图灵：楼兰
你的神秘技术宝藏

上一部分，我们可以搭建RocketMQ集群，然后也可以用命令行往RocketMQ写入消息并进行消费了。这一部分我们就来看怎么在项目中用上RocketMQ。

一、RocketMQ原生API使用

使用RocketMQ的原生API开发是最简单也是目前看来最牢靠的方式。这里我们用SpringBoot来搭建一系列消息生产者和消息消费者，来访问我们之前搭建的RocketMQ集群。

1、测试环境搭建

首先创建一个基于Maven的SpringBoot工程，引入如下依赖：

```
<dependency>
        <groupId>org.apache.rocketmq</groupId>
        <artifactId>rocketmq-client</artifactId>
        <version>4.7.1</version>
</dependency>
```

另外还与一些依赖，例如openmessage、acl等扩展功能还需要添加对应的依赖。具体可以参见RocketMQ源码中的example模块。在RocketMQ源码包中的example模块提供了非常详尽的测试代码，也可以拿来直接调试。我们这里就用源码包中的示例来连接我们自己搭建的RocketMQ集群来进行演示。

> RocketMQ的官网上有很多经典的测试代码，这些代码虽然依赖的版本比较老，但是还是都可以运行的。所以我们还是以官网上的顺序进行学习。

但是在调试这些代码的时候要注意一个问题：这些测试代码中的生产者和消费者都需要依赖NameServer才能运行，只需要将NameServer指向我们自己搭建的RocketMQ集群，而不需要管Broker在哪里，就可以连接我们自己的自己的RocketMQ集群。而RocketMQ提供的生产者和消费者寻找NameServer的方式有两种：

1、在代码中指定namesrvAddr属性。例如：consumer.setNamesrvAddr("127.0.0.1:9876");

2、通过NAMESRV_ADDR环境变量来指定。多个NameServer之间用分号连接。

2、RocketMQ的编程模型

然后RocketMQ的生产者和消费者的编程模型都是有个比较固定的步骤的，掌握这个固定的步骤，对于我们学习源码以及以后使用都是很有帮助的。

- 消息发送者的固定步骤

  1.创建消息生产者producer，并制定生产者组名
  2.指定Nameserver地址
  3.启动producer
  4.创建消息对象，指定主题Topic、Tag和消息体
  5.发送消息
  6.关闭生产者producer

- 消息消费者的固定步骤

  1.创建消费者Consumer，制定消费者组名
  2.指定Nameserver地址
  3.订阅主题Topic和Tag
  4.设置回调函数，处理消息
  5.启动消费者consumer

3、RocketMQ的消息样例

那我们来逐一连接下RocketMQ都支持哪些类型的消息：

3.1 基本样例

基本样例部分我们使用消息生产者分别通过三种方式发送消息，同步发送、异步发送以及单向发送。

然后使用消费者来消费这些消息。

1、同步发送消息的样例见：org.apache.rocketmq.example.simple.Producer

> 等待消息返回后再继续进行下面的操作。

2、异步发送消息的样例见：org.apache.rocketmq.example.simple.AsyncProducer

> 这个示例有个比较有趣的地方就是引入了一个countDownLatch来保证所有消息回调方法都执行完了再关闭Producer。 所以从这里可以看出，RocketMQ的Producer也是一个服务端，在往Broker发送消息的时候也要作为服务端提供服务。

3、单向发送消息的样例：

```
1 public class OnewayProducer {
2     public static void main(String[] args) throws Exception{
3         //Instantiate with a producer group name.
4         DefaultMQProducer producer = new
  DefaultMQProducer("please_rename_unique_group_name");
5         // Specify name server addresses.
```

```
        producer.setNamesrvAddr("localhost:9876");
        //Launch the instance.
        producer.start();
        for (int i = 0; i < 100; i++) {
            //Create a message instance, specifying topic, tag and message body.
            Message msg = new Message("TopicTest" /* Topic */,
                "TagA" /* Tag */,
                ("Hello RocketMQ " +
                    i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            //Call send message to deliver message to one of brokers.
            producer.sendOneway(msg);
        }
        //Wait for sending to complete
        Thread.sleep(5000);
        producer.shutdown();
    }
}
```

