目录

[1 RabbitMQ环境 2](#_Toc36315478)

[1.1 RabbitMQ原理讲解 2](#_Toc36315479)

[1.2 官网 2](#_Toc36315480)

[1.3 所用端口 2](#_Toc36315481)

[1.4 Linux环境搭建 3](#_Toc36315482)

[1.4.1 系统准备 3](#_Toc36315483)

[1.4.2 安装软件-erlang 3](#_Toc36315484)

[1.4.3 安装软件-socat 3](#_Toc36315485)

[1.4.4 安装软件-RabbitMQ-SERVER 3](#_Toc36315486)

[1.4.5 设置开机启动 3](#_Toc36315487)

[1.4.6 启动关闭RabbbitMQ-Server 4](#_Toc36315488)

[1.5 Windows安装RabbitMQ 4](#_Toc36315489)

[1.5.1 安装必要的软件 4](#_Toc36315490)

[1.5.2 安装管理插件、启动、登录 4](#_Toc36315491)

[2 RabbitMQ基础开发 5](#_Toc36315492)

[2.1 Maven添加依RabbitMQ的jar包依赖 5](#_Toc36315493)

[2.2 创建创建连接和信道 5](#_Toc36315494)

[2.3 声明一个队列（货架子） 6](#_Toc36315495)

[2.4 使用信道发送数据 6](#_Toc36315496)

[2.5 2.6 最简单的消息接收 6](#_Toc36315497)

[2.6 最简单的消息接收 6](#_Toc36315498)

[2.7 队列声明三个参数含义 7](#_Toc36315499)

[2.7.1 队列的自动删除 7](#_Toc36315500)

[2.7.2 durable队列持久化 7](#_Toc36315501)

[2.7.3 exclusive 队列的独占 7](#_Toc36315502)

[2.8 给包裹贴一张详情单 7](#_Toc36315503)

[2.9 交换机Exchange 8](#_Toc36315504)

[2.9.1 编写代码创建一个直连型交换机 8](#_Toc36315505)

[2.9.2 交换机绑定队列 8](#_Toc36315506)

[2.9.3 direct直连型交换机 9](#_Toc36315507)

[2.9.4 fanout型交换机(废掉路由规则) 9](#_Toc36315508)

[2.9.5 topic主题型交换机（主题匹配） 9](#_Toc36315509)

[2.9.6 Header自定义属性型 10](#_Toc36315510)

[2.10 设置大小限制的队列 11](#_Toc36315511)

[2.11 生产端确认消息是否发送到了MQ 11](#_Toc36315512)

[2.12 Return 作用（确保路由规则设置正确） 11](#_Toc36315513)

[2.13 消费者限流QOS（削峰作用、重回队列） 12](#_Toc36315514)

[2.14 自定义消费者处理 12](#_Toc36315515)

[2.15 TTL（消息的最大生存时长） 12](#_Toc36315516)

[2.16 死信队列（问题产品（专用）货架） 13](#_Toc36315517)

[3 RabbitMQ和Spring的整合 14](#_Toc36315518)

[3.1 重要的类 14](#_Toc36315519)

[3.2 RabbitAdmin管理交换机/队列/绑定 15](#_Toc36315520)

[3.2.1 创建SpringBoot工程 15](#_Toc36315521)

[3.2.2 使用RabbitAdmin创建交换机队列绑定 16](#_Toc36315522)

[3.2.3 rabbitAdmin清空队列消息 16](#_Toc36315523)

[3.2.4 使用Spring创建交换机和队列（不用刻意声明） 16](#_Toc36315524)

[3.3 RabbitTemplate用于发送和接收消息 17](#_Toc36315525)

[3.3.1 使用SimpleMessageListenerContainer监听队列消息 18](#_Toc36315526)

[3.3.2 MessageListenerAdapter消息监听适配器 19](#_Toc36315527)

[3.4 类型转换器（实现接口MessageConverter） 20](#_Toc36315528)

[3.5 还可以给不同的队列不同的处理方法 20](#_Toc36315529)

[3.5.1 发送并接收到json 21](#_Toc36315530)

[3.5.2 消费者端直接收到java对象 22](#_Toc36315531)

[3.5.3 全局转换器（多种类型） 23](#_Toc36315532)

[3.6 开始整合SpringBoot 23](#_Toc36315533)

[3.6.1 配置Application.properties 23](#_Toc36315534)

[3.6.2 定义一个空配置文件自动扫描bean 24](#_Toc36315535)

[3.6.3 生产端发送数据 24](#_Toc36315536)

[3.6.4 使用@RabbitListener消费数据 25](#_Toc36315537)

[3.6.5 使用@QueueBinding创建队列、交换机和绑定 25](#_Toc36315538)

