**消息中间件解决方案RocketMQ**

传智播客.黑马程序员

# 课程目标

目标1：理解消息中间件概念

目标2：理解RocketMQ组件

目标3：掌握RocketMQ安装

目标4：掌握RocketMQ消息发送与接收

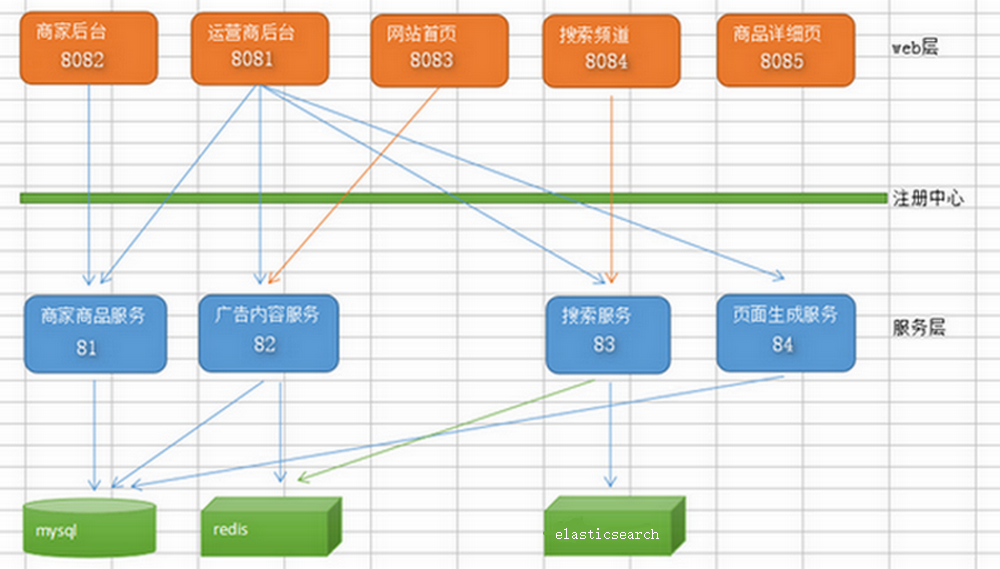
目标5：掌握Spring集成RocketMQ

# 1. RocketMQ介绍

## 1.1消息中间件

### 1.1.1品优购系统模块调用关系分析

我们现在讲品优购已经完成的功能模块梳理如下：



我们已经完成了5个web模块和4个服务模块。其中运营商后台的调用关系最多，用到了商家商品服务、广告内容服务、搜索服务和页面生成服务。这种模块之间的依赖也称之为耦合。而耦合越多，之后的维护工作就越困难。那么如果改善系统模块调用关系、减少模块之间的耦合呢？我们接下来就介绍一种解决方案----消息中间件。

### 1.1.2什么是消息中间件

消息中间件利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行[分布式系统](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F%E7%B3%BB%E7%BB%9F)的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。对于消息中间件，常见的角色大致也就有Producer（生产者）、Consumer（消费者）。

消息中间件常见的作用有：应用解耦、异步处理(提高系统响应速度)、流量削峰(高峰堆积消息，峰后继续处理消息)、日志处理(分布式日志，一般使用kafka)、纯粹通讯。

**常见的消息中间件产品**:

（1）RocketMQ

Rocketmq是一款分布式，队列模型的消息中间件，具有低延迟，高性能和可靠性，万亿级容量和灵活的可扩展性。由阿里巴巴研发与开源，原名Metaq，3.0版本改名为RocketMQ，借鉴参考了JMS规范的MQ实现，但有些概念与标准JMS规范不同。

设计时参考了优秀的开源消息中间件kafka，并且结合阿里实际业务需求在天猫双十一的场景，实现业务削峰，分布式事务的优秀框架。之前在github上是alibaba，现在已经转移到apache旗下。

（2）ActiveMQ

ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的 JMS Provider实现。我们在本次课程中介绍 ActiveMQ的使用。

（3）RabbitMQ

AMQP协议的领导实现，支持多种场景。淘宝的MySQL集群内部有使用它进行通讯，OpenStack开源云平台的通信组件，最先在金融行业得到运用。

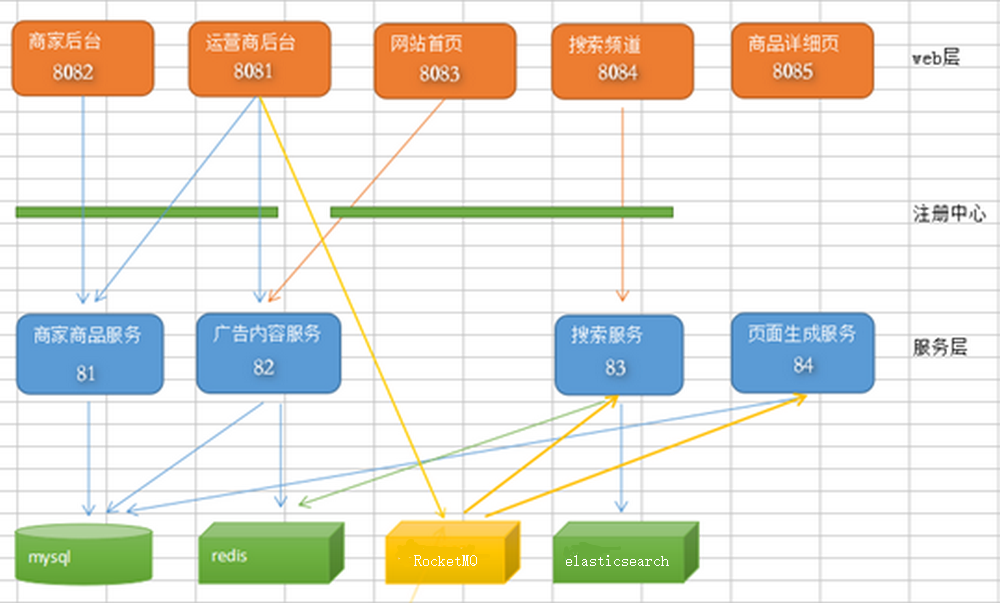
（4）ZeroMQ

号称史上最快的消息队列系统，ZeroMQ是一个非常轻量级的消息系统，专门为高吞吐量/低延迟的场景开发，在金融界的应用中经常可以发现它。与RabbitMQ相比，ZeroMQ支持许多高级消息场景，但是你必须实现ZeroMQ框架中的各个块（比如Socket或Device等）。ZeroMQ非常灵活，但是你必须学习它的80页的手册。

（5）Kafka

Apache下的一个子项目，特点：高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；完全的分布式系统。适合处理海量数据，在分布式日志领域相当成熟。

### 1.1.3改造系统模块调用关系



我们通过引入消息中间件RocketMQ,使得运营商系统与搜索服务、页面生成服务解除了耦合。

## 1.2RocketMQ简介

### 1.2.1什么是RocketMQ

RocketMQ作为一款纯Java开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。

支持事务消息、顺序消息、批量消息、定时消息、消息回溯等能力。

同时，广泛应用于多个领域，包括异步通信解耦、企业解决方案、金融支付、电信、电子商务、快递物流、广告营销、社交、即时通信、移动应用、手游、视频、物联网、车联网等。

RocketMQ特性：

1. 参考了JMS规范，但有些概念与标准JMS规范不同

2. 底层通讯代码采用Netty NIO框架进行数据通信

3. 3.X版本开始摈弃了Zookeeper，内部采用更轻量级的NameServer进行网络路由，提高服务性能，**天然支持消息重试机制**

4. **天然支持集群模型，消费者负载均衡**

5. 采用零拷贝原理，顺序写盘，支持**亿级消息堆积能力**

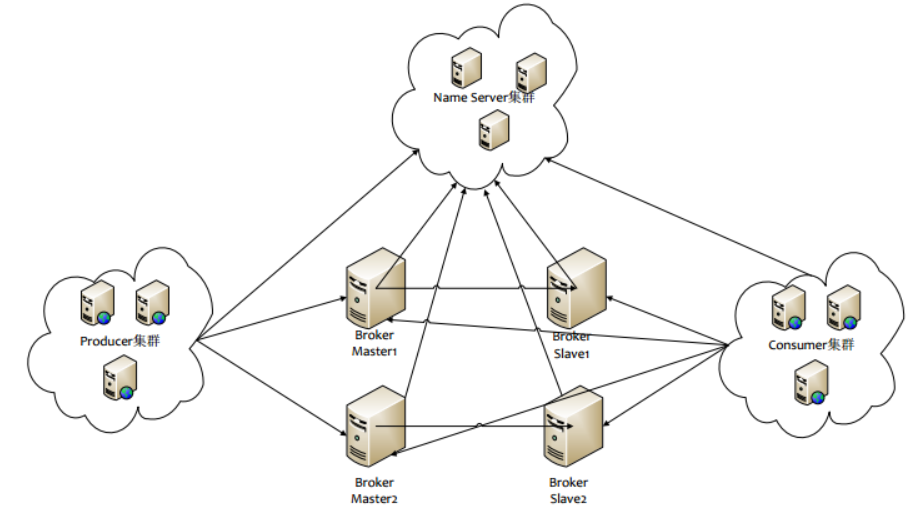
6. 提供**丰富的消息机制**，如顺序消息，**事务消息**等等

7. 提供**丰富的消息拉取模式**

8. 能够**保证严格的消息顺序**

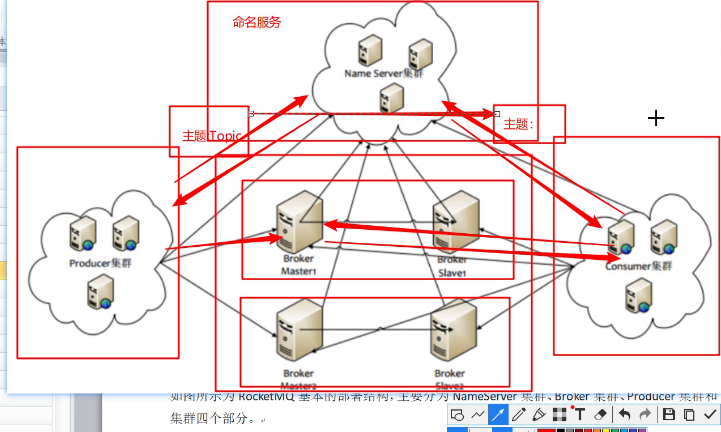
9. **高效的订阅者水平扩展能力**

10. 实时的消息订阅机制，支持广播模式



如图所示为RocketMQ基本的部署结构，主要分为NameServer集群、Broker集群、Producer集群和Consumer集群四个部分。

解释 ：



运行流程： producer 发送主题 topic 到 name server 集群， consumer 通过同一个主题topic连接 nameserver 集群。

