
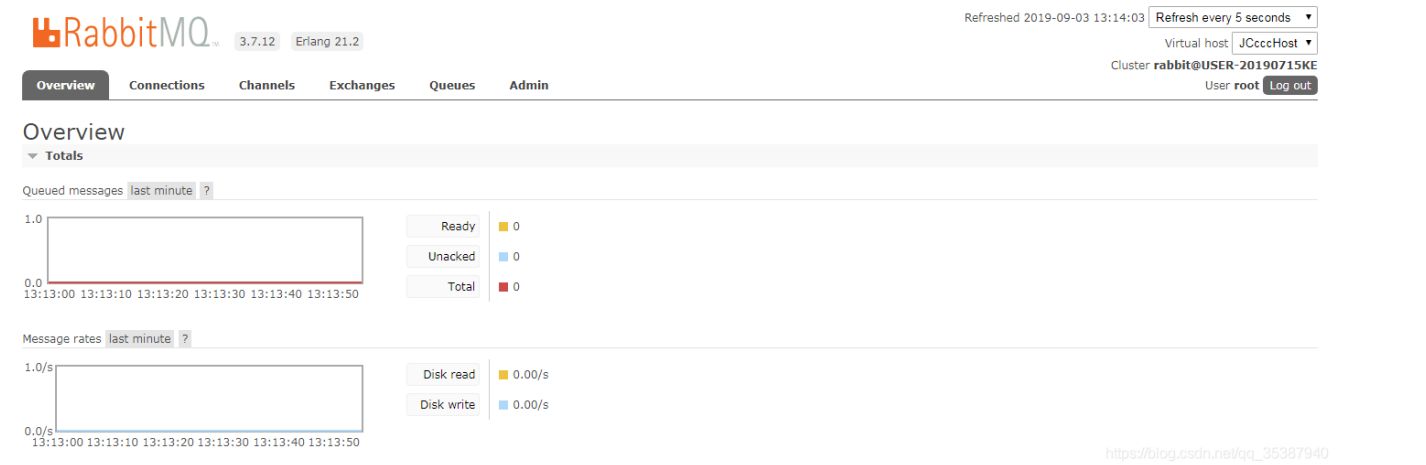


Springboot 整合RabbitMq ， 用心看完这一篇就够了

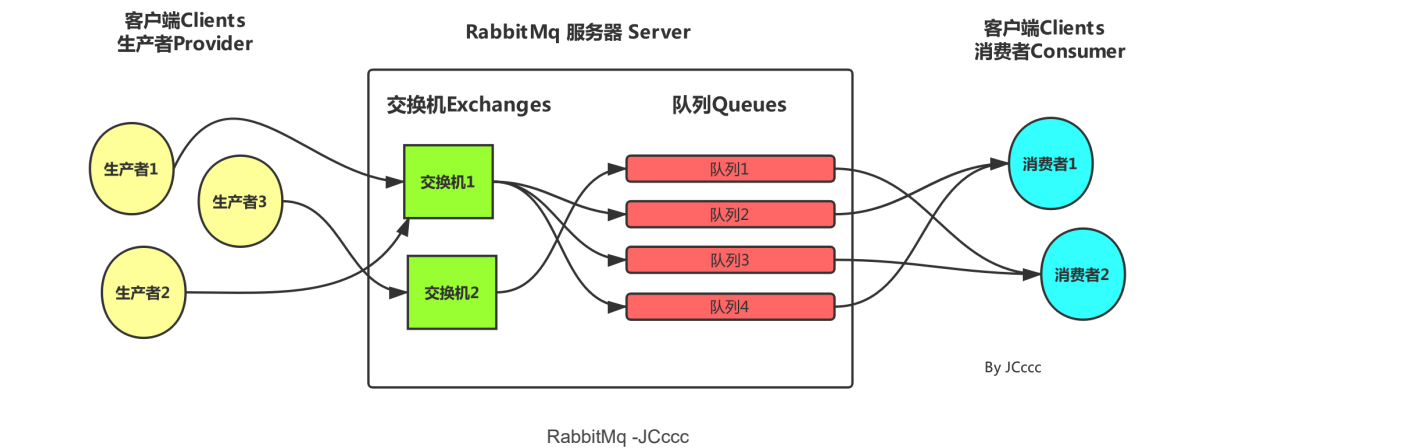
 跟我一起玩转 SpringBoot 同时被 2 个专栏收录

该篇文章内容较多，包括有rabbitMq相关的一些简单理论介绍，provider消息推送实例，consumer消息消费实例，Direct、Topic、Fanout的使用，消息回调、手动确认等。（但是关于rabbitMq的安装，就不介绍了）

在安装完rabbitMq后，输入<http://ip:15672/>，是可以看到一个简单后台管理界面的。



在这个界面里面我们可以做些什么？
可以手动创建虚拟host，创建用户，分配权限，创建交换机，创建队列等等，还有查看队列消息，消费效率，推送效率等等。
以上这些管理界面的操作在这篇暂时不做扩展描述，我想着重介绍后面实例里会使用到的。
首先先介绍一个简单的一个消息推送到接收的流程，提供一个简单的图：



黄色的圈圈就是我们的消息推送服务，将消息推送到 中间方框里面也就是 rabbitMq的服务器，然后经过服务器里面的交换机、队列等各种关系（后面会详细讲）将数据处理入列后，最终右边的蓝色圈圈消费者获取对应监听的消息。

常用的交换机有以下三种，因为消费者是从队列获取信息的，队列是绑定交换机的（一般），所以对应的消息推送/接收模式也会有以下几种：

Direct Exchange

直连型交换机，根据消息携带的路由键将消息投递给对应队列。

大致流程，有一个队列绑定到一个直连交换机上，同时赋予一个路由键 routing key 。
然后当一个消息携带着路由值为X，这个消息通过生产者发送给交换机时，交换机就会根据这个路由值X去寻找绑定值也是X的队列。

Fanout Exchange

扇型交换机，这个交换机没有路由键概念，就算你绑了路由键也是无视的。这个交换机在接收到消息后，会直接转发到绑定到它上面的所有队列。

Topic Exchange

主题交换机，这个交换机其实跟直连交换机流程差不多，但是它的特点就是在它的路由键和绑定键之间是有规则的。简单地介绍下规则：

*** (星号) 用来表示一个单词 (必须出现的)**

(井号) 用来表示任意数量 (零个或多个) 单词

通配的绑定键是跟队列进行绑定的，举个小例子

队列Q1 绑定键为 *.TT.* 队列Q2绑定键为 TT.#

如果一条消息携带的路由键为 A.TT.B，那么队列Q1将会收到；

如果一条消息携带的路由键为TT.AA.BB，那么队列Q2将会收到；

主题交换机是非常强大的，为啥这么膨胀？

当一个队列的绑定键为 "#" (井号) 的时候，这个队列将会无视消息的路由键，接收所有的消息。

当 * (星号) 和 # (井号) 这两个特殊字符都未在绑定键中出现的时候，此时主题交换机就拥有的直连交换机的行为。

所以主题交换机也就实现了扇形交换机的功能，和直连交换机的功能。

另外还有 Header Exchange 头交换机， Default Exchange 默认交换机， Dead Letter Exchange 死信交换机，这几个该篇暂不做讲述。