# RabbitMQ环境

## RabbitMQ原理讲解

rabbitMQ是一个消息中间件，消息中间件就是一家负责联系厂家和消费者的快递公司，整个课程就以生产手机和消费手机为例

## 官网

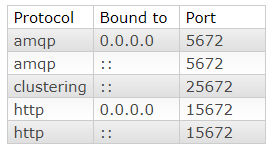
<http://www.rabbitmq.com/>

## 所用端口

5672：amqp专用端口，开发必用

15672：管理界面专用

25672：集群专用



## Linux环境搭建

### 系统准备

VMWare 15.0

RHEL7.2/CentOS 7.2

### 安装软件-erlang

http://www.erlang.org/

<http://www.erlang.org/downloads>

rpm –i er\*

### 安装软件-socat

<https://pkgs.org/download/socat>

rpm –I so\*

### 安装软件-RabbitMQ-SERVER

http://www.rabbitmq.com/

http://www.rabbitmq.com/download.html

rpm –I rabb\*

### 设置开机启动

chkconfig rabbitmq-server on

### 启动关闭RabbbitMQ-Server

启动 rabbitmq-server start &

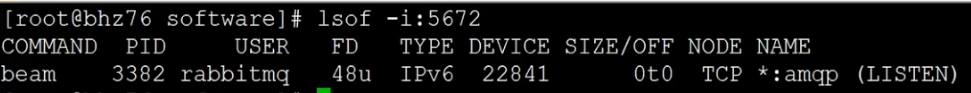
停止 rabbitmqctl app\_stop

管理插件：rabbitmq-plugins enable rabbitmq\_management

访问地址：http://127.0.0.1:15672/

管理员帐号：guest/guest

lsof –i:5672 检查是否启动，可以看到rabbitmq使用的amqp协议



## Windows安装RabbitMQ

### 安装必要的软件

Jdk1.8以上，Java环境的要求，至少要在8版本，或者更高版本，因为会使用lambda表达式

安装otp\_win64\_21.3.exe

安装rabbitmq-server-3.7.14.exe

### 安装管理插件、启动、登录

安装并启动UI插件的命令

rabbitmq-plugins.bat enable rabbitmq\_management

启动服务器的命令

rabbitmq-server.bat

访问地址

http://127.0.0.1:15672/

管理员帐号

guest/guest

# RabbitMQ基础开发

## Maven添加依RabbitMQ的jar包依赖

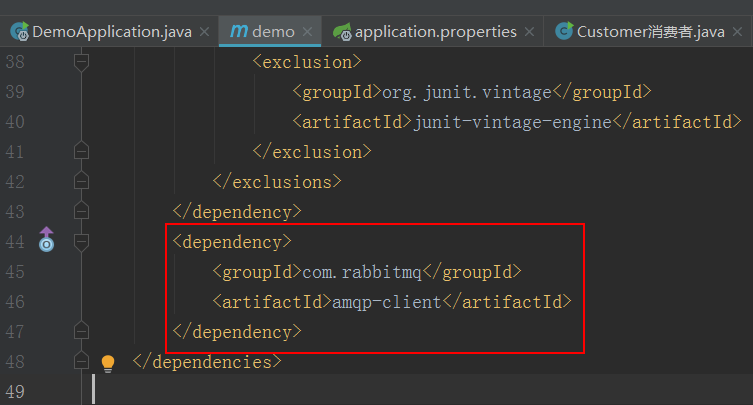
配置Maven，添加依赖。把下面的依赖包加入到dependency

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

</dependency>



## 创建创建连接和信道

// 创建连接工厂，用来修建高速公路

ConnectionFactory connectionFactory = new ConnectionFactory();

connectionFactory.setHost("localhost");

connectionFactory.setPort(5672);

connectionFactory.setUsername("guest");

connectionFactory.setPassword("guest");

// 修建高速公路

Connection connection = connectionFactory.newConnection();

// 画定双向车道

Channel channel = connection.createChannel();

## 声明一个队列（货架子）

// 声明一个消息队列，（存放小米手机的货架子）

channel.queueDeclare("QueueXiaoMi2", true, false, false, null);

## 使用信道发送数据

// 发送数据

String msg = "一部小米手机";

channel.basicPublish("","Queue小米队列-自动删除",null,msg.getBytes());

## 2.6 最简单的消息接收

发送数据，接收数据

不使用交换机，路由规则即交换机的方式

// 接收消息

DefaultConsumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag,

Envelope envelope,

AMQP.BasicProperties properties,

byte[] body)

throws IOException

{

String strRecv = "我收到了" + new String(body);

System.out.println(strRecv);

}

};

channel.basicConsume("QueueXiaoMi", true,consumer);

## 最简单的消息接收

Envelope envelope = delivery.getEnvelope();

System.out.println("消费端: " + msg + " EnvlopeTag = " + envelope.getDeliveryTag());

getDeliveryTag 表示一个消息的ID ，相当于一个包裹的唯一标识，用来进行信息确认。

## 队列声明三个参数含义

### 队列的自动删除

boolean autoDelete

在第一次使用后，如果没有连接，并且队列里没有数据，队列就会自动删除。

### durable队列持久化

消息持久化

channel.basicPublish("", "routingkey", MessageProperties.PERSISTENT\_TEXT\_PLAIN

deliveryMode 1=内存，2=序列化

### exclusive 队列的独占

boolean exclusive

同一时刻，只能有一个程序连接队列。

独占队列必定是自动删除的队列，给自己用的，无法实现进程间的通信。

## 给包裹贴一张详情单



Expiration 过期时间，毫秒

// 系统内置的常见属性

MessageProperties.PERSISTENT\_TEXT\_PLAIN

// 自定义包裹属性

BasicProperties prop = new BasicProperties()

.builder()

.contentEncoding("UTF-8")

.expiration("10000")

.deliveryMode(2)

.headers(headers)

.build();

