图灵：楼兰

RocketMQ第五期增强版

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>4.9.1</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：

```
public class OnewayProducer {
    public static void main(String[] args) throws Exception{
        //Instantiate with a producer group name.
        DefaultMQProducer producer = new
DefaultMQProducer("please_rename_unique_group_name");
        // 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。

> 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。

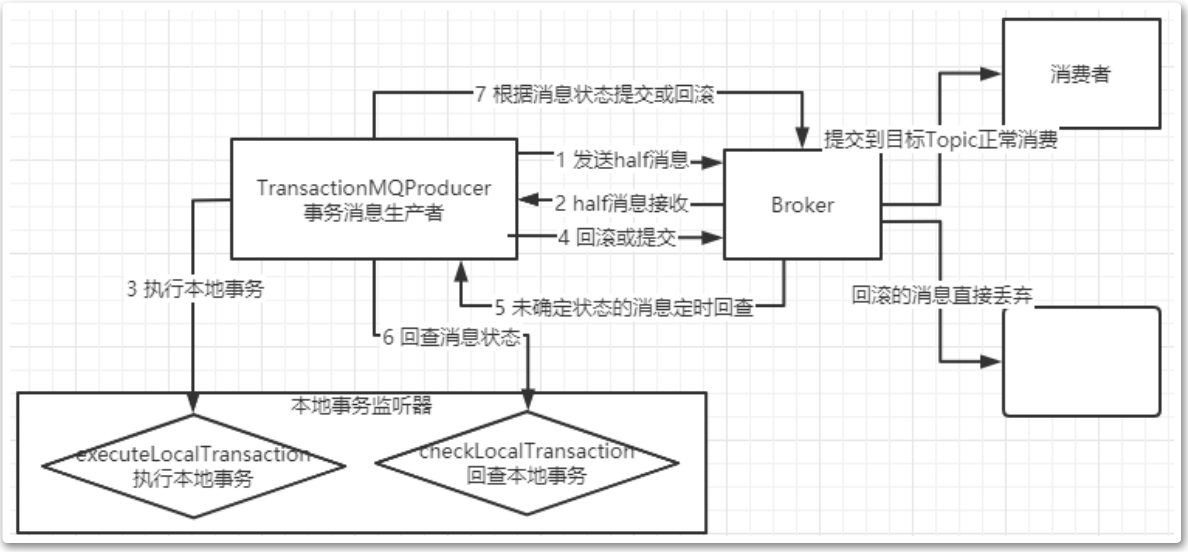
3、事务消息将在 Broker 配置文件中的参数 transactionMsgTimeout 这样的特定时间长度之后被检查。当发送事务消息时，用户还可以通过设置用户属性 CHECK_IMMUNITY_TIME_IN_SECONDS 来改变这个限制，该参数优先于 `transactionMsgTimeout` 参数。

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>4.9.1</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;
```

```
10
11  import javax.annotation.Resource;
12  import java.io.UnsupportedEncodingException;
13
14  /**
15   * @author : 楼兰
16   * @description:
17   **/
18  @Component
19  public class SpringProducer {
20
21      @Resource
22      private RocketMQTemplate rocketMQTemplate;
23      //发送普通消息的示例
24      public void sendMessage(String topic,String msg){
25          this.rocketMQTemplate.convertAndSend(topic,msg);
26      }
27      //发送事务消息的示例
28      public void sendMessageInTransaction(String topic,String msg) throws
    InterruptedException {
29          String[] tags = new String[] {"TagA", "TagB", "TagC", "TagD",
    "TagE"};
30          for (int i = 0; i < 10; i++) {
31              Message<String> message =
    MessageBuilder.withPayload(msg).build();
32              String destination =topic+":"+tags[i % tags.length];
33              SendResult sendResult =
    rocketMQTemplate.sendMessageInTransaction(destination,
    message,destination);
34              System.out.printf("%s%n", sendResult);
35
36              Thread.sleep(10);
37          }
38      }
39  }
```

消息消费者

```
1   package com.roy.rocket.basic;
2
3   import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
4   import org.apache.rocketmq.spring.core.RocketMQListener;
5   import org.springframework.stereotype.Component;
6
7   /**
8    * @author : 楼兰
9    * @description:
10   **/
```

```
@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
 * @description:
 **/

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