好了，一些简单的介绍到这里为止， 接下来我们来一起编码。

本次实例教程需要创建2个springboot项目，一个 rabbitmq-provider (生产者)，一个rabbitmq-consumer (消费者)。

首先创建 rabbitmq-provider，

pom.xml里用到的jar依赖：

```
<!--rabbitmq-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

然后application.yml：

ps：里面的虚拟host配置项不是必须的，我自己在rabbitmq服务上创建了自己的虚拟host，所以我配置了；你们不创建，就不用加这个配置项。

```
server:
  port: 8021
spring:
  #给项目来个名字
  application:
    name: rabbitmq-provider
  #配置rabbitMq 服务器
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: root
    password: root
    #虚拟host 可以不设置,使用server默认host
    virtual-host: JCcccHost
```

接着我们先使用下direct exchange(直连型交换机),创建DirectRabbitConfig.java（对于队列和交换机持久化以及连接使用设置，在注释里有说明，后面的不同交换机的配置就不做同样说明了）：

```
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Configuration
public class DirectRabbitConfig {

    //队列 起名: TestDirectQueue
    @Bean
    public Queue TestDirectQueue() {
        // durable:是否持久化,默认是false,持久化队列:会被存储在磁盘上,当消息代理重启时仍然存在,暂存队列:当前连接有效
        // exclusive:默认也是false,只能被当前创建的连接使用,而且当连接关闭后队列即被删除。此参考优先级高于durable
        // autoDelete:是否自动删除,当没有生产者或者消费者使用此队列,该队列会自动删除。
        // return new Queue("TestDirectQueue",true,true,false);

        //一般设置一下队列的持久化就好,其余两个就是默认false
        return new Queue("TestDirectQueue",true);
    }

    //Direct交换机 起名: TestDirectExchange
    @Bean
    DirectExchange TestDirectExchange() {
        // return new DirectExchange("TestDirectExchange",true,true);
        return new DirectExchange("TestDirectExchange",true,false);
    }

    //绑定 将队列和交换机绑定,并设置用于匹配键: TestDirectRouting
    @Bean
    Binding bindingDirect() {
        return BindingBuilder.bind(TestDirectQueue()).to(TestDirectExchange()).with("TestDirectRouting");
    }

    @Bean
    DirectExchange lonelyDirectExchange() {
        return new DirectExchange("lonelyDirectExchange");
    }

}
```

然后写个简单的接口进行消息推送（根据需求也可以改为定时任务等等，具体看需求），SendMessageController.java：

```
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.HashMap;
import java.util.Map;
import java.util.UUID;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@RestController
public class SendMessageController {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @GetMapping("/sendMessage")
    public String sendMessage() {
        Map<String, Object> map = new HashMap<>();
        map.put("key", "value");
        rabbitTemplate.convertAndSend("TestDirectQueue", map);
        return "sendMessage success";
    }
}
```

```

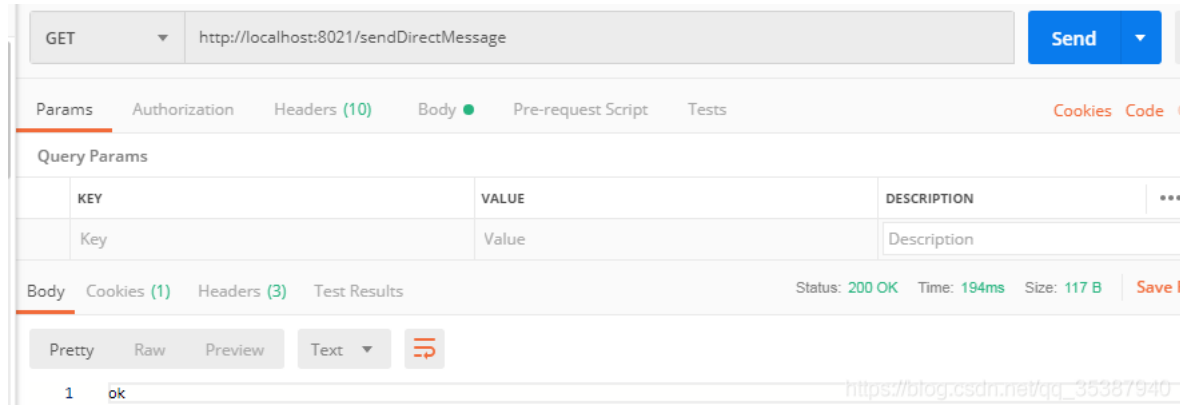
* @CreateTime : 2019/9/3 | * @Description :
**/
@RestController
public class SendMessageController {

    @Autowired
    RabbitTemplate rabbitTemplate; //使用RabbitTemplate,这提供了接收/发送等等方法

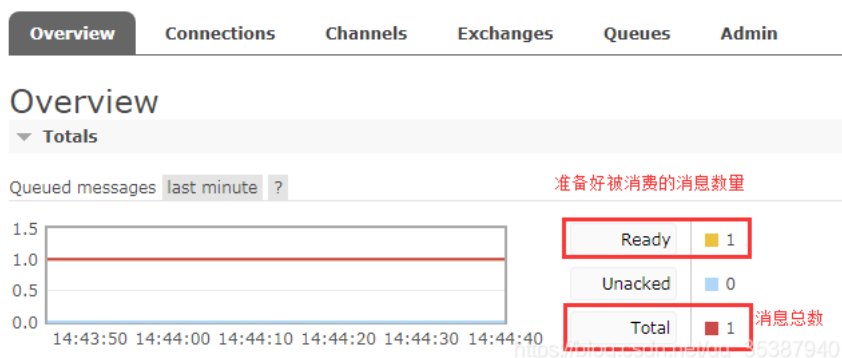
    @GetMapping("/sendDirectMessage")
    public String sendDirectMessage() {
        String messageId = String.valueOf(UUID.randomUUID());
        String messageData = "test message, hello!";
        String createTime = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
        Map<String, Object> map = new HashMap<>();
        map.put("messageId", messageId);
        map.put("messageData", messageData);
        map.put("createTime", createTime);
        //将消息携带绑定键值: TestDirectRouting 发送到交换机TestDirectExchange
        rabbitTemplate.convertAndSend("TestDirectExchange", "TestDirectRouting", map);
        return "ok";
    }
}

```

把rabbitmq-provider项目运行，调用下接口：



因为我们目前还没弄消费者 rabbitmq-consumer，消息没有被消费的，我们去rabbitMq管理页面看看，是否推送成功：



再看看队列（界面上的各个英文项代表什么意思，可以自己查查哈，对理解还是有帮助的）：

Overview

Connections

Channels

Exchanges

Queues

Admin

Queues

▼ All queues (1)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates			+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
JCcccHost	TestDirectQueue	D	idle	1	0	1	0.00/s			

▶ Add a new queue

https://blog.csdn.net/qq_35387940

很好，消息已经推送到rabbitMq服务器上面了。

接下来，创建rabbitmq-consumer项目：

pom.xml里的jar依赖：