// 检查包裹的详情单是否正确

Map<String,Object> headers = properties.getHeaders();

if(headers != null) {

Iterator<Map.Entry<String, Object>> it = headers.entrySet().iterator();

while(it.hasNext()) {

Map.Entry<String, Object> ent = (Map.Entry<String, Object>)it.next();

System.out.println( ent.getKey() + " = " + ent.getValue());

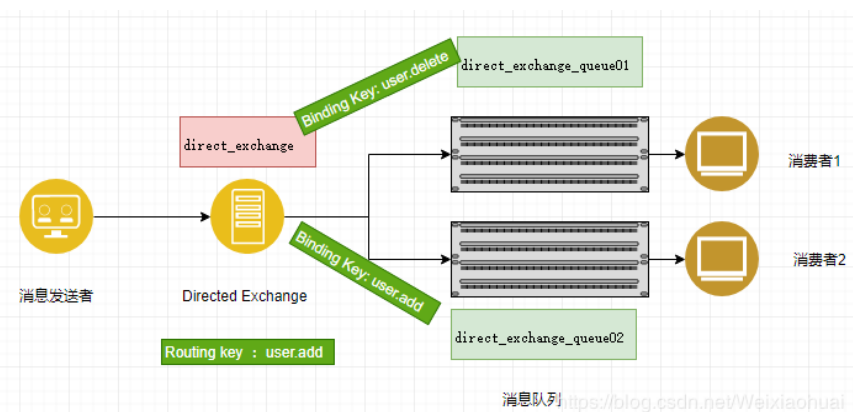
}

}

## 交换机Exchange

交换机的好处是可以更灵活的配置把消息路由到队列

交换机的几种类型：直连型交换机，fanout交换机，topic交换机，header型交换机



### 编写代码创建一个直连型交换机

// 声明一个交换机

channel.exchangeDeclare("ExchangeXiaoMi2", "direct", true, false, null);

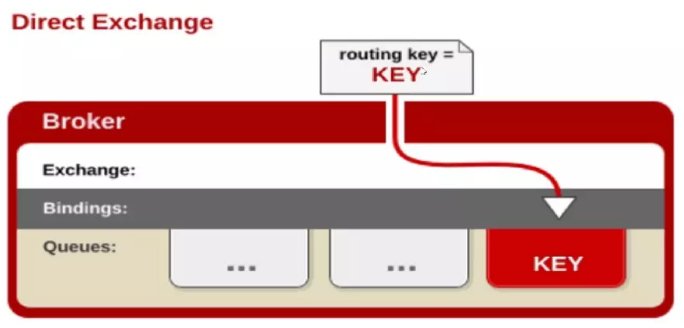
### 交换机绑定队列

channel.queueBind("队列名","交换机名", "routingKey路由键",特殊属性Map);

### direct直连型交换机

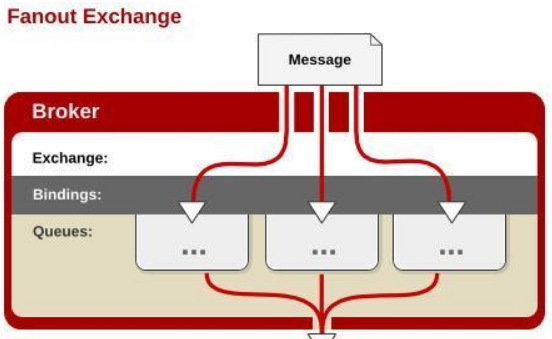
与路由规则完全一样的。

不处理路由键,直接使用给定的路由键进行匹配，完全匹配。你只需要简单的将队列绑定到交换机上。一个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。Fanout交换机转发消息是最快的。



### fanout型交换机(废掉路由规则)

不处理路由键。你只需要简单的将队列绑定到交换机上。一个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。Fanout交换机转发消息是最快的。



### topic主题型交换机（主题匹配）

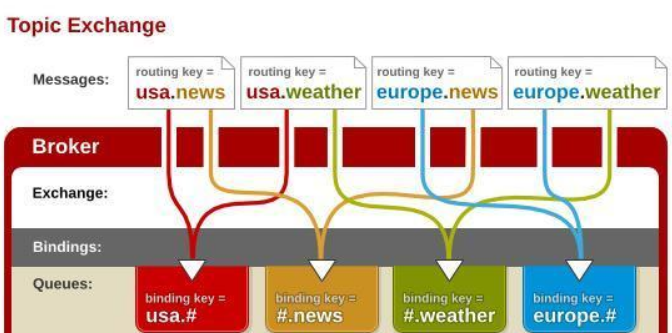
将路由键和某模式进行匹配。此时队列需要绑定要一个模式上。

“.”单词的分隔符(不是必须，可以使用其他分隔符)

“\*” 可以匹配一个单词，也只可以是0个单词

“#” 可以匹配多个单词，或者是0个

当一个队列被绑定为routing key为”#”时，它将会接收所有的消息，此时等价于fanout类型交换机。当routing key不包含”\*”和”#”时，等价于direct型交换机。



### Header自定义属性型

不处理路由键。而是根据发送的消息内容中的headers属性进行匹配。在绑定Queue与Exchange时指定一组键值对；当消息发送到RabbitMQ时会取到该消息的headers与Exchange绑定时指定的键值对进行匹配；如果完全匹配则消息会路由到该队列，否则不会路由到该队列。