> 关键点就是使用producer.sendOneWay方式来发送消息，这个方法没有返回值，也没有回调。就是只管把消息发出去就行了。

4、使用消费者消费消息。

消费者消费消息有两种模式，一种是消费者主动去Broker上拉取消息的拉模式，另一种是消费者等待Broker把消息推送过来的推模式。

拉模式的样例见：org.apache.rocketmq.example.simple.PullConsumer

推模式的样例见：org.apache.rocketmq.example.simple.PushConsumer

> 通常情况下，用推模式比较简单。
>
> 实际上RocketMQ的推模式也是由拉模式封装出来的。
>
> 4.7.1版本中DefaultMQPullConsumerImpl这个消费者类已标记为过期，但是还是可以使用的。替换的类是DefaultLitePullConsumerImpl。

3.2 顺序消息

顺序消息生产者样例见：org.apache.rocketmq.example.order.Producer

顺序消息消费者样例见：org.apache.rocketmq.example.order.Consumer

> 验证时，可以启动多个Consumer实例，观察下每一个订单的消息分配以及每个订单下多个步骤的消费顺序。
>
> 不管订单在多个Consumer实例之前是如何分配的，每个订单下的多条消息顺序都是固定从0~5的。
>
> RocketMQ保证的是消息的局部有序，而不是全局有序。

> 先从控制台上看下List mqs是什么。
>
> 再回看我们的样例，实际上，RocketMQ也只保证了每个OrderID的所有消息有序(发到了同一个queue)，而并不能保证所有消息都有序。所以这就涉及到了RocketMQ消息有序的原理。要保证最终消费到的消息是有序的，需要从Producer、Broker、Consumer三个步骤都保证消息有序才行。

> 首先在发送者端：在默认情况下，消息发送者会采取Round Robin轮询方式把消息发送到不同的MessageQueue(分区队列)，而消费者消费的时候也从多个MessageQueue上拉取消息，这种情况下消息是不能保证顺序的。而只有当一组有序的消息发送到同一个MessageQueue上时，才能利用MessageQueue先进先出的特性保证这一组消息有序。
>
> 而Broker中一个队列内的消息是可以保证有序的。
>
> 然后在消费者端：消费者会从多个消息队列上去拿消息。这时虽然每个消息队列上的消息是有序的，但是多个队列之间的消息仍然是乱序的。消费者端要保证消息有序，就需要按队列一个一个来取消息，即取完一个队列的消息后，再去取下一个队列的消息。而给consumer注入的MessageListenerOrderly对象，在RocketMQ内部就会通过锁队列的方式保证消息是一个一个队列来取的。MessageListenerConcurrently这个消息监听器则不会锁队列，每次都是从多个Message中取一批数据（默认不超过32条）。因此也无法保证消息有序。

3.3 广播消息

广播消息的消息生产者样例见：org.apache.rocketmq.example.broadcast.PushConsumer

> 广播消息并没有特定的消息消费者样例，这是因为这涉及到消费者的集群消费模式。在集群状态(MessageModel.CLUSTERING)下，每一条消息只会被同一个消费者组中的一个实例消费到(这跟kafka和rabbitMQ的集群模式是一样的)。而广播模式则是把消息发给了所有订阅了对应主题的消费者，而不管消费者是不是同一个消费者组。

3.4 延迟消息

延迟消息的生产者案例

```
 public class ScheduledMessageProducer {

     public static void main(String[] args) throws Exception {
         // Instantiate a producer to send scheduled messages
         DefaultMQProducer producer = new
DefaultMQProducer("ExampleProducerGroup");
         // Launch producer
         producer.start();
         int totalMessagesToSend = 100;
         for (int i = 0; i < totalMessagesToSend; i++) {
             Message message = new Message("TestTopic", ("Hello scheduled
message " + i).getBytes());
             // This message will be delivered to consumer 10 seconds
later.
             message.setDelayTimeLevel(3);
             // Send the message
             producer.send(message);
         }

         // Shutdown producer after use.
         producer.shutdown();
     }

 }
```

> 延迟消息实现的效果就是在调用producer.send方法后，消息并不会立即发送出去，而是会等一段时间再发送出去。这是RocketMQ特有的一个功能。
>
> 那会延迟多久呢？延迟时间的设置就是在Message消息对象上设置一个延迟级别
> message.setDelayTimeLevel(3);