```

```
22      private ConcurrentHashMap<Object, String> localTrans = new
    ConcurrentHashMap<>();
23      @Override
24      public RocketMQLocalTransactionState executeLocalTransaction(Message
    msg, Object arg) {
25          Object id = msg.getHeaders().get("id");
26          String destination = arg.toString();
27          localTrans.put(id,destination);
28          org.apache.rocketmq.common.message.Message message =
    RocketMQUtil.convertToRocketMessage(new StringMessageConverter(),"UTF-
    8",destination, msg);
29          String tags = message.getTags();
30          if(StringUtils.contains(tags,"TagA")){
31              return RocketMQLocalTransactionState.COMMIT;
32          }else if(StringUtils.contains(tags,"TagB")){
33              return RocketMQLocalTransactionState.ROLLBACK;
34          }else{
35              return RocketMQLocalTransactionState.UNKNOWN;
36          }
37      }
38
39      @Override
40      public RocketMQLocalTransactionState checkLocalTransaction(Message msg)
    {
41          //SpringBoot的消息对象中，并没有transactionId这个属性。跟原生API不一样。
42  //          String destination = localTrans.get(msg.getTransactionId());
43          return RocketMQLocalTransactionState.COMMIT;
44      }
45  }
46
```

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。

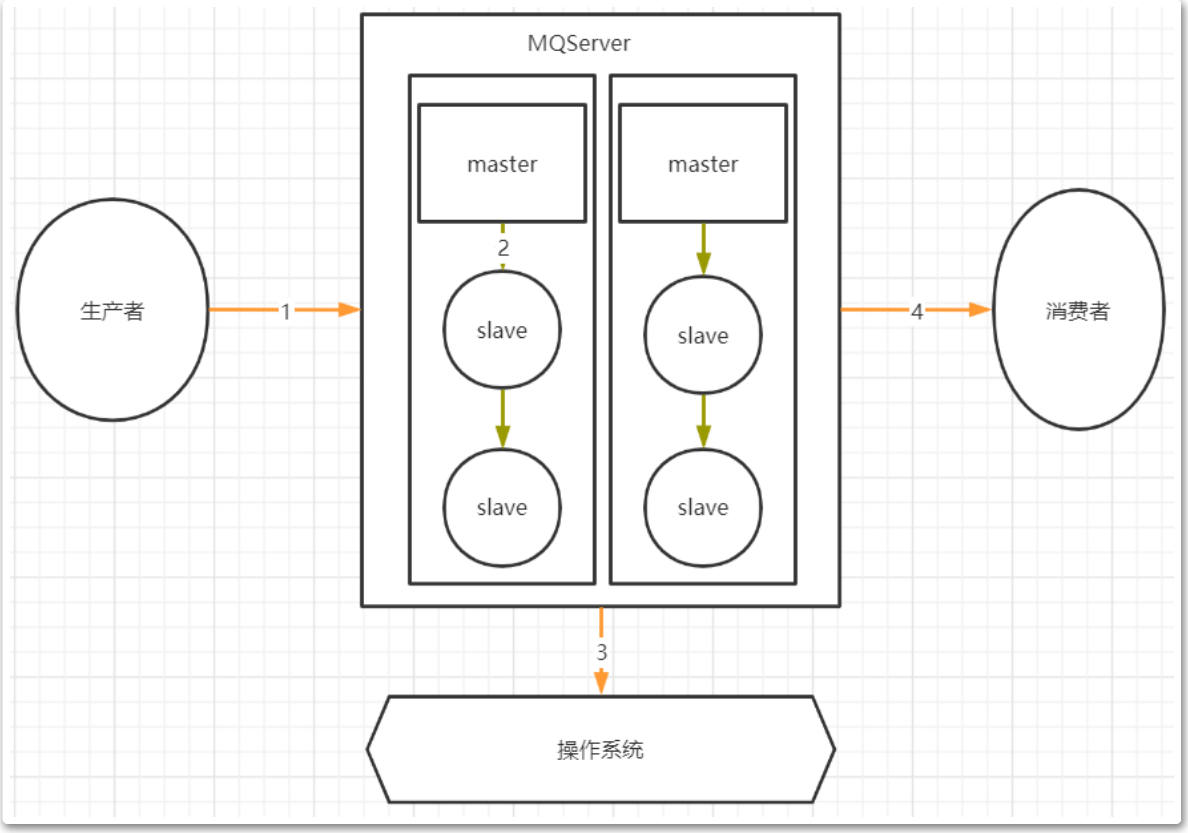
三、RocketMQ使用中常见的问题

3.1、使用RocketMQ如何保证消息不丢失?