匹配规则x-match有下列两种类型：

x-match = all ：表示**所有的**键值对都匹配才能接受到消息（不包括x-match）

x-match = any ：表示**至少有一个**键值对匹配就能接受到消息

// header交换机

channel.exchangeDeclare("exchange交换机小米-header-mathc-all","headers",true,false,null);

channel.queueDeclare("Queue小米header-mathc-all", true, false, false, null);

//设置消息头键值对信息

Map<String, Object> headers = new Hashtable<String,Object>();

//这里x-match有两种类型

//all:表示所有的键值对都匹配才能接受到消息

//any:表示最少一个键值对匹配就能接受到消息

headers.put("x-match", "any");

headers.put("name", "jack");

headers.put("age" , 31);

channel.queueBind("Queue小米header-mathc-all","exchange交换机小米-header-mathc-all", "", headers);

## 设置大小限制的队列

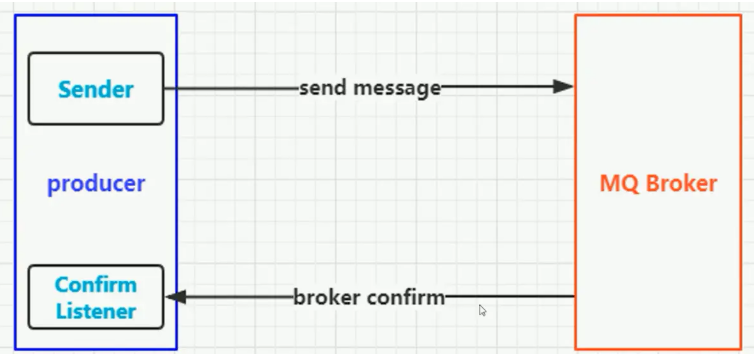
Map<String, Object> props = new HashMap<String, Object>();

props.put("x-max-length", 2);

channel.queueDeclare("Queue最多2条的队列", true, false, false, props);

## 生产端确认消息是否发送到了MQ

生产者将信道设置成confirm模式，一旦消息被被MQ接收了，MQ就会发送一个确认给生产者(包含消息的唯一ID)；否则就表示消息没有到达MQ，可能由于网络闪断等原因。



// 开启发送方确认模式

channel.confirmSelect();

chann.addConfirmListener((deliveryTag, multiple) -> {

// 成功发送到的处理

},(deliveryTag, multiple) -> {

// 发送失败的处理，这种情况很难被测试到。

});

## Return 作用（确保路由规则设置正确）

在某种情况下，如果我们在发送消息的时候，当前的路由key错误，需要监听这种不可达的消息，就要使用return listener。

basicPublish 的参数Mandatory为false时，如果消息无法正确路由到队列后，会被MQ直接丢失

basicPublish 的参数mandatory设置true，消息无法正确路由到队列后，会返还给发送者

用于处理交换机路由错误和rountingKey路由错误的处理。下面的false就是Mandatory

channel.basicPublish("Exchang ","鸡肋啊鸡肋",**false**,null,msg.getBytes());

channel.addReturnListener(new ReturnListener() {

@Override

public void handleReturn(int arg0, String arg1, String arg2, String arg3, BasicProperties arg4, byte[] arg5)

throws IOException {

//此处便是执行Basic.Return之后回调的地方

String message = new String(arg5);

System.out.println("Basic.Return返回的结果: "+message);

}

});

## 消费者限流QOS（削峰作用、重回队列）

//同一时刻服务器只会发送一条消息给消费者

channel.basicQos(10000);

据说prefetchSize 和global这两项，rabbitmq没有实现，暂且不研究

channel.basicQos(0, 10000, false);

消费者确认一条，再处理一条。

最后一个false，是否为多条。

channel.basicAck(delivery.getEnvelope().getDeliveryTag(),false);

第一个false，决定是否为多条。

第二个false，决定是否让消息重回队列。

channel.basicNack(delivery.getEnvelope().getDeliveryTag(),false,false);

消费时必须取消自动确认

channel.basicConsume("队列Qos限流",true,cosumer);

## 自定义消费者处理

继承子DefaultConsumer

## TTL（消息的最大生存时长）

1. TTL是Time To Live的缩写, 也就是生存时间
2. RabbitMQ支持消息的过期时间, 在消息发送时可以进行指定
3. RabbitMQ支持队列的过期时间, 从消息入队列开始计算, 只要超过了队列的超时时间配置, 那么消息会自动清除

队列里的**所有消息**最长的生存时间60秒

Map<String, Object> argss = new HashMap<String, Object>();

argss.put("x-message-ttl",60000);// 6秒后消息还没有被消费，会被丢弃

channel.queueDeclare(queueName, durable, exclusive, autoDelete, argss);

发送的这**一条消息**的生存时间为6秒，超过6秒后，消息自动被中间件删除

AMQP.BasicProperties.Builder builder = new AMQP.BasicProperties.Builder();

builder.expiration("6000");

AMQP.BasicProperties properties = builder.build();

channel.basicPublish(exchangeName,routingKey,mandatory,properties,"ttlTestMessage".getBytes());