> 开源版本的RocketMQ中，对延迟消息并不支持任意时间的延迟设定(商业版本中支持)，而是只支持18个固定的延迟级别，1到18分别对应messageDelayLevel=1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h。这从哪里看出来的？其实从rocketmq-console控制台就能看出来。而这18个延迟级别也支持自行定义，不过一般情况下最好不要自定义修改。
>
> 那这么好用的延迟消息是怎么实现的？这18个延迟级别除了在延迟消息中用，还有什么地方用到了？别急，我们会在后面部分进行详细讲解。

3.5 批量消息

批量消息是指将多条消息合并成一个批量消息，一次发送出去。这样的好处是可以减少网络IO，提升吞吐量。

批量消息的消息生产者样例见：org.apache.rocketmq.example.batch.SimpleBatchProducer和org.apache.rocketmq.example.batch.SplitBatchProducer

> 相信大家在官网以及测试代码中都看到了关键的注释：如果批量消息大于1MB就不要用一个批次发送，而要拆分成多个批次消息发送。也就是说，一个批次消息的大小不要超过1MB
>
> 实际使用时，这个1MB的限制可以稍微扩大点，实际最大的限制是4194304字节，大概4MB。但是使用批量消息时，这个消息长度确实是必须考虑的一个问题。而且批量消息的使用是有一定限制的，这些消息应该有相同的Topic，相同的waitStoreMsgOK。而且不能是延迟消息、事务消息等。

3.6 过滤消息

在大多数情况下，可以使用Message的Tag属性来简单快速的过滤信息。

使用Tag过滤消息的消息生产者案例见：org.apache.rocketmq.example.filter.TagFilterProducer

使用Tag过滤消息的消息消费者案例见：org.apache.rocketmq.example.filter.TagFilterConsumer

> 主要是看消息消费者。consumer.subscribe("TagFilterTest", "TagA || TagC"); 这句只订阅TagA和TagC的消息。
>
> TAG是RocketMQ中特有的一个消息属性。RocketMQ的最佳实践中就建议，使用RocketMQ时，一个应用可以就用一个Topic，而应用中的不同业务就用TAG来区分。

但是，这种方式有一个很大的限制，就是一个消息只能有一个TAG，这在一些比较复杂的场景就有点不足了。 这时候，可以使用SQL表达式来对消息进行过滤。

SQL过滤的消息生产者案例见：org.apache.rocketmq.example.filter.SqlFilterProducer

SQL过滤的消息消费者案例见：org.apache.rocketmq.example.filter.SqlFilterConsumer

> 这个模式的关键是在消费者端使用MessageSelector.bySql(String sql)返回的一个MessageSelector。这里面的sql语句是按照SQL92标准来执行的。sql中可以使用的参数有默认的TAGS和一个在生产者中加入的a属性。
>
> SQL92语法：
>
> RocketMQ只定义了一些基本语法来支持这个特性。你也可以很容易地扩展它。
>
> - 数值比较，比如：**>，>=，<，<=，BETWEEN，=；**
> - 字符比较，比如：**=，<>，IN；**
> - **IS NULL** 或者 **IS NOT NULL；**
> - 逻辑符号 **AND，OR，NOT；**
>
> 常量支持类型为：
>
> - 数值，比如：**123，3.1415；**

> - 字符，比如：**'abc'，必须用单引号包裹起来；**
> - **NULL**，特殊的常量
> - 布尔值，**TRUE** 或 **FALSE**
>
> 使用注意：只有推模式的消费者可以使用SQL过滤。拉模式是用不了的。

> 大家想一下，这个消息过滤是在Broker端进行的还是在Consumer端进行的?

3.7 事务消息

这个事务消息是RocketMQ提供的一个非常有特色的功能，需要着重理解。

首先，我们了解下什么是事务消息。官网的介绍是：事务消息是在分布式系统中保证最终一致性的两阶段提交的消息实现。他可以保证本地事务执行与消息发送两个操作的原子性，也就是这两个操作一起成功或者一起失败。

其次，我们来理解下事务消息的编程模型。事务消息只保证消息发送者的本地事务与发消息这两个操作的原子性，因此，事务消息的示例只涉及到消息发送者，对于消息消费者来说，并没有什么特别的。

