# RabbitMQ

## 主要内容

1. RabbitMQ概述
2. RabbitMQ原理
3. 安装RabbitMQ
4. RabbitMQ账户管理
5. RabbitMQ名词解释
6. RabbitMQ的Web管理界面
7. 使用RabbitMQ多种模型
8. SpringBoot中使用RabbitMQ
9. RabbitMQ的消息监视
10. RabbitMQ的消费和签收机制
11. RabbitMQ重复消费处理
12. RabbitMQ延时消息和死信消息

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| RabbmitMQ简介 | 掌握 |
| RabbitMQ原理 | 掌握 |
| 安装RabbitMQ | 掌握 |
| RabbitMQ Web管理界面 | 掌握 |
| RabbitMQ多种模型 | 掌握 |
| 使用Spring Boot中使用RabbitMQ | 掌握 |
| RabbitMQ的签收机制及重复消费处理 | 掌握 |
| RabbitMQ中的延时消息及死信消息 | 掌握 |

# 【熟悉】RabbitMQ概述

## 生活中的案例[生产中的问题]为什么要使用MQ

 1，学生问问题的例子

 2，分布式项目中RPC的调用处理时间过长的问题

## 为什么要使用MQ

微服务架构后，**链式调用是我们在写程序时候的一般流程**，为了这完成一个整体功能会把它拆分成多个函数（或子模块）比如模块A调用模块B，模块B调用模块C，模块C调用模块D。但是大型分布式应用中，系统间的RPC交互复杂，一个功能后面要调用上百个接口并非不可能，从单机架构过渡到分布式微服务架构，这样的架构有没有问题呢？有

    根据上面的几个问题，在设置系统时可以明确要达到的目标

1，要做到系统解耦，当新的模块进来时，可以做到代码改动最小;  **能够解耦**

2，设置流程缓冲池，可以让后端系统按自身吞吐能力进行消费，不被冲垮; **能够削峰**

3，强弱依赖梳理能把非关键调用链路的操作异步化并提升整体系统的吞吐能力;**能够异步**

什么是MQ

### 定义

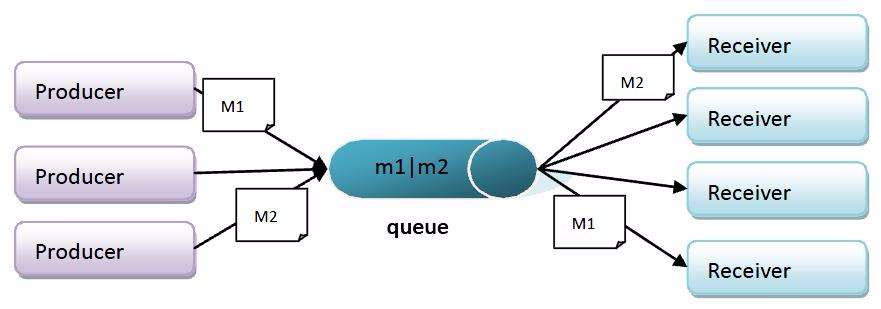
面向消息的中间件(message-oriented middleware0) MOM能够很好的解决以上的问题。

是指利用**高效可靠的消息传递机制进行与平台无关的数据交流**，并基于数据通信来进行分布式系统的集成。

通过提供消息传递和消息排队模型在分布式环境下提供应用解耦，弹性伸缩，冗余存储，流量削峰，异步通信，数据同步等

大致流程

发送者把消息发给消息服务器，消息服务器把消息存放在若干队列/主题中，在合适的时候，消息服务器会把消息转发给接受者。在这个过程中，发送和接受是异步的,也就是发送无需等待，发送者和接受者的生命周期也没有必然关系在发布pub/订阅sub模式下，也可以完成一对多的通信，可以让一个消息有多个接受者[微信订阅号就是这样的]



### 特点

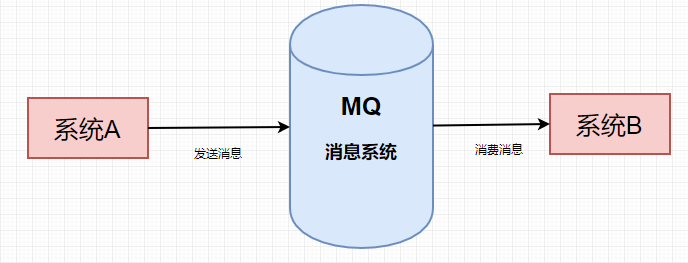
#### 异步处理模式

消息发送者可以发送一个消息而无需等待响应。消息发送者把消息发送到一条虚拟的通道(主题或队列)上;

消息接收者则订阅或监听该通道。一条信息可能最终转发给一个或多个消息接收者，这些接收者都无需对消息发送者做出回应。整个过程都是异步的。

案例：

也就是说，一个系统和另一个系统间进行通信的时候，假如系统A希望发送一个消息给系统B，让它去处理，但是系统A不关注系统B到底怎么处理或者有没有处理好，所以系统A把消息发送给MQ，然后就不管这条消息的“死活” 了，接着系统B从MQ里面消费出来处理即可。至于怎么处理，是否处理完毕，什么时候处理，都是系统B的事，与系统A无关。



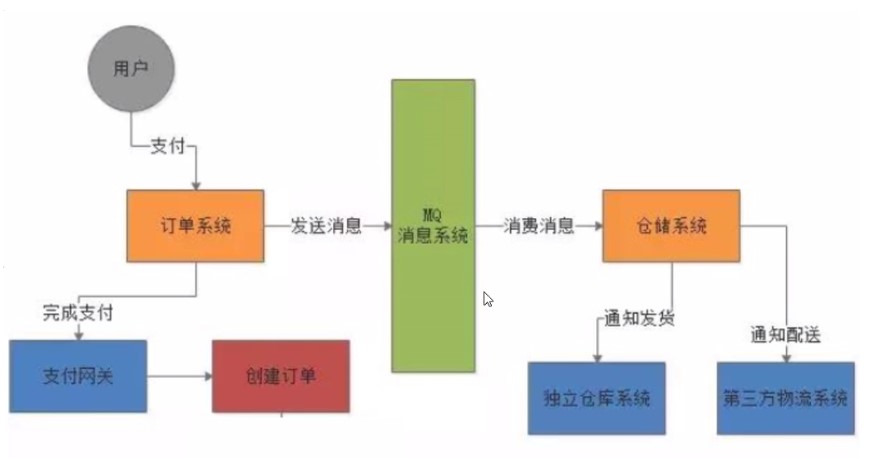
这样的一种通信方式，就是所谓的“异步”通信方式，对于系统A来说，只要把消息发给MQ,然后系统B就会异步处去进行处理了，系统A不能“同步”的等待系统B处理完。这样的好处是什么呢？解耦

#### 应用系统的解耦

  发送者和接收者不必了解对方，只需要确认消息

  发送者和接收者不必同时在线

#### 现实中的业务



## RabbitMQ概述及注意点

RabbitMQ是实现了**高级消息队列协议**（**AMQP**）的开源消息代理软件（亦称面向消息的中间件）。

RabbitMQ服务器是用**Erlang**语言编写的，而集群和故障转移是构建在开放电信平台框架上的。所有主要的编程语言均有与代理接口通讯的客户端库。

总结：RabbitMQ 实现了AMQP 协议来构建自己的消息队列

      RabbitMQ  是Erlang 语言写的，但是我们有操作RabbitMQ的java的驱动

面试问题： Activemq 和RabbitMq 的区别

根本的区别：Activemq 他实现的是**JMS 协议（Java 消息协议**）

           RabbitMQ  实现的是**AMQP 协议（高级消息队列协议**）

Activemq ： 是Java 写的 缺点 特别站内存

RabbitMQ : Erlang  写的： 吞吐更多，延时更低！！

当然：区别还有很大，你一学就知道了

## Rabbitmq的特点

RabbitMQ 最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。具体特点包括：

### 可靠性（Reliability）

RabbitMQ 使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

### 灵活的路由（Flexible Routing）

在消息进入队列之前，通过 Exchange 来路由消息的。对于典型的路由功能，RabbitMQ 已经提供了一些内置的 Exchange 来实现。针对更复杂的路由功能，可以将多个 Exchange 绑定在一起，也通过插件机制实现自己的 Exchange 。