Name Sever 主题服务，主题 topic 需要同一个才能通信。

Broker 是消息传递者，消息传递的中间人。

### 1.2.2 NameServer集群

NameServer的作用是注册中心，类似于Zookeeper，但又有区别于它的地方。每个NameServer节点互相之间是独立的，没有任何信息交互，也就不存在任何的选主或者主从切换之类的问题，因此NameServer与Zookeeper相比更轻量级。单个NameServer节点中存储了活跃的Broker列表（包括master和slave），这里活跃的定义是与NameServer保持有心跳。

### 1.2.3 Broker集群

Broker是具体提供业务的服务器，单个Broker节点与所有的NameServer节点保持长连接及心跳，并会定时将Topic信息注册到NameServer，顺带一提底层的通信和连接都是基于Netty实现的。

Broker中分master和slave两种角色，每个master可以对应多个slave，但一个slave只能对应一个master，master和slave通过指定相同的Brokername，不同的BrokerId （master为0）成为一个组。master和slave之间的同步方式分为同步双写和异步复制，异步复制方式master和slave之间虽然会存在少量的延迟，但性能较同步双写方式要高出10%左右。

### 1.2.4 Producer集群

#### 1.2.4.1 与nameserver的关系

单个Producer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。与nameserver之间没有心跳。

#### 1.2.4.2 与broker的关系

单个Producer和与其关联的所有broker保持长连接，并维持心跳。默认情况下消息发送采用轮询方式，会均匀发到对应Topic的所有queue中。

### 1.2.5 Consumer集群

#### 1.2.5.1 与nameserver的关系

单个Consumer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。与nameserver之间没有心跳。

#### 1.2.5.2 与broker的关系

单个Consumer和与其关联的所有broker保持长连接，并维持心跳，失去心跳后，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。

### 1.2.6 消费模式

#### 1.2.6.1 集群模式

在默认情况下，就是集群消费，此时消息发出去后将只有一个消费者能获取消息。

#### 1.2.6.2 广播模式

另一种消费模式，是广播消费。广播消费，类似于ActiveMQ中的发布订阅模式，消息会发给Consume Group中的每一个消费者进行消费。

## 1.3RocketMQ下载与安装

### 1.3.1 下载

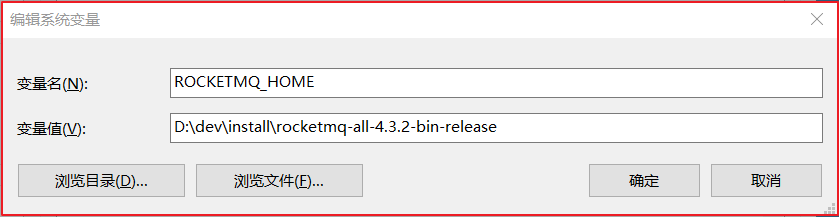
官方网站下载：<http://rocketmq.apache.org/>

我们这里使用的是rocketmq-all-4.3.2-bin-release.zip

### 1.3.2 安装

将下载好的文件解压到D:\dev\install目录下，然后配置环境变量

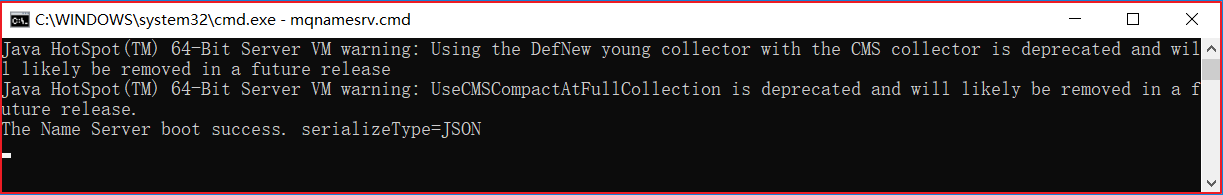
ROCKETMQ\_HOME=D:\dev\install\rocketmq-all-4.3.2-bin-release



### 1.3.3 启动操作

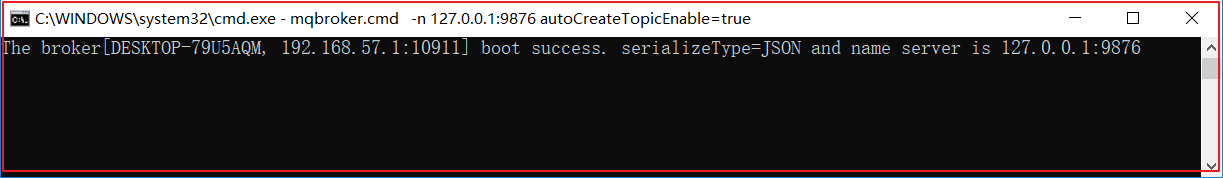
#### 1.3.3.1 启动NAMESERVER

cmd命令框执行进入至‘MQ文件夹\bin’下，然后执行‘start mqnamesrv.cmd’，启动NAMESERVER。成功后会弹出提示框，此框勿关闭。



#### 1.3.3.2 启动BROKER

cmd命令框执行进入至‘MQ文件夹\bin’下，然后执行start mqbroker.cmd -n 127.0.0.1:9876 autoCreateTopicEnable=true ，启动BROKER。成功后会弹出提示框，此框勿关闭。



注意：假如弹出提示框提示‘错误: 找不到或无法加载主类 xxxxxx’。打开runbroker.cmd，然后将%CLASSPATH%加上英文双引号。保存并重新执行start语句。