事务消息生产者的案例见：org.apache.rocketmq.example.transaction.TransactionProducer

> 事务消息的关键是在TransactionMQProducer中指定了一个TransactionListener事务监听器，这个事务监听器就是事务消息的关键控制器。源码中的案例有点复杂，我这里准备了一个更清晰明了的事务监听器示例

```
public class TransactionListenerImpl implements TransactionListener {
    //在提交完事务消息后执行。
    //返回COMMIT_MESSAGE状态的消息会立即被消费者消费到。
    //返回ROLLBACK_MESSAGE状态的消息会被丢弃。
    //返回UNKNOWN状态的消息会由Broker过一段时间再来回查事务的状态。
    @Override
    public LocalTransactionState executeLocalTransaction(Message msg,
Object arg) {
        String tags = msg.getTags();
        //TagA的消息会立即被消费者消费到
        if(StringUtils.contains(tags,"TagA")){
            return LocalTransactionState.COMMIT_MESSAGE;
        //TagB的消息会被丢弃
        }else if(StringUtils.contains(tags,"TagB")){
            return LocalTransactionState.ROLLBACK_MESSAGE;
        //其他消息会等待Broker进行事务状态回查。
        }else{
            return LocalTransactionState.UNKNOW;
        }
    }
    //在对UNKNOWN状态的消息进行状态回查时执行。返回的结果是一样的。
    @Override
    public LocalTransactionState checkLocalTransaction(MessageExt msg) {
        String tags = msg.getTags();
        //TagC的消息过一段时间会被消费者消费到
        if(StringUtils.contains(tags,"TagC")){
            return LocalTransactionState.COMMIT_MESSAGE;
        //TagD的消息也会在状态回查时被丢弃掉
        }else if(StringUtils.contains(tags,"TagD")){
            return LocalTransactionState.ROLLBACK_MESSAGE;
        //剩下TagE的消息会在多次状态回查后最终丢弃
        }else{
            return LocalTransactionState.UNKNOW;
```

```
        }
    }
}
```

然后，我们要了解下事务消息的使用限制：

1、事务消息不支持延迟消息和批量消息。

2、为了避免单个消息被检查太多次而导致半队列消息累积，我们默认将单个消息的检查次数限制为 15 次，但是用户可以通过 Broker 配置文件的 `transactionCheckMax` 参数来修改此限制。如果已经检查某条消息超过 N 次的话（ N = `transactionCheckMax` ） 则 Broker 将丢弃此消息，并在默认情况下同时打印错误日志。用户可以通过重写 `AbstractTransactionCheckListener` 类来修改这个行为。

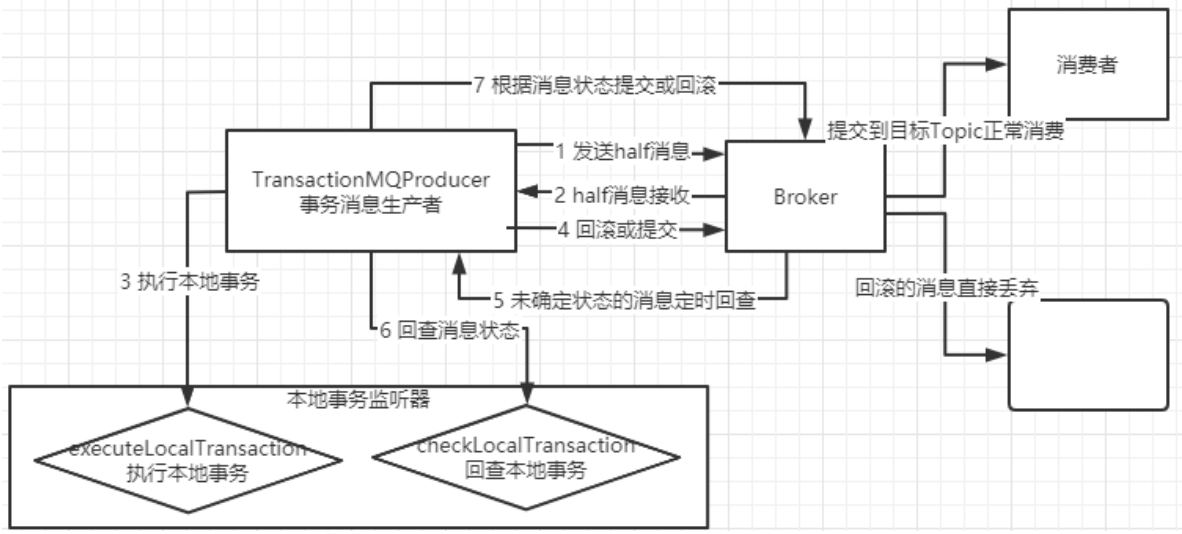
3、事务消息将在 Broker 配置文件中的参数 transactionMsgTimeout 这样的特定时间长度之后被检查。当发送事务消息时，用户还可以通过设置用户属性 CHECK_IMMUNITY_TIME_IN_SECONDS 来改变这个限制，该参数优先于 `transactionMsgTimeout` 参数。