这个是在面试时，关于MQ，面试官最喜欢问的问题。这个问题是所有MQ都需要面对的一个共性问题。大致的解决思路都是一致的，但是针对不同的MQ产品又有不同的解决方案。分析这个问题要从以下几个角度入手：

1、哪些环节会有丢消息的可能?

我们考虑一个通用的MQ场景：



其中，1，2，4三个场景都是跨网络的，而跨网络就肯定会有丢消息的可能。

然后关于3这个环节，通常MQ存盘时都会先写入操作系统的缓存page cache中，然后再由操作系统异步的将消息写入硬盘。这个中间有个时间差，就可能会造成消息丢失。如果服务挂了，缓存中还没有来得及写入硬盘的消息就会丢失。

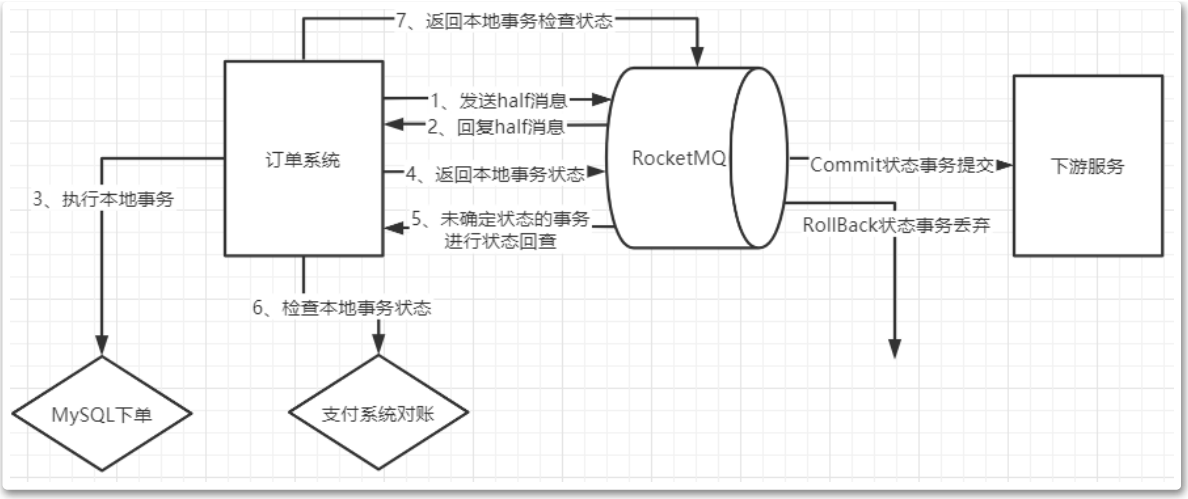
这个是MQ场景都会面对的通用的丢消息问题。那我们看看用RocketMQ时要如何解决这个问题

2、RocketMQ消息零丢失方案

1》 生产者使用事务消息机制保证消息零丢失

这个结论比较容易理解，因为RocketMQ的事务消息机制就是为了保证零丢失来设计的，并且经过阿里的验证，肯定是非常靠谱的。

但是如果深入一点的话，我们还是要理解下这个事务消息到底是不是靠谱。我们以最常见的电商订单场景为例，来简单分析下事务消息机制如何保证消息不丢失。我们看下下面这个流程图：



1、为什么要发送个half消息？有什么用？

这个half消息是在订单系统进行下单操作前发送，并且对下游服务的消费者是不可见的。那这个消息的作用更多的体现在确认RocketMQ的服务是否正常。相当于嗅探下RocketMQ服务是否正常，并且通知RocketMQ，我马上就要发一个很重要的消息了，你做好准备。

2.half消息如果写入失败了怎么办？

如果没有half消息这个流程，那我们通常是会在订单系统中先完成下单，再发送消息给MQ。这时候写入消息到MQ如果失败就会非常尴尬了。而half消息如果写入失败，我们就可以认为MQ的服务是有问题的，这时，就不能通知下游服务了。我们可以在下单时给订单一个状态标记，然后等待MQ服务正常后再进行补偿操作，等MQ服务正常后重新下单通知下游服务。