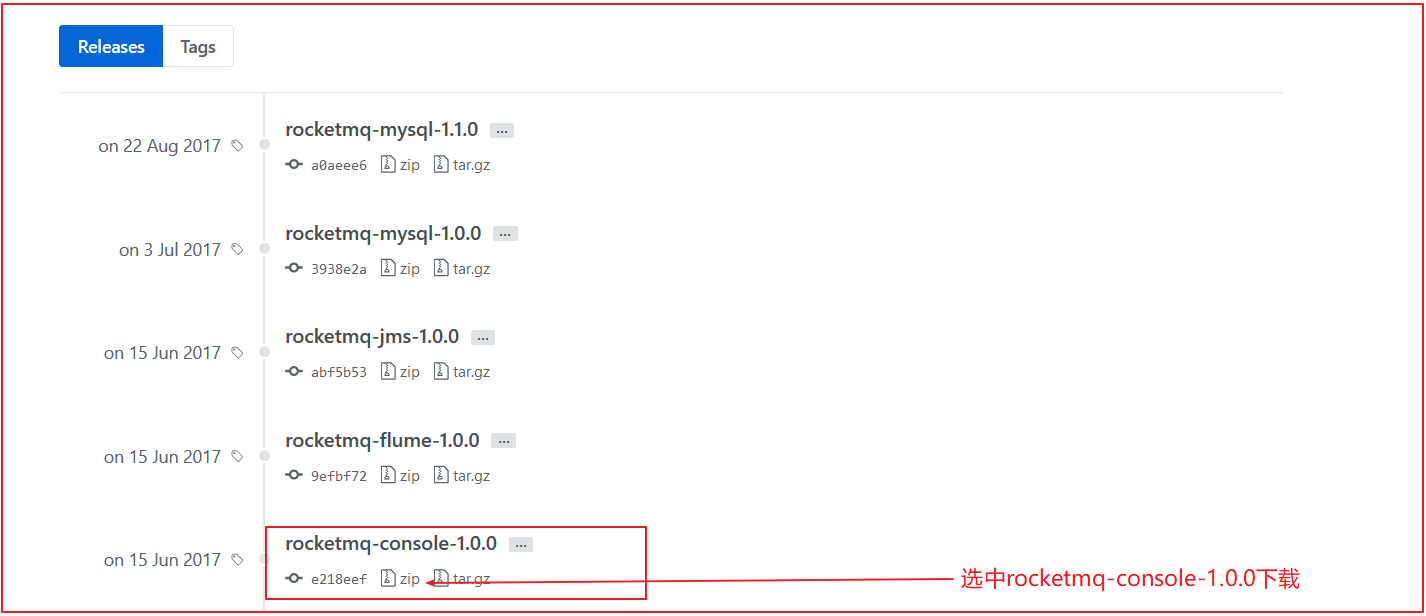


### 1.3.4 RocketMQ插件部署

RocketMQ插件是一个基于SpringBoot编写的可视化插件，主要用于对RocketMQ提供了可视化的管理界面。

#### 1.3.4.1 下载

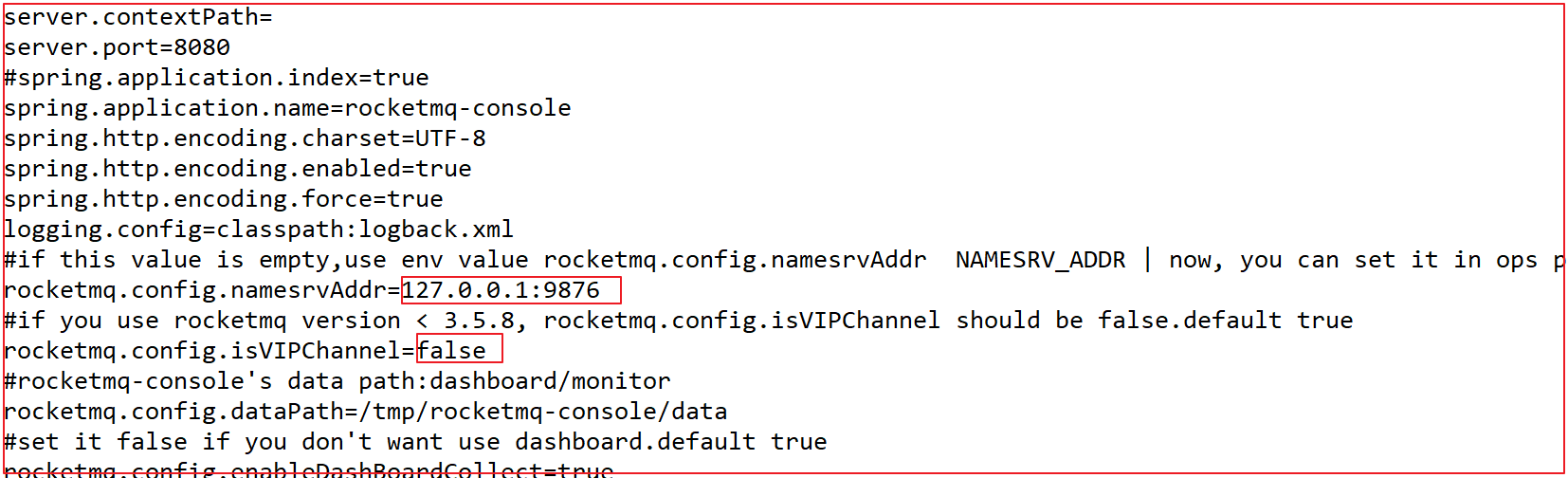
下载地址：https://github.com/apache/rocketmq-externals/releases



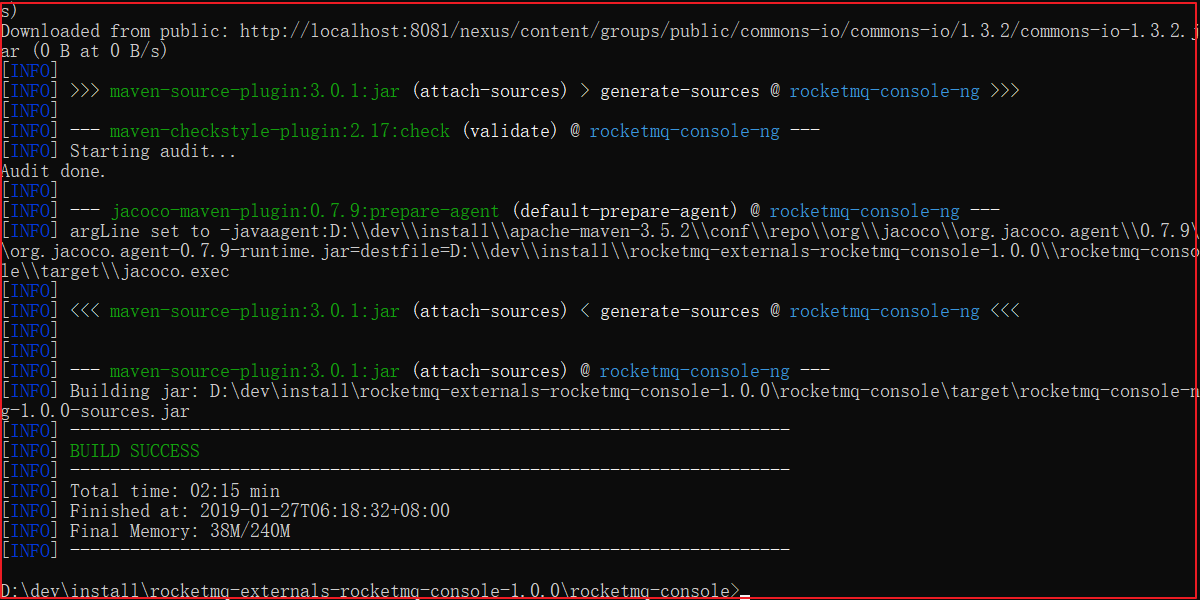
#### 1.3.4.2 编译启动

注意：在执行maven命令前，先修改下配置文件的nameserver地址

修改rocketmq-console\src\main\resources\application.properties，修改如下：



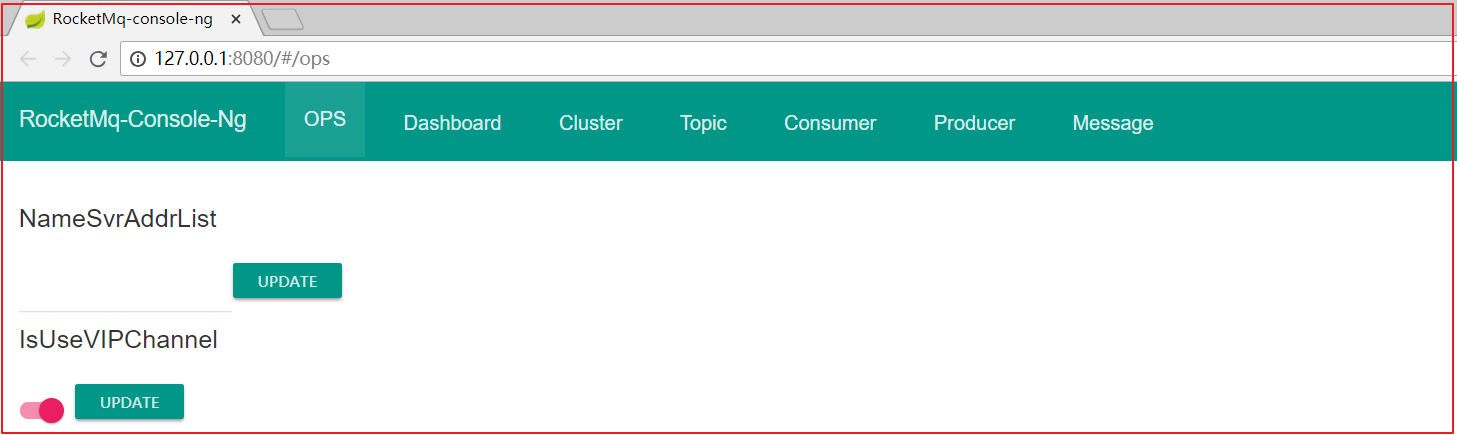
进入 \rocketmq-externals\rocketmq-console 文件夹，执行 mvn clean package -Dmaven.test.skip=true ，编译打包。



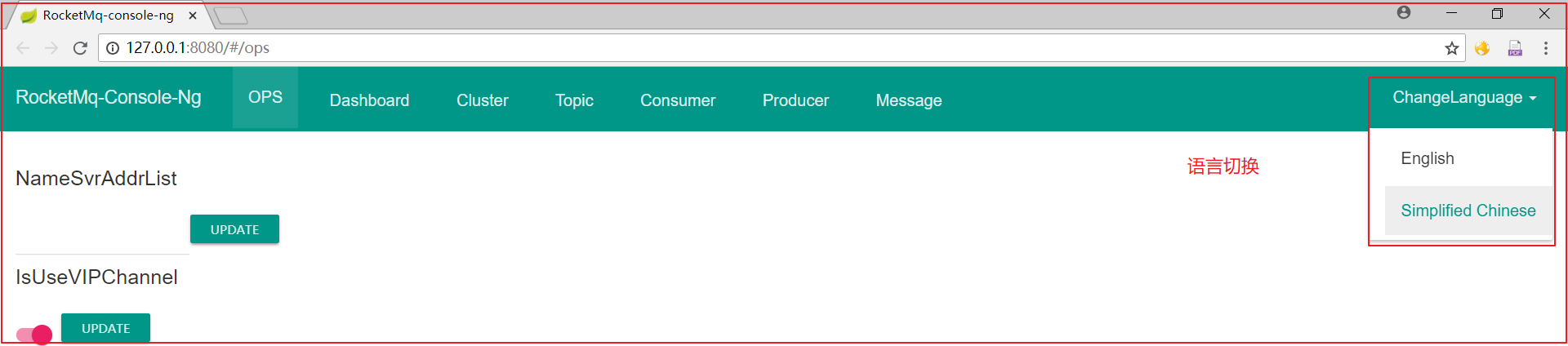
编译成功之后，cmd进入‘target’文件夹，执行java -jar rocketmq-console-ng-1.0.0.jar ，启动‘rocketmq-console-ng-1.0.0.jar’。

#### 1.3.4.3 测试

输入地址[http://127.0.0.1:8080/#/ops](http://127.0.0.1:8080/%23/ops)



