *                   "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h";
     */
    public void orderDelaySend(String topic, Object message, long timeout,int delayLevel){
        rocketMQTemplate.syncSend(topic, MessageBuilder.withPayload(message).build(),timeout,delayLevel);
    }
```

timeout指定的是消息发送超时间

delaylevel来指定消息延时时长

rcmq 5.x才开始支持精确延时，目前还是阶梯指定的

2.在业务代码使用生产者工具类发消息

```java
    @PostMapping("/testMQ")
    @PostConstruct
    public BaseResponse<String> testMQ() throws InterruptedException {
        Order order= Order.builder().orderNum("hhhahaha").createTime(new Timestamp(System.currentTimeMillis())).build();

        rocketMQDelayProducer.orderDelaySend( topic: "order-delay",order, timeout: 3000, delayLevel: 3);

        return null;
    }
}
```

比如刚创建完订单时

3.构建消费者，自定义消费逻辑

```java
@Component
@RocketMQMessageListener(consumerGroup = "order-delay",topic = "order-delay")
public class DelayConsumer implements RocketMQListener<String> {

    @Override
    public void onMessage(String s) {
        Order order= JSONObject.parseObject(s,Order.class);
        System.out.println(order.getOrderNum()+"/"+order.getCreateTime());
        System.out.println("-----"+new Timestamp(System.currentTimeMillis()));
    }
}
```

这里是简单的逻辑，实战时还需要考虑redis保证幂等性，根据订单状态判断是否需要回滚库存，根据本地事务方法执行结构，返回成功或失败信息

PS：如果是原生的话，需要在消费者监听器的consumeMessage方法中手动返回消费是否成功的消息，broker据此判断是否重试。

但是如果使用spring版本的，我们只能实现RocketMQListener接口的onMessage方法，而此方法是void型的，那么消费失败时如何重试呢?

在rocketmq的github上的issue中找到了答案，原来这里默认就是消费者处理时抛出异常时就会自动重试

```java
    @Override
    public void onMessage(String s) {
        Order order= JSONObject.parseObject(s,Order.class);
        System.out.println(order.getOrderNum()+"/"+order.getCreateTime());
        System.out.println("-----"+new Timestamp(System.currentTimeMillis()));
        throw new RuntimeException("相似了");

    }
}
```

```
hhhahaha/2023-10-23 21:03:49.0
-----2023-10-23 21:03:59.58
hhhahaha/2023-10-23 21:03:49.0
-----2023-10-23 21:04:09.701
hhhahaha/2023-10-23 21:03:49.0
-----2023-10-23 21:04:39.733
hhhahaha/2023-10-23 21:03:49.0
-----2023-10-23 21:05:39.83
```

如图，我们抛出异常，消费重试了3次