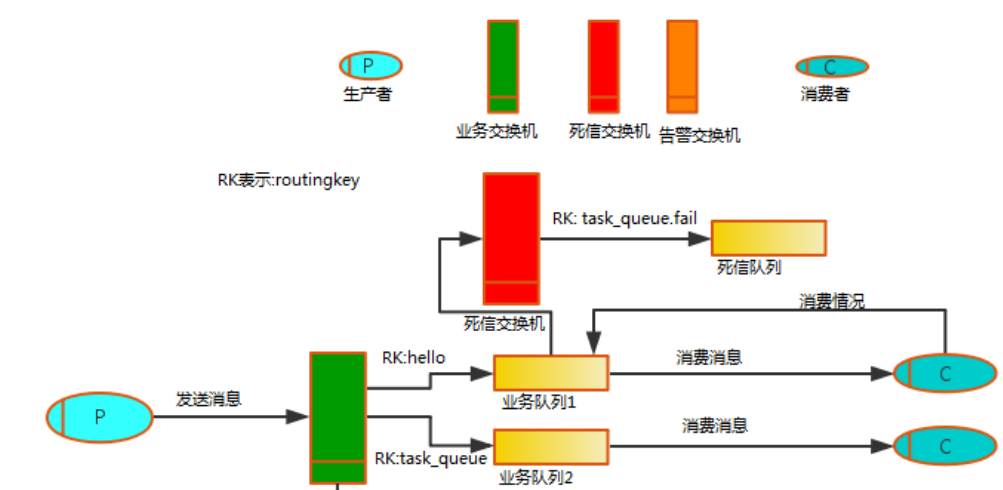
## 死信队列（问题产品（专用）货架）

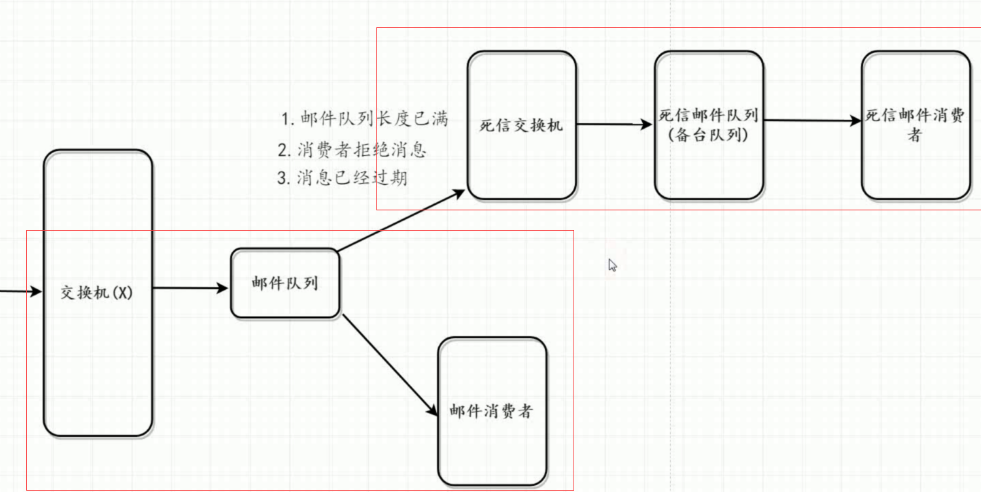
测试时，让消息过期后，消息会被转移到死信队列上

出现死信的三种情况

* 消息被拒绝（basic.reject/ basic.nack）并且requeue=false
* 消息TTL过期（参考：RabbitMQ之TTL（Time-To-Live 过期时间））
* 队列达到最大长度

给一个正常队列A添加一个死信队列DLX，当A中出现死信时，会被自动路由到DLX交换机对应的队列，从而不去影响正常的队列处理数据





Map<String, Object> agruments = new HashMap<String, Object>();

agruments.put("x-dead-letter-exchange", "dlx.exchange");

//这个agruments属性，要设置到声明正常队列上

channel.queueDeclare(queueName, true, false, false, agruments);

channel.queueBind(queueName, exchangeName, routingKey);

//要进行死信队列的声明:

channel.exchangeDeclare("dlx.exchange", "topic", true, false, null);

channel.queueDeclare("dlx.queue", true, false, false, null);

channel.queueBind("dlx.queue", "dlx.exchange", "#");

# RabbitMQ和Spring的整合

## 重要的类

RabbitAdmin可以很好的处理交换机和队列，修改，删除并且可以注解到spring里

RabbitTemplate类负责收发消息

## RabbitAdmin管理交换机/队列/绑定

### 创建SpringBoot工程

添加包依赖

<groupid>com.rabbitmq</groupid>

<artifactid>amqp-client<artifactid>

<version>3.6.5</version>

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

</dependency>

<dependency>

<groupid>com.springframework.boot<groupid>

<artifactid>spring-boot-starter-amqp<artifactid>

</dependency>

版本使用boot默认

@Configuration 表示这是个XML配置文件类

@ComponentScan({“com.siki.\*”})

@Bean表示这个类要注入到Spring容器的

ConnectionFactory 注意不再是rabbit里的了，而是spring里面的

在测试类里使用@Autowired注解直接使用bean

@Bean

public ConnectionFactory connectionFactory(){

CachingConnectionFactory connectionFactory = new CachingConnectionFactory();

connectionFactory.setAddresses("192.168.11.76:5672");

connectionFactory.setUsername("guest");

connectionFactory.setPassword("guest");

connectionFactory.setVirtualHost("/");

return connectionFactory;

}

@Bean

public RabbitAdmin rabbitAdmin(ConnectionFactory connectionFactory) {

RabbitAdmin rabbitAdmin = new RabbitAdmin(connectionFactory);

rabbitAdmin.setAutoStartup(true);

return rabbitAdmin;

