Rabbitmq开发手册

**文档修订摘要**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **日期** | **版本号** | **修订章节** | **描述** | **作者** | **评审者** | **评审日期** |
| 2017-8-30 | V0.1 |  | 初稿 | 何丹丹 |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

## 基本概念

### AMQP

Advanced Message Queuing Protocol,高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

### ConnectionFactory、Connection、Channel

ConnectionFactory、Connection、Channel都是RabbitMQ对外提供的API中最基本的对象。

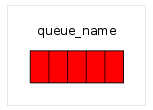
Connection是RabbitMQ的socket链接，它封装了socket协议相关部分逻辑。

ConnnectionFactory为Connection的制造工厂。

Channel是我们与RabbitMQ打交道的最重要的一个接口，我们大部分的业务操作是在Channel这个接口中完成，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息。

### Queue

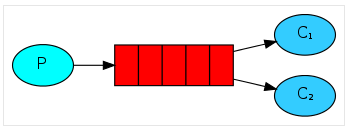
Queue（队列）是RabbitMQ的内部对象，用于存储消息，用下图表示。



RabbitMQ中的消息都只能存储在Queue中，生产者（下图中P）生产消息并最终投递到Queue中，消费者（下图中C）可以从Queue中获取消息并消费。

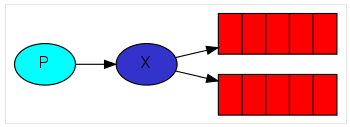


多个消费者可以订阅同一个Queue，这是Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。



### Exchange

在上一节我们看到生产这将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange(交换器，下图中X)，由Exchange将消息路由都一个或多个Queue中（或者丢弃）。



Exchange是按照什么逻辑将消息路由到Queue的？这个将在Binding一节介绍。

RabbitMQ中的Exchange有四种类型，不同的类型有着不同的路由策略，这将在Exchange Types一节介绍。

### Routing key

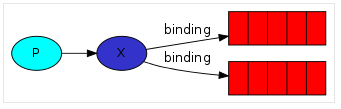
生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。

在Exchange Type与binding key固定的情况下，我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。

RabbitMQ为routing key设定的长度限制为255 bytes。

### Binding

RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。



### Binding key

在绑定(Binding)Exchange与Queue的同时，一般会指定一个binding key；消费者将消息发送给Exchange时，一般会指定一个routing key；当binding key与routing key相匹配时，消息将会被路由到对应的Queue中。

在绑定多个Queue到同一个Exchange的时候，这些Binding允许使用相同的binding key。

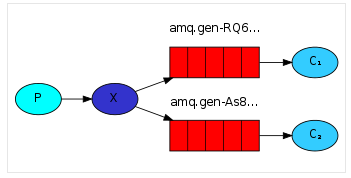
Binding key并不是在所有情况下都生效，它依赖于Exchange Type，比如fanout类型的Exchange就会无视binding key,而是将消息路由到所有绑定到该Exchange的Queue。

### Exchange Types

RabbitMQ常用的Exchange Type有fanout、direct、topic、headers这四种。

#### fanout

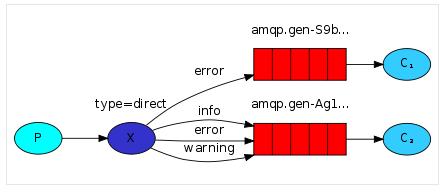
fanout类型的Exchange路由规则非常简单，它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。



上图中，生产者(P)发送到Exchange(X)的所有消息都会路由到图中的两个Queue，并最终被两个消费者(C1与C2)消费。

#### direct

direct类型的Exchange路由规则也很简单，它会把消息路由到那些binding key与routing key完全匹配的Queue中。

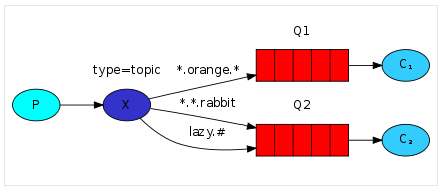


以上图的配置为例，我们以routingKey=”error”发送消息到Exchange，则消息会路由到Queue1（amqp.gen-S9b…，这是由RabbitMQ自动生成的Queue名称）和Queue2（amqp.gen-Agl…）；如果我们以routingKey=”info”或routingKey=”warning”来发送消息，则消息只会路由到Queue2。如果我们以其他routingKey发送消息，则消息不会路由到这两个Queue中。

#### topic

前面讲到direct类型的Exchange路由规则是完全匹配的binding key与routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchange相似，也是将消息路由到binding key与routing key相匹配的Queue中，但这里的匹配规则有些不同，它约定：

* routing key为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”
* binding key与routing key一样也是句点号“. ”分隔的字符串
* binding key中可以存在两种特殊字符“\*”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）



以上图中的配置为例，routingKey=”quick.orange.rabbit”的消息会同时路由到Q1与Q2，routingKey=”lazy.orange.fox”的消息会路由到Q1与Q2，routingKey=”lazy.brown.fox”的消息会路由到Q2，routingKey=”lazy.pink.rabbit”的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）；routingKey=”quick.brown.fox”、routingKey=”orange”、routingKey=”quick.orange.male.rabbit”的消息将会被丢弃，因为它们没有匹配任何bindingKey。

#### headers

headers类型的Exchange不依赖于routing key与binding key的匹配规则来路由消息，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Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。

### Message acknowledgment（消息回执）

在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机的情况，这种情况下就可能会导致消息丢失，为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，除非它的RabbitMQ连接断开。

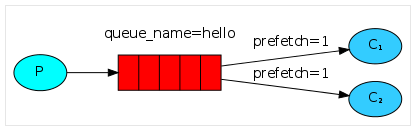
这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的Bug---Queue中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑。

