链滴

RabbitMQ 学习笔记

作者：marshalby2
原文链接：https://ld246.com/article/1602292467211
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

前言

本篇文章是我学习RabbitMQ的笔记，主要包括两大部分，第一部分是RabbitMQ的安装与基本概念第二部分是在SpringBoot项目中编写测试RabbitMQ。

RabbitMQ基础

简介

RabbitMQ是实现了高级消息队列协议（AMQP）的开源消息代理软件（也称为消息中间件），主要来实现异步通信和程序解耦。

安装

推荐使用Docker来安装

1. 拉取RabbitMQ镜像

```
docker pull rabbitmq
```

2. 创建RabbitMQ容器

```
docker run -p 5672:5672 -p 15672:15672 --name rabbitmq -d rabbitmq
```

3. 进入RabbitMQ容器

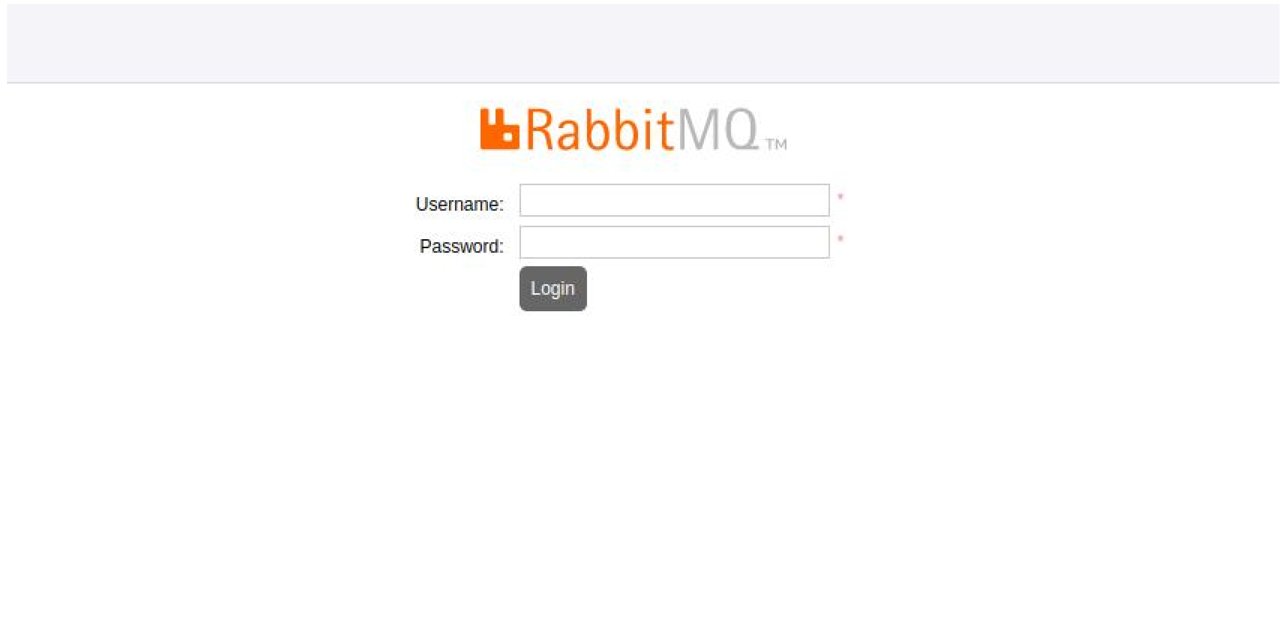
```
docker exec -it rabbitmq /bin/bash
```