```
<!--rabbitmq-->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
```

然后是 application.yml：

```
server:
  port: 8022
spring:
  #给项目来个名字
  application:
    name: rabbitmq-consumer
  #配置rabbitMq 服务器
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: root
    password: root
    #虚拟host 可以不设置,使用server默认host
    virtual-host: JCcccHost
```

然后一样，创建DirectRabbitConfig.java（消费者单纯的使用，其实可以不用添加这个配置，直接建后面的监听就好，使用注解来让监听器监听对应的队列即可。配置上了的话，其实消费者也是生成者的身份，也能推送该消息。）：

```
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
```

```

**/ | @Configuration
public class DirectRabbitConfig {

    //队列 起名: TestDirectQueue
    @Bean
    public Queue TestDirectQueue() {
        return new Queue("TestDirectQueue", true);
    }

    //Direct交换机 起名: TestDirectExchange
    @Bean
    DirectExchange TestDirectExchange() {
        return new DirectExchange("TestDirectExchange");
    }

    //绑定 将队列和交换机绑定, 并设置用于匹配键: TestDirectRouting
    @Bean
    Binding bindingDirect() {
        return BindingBuilder.bind(TestDirectQueue()).to(TestDirectExchange()).with("TestDirectRouting");
    }
}

```

然后是创建消息接收监听类, DirectReceiver.java:

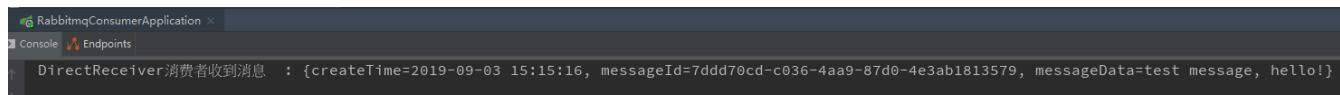
```

@Component
@RabbitListener(queues = "TestDirectQueue")//监听的队列名称 TestDirectQueue
public class DirectReceiver {

    @RabbitHandler
    public void process(Map testMessage) {
        System.out.println("DirectReceiver消费者收到消息 : " + testMessage.toString());
    }
}

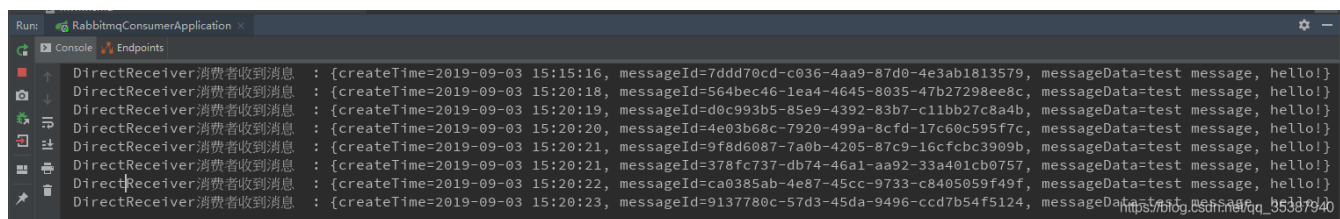
```

然后将rabbitmq-consumer项目运行起来, 可以看到把之前推送的那条消息消费下来了:



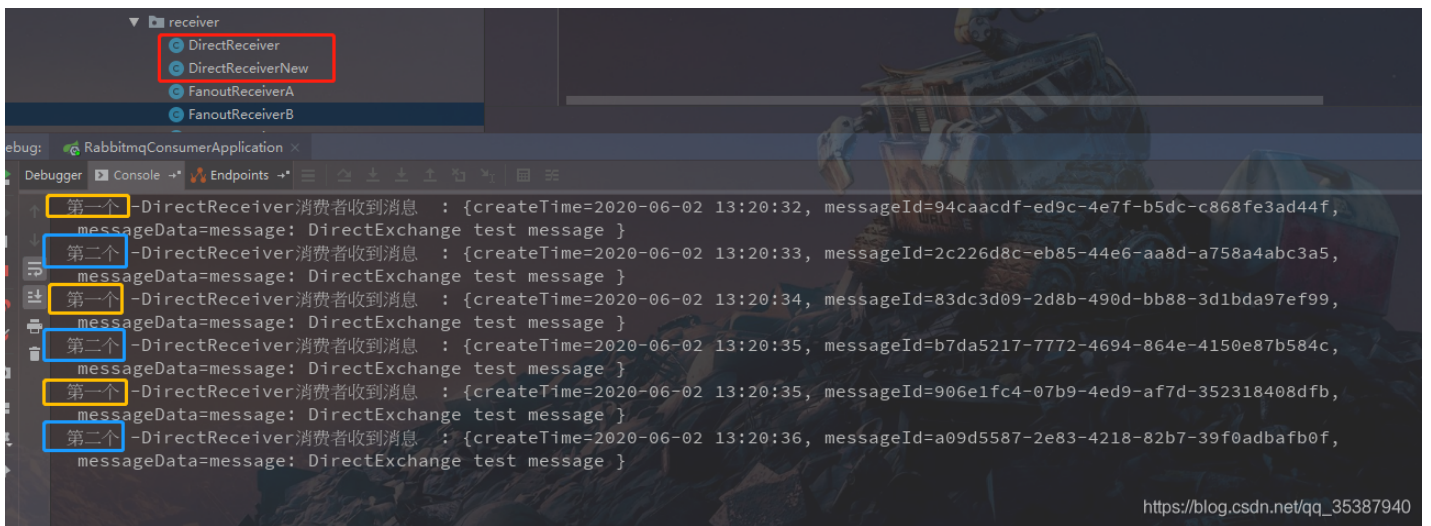
The screenshot shows the console output of the RabbitmqConsumerApplication. It displays a single log entry: "DirectReceiver消费者收到消息 : {createTime=2019-09-03 15:15:16, messageId=7ddd70cd-c036-4aa9-87d0-4e3ab1813579, messageData=test message, hello!}"

然后可以继续调用rabbitmq-provider项目的推送消息接口, 可以看到消费者即时消费消息:



The screenshot shows the console output of the RabbitmqConsumerApplication after multiple messages are pushed. It displays a series of log entries, each showing a message received by DirectReceiver with a unique messageId and the message data "test message, hello!". The messages are received in chronological order, demonstrating that the consumer is processing messages as they arrive.

那么直连交换机既然是一对一, 那如果咱们配置多台监听绑定到同一个直连交互的同一个队列, 会怎么样?



可以看到是实现了轮询的方式对消息进行消费，而且不存在重复消费。

接着，我们使用Topic Exchange 主题交换机。

在rabbitmq-provider项目里面创建TopicRabbitConfig.java:

```
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */

@Configuration
public class TopicRabbitConfig {
    //绑定键
    public final static String man = "topic.man";
    public final static String woman = "topic.woman";

    @Bean
    public Queue firstQueue() {
        return new Queue(TopicRabbitConfig.man);
    }

    @Bean
    public Queue secondQueue() {
        return new Queue(TopicRabbitConfig.woman);
    }

    @Bean
    TopicExchange exchange() {
        return new TopicExchange("topicExchange");
    }

    //将firstQueue和topicExchange绑定,而且绑定的键值为topic.man
    //这样只要是消息携带的路由键是topic.man,才会分发到该队列
    @Bean
    Binding bindingExchangeMessage() {
        return BindingBuilder.bind(firstQueue()).to(exchange()).with(man);
    }
}
```

```

//将secondQueue和topicExchange绑定,而且绑定的键值为用上题配路由键规则topic.#
// 这样只要是消息携带的路由键是以topic.开头,都会分发到该队列
@Bean
Binding bindingExchangeMessage2() {
    return BindingBuilder.bind(secondQueue()).to(exchange()).with("topic.#");
}
}

```

然后添加多2个接口, 用于推送消息到主题交换机:

```

@GetMapping("/sendTopicMessage1")
public String sendTopicMessage1() {
    String messageId = String.valueOf(UUID.randomUUID());
    String messageData = "message: M A N ";
    String createTime = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    Map<String, Object> manMap = new HashMap<>();
    manMap.put("messageId", messageId);
    manMap.put("messageData", messageData);
    manMap.put("createTime", createTime);
    rabbitTemplate.convertAndSend("topicExchange", "topic.man", manMap);
    return "ok";
}

@GetMapping("/sendTopicMessage2")
public String sendTopicMessage2() {
    String messageId = String.valueOf(UUID.randomUUID());
    String messageData = "message: woman is all ";
    String createTime = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    Map<String, Object> womanMap = new HashMap<>();
    womanMap.put("messageId", messageId);
    womanMap.put("messageData", messageData);
    womanMap.put("createTime", createTime);
    rabbitTemplate.convertAndSend("topicExchange", "topic.woman", womanMap);
    return "ok";
}
}

```

生产者这边已经完事, 先不急着重运行, 在rabbitmq-consumer项目上, 创建TopicManReceiver.java:

```

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;
import java.util.Map;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Component
@RabbitListener(queues = "topic.man")
public class TopicManReceiver {

    @RabbitHandler
    public void process(Map testMessage) {
        System.out.println("TopicManReceiver消费者收到消息 : " + testMessage.toString());
    }
}

```

再创建一个TopicTotalReceiver.java:

```

package com.elegant.rabbitmqconsumer.receiver;

import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

```



```

import java.util.Map;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */

@Component
@RabbitListener(queues = "topic.woman")
public class TopicTotalReceiver {

    @RabbitHandler
    public void process(Map testMessage) {
        System.out.println("TopicTotalReceiver消费者收到消息 : " + testMessage.toString());
    }
}

```

同样，加主题交换机的相关配置，TopicRabbitConfig.java（消费者一定要加这个配置吗？不需要的其实，理由在前面已经说过了。）：

```

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */

@Configuration
public class TopicRabbitConfig {
    //绑定键
    public final static String man = "topic.man";
    public final static String woman = "topic.woman";

    @Bean
    public Queue firstQueue() {
        return new Queue(TopicRabbitConfig.man);
    }

    @Bean
    public Queue secondQueue() {
        return new Queue(TopicRabbitConfig.woman);
    }

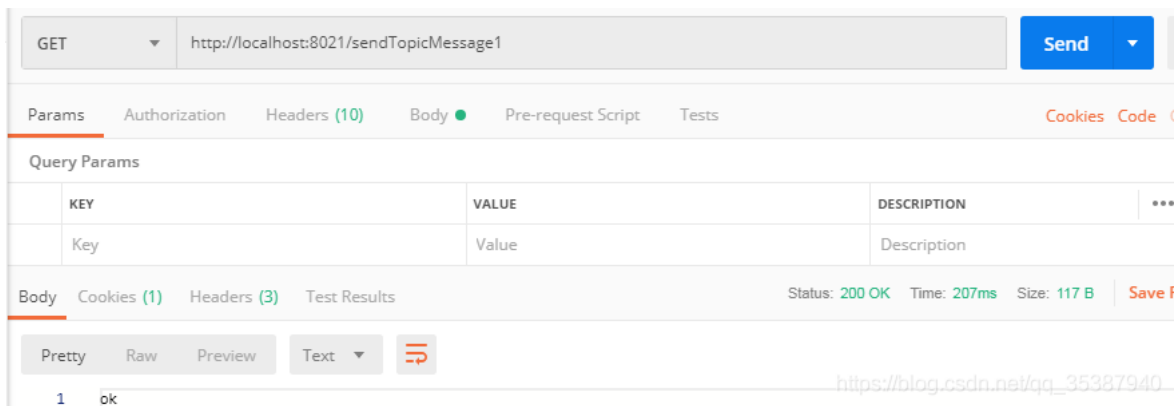
    @Bean
    TopicExchange exchange() {
        return new TopicExchange("topicExchange");
    }

    //将firstQueue和topicExchange绑定,而且绑定的键值为topic.man
    //这样只要是消息携带的路由键是topic.man,才会分发到该队列
    @Bean
    Binding bindingExchangeMessage() {
        return BindingBuilder.bind(firstQueue()).to(exchange()).with(man);
    }

