1. Rabbitmq发送消息的可靠性
2. 消息的可靠投递

Rabbitmq为我们提供了两种方式来控制消息的投递可靠性，分别是confirm确认模式，return退回模式。我们将利用这两个callback来控制消息点的可靠性投递。

1. confirm模式

当消息从生产者发送到交换机时，不论发送是否成功都会返回一个confirmCallback，当发送成功时这个confirmCallback为true，当发送失败时这个confirmCallback为false。

1. return模式

消息从交换机发送到队列时，若投递失败则会返回一个returnCallback。

1. confirm模式实例
2. 创建Maven项目

创建普通的maven项目。

1. 配置pom.xml

添加Spring依赖，Rabbitmq依赖。

<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>5.1.17.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit</artifactId>  
 <version>2.1.8.RELEASE</version>  
 </dependency>  
  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-test</artifactId>  
 <version>5.1.7.RELEASE</version>  
 </dependency>  
</dependencies>  
  
<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.8.0</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

1. 配置rabbitmq连接配置

将rabbitmq的连接信息都抽取到一个properties文件中。

rabbitmq.host=192.168.211.100  
rabbitmq.port=5672  
rabbitmq.username=  
rabbitmq.password=  
rabbitmq.virtual-host=/

1. 配置Spring核心配置文件

将交换机，队列，连接工厂，rabbitTemplate首先配置到容器中，方便之后直接注入容器。

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:rabbit="http://www.springframework.org/schema/rabbit"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 https://www.springframework.org/schema/context/spring-context.xsd  
 http://www.springframework.org/schema/rabbit  
 http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">  
 <!--加载配置文件-->  
 <context:property-placeholder location="classpath:rabbitmq.properties"/>  
  
 <!-- 定义rabbitmq connectionFactory -->  
 <rabbit:connection-factory id="connectionFactory" host="${rabbitmq.host}"  
 port="${rabbitmq.port}"  
 username="${rabbitmq.username}"  
 password="${rabbitmq.password}"  
 virtual-host="${rabbitmq.virtual-host}"/>  
 <!--定义管理交换机、队列-->  
 <rabbit:admin connection-factory="connectionFactory"/>  
  
 <!--定义rabbitTemplate对象操作可以在代码中方便发送消息-->  
 <rabbit:template id="rabbitTemplate" connection-factory="connectionFactory"/>  
  
<!-- 消息可靠性投递（生产端）-->  
 <rabbit:queue id="test\_queue\_confirm" name="test\_queue\_confirm"></rabbit:queue>  
 <rabbit:direct-exchange name="test\_exchange\_confirm">  
 <rabbit:bindings>  
 <rabbit:binding queue="test\_queue\_confirm" key="confirm"></rabbit:binding>  
 </rabbit:bindings>  
 </rabbit:direct-exchange>  
  
</beans>

1. 测试Confirm模式

（1）开启confirm模式

如何测试Confirm模式呢？首先我们要开启Confrim模式，在Spring的配置文件中，将connectionFactory的配置添加publisher-confirms=”true”，即开启confirm模式。

（2）在rabbitTemplate中定义ConfrimCallBack回调函数

我们在测试类中注入rabbitTemplate之后，需要使用它的setConfirmCallback方法来定义回调函数。该回调函数有一个函数confirm需要重写，函数体就是当消息从发送方发送后执行操作，该confirm函数有三个参数：

*\** ***@param*** *correlationData 相关配置信息  
\** ***@param*** *ack 交换机是否成功收到了消息  
\** ***@param*** *cause 失败的原因*