4. 使用管理插件

在RabbitMQ容器里输入下面的指令即可开启插件

rabbitmq-plugins enable rabbitmq_management

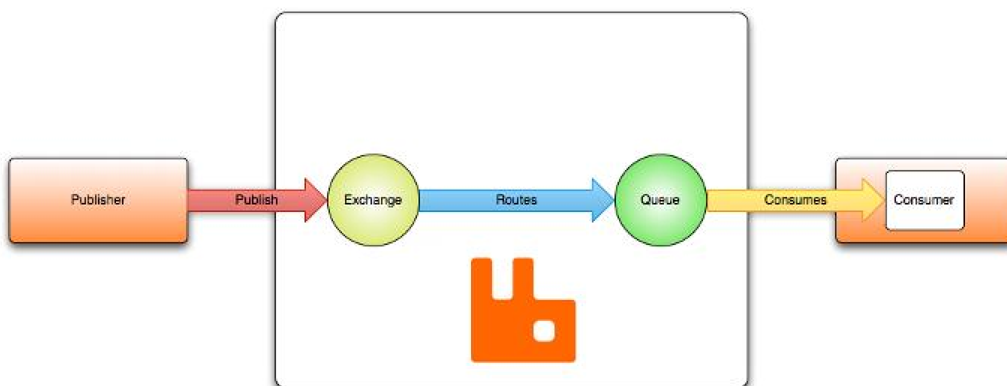
在浏览器输入 http://127.0.0.1:15672/ 出现下图所示界面



然后输入默认的账号guest，密码也是guest 就可以登录

RabbitMQ模型

RabbitMQ支持的是 AMQP 0-9-1 Model 模型，如下图所示



简单描述以下这个模型，一个 **发布者**（Publisher） 给 **交换机**（Exchange） 发送一条消息，然后
换机（Exchange） 根据 **路由关系**（binding） 将消息复制分发给和它绑定的 **队列**（Queue） ，
后，订阅了 **队列**（Queue） 的 **消费者**（Consumer） 消费这些消息。

1. 发布（生产）者（Publisher/Producer）

指的是给交换机发送消息的程序

2. 队列（Queue）

队列本质上是一个消息缓冲区，负责接收来自交换机的消息

3. 消费者（Consumer）

指的是从队列中消费消息的程序

4. 绑定（Bindings）

交换机和一个队列的关系称为一个binding

5. 交换机（Exchange）

生产者将消息发送给交换机，交换机再根据绑定关系把这些消息分发给对应的队列，交换机有四种类
，分别是

Direct Exchange

这是默认的交换机类型，Direct Exchange 会根据一个 **路由键（routing key）** 去分发消息给对应
队列。它的过程是这样的，生产者发消息的时候会指定一个routing key R， Direct Exchange 绑定
列时也会指定一个 routing key K， 当 Direct Exchange 接收到消息的时候， 会给那些满足 （K =
） 的队列分发消息。

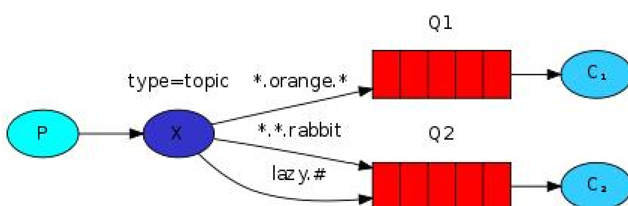
Fanout Exchange

这种类型的交换机会把接收到的消息分发给所有和自己绑定的队列（会忽略routing key），就是广
模式。

Topic Exchange

Topic 类型的交换机和direct类型的效果很像，也是根据路由键来匹配队列，但是匹配规则更加灵活。
opic Exchange 可以根据通配符 * 和 # 来匹配，其中：

- *: 表示可以替换一个字符
- #: 表示可以替换一个或者多个字符



如上图所示，共有三个绑定关系（bindings），交换机 X 和 队列 Q1有一个绑定关系 *.orange.*，和
队列 Q2 有两个绑定关系，*.*.rabbit" 和 lazy.#。举例说明一下：