4、事务性消息可能不止一次被检查或消费。

5、提交给用户的目标主题消息可能会失败，目前这依日志的记录而定。它的高可用性通过 RocketMQ 本身的高可用性机制来保证，如果希望确保事务消息不丢失、并且事务完整性得到保证，建议使用同步的双重写入机制。

6、事务消息的生产者 ID 不能与其他类型消息的生产者 ID 共享。与其他类型的消息不同，事务消息允许反向查询、MQ服务器能通过它们的生产者 ID 查询到消费者。

接下来，我们还要了解下事务消息的实现机制，参见下图：



事务消息机制的关键是在发送消息时，会将消息转为一个half半消息，并存入RocketMQ内部的一个 RMQ_SYS_TRANS_HALF_TOPIC 这个Topic，这样对消费者是不可见的。再经过一系列事务检查通过后，再将消息转存到目标Topic，这样对消费者就可见了。

最后，我们还需要思考下事务消息的作用。

大家想一下这个事务消息跟分布式事务有什么关系？为什么扯到了分布式事务相关的两阶段提交上了？事务消息只保证了发送者本地事务和发送消息这两个操作的原子性，但是并不保证消费者本地事务的原子性，所以，事务消息只保证了分布式事务的一半。但是即使这样，**对于复杂的分布式事务，RocketMQ提供的事务消息也是目前业内最佳的降级方案。**

3.8 ACL权限控制

权限控制（ACL）主要为RocketMQ提供Topic资源级别的用户访问控制。用户在使用RocketMQ权限控制时，可以在Client客户端通过 RPCHook注入AccessKey和SecretKey签名；同时，将对应的权限控制属性（包括Topic访问权限、IP白名单和AccessKey和SecretKey签名等）设置在$ROCKETMQ_HOME/conf/plain_acl.yml的配置文件中。Broker端对AccessKey所拥有的权限进行校验，校验不过，抛出异常； ACL客户端可以参考：**org.apache.rocketmq.example.simple**包下面的**AclClient**代码。

> 注意，如果要在自己的客户端中使用RocketMQ的ACL功能，还需要引入一个单独的依赖包
>
> ```
> <dependency>
>     <groupId>org.apache.rocketmq</groupId>
>     <artifactId>rocketmq-acl</artifactId>
>     <version>4.7.1</version>
> </dependency>
> ```

而Broker端具体的配置信息可以参见源码包下docs/cn/acl/user_guide.md。主要是在broker.conf中打开acl的标志：aclEnable=true。然后就可以用plain_acl.yml来进行权限配置了。并且这个配置文件是热加载的，也就是说要修改配置时，只要修改配置文件就可以了，不用重启Broker服务。我们来简单分析下源码中的plan_acl.yml的配置：

```
#全局白名单，不受ACL控制
#通常需要将主从架构中的所有节点加进来
globalWhiteRemoteAddresses:
- 10.10.103.*
- 192.168.0.*

accounts:
#第一个账户
- accessKey: RocketMQ
  secretKey: 12345678
  whiteRemoteAddress:
  admin: false
  defaultTopicPerm: DENY #默认Topic访问策略是拒绝
  defaultGroupPerm: SUB #默认Group访问策略是只允许订阅
  topicPerms:
  - topicA=DENY #topicA拒绝
  - topicB=PUB|SUB #topicB允许发布和订阅消息
  - topicC=SUB #topicC只允许订阅
  groupPerms:
  # the group should convert to retry topic
  - groupA=DENY
  - groupB=PUB|SUB
  - groupC=SUB
#第二个账户，只要是来自192.168.1.*的IP，就可以访问所有资源
- accessKey: rocketmq2
  secretKey: 12345678
  whiteRemoteAddress: 192.168.1.*
  # if it is admin, it could access all resources
  admin: true
```