### Message durability（消息持久化）

如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化，这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。

### Prefetch count

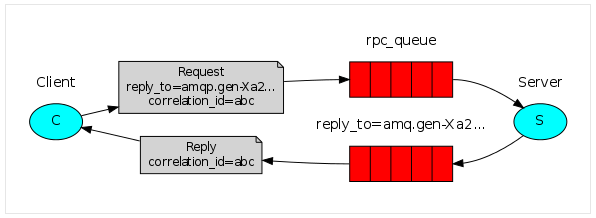
前面我们讲到如果有多个消费者同时订阅同一个Queue中的消息，Queue中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完工作并一直空闲的情况，我们可以通过设置prefetchCount来限制Queue每次发送给每个消费者的消息数，比如我们设置prefectchCount=1,则Queue每次给每个消费者发送一条消息；消费者处理完这条消息后Queue会再给该消费者发送一条消息。



### RPC

MQ本身是基于异步的消息处理，前面的示例中所有的生产者(P)将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败。

但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC。在RabbitMQ中也支持RPC。



RabbitMQ中实现RPC的机制是：

客户端发送请求（消息）时，在消息的属性（MessageProperties，在AMQP协议中定义了14中properties，这些属性会随着消息一起发送）中设置两个值replyTo（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和correlationId（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）

* 服务器端收到消息并处理
* 服务器端处理完消息后，将生成一条应答消息到replyTo指定的Queue，同时带上correlationId属性
* 客户端之前已订阅replyTo指定的Queue，从中收到服务器的应答消息后，根据其中的correlationId属性分析哪条请求被执行了，根据执行结果进行后续业务处理

## MQ组件使用指南

### 配置

目前组件仅支持rabbitmq,以rabbitmq为例配置：

#### 引入jar包

<dependency>  
 <groupId>com.rabbitmq</groupId>  
 <artifactId>amqp-client</artifactId>  
 <version>4.2.0</version>  
 </dependency>

<dependency>  
 <groupId>org.springframework.amqp</groupId>  
 <artifactId>spring-rabbit</artifactId>  
 <version>2.0.0.RELEASE</version>  
</dependency>

<dependency>  
 <groupId>com.aii.ipaas</groupId>  
 <artifactId>ipaas-mq</artifactId>  
 <version>1.0.0-SNAPSHOT</version>  
 </dependency>

#### 配置文件示例

core-context-sender.xml --- 发送端配置文件示例

core-context-consumer.xml --- 消费端配置文件示例

#### Rabbitmq连接服务配置项

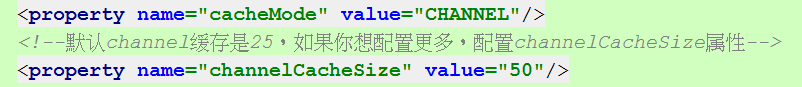
Rabbitmq连接服务用于管理RabbitMq的连接工厂和所有的Queue、Exchange、Binding

##### 构建rabbitmq连接工厂---非集群

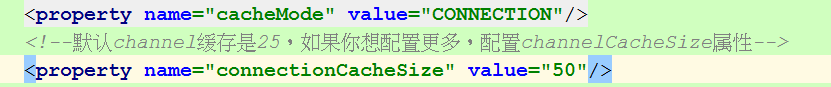
*<!--基于RabbitMQ连接工厂对象构建spring-rabbit的连接工厂对象Wrapper-->*<**bean id="mqConnectionFactory" class="org.springframework.amqp.rabbit.connection.CachingConnectionFactory"**>  
 *<!--ip-->* <**property name="host" value="10.1.239.44"**/>  
 *<!--port-->* <**property name="port" value="5672"**/>  
 *<!--username-->* <**property name="username" value="admin"**/>  
 *<!--password-->* <**property name="password" value="admin123"**/>  
 *<!--限制连接的总数，当达到限制值时将channelCheckoutTimeout来等待空闲连接，如果超时了，将抛出AmqpTimeoutException-->* <**property name="connectionLimit" value="5"**/>  
 <**property name="channelCheckoutTimeout" value="3000"**/>  
 *<!--默认channel缓存是25，如果你想配置更多，配置channelCacheSize属性-->* <**property name="channelCacheSize" value="50"**/>  
 *<!--确认消息-->* <**property name="publisherConfirms" value="true"**/>  
 *<!--返回消息-->* <**property name="publisherReturns" value="true"**/>  
</**bean**>

cacheMode包括CONNECTION、CHANNEL

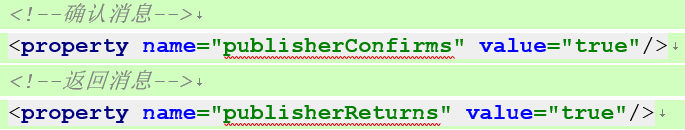
默认cacheMode为CHANNEL，缓存大小属性为channelCacheSize



如果cacheMode为CONNECTION，缓存大小属性为connectionCacheSize.



发布者确认和返回可以通过设置publisherConfirms和publisherReturns来设置



发布者确认与返回：

对于返回消息，模版的mandatory属性必须设置为true,此功能需要将CachingConnectionFactory的publisherReturns属性设置为true.返回是通过注册在RabbitTemplate.ReturnCallback(通过调用setReturnCallback(ReturnCallback callback))来返回给客户端的。回调必须实现下面的方法：

void returnedMessage(Message message,int replyCode,String replyText,String exchange,String routingKey);

每个RabbitTemplate只支持一个ReturnCallback.