1. 假设现在有一个消息设置了routing key 为 quick.orange.rabbit，那么这个消息会被交换机分给 Q1 和 Q2。

2. 假设现在有一个消息设置了routing key 为 lazy.orange.elephant，那么这个消息也会被交换机发给 Q1 和 Q2。

3. 假设现在有一个消息设置了routing key 为 quick.orange.fox，那么这个消息只会被交换机分发 Q1。

4. 假设现在有一个消息设置了routing key 为 lazy.pink.rabbit，那么这个消息会被交换机分发给 Q，并且只会分发一次。

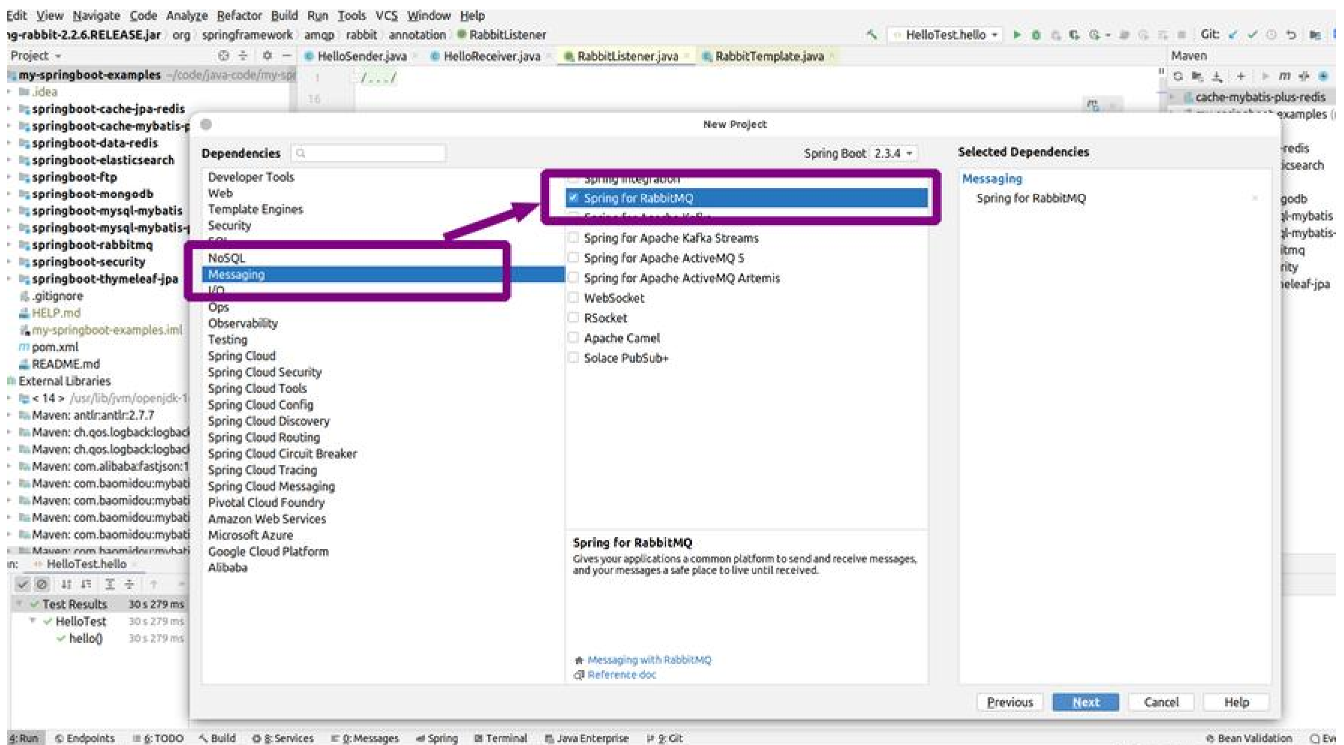
Headers Exchange

Headers 类型的交换机，舍弃了routing key，而是根据 **消息头（message headers）** 来匹配队列有两种匹配模式，消息头带有一个参数x-match，如果这个参数的值是all，代表匹配所有；如果是any代表匹配任意一个值。

SpringBoot 整合 RabbitMQ

创建RabbitMQ项目

通过IDEA创建项目时选择RabbitMQ的依赖



创建完成后，pom.xml 文件会包含以下两个依赖（我创建的时候，遇到一个问题，spring-boot-startr-parent 版本高于2.3.0会报错，找不到spring-rabbit-test依赖，所以我手动将spring-boot-starter-arent 的版本改为了2.3.0.）

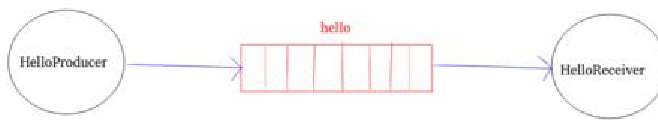
```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
```

```
<dependency>
   <groupId>org.springframework.amqp</groupId>
   <artifactId>spring-rabbit-test</artifactId>
</dependency>
```

项目创建好了，我们就开始写测试demo

一对一使用

先通过一个简单的入门程序，来感受一些RabbitMQ消息队列，模型图如下：



1. 创建RabbitMQ配置类

HelloConfig这个配置类注入了一个名为**hello**的队列

```
@Configuration
public class HelloConfig {
   @Bean
   public Queue helloQueue() {
      return new Queue("hello");
   }
}
```

2. 创建一个生产者

HelloProducer这个类负责生产消息

```
@Component
@Slf4j
public class HelloProducer {

   @Autowired
   private RabbitTemplate template;

   public void produce(int i) {
```

```
        String message = "hello : " + i;
        log.info("hello produce: ================> " + message);
        // 参考：<a>https://www.rabbitmq.com/tutorials/tutorial-one-python.html<a/>
        // The queue name needs to be specified in the routing_key parameter:
        // convertAndSend函数有三个参数，第一个是交换机名称，如果不给就是默认的；第二个是由key指队列名称；第三个是消息体
        template.convertAndSend("hello", message);
    }
}
```