### 消息集群（Clustering）

多个 RabbitMQ 服务器可以组成一个集群，形成一个逻辑 Broker 。

### 高可用（Highly Available Queues）

队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。

### 多种协议（Multi-protocol）

RabbitMQ 支持多种消息队列协议，比如 STOMP、MQTT 等等。 smtp

### 多语言客户端（Many Clients）

RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。

### 管理界面（Management UI）

RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。

### 跟踪机制（Tracing）

如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。

### 插件机制（Plugin System）

RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

# **【掌握】RabbitMQ安装**

## 安装前的说明

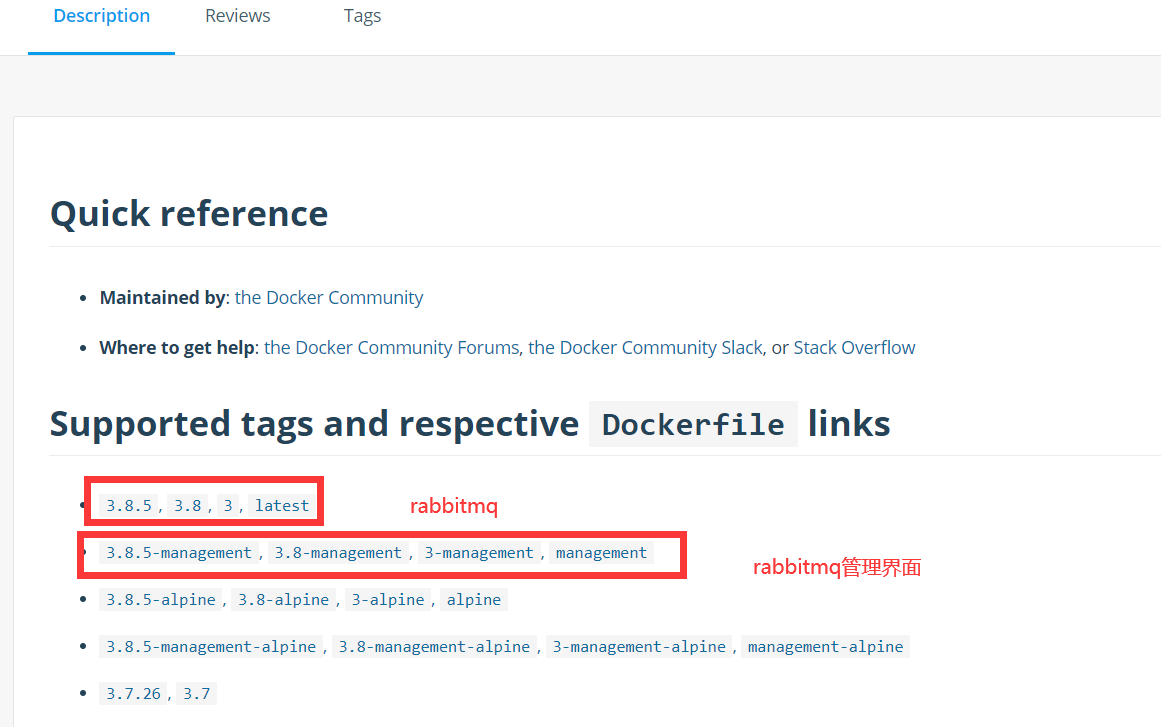
Rabbitmq->Erlang -> 安装Erlang 虚拟机->跑Rabbitmq 比较麻烦

Rabbitmq 对docker的支持非常到位！官网经常更新镜像，所以怎么办呢。废话，用docker跑啊

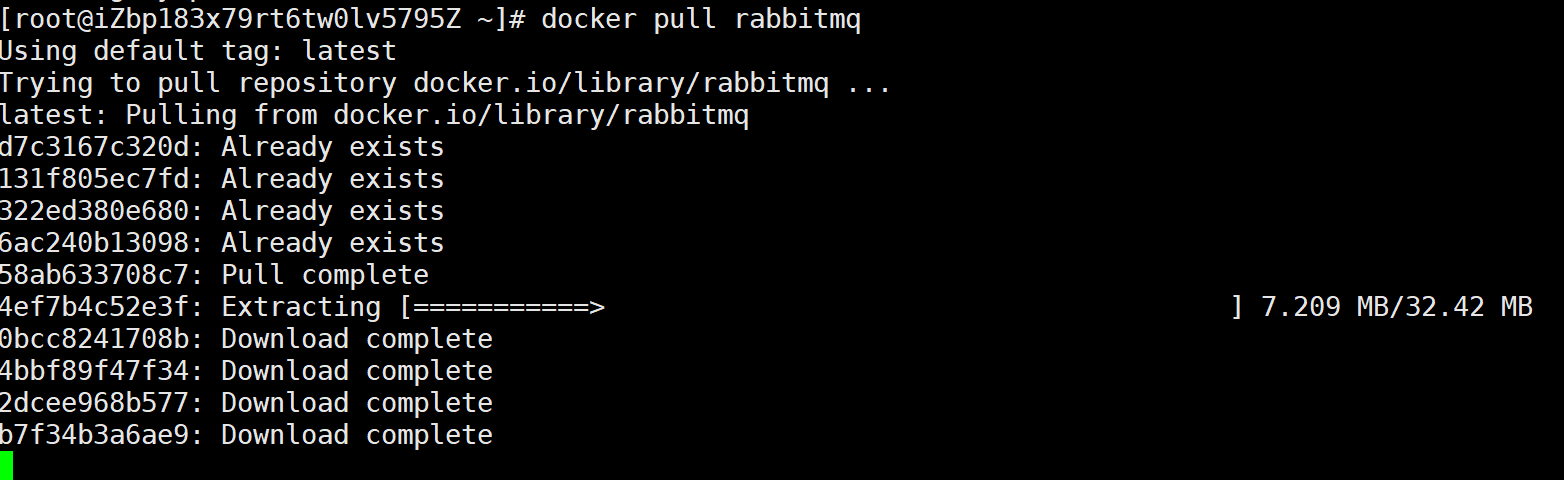
## 使用docker安装

### 下载镜像

<https://hub.docker.com/_/rabbitmq>



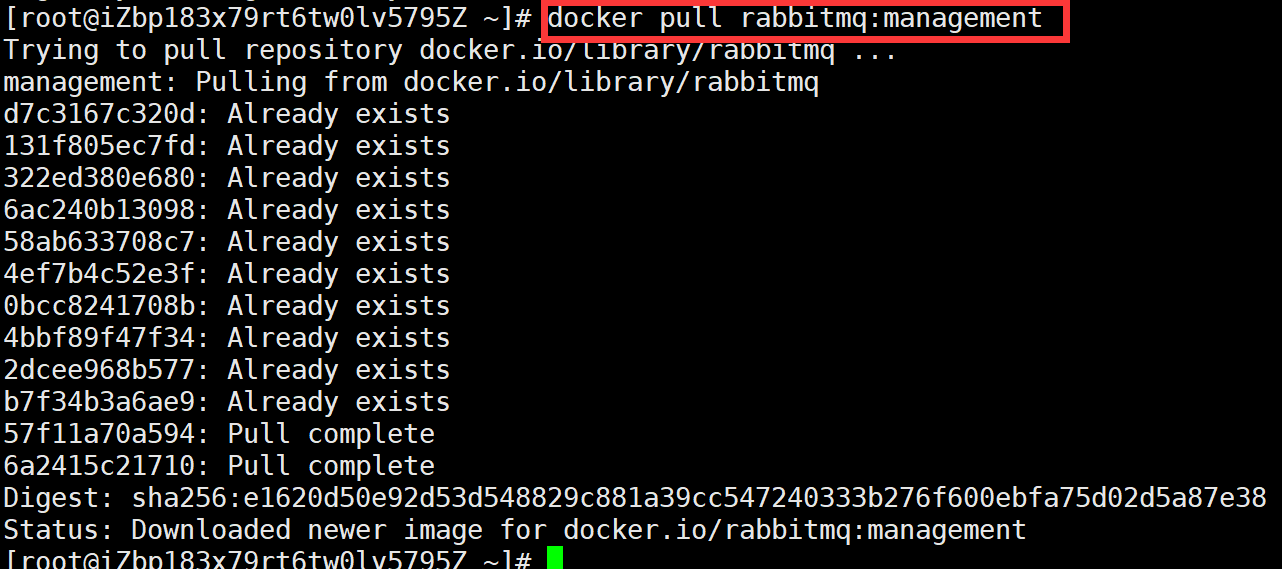
docker pull rabbitmq



IMG_258

Rabbitmq 里面也有控制台界面，但是它们不是一起的，你需要控制台的，需要下载另一个镜像

docker pull rabbitmq:management



### 运行容器

15672：图形化界面的端口

5672  ：数据的端口

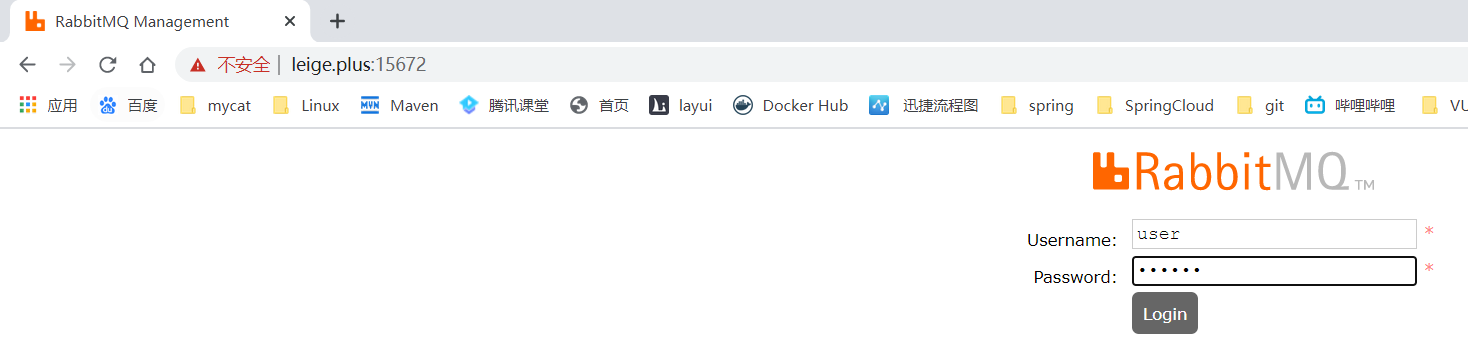
docker run --name rabbitmq -p 15672:15672 -p 5672:5672  -e RABBITMQ\_DEFAULT\_USER=user -e RABBITMQ\_DEFAULT\_PASS=123456 -d rabbitmq:3-management