根据ack的值，来判断交换机是否成功收到了消息，作出相应的操作。

1. Return模式实例
2. 开启回退模式

同样是在connection-factory的配置中添加publisher-return=”true”。

1. 设置回调函数

使用rabbitTemplate的setReturnCallback函数来设置回调函数，当消息成功地从交换机发送到队列时，该回调函数不会执行；当消息没有发送到队列时，该消息默认会被丢弃，则回调函数也不会执行。要想执行该回调函数，就得设置交换机处理失败消息的模式。我们可以在回调函数中设置当生产者监听到返回的消息时该做出什么样的操作。

rabbitTemplate.setReturnCallback(new RabbitTemplate.ReturnCallback() {  
 @Override  
 public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey) {  
 System.*out*.println("return 执行了");  
 System.*out*.println(message);  
 System.*out*.println(replyCode);  
 System.*out*.println(replyText);  
 System.*out*.println(exchange);  
 System.*out*.println(routingKey);  
 }  
});

该returnMessage函数的参数如下所示：

message: 消息对象

replyCode: 错误码

replyText: 错误信息

exchange: 交换机

reoutingKey: 路由键

1. 设置交换机处理失败消息的模式

设置完后，交换机就会将发送失败的消息返回给生产者，生产者通过回调函数就会得到该消息，作出相应的操作。

rabbitTemplate.setMandatory(true);

1. 事务模式

RabbitmQ中也有事务模式，使用通道的如下方式，完成事务控制：

txSelect()：将当前通道设置为事务模式

txCommit(): 发送成功提交事务

txRollback(): 发送失败回滚事务

1. RabbitMQ的消费端的可靠性
2. Consumer Ack

当消息从RabbitMQ发送到消费端后，消费端收到消息后通过Ack的方式确认。共有三种确认方式：

1. 自动确认

当消费端接受到消息后，就自动发送回执给RabbitMQ说明该消息已经收到了，不论该消息后面的处理是否成功，若后面的消息处理步骤是失败的，那么该消息等于被丢弃了。

acknowledge=”none”

1. 手动确认

当消费端接受到消息后，不会自动地发送回执，而是等到后面的消息处理阶段成功后，再手动通过调用通道的basicAck()方法发送回执给RabbitMQ；若后面的消息处理失败了，我们也可以通过调用通道的basicNack()方法让其自动重新发送消息。

acknowledge=”manaul”

1. 根据异常情况确认

acknowledge=”auto”

1. Ack自动确认实例

在消费端进行Ack实例测试。

1. 创建Maven项目
2. 配置pom.xml

同生产者。

1. 配置rabbitmq.properties

同生产者。

1. 配置Spring核心配置
2. 加载rabbitmq.properties文件

<context:property-placeholder location="classpath:rabbitmq.properties"/>

1. 定义connectionFactory

<rabbit:connection-factory id="connectionFactory" host="${rabbitmq.host}"  
 port="${rabbitmq.port}"  
 username="${rabbitmq.username}"  
 password="${rabbitmq.password}"  
 virtual-host="${rabbitmq.virtual-host}"  
 publisher-confirms="true"  
 publisher-returns="true"  
  
/>

1. 设置包扫描

用包扫描的方式，就不需要在Spring配置文件中定义监听器了，直接在监听器类上添加@Component注解即可。

<context:component-scan base-package="com.rabbitmq.listener"/>

1. 定义监听器容器

监听器容器存储着所有的监听器。

<rabbit:listener-container connection-factory="connectionFactory">  
 <rabbit:listener ref="" queues="" ></rabbit:listener>  
</rabbit:listener-container>

1. 创建自动确认监听类

创建一个自动确认监听类用于消费者监听队列，该监听类需要实现MessageListener接口，重写了onMessage方法，该方法用于监听到消息进行响应操作。注意，该方法上需要添加@Component注解，以便能够被扫描到。

@Component  
public class AckListener implements MessageListener {  
 @Override  
 public void onMessage(Message message) {  
 System.*out*.println(new String(message.getBody()));  
 }  
}

1. 测试类

在test包下创建一个测试类进行不间断地监听测试。

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(locations = "classpath:spring-rabbitmq-producer.xml")  
public class ConsumerTest {  
 @Test  
 public void test(){  
 while(true){  
  