    //将secondQueue和topicExchange绑定,而且绑定的键值为用上通配路由键规则topic.#
    // 这样只要是消息携带的路由键是以topic.开头,都会分发到该队列
    @Bean
    Binding bindingExchangeMessage2() {
        return BindingBuilder.bind(secondQueue()).to(exchange()).with("topic.#");
    }
}

```

然后把rabbitmq-provider, rabbitmq-consumer两个项目都跑起来, 先调用/sendTopicMessage1 接口:



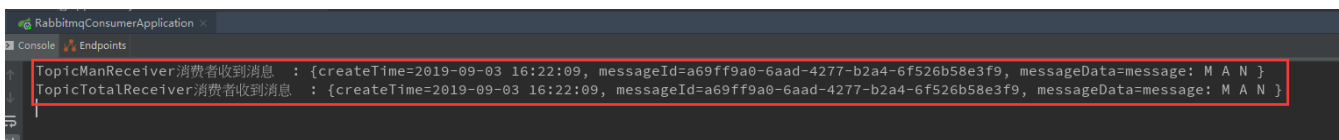
然后看消费者rabbitmq-consumer的控制台输出情况:

TopicManReceiver监听队列1, 绑定键为: topic.man

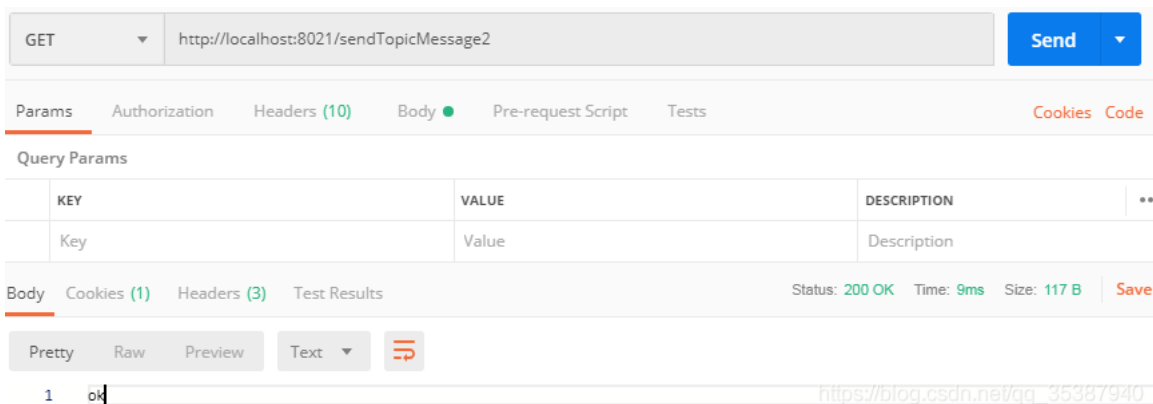
TopicTotalReceiver监听队列2, 绑定键为: topic.#

而当前推送的消息, 携带的路由键为: topic.man

所以可以看到两个监听消费者receiver都成功消费到了消息, 因为这两个receiver监听的队列的绑定键都能与这条消息携带的路由键匹配上。



接下来调用接口/sendTopicMessage2:



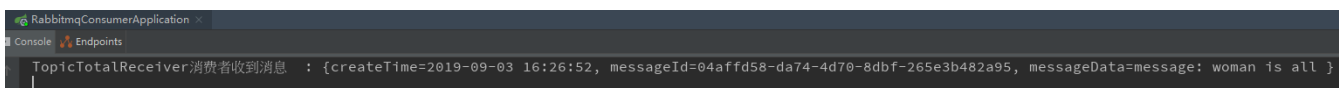
然后看消费者rabbitmq-consumer的控制台输出情况:

TopicManReceiver监听队列1, 绑定键为: topic.man

TopicTotalReceiver监听队列2, 绑定键为: topic.#

而当前推送的消息, 携带的路由键为: topic.woman

所以可以看到两个监听消费者只有TopicTotalReceiver成功消费到了消息。



接下来是使用Fanout Exchange 扇型交换机。

同样地, 先在rabbitmq-provider项目上创建FanoutRabbitConfig.java:

```

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.FanoutExchange;
  
```

```

import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */

@Configuration
public class FanoutRabbitConfig {

    /**
     * 创建三个队列 : fanout.A fanout.B fanout.C
     * 将三个队列都绑定在交换机 fanoutExchange 上
     * 因为是扇型交换机,路由键无需配置,配置也不起作用
     */

    @Bean
    public Queue queueA() {
        return new Queue("fanout.A");
    }

    @Bean
    public Queue queueB() {
        return new Queue("fanout.B");
    }

    @Bean
    public Queue queueC() {
        return new Queue("fanout.C");
    }

    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

    @Bean
    Binding bindingExchangeA() {
        return BindingBuilder.bind(queueA()).to(fanoutExchange());
    }

    @Bean
    Binding bindingExchangeB() {
        return BindingBuilder.bind(queueB()).to(fanoutExchange());
    }

    @Bean
    Binding bindingExchangeC() {
        return BindingBuilder.bind(queueC()).to(fanoutExchange());
    }
}

```

然后是写一个接口用于推送消息,

```

@GetMapping("/sendFanoutMessage")
public String sendFanoutMessage() {
    String messageId = String.valueOf(UUID.randomUUID());
    String messageData = "message: testFanoutMessage ";
    String createTime = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    Map<String, Object> map = new HashMap<>();
    map.put("messageId", messageId);
    map.put("messageData", messageData);
    map.put("createTime", createTime);
    rabbitTemplate.convertAndSend("fanoutExchange", null, map);
    return "ok";
}

```

```
}
```

接着在rabbitmq-consumer项目里加上消息消费类，

FanoutReceiverA.java:

```
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;
import java.util.Map;
/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Component
@RabbitListener(queues = "fanout.A")
public class FanoutReceiverA {

    @RabbitHandler
    public void process(Map testMessage) {
        System.out.println("FanoutReceiverA消费者收到消息 : " +testMessage.toString());
    }

}
```

FanoutReceiverB.java:

```
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;
import java.util.Map;
/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Component
@RabbitListener(queues = "fanout.B")
public class FanoutReceiverB {

    @RabbitHandler
    public void process(Map testMessage) {
        System.out.println("FanoutReceiverB消费者收到消息 : " +testMessage.toString());
    }

}
```

FanoutReceiverC.java:

```
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;
import java.util.Map;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Component
@RabbitListener(queues = "fanout.C")
public class FanoutReceiverC {

    @RabbitHandler
    public void process(Map testMessage) {
        System.out.println("FanoutReceiverC消费者收到消息 : " +testMessage.toString());
    }

}
```

```

    }
}

```

然后加上扇型交换机的配置类，FanoutRabbitConfig.java（消费者真的要加这个配置吗？不需要的其实，理由在前面已经说过了）：

```

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.FanoutExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Configuration
public class FanoutRabbitConfig {

    /**
     * 创建三个队列：fanout.A fanout.B fanout.C
     * 将三个队列都绑定在交换机 fanoutExchange 上
     * 因为是扇型交换机，路由键无需配置，配置也不起作用
     */

    @Bean
    public Queue queueA() {
        return new Queue("fanout.A");
    }

    @Bean
    public Queue queueB() {
        return new Queue("fanout.B");
    }

    @Bean
    public Queue queueC() {
        return new Queue("fanout.C");
    }

    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

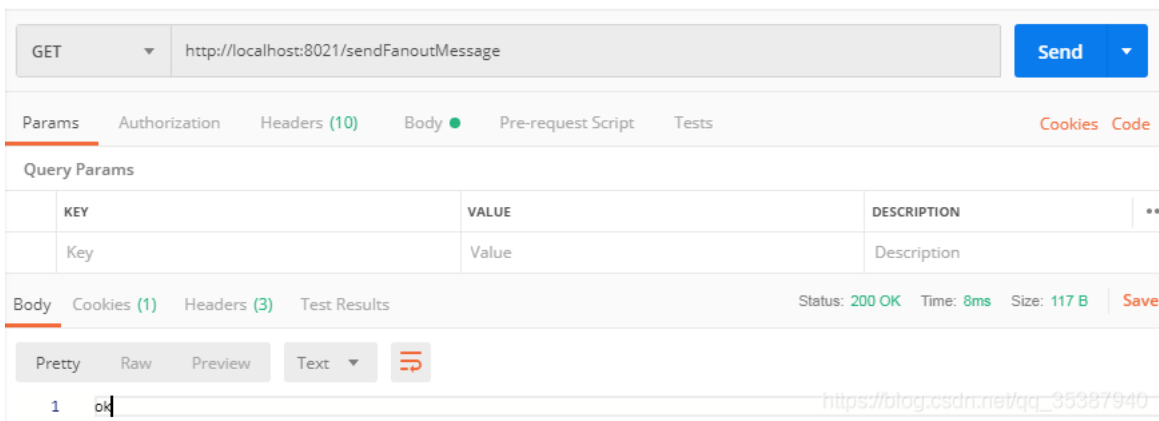
    @Bean
    Binding bindingExchangeA() {
        return BindingBuilder.bind(queueA()).to(fanoutExchange());
    }