}

### 使用RabbitAdmin创建交换机队列绑定

@Test

public void testRabbitAdmin() {

Exchange exc1 = new DirectExchange("directExchangeByAdmin",false,true);

Exchange exc2 = new TopicExchange("TopicExchangeByAdmin",false,true);

Exchange exc3 = new FanoutExchange("FanoutExchangeByAdmin",false,true);

rabbitAdmin.declareExchange(exc1);

rabbitAdmin.declareExchange(exc2);

rabbitAdmin.declareExchange(exc3);

Queue que = new Queue("XiaomiHuoJiaByAdmin",false,false,true);

rabbitAdmin.declareQueue(que);

// 第一种绑定方式

Binding binding = new Binding("XiaomiHuoJiaByAdmin",

DestinationType.QUEUE,"directExchangeByAdmin","routingkey.\*",null);

rabbitAdmin.declareBinding(binding);

// 第二种绑定方式

rabbitAdmin.declareBinding(

BindingBuilder.bind(new Queue("XiaoMiHuoJiaTopic",false,false,true))

.to(exc2)

.with("xiaomi.#")

.and(null)

);

}

### rabbitAdmin清空队列消息

rabbitAdmin.purgeQueue("XiaomiHuoJiaByAdmin");

### 使用Spring创建交换机和队列（不用刻意声明）

直接Bean即可，不用刻意声明交换机和队列，下面的例子实现一个交换机绑定两个队列，使用不同的规则进行匹配

@Bean

public Queue queue1() {

Queue queue = new Queue("queueByAdmin",false,false,false);

return queue;

}

@Bean

public Queue queue2() {

Queue queue = new Queue("queueByAdmin2",false,false,false);

return queue;

}

@Bean

public Exchange exchangeDirect() {

Exchange exc = new DirectExchange("directExcByAdmin",false,false,null);

return exc;

}

@Bean

public Binding bindingXiaoMi() {

Binding bind =

BindingBuilder.bind(queue1()).to(exchangeDirect ()).with("routingKey").and(null);

return bind;

}

@Bean

public Binding bindingHongMi() {

Binding bind = BindingBuilder.bind(queue2()).to(exchangeTopic()).with("HontMi.\*").and(null);

return bind;

}

## RabbitTemplate用于发送和接收消息

// 注入一个bean对象

@Bean

public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {

RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);

return rabbitTemplate;

}

// 准备一个要发送的包裹

MessageProperties prop = new MessageProperties();

prop.setContentEncoding("UTF-8");

prop.setContentType("text/plain");// 设置为文本消息

prop.getHeaders().put("收件人手机号", "15912345678");

Message message = new Message("hello1234".getBytes(),prop);

// 发送消息到指定的交换器，指定的路由键

rabbitTemplate.send("directExcByAdmin","routingKey",message);

// 发送对象消息消息

rabbitTemplate.convertAndSend("directExcByAdmin ","routingKey", "一部小米手机");

System.out.println("send rabbitmq finished");

// 消息的消费接收

Message msgRecv = rabbitTemplate.receive("queueByAdmin");

String msgBody = new String(msgRecv.getBody());

System.out.println("hello rabbitmq : " + msgBody);

### 使用SimpleMessageListenerContainer监听队列消息

需要运行在SpringBoot程序下

1. **MessageListener**

// 创建监听器容器（里面有多个监听器）

@Bean

public SimpleMessageListenerContainer msgListener(ConnectionFactory connectionFactory) {

SimpleMessageListenerContainer msgListener = new SimpleMessageListenerContainer(connectionFactory);

msgListener.setQueues(queue1(),queue2());

// 当前消费者是一个

msgListener.setConcurrentConsumers(1);

// 最多5个消费者

msgListener.setMaxConcurrentConsumers(5);

msgListener.setAcknowledgeMode(AcknowledgeMode.AUTO);

// 生成自定义的ConsumerTag，就是针对每一个消息指定一个tag标签

msgListener.setConsumerTagStrategy(new ConsumerTagStrategy() {

@Override

public String createConsumerTag(String queue) {

// TODO Auto-generated method stub

return queue + "\_" + UUID.randomUUID().toString();

}

});

// 添加一个真的监听器用来干活

msgListener.setMessageListener(new MessageListener() {

@Override

public void onMessage(Message message) {

MessageProperties prop = message.getMessageProperties();

String strBody = new String(message.getBody());

System.out.println("收到包裹内容：" + strBody);

// 收到详情单

Map<String,Object> headers = prop.getHeaders();

Iterator<Map.Entry<String,Object>> it = headers.entrySet().iterator();

while(it.hasNext()) {

Map.Entry<String,Object> oneItem = it.next();

System.out.println(oneItem.getKey() + " : " + oneItem.getValue());

}

}

});

return msgListener;

}

1. **ChannelAwareMessageListener**

msgListener.setMessageListener(new ChannelAwareMessageListener() {

// onMessage会多一个通道的参数

@Override

public void onMessage(Message message, Channel channel) throws Exception {

System.out.println(new String(message.getBody()));

channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);

}

});

### MessageListenerAdapter消息监听适配器

使用MessageListenerAdapter处理器进行消息队列监听处理，如果容器没有设置setDefaultListenerMethod，则处理器中默认的处理方法名是handleMessage，如果设置了setDefaultListenerMethod，则处理器中处理消息的方法名就是setDefaultListenerMethod方法参数设置的值。**也可以通过setQueueOrTagToMethodName方法为不同的队列设置不同的消息处理方法。**