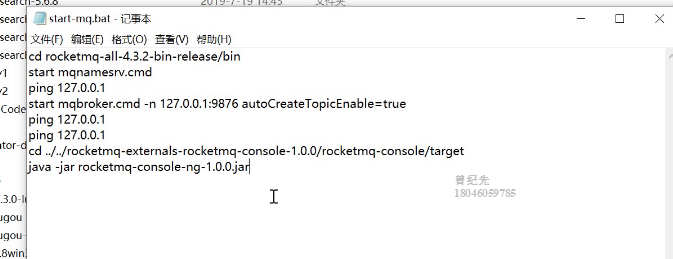
上面显示的是英文，如果想显示中文，可以点击语言切换，选中Chinese



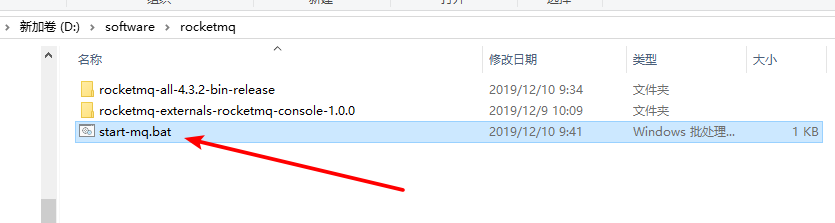
切换后如下图：



启动脚本：



### 启动脚本



cd rocketmq-all-4.3.2-bin-release/bin

start mqnamesrv.cmd

ping 127.0.0.1

start mqbroker.cmd -n 127.0.0.1:9876 autoCreateTopicEnable=true

ping 127.0.0.1

cd ../../rocketmq-externals-rocketmq-console-1.0.0/rocketmq-console/target

java -jar rocketmq-console-ng-1.0.0.jar

# 2. RocketMQ入门小Demo

RocketMQ消息消费类型有2种类型，我们先实现一次集群消息模式，再实现广播模式。集群模式也就是消息只能同时被一个消费者读取，而广播模式则可以同时被所有消费者读取。

## 2.1 同步消息

### 2.1.1 消息生产者

#### 创建工程rocketmq\_producer(jar)，引入依赖

|  |
| --- |
| *<!--依赖包-->* <**dependencies**>  <**dependency**>  <**groupId**>org.apache.rocketmq</**groupId**>  <**artifactId**>rocketmq-client</**artifactId**>  <**version**>4.3.2</**version**>  </**dependency**> </**dependencies**> |

#### 创建类com.itheima.mq.Producer如下：

消息发送步骤：

1.创建生产者-producer = new DefaultMQProducer(组名)

2.设置NameServer地址-producer.setNamesrvAddr(),如果集群环境，则用分号分隔

3.启动生产者-producer.start()

4.创建消息-message = new Message(主题名,标签名,消息key名,消息内容.getBytes(RemotingHelper.DEFAULT\_CHARSET));

5.发送消息，接收结果-sendResult = producer.send(message)

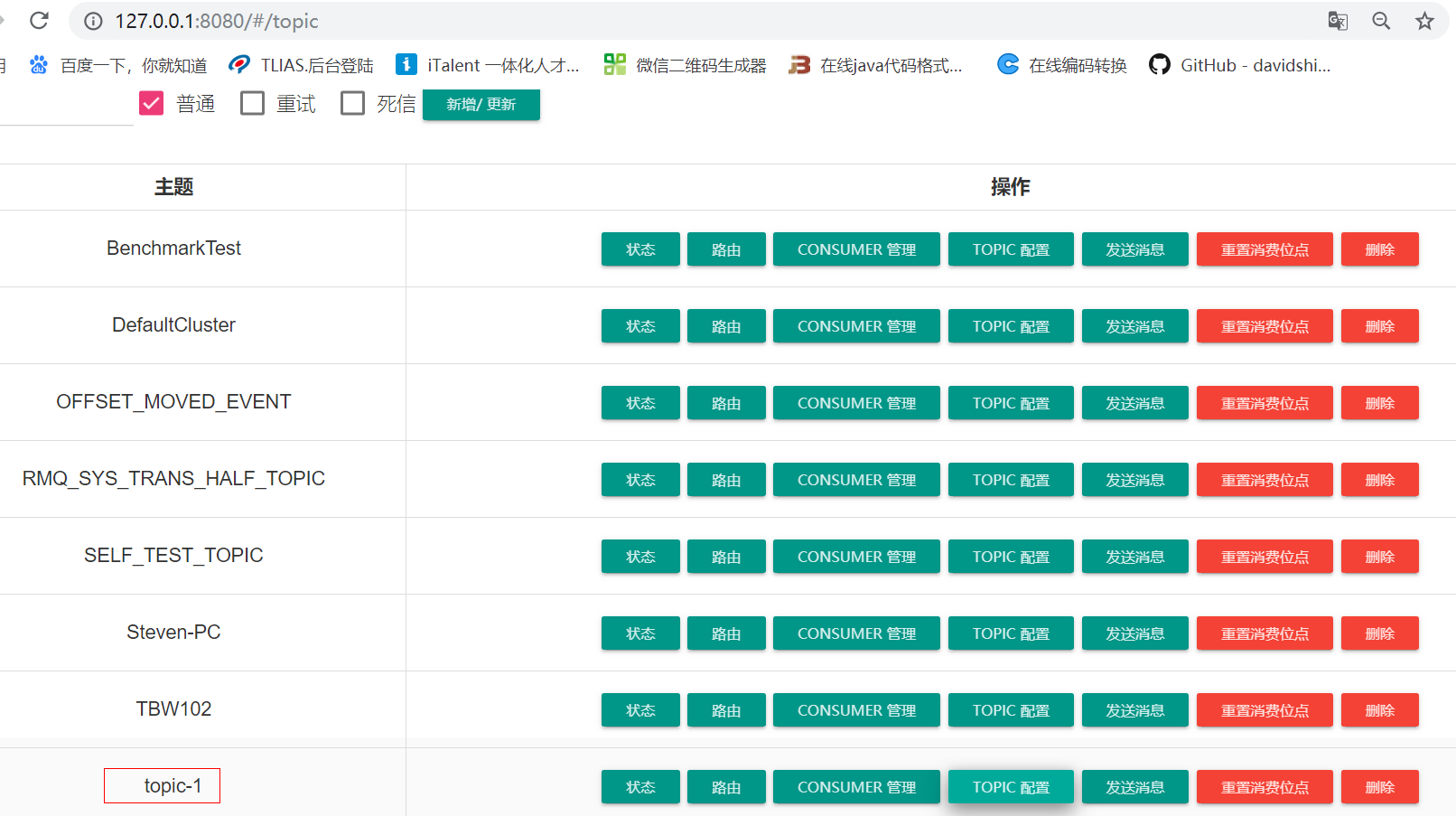
6.输出sendResult查看消息是否成功送达

7.如果不再发送消息，关闭生产者-producer.shutdown

代码：

|  |
| --- |
| */\*\*  \* rocketmq入门-生产者  \** ***@author*** *Steven  \** ***@description*** *com.itheima.mq  \*/* **public class** Producer {  */\*\*  \* 发送一条普通消息  \** ***@param args*** *\*/* **public static void** main(String[] args) **throws** Exception{  *//1.创建生产者-producer = new DefaultMQProducer(组名)* DefaultMQProducer producer = **new** DefaultMQProducer(**"producerGroup"**);  *//2.设置NameServer地址-producer.setNamesrvAddr(),如果集群环境，则用分号分隔* producer.setNamesrvAddr(**"127.0.0.1:9876"**);  *//3.启动生产者-producer.start()* producer.start();  *//4.创建消息-message = new Message(主题名,标签名,消息key名,消息内容.getBytes(RemotingHelper.DEFAULT\_CHARSET));* Message message = **new** Message(  **"topic-1"**,  **"tag-1"**,  **"key-1"**,  **"这是我第1次发送MQ消息"**.getBytes(RemotingHelper.***DEFAULT\_CHARSET***)  );  *//5.发送消息，接收结果-sendResult = producer.send(message)* SendResult sendResult = producer.send(message);  *//6.输出sendResult查看消息是否成功送达* System.***out***.println(sendResult);  *//7.如果不再发送消息，关闭生产者-producer.shutdown* producer.shutdown();  } } |

#### 执行main方法后，查看控制台主题信息，如下：



测试异常：运行生产者

**解决RocketMQ报No route info of this topic：异常**

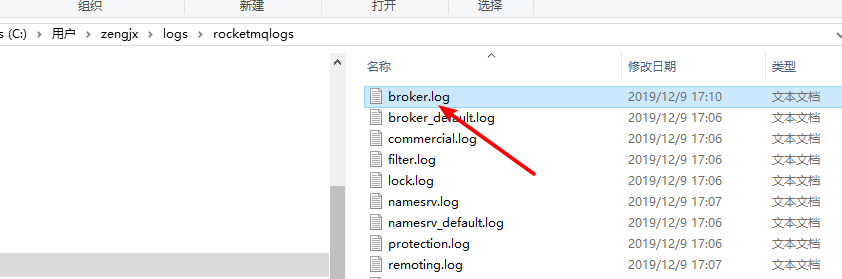
Exception in thread "main" org.apache.rocketmq.client.exception.MQClientException: No route info of this topic, 1234

See http://rocketmq.apache.org/docs/faq/ for further details.