    @Bean
    Binding bindingExchangeB() {
        return BindingBuilder.bind(queueB()).to(fanoutExchange());
    }

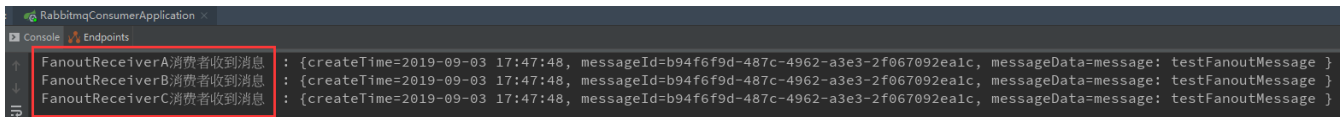
    @Bean
    Binding bindingExchangeC() {
        return BindingBuilder.bind(queueC()).to(fanoutExchange());
    }
}

```

最后将rabbitmq-provider和rabbitmq-consumer项目都跑起来，调用下接口/sendFanoutMessage：



然后看看rabbitmq-consumer项目的控制台情况：



可以看到只要发送到 fanoutExchange 这个扇型交换机的消息，三个队列都绑定这个交换机，所以三个消息接收类都监听到了这条消息。

到了这里其实三个常用的交换机的使用我们已经完毕了，那么接下来我们继续讲讲消息的回调，其实就是消息确认（生产者推送消息成功，消费者接收消息成功）。

在rabbitmq-provider项目的application.yml文件上，加上消息确认的配置项后：

ps： 本篇文章使用springboot版本为 2.1.7.RELEASE；
如果你们在配置确认回调，测试发现无法触发回调函数，那么存在原因也许是因为版本导致的配置项不起效，
可以把publisher-confirms: true 替换为 publisher-confirm-type: correlated

```
server:
  port: 8021
spring:
  #给项目来个名字
  application:
    name: rabbitmq-provider
  #配置rabbitMq 服务器
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: root
    password: root
    #虚拟host 可以不设置,使用server默认host
    virtual-host: JCcccHost
    #消息确认配置项

    #确认消息已发送到交换机(Exchange)
    publisher-confirms: true
    #确认消息已发送到队列(Queue)
    publisher-returns: true
```

然后是配置相关的消息确认回调函数，RabbitConfig.java：

```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.connection.CorrelationData;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/3
 * @Description :
 */
@Configuration
public class RabbitConfig {

    @Bean
    public RabbitTemplate createRabbitTemplate(ConnectionFactory connectionFactory){
        RabbitTemplate rabbitTemplate = new RabbitTemplate();
        rabbitTemplate.setConnectionFactory(connectionFactory);
        //设置开启Mandatory,才能触发回调函数,无论消息推送结果怎么样都强制调用回调函数
        rabbitTemplate.setMandatory(true);

        rabbitTemplate.setConfirmCallback(new RabbitTemplate.ConfirmCallback() {
            @Override
            public void confirm(CorrelationData correlationData, boolean ack, String cause) {
                System.out.println("ConfirmCallback: "+"相关数据: "+correlationData);
                System.out.println("ConfirmCallback: "+"确认情况: "+ack);
                System.out.println("ConfirmCallback: "+"原因: "+cause);
            }
        });

        rabbitTemplate.setReturnCallback(new RabbitTemplate.ReturnCallback() {
            @Override
            public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey) {
                System.out.println("ReturnCallback: "+"消息: "+message);
                System.out.println("ReturnCallback: "+"回应码: "+replyCode);
                System.out.println("ReturnCallback: "+"回应信息: "+replyText);
                System.out.println("ReturnCallback: "+"交换机: "+exchange);
                System.out.println("ReturnCallback: "+"路由键: "+routingKey);
            }
        });

        return rabbitTemplate;
    }
}

```

到这里，生产者推送消息的消息确认调用回调函数已经完毕。

可以看到上面写了两个回调函数，一个叫 ConfirmCallback，一个叫 ReturnCallback；那么以上这两种回调函数都是在什么情况会触发呢？

先从总体的情况分析，推送消息存在四种情况：

- ①消息推送到server，但是在server里找不到交换机
- ②消息推送到server，找到交换机了，但是没找到队列
- ③消息推送到server，交换机和队列啥都没找到
- ④消息推送成功

那么我先写几个接口来分别测试和认证下以上4种情况，消息确认触发回调函数的情况：

①消息推送到server，但是在server里找不到交换机

写个测试接口，把消息推送到名为'non-existent-exchange'的交换机上（这个交换机是没有创建没有配置的）：

```

@GetMapping("/TestMessageAck")
public String TestMessageAck() {
    String messageId = String.valueOf(UUID.randomUUID());
    String messageData = "message: non-existent-exchange test message ";
    String createTime = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    Map<String, Object> map = new HashMap<>();
    map.put("messageId", messageId);
    map.put("messageData", messageData);
    map.put("createTime", createTime);
    rabbitTemplate.convertAndSend("non-existent-exchange", "TestDirectRouting", map);
    return "ok";
}

```

调用接口，查看rabbitmq-provuder项目的控制台输出情况（原因里面有说，没有找到交换机'non-existent-exchange'）：

```
2019-09-04 09:37:45.197 ERROR 8172 --- [ 127.0.0.1:5672] o.s.a.r.c.CachingConnectionFactory : Channel shutdown: channel error
ConfirmCallback: 相关数据: null
ConfirmCallback: 确认情况: false
ConfirmCallback: 原因: channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT_FOUND - no exchange
```

结论：①这种情况触发的是 ConfirmCallback 回调函数。

②消息推送到server，找到交换机了，但是没找到队列

这种情况就是需要新增一个交换机，但是不给这个交换机绑定队列，我来简单地在DirectRabbitConfig里面新增一个直连交换机，名叫'lonelyDirectExchange'，但没给它做任何绑定配置操作：

```
@Bean
DirectExchange lonelyDirectExchange() {
    return new DirectExchange("lonelyDirectExchange");
}
```

然后写个测试接口，把消息推送到名为'lonelyDirectExchange'的交换机上（这个交换机是没有任何队列配置的）：