对于发布者确认，模版需要将CacheConnectionFactory中的publisherConfirms属性设置为true.确认是通过注册在RabbitTemplate.ConfirmCallback(通过调用setConfirmCallback callback)发送给client的。回调必须实现下面的方法：

void confirm(CorrelationData correlationData,Boolean ack,String cause);

correlationData对象是在发送原始消息的时候，由client提供的，ack为true是表示确认，为false时，表示不确认(nack)。对于nack,cause可能会包含nack的原因。

RabbitTemplate只支持一个ConfirmCallback.

当rabbit模板完成发送操作时，会关闭通道; 这可以排除当连接工厂缓存满时(缓存中还有空间，通道没有物理关闭，返回／确认正常处理)确认和返回的接待问题.

当缓存满了的时候, 框架会延迟５秒来关闭,以为接收确认／返回消息留有时间．当使用确认时，通道会在收到最后一个确认时关闭.

当使用返回时，通道会保持５秒的打开状态.一般建议将连接工厂的channelCacheSize 设为足够大，这样发布消息的通道就会返回到缓存中，而不是被关闭.

你可以使用RabbitMQ管理插件来监控通道的使用情况;如果你看到通道打开／关闭的非常迅速，那么你必须考虑加大缓存，从而减少服务器的开销．

##### 构建rabbitmq连接工厂---集群

集群环境中，使用addresses属性

*<!--基于RabbitMQ连接工厂对象构建spring-rabbit的连接工厂对象Wrapper-->*<**bean id="mqConnectionFactory" class="org.springframework.amqp.rabbit.connection.CachingConnectionFactory"**>   
 *<!--集群环境-->* <**property name="addresses" value="10.1.239.44:5672,10.11.17.125:5672"**/>  
 *<!--username-->* <**property name="username" value="admin"**/>  
 *<!--password-->* <**property name="password" value="admin123"**/>  
 *<!--限制连接的总数，当达到限制值时将channelCheckoutTimeout来等待空闲连接，如果超时了，将抛出AmqpTimeoutException-->* <**property name="connectionLimit" value="20"**/>  
 <**property name="channelCheckoutTimeout" value="20000"**/>  
 *<!--默认channel缓存是25，如果你想配置更多，配置channelCacheSize属性-->* <**property name="channelCacheSize" value="100"**/>  
 *<!--确认消息-->* <**property name="publisherConfirms" value="true"**/>  
 *<!--返回消息-->* <**property name="publisherReturns" value="true"**/>  
</**bean**>

##### 构建RabbitAdmin对象

它负责创建Queue、Exchange、Bind对象

*<!--构建RabbitAdmin对象，它负责创建Queue、Exchange、Bind对象-->*<**bean id="rabbitAdmin" class="org.springframework.amqp.rabbit.core.RabbitAdmin"**>  
 <**constructor-arg name="connectionFactory" ref="mqConnectionFactory"**/>  
 <**property name="autoStartup" value="true"**></**property**>  
</**bean**>

##### 构建消息类型转换对象

<**bean id="messageConverter" class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter"**/>

##### 构建Rabbit Template对象

用于发送RabbitMQ消息

*<!--构建Rabbit Template对象，用于发送RabbitMQ消息-->*<**bean id="rabbitTemplate" class="org.springframework.amqp.rabbit.core.RabbitTemplate"**>  
 *<!--消息工厂-->* <**constructor-arg name="connectionFactory" ref="mqConnectionFactory"**/>  
 *<!--超时-->* <**property name="receiveTimeout" value="3000"**/>  
 *<!--mandatory-->* <**property name="mandatory" value="true"**/>  
 *<!--消息类型转换器-->* <**property name="messageConverter" ref="messageConverter"**></**property**>  
</**bean**>

##### 构建Exchange

Exchange包括Direct、Fanout、Topic等类型

###### Direct类型

*<!--定义AMQP Direct 类型Exchange-->*<**bean id="directMsgExchange" class="org.springframework.amqp.core.DirectExchange"**>  
 *<!--Exchange 命名-->* <**constructor-arg name="name" value="directMsgExchange"**/>  
 *<!--是否持久化-->* <**constructor-arg name="durable" value="true"**></**constructor-arg**>  
 *<!--是否自动删除-->* <**constructor-arg name="autoDelete" value="false"**/>  
 *<!--定义AMQP Queue创建所需的RabbitAdmin对象-->* <**property name="adminsThatShouldDeclare" ref="rabbitAdmin"**/>  
 *<!--判断是否需要在连接RabbitMQ后创建Exchange-->* <**property name="shouldDeclare" value="true"**></**property**>  
</**bean**>

###### Fanout类型

Fanout类型Exchange适用广播类型的消息

*<!--定义fanout类型-->*<**bean id="fanoutMsgExchange" class="org.springframework.amqp.core.FanoutExchange"**>  
 <**constructor-arg name="name" value="fanoutMsgExchange"**/>  
 <**constructor-arg name="autoDelete" value="true"**/>  
 <**constructor-arg name="durable" value="false"**/>  
</**bean**>

###### Topic类型

<**bean id="topicMsgExchange" class="org.springframework.amqp.core.TopicExchange"**>  
 <**constructor-arg name="name" value="topicMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="durable" value="true"**></**constructor-arg**>  
 <**constructor-arg name="autoDelete" value="true"**></**constructor-arg**>  
</**bean**>

##### 构建Queue

*<!--定义支付类AMQP Queue-->*<**bean id="clientPayMsgQueue" class="org.springframework.amqp.core.Queue"**>  
 *<!--Queue 命名-->* <**constructor-arg name="name" value="clientPayMsgQueue"**/>  
 *<!--是否自动删除-->* <**constructor-arg name="autoDelete" value="false"**/>  
 *<!--是否持久化-->* <**constructor-arg name="durable" value="true"**/>  
 *<!--是否是排他性队列-->* <**constructor-arg name="exclusive" value="false"**/>  
 *<!--定义AMQP Queue创建所需的RabbitAdmin对象-->* <**property name="adminsThatShouldDeclare" ref="rabbitAdmin"**/>  
 *<!-- 判断是否需要在连接RabbitMQ后创建Queue-->* <**property name="shouldDeclare" value="true"**></**property**>  
</**bean**>