**rocketmq-4.2.0在Windows下启动broker报load delayOffset.json failed错误问题解决**

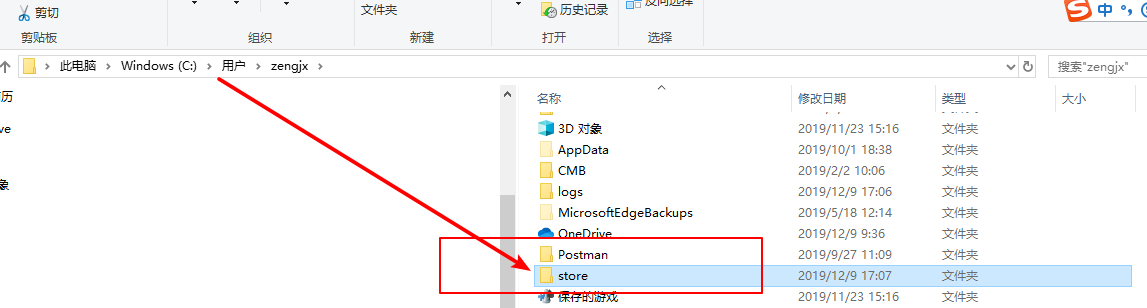
**https://blog.csdn.net/zhangjunli/article/details/80409018**

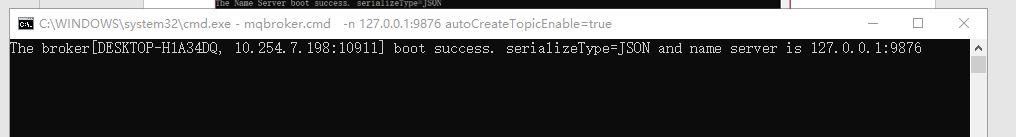
查看：



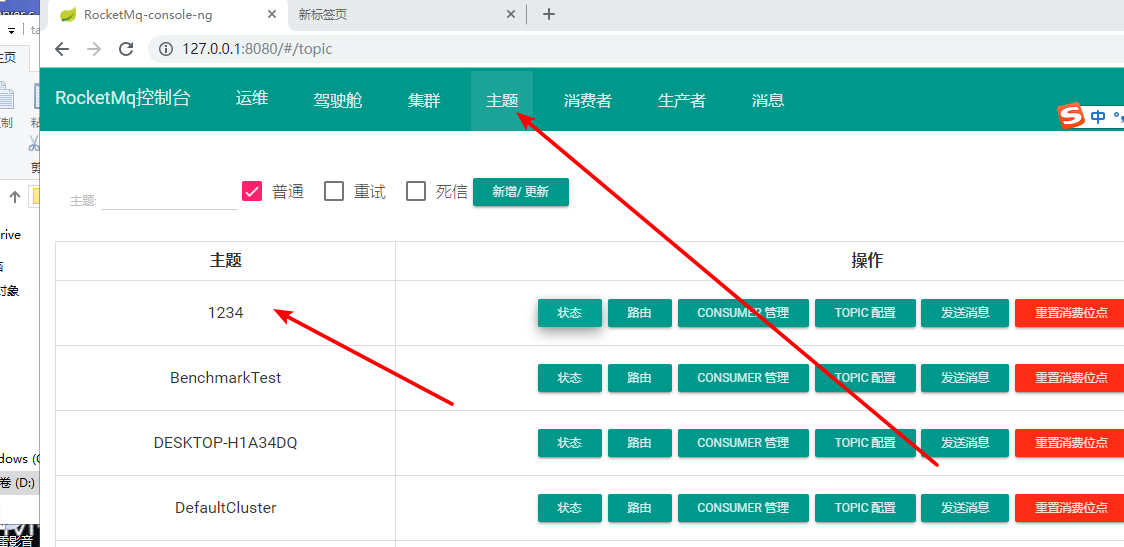
查看里面的报错信息

删除store：





查看主题：



说明发送成功。

### 2.1.2 消息消费者

创建工程rocketmq\_consumer(jar包)

#### 引入依赖

|  |
| --- |
| *<!--依赖-->* <**dependencies**>  <**dependency**>  <**groupId**>org.apache.rocketmq</**groupId**>  <**artifactId**>rocketmq-client</**artifactId**>  <**version**>4.3.2</**version**>  </**dependency**> </**dependencies**> |

#### 创建类com.itheima.mq.Consumer，main方法如下：

消息消费步骤：

1.创建推动式消费者-DefaultMQPushConsumer consumer = new..(consumerGroup)

2.设置NameServer地址，如果是集群环境，则用分号隔开-setNameservAddr

3.设置消息的主题(与生产者要一至)与标签(\*号代表所有)-consumer.subscribe("topic-1","\*")

4.创建消息监听器-consumer.setMessageListener(new MessageListenerConcurrently(){})

4.1.循环读取消息-msgs.for

4.1.1输出消息-主题、标签、消息key、内容(body)

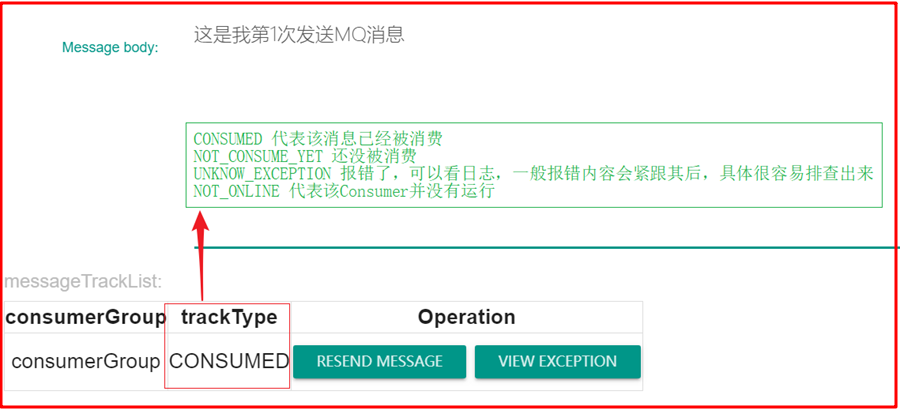
4.2.返回消息读取状态-CONSUME\_SUCCESS

5.启动消费者-consumer.start()

代码

|  |
| --- |
| */\*\*  \* rocketmq入门-消费者  \** ***@author*** *Steven  \** ***@description*** *com.itheima.mq  \*/* **public class** Consumer {  */\*\*  \* 读取同步消息  \** ***@param args*** *\*/* **public static void** main(String[] args) **throws** Exception{  *//1.创建推动式消费者-DefaultMQPushConsumer consumer = new..(consumerGroup)* DefaultMQPushConsumer consumer = **new** DefaultMQPushConsumer(**"consumerGroup"**);  *//2.设置NameServer地址，如果是集群环境，则用分号隔开-setNameservAddr* consumer.setNamesrvAddr(**"127.0.0.1:9876"**);  *//3.设置消息的主题(与生产者要一至)与标签(\*号代表所有)-consumer.subscribe("topic-1","\*")* consumer.subscribe(**"topic-1"**,**"\*"**);  *//4.创建消息监听器-consumer.setMessageListener(new MessageListenerConcurrently(){})* consumer.setMessageListener(**new** MessageListenerConcurrently() {  **public** ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  **try** {  *//4.1.循环读取消息-msgs.for* **for** (MessageExt msg : msgs) {  *//4.1.1输出消息-主题、标签、消息key、内容(body)* String topic = msg.getTopic(); *//主题* String tags = msg.getTags(); *//标签* String keys = msg.getKeys(); *//消息key  //内容* String body = **new** String(msg.getBody(), RemotingHelper.***DEFAULT\_CHARSET***);  System.***out***.println(**"topic:"** + topic + **",tag:"** + tags + **",key:"** + keys + **",body:"** + body);  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  *//4.2.返回消息读取状态-CONSUME\_SUCCESS* **return** ConsumeConcurrentlyStatus.***CONSUME\_SUCCESS***;  }  });  *//5.启动消费者-consumer.start()* consumer.start();  } } |