```
@GetMapping("/TestMessageAck2")
public String TestMessageAck2() {
    String messageId = String.valueOf(UUID.randomUUID());
    String messageData = "message: lonelyDirectExchange test message ";
    String createTime = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
    Map<String, Object> map = new HashMap<>();
    map.put("messageId", messageId);
    map.put("messageData", messageData);
    map.put("createTime", createTime);
    rabbitTemplate.convertAndSend("lonelyDirectExchange", "TestDirectRouting", map);
    return "ok";
}
```

路由键

调用接口，查看rabbitmq-provuder项目的控制台输出情况：

```
ReturnCallback: 消息: (Body: '{createTime=2019-09-04 09:48:01, messageId=563077d9-0a77-4c27-8794-ecfb183eac80, messageData=message: lonelyDirectExchange test message '}')
ReturnCallback: 回应码: 312
ReturnCallback: 回应信息: NO_ROUTE
ReturnCallback: 交换机: lonelyDirectExchange
ReturnCallback: 路由键: TestDirectRouting
```

```
ConfirmCallback: 相关数据: null
ConfirmCallback: 确认情况: true
ConfirmCallback: 原因: null
```

可以看到这种情况，两个函数都被调用了；

这种情况下，消息是推送成功到服务器了的，所以ConfirmCallback对消息确认情况是true；

而在ReturnCallback回调函数的打印参数里面可以看到，消息是推送到了交换机成功了，但是在路由分发到队列的时候，找不到队列，所以报了错误 NO_ROUTE。

结论：②这种情况触发的是 ConfirmCallback和ReturnCallback两个回调函数。

③消息推送到server，交换机和队列啥都没找到

这种情况其实一看就觉得跟①很像，没错，③和①情况回调是一致的，所以不做结果说明了。

结论：③这种情况触发的是 ConfirmCallback 回调函数。

④消息推送成功

那么测试下，按照正常调用之前消息推送的接口就行，就调用下 /sendFanoutMessage接口，可以看到控制台输出：


```
ConfirmCallback: 相关数据: null
ConfirmCallback: 确认情况: true
ConfirmCallback: 原因: null
```

结论：④这种情况触发的是 ConfirmCallback 回调函数。没有ReturnCallback回调函数

以上是生产者推送消息的消息确认 回调函数的使用介绍（可以在回调函数根据需求做对应的扩展或者业务数据处理）。

接下来我们继续，消费者接收到消息的消息确认机制。

和生产者的消息确认机制不同，因为消息接收本来就是在监听消息，符合条件的消息就会消费下来。所以，消息接收的确认机制主要存在三种模式：

①自动确认，这也是默认的消息确认情况。 AcknowledgeMode.NONE

RabbitMQ成功将消息发出（即将消息成功写入TCP Socket）中立即即认为本次投递已经被正确处理，不管消费者端是否成功处理本次投递。

所以这种情况如果消费端消费逻辑抛出异常，也就是消费端没有处理成功这条消息，那么就相当于丢失了消息。

一般这种情况我们都是使用try catch捕捉异常后，打印日志用于追踪数据，这样找出对应数据再做后续处理。

② 根据情况确认，这个不做介绍

③ 手动确认，这个比较关键，也是我们配置接收消息确认机制时，多数选择的模式。

消费者收到消息后，手动调用basic.ack/basic.nack/basic.reject后，RabbitMQ收到这些消息后，才认为本次投递成功。

basic.ack用于肯定确认

basic.nack用于否定确认（注意：这是AMQP 0-9-1的RabbitMQ扩展）

basic.reject用于否定确认，但与basic.nack相比有一个限制：一次只能拒绝单条消息

消费者端以上的3个方法都表示消息已经被正确投递，但是basic.ack表示消息已经被正确处理。

而basic.nack,basic.reject表示没有被正确处理：

着重讲下reject，因为有时候一些场景是需要重新入列的。

channel.basicReject(deliveryTag, true); 拒绝消费当前消息，如果第二参数传入true，就是将数据重新丢回队列里，那么下次还会消费这消息。设置false，就是告诉服务器，我已经知道这条消息数据了，因为一些原因拒绝它，而且服务器也把这个消息丢掉就行。下次不想再消费这条消息了。

使用拒绝后重新入列这个确认模式要谨慎，因为一般都是出现异常的时候，catch异常再拒绝入列，选择是否重入列。

但是如果使用不当会导致一些每次都被你重入列的消息一直消费-入列-消费-入列这样循环，会导致消息积压。

顺便也简单讲讲 nack，这个也是相当于设置不消费某条消息。

channel.basicNack(deliveryTag, false, true);

第一个参数依然是当前消息到的数据的唯一id;

第二个参数是指是否针对多条消息；如果是true，也就是说一次性针对当前通道的消息的tagID小于当前这条消息的，都拒绝确认。

第三个参数是指是否重新入列，也就是指不确认的消息是否重新丢回到队列里面去。

同样使用不确认后重新入列这个确认模式要谨慎，因为这里也可能因为考虑不周出现消息一直被重新丢回去的情况，导致积压。

看了上面这么多介绍，接下来我们一起配置下，看看一般的消息接收 手动确认是怎么样的。

在消费者项目里，

新建MessageListenerConfig.java上添加代码相关的配置代码：

```
import com.elegant.rabbitmqconsumer.receiver.MyAckReceiver;
import org.springframework.amqp.core.AcknowledgeMode;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
```

```

import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @Author : JCccc
 * @CreateTime : 2019/9/4
 * @Description :
 */
@Configuration
public class MessageListenerConfig {

    @Autowired
    private CachingConnectionFactory connectionFactory;

    @Autowired
    private MyAckReceiver myAckReceiver;//消息接收处理类

    @Bean
    public SimpleMessageListenerContainer simpleMessageListenerContainer() {
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);
        container.setConcurrentConsumers(1);
        container.setMaxConcurrentConsumers(1);
        container.setAcknowledgeMode(AcknowledgeMode.MANUAL); // RabbitMQ默认是自动确认，这里改为手动确认消息
        //设置一个队列
        container.setQueueNames("TestDirectQueue");
        //如果同时设置多个如下： 前提是队列都是必须已经创建存在的
        // container.setQueueNames("TestDirectQueue","TestDirectQueue2","TestDirectQueue3");

        //另一种设置队列的方法,如果使用这种情况,那么要设置多个,就使用addQueues
        //container.setQueues(new Queue("TestDirectQueue",true));
        //container.addQueues(new Queue("TestDirectQueue2",true));
        //container.addQueues(new Queue("TestDirectQueue3",true));
        container.setMessageListener(myAckReceiver);

        return container;
    }
}

```

对应的手动确认消息监听类，MyAckReceiver.java（手动确认模式需要实现 ChannelAwareMessageListener）：
 //之前的相关监听器可以先注释掉，以免造成多个同类型监听器都监听同一个队列。
 //这里的获取消息转换，仅作参考，如果报数组越界可以自己根据格式去调整。

```

import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Component;
import java.util.HashMap;
import java.util.Map;