##### 构建Binding

###### 绑定Direct Exchange与Queue

绑定Queue与Exchange，routingKey必须配置：

<**util:map id="emptyMap" map-class="java.util.HashMap"**/>  
  
*<!--创建Exchange和Queue之间的Bind-->*<**bean id="clientPayMsgBind" class="org.springframework.amqp.core.Binding"**>  
 *<!--目的Queue-->* <**constructor-arg name="destination" value="clientPayMsgQueue"**></**constructor-arg**>  
 *<!--终端类型为Queue-->* <**constructor-arg name="destinationType" value="QUEUE"**/>  
 *<!--exchange名称-->* <**constructor-arg name="exchange" value="clientMsgExchange"**/>  
 *<!--routingKey-->* <**constructor-arg name="routingKey" value="payMsg"**/>  
 *<!--参数-->* <**constructor-arg name="arguments" ref="emptyMap"**/>  
</**bean**>

###### 绑定Fanout Exchange与Queue

可以不设定RoutingKey,每一个与Fanout Exchange绑定的Queue都可以发送到Exchange上收到的任何消息。

可以使用fanoutBind发送消息，fanoutBind1与fanoutBind2绑定的queue都可以收到消息。

*<!--绑定FanoutExchange-->*<**bean id="fanoutBind" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="fanoutMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="EXCHANGE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="fanoutMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value=""**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>  
  
*<!--绑定FanoutExchange与Fanout Queue1-->*<**bean id="fanoutBind1" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="fanoutQueue1"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="fanoutMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value=""**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>  
*<!--绑定FanoutExchange与Fanout Queue2-->*<**bean id="fanoutBind2" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="fanoutQueue2"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="fanoutMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value=""**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>

###### 绑定Topic Exchange与Queue

Topic与Direct类似，只是匹配上支持了“模式”，在“点分”的routing\_key形式中，可以使用两个通配符：

\*表示一个词

#表示零个或多个词

*<!--构建Binding-->*<**bean id="topicBind" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="topicQueue1"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="topicMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value="say.hello.\*"**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>  
  
<**bean id="topicBind2" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="topicQueue2"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="topicMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value="say.\*"**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>  
*<!-- 定义Topic类型 end -->*

#### Rabbitmq Conf对象配置项

Rabbitmq Conf主要用于管理mq客户端所需的的所有配置项，Direct、Fanout类型的可以直接设置Binding作为Conf对象的参数， Topic类型建议直接设置exchange、queue和routingKey，因为如果直接使用Binding对象，按routingKey匹配的优势就没有了。

*<!---Direct 类型-->* <**bean id="payMsgConf" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="binding" ref="clientPayMsgBind"**></**property**>  
 </**bean**>  
 *<!---Fanout 类型-->* <**bean id="fanOutConf" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="binding" ref="fanoutBind"**></**property**>  
 </**bean**>  
 *<!---topic 类型-->  
 <!--接收-->* <**bean id="topicConf1" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="queue" value="topicQueue1"**></**property**>  
 <**property name="exchange" value="topicMsgExchange"**></**property**>  
 </**bean**>  
 *<!--发送-->* <**bean id="topicConf" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="exchange" value="topicMsgExchange"**></**property**>  
 <**property name="routingKey" value="say.haha"**></**property**>  
 </**bean**>  
</**beans**>

#### 发送端配置

##### 发送工厂配置

<**bean id="senderFactory" class="com.aii.ipaas.mq.base.MqSenderFactoryBean"**></**bean**>

##### 发送客户端配置

消息发送客户端，目前仅支持rabbitmq

*<!--消息发送端-->*<**bean id="sender" factory-bean="senderFactory" factory-method="getRabbitMqClient"**>  
 <**constructor-arg name="conf" ref="payMsgConf"**></**constructor-arg**>  
</**bean**>

##### 代码示例

**public void** send() **throws** Exception {  
 *// 获取sender客户端* MessageSender sender = (MessageSender) *applicationContext*.getBean(**"sender"**);  
  
 **for**(**int** i = 0;i < 100;i++){  
 *// 发送对象* OrderInfo orderInfo = **new** OrderInfo();  
 orderInfo.setId(i);  
 orderInfo.setOfferName(**"U7"**);  
 *// 发送消息* sender.send(orderInfo);  
 }  
  
}

#### 消费端配置

消费端支持两种方式：消费池、监听容器

##### 消费池

特点：CPU占用率较低，消费速度相对于监听容器慢

消费池：**com.aii.ipaas.mq.rabbitmq.consumer.ConsumerPoolManage**

消费池支持自定义任务数量、消费池核心线程数、最大线程数等。

*<!--消息消费端-->*<**bean id="consumer" factory-bean="consumerFactory" factory-method="getRabbitMqClient"** >  
 <**constructor-arg name="conf" ref="directMsgConf"**></**constructor-arg**>  
</**bean**>  
*<!--消息处理类-->*<**bean id="msgProcess" class="test.com.aii.ipaas.mq.consumer.processer.UserMessageProcess"**>

</**bean**>

*<!--消费池-->*<**bean id="consumerPoolManage" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerPoolManage"**>  
 <**property name="messageProcess" ref="msgProcess"**/>  
 <**property name="messageConsumer" ref="consumer"**/>  
 <**property name="taskCount" value="1000"**/>  
 <**property name="corePoolSize" value="100"**/>  
 <**property name="maxPoolSize" value="200"**/>  
 <**property name="capacity" value="20000"**/>