二、SpringBoot整合RocketMQ

1、快速实战

这部分我们看下SpringBoot如何快速集成RocketMQ。

> 在使用SpringBoot的starter集成包时，要特别注意版本。因为SpringBoot集成RocketMQ的starter依赖是由Spring社区提供的，目前正在快速迭代的过程当中，不同版本之间的差距非常大，甚至基础的底层对象都会经常有改动。例如如果使用rocketmq-spring-boot-starter:2.0.4版本开发的代码，升级到目前最新的rocketmq-spring-boot-starter:2.1.1后，基本就用不了了。

我们创建一个maven工程，引入关键依赖：

```
<dependencies>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-spring-boot-starter</artifactId>
            <version>2.1.1</version>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-core</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.springframework</groupId>
                    <artifactId>spring-webmvc</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>2.1.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
            <version>2.9.2</version>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
            <version>2.9.2</version>
        </dependency>
    </dependencies>
```

> rocketmq-spring-boot-starter:2.1.1引入的SpringBoot包版本是2.0.5.RELEASE，这里把SpringBoot的依赖包升级了一下。

然后我们以SpringBoot的方式，快速创建一个简单的Demo

启动类：

```
@SpringBootApplication
public class RocketMQScApplication {

    public static void main(String[] args) {
        SpringApplication.run(RocketMQScApplication.class,args);
    }
}
```

配置文件 application.properties

```
#NameServer地址
rocketmq.name-server=192.168.232.128:9876
#默认的消息生产者组
rocketmq.producer.group=springBootGroup
```

消息生产者

```
package com.roy.rocket.basic;

import org.apache.rocketmq.client.exception.MQClientException;
import org.apache.rocketmq.client.producer.SendResult;
import org.apache.rocketmq.spring.annotation.RocketMQTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQTemplate;
import org.springframework.messaging.Message;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

import javax.annotation.Resource;
import java.io.UnsupportedEncodingException;

/**
 * @author : 楼兰
 * @date : Created in 2020/10/22
 * @description:
 **/
@Component
public class SpringProducer {

    @Resource
    private RocketMQTemplate rocketMQTemplate;
    //发送普通消息的示例
    public void sendMessage(String topic,String msg){
        this.rocketMQTemplate.convertAndSend(topic,msg);
    }
    //发送事务消息的示例
    public void sendMessageInTransaction(String topic,String msg) throws
InterruptedException {
        String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD",
"TagE"};
        for (int i = 0; i < 10; i++) {
            Message<String> message =
MessageBuilder.withPayload(msg).build();
            String destination =topic+":"+tags[i % tags.length];
            SendResult sendResult =
rocketMQTemplate.sendMessageInTransaction(destination,
message,destination);
```

```
            System.out.printf("%s%n", sendResult);

            Thread.sleep(10);
        }
    }
}
```

消息消费者

```
package com.roy.rocket.basic;

import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

/**
 * @author : 楼兰
 * @date : Created in 2020/10/22
 * @description:
 **/
@Component
@RocketMQMessageListener(consumerGroup = "MyConsumerGroup", topic =
"TestTopic")
public class SpringConsumer implements RocketMQListener<String> {
    @Override
    public void onMessage(String message) {
        System.out.println("Received message : "+ message);
    }
}

```

> SpringBoot集成RocketMQ，消费者部分的核心就在这个@RocketMQMessageListener注解上。所有消费者的核心功能也都会集成到这个注解中。所以我们还要注意下这个注解里面的属性：
>
> 例如：消息过滤可以由里面的selectorType属性和selectorExpression来定制
>
> 消息有序消费还是并发消费则由consumeMode属性定制。
>
> 消费者是集群部署还是广播部署由messageModel属性定制。