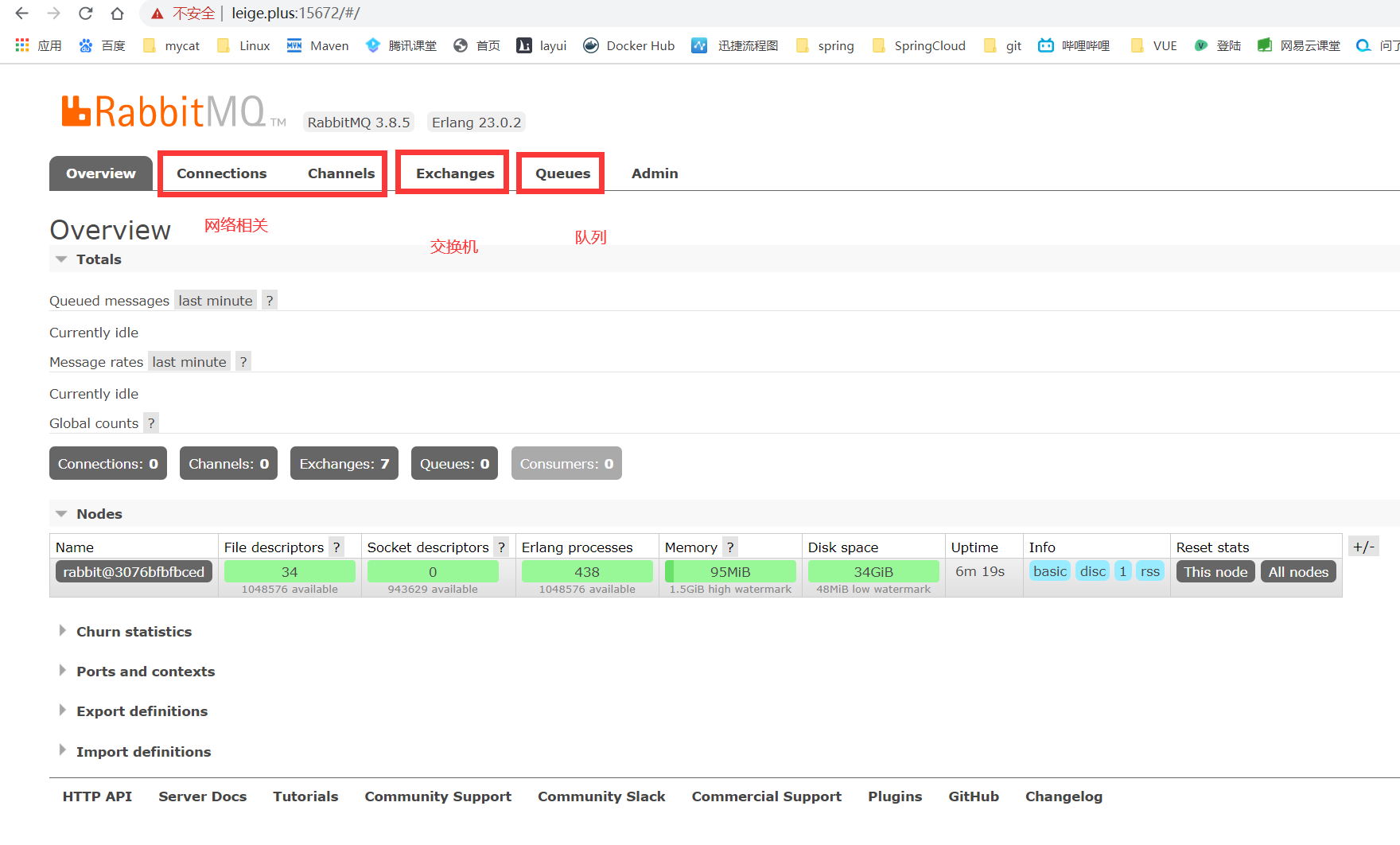
### 开发阿里云端口



### 访问rabbitMQ

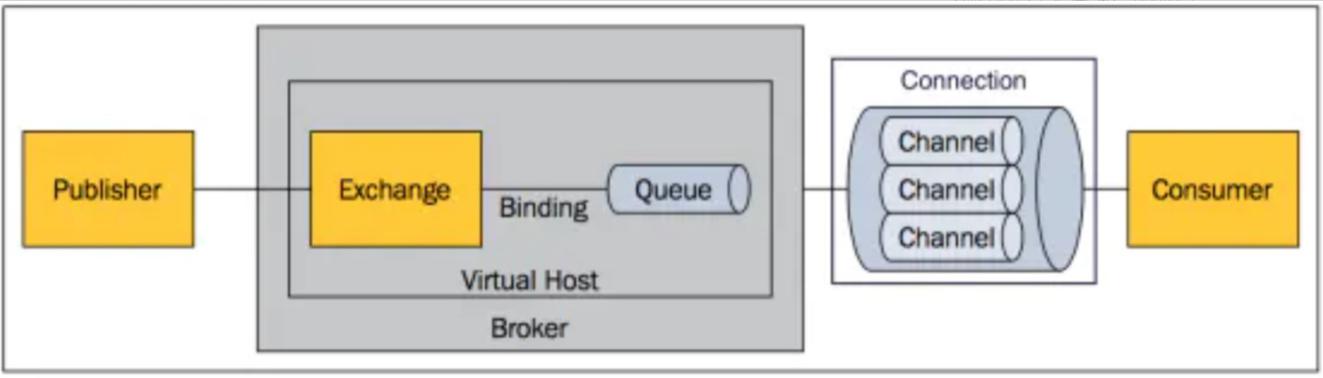
ip:15672





# **【掌握】RabbitMQ名词解释**

## 图说基本概念



## 名词解释

### Message

消息，消息是不具名的，它由消息头和消息体组成。消息体是不透明的，而消息头则由一系列的可选属性组成，这些属性包括routing-key（路由键）、priority（相对于其他消息的优先权）、delivery-mode（指出该消息可能需要持久性存储）等。

### Publisher

消息的生产者，也是一个向交换器发布消息的客户端应用程序。

### Exchange

交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。

### Binding

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。

### Queue

消息队列，用来保存消息到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

### Connection

网络连接，比如一个TCP连接。

### Channel

信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内地虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