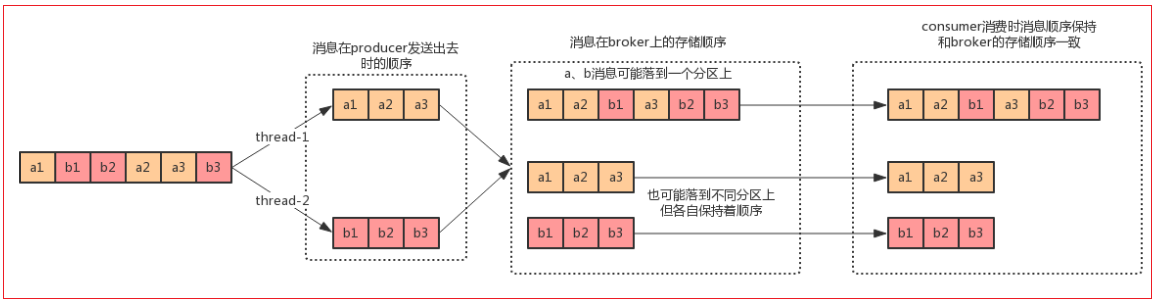
执行后可以查看rocketmq-console



## 2.2 RocketMQ顺序消息

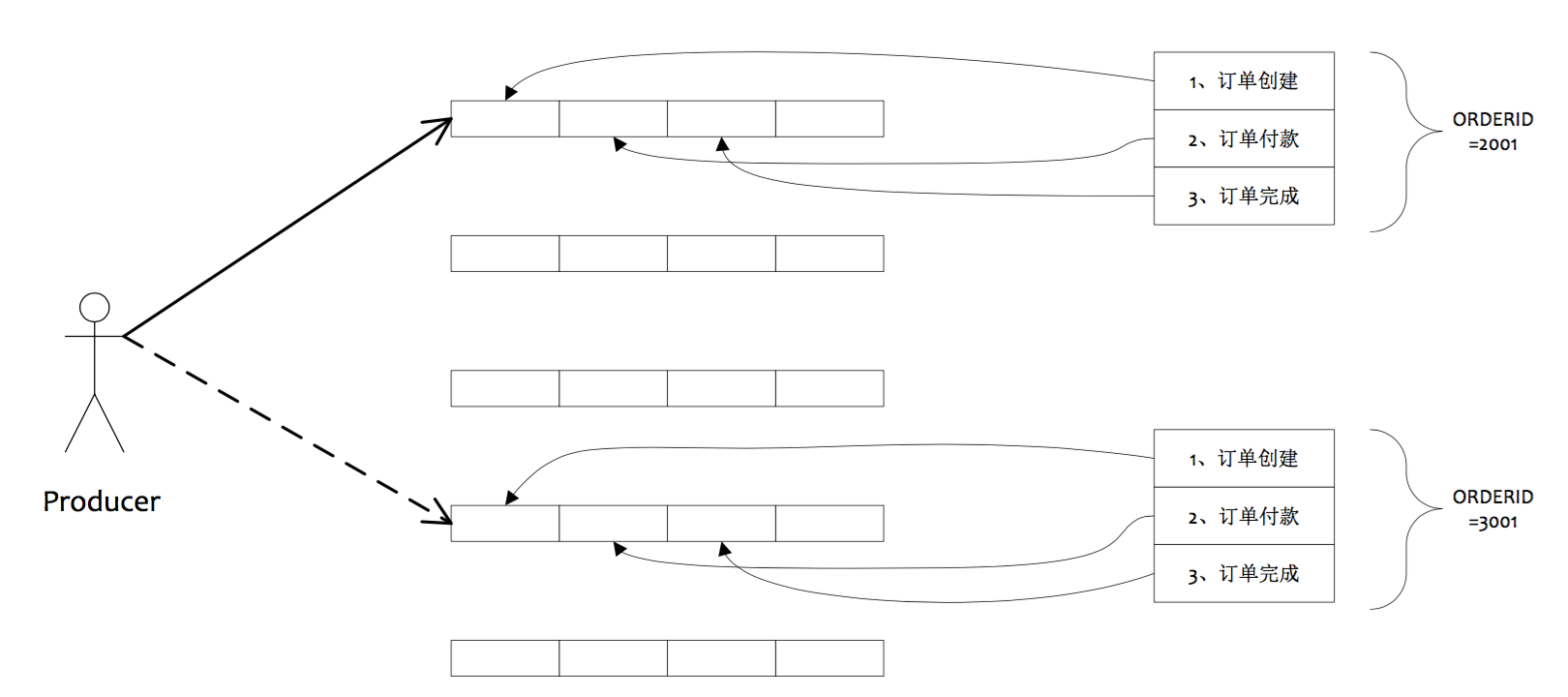
消息有序指的是可以按照消息的发送顺序来消费(FIFO)。RocketMQ可以严格的保证消息有序，可以分为分区有序或者全局有序。

顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue(分区队列)；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则就保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。



上图描述出了多个分区队列间有有序，执行流程

我们怎么实现全局有序？请看下图：



### 2.2.1全局顺序消息生产者

复制刚才的同步消息生产者，再改造一下

|  |
| --- |
| */\*\*  \* rocketmq入门-生产者-全局有序  \** ***@author*** *Steven  \** ***@description*** *com.itheima.mq  \*/* **public class** ProducerOrder {   */\*\*  \* 发送一条全局有序消息  \** ***@param args*** *\*/* **public static void** main(String[] args) **throws** Exception{  *//1.创建生产者-producer = new DefaultMQProducer(组名)* DefaultMQProducer producer = **new** DefaultMQProducer(**"producerGroup"**);  *//2.设置NameServer地址-producer.setNamesrvAddr(),如果集群环境，则用分号分隔* producer.setNamesrvAddr(**"127.0.0.1:9876"**);  *//3.启动生产者-producer.start()* producer.start();  *//4.创建消息-message = new Message(主题名,标签名,消息key名,消息内容.getBytes(RemotingHelper.DEFAULT\_CHARSET));* Message message = **new** Message(  **"topic-order"**,  **"tag-order"**,  **"key-order"**,  **"这是我第1次发送全局顺序MQ消息"**.getBytes(RemotingHelper.***DEFAULT\_CHARSET***)  );  *//5.发送消息，接收结果-sendResult = producer.send(message)* SendResult sendResult = producer.send(message, **new** MessageQueueSelector() {  */\*\*  \** ***@param mqs*** *单机RocketMQ的分区队列列表(4个)  \** ***@param msg*** *发送的消息  \** ***@param arg*** *send方法的第3个入参  \** ***@return*** *存储的分区队列**\*/* **public** MessageQueue select(List<MessageQueue> mqs, Message msg, Object arg) {  Integer id = (Integer) arg; *//获取外面第三个参数传入的id:18* **int** index = id % mqs.size(); *//用id求模队列总数，完成负载存储队列下标  //返回存储的分区* **return** mqs.get(index);  }  },18); *//此参数会传入select()方法的Object arg中，一般传入唯一id  //6.输出sendResult查看消息是否成功送达* System.***out***.println(sendResult);  *//7.如果不再发送消息，关闭生产者-producer.shutdown* producer.shutdown();  } } |

我们打开控制台，可以看到消息发送到了第2个队列中了。



### 2.2.2全局顺序消息消费者

复制刚才的同步消息消费者，再改造一下

|  |
| --- |
| */\*\*  \* rocketmq入门-消费者-全局有序  \** ***@author*** *Steven  \** ***@description*** *com.itheima.mq  \*/* **public class** ConsumerOrder {  */\*\*  \* 读取全局有序消息  \** ***@param args*** *\*/* **public static void** main(String[] args) **throws** Exception{  *//1.创建推动式消费者-DefaultMQPushConsumer consumer = new..(consumerGroup)* DefaultMQPushConsumer consumer = **new** DefaultMQPushConsumer(**"consumerGroup"**);  *//2.设置NameServer地址，如果是集群环境，则用分号隔开-setNameservAddr* consumer.setNamesrvAddr(**"127.0.0.1:9876"**);  *//3.设置消息的主题(与生产者要一至)与标签(\*号代表所有)-consumer.subscribe("topic-1","\*")* consumer.subscribe(**"topic-order"**,**"\*"**);  */\*\*  \* 设置Consumer第一次启动是从队列头部开始消费还是队列尾部开始消费<br>  \* 如果非第一次启动，那么按照上次消费的位置继续消费  \*/* consumer.setConsumeFromWhere(ConsumeFromWhere.***CONSUME\_FROM\_FIRST\_OFFSET***);  *//4.创建消息监听器-consumer.setMessageListener(new MessageListenerConcurrently(){})* consumer.setMessageListener(**new** MessageListenerOrderly() {  **public** ConsumeOrderlyStatus consumeMessage(List<MessageExt> msgs, ConsumeOrderlyContext context) {  **try** {  *//4.1.循环读取消息-msgs.for* **for** (MessageExt msg : msgs) {  *//4.1.1输出消息-主题、标签、消息key、内容(body)* String topic = msg.getTopic(); *//主题* String tags = msg.getTags(); *//标签* String keys = msg.getKeys(); *//消息key  //内容* String body = **new** String(msg.getBody(), RemotingHelper.***DEFAULT\_CHARSET***);  System.***out***.println(**"topic:"** + topic + **",tag:"** + tags + **",key:"** + keys + **",body:"** + body);  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  *//4.2.返回消息读取状态-CONSUME\_SUCCESS* **return** ConsumeOrderlyStatus.***SUCCESS***;  }  });  *//5.启动消费者-consumer.start()* consumer.start();  } } |

## 2.3批量发送

上面发送消息的时候，每次都是发送单条Message对象，能否批量发送呢？答案是肯定的。

### 2.3.1 消息生产者

复制同步消息生产者Producer，改名ProducerBatch改造代码如下：

|  |
| --- |
| */\*\*  \* rocketmq入门-生产者-批量发送  \** ***@author*** *Steven  \** ***@description*** *com.itheima.mq  \*/* **public class** ProducerBatch {   */\*\*  \* 发送批量同步消息  \** ***@param args*** *\*/* **public static void** main(String[] args) **throws** Exception{  *//1.创建生产者-producer = new DefaultMQProducer(组名)* DefaultMQProducer producer = **new** DefaultMQProducer(**"producerGroup"**);  *//2.设置NameServer地址-producer.setNamesrvAddr(),如果集群环境，则用分号分隔* producer.setNamesrvAddr(**"127.0.0.1:9876"**);  *//3.启动生产者-producer.start()* producer.start();  List<Message> messageList = **new** ArrayList<Message>();  *//4.创建消息-message = new Message(主题名,标签名,消息key名,消息内容.getBytes(RemotingHelper.DEFAULT\_CHARSET));* **for** (**int** i = 0; i < 10; i++) {  Message message = **new** Message(  **"topic-1"**,  **"tag-"** + i,  **"key-"** + i,  **"这是我第2次发送MQ消息"**.getBytes(RemotingHelper.***DEFAULT\_CHARSET***)  );  messageList.add(message);  }  *//5.发送消息，接收结果-sendResult = producer.send(message)* SendResult sendResult = producer.send(messageList);  *//6.输出sendResult查看消息是否成功送达* System.***out***.println(sendResult);  *//7.如果不再发送消息，关闭生产者-producer.shutdown* producer.shutdown();  } } |

## 2.4 广播模式消费

我们测试的时候，可以发现消息只有一个消费者能收到，如果我们想实现消息广播，让每个消费者都能收到消息也是可以实现的。

广播模式消费非常简单，只需要在消费方设置一下消费模式即可。

MessageModel.BROADCASTING:广播模式

MessageModel.CLUSTERING：集群模式

### 2.4.1 消息消费者

广播消费模式其实就是每个消费者都能读取到消息，我们这里只需要将消费者的消费模式设置成广播模式即可。consumer.setMessageModel(MessageModel.BROADCASTING)