@Component

public class MyAckReceiver implements ChannelAwareMessageListener {

    @Override
    public void onMessage(Message message, Channel channel) throws Exception {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        try {
            //因为传递消息的时候用的map传递,所以将Map从Message内取出需要做些处理
            String msg = message.toString();
            String[] msgArray = msg.split("'");//可以点进Message里面看源码,单引号直接的数据就是我们的map消息数据
            Map<String, String> msgMap = mapStringToMap(msgArray[1].trim(),3);
            String messageId=msgMap.get("messageId");
            String messageData=msgMap.get("messageData");
            String createTime=msgMap.get("createTime");
            System.out.println(" MyAckReceiver messageId:"+messageId+" messageData:"+messageData+" createTime:"+createTime);
        }
    }
}

```

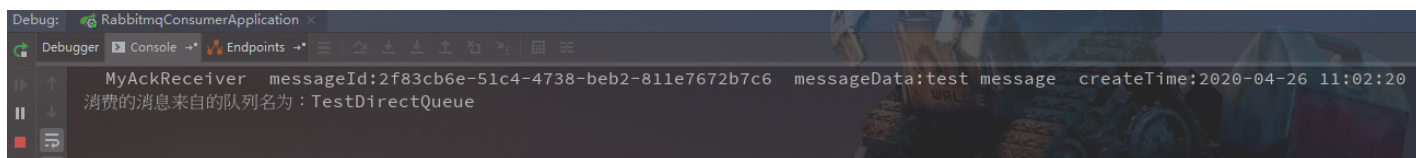
```

        System.out.println("消费的主题消息来自: "+message.getMessageProperties().getConsumerQueue());
        channel.basicAck(deliveryTag, true); //第二个参数, 手动确认可以被批处理, 当该参数为 true 时, 则可以一次性确认 delivery_tag 小于等于
    }
    channel.basicReject(deliveryTag, true); //第二个参数, true会重新放回队列, 所以需要自己根据业务逻辑判断什么时候使用拒绝
    channel.basicReject(deliveryTag, false);
    e.printStackTrace();
}
}

//{key=value,key=value,key=value} 格式转换成map
private Map<String, String> mapStringToMap(String str,int entryNum ) {
    str = str.substring(1, str.length() - 1);
    String[] strArray = str.split(",");
    Map<String, String> map = new HashMap<String, String>();
    for (String string : strArray) {
        String key = string.split("=")[0].trim();
        String value = string.split("=")[1];
        map.put(key, value);
    }
    return map;
}
}
}

```

这时, 先调用接口/sendDirectMessage, 给直连交换机TestDirectExchange 的队列TestDirectQueue 推送一条消息, 可以看到监听器正常消费了下来:



到这里, 我们其实已经掌握了怎么去使用消息消费的手动确认了。

但是这个场景往往不够! 因为很多伙伴之前给我评论反应, 他们需要这个消费者项目里面, 监听的好几个队列都想变成手动确认模式, 而且处理的消息业务逻辑不一样。

没有问题, 接下来看代码

场景: 除了直连交换机的队列TestDirectQueue需要变成手动确认以外, 我们还需要将一个其他的队列或者多个队列也变成手动确认, 而且不同队列实现不同的业务处理。

那么我们需要做的第一步, 往SimpleMessageListenerContainer里添加多个队列:

```

@Bean
public SimpleMessageListenerContainer simpleMessageListenerContainer() {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConcurrentConsumers(1);
    container.setMaxConcurrentConsumers(1);
    container.setAcknowledgeMode(AcknowledgeMode.MANUAL); // RabbitMQ默认是自动
    //设置一个队列
    //container.setQueueNames("TestDirectQueue");
    //如果同时设置多个如下: 前提是队列都是必须已经创建存在的
    container.setQueueNames("TestDirectQueue", "fanout.A");
    //另一种设置队列的方法, 如果使用这种情况, 那么要设置多个, 就使用addQueues
    //container.setQueues(new Queue("TestDirectQueue", true));
    //container.addQueues(new Queue("TestDirectQueue2", true));
    //container.addQueues(new Queue("TestDirectQueue3", true));
    container.setMessageListener(myAckReceiver);

    return container;
}

```

新设置的队列
https://blog.csdn.net/qq_35387940

然后我们的手动确认消息监听类，MyAckReceiver.java 就可以同时将上面设置到的队列的消息都消费下来。

但是我们需要做不用的业务逻辑处理，那么只需要 根据消息来自的队列名进行区分处理即可，如：

```
import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.listener.api.ChannelAwareMessageListener;
import org.springframework.stereotype.Component;
import java.util.HashMap;
import java.util.Map;

@Component
public class MyAckReceiver implements ChannelAwareMessageListener {

    @Override
    public void onMessage(Message message, Channel channel) throws Exception {
        long deliveryTag = message.getMessageProperties().getDeliveryTag();
        try {
            //因为传递消息的时候用的map传递,所以将Map从Message内取出需要做些处理
            String msg = message.toString();
            String[] msgArray = msg.split("");//可以点进Message里面看源码,单引号直接的数据就是我们的map消息数据
            Map<String, String> msgMap = mapStringToMap(msgArray[1].trim(),3);
            String messageId=msgMap.get("messageId");
            String messageData=msgMap.get("messageData");
            String createTime=msgMap.get("createTime");

            if ("TestDirectQueue".equals(message.getMessageProperties().getConsumerQueue())){
                System.out.println("消费的消息来自的队列名为: "+message.getMessageProperties().getConsumerQueue());
                System.out.println("消息成功消费到  messageId:"+messageId+ " messageData:"+messageData+ " createTime:"+createTime);
                System.out.println("执行TestDirectQueue中的消息的业务处理流程.....");
            }

            if ("fanout.A".equals(message.getMessageProperties().getConsumerQueue())){
                System.out.println("消费的消息来自的队列名为: "+message.getMessageProperties().getConsumerQueue());
                System.out.println("消息成功消费到  messageId:"+messageId+ " messageData:"+messageData+ " createTime:"+createTime);
                System.out.println("执行fanout.A中的消息的业务处理流程.....");
            }

            channel.basicAck(deliveryTag, true);
            channel.basicReject(deliveryTag, true);//为true会重新放回队列
        } catch (Exception e) {
            channel.basicReject(deliveryTag, false);
            e.printStackTrace();
        }

        // {key=value,key=value,key=value} 格式转换成map
        private Map<String, String> mapStringToMap(String str,int enNum) {
            str = str.substring(1, str.length() - 1);
            String[] strArray = str.split(",");
            Map<String, String> map = new HashMap<String, String>();
            for (String string : strArray) {
                String key = string.split("=")[0].trim();
                String value = string.split("=")[1];
                map.put(key, value);
            }
            return map;
        }
    }
}
```

ok，这时候我们来分别往不同队列推送消息，看看效果：

调用接口/sendDirectMessage 和 /sendFanoutMessage ,



如果你还想新增其他的监听队列，也就是按照这种方式新增配置即可（或者完全可以分开多个消费者项目去监听处理）。

好，这篇Springboot整合rabbitMq教程就暂且到此。