3.订单系统写数据库失败了怎么办？

这个问题我们同样比较下没有使用事务消息机制时会怎么办？如果没有使用事务消息，我们只能判断下单失败，抛出了异常，那就不往MQ发消息了，这样至少保证不会对下游服务进行错误的通知。但是这样的话，如果过一段时间数据库恢复过来了，这个消息就无法再次发送了。当然，也可以设计另外的补偿机制，例如将订单数据缓存起来，再启动一个线程定时尝试往数据库写。而如果使用事务消息机制，就可以有一种更优雅的方案。

如果下单时，写数据库失败(可能是数据库崩了，需要等一段时间才能恢复)。那我们可以另外找个地方把订单消息先缓存起来(Redis、文本或者其他方式)，然后给RocketMQ返回一个UNKNOWN状态。这样RocketMQ就会过一段时间来回查事务状态。我们就可以在回查事务状态时再尝试把订单数据写入数据库，如果数据库这

时候已经恢复了，那就能完整正常的下单，再继续后面的业务。这样这个订单的消息就不会因为数据库临时崩了而丢失。

4.half消息写入成功后RocketMQ挂了怎么办?

我们需要注意下，在事务消息的处理机制中，未知状态的事务状态回查是由RocketMQ的Broker主动发起的。也就是说如果出现了这种情况，那RocketMQ就不会回调到事务消息中回查事务状态的服务。这时，我们就可以将订单一直标记为"新下单"的状态。而等RocketMQ恢复后，只要存储的消息没有丢失，RocketMQ就会再次继续状态回查的流程。

5.下单成功后如何优雅的等待支付成功?

在订单场景下，通常会要求下单完成后，客户在一定时间内，例如10分钟，内完成订单支付，支付完成后才会通知下游服务进行进一步的营销补偿。

如果不用事务消息，那通常会怎么办?

最简单的方式是启动一个定时任务，每隔一段时间扫描订单表，比对未支付的订单的下单时间，将超过时间的订单回收。这种方式显然是有很大问题的，需要定时扫描很庞大的一个订单信息，这对系统是个不小的压力。

那更进一步的方案是什么呢？是不是就可以使用RocketMQ提供的延迟消息机制。往MQ发一个延迟1分钟的消息，消费到这个消息后去检查订单的支付状态，如果订单已经支付，就往下游发送下单的通知。而如果没有支付，就再发一个延迟1分钟的消息。最终在第十个消息时把订单回收。这个方案就不用对全部的订单表进行扫描，而只需要每次处理一个单独的订单消息。

那如果使用上了事务消息呢？我们就可以用事务消息的状态回查机制来替代定时的任务。在下单时，给Broker返回一个UNKNOWN的未知状态。而在状态回查的方法中去查询订单的支付状态。这样整个业务逻辑就会简单很多。我们只需要配置RocketMQ中的事务消息回查次数(默认15次)和事务回查间隔时间(messageDelayLevel)，就可以更优雅的完成这个支付状态检查的需求。

6、事务消息机制的作用

整体来说，在订单这个场景下，消息不丢失的问题实际上就还是转化成了下单这个业务与下游服务的业务的分布式事务一致性问题。而事务一致性问题一直以来都是一个非常复杂的问题。而RocketMQ的事务消息机制，实际上只保证了整个事务消息的一半，他保证的是订单系统下单和发消息这两个事件的事务一致性，而对下游