### Consumer

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

### Virtual Host

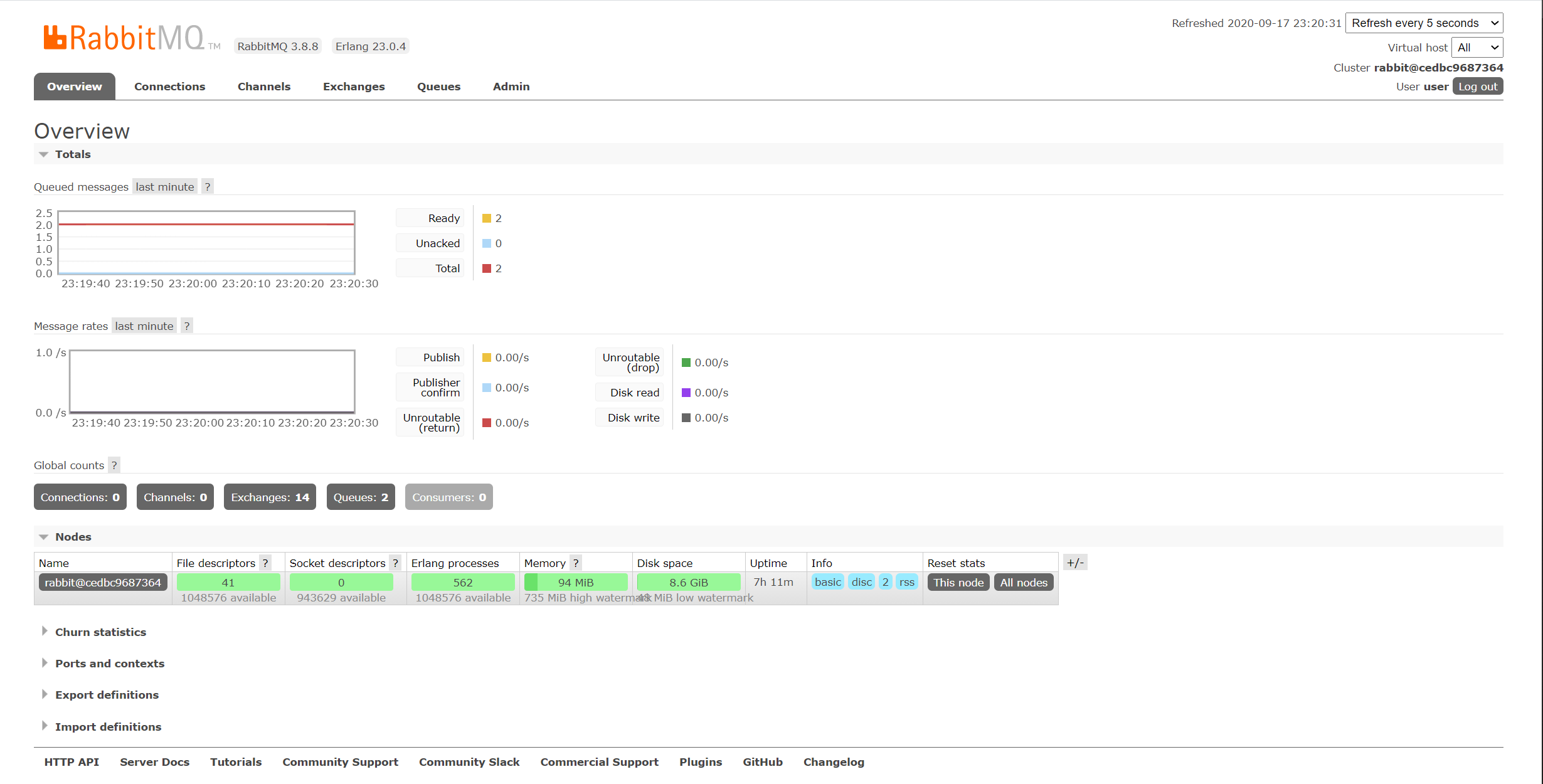
虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。

### Broker

表示消息队列服务器实体

# **【熟悉】web管理界面介绍**

## overview概览



connections：无论生产者还是消费者，都需要与RabbitMQ建立连接后才可以完成消息的生产和消费，在这里可以查看连接情况

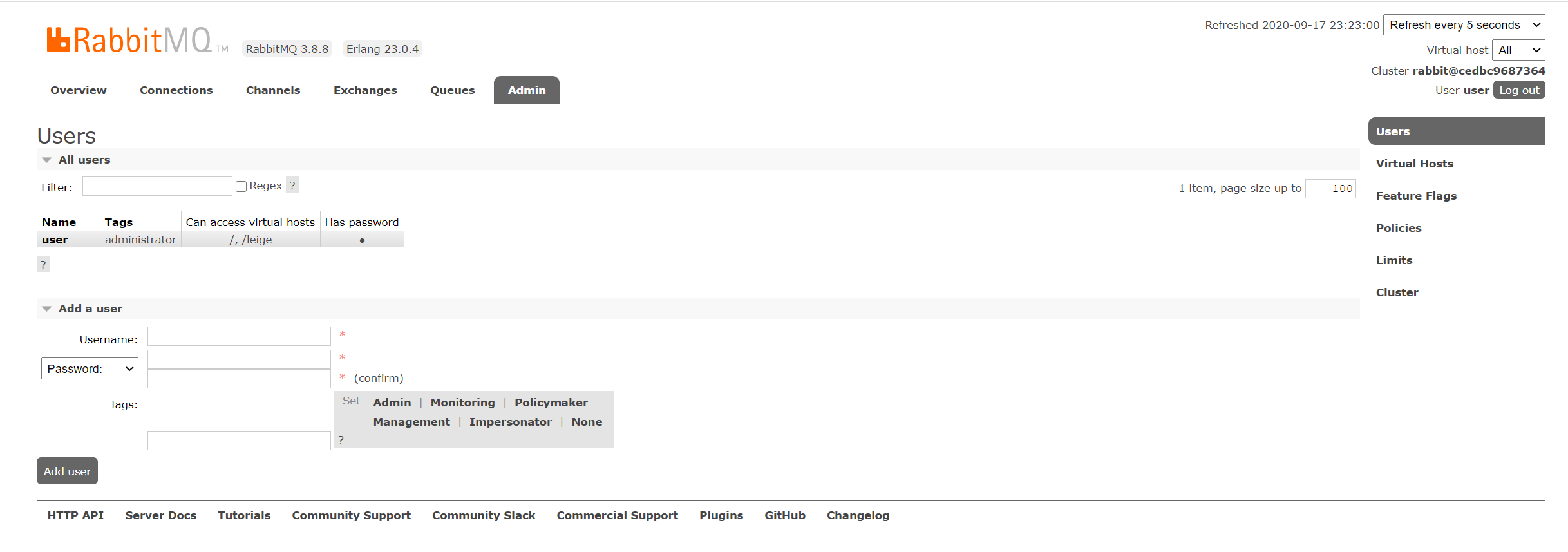
channels：通道，建立连接后，会形成通道，消息的投递获取依赖通道。

Exchanges：交换机，用来实现消息的路由

Queues：队列，即消息队列，消息存放在队列中，等待消费，消费后被移除队列。

## Admin用户和虚拟主机管理

### 添加用户



上面的Tags选项，其实是指定用户的角色，可选的有以下几个：

超级管理员(administrator)

可登陆管理控制台，可查看所有的信息，并且可以对用户，策略(policy)进行操作。

监控者(monitoring)

可登陆管理控制台，同时可以查看rabbitmq节点的相关信息(进程数，内存使用情况，磁盘使用情况等)

策略制定者(policymaker)

可登陆管理控制台, 同时可以对policy进行管理。但无法查看节点的相关信息(上图红框标识的部分)。

普通管理者(management)