 }  
 }  
}

1. Ack手动确认实例

其他的设置同自动确认都是相同的，不同的只有监听类与监听类容器的配置不同。

1. 监听容器

在Spring核心配置文件中配置监听类的容器时，需要添加一个acknowledge=”manual”属性，默认是none。

<rabbit:listener-container connection-factory="connectionFactory" acknowledge="manual">  
 <rabbit:listener ref="ackListener" queue-names="test\_queue\_confirm" ></rabbit:listener>  
</rabbit:listener-container>

1. 监听类

手动确认的监听类在消息处理成功后通过调用通道的basicAck()方法来返回确认给RabbitMq，因此，在监听类中需要使用通道这个参数，然而我们实现的MessageListener接口的onMesage方法是没有通道这个参数的。因此，我们需要寻找MessageListener接口的子接口ChannelAwareMessageListener，它的onMessage方法有通道参数，因此将实现MessageListener接口改为实现channelAwareMessageListener接口。

该onMessage方法体中有3段代码，一段是用于接收消息，一段是用于处理业务逻辑，一段是根据消息处理成功与否来执行返回确认操作或者重新发送消息。通道的basicAck()方法中有两个参数：deliveryTag，multiple。deliveryTag表示当前收到的消息的标签，multiple若为true表示允许多条消息同时被签收。要使用deliveryTag，首先要从消息中取出该消息的标签，通过getMessageProperties().getDeliveryTag()获取标签。通道的另一个方法basicNack()还有另外一个参数requeue，若为true表示消息处理失败后能够重新回到queue中，rabbitMQ会重新发送该消息给消费者。

@Component  
public class AckListener implements ChannelAwareMessageListener {  
 @Override  
 public void onMessage(Message message, Channel channel) throws Exception {  
 //获取消息的Tag  
 long deliveryTag = message.getMessageProperties().getDeliveryTag();  
  
 try {  
 //接受消息  
 System.*out*.println(new String(message.getBody()));  
  
 //处理业务逻辑  
 System.*out*.println("业务逻辑");  
 //int i = 3 / 0;  
 //手动签收  
 channel.basicAck(deliveryTag, true);  
 } catch (Exception e) {  
 //e.printStackTrace();  
  
 //拒绝 签收  
 channel.basicNack(deliveryTag,true, true);  
 Thread.*sleep*(1000);  
 }  
  
 }  
  
}

1. 消费端限流

（一）概述

RabbitMQ能够削峰平谷，保障系统流量的稳定，但是若在消费端没有限制，那么消费端会有崩掉的风险，因此，我们要给消费端限流，限制每次消费端能够从Rabbitmq获取的消息数量。

（二）消费端限流机制

1. 设置Ack机制为手动确认，因为只有手动确认，我们才能够通过限制是否返回确认来限制消费端的流量。在监听类容器中设置属性acknowledge=”manual”

2.设置限流数量，在监听类容器中设置属性prefetch，值为想要限制的流量。

<rabbit:listener-container connection-factory="connectionFactory" acknowledge="manual" prefetch="2">  
 <rabbit:listener ref="ackListener" queue-names="test\_queue\_confirm" ></rabbit:listener>  
</rabbit:listener-container>

1. TTL
2. 概述

TTL全称为Time To Live（存活时间），通俗意义上就是还有多长时间存活着，若到达存活时间后，该消息还没有被消费掉，那么会被自动地清除。RabbitMQ可以对消息设置TTL，也可以对整个队列设置TTL。

1. 队列TTL设置

队列的TTL需要在创建队列的时候就设置好，因此我们需要在生产者的Spring配置文件中为队列添加<rabbit:queue-argumeents>子标签，该子标签内还有<entry>子标签，为该子标签添加属性key=”x-message-ttl”，代表TTL，value=”10000”，代表TTL值为10000毫秒，value-type=”java.lang.Integer”代表TTL为整数。并且为该队列绑定相应的交换机。