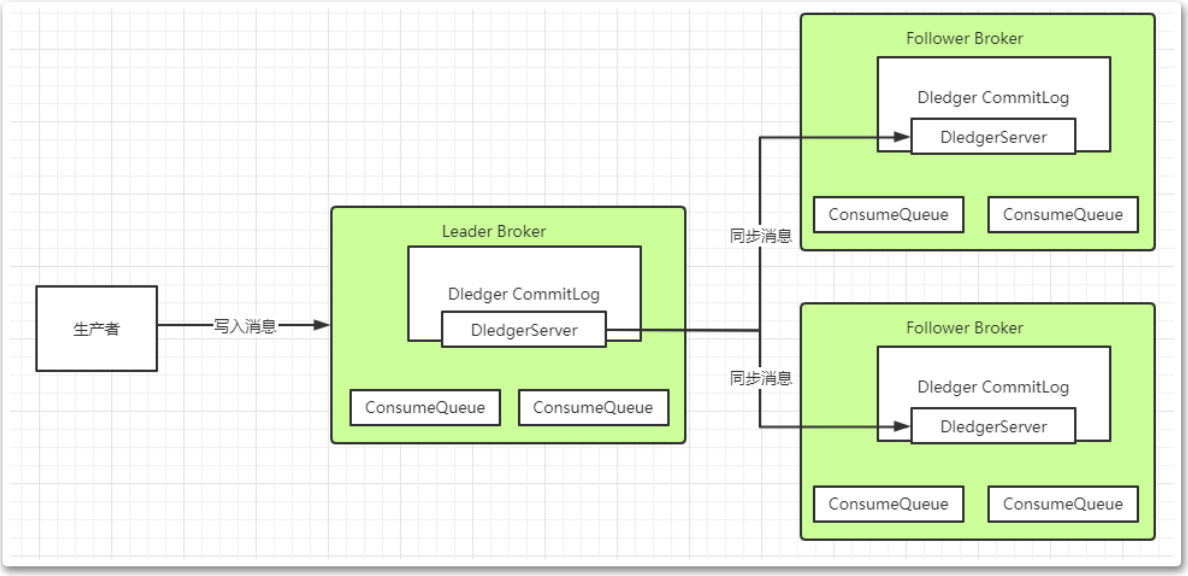
服务的事务并没有保证。但是即便如此，也是分布式事务的一个很好的降级方案。目前来看，也是业内最好的降级方案。

2》RocketMQ配置同步刷盘+Dledger主从架构保证MQ主从同步时不会丢消息

1、同步刷盘

这个从我们之前的分析，就很好理解了。我们可以简单的把RocketMQ的刷盘方式flushDiskType配置成同步刷盘就可以保证消息在刷盘过程中不会丢失了。

2、Dledger的文件同步



在使用Dledger技术搭建的RocketMQ集群中，Dledger会通过两阶段提交的方式保证文件在主从之间成功同步。

> 简单来说，数据同步会通过两个阶段，一个是uncommitted阶段，一个是commited阶段。
>
> Leader Broker上的Dledger收到一条数据后，会标记为uncommitted状态，然后他通过自己的DledgerServer组件把这个uncommitted数据发给Follower Broker的DledgerServer组件。
>
> 接着Follower Broker的DledgerServer收到uncommitted消息之后，必须返回一个ack给Leader Broker的Dledger。然后如果Leader Broker收到超过半数的Follower Broker返回的ack之后，就会把消息标记为committed状态。

> 再接下来， Leader Broker上的DledgerServer就会发送committed消息给Follower Broker上的DledgerServer，让他们把消息也标记为committed状态。这样，就基于Raft协议完成了两阶段的数据同步。

3》消费者端不要使用异步消费机制

正常情况下，消费者端都是需要先处理本地事务，然后再给MQ一个ACK响应，这时MQ就会修改Offset，将消息标记为已消费，从而不再往其他消费者推送消息。所以在Broker的这种重新推送机制下，消息是不会在传输过程中丢失的。但是也会有下面这种情况会造成服务端消息丢失：

```
 DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("please_rename_unique_group_name_4");
        consumer.registerMessageListener(new MessageListenerConcurrently() {
            @Override
            public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,

 ConsumeConcurrentlyContext context) {
                new Thread(){
                    public void run(){
                        //处理业务逻辑
                        System.out.printf("%s Receive New Messages: %s %n", Thread.currentThread().getName(), msgs);
                    }
                };
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            }
        });
```

这种异步消费的方式，就有可能造成消息状态返回后消费者本地业务逻辑处理失败造成消息丢失的可能。

4》RocketMQ特有的问题，NameServer挂了如何保证消息不丢失?

NameServer在RocketMQ中，是扮演的一个路由中心的角色，提供到Broker的路由功能。但是其实路由中心这样的功能，在所有的MQ中都是需要的。kafka是用zookeeper和一个作为Controller的Broker一起来提供路由服务，整个功能是相当复杂纠结的。而RabbitMQ是由每一个Broker来提供路由服务。而只有RocketMQ把这个路由中心单独抽取了出来，并独立部署。

这个NameServer之前都了解过，集群中任意多的节点挂掉，都不会影响他提供的路由功能。那**如果集群中所有的NameServer节点都挂了呢?**

有很多人就会认为在生产者和消费者中都会有全部路由信息的缓存副本，那整个服务可以正常工作一段时间。其实这个问题大家可以做一下实验，当NameServer全部挂了后，生产者和消费者是立即就无法工作了的。至于为什么，可以去源码中找找答案。