自定义消息处理类，注意方法名和参数类型必须要一致

public class MessageDelegate {

public void handleMessage(byte[] messageBody1) {

System.out.println("收到货物 2： " + new String( messageBody1));

}

}

SimpleMessageListenerContainer msgListener = new

SimpleMessageListenerContainer(connectionFactory);

MessageListenerAdapter lisAdaper = new MessageListenerAdapter(new MessageDelegate());

msgListener.setMessageListener(lisAdaper);

// 也可以自己指定消息处理类的方法

lisAdaper.setDefaultListenerMethod("MyHandle");

## 类型转换器（实现接口MessageConverter）

listenerAdapter.setMessageConverter(new MyTextMsgConverter());

public class MyTextMsgConverter implements MessageConverter {

// java对象转换为Message对象

@Override

public Message toMessage(Object o, MessageProperties messageProperties) throws MessageConversionException {

Message msg = new Message(o.toString().getBytes(),messageProperties);

return msg;

}

// Message对象转换为java对象

@Override

public Object fromMessage(Message message) throws MessageConversionException {

String contentType = message.getMessageProperties().getContentType();

if(contentType != null && contentType.equals("text")){

return new String(message.getBody());

}

return message.getBody();

}

}

## 还可以给不同的队列不同的处理方法

指定不同的队列不同的处理方法名

Map<String,String> queueMethod = new HashMap<>();

queueMethod.put("queueByAdmin", "MyHandle1");

queueMethod.put("queueByAdmin2", "MyHandle2");

lisAdaper.setQueueOrTagToMethodName(queueMethod);

public void MyHandle1(String messageBody1) {

System.out.println("收到货物 1： " + messageBody1);

}

public void MyHandle2(String messageBody1) {

System.out.println("收到货物 2： " + messageBody1);

}

### 发送并接收到json

在pom.xml中需要添加jackson的引用，才能够实现把自定义的对象转换为json字符串

<!--json支持的依赖-->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

</dependency>

@Test

public void testSendJsonMessage() throws Exception {

Order order = new Order();

order.setId("001");

order.setName("消息订单");

order.setContent("描述信息");

ObjectMapper mapper = new ObjectMapper();

String json = mapper.writeValueAsString(order);

System.err.println("order 4 json: " + json);

MessageProperties messageProperties = new MessageProperties();

//这里注意一定要修改contentType为 application/json

messageProperties.setContentType("application/json");

Message message = new Message(json.getBytes(), messageProperties);

rabbitTemplate.send("topic001", "spring.order", message);

}

// 支持json格式的转换器

MessageListenerAdapter adapter = new MessageListenerAdapter(new MessageDelegate());

adapter.setDefaultListenerMethod("consumeMessage");

Jackson2JsonMessageConverter jackson2JsonMessageConverter = new Jackson2JsonMessageConverter();

adapter.setMessageConverter(jackson2JsonMessageConverter);

container.setMessageListener(adapter);

// 消费端代码

public void consumeMessage(Map messageBody) {

System.err.println("map方法, 消息内容:" + messageBody);

}

### 消费者端直接收到java对象

// 1.2 DefaultJackson2JavaTypeMapper & Jackson2JsonMessageConverter 支持java对象转换

MessageListenerAdapter adapter = new MessageListenerAdapter(new MessageDelegate());

adapter.setDefaultListenerMethod("consumeMessage");

Jackson2JsonMessageConverter jackson2JsonMessageConverter = new

Jackson2JsonMessageConverter();

DefaultJackson2JavaTypeMapper javaTypeMapper = new DefaultJackson2JavaTypeMapper();

jackson2JsonMessageConverter.setJavaTypeMapper(javaTypeMapper);

adapter.setMessageConverter(jackson2JsonMessageConverter);

container.setMessageListener(adapter);

public void consumeMessage(Order order) {

System.err.println("order对象, 消息内容, id: " + order.getId() +

", name: " + order.getName() +

", content: "+ order.getContent());

}

@Test

public void testSendJavaClassMessage() throws Exception {

Order order = new Order();

order.setId("001");

order.setName("订单消息");

order.setContent("订单描述信息");

ObjectMapper mapper = new ObjectMapper();

String json = mapper.writeValueAsString(order);

System.err.println("order 4 json: " + json);

MessageProperties messageProperties = new MessageProperties();

messageProperties.getHeaders().put("\_\_TypeId\_\_", "com.spring.entity.Order");

Message message = new Message(json.getBytes(), messageProperties);

rabbitTemplate.send("topic001", "spring.order", message);

}

### 全局转换器（多种类型）

根据内容类型自定义转换器，需要使用全局消息转换器

ContentTypeDelegatingMessageConverter

MessageListenerAdapter listenerAdapter = new MessageListenerAdapter();

ContentTypeDelegatingMessageConverter convert = new

ContentTypeDelegatingMessageConverter();

Jackson2JsonMessageConverter jsonConverter = new Jackson2JsonMessageConverter();

convert.addDelegate("text",new MyTextMsgConverter());

convert.addDelegate("json",jsonConverter);

convert.addDelegate("jpg-file",new MyJpgConverter());

convert.addDelegate("pdf-file",new MyPdfConverter());

listenerAdapter.setMessageConverter(convert);