3. 创建一个消费者

HelloReceiver类用来监听队列hello，负责消费从队列hello分发过来的消息。

@RabbitListener(queues = "hello")这个注解可以标注在类上面，也可以标注在方法上，当标注在上时，还需要通过 @RabbitHandler来标注接收消息的方法

```
@Component
@RabbitListener(queues = "hello")
@Slf4j
public class HelloReceiver {

    @RabbitHandler
    public void process(String message) {
        log.info("hello receive : ================> " + message);
    }
}
```

4. 测试类

测试类通过注入生产者HelloProducer，通过循环调用HelloProducer类的produce方法来产生消息

```
@SpringBootTest
@Slf4j
public class HelloTest {

    @Autowired
    private HelloProducer helloProducer;

    @Test
    public void hello() throws Exception{
        int i = 0;
        while (i < 10) {
            helloProducer.produce(i);
            Thread.sleep(3000);
            i++;
        }
    }
}
```

输出结果：

```
hello produce: ================> hello : 0
hello receive : ================> hello : 0
```

```
hello produce: ================> hello : 1
hello receive : =================> hello : 1
hello produce: ================> hello : 2
hello receive : =================> hello : 2
.....
```

5. 结论

从输出日志可以看出，生产者每生产一个消息，消费者就消费一个，这是最简单的一对一消息模型

一对多和多对多使用

1. 创建一个配置类

```
@Configuration
public class WorkQueueConfig {
    @Bean
    public Queue workQueue() {
        return new Queue("workQueue");
    }
}
```

2. 创建两个生产者

这两个生产者都将消息发送给队列workQueue

```
@Slf4j
public class WorkProducerA {

    @Autowired
    private RabbitTemplate template;

    public void produce(int i) {
        template.convertAndSend("workQueue", ("WorkProducerA  message ******* " + i));
    }
}

@Component
@Slf4j
public class WorkProducerB {

    @Autowired
    private RabbitTemplate template;

    public void produce(int i) {
        template.convertAndSend("workQueue", ("WorkProducerB  message ******* " + i));
    }
}
```

3. 创建两个消费者

这两个消费者都监听队列workQueue

```
@Component
@RabbitListener(queues = "workQueue")
@Slf4j
public class WorkReceiverA {

    @RabbitHandler
    public void process(String message) {
        log.info("workReceiverA :  " + message);
    }
}

@Component
@RabbitListener(queues = "workQueue")
@Slf4j
public class WorkReceiverB {

    @RabbitHandler
    public void process(String message) {
        log.info("workReceiverB :  " + message);
    }
}
```

4. 测试

测试类有两个方法，分别是测试一对多和多对多的

```
@SpringBootTest
public class WorkQueueTest {

    @Autowired
    private WorkProducerA workProducerA;
    @Autowired
    private WorkProducerB workProducerB;

    /**
     *  一对多
     *
     * 参考：<a>https://www.rabbitmq.com/tutorials/tutorial-two-python.html</a>
     * <p>
     * By default, RabbitMQ will send each message to the next consumer, in sequence.
     * On average every consumer will get the same number of messages
     */
    @Test
    public void oneToMany() {
        int i = 0;
        while (i < 20) {
            workProducerA.produce(i);
            i++;
        }
    }
```

```
    /**
     *  多对多
     *
     */
    @Test
    public void manyToMany() {
        int i = 0;
        while (i < 40) {
            workProducerA.produce(i);
            workProducerB.produce(i);
            i++;
        }
    }

}
```

一对多输出结果：

```
workReceiverB : WorkProducerA  message ******* 0
workReceiverA : WorkProducerA  message ******* 1
 workReceiverB : WorkProducerA  message ******* 2
workReceiverA : WorkProducerA  message ******* 3
.....
workReceiverB : WorkProducerA  message ******* 18
workReceiverA : WorkProducerA  message ******* 19
```

5. 结论

当同一个队列向多个消费者提供消息时，采取的是均分策略，每个消费者会得到数量相同的消息

多对多输出结果：

```
workReceiverB : WorkProducerA  message ******* 0
workReceiverA : WorkProducerB  message ******* 0
 workReceiverA : WorkProducerB  message ******* 1
workReceiverB : WorkProducerA  message ******* 1
workReceiverA : WorkProducerB  message ******* 2
 workReceiverB : WorkProducerA  message ******* 2
.....
workReceiverB : WorkProducerA  message ******* 39
workReceiverA : WorkProducerB  message ******* 39
```