<**property name="intervalMils" value="1000"**/>  
</**bean**>

主要参数：

**messageProcess --- 消息处理类**

**messageConsumer --- 消费端**

**taskCount --- 任务数量**

**corePoolSize --- 核心线程数**

**maxPoolSize --- 最大线程数**

**capacity --- 线程池缓冲区大小**

**intervalMils --- 消费间隔时间（重要）**

注意：尽量避免多条消息中处理同一条数据，因为消费池的多线程不能保证数据提交的顺序。如果需要处理同一条数据，需要在自定义的process类中做处理，方案有两种：一是将唯一id传入消息中，消费端处理类中将该id缓存起来，每次处理时，判断下缓存中是否有该id，如果有，不处理，如果没有，处理。处理完成后再将该id移出缓存。方案二：修改表结构，增加锁状态字段，处理之前先将该行锁状态设置为lock，提交后再解锁。方法入口增加锁判断，如果该行数据已被锁，跳过。

##### 监听容器

特点：消费速度快，但CPU占用率很高

###### SimpleMessageListenerContainer

示例：

*<!--&lt;!&ndash;使用系统自带的监听类&ndash;&gt;-->*<**bean id="queueListen" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerListener"**>  
 <**property name="messageProcess" ref="msgProcess"**></**property**>  
</**bean**>  
*<!--&lt;!&ndash;配置监听容器&ndash;&gt;-->*<**bean id="listenContainer" class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer"**>  
 <**property name="queueNames" value="directMsgQueue"**></**property**>  
 <**property name="connectionFactory" ref="mqConnectionFactory"**></**property**>  
 <**property name="channelAwareMessageListener" ref="queueListen"**></**property**>  
 <**property name="concurrentConsumers" value="20"**/>  
 <**property name="maxConcurrentConsumers" value="200"**/>  
 <**property name="autoStartup" value="true"**/>  
 <**property name="acknowledgeMode" value="MANUAL"**/>  
</**bean**>

主要参数：

**queueNames – 监听queue名称，支持监听多个queue，多个queue名称用逗号隔开。**

**connectionFactory – 连接工厂**

**channelAwareMessageListener – 监听器（可以使用系统自带的，也可以自定义）**

**concurrentConsumers – 当前容器中最少线程数量，依据实际消息接收并发量去设置这个值**

**maxConcurrentConsumers – 当前容器中最大线程数量，依据实际消息接收并发量去设置这个值**

**autoStartup – 是否自动启动**

**acknowledgeMode – 回复ACK，AUTO表示自动回复，MANUAL表示手动回复，NONE表示不回复**

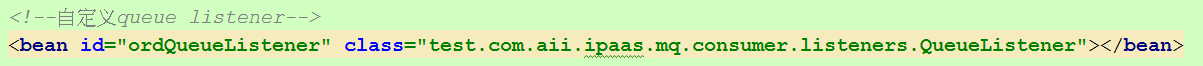
**需要注意的是，listenerContainer虽然可以同时监控多个queue，但一个container只有一个监听，多个queue共用一个process处理类。如果需要配置多个process处理类，可以配置多个listenerContainer。**

多Container配置示例：

*<!--消息处理类-->*<**bean id="msgProcess" class="com.aii.ipaas.sla.mq.SlaMessageProcess"**></**bean**>  
<**bean id="rpcMsgProcess" class="com.aii.ipaas.sla.mq.SlaRpcMessageProcess"**></**bean**>  
<**bean id="queueListen" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerListener"**>  
 <**property name="messageProcess" ref="msgProcess"**></**property**>  
</**bean**>  
<**bean id="rpcListen" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerListener"**>  
 <**property name="messageProcess" ref="rpcMsgProcess"**></**property**>  
</**bean**>  
*<!--配置监听容器-->*<**bean id="listenContainer" class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer"**>  
 <**property name="queueNames" value="directMsgQueue"**></**property**>  
 <**property name="connectionFactory" ref="mqConnectionFactory"**></**property**>  
 <**property name="channelAwareMessageListener" ref="queueListen"**></**property**>  
 <**property name="concurrentConsumers" value="20"**/>  
 <**property name="maxConcurrentConsumers" value="200"**/>  
 <**property name="autoStartup" value="true"**/>  
 <**property name="acknowledgeMode" value="MANUAL"**/>  
 <**property name="concurrency" value="5"**/>  
</**bean**>  
<**bean id="rpcContainer" class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer"**>  
 <**property name="queueNames" value="rpc\_queue"**></**property**>  
 <**property name="connectionFactory" ref="mqConnectionFactory"**></**property**>  
 <**property name="channelAwareMessageListener" ref="rpcListen"**></**property**>  
 <**property name="concurrentConsumers" value="20"**/>  
 <**property name="maxConcurrentConsumers" value="200"**/>  
 <**property name="autoStartup" value="true"**/>  
 <**property name="acknowledgeMode" value="MANUAL"**/>  
 <**property name="concurrency" value="5"**/>  
</**bean**>

你也可以自定义监听器类，实现接口ChannelAwareMessageListner即可

自定义的监听类需要注意的是，处理完消息后要手动回复Ack,若不自动回复，消息不被服务端删除，继续发送给下一个消费者



自定义监听类实例：

QueueListener.class代码如下：



### 场景案例

#### Direct类型应用

场景：构建两个Direct类型Queue，一个用于存放支付(Pay)信息，一个用于存放订购(Ord)信息，它们使用同一个Exchange. 发送消息时根据不同的routingKey，发送到相应的Queue。