生产者发送的代码

MessageProperties mp = new MessageProperties();

mp.setContentType("jpg-file");

mp.getHeaders().put("extName","jpg");

File file1 = new File("J:\\JavaEE 3408 RabbitMQ消息中间件（基于IDEA2018）\\资料\\dog.jpg");

try {

byte[] bytes = Files.readAllBytes(file1.toPath());

Message msg = new Message(bytes, mp);

rabbitTemplate.send("交换机spring短信-topic", "0086.13977778888", msg);

} catch (IOException e) {

e.printStackTrace();

}

## 使用SpringBoot的配置文件和注解进行收发消息

### 配置Application.properties

配置后，就可以直接使用RabbitTemplate的实力化bean了，连自己实例化bean都省了

spring.rabbitmq.addresses=localhost:5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

spring.rabbitmq.virtual-host=/

spring.rabbitmq.connection-timeout=15000

spring.rabbitmq.publisher-confirms=true

spring.rabbitmq.publisher-returns=true

spring.rabbitmq.template.mandatory=trueb

### 定义一个空配置文件自动扫描bean

@Configuration

@ComponentScan({"com.baidu.rabbitspringboot1.\*"})

public class RabbitConfig { }

### 生产端发送数据

@Autowired

RabbitTemplate rabbitTemplate;

RabbitTemplate.ConfirmCallback confirmCallback = new RabbitTemplate.ConfirmCallback() {

@Override

public void confirm(CorrelationData correlationData, boolean ack, String cause) {

System.out.println("correlationData = " + correlationData);

if(ack == false){

System.out.println("发送到MQ成功，更新数据库，标志发送成功，确认文字是" + cause);

}else{

System.out.println("发送到MQ出现了异常，原因是" + cause);

}

}

};

RabbitTemplate.ReturnCallback returnCallback = new RabbitTemplate.ReturnCallback() {

@Override

public void returnedMessage(org.springframework.amqp.core.Message message,

int replycode, String replyText, String exchange, String routingKey) {

System.out.println("收到确认的错误，消息为" + replyText);

}

};

public void send(Object msgEntity, Map<String,Object> props){

MessageHeaders mh = new MessageHeaders(props);

Message msg = MessageBuilder.createMessage(msgEntity,mh);

rabbitTemplate.setConfirmCallback(confirmCallback);

rabbitTemplate.setReturnCallback(returnCallback);

CorrelationData cd = new CorrelationData(UUID.randomUUID().toString());

rabbitTemplate.convertAndSend("交换机spring短信-direct","发送短信",msg,cd);

}

### 使用@RabbitListener消费数据

@RabbitListener是一个组合注解，可以组合其它注解一起来完成对队列的消息接收和监听，主要包括@QueueBinding、@Queue、@Exchange

import com.rabbitmq.client.Channel;

import org.springframework.messaging.Message;

@RabbitListener(

bindings = @QueueBinding(

value=@Queue(value="队列-注解",durable = "true",exclusive = "false",autoDelete = "false"),

exchange = @Exchange(value="交换机-注解", durable = "true",type = "direct"),

key = "注解-路由键"

)

)

@RabbitHandler

public void onMessage(Message msg, Channel channel) throws Exception{

System.out.println("收到的消息体：" + msg.getPayload());

Long deliveryTag = (Long) msg.getHeaders().get(AmqpHeaders.DELIVERY\_TAG);

System.out.println("包裹标签 = " + deliveryTag);

channel.basicAck(deliveryTag,false);

}

### 使用@QueueBinding创建队列、交换机和绑定

@RabbitListener(

bindings = @QueueBinding(

value = @Queue(value="队列spring短信",durable = "true",exclusive = "false",autoDelete = "false"),

exchange = @Exchange(value="交换机spring短信-direct", durable = "true",type = "direct"),

key = "发送短信"

)

)

### 发送实体类对象

要求：实体类必须实现接口 Serializable

实体类最好定义 序列号ID

public class MyUser implements Serializable {

private static final long serialVersionUID = 870292457463003416L;

private String userName;

...

}

发送代码

private static final long serialVersionUID = 870292457463003416L;

public void sendUser(MyUser user, Map<String,Object> props){

MessageHeaders mh = new MessageHeaders(props);

Message msg = MessageBuilder.createMessage(user,mh);

rabbitTemplate.setConfirmCallback(confirmCallback);

rabbitTemplate.setReturnCallback(returnCallback);

CorrelationData cd = new CorrelationData(UUID.randomUUID().toString());

rabbitTemplate.convertAndSend("exchange-properties","key.123",msg,cd);

}

public void sendUser(MyUser user){

rabbitTemplate.setConfirmCallback(confirmCallback);

rabbitTemplate.setReturnCallback(returnCallback);

CorrelationData cd = new CorrelationData(UUID.randomUUID().toString());

rabbitTemplate.convertAndSend("exchange-properties","key.abc",user,cd);

}

### 接收实体类对象

实际就是把Message拆成了两个注解

@RabbitHandler

public void onMessageMyUser(@Payload MyUser oneUser, Channel channel,

@Headers Map<String,Object> headers) throws Exception{

System.out.println("收到的消息体：" + oneUser.getUserName());

Long deliveryTag = (Long)headers.get(AmqpHeaders.DELIVERY\_TAG);

System.out.println("2包裹标签 = " + deliveryTag);

channel.basicAck(deliveryTag,false);

}