测试时，同时开启2个以上的消费者，再次运行生产者，观察每个消费者控制台的输出，会发现所有启动起来的消费者都会接收到消息。

复制同步消息消费者Consumer，改名ConsumerBroadcasting并改造，代码如下：

|  |
| --- |
| */\*\*  \* rocketmq入门-消费者-广播模式  \** ***@author*** *Steven  \** ***@description*** *com.itheima.mq  \*/* **public class** ConsumerBroadcasting {  */\*\*  \* 读取一条普通消息  \** ***@param args*** *\*/* **public static void** main(String[] args) **throws** Exception{  *//1.创建推动式消费者-DefaultMQPushConsumer consumer = new..(consumerGroup)* DefaultMQPushConsumer consumer = **new** DefaultMQPushConsumer(**"consumerGroup"**);  *//2.设置NameServer地址，如果是集群环境，则用分号隔开-setNameservAddr* consumer.setNamesrvAddr(**"127.0.0.1:9876"**);  *//3.设置消息的主题(与生产者要一至)与标签(\*号代表所有)-consumer.subscribe("topic-1","\*")* consumer.subscribe(**"topic-1"**,**"\*"**);  *//设置消费者的消费接收模式为广播，默认为集群模式* consumer.setMessageModel(MessageModel.***BROADCASTING***);  *//4.创建消息监听器-consumer.setMessageListener(new MessageListenerConcurrently(){})* consumer.setMessageListener(**new** MessageListenerConcurrently() {  **public** ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  **try** {  *//4.1.循环读取消息-msgs.for* **for** (MessageExt msg : msgs) {  *//4.1.1输出消息-主题、标签、消息key、内容(body)* String topic = msg.getTopic(); *//主题* String tags = msg.getTags(); *//标签* String keys = msg.getKeys(); *//消息key  //内容* String body = **new** String(msg.getBody(), RemotingHelper.***DEFAULT\_CHARSET***);  System.***out***.println(**"topic:"** + topic + **",tag:"** + tags + **",key:"** + keys + **",body:"** + body);  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  *//4.2.返回消息读取状态-CONSUME\_SUCCESS* **return** ConsumeConcurrentlyStatus.***CONSUME\_SUCCESS***;  }  });  *//5.启动消费者-consumer.start()* consumer.start();  } } |

# 3.Spring整合RocketMQ

## 3.1 整合准备

Spring与RocketMQ的整合非常简单，我们只需将之前入门案例的生产者与消费者配置入IOC容器就行了

创建工程spring\_rocketmq

pom.xml引入依赖：

|  |
| --- |
| *<!--依赖包-->* <**dependencies**>  <**dependency**>  <**groupId**>org.apache.rocketmq</**groupId**>  <**artifactId**>rocketmq-client</**artifactId**>  <**version**>4.3.2</**version**>  </**dependency**>  *<!--Spring依赖包-->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>  *<!--测试依赖-->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**> </**dependencies**> |

## 3.2 集群模式消费

### 3.2.1消息生产者

#### 创建applicationContext-producer.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>   *<!--创建一个DefaultMQProducer  init-method:对象创建后调用方法  destroy-method:对象销毁后调用方法  -->* <**bean id="defaultMQProducer" class="org.apache.rocketmq.client.producer.DefaultMQProducer"  init-method="start" destroy-method="shutdown"**>  *<!--设置组名字-->* <**property name="producerGroup" value="spring-producer-group"** />  *<!--设置NameServer地址-->* <**property name="namesrvAddr" value="127.0.0.1:9876"** />  </**bean**> </**beans**> |

#### 创建消息发送测试类SpringMQTest

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(locations = **"classpath\*:applicationContent-\*.xml"**) **public class** SpringMQTest {  @Autowired  **private** DefaultMQProducer **producer**;   @Test  **public void** testSendMsg() **throws** Exception{  *//4.创建消息-message = new Message(主题名,标签名,消息key名,消息内容.getBytes(RemotingHelper.DEFAULT\_CHARSET));* Message message = **new** Message(  **"topic-spring"**,  **"tag-1"**,  **"key-1"**,  **"这是我第1次发送Spring-MQ消息"**.getBytes(RemotingHelper.***DEFAULT\_CHARSET***)  );  *//5.发送消息，接收结果-sendResult = producer.send(message)* SendResult sendResult = **producer**.send(message);  *//6.输出sendResult查看消息是否成功送达* System.***out***.println(sendResult);  } } |

### 3.2.2消息消费者

#### 创建applicationContext-consumer.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!--创建一个DefaultMQPushConsumer对象，用于消息接收-->* <**bean id="rocketmqConsumer" class="org.apache.rocketmq.client.consumer.DefaultMQPushConsumer" init-method="start" destroy-method="shutdown"**>  *<!--设置消费者组名-->* <**property name="consumerGroup" value="spring-consumer-group"**/>  *<!--设置NameServer地址-->* <**property name="namesrvAddr" value="127.0.0.1:9876"**/>  *<!--设置消息的主题(与生产者要一至)与标签(\*号代表所有)-->* <**property name="subscription"**>  <**map**>  *<!--指定对应主体和标签-->* <**entry key="topic-spring"**>  *<!--标签(\*号代表所有)-->* <**value**>\*</**value**>  </**entry**>  </**map**>  </**property**>  *<!--指定消息监听器，用于读取消息-->* <**property name="messageListener" ref="messageListener"**/>  </**bean**>  *<!--消息监听器-->* <**bean id="messageListener" class="com.itheima.mq.listener.MessageListener"** /> </**beans**> |

#### 创建监听器

|  |
| --- |
| */\*\*  \* 消息监听器  \** ***@author*** *Steven  \** ***@version*** *1.0  \** ***@description*** *com.itheima.mq.listener  \*/* **public class** MessageListener **implements** MessageListenerConcurrently {  **public** ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  **try** {  *//4.1.循环读取消息-msgs.for* **for** (MessageExt msg : msgs) {  *//4.1.1输出消息-主题、标签、消息key、内容(body)* String topic = msg.getTopic(); *//主题* String tags = msg.getTags(); *//标签* String keys = msg.getKeys(); *//消息key  //内容* String body = **new** String(msg.getBody(), RemotingHelper.***DEFAULT\_CHARSET***);  System.***out***.println(**"topic:"** + topic + **",tag:"** + tags + **",key:"** + keys + **",body:"** + body);  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  *//4.2.返回消息读取状态-CONSUME\_SUCCESS* **return** ConsumeConcurrentlyStatus.***CONSUME\_SUCCESS***;  } } |

#### 在SpringMQTest中添加测试方法

|  |
| --- |
| @Test **public void** testReadMsg(){  **try** {  *//等待用户输入，阻塞结束方法* System.***in***.read();  } **catch** (IOException e) {  e.printStackTrace();  } } |

## 3.3 广播模式消费

如果多个应用都需要接收消息，消息模式需要设置成广播消息模式，我们只需要修改一下rocketmq-consumer工程的applicationContent-consumer.xml配置文件即可。

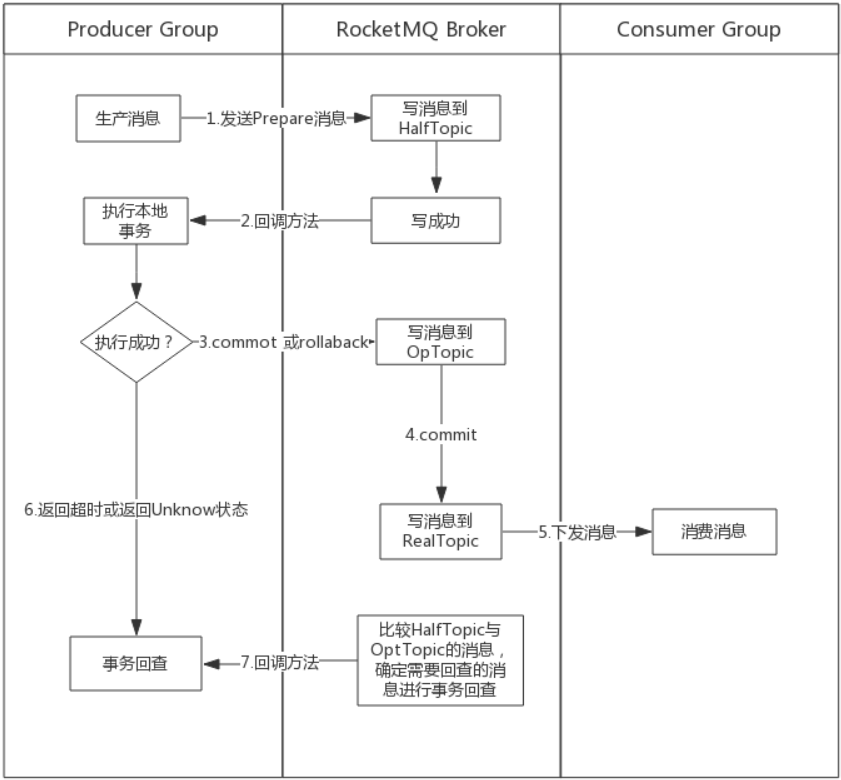
|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>  *<!--创建一个DefaultMQPushConsumer对象，用于消息接收-->* <**bean id="rocketmqConsumer" class="org.apache.rocketmq.client.consumer.DefaultMQPushConsumer" init-method="start" destroy-method="shutdown"**>  *<!--设置消费者组名-->* <**property name="consumerGroup" value="spring-consumer-group"**/>  *<!--设置NameServer地址-->* <**property name="namesrvAddr" value="127.0.0.1:9876"**/>  *<!--设置消息的主题(与生产者要一至)与标签(\*号代表所有)-->* <**property name="subscription"**>  <**map**>  *<!--指定对应主体和标签-->* <**entry key="topic-spring"**>  *<!--标签(\*号代表所有)-->* <**value**>\*</**value**>  </**entry**>  </**map**>  </**property**>  *<!--  消息模式设置  广播模式：BROADCASTING  集群模式：CLUSTERING  -->* <**property name="messageModel" value="BROADCASTING"**/>  *<!--指定消息监听器，用于读取消息-->* <**property name="messageListener" ref="messageListener"**/>  </**bean**>  *<!--消息监听器-->* <**bean id="messageListener" class="com.itheima.mq.listener.MessageListener"** /> </**beans**> |