然后关于事务消息，还需要配置一个事务消息监听器：

```
package com.roy.rocket.config;

import org.apache.commons.lang3.StringUtils;
import org.apache.rocketmq.client.producer.LocalTransactionState;
import org.apache.rocketmq.spring.annotation.RocketMQTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQLocalTransactionListener;
import org.apache.rocketmq.spring.core.RocketMQLocalTransactionState;
import org.apache.rocketmq.spring.support.RocketMQUtil;
import org.springframework.messaging.Message;
import org.springframework.messaging.converter.StringMessageConverter;

import java.util.concurrent.ConcurrentHashMap;

/**
 * @author : 楼兰
```

```
 * @date : Created in 2020/11/5
 * @description:
 **/

@RocketMQTransactionListener(rocketMQTemplateBeanName = "rocketMQTemplate")
public class MyTransactionImpl implements RocketMQLocalTransactionListener
{

    private ConcurrentHashMap<Object, String> localTrans = new
ConcurrentHashMap<>();
    @Override
    public RocketMQLocalTransactionState executeLocalTransaction(Message
msg, Object arg) {
        Object id = msg.getHeaders().get("id");
        String destination = arg.toString();
        localTrans.put(id,destination);
        org.apache.rocketmq.common.message.Message message =
RocketMQUtil.convertToRocketMessage(new StringMessageConverter(),"UTF-
8",destination, msg);
        String tags = message.getTags();
        if(StringUtils.contains(tags,"TagA")){
            return RocketMQLocalTransactionState.COMMIT;
        }else if(StringUtils.contains(tags,"TagB")){
            return RocketMQLocalTransactionState.ROLLBACK;
        }else{
            return RocketMQLocalTransactionState.UNKNOWN;
        }
    }

    @Override
    public RocketMQLocalTransactionState checkLocalTransaction(Message msg)
{
        //SpringBoot的消息对象中，并没有transactionId这个属性。跟原生API不一样。
//        String destination = localTrans.get(msg.getTransactionId());
        return RocketMQLocalTransactionState.COMMIT;
    }
}

```

这样我们启动应用后，就能够通过访问 http://localhost:8080/MQTest/sendMessage?message=123 接口来发送一条简单消息。并在SpringConsumer中消费到。

也可以通过访问http://localhost:8080/MQTest/sendTransactionMessage?message=123 ，来发送一条事务消息。

> 这里可以看到，对事务消息，SpringBoot进行封装时，就缺少了transactionId，这在事务控制中是非常关键的。

2、其他更多消息类型：

对于其他的消息类型，文档中就不一一记录了。具体可以参见源码中的junit测试案例。

3、总结：

- SpringBoot 引入org.apache.rocketmq:rocketmq-spring-boot-starter依赖后，就可以通过内置的RocketMQTemplate来与RocketMQ交互。相关属性都以rockemq.开头。具体所有的配置信息

可以参见org.apache.rocketmq.spring.autoconfigure.RocketMQProperties这个类。

- SpringBoot依赖中的Message对象和RocketMQ-client中的Message对象是两个不同的对象，这在使用的时候要非常容易弄错。例如RocketMQ-client中的Message里的TAG属性，在SpringBoot依赖中的Message中就没有。Tag属性被移到了发送目标中，与Topic一起，以Topic:Tag的方式指定。
- 最后强调一次，一定要注意版本。rocketmq-spring-boot-starter的更新进度一般都会略慢于RocketMQ的版本更新，并且版本不同会引发很多奇怪的问题。apache有一个官方的rocketmq-spring示例，地址：https://github.com/apache/rocketmq-spring.git 以后如果版本更新了，可以参考下这个示例代码。

三、SpringCloudStream整合RocketMQ

SpringCloudStream是Spring社区提供的一个统一的消息驱动框架，目的是想要以一个统一的编程模型来对接所有的MQ消息中间件产品。我们还是来看看SpringCloudStream如何来集成RocketMQ。

1、快速实战

创建Maven工程，引入依赖：

```
<dependencies>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
            <version>4.7.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-acl</artifactId>
            <version>4.7.1</version>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-stream-rocketmq</artifactId>
            <version>2.2.3.RELEASE</version>
            <exclusions>
                <exclusion>
                    <groupId>org.apache.rocketmq</groupId>
                    <artifactId>rocketmq-client</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.apache.rocketmq</groupId>
                    <artifactId>rocketmq-acl</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>2.3.3.RELEASE</version>
        </dependency>
    </dependencies>
```

应用启动类：

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Sink;
import org.springframework.cloud.stream.messaging.Source;

/**
 * @author : 楼兰
 * @date : Created in 2020/10/22
 * @description:
 **/
@EnableBinding({Source.class, Sink.class})
@SpringBootApplication
public class ScRocketMQApplication {

    public static void main(String[] args) {
        SpringApplication.run(ScRocketMQApplication.class,args);
    }
}
```

> 注意这个@EnableBinding({Source.class, Sink.class})注解，这是SpringCloudStream引入的Binder配置。

然后增加配置文件application.properties