1. 定义Direct类型Exchange

*<!--定义AMQP Direct 类型Exchange-->*<**bean id="directMsgExchange" class="org.springframework.amqp.core.DirectExchange"**>  
 *<!--Exchange 命名-->* <**constructor-arg name="name" value="directMsgExchange"**/>  
 *<!--是否持久化-->* <**constructor-arg name="durable" value="true"**></**constructor-arg**>  
 *<!--是否自动删除-->* <**constructor-arg name="autoDelete" value="false"**/>  
 *<!--定义AMQP Queue创建所需的RabbitAdmin对象-->* <**property name="adminsThatShouldDeclare" ref="rabbitAdmin"**/>  
 *<!--判断是否需要在连接RabbitMQ后创建Exchange-->* <**property name="shouldDeclare" value="true"**></**property**>  
</**bean**>

1. 定义支付类Queue、订单类Queue

<**bean id="clientPayMsgQueue" class="org.springframework.amqp.core.Queue"**>  
 *<!--Queue 命名-->* <**constructor-arg name="name" value="clientPayMsgQueue"**/>  
 *<!--是否自动删除-->* <**constructor-arg name="autoDelete" value="false"**/>  
 *<!--是否持久化-->* <**constructor-arg name="durable" value="true"**/>  
 *<!--是否是排他性队列-->* <**constructor-arg name="exclusive" value="false"**/>  
 *<!--定义AMQP Queue创建所需的RabbitAdmin对象-->* <**property name="adminsThatShouldDeclare" ref="rabbitAdmin"**/>  
 *<!-- 判断是否需要在连接RabbitMQ后创建Queue-->* <**property name="shouldDeclare" value="true"**></**property**>  
</**bean**>

*<!--定义订单类Queue-->*<**bean id="clientOrdMsgQueue" class="org.springframework.amqp.core.Queue"**>  
 *<!--Queue 命名-->* <**constructor-arg name="name" value="clientOrdMsgQueue"**/>  
 *<!--是否自动删除-->* <**constructor-arg name="autoDelete" value="false"**/>  
 *<!--是否持久化-->* <**constructor-arg name="durable" value="true"**/>  
 *<!--是否是排他性队列-->* <**constructor-arg name="exclusive" value="false"**/>  
 *<!--定义AMQP Queue创建所需的RabbitAdmin对象-->* <**property name="adminsThatShouldDeclare" ref="rabbitAdmin"**/>  
 *<!--判断是否需要在连接RabbitMQ后创建Queue-->* <**property name="shouldDeclare" value="true"**/>  
</**bean**>

1. 绑定Exchange与Queue

<**util:map id="emptyMap" map-class="java.util.HashMap"**/>  
*<!--创建Exchange和支付类Queue之间的Bind-->*<**bean id="clientPayMsgBind" class="org.springframework.amqp.core.Binding"**>  
 *<!--目的Queue-->* <**constructor-arg name="destination" value="clientPayMsgQueue"**></**constructor-arg**>  
 *<!--终端类型为Queue-->* <**constructor-arg name="destinationType" value="QUEUE"**/>  
 *<!--exchange名称-->* <**constructor-arg name="exchange" value="clientMsgExchange"**/>  
 *<!--routingKey-->* <**constructor-arg name="routingKey" value="payMsg"**/>  
 *<!--参数-->* <**constructor-arg name="arguments" ref="emptyMap"**/>  
</**bean**>

*<!--创建Exchange和订单类Queue之间的Bind-->*<**bean id="clientOrdMsgBind" class="org.springframework.amqp.core.Binding"**>  
 *<!--目的Queue-->* <**constructor-arg name="destination" value="clientOrdMsgQueue"**></**constructor-arg**>  
 *<!--终端类型为Queue-->* <**constructor-arg name="destinationType" value="QUEUE"**/>  
 *<!--exchange名称-->* <**constructor-arg name="exchange" value="clientMsgExchange"**/>  
 *<!--routingKey-->* <**constructor-arg name="routingKey" value="ordMsg"**/>  
 *<!--参数-->* <**constructor-arg name="arguments" ref="emptyMap"**/>  
</**bean**>

1. 配置mqConf

*<!---Direct 类型-->*<**bean id="payMsgConf" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="binding" ref="clientPayMsgBind"**></**property**>  
</**bean**>

1. 发送、消费端配置

*<!--发送配置begin-->*

*<!--发送工厂类-->*<**bean id="senderFactory" class="com.aii.ipaas.mq.base.MqSenderFactoryBean"**></**bean**>

*<!--消息发送端-->*<**bean id="sender" factory-bean="senderFactory" factory-method="getRabbitMqClient"**>  
 <**constructor-arg name="conf" ref="payMsgConf"**></**constructor-arg**>  
</**bean**>

*<!--发送配置end-->*

*<!—消费端begin-->*

*<!--消息消费端-->*<**bean id="consumer" factory-bean="consumerFactory" factory-method="getRabbitMqClient"** >  
 <**constructor-arg name="conf" ref="directMsgConf"**></**constructor-arg**>  
</**bean**>  
*<!--消息处理类-->*<**bean id="msgProcess" class="test.com.aii.ipaas.mq.consumer.processer.UserMessageProcess"**>

</**bean**>

*<!--消费池方式begin-->*<**bean id="consumerPoolManage" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerPoolManage"**>  
 <**property name="messageProcess" ref="msgProcess"**/>  
 <**property name="messageConsumer" ref="consumer"**/>  
 <**property name="taskCount" value="1000"**/>  
 <**property name="corePoolSize" value="100"**/>  
 <**property name="maxPoolSize" value="200"**/>  
 <**property name="capacity" value="20000"**/>  
</**bean**>