结论：当多个生产者给同一个队列发送消息，而这个队列给多个消费者提供消息时，也是平均分配的

Direct Exchange

在实际应用中，我们可能需要根据一些实际情况，将特定类型的消息分发给特定的队列，再通过队列消息提供给特定的消费者。举个例子，我们需要做一个处理日志的功能，我们的应用程序（日志生产）会产生的error类型日志和info类型日志，现在有两个日志消费者，其中一个只接收error类型的日，另一个则接收error和info类型的日志，现在我们通过代码来实现。

1. 创建一个 Direct Exchange 配置类

该配置类创建了两个队列logQueueA 和 logQueueB，创建了一个Direcrt类型交换机（direct.logs，然后将队列logQueueA和交换机direct.logs绑定，并指定router key为error，再将队列logQueue和交换机direct.logs绑定，分别指定router key为error和info。

```
@Configuration
public class DirectConfig {


    @Bean
    public Queue logQueueA() {
        return new Queue("logQueueA");
    }

    @Bean
    public Queue logQueueB() {
        return new Queue("logQueueB");
    }

    @Bean
    public DirectExchange directExchange() {
        return new DirectExchange("direct.logs");
    }


    /**
     * logQueueA 和 error 日志绑定
     *
     * @return
     */
    @Bean
    public Binding bindingExchangeErrorLogs() {
        return BindingBuilder.bind(logQueueA()).to(directExchange()).with("error");
    }


    /**
     * logQueueB 和 error 日志绑定
     *
     * @return
     */
    @Bean
    public Binding bindingExchangeMixLogsA() {
        return BindingBuilder.bind(logQueueB()).to(directExchange()).with("error");
    }

    /**
     * logQueueB 和 info 日志绑定
     *
     * @return
     */
    @Bean
    public Binding bindingExchangeMixLogsB() {
        return BindingBuilder.bind(logQueueB()).to(directExchange()).with("info");
    }
```

```
}
```

2. 创建一个日志生产者

模拟生产日志消息，根据传入的参数类型，决定日志类型和router key

```
@Configuration
public class LogsProducer {

    @Autowired
    private RabbitTemplate template;


    /**
     * 根据参数指定routingKey
     *
     * @param type
     */
    public void produce(String type) {
        template.convertAndSend("direct.logs", type, "This is " + type + " logs");
    }

}
```

3. 创建两个日志消费者

消费者LogReceiverA监听队列logQueueA

```
@Component
@Slf4j
@RabbitListener(queues = "logQueueA")
public class LogReceiverA {

    @RabbitHandler
    public void receive(String message) {
        log.info("error receiver receive : " + message);
    }

}
```

消费者LogReceiverB监听队列logQueueB

```
@Component
@Slf4j
@RabbitListener(queues = "logQueueB")
public class LogReceiverB {

    @RabbitHandler
    public void receive(String message) {
        log.info("mix receiver receive : " + message);
    }

}
```

4. 测试类

测试类有两个方法，分别测试生产error类型日志和info类型日志，当生产error类型日志时，LogRece verA和LogReceiverB两个消费者都接收到了日志消息；当生产info日志时，只有消费者LogReceiver 接收到日志消息。

```
@SpringBootTest
public class DirectTest {

    @Autowired
    private LogsProducer logsProducer;

    @Test
    public void testErrorLog() {
        logsProducer.produce("error");
        // mix receiver receive : This is error logs
        // error receiver receive : This is error logs
    }

    @Test
    public void testInfoLog() {
        logsProducer.produce("info");
        // mix receiver receive : This is info logs
    }
}
```