那再回到我们的消息不丢失的问题，在这种情况下，RocketMQ相当于整个服务都不可用了，那他本身肯定无法给我们保证消息不丢失了。我们只能自己设计一个降级方案来处理这个问题了。例如在订单系统中，如果多次尝试发送RocketMQ不成功，那就只能另外找给地方(Redis、文件或者内存等)把订单消息缓存下来，然后起一个线程定时的扫描这些失败的订单消息，尝试往RocketMQ发送。这样等RocketMQ的服务恢复过来后，就能第一时间把这些消息重新发送出去。整个这套降级的机制，在大型互联网项目中，都是必须要有的。

5》RocketMQ消息零丢失方案总结

完整分析过后，整个RocketMQ消息零丢失的方案其实挺简单

- 生产者使用事务消息机制。
- Broker配置同步刷盘+Dledger主从架构
- 消费者不要使用异步消费。
- 整个MQ挂了之后准备降级方案

那这套方案是不是就很完美呢？其实很明显，这整套的消息零丢失方案，在各个环节都大量的降低了系统的处理性能以及吞吐量。在很多场景下，这套方案带来的性能损失的代价可能远远大于部分消息丢失的代价。所以，我们在设计RocketMQ使用方案时，要根据实际的业务情况来考虑。例如，如果针对所有服务器都在同一个机房的场景，完全可以把Broker配置成异步刷盘来提升吞吐量。而在有些对消息可靠性要求没有那么高的场景，在生产者端就可以采用其他一些更简单的方案来提升吞吐，而采用定时对账、补偿的机制来提高消息的可靠性。而如果消费者不需要进行消息存盘，那使用异步消费的机制带来的性能提升也是非常显著的。

总之，这套消息零丢失方案的总结是为了在设计RocketMQ使用方案时的一个很好的参考。

3.2、使用RocketMQ如何保证消息顺序

1、为什么要保证消息有序?

这个也是面试时最常见的问题，需要对MQ场景有一定的深入理解。例如如果我们有个大数据系统，需要对业务系统的日志进行收集分析，这时候为了减少对业务系统的影响，通常都会通过MQ来做消息中转。而这时候，对消息的顺序就有一定的要求了。例如我们考虑下面这一系列的操作。

1. 用户的积分默认是0分，而新注册用户设置为默认的10分。
2. 用户有奖励行为，积分+2分。
3. 用户有不正当行为，积分-3分。

这样一组操作，正常用户积分要变成9分。但是如果顺序乱了，这个结果就全部对不了。这时，就需要对这一组操作，保证消息都是有序的。

2、如何保证消息有序?

MQ的顺序问题分为全局有序和局部有序。

- 全局有序：整个MQ系统的所有消息严格按照队列先入先出顺序进行消费。
- 局部有序：只保证一部分关键消息的消费顺序。

首先 我们需要分析下这个问题，在通常的业务场景中，全局有序和局部有序哪个更重要？其实在大部分的MQ业务场景，我们只需要能够保证局部有序就可以了。例如我们用QQ聊天，只需要保证一个聊天窗口里的消息有序就可以了。而对于电商订单场景，也只要保证一个订单的所有消息是有序的就可以了。至于全局消息的顺序，并不会太关心。而通常意义下，全局有序都可以压缩成局部有序的问题。例如以前我们常用的聊天室，就是个典型的需要保证消息全局有序的场景。但是这种场景，通常可以压缩成只有一个聊天窗口的QQ来理解。即整个系统只有一个聊天通道，这样就可以用QQ那种保证一个聊天窗口消息有序的方式来保证整个系统的全局消息有序。

然后 落地到RocketMQ。通常情况下，发送者发送消息时，会通过MessageQueue轮询的方式保证消息尽量均匀的分布到所有的MessageQueue上，而消费者也就同样需要从多个MessageQueue上消费消息。而MessageQueue是RocketMQ存储消息的最小单元，他们之间的消息都是互相隔离的，在这种情况下，是无法保证消息全局有序的。

而对于局部有序的要求，只需要将有序的一组消息都存入同一个MessageQueue里，这样MessageQueue的FIFO设计天生就可以保证这一组消息的有序。RocketMQ中，可以在发送者发送消息时指定一个MessageSelector对象，让这个对象来决定消息发入哪一个MessageQueue。这样就可以保证一组有序的消息能够发到同一个MessageQueue里。

另外，通常所谓的保证Topic全局消息有序的方式，就是将Topic配置成只有一个MessageQueue队列(默认是4个)。这样天生就能保证消息全局有序了。这个说法其实就是我们将聊天室场景压缩成只有一个聊天窗口的QQ一样的理解方式。而这种方式对整个Topic的消息吞吐影响是非常大的，如果这样用，基本上就没有用MQ的必要了。

3.3、使用RocketMQ如何快速处理积压消息?