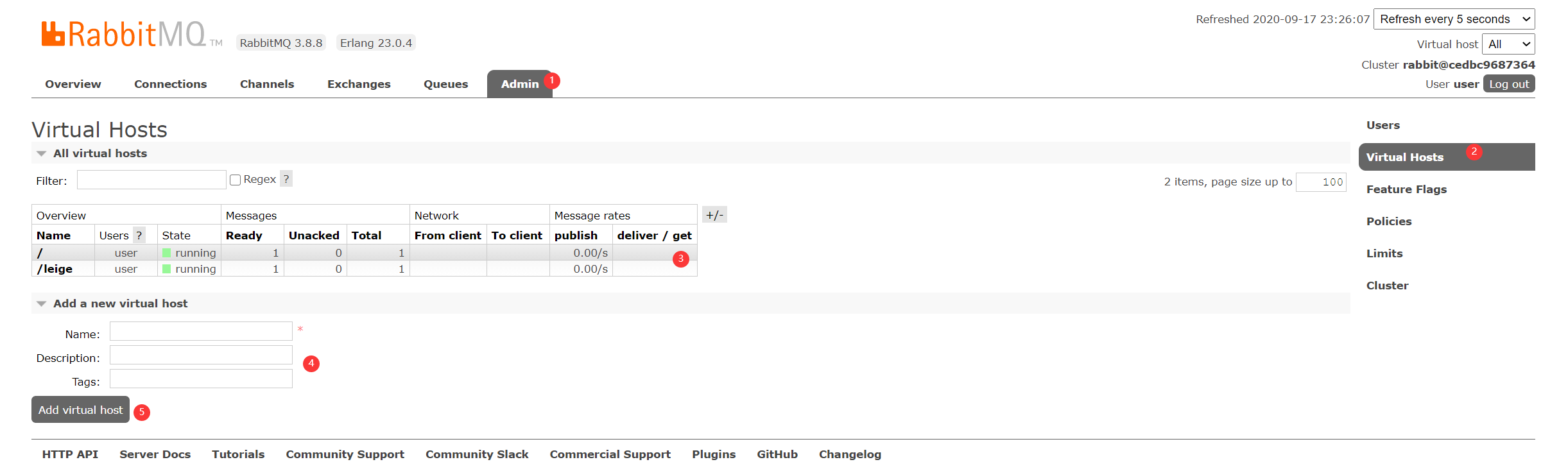
仅可登陆管理控制台，无法看到节点信息，也无法对策略进行管理。

其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

### 创建虚拟主机

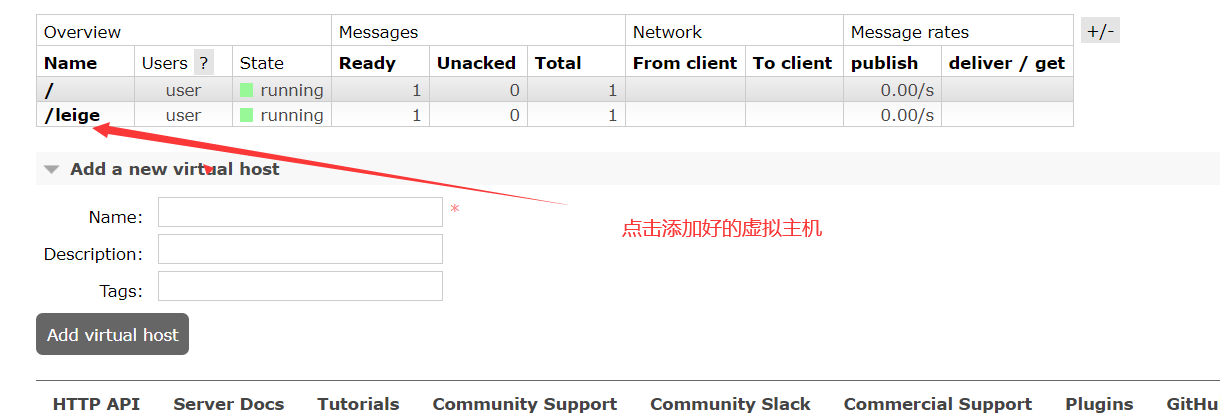
虚拟主机是什么  
 为了让各个用户可以互不干扰的工作，RabbitMQ添加了虚拟主机（Virtual Hosts）的概念。其实就是一个独立的访问路径，不同用户使用不同路径，各自有自己的队列、交换机，互相不会影响。



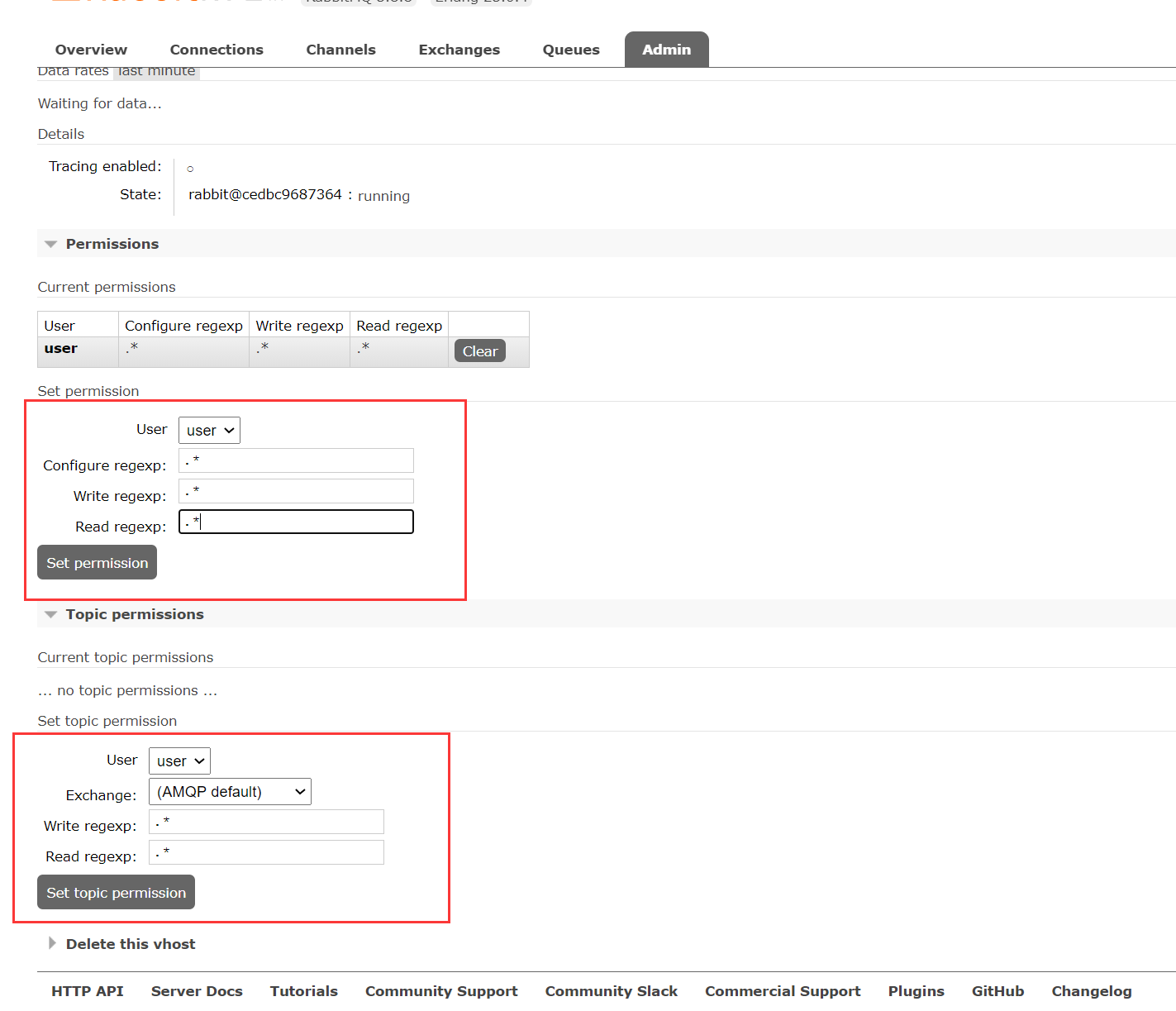
### 绑定虚拟主机和用户

创建好虚拟主机，我们还要给用户添加访问权限：

点击添加好的虚拟主机：



进入虚拟机设置界面:

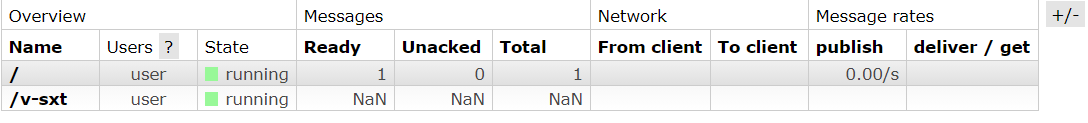


## 创建v-sxt的虚拟主机和sxt用户并分配权限

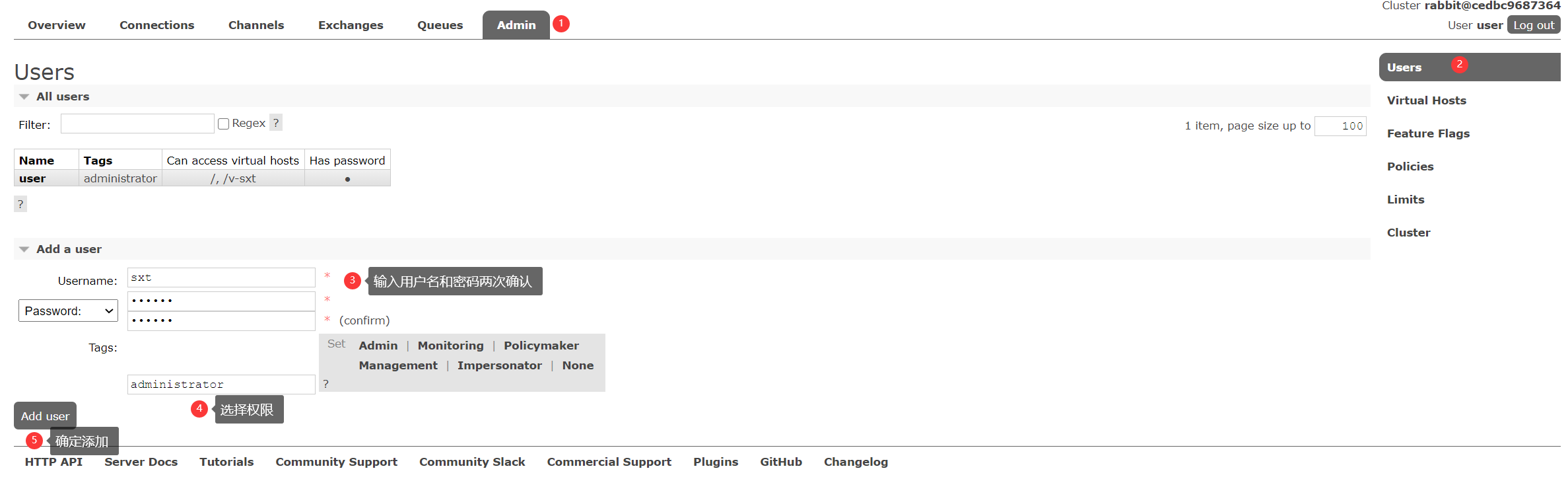
### 创建v-sxt虚拟主机



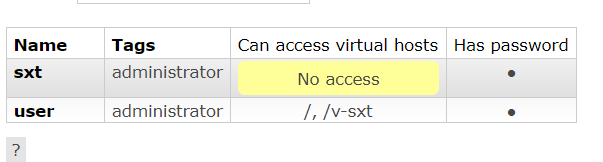
添加成功



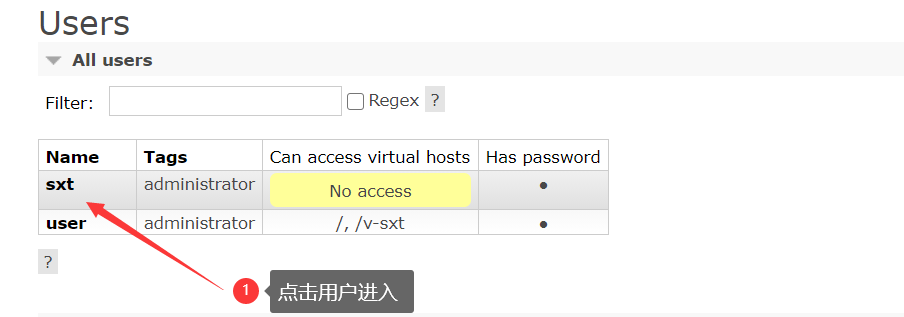
### 创建sxt用户

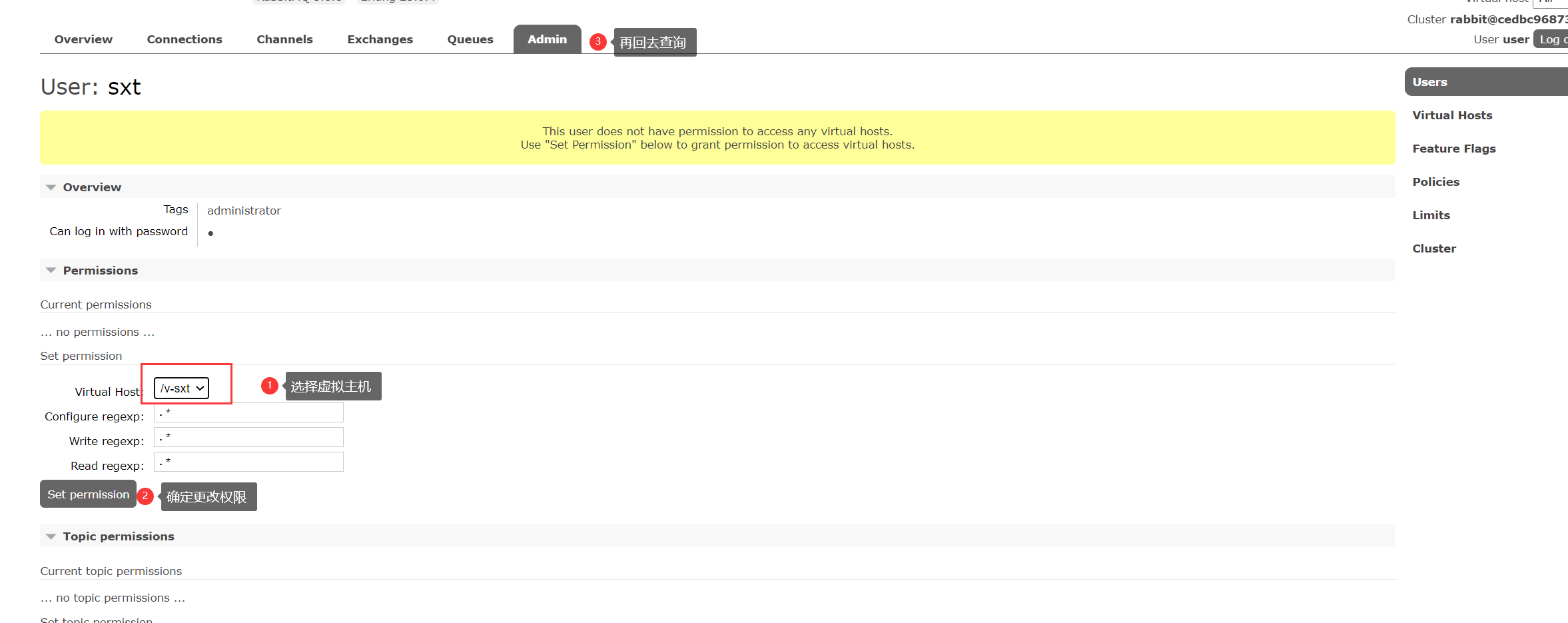


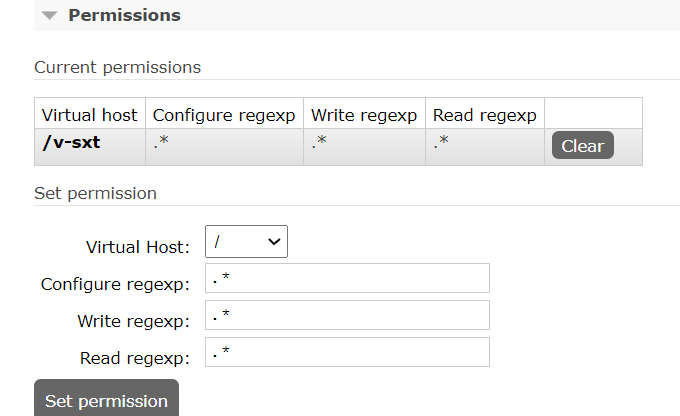
添加成功，但是现在不能访问任何虚拟主机

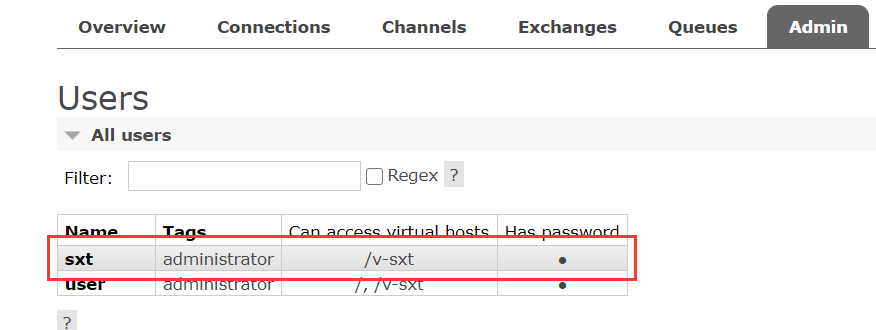


### 分配权限

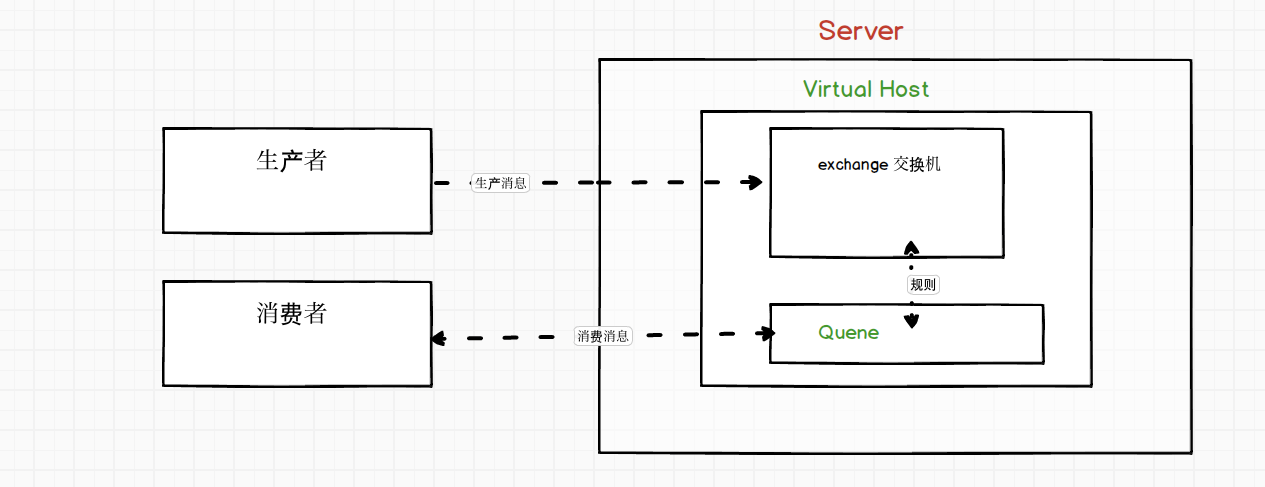






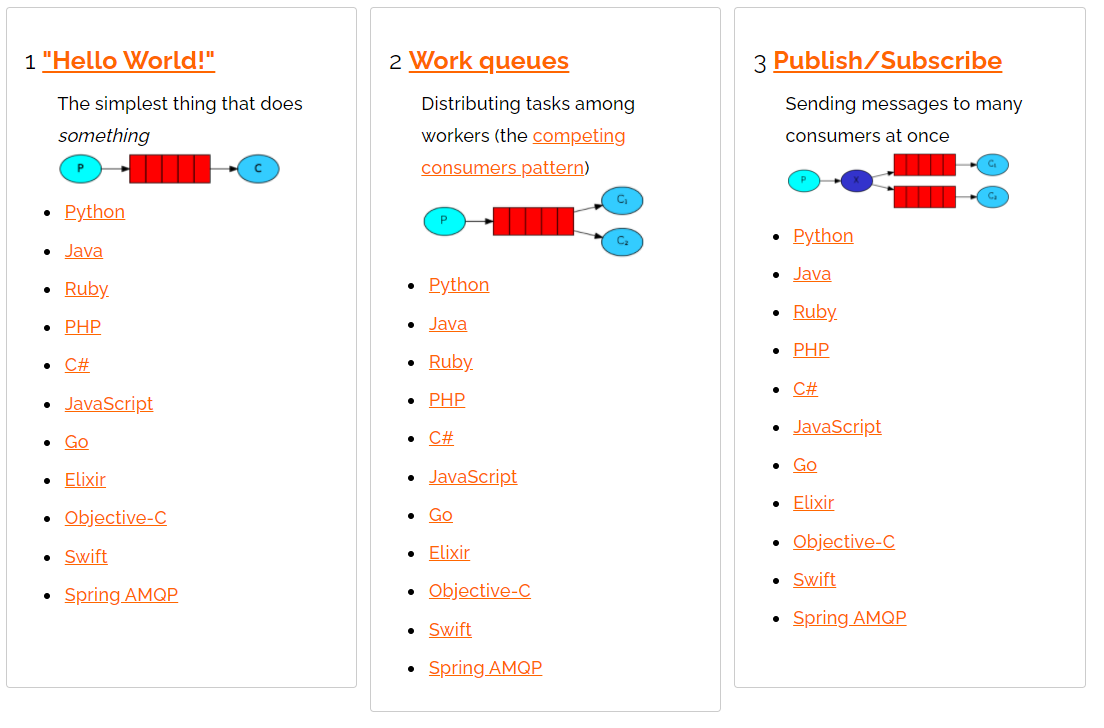
、  
  
  
  
  
  