5.结论

Direct Exchange 通过router key来对消息进行匹配

Fanout Exchange

Fanout Exhcnage 主要应用于广播消息，该类型的交换机会把生产者生产的消息通过广播的形式发给所有和自己绑定的交换机。

1. 创建Fanout Exchange配置类

该配置类创建了三个队列broadcastQueueA、broadcastQueueB、broadcastQueueC，又创建了个Fanout Exhcnage （fanout.broadcast），再将这三个队列都和交换机（fanout.broadcast）绑 。

```
@Configuration
public class FanoutConfig {

    @Bean
    public Queue broadcastQueueA() {
        return new Queue("broadcastQueueA");
    }

    @Bean
    public Queue broadcastQueueB() {
        return new Queue("broadcastQueueB");
```

```
    }

    @Bean
    public Queue broadcastQueueC() {
        return new Queue("broadcastQueueC");
    }

    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanout.broadcast");
    }


    @Bean
    public Binding bindingExchangeMessageA() {
        return BindingBuilder.bind(broadcastQueueA()).to(fanoutExchange());
    }

    @Bean
    public Binding bindingExchangeMessageB() {
        return BindingBuilder.bind(broadcastQueueB()).to(fanoutExchange());
    }

    @Bean
    public Binding bindingExchangeMessageC() {
        return BindingBuilder.bind(broadcastQueueC()).to(fanoutExchange());
    }
}
```

2. 创建生产者

该生产者主要生产一条广播消息

```
@Component
@Slf4j
public class BroadcastProducer {

    @Autowired
    private RabbitTemplate template;

    public void send() {
        String context = "This is a broadcast message";
        template.convertAndSend("fanout.broadcast","", context);
    }
}
```

3. 创建消费者

消费者BroadcastReceiverA监听队列broadcastQueueA

```
@Component
@Slf4j
@RabbitListener(queues = "broadcastQueueA")
```

```
public class BroadcastReceiverA {

    @RabbitHandler
    public void process(String message) {
        log.info("BroadcastReceiverA receive : " + message);
    }

}
```

消费者BroadcastReceiverB监听队列broadcastQueueB

```
@Component
@Slf4j
@RabbitListener(queues = "broadcastQueueB")
public class BroadcastReceiverB {

    @RabbitHandler
    public void process(String message) {
        log.info("BroadcastReceiverB receive : " + message);
    }

}
```

消费者BroadcastReceiverC监听队列broadcastQueueC

```
@Component
@Slf4j
@RabbitListener(queues = "broadcastQueueC")
public class BroadcastReceiverC {

    @RabbitHandler
    public void process(String message) {
        log.info("BroadcastReceiverC receive : " + message);
    }

}
```

4. 测试类

```
@SpringBootTest
public class FanoutTest {

    @Autowired
    private BroadcastProducer fanoutSender;

    /**
     * 发布订阅模式
     *
     */
    @Test
    public void testFanout() {
        fanoutSender.send();

    }
```

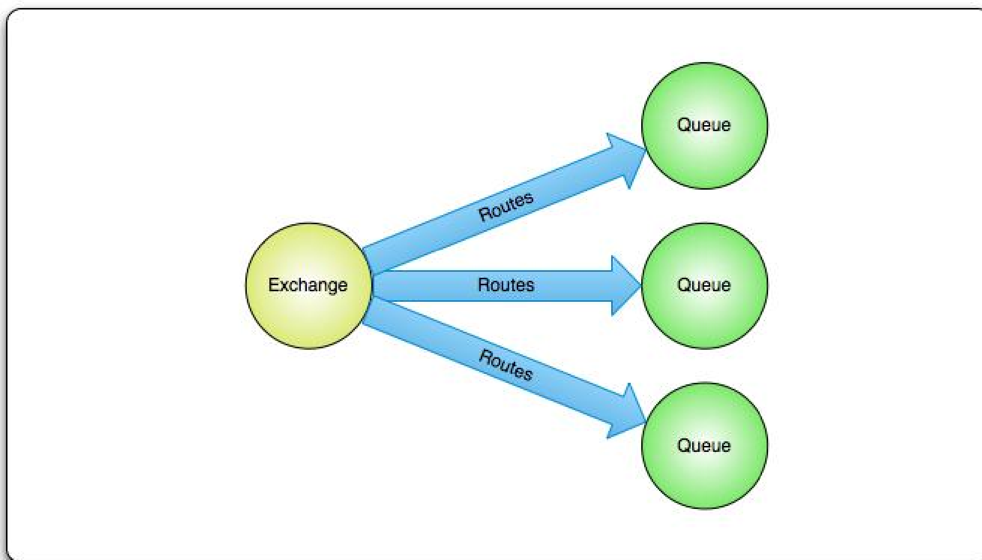
```
}
```

输出结果：

```
BroadcastReceiverA receive : This is a broadcast message
BroadcastReceiverB receive : This is a broadcast message
BroadcastReceiverC receive : This is a broadcast message
```