1、如何确定RocketMQ有大量的消息积压?

在正常情况下，使用MQ都会要尽量保证他的消息生产速度和消费速度整体上是平衡的，但是如果部分消费者系统出现故障，就会造成大量的消息积累。这类问题通常在实际工作中会出现得比较隐蔽。例如某一天一个数据库突然挂了，大家大概率就会集中处理数据库的问题。等好不容易把数据库恢复过来了，这时基于这个数据库服务的消费者程序就会积累大量的消息。或者网络波动等情况，也会导致消息大量的积累。这在一些大型的互联网项目中，消息积压的速度是相当恐怖的。所以消息积压是个需要时时关注的问题。

对于消息积压，如果是RocketMQ或者kafka还好，他们的消息积压不会对性能造成很大的影响。而如果是RabbitMQ的话，那就惨了，大量的消息积压可以瞬间造成性能直线下滑。

对于RocketMQ来说，有个最简单的方式来确定消息是否有积压。那就是使用web控制台，就能直接看到消息的积压情况。

在Web控制台的主题页面，可以通过 Consumer管理 按钮实时看到消息的积压情况。


TestTopic订阅组

订阅组	MyConsumerGroup	延迟	0	最后消费时间	2020-11-28 16:53:35

Broker	队列	消费者终端	代理者位点	消费者位点	差值	上次时间
worker1	0		9	9	0	2020-11-28 16:53:35
worker1	1		10	10	0	2020-11-28 16:53:35
worker1	2		7	7	0	2020-11-28 16:53:35
worker1	3		7	7	0	2020-11-28 16:53:35

另外，也可以通过mqadmin指令在后台检查各个Topic的消息延迟情况。

还有RocketMQ也会在他的 ${storePathRootDir}/config 目录下落地一系列的json文件，也可以用来跟踪消息积压情况。

2、如何处理大量积压的消息?

其实我们回顾下RocketMQ的负载均衡的内容就不难想到解决方案。

如果Topic下的MessageQueue配置得是足够多的，那每个Consumer实际上会分配多个MessageQueue来进行消费。这个时候，就可以简单的通过增加Consumer的服务节点数量来加快消息的消费，等积压消息消费完了，再恢复成正常情况。最极限的情况是把Consumer的节点个数设置成跟MessageQueue的个数相同。但是如果此时再继续增加Consumer的服务节点就没有用了。

而如果Topic下的MessageQueue配置得不够多的话，那就不能用上面这种增加Consumer节点个数的方法了。这时怎么办呢？ 这时如果要快速处理积压的消息，可以创建一个新的Topic，配置足够多的MessageQueue。然后把所有消费者节点的目标Topic转向新的Topic，并紧急上线一组新的消费者，只负责消费旧Topic中的消息，并转储到新的Topic中，这个速度是可以很快的。然后在新的Topic上，就可以通过增加消费者个数来提高消费速度了。之后再根据情况恢复成正常情况。

> 在官网中，还分析了一个特殊的情况。就是如果RocketMQ原本是采用的普通方式搭建主从架构，而现在想要中途改为使用Dledger高可用集群，这时候如果不想历史消息丢失，就需要先将消息进行对齐，也就是要消费者把所有的消息都消费完，再来切换主从架构。因为Dledger集群会接管RocketMQ原有的CommitLog日志，所以切换主从架构时，如果有消息没有消费完，这些消息是存在旧的CommitLog中的，就无法再进行消费了。这个场景下也是需要尽快的处理掉积压的消息。

3.4、RocketMQ的消息轨迹

RocketMQ默认提供了消息轨迹的功能，这个功能在排查问题时是非常有用的。

1、RocketMQ消息轨迹数据的关键属性：

Producer端	Consumer端	Broker端
生产实例信息	消费实例信息	消息的Topic
发送消息时间	投递时间,投递轮次	消息存储位置
消息是否发送成功	消息是否消费成功	消息的Key值
发送耗时	消费耗时	消息的Tag值