<rabbit:queue name="test\_queue\_ttl" id="test\_queue\_ttl">  
 <rabbit:queue-arguments>  
 <entry key="x-message-ttl" value="10000" value-type="java.lang.Integer"></entry>  
 </rabbit:queue-arguments>  
</rabbit:queue>  
  
<rabbit:topic-exchange name="test\_exchange\_ttl">  
 <rabbit:bindings>  
 <rabbit:binding pattern="ttl.#" queue="test\_queue\_ttl"></rabbit:binding>  
 </rabbit:bindings>  
</rabbit:topic-exchange>

1. 消息的TTL设置

消息的TTL设置与队列的TTL设置不同，这是在使用RabbitTemplate发送消息时设置的，它需要一个MessagePosProcessor对象，该对象为消息后处理对象，可以设置一些消息的参数信息。通常，我们要新建一个MessagePostProcessor对象，该对象有一个重写方法postProcessMessage，在方法体中我们可以设置消息的TTL。最终，将MessagePostProcessor对象传到RabbitTemplate的convertAndSend方法中。

MessagePostProcessor messagePostProcessor = new MessagePostProcessor() {  
  
 @Override  
 public Message postProcessMessage(Message message) throws AmqpException {  
 message.getMessageProperties().setExpiration("5000");  
 return message;  
 }  
 };  
  
 rabbitTemplate.convertAndSend("test\_exchange\_ttl", "ttl.hehe", "hello\_world", messagePostProcessor);  
}

1. 注意事项
2. 如果同时设置了队列的TTL和消息的TTL，则以时间短的为准。
3. 队列过期后，会将队列上所有的消息全部移除。
4. 消息过期后，只有当消息在队列顶端时，才会被判断是否过期，然后被移除。如果该消息并不在队列顶端，即使过期也不会被移除。
5. 死信队列
6. 概述

死信队列（DLX），全称为Dead Letter Exchange，当消息成为死信（Dead Message）后，可以被重新发送到另一个交换机中，这个交换机就是DLX，死信交换机可以将死信发送到死信队列中，被其他的消费者进行消费。

1. 称为死信的情况
2. 队列消息长度到达限制，多出来的消息成为死信。
3. 消费者拒绝消费消息，即消息处理失败，通道调用basicNack/basicReject函数，并且不把消息重新放回到原目标队列中，requeue=false，该消息成为死信。
4. 原队列存在消息过期设置，消息到达存活时间未被消费掉，成为死信。
5. 队列绑定死信交换机

当正常的队列中存在死信时，若要将死信发送到死信交换机中，需要为该队列绑定死信交换机，为该队列设置参数 x-dead-letter-exchange，值为死信交换机的名称，设置参数x-dead-letter-routing-key，该参数为正常队列发送到死信交换机的routing-key。

1. 代码实现
2. 声明正常队列和正常交换机

<rabbit:queue name="test\_queue\_dlx" id="test\_queue\_dlx"></rabbit:queue>  
<rabbit:topic-exchange name="test\_exchange\_dlx">  
 <rabbit:bindings>  
 <rabbit:binding pattern="test.dlx.#" queue="test\_queue\_dlx"></rabbit:binding>  
 </rabbit:bindings>  
</rabbit:topic-exchange>

1. 声明死信队列和死信交换机

死信队列和死信交换机和正常的没有区别。

<rabbit:queue name="queue\_dlx" id="queue\_dlx"></rabbit:queue>  
<rabbit:topic-exchange name="exchange\_dlx">  
 <rabbit:bindings>  
 <rabbit:binding pattern="dlx.#" queue="queue\_dlx"></rabbit:binding>  
 </rabbit:bindings>  
</rabbit:topic-exchange>