？**【掌握】RabbitMQ说明及准备工作**

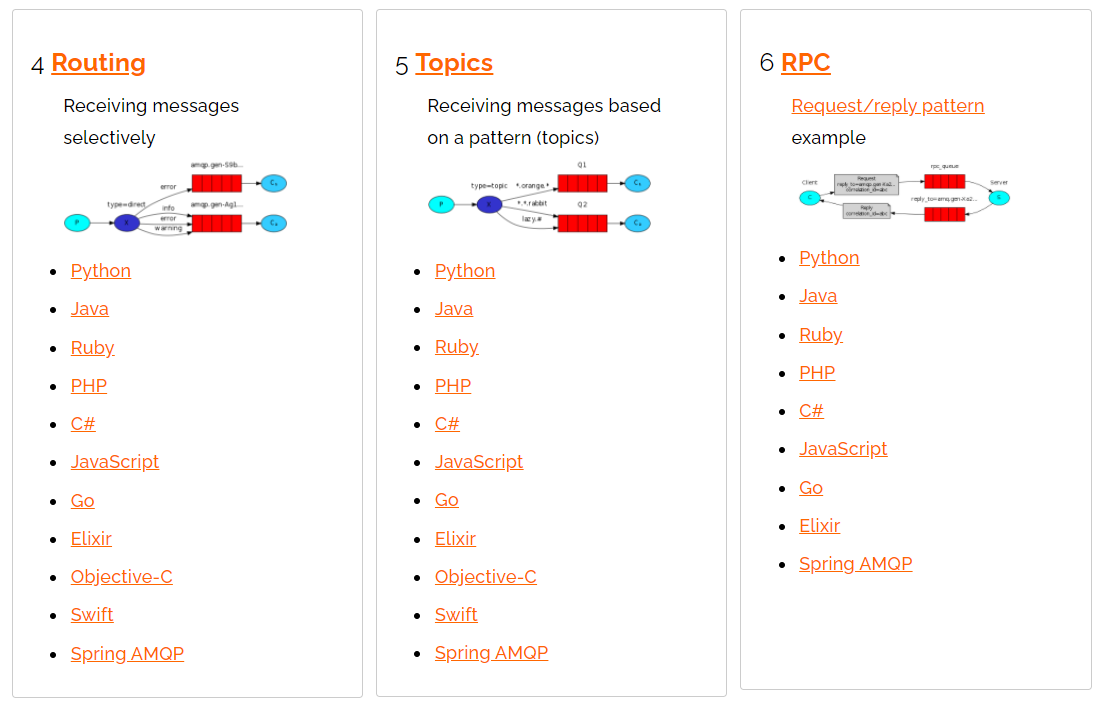
## AMQP协议的回顾



## RabbitMQ支持的消息模型

<https://www.rabbitmq.com/getstarted.html>

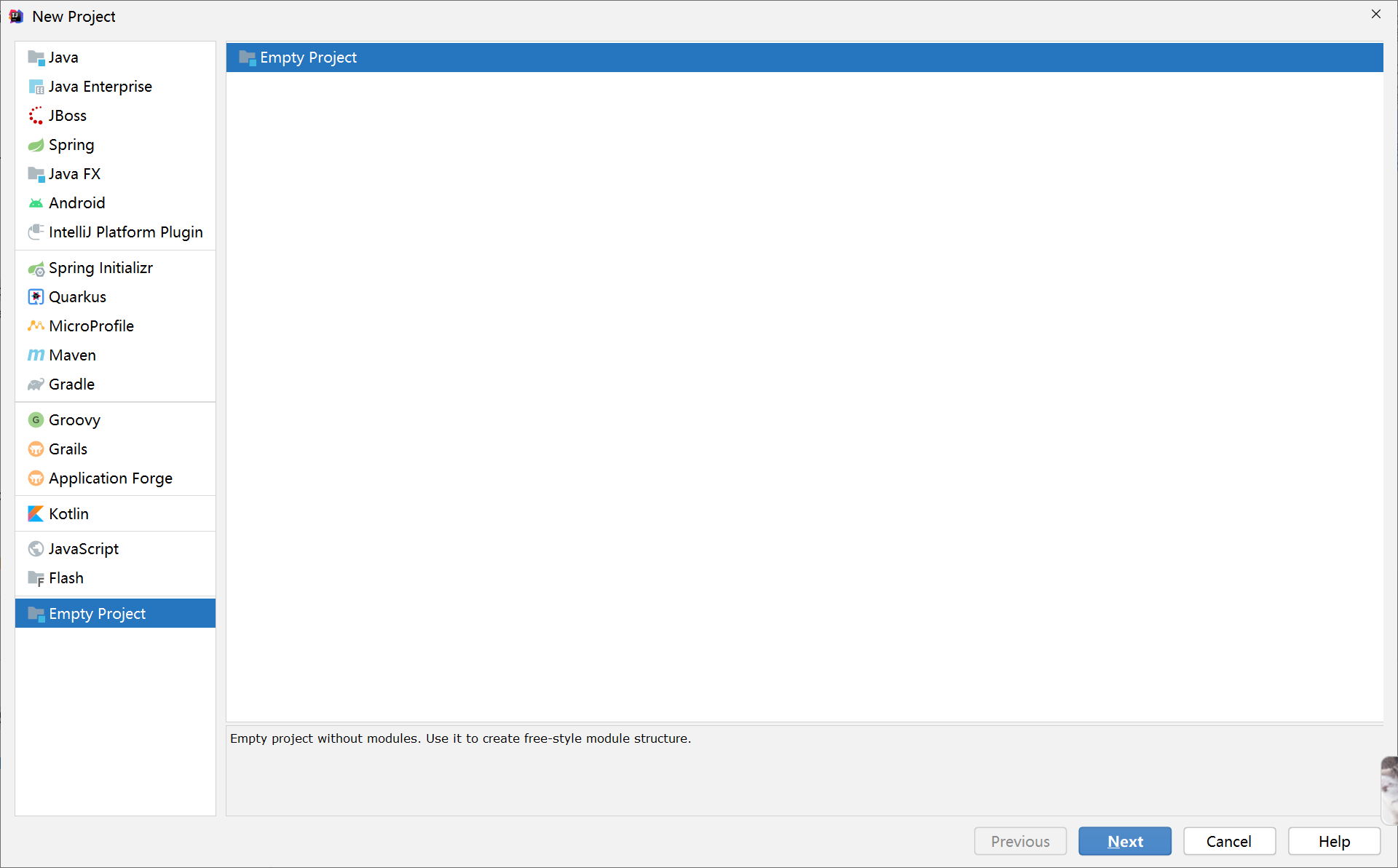


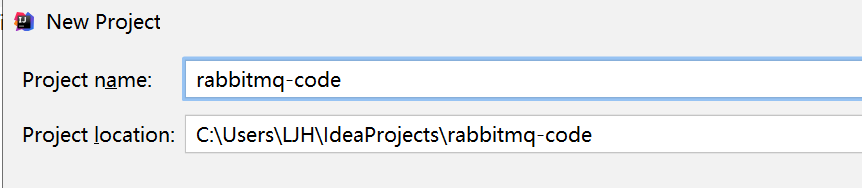




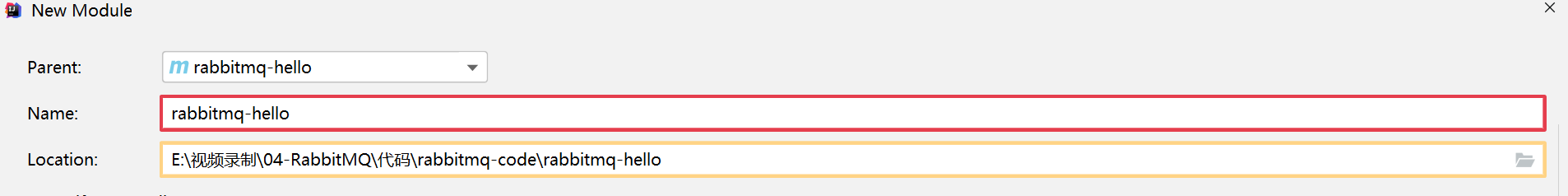
注意3.7的版本不支持第7种模式

## 创建空项目rabbitmq-code





## 创建rabbitmq-hello项目

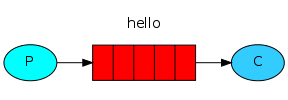


## 修改pom.xml引入依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**>  <**dependency**>  <**groupId**>com.rabbitmq</**groupId**>  <**artifactId**>amqp-client</**artifactId**>  <**version**>5.9.0</**version**>  </**dependency**> </**dependencies**> |

# **【掌握】第一种模型(直连)**

## 概述



在上图的模型中，有以下概念：

P：生产者，也就是要发送消息的程序

C：消费者：消息的接受者，会一直等待消息到来。

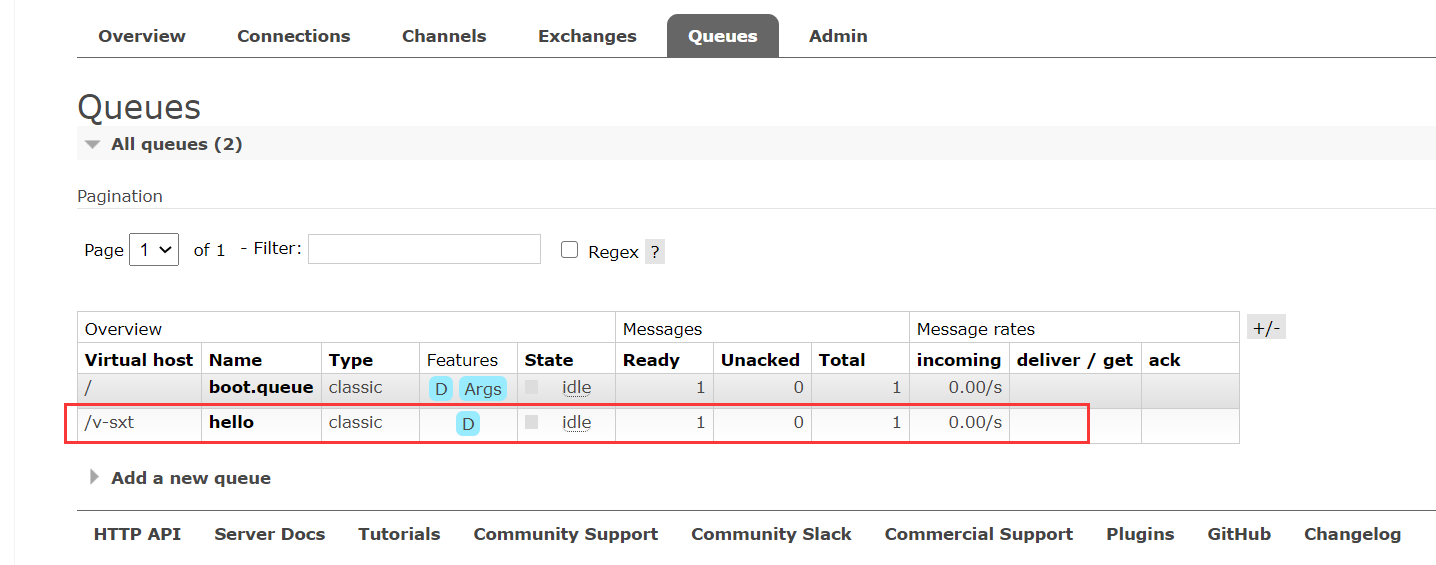
queue：消息队列，图中红色部分。类似一个邮箱，可以缓存消息；生产者向其中投递消息，消费者从其中取出消息。

## 创建生产者并发送测试

|  |
| --- |
| **package** a\_direct; **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory; **import** org.junit.Test; **import** java.io.IOException; **import** java.util.concurrent.TimeoutException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息生产者  \*/* **public class** Producer {  @Test  **public void** sendMessage() **throws** IOException, TimeoutException {  *//创建连接工厂* ConnectionFactory connectionFactory=**new** ConnectionFactory();  *//设置相关参数* connectionFactory.setHost(**"81.68.123.67"**);  connectionFactory.setPort(5672);  connectionFactory.setUsername(**"sxt"**);  connectionFactory.setPassword(**"123456"**);  connectionFactory.setVirtualHost(**"/v-sxt"**);  *//从连接工厂里面创建一个连接* Connection connection = connectionFactory.newConnection();  *//创建通道* Channel channel = connection.createChannel();  */\*\*  \* 绑定队列  \* 参数1：队列名 如果发送消息时，队列在mq里面不存在，它会创建一个新的队列  \* 参数2：是否持久化 如果为false 当MQ重启时，消息会丢失  \* 参数3：是否独享。true代表只有当前的connection可以访问这个队列  \* 参数4：是否自动删除 是否用完之后就删除  \* 参数5：其它参数  \*/* channel.queueDeclare(**"hello"**,**true**,**false**,**false**,**null**);   */\*\*  \* 发送消息  \* 参数1 交换机名称 暂时用不到  \* 参数2 队列名  \* 参数3 基础参数 是否持久  \* 参数4 消息的内容  \*/* channel.basicPublish(**""**,**"hello"**,**null**,**"hello rabbitmq"**.getBytes());   *//关闭通道和连接* channel.close();  connection.close();  System.***out***.println(**"消息发送成功"**);  } } |
|  |