5. 结论

Fanout Exhcnage 不需要指定router key，它会将接收到的来自生产者的生产的消息发送给所有和自绑定的队列，它的路由效果如下图所示：



Topic Exchange

Topic Exchange也是根据router key来匹配队列，但是匹配规则很灵活，因为可以根据通配符来匹配我们通过一个匹配颜色的例子来演示

1. 创建 Topic Exchange 配置类

该配置类创建了两个队列colorQueueA和colorQueueB，以及一个Topic Exchange（topic.color）再将队列colorQueueA和交换机绑定，并指定了router ky 为 *.blue.*，将队列colorQueueB和交换绑定，并指定了router key为 green.#

```
@Configuration
public class TopicConfig {

    @Bean
    public Queue topicQueueA() {
        return new Queue("colorQueueA");
    }
```

```
    @Bean
    public Queue topicQueueB() {
        return new Queue("colorQueueB");
    }

    @Bean
    public TopicExchange colorTopicExchange() {
        return new TopicExchange("topic.color");
    }

    @Bean
    public Binding bindingExchangeTopicQueueA() {
        return BindingBuilder.bind(topicQueueA()).to(colorTopicExchange()).with("*.blue.*");
    }

    @Bean
    public Binding bindingExchangeTopicQueueB() {
        return BindingBuilder.bind(topicQueueB()).to(colorTopicExchange()).with("green.#");
    }
}
```

2. 创建生产者

根据传入的参数，指定router key

```
@Component
@Slf4j
public class ColorProducer {

    @Autowired
    private RabbitTemplate template;

    public void produce(String criteria) {
        template.convertAndSend("topic.color", criteria, "This is " + criteria);
    }

}
```

3. 创建两个消费者

消费者ColorReceiverA监听队列colorQueueA

```
@Component
@Slf4j
@RabbitListener(queues = "colorQueueA")
public class ColorReceiverA {

    @RabbitHandler
    public void receive(String color) {
        log.info("ColorReceiverA receive: " + color);
    }

}
```

消费者ColorReceiverB监听队列colorQueueB

```
@Component
@Slf4j
@RabbitListener(queues = "colorQueueB")
public class ColorReceiverB {

    @RabbitHandler
    public void receive(String color) {
        log.info("ColorReceiverB receive: " + color);
    }

}
```

4. 测试类

在测试类中，我们写了两个测试方法，分别指定router key为yellow.blue.red 和 green.blue.red

```
@SpringBootTest
public class TopicTest {

    @Autowired
    private ColorProducer colorProducer;


    @Test
    public void test1() {
        String criteria = "yellow.blue.red";
        colorProducer.produce(criteria);
    }


    @Test
    public void test2() {
        String criteria = "green.blue.red";
        colorProducer.produce(criteria);
    }

}
```

输出结果：

test1方法输出

```
ColorReceiverA receive: This is yellow.blue.red
```

test2方法输出

```
ColorReceiverA receive: This is green.blue.red\
ColorReceiverB receive: This is green.blue.red
```

5. 结论

Topic Exchange 中的router key可以写成通配符的格式，如*.blue.*或者green.#，这样使得匹配规更加灵活

Headers Exchange

Headers Exchange 没有router key了，而是根据消息头来匹配。消息头内容是key-value键值对格，可以指定多个key-value键值对，所以需要指定匹配模式是all还是any，即匹配所有的key-value键对，还是只匹配其中一条。

1. 创建Headers Exchange配置类

该配置类创建了两个队列imageQueueA和imageQueueB，以及一个Headers Exchange（headers.mage），再将imageQueueA和交换机（headers.image）绑定，并且指定匹配模式为all，再将imaeQueueA和交换机（headers.image）绑定，并且指定匹配模式为any

```
@Configuration
public class HeadersConfig {

    @Bean
    public Queue imageQueueA() {
        return new Queue("imageQueueA");
    }

    @Bean
    public Queue imageQueueB() {
        return new Queue("imageQueueB");
    }


    @Bean
    public HeadersExchange headersExchange() {
        return new HeadersExchange("headers.image");
    }

    /**
     * 匹配type=jpg且size=12的消息，分发给队列imageQueueA
     *
     * @return
     */
    @Bean
    public Binding bindingHeadersExchangeA() {
        Map<String, Object> keys = Maps.newHashMap();
        keys.put("type", "jpg");
        keys.put("size", 12);
        return BindingBuilder.bind(imageQueueA()).to(headersExchange()).whereAll(keys).match(
;
    }

    /**
     *  匹配type=png或者size=6的消息，分发给队列imageQueueB
     *
     * @return
     */
```

```
    @Bean
    public Binding bindingHeadersExchangeB() {
        Map<String, Object> keys = Maps.newHashMap();
        keys.put("type", "png");
        keys.put("size", 6);
        return BindingBuilder.bind(imageQueueB()).to(headersExchange()).whereAny(keys).match
();
    }
}
```