```properties
#ScStream通用的配置以spring.cloud.stream开头
spring.cloud.stream.bindings.input.destination=TestTopic
spring.cloud.stream.bindings.input.group=scGroup
spring.cloud.stream.bindings.output.destination=TestTopic
#rocketMQ的个性化配置以spring.cloud.stream.rocketmq开头
#spring.cloud.stream.rocketmq.binder.name-server=192.168.232.128:9876;192.168.232.129:9876;192.168.232.130:9876
spring.cloud.stream.rocketmq.binder.name-server=192.168.232.128:9876

```

> SpringCloudStream中，一个binding对应一个消息通道。这其中配置的input，是在Sink.class中定义的，对应一个消息消费者。而output，是在Source.class中定义的，对应一个消息生产者。

然后就可以增加消息消费者：

```java
package com.roy.scrocket.basic;

import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;
import org.springframework.stereotype.Component;

/**
 * @author : 楼兰
 * @date : Created in 2020/10/22
 * @description:
 **/
@Component
public class ScConsumer {

    @StreamListener(Sink.INPUT)
    public void onMessage(String messsage){

```

```
        System.out.println("received message:"+messsage+" from binding:"+Sink.INPUT);
    }
}

```

消息生产者：

```
package com.roy.scrocket.basic;

import org.apache.rocketmq.common.message.MessageConst;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHeaders;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Component;

import javax.annotation.Resource;
import java.util.HashMap;
import java.util.Map;

/**
 * @author : 楼兰
 * @date : Created in 2020/10/22
 * @description:
 **/
@Component
public class ScProducer {

    @Resource
    private Source source;

    public void sendMessage(String msg){
        Map<String, Object> headers = new HashMap<>();
        headers.put(MessageConst.PROPERTY_TAGS, "testTag");
        MessageHeaders messageHeaders = new MessageHeaders(headers);
        Message<String> message = MessageBuilder.createMessage(msg, messageHeaders);
        this.source.output().send(message);
    }
}

```

最后增加一个Controller类用于测试：

```
package com.roy.scrocket.controller;

import com.roy.scrocket.basic.ScProducer;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.Resource;

/**
 * @author : 楼兰
 * @date : Created in 2020/10/27
```

```
 * @description:
 **/
@RestController
@RequestMapping("/MQTest")
public class MQTestController {

    @Resource
    private ScProducer producer;
    @RequestMapping("/sendMessage")
    public String sendMessage(String message){
        producer.sendMessage(message);
        return "消息发送完成";
    }
}

```

启动应用后，就可以访问http://localhost:8080/MQTest/sendMessage?message=123，给RocketMQ发送一条消息到TestTopic，并在ScConsumer中消费到了。

2、总结

- 关于SpringCloudStream。这是一套几乎通用的消息中间件编程框架，例如从对接RocketMQ换到对接Kafka，业务代码几乎不需要动，只需要更换pom依赖并且修改配置文件就行了。但是，由于各个MQ产品都有自己的业务模型，差距非常大，所以使用使用SpringCloudStream时要注意业务模型转换。并且在实际使用中，要非常注意各个MQ的个性化配置属性。例如RocketMQ的个性化属性都是以spring.cloud.stream.rocketmq开头，只有通过这些属性才能用上RocketMQ的延迟消息、排序消息、事务消息等个性化功能。
- SpringCloudStream是Spring社区提供的一套统一框架，但是官方目前只封装了kafka、kafka Stream、RabbitMQ的具体依赖。而RocketMQ的依赖是交由厂商自己维护的，也就是由阿里巴巴自己来维护。这个维护力度显然是有不小差距的。所以一方面可以看到之前在使用SpringBoot时着重强调的版本问题，在使用SpringCloudStream中被放大了很多。spring-cloud-starter-stream-rocketmq目前最新的2.2.3.RELEASE版本中包含的rocketmq-client版本还是4.4.0。这个差距就非常大了。另一方面，RocketMQ这帮大神不屑于写文档的问题也特别严重，SpringCloudStream中关于RocketMQ的个性化配置几乎很难找到完整的文档。
- 总之，对于RocketMQ来说，SpringCloudStream目前来说还并不是一个非常好的集成方案。这方面跟kafka和Rabbit还没法比。所以使用时要慎重。

> 有道云分享链接：
>
> 文档：VIP02-RocketMQ开发模型.md
> 链接：http://note.youdao.com/noteshare?id=17381ee1f3d7a132a25d48ec98df2e91&sub=4A3A676F874A4404A4A3EDC81780DD97