*<!--消费池方式end-->*

*<!--消费监听容器方式begin-->*

*<!--&lt;!&ndash;使用系统自带的监听类&ndash;&gt;-->*<**bean id="queueListen" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerListener"**>  
 <**property name="messageProcess" ref="msgProcess"**></**property**>  
</**bean**>  
*<!--&lt;!&ndash;配置监听容器&ndash;&gt;-->*<**bean id="listenContainer" class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer"**>  
 <**property name="queueNames" value="directMsgQueue"**></**property**>  
 <**property name="connectionFactory" ref="mqConnectionFactory"**></**property**>  
 <**property name="channelAwareMessageListener" ref="queueListen"**></**property**>  
 <**property name="concurrentConsumers" value="20"**/>  
 <**property name="maxConcurrentConsumers" value="200"**/>  
 <**property name="autoStartup" value="true"**/>  
 <**property name="acknowledgeMode" value="MANUAL"**/>  
</**bean**>

*<!--消费监听容器方式end-->*

*<!—消费端end-->*

1. 发送消息

代码示例：

**public void** send() **throws** Exception {  
 *// 获取sender客户端* MessageSender sender = (MessageSender) *applicationContext*.getBean(**"sender"**);  
 **for**(**int** i = 0;i < 100;i++){  
 *// 发送对象* OrderInfo orderInfo = **new** OrderInfo();  
 orderInfo.setId(i);  
 orderInfo.setOfferName(**"U7"**);  
 *// 发送消息* sender.send(orderInfo);  
 }  
}

1. 消费消息

代码示例：

@Test  
**public void** consumer() **throws** Exception {  
 *// 获取消费池对象* ConsumerPoolManage consumerPool = (ConsumerPoolManage) *applicationContext*.getBean(**"consumerPoolManage"**);  
 *// 启动消费池* consumerPool.start();  
 **while** (**true**) {  
 **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {System.***out***.println(**"interrupt "** + e);  
 }  
 }  
}

监听容器配置：

启动spring配置文件即可启动监听容器。

**private static** ApplicationContext *applicationContext* = **new** FileSystemXmlApplicationContext(**"classpath:core-context-consumer.xml"**);

#### Fanout类型应用

场景：定义一个Fanout类型的Exchange，往Exchange发送的任何消息都会广播到与该Exchange绑定的Queue上。

1. 构建Fanout类型的Exchange

*<!--定义fanout类型-->*<**bean id="fanoutMsgExchange" class="org.springframework.amqp.core.FanoutExchange"**>  
 <**constructor-arg name="name" value="fanoutMsgExchange"**/>  
 <**constructor-arg name="autoDelete" value="true"**/>  
 <**constructor-arg name="durable" value="false"**/>  
</**bean**>

1. 构建两个Queue，用于绑定到fanoutMsgExchange上

*<!--定义Fanout Queue1-->*<**bean id="fanoutQueue1" class="org.springframework.amqp.core.Queue"**>  
 <**constructor-arg name="name" value="fanoutQueue1"**></**constructor-arg**>  
 <**constructor-arg name="durable" value="false"**></**constructor-arg**>  
</**bean**>  
*<!--定义Fanout Queue2-->*<**bean id="fanoutQueue2" class="org.springframework.amqp.core.Queue"**>  
 <**constructor-arg name="name" value="fanoutQueue2"**></**constructor-arg**>  
 <**constructor-arg name="durable" value="false"**></**constructor-arg**>  
</**bean**>

1. 绑定fanoutQueue1、fanoutQueue2到fanoutMsgExchange上

*<!--绑定FanoutExchange与Fanout Queue1-->*<**bean id="fanoutBind1" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="fanoutQueue1"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="fanoutMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value=""**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>  
*<!--绑定FanoutExchange与Fanout Queue2-->*<**bean id="fanoutBind2" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="fanoutQueue2"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="fanoutMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value=""**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>

1. 配置mqConf

*<!--- Fanout 类型-->*<**bean id="fanOutConf" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="binding" ref="fanoutBind"**></**property**>  
</**bean**>

1. 发送、消费端配置

*<!—发送端配置begin-->*

*<!--发送工厂类-->*<**bean id="senderFactory" class="com.aii.ipaas.mq.base.MqSenderFactoryBean"**></**bean**>  
<**bean id="fanoutSender" factory-bean="senderFactory" factory-method="getRabbitMqClient"**>  
 <**constructor-arg name="conf" ref="fanOutConf"**></**constructor-arg**>  
</**bean**>

*<!—发送端配置end-->*

*<!--消费端配置begin-->*

*<!--消费工厂类-->*<**bean id="consumerFactory" class="com.aii.ipaas.mq.base.MqConsumerFactoryBean"**></**bean**>  
*<!--消息消费端-->*<**bean id="consumer" factory-bean="consumerFactory" factory-method="getRabbitMqClient"** >  
 <**constructor-arg name="conf" ref="fanOutConf"**></**constructor-arg**>  
</**bean**>  
*<!--消息处理类-->*<**bean id="msgProcess" class="test.com.aii.ipaas.mq.consumer.processer.UserMessageProcess"**></**bean**>  
*<!--消费池-->*

<**bean id="consumerPoolManage" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerPoolManage"**>  
 <**property name="messageProcess" ref="msgProcess"**/>  
 <**property name="messageConsumer" ref="consumer"**/>  
 <**property name="taskCount" value="1000"**/>  
 <**property name="corePoolSize" value="100"**/>  
 <**property name="maxPoolSize" value="200"**/>  
 <**property name="capacity" value="20000"**/>  
</**bean**>

*<!--消费端配置end-->*

1. 发送消息

**public void** send() **throws** Exception {  
 *// 获取sender客户端* MessageSender sender = (MessageSender) *applicationContext*.getBean(**"fanoutSender"**);  
 **for**(**int** i = 0;i < 100;i++){  
 *// 发送对象* OrderInfo orderInfo = **new** OrderInfo();  
 orderInfo.setId(i);  
 orderInfo.setOfferName(**"U7"**);  
 *// 发送消息* sender.send(orderInfo);  
 }  
}