2、消息轨迹配置

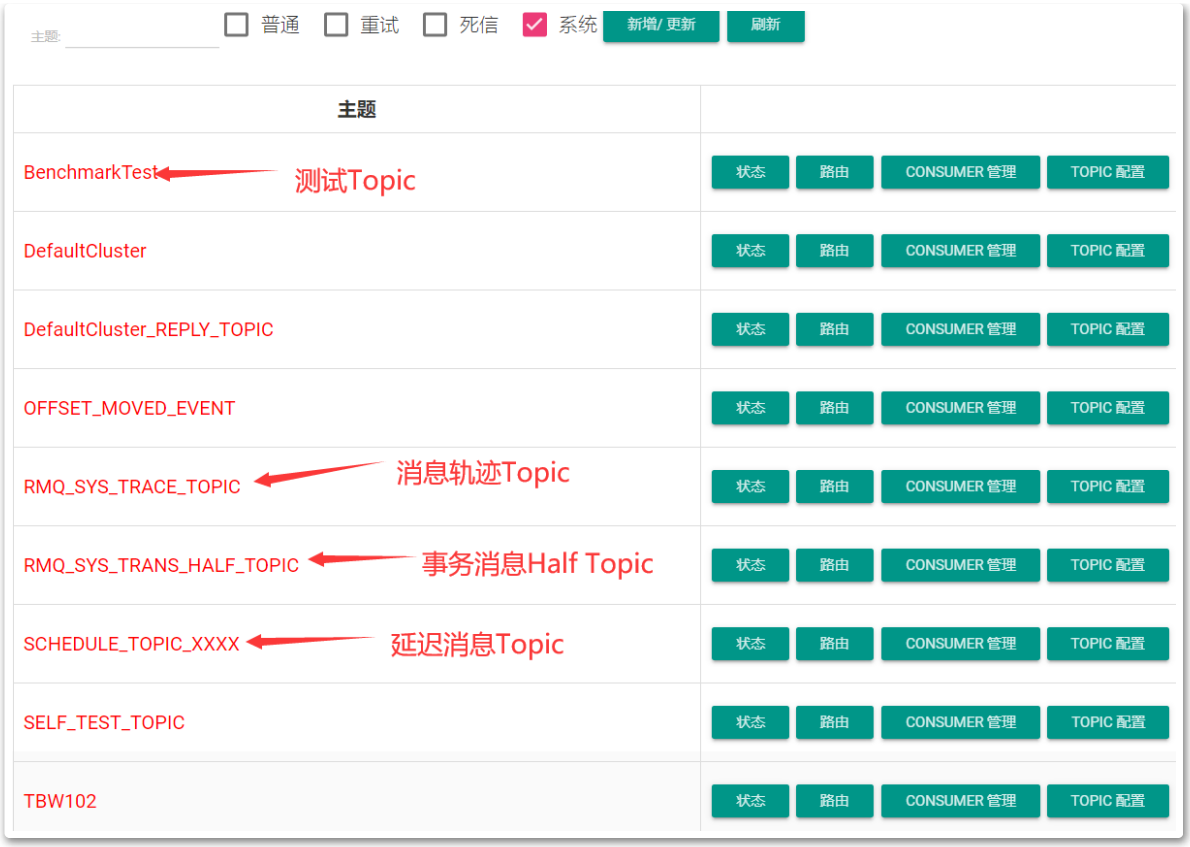
打开消息轨迹功能，需要在broker.conf中打开一个关键配置：

```
traceTopicEnable=true
```

这个配置的默认值是false。也就是说默认是关闭的。

3、消息轨迹数据存储

默认情况下，消息轨迹数据是存于一个系统级别的Topic ,RMQ_SYS_TRACE_TOPIC。这个Topic在Broker节点启动时，会自动创建出来。



另外，也支持客户端自定义轨迹数据存储的Topic。

在客户端的两个核心对象 DefaultMQProducer和DefaultMQPushConsumer，他们的构造函数中，都有两个可选的参数来打开消息轨迹存储

- **enableMsgTrace**：是否打开消息轨迹。默认是false。
- **customizedTraceTopic**：配置将消息轨迹数据存储到用户指定的Topic 。

> 有道云笔记链接地址：
>
> 文档：五期VIP02-深入掌握RocketMQ开发模型与...
> 链接：http://note.youdao.com/noteshare?id=d2c0fd4bb3ffe7b7cc4a5c16ab4de6b9&sub=DDACA76068684364B603958B8B2DEF48