1. 正常队列绑定死信交换机

为正常队列设置参数来绑定死信交换机。这些参数的名称可以在rabbitmq的管理控制台的queue页面中通过arguments参数获取到。

<rabbit:queue name="test\_queue\_dlx" id="test\_queue\_dlx">  
 <rabbit:queue-arguments>  
 <entry key="x-dead-letter-exchange" value="exchange\_dlx"/>  
 <entry key="x-dead-letter-routing-key" value="dlx.h"/>  
 </rabbit:queue-arguments>  
</rabbit:queue>

1. 设置死信队列的产生条件

产生死信队列是有产生条件的，要么是队列过期，要么是队列长度限制。我们可以在正常queue的定义中设置这些条件，来产生死信队列。

<entry key="x-message-ttl" value="10000" value-type="java.lang.Integer"/>  
<entry key="x-max-length" value="10" value-type="java.lang.Integer"/>

1. 测试
2. 测试队列过期消息称为死信

发送一条正常的消息给正常的交换机和队列，过了TTL后它就会称为死信并且发送给死信交换机和队列。

@RunWith(SpringJUnit4ClassRunner.class)  
@ContextConfiguration(locations = "classpath:spring-rabbitmq-producer.xml")  
public class DeadTest {  
 @Autowired  
 private RabbitTemplate rabbitTemplate;  
  
 @Test  
 public void test(){  
 //测试TTL  
 rabbitTemplate.convertAndSend("test\_exchange\_dlx", "test.dlx.ddd", "Dead Msg");  
 }  
}

1. 测试队列长度限制，多余的消息称为死信

只要发送大于队列长度的消息，多出来的消息就会成为死信发送到死信队列中。

for(int i = 0; i < 20; ++i){  
 rabbitTemplate.convertAndSend("test\_exchange\_dlx", "test.dlx.e", "ddd");  
}

1. 测试消费者拒绝接受消息，消息称为死信

当消费者处理消息失败时，会拒绝接受这条消息，且没有设置重新发回到原有的队列中，该消息就会成为死信，发送到死信队列中。

在消费者处理消息的异常代码段，将通道的basicNack函数的requeue参数设置为false；同时在Spring配置文件中配置监听器时持续监听正常队列即可，拒收的消息会被该正常队列发送到死信交换机和队列中。

1. 延迟队列
2. 概述

延迟队列，即消息进入队列后不会被立即消费，而是在到达指定的时间后，才会被消费掉。这种队列在许多场景中可以应用，如下单未支付的倒计时，只要是与计时有关的场景，都可以使用延迟队列实现。

1. Rabbitmq中的延迟队列

Rabbitmq中没有延迟队列，但是我们可以通过TTL + 死信队列的方式来实现延迟队列。即首先设置队列的TTL为指定的时间，过了该时间后，死信被发送给死信交换机，继而发送给死信队列，最终到达消费者，这就实现了延迟队列的功能。

1. 日志与监控
2. Rabbitmq日志

日志记录了客户端和Rabbitmq服务器的交互情况，包含RabbitMQ的版本号，Erlang的版本号，RabbitMQ的服务节点名称，cookie的hash值，RabbitMQ配置文件地址，内存限制，硬盘限制。日志默认存放在 [/var/log/rabbitmq/rabbit@xxx.log](mailto:/var/log/rabbitmq/rabbit@xxx.log)。