## RocketMQ事务消息

在RocketMQ4.3.0版本后，开放了事务消息这一特性，对于分布式事务而言，最常说的还是二阶段提交协议。

### RocketMQ事务消息流程

RocketMQ的事务消息，主要是通过消息的异步处理，可以保证本地事务和消息发送同时成功执行或失败，从而保证数据的最终一致性，这里我们先看看一条事务消息从诞生到结束的整个时间线流程：



事务消息的成功投递是需要经历三个Topic的，分别是：

Half Topic：用于记录所有的prepare消息

Op Half Topic：记录已经提交了状态的prepare消息

Real Topic：事务消息真正的Topic,在Commit后会才会将消息写入该Topic，从而进行消息的投递

事务消息共有三种状态，提交状态、回滚状态、中间状态：

- TransactionStatus.CommitTransaction: 提交事务，它允许消费者消费此消息。

- TransactionStatus.RollbackTransaction: 回滚事务，它代表该消息将被删除，不允许被消费。

- TransactionStatus.Unknown: 中间状态，它代表需要检查消息队列来确定状态。

### 事务消息生产者

我们创建一个事务消息生产者TransactionProducer,事务消息发送消息对象是TransactionMQProducer，为了实现本地事务操作和回查，我们需要创建一个监听器，监听器需要实现TransactionListener接口，实现代码如下：

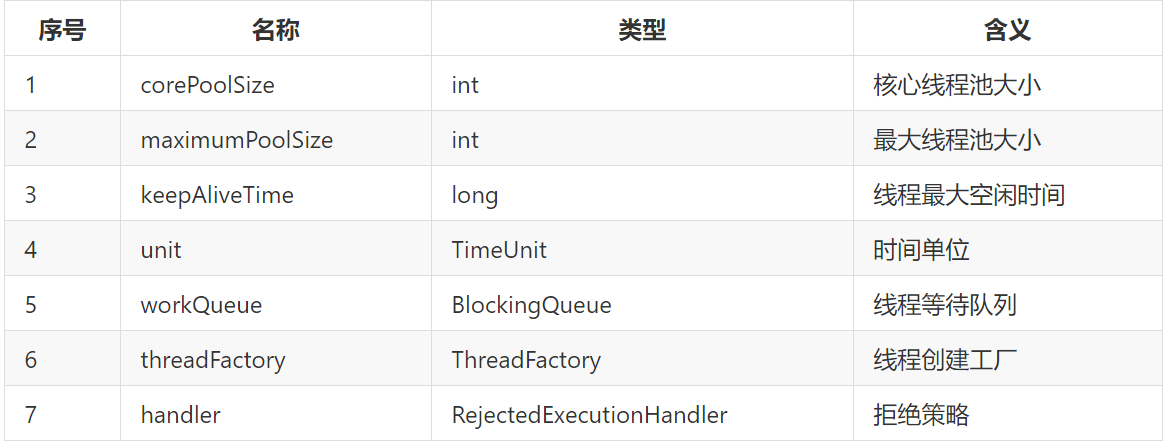
#### 监听器TransactionListenerImpl，代码如下：

**public class** TransactionListenerImpl **implements** TransactionListener {  
  
 *//存储当前线程对应的事务状态-线程安全的Map* **private** ConcurrentHashMap<String, Integer> **localTrans** = **new** ConcurrentHashMap<String, Integer>();  
 */\*\*\*  
 \* 发送prepare消息成功后回调该方法用于执行本地事务  
 \** ***@param msg****:回传的消息，利用transactionId即可获取到该消息的唯一Id  
 \** ***@param arg****:调用send方法时传递的参数，当send时候若有额外的参数可以传递到send方法中，这里能获取到  
 \** ***@return*** *\*/* @Override  
 **public** LocalTransactionState executeLocalTransaction(Message msg, Object arg) {  
 *//获取线程ID* String transactionId = msg.getTransactionId();  
 *//初始状态为0* **localTrans**.put(transactionId,0);  
 **try** {  
 *//此处执行本地事务操作* System.***out***.println(**"....执行本地事务"**);  
 Thread.*sleep*(70000);  
 System.***out***.println(**"....执行完成本地事务"**);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 *//发生异常，则回滚消息* **localTrans**.put(transactionId,2);  
 **return** LocalTransactionState.***UNKNOW***;  
 }  
 *//修改状态* **localTrans**.put(transactionId,1);  
 System.***out***.println(**"executeLocalTransaction------状态为1"**);  
 *//本地事务操作如果成功了，则提交该消息，让该消息可见* **return** LocalTransactionState.***UNKNOW***;  
 }  
 */\*\*\*  
 \* 消息回查-用于检查本地事务状态，并回应消息队列的检查请求  
 \** ***@param msg*** *\** ***@return*** *\*/* @Override  
 **public** LocalTransactionState checkLocalTransaction(MessageExt msg) {  
 *//获取事务id* String transactionId = msg.getTransactionId();  
 *//通过事务id获取对应的本地事务执行状态* Integer status = **localTrans**.get(transactionId);  
 System.***out***.println(**"消息回查-----"**+status);  
 **switch** (status){  
 **case** 0:  
 **return** LocalTransactionState.***UNKNOW***;  
 **case** 1:  
 **return** LocalTransactionState.***COMMIT\_MESSAGE***;  
 **case** 2:  
 **return** LocalTransactionState.***ROLLBACK\_MESSAGE***;  
 }  
 **return** LocalTransactionState.***UNKNOW***;  
 }  
}

#### 创建消息发送对象TransactionProducer,代码如下：

**public class** TransactionProducer {  
  
 **public static void** main(String[] args) **throws** MQClientException, UnsupportedEncodingException, InterruptedException {  
 *//创建事务消息发送对象* TransactionMQProducer producer = **new** TransactionMQProducer(**"transaction\_producer\_group\_name"**);  
 *//设置namesrv地址* producer.setNamesrvAddr(**"127.0.0.1:9876"**);  
 *//创建监听器* TransactionListener transactionListener = **new** TransactionListenerImpl();  
 *//创建线程池* ExecutorService executorService = **new** ThreadPoolExecutor(  
 2,  
 5,  
 100,  
 TimeUnit.***SECONDS***,  
 **new** ArrayBlockingQueue<Runnable>(  
 2000),  
 **new** ThreadFactory() {  
 @Override  
 **public** Thread newThread(Runnable runnable) {  
 Thread thread = **new** Thread(runnable);  
 thread.setName(**"client-transaction-msg-check-thread"**);  
 **return** thread;  
 }  
 }  
 );  
 *//设置线程池* producer.setExecutorService(executorService);  
 *//设置监听器* producer.setTransactionListener(transactionListener);  
 *//启动producer* producer.start();  
  
 *//创建消息* Message message = **new** Message(  
 **"topic-trans"**,  
 **"tags-trans-a"**,  
 **"keys-trans-a"**,  
 **"hello"**.getBytes(RemotingHelper.***DEFAULT\_CHARSET***));  
  
 *//发送事务消息,此时消息不可见，当前发送的是半消息，不能在消息里查到* TransactionSendResult transactionSendResult = producer.sendMessageInTransaction(message, **"发送消息，回传所需数据！"**);  
 System.***out***.println(transactionSendResult);  
  
 *//休眠* Thread.*sleep*(120000);  
 *//关闭* producer.shutdown();  
 }  
}

#### 小知识：上述用到了线程池，相关参数如下：





### 事务消息消费

事务消息的消费者和普通消费者一样，这里我们就不做介绍了，直接贴代码：

**public class** TransactionConsumer {  
   
 **public static void** main(String[] args) **throws** MQClientException {  
 *//创建DefaultMQPushConsumer* DefaultMQPushConsumer consumer = **new** DefaultMQPushConsumer(**"transaction\_consumer\_group\_name"**);  
 *//设置nameserver地址* consumer.setNamesrvAddr(**"127.0.0.1:9876"**);  
 *//设置每次拉去的消息个数* consumer.setConsumeMessageBatchMaxSize(5);  
 *//设置消费顺序* consumer.setConsumeFromWhere(ConsumeFromWhere.***CONSUME\_FROM\_FIRST\_OFFSET***);  
 *//设置监听的消息* consumer.subscribe(**"topic-trans"**,**"\*"**);  
 *//消息监听* consumer.registerMessageListener(**new** MessageListenerConcurrently() {  
 @Override  
 **public** ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext context) {  
 **try** {  
 **for** (MessageExt msg : msgs) {  
 String topic = msg.getTopic();  
 String tags = msg.getTags();  
 String keys = msg.getKeys();  
 String body = **new** String(msg.getBody(), RemotingHelper.***DEFAULT\_CHARSET***);  
 System.***out***.println(**"topic:"**+topic+**",tags:"**+tags+**",keys:"**+keys+**",body:"**+body);  
 }  
 } **catch** (UnsupportedEncodingException e) {  
 e.printStackTrace();  
 }  
 **return** ConsumeConcurrentlyStatus.***CONSUME\_SUCCESS***;  
 }  
 });  
 *//启动消费* consumer.start();  
 }  
}

### 分布式事务实现流程

MQ事务消息解决分布式事务问题，但第三方MQ支持事务消息的中间件不多，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。 以阿里的 RocketMQ 中间件为例，其思路大致为： 第一阶段Prepared消息，会拿到消息的地址。 第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。 也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