1. 消费消息

@Test  
**public void** consumer() **throws** Exception {  
 *// 获取消费池对象* ConsumerPoolManage consumerPool = (ConsumerPoolManage) *applicationContext*.getBean(**"consumerPoolManage"**);  
 *// 启动消费池* consumerPool.start();  
 **while** (**true**) {  
 **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {System.***out***.println(**"interrupt "** + e);  
 }  
 }  
}

#### Topic类型应用

场景：定义一个topic类型的Exchange，根据routingKey值去匹配不同的Queue

1. 构建Topic类型的Exchange

<**bean id="topicMsgExchange" class="org.springframework.amqp.core.TopicExchange"**>  
 <**constructor-arg name="name" value="topicMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="durable" value="true"**></**constructor-arg**>  
 <**constructor-arg name="autoDelete" value="true"**></**constructor-arg**>  
</**bean**>

1. 构建两个Queue

*<!--构建Queue-->*<**bean id="topicQueue1" class="org.springframework.amqp.core.Queue"**>  
 <**constructor-arg name="name" value="topicQueue1"**></**constructor-arg**>  
 <**constructor-arg name="durable" value="true"**></**constructor-arg**>  
</**bean**>  
<**bean id="topicQueue2" class="org.springframework.amqp.core.Queue"**>  
 <**constructor-arg name="name" value="topicQueue2"**></**constructor-arg**>  
 <**constructor-arg name="durable" value="true"**></**constructor-arg**>  
</**bean**>

1. 绑定Queue到Exchange上，设定不同的routingKey

*<!--构建Binding-->*<**bean id="topicBind" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="topicQueue1"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="topicMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value="say.hello.\*"**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>  
  
<**bean id="topicBind2" class="org.springframework.amqp.core.Binding"**>  
 <**constructor-arg name="destination" value="topicQueue2"**></**constructor-arg**>  
 <**constructor-arg name="destinationType" value="QUEUE"**></**constructor-arg**>  
 <**constructor-arg name="exchange" value="topicMsgExchange"**></**constructor-arg**>  
 <**constructor-arg name="routingKey" value="say.\*"**></**constructor-arg**>  
 <**constructor-arg name="arguments" ref="emptyMap"**></**constructor-arg**>  
</**bean**>

1. 配置mqConf

*<!--发送-->*<**bean id="topicConf" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="exchange" value="topicMsgExchange"**></**property**>  
 <**property name="routingKey" value="say.hi"**></**property**>  
</**bean**>

*<!--接收-->*<**bean id="topicConf2" class="com.aii.ipaas.mq.vo.Configuration"**>  
 <**property name="rabbitTemplate" ref="rabbitTemplate"**></**property**>  
 <**property name="queue" value="topicQueue2"**></**property**>  
 <**property name="exchange" value="topicMsgExchange"**></**property**>  
</**bean**>

1. 发送、消费端配置

*<!--发送端配置begin-->*

*<!--发送工厂类-->*<**bean id="senderFactory" class="com.aii.ipaas.mq.base.MqSenderFactoryBean"**></**bean**>

<**bean id="topicSender" factory-bean="senderFactory" factory-method="getRabbitMqClient"**>  
 <**constructor-arg name="conf" ref="topicConf"**></**constructor-arg**>  
</**bean**>

*<!--发送端配置end-->*

*<!--消费工厂类 begin-->*

*<!--消费工厂类-->*<**bean id="consumerFactory" class="com.aii.ipaas.mq.base.MqConsumerFactoryBean"**></**bean**>  
*<!--消息消费端-->*<**bean id="consumer" factory-bean="consumerFactory" factory-method="getRabbitMqClient"** >  
 <**constructor-arg name="conf" ref="topicConf2"**></**constructor-arg**>  
</**bean**>  
*<!--消息处理类-->*<**bean id="msgProcess" class="test.com.aii.ipaas.mq.consumer.processer.UserMessageProcess"**></**bean**>  
*<!--消费池-->*

<**bean id="consumerPoolManage" class="com.aii.ipaas.mq.rabbitmq.consumer.ConsumerPoolManage"**>  
 <**property name="messageProcess" ref="msgProcess"**/>  
 <**property name="messageConsumer" ref="consumer"**/>  
 <**property name="taskCount" value="1000"**/>  
 <**property name="corePoolSize" value="100"**/>  
 <**property name="maxPoolSize" value="200"**/>  
 <**property name="capacity" value="20000"**/>  
</**bean**>

*<!--消费端配置end-->*

1. 发送消息

**public void** send() **throws** Exception {  
 *// 获取sender客户端* MessageSender sender = (MessageSender) *applicationContext*.getBean(**"topicSender"**);  
 **for**(**int** i = 0;i < 100;i++){  
 *// 发送对象* OrderInfo orderInfo = **new** OrderInfo();  
 orderInfo.setId(i);  
 orderInfo.setOfferName(**"U7"**);  
 *// 发送消息* sender.send(orderInfo);  
 }  
}

1. 消费消息

@Test  
**public void** consumer() **throws** Exception {  
 *// 获取消费池对象* ConsumerPoolManage consumerPool = (ConsumerPoolManage) *applicationContext*.getBean(**"consumerPoolManage"**);  
 *// 启动消费池* consumerPool.start();  
 **while** (**true**) {  
 **try** {  
 Thread.*sleep*(2000);  
 } **catch** (InterruptedException e) {System.***out***.println(**"interrupt "** + e);  
 }  
 }  
}