2. 创建生产者类

为了方便传参，我们定义了一个类Image

```
@Builder
@ToString
@Getter
public class Image implements Serializable {
    private static final long serialVersionUID = 8617592564349459927L;
    private String type;
    private int size;
}
```

生产者类，根据传入的参数指定消息头内容

```
@Component
public class ImagesProducer {

    @Autowired
    private RabbitTemplate template;

    public void produce(Image image) {
        MessageProperties properties = new MessageProperties();
        properties.setHeader("type", image.getType());
        properties.setHeader("size", image.getSize());
        template.convertAndSend("headers.image", "", new Message(image.toString().getBytes(),
properties));
    }
}
```

3. 创建消费者类

消费者ImageReceiverA监听队列imageQueueA

```
@Component
@Slf4j
public class ImageReceiverA {

    @RabbitListener(queues = "imageQueueA")
    public void receive(byte[] image) {
        log.info("ImageReceiverA receive : " + new String(image));
    }
}
```

消费者ImageReceiverB监听队列imageQueueB

```
@Component
@Slf4j
public class ImageReceiverB {

    @RabbitListener(queues = "imageQueueB")
    public void receive(byte[] image) {
        log.info("ImageReceiverB receive : " + new String(image));
    }
}
```

4. 测试类

```
@SpringBootTest
public class HeadersTest {

    @Autowired
    private ImagesProducer imagesProducer;

    @Test
    public void testAll() {
        imagesProducer.produce(Image.builder().type("jpg").size(12).build());
        // ImageReceiverA receive : Image(type=jpg, size=12)
    }

    @Test
    public void testAny() {
        imagesProducer.produce(Image.builder().type("png").size(12).build());
        // ImageReceiverB receive : Image(type=png, size=12)
    }

    @Test
    public void testAny2() {
        imagesProducer.produce(Image.builder().type("jpg").size(6).build());
        // ImageReceiverB receive : Image(type=jpg, size=6)
    }
}
```

输出结果：

testAll方法输出：

```
ImageReceiverA receive : Image(type=jpg, size=12)
```

testAny方法输出：

```
mageReceiverB receive : Image(type=png, size=12)
```

testAny2方法输出：

```
ImageReceiverB receive : Image(type=jpg, size=6)
```

5. 结论

Headers Exchange 可以灵活增加或减少消息头中的键值对来实现匹配。

传输对象

前面的例子都是传的字符串，在实际的开发中经常需要传输对象，现在我们写一个例子来传输对象

1. 创建一个实体类

```
@Builder
@ToString
public class User implements Serializable {
    private static final long serialVersionUID = 4409039369681054682L;
    private String name;
    private int age;
}
```

2. 创建配置类

这里使用的是 Topic Exchange

```
@Configuration
public class UserConfig {

    @Bean
    public Queue userInfo() {
        return new Queue("user.info");
    }

    @Bean
    public TopicExchange userTopicExchange() {
        return new TopicExchange("topic.user");
    }

    @Bean
    public Binding bindingTopicExchange() {
        return BindingBuilder.bind(userInfo()).to(userTopicExchange()).with("user.info");
    }
}
```

3. 创建生产者

```
@Component
public class UserProducer {

    @Autowired
    private RabbitTemplate template;

    public void produce() {
        template.convertAndSend("topic.user", "user.info", User.builder().name("Tom").age(20).b
ild());
    }
```

```
}
```

4. 创建消费者

```
@Component
@Slf4j
public class UserReceiver {

    /**
     *  RabbitListener注解可以直接作用于方法
     *
     * @param user
     */
    @RabbitListener(queues = "user.info")
    public void receive(User user) {
        log.info("receive user : " + user);
    }

}
```

5. 测试类

```
@SpringBootTest
public class UserTest {

    @Autowired
    private UserProducer userProducer;


    @Test
    public void test() {
        userProducer.produce();
    }
}
```

输出结果：

```
receive user : User(name=Tom, age=20)
```

最后

RabbitMQ的基本概念和SpringBoot整合RabbitMQ的常用模式都总结完毕了，以后项目中需要实际用RabbitMQ时，再去深入研究吧。

本篇文章的完整代码