1. rabbitmqctl管理监控

我们可以通过rabbitmq的管理控制台直接查看，也可以通过命令的方式来监控。

1. 查看队列

rabbitmqctl list\_queues

1. 查看交换机

rabbitmqctl list\_exchanges

1. 查看用户

rabbitmqctl list\_users

1. 查看连接

rabbitmqctl list\_connections

1. 查看消费者信息

rabbitmqctl list\_consumers

1. 查看环境变量

rabbitmqctl environment

1. 查看未被确认的队列

rabbitmqctl list\_queues name messages\_unacknowledged

1. 查看单个队列的内存使用情况

rabbitmqctl list\_queues name memory

1. 查看准备就绪的队列

rabbitmqctl list\_queus name messages\_ready

1. 消息追踪
2. 概述

消息传递的过程中总会有各种各样的异常发生，我们需要通过追踪这些消息的发送过程，投递过程，来定位消息传递的异常位置，提高开发的效率。

1. Firehose和rabbitmq\_tracing插件

在Rabbiitmq中可以使用Firehose和rabbitmq\_tracing插件来实现消息的追踪功能。

1. Firehose

（1）概述

Firehose的机制是将生产者发送给Rabbitmq的消息以及Rabbitmq投递给消费者的消息按照指定的格式发送到默认的交换机中。这个默认的交换机的名称为amq.rabbitmq.trace，它是一个topic类型的交换机。同时，这个消息会在原有的消息基础上添加消息头，包括一系列的消息相关的信息，如交换机名称，路由键，节点名称，虚拟机名称，连接名称，通道数目，用户名称，路由的队列名称。这些信息能够有效地追踪消息的发送过程。

发送到amq.rabbitmq.trace默认交换机的消息的routing key 为两种，一种是生产者投递到交换机的消息，routing key 为publish.exchangename，后面为实际的交换机名称；一种是消费者从queue中获取的消息，routing key为deliver.queuname，后面为实际的队列名称。

（2）开启插件

通过rabbitmqctl trace\_on 开启Firehose命令

通过rabbitmqctl trace\_off 关闭Firehose命令

注意，打开该插件会影响消息的写入功能，请合理地使用。

1. rabbitmq\_tracing

（1）概述

rabibtmq\_tracing比Firehose多了一层GUI的封装，更容易使用了。

（2）启用插件

启用插件： rabbitmq-plugins enable rabbitmq\_tracing

（3）使用插件

当我们启用该插件后，在rabbitmq的管理控制台的admin子页面就会出现trace的入口，我们自行添加一个trace，选择要监听的虚拟机，以及routing key规则。设置好规则后，那些被Firehose默认发送到amq.rabbit,mq.trace交换机中的消息只要符合trace的routing key规则都会被记录下消息的消息头到日志文件trace.log中。这样就记录下了消息的发送与投递过程，便于我们的开发纠错。

1. 消息可靠性保障（消息补偿）

如何保障消息能够全部发送出去且全部被消费成功呢？

首先，消息从生产者端被保存到业务数据库中，同时发送到Rabbitmq的服务器中，消费者从队列中取出消息，消费成功后，发送确认消息给队列，这时，一个回调检查服务监听到了消息的确认，将该消息存储到回调检查的数据库中。

生产端在发送完该消息后，会延迟一段时间再次发送该消息，回调检查服务通过监听该延迟消息，对比该延迟消息是否存在于回调检查的数据库中，如果存在，说明该消息已经消费成功，对于延迟消息不做处理；如果不存在，则调用生产者重新发送该消息。

如果，延迟发送的消息也失败了，那么还有最后一个保障：定时检查服务，我们定时地对比生产者端发送消息的业务数据库以及回调检查服务的数据库，如果数据由确实，那么就会调用生产者，重新发送消息。

1. 消息幂等性保障
2. 概述

幂等性指的是一次或者多次请求某一个资源，对于该资源来说应该得到同样的结果。在MQ中指的是消费多条相同的消息，得到与消费该消息一次相同的结果。

1. 乐观锁机制

假设有一条消息发送到RabbitMQ中，该消息是为了将数据库中的金额减去500元，然而，消费者消费出错，导致该消息被重新发送了多次，那么在没有消息幂等性保障的情况下被消费了多次，导致业务逻辑出错。

乐观锁机制是解决消息幂等性问题的，它给消息加上了version版本号，根据version来设定消费的规则，如设定version=1时代表着这是第一次消费该消息。当同样的消息被消费过一次后，将version加1，那么下次同样的消息过来后，就无法匹配上消费的规则，自然就无法被消费。