## 创建消费者并发送测试Consumer

|  |
| --- |
| **package** a\_direct;  **import** com.rabbitmq.client.\*; **import** org.junit.Test;  **import** java.io.IOException; **import** java.util.concurrent.TimeoutException;  */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息消费者  \*/* **public class** Consumer {   @Test  **public void** receiveMessage() **throws** IOException, TimeoutException {  *//创建连接工厂* ConnectionFactory connectionFactory=**new** ConnectionFactory();  *//设置相关参数* connectionFactory.setHost(**"81.68.123.67"**);  connectionFactory.setPort(5672);  connectionFactory.setUsername(**"sxt"**);  connectionFactory.setPassword(**"123456"**);  connectionFactory.setVirtualHost(**"/v-sxt"**);  *//从连接工厂里面创建一个连接* Connection connection = connectionFactory.newConnection();  *//创建通道* Channel channel = connection.createChannel();  */\*\*  \* 绑定队列  \* 参数1：队列名 如果发送消息时，队列在mq里面不存在，它会创建一个新的队列  \* 参数2：是否持久化 如果为false 当MQ重启时，消息会丢失  \* 参数3：是否独享。true代表只有当前的connection可以访问这个队列  \* 参数4：是否自动删除 是否用完之后就删除  \* 参数5：其它参数  \*/* channel.queueDeclare(**"hello"**,**true**,**false**,**false**,**null**);   */\*\*  \* 接收消息  \* 参数1：队列名  \* 参数2：是否自动签收  \* 参数3：回调接口  \*  \*/* channel.basicConsume(**"hello"**,**true**,**new** DefaultConsumer(channel){  */\*\*  \** ***@param consumerTag*** *\** ***@param envelope*** *\** ***@param properties*** *\** ***@param body*** *消息体  \** ***@throws*** *IOException  \*/* @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者收到消息："**+**new** String(body));  }  });   *//不能让程序结束* System.***in***.read();  *//关闭 // channel.close(); // connection.close(); // System.out.println("消息消费成功");* } } |



## 有啥用？

这种就是一个点对点的发送和消费，一个生产者，一个消费者，可以用于登陆发短信等功能

## 优化工具类

|  |
| --- |
| **package** utils;  **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory;  */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 工具类  \*/* **public class** RabbitMQUtils {   **private static** ConnectionFactory *connectionFactory*;  **static**{  *//创建连接工厂  connectionFactory*=**new** ConnectionFactory();  *//设置相关参数  connectionFactory*.setHost(**"81.68.123.67"**);  *connectionFactory*.setPort(5672);  *connectionFactory*.setUsername(**"sxt"**);  *connectionFactory*.setPassword(**"123456"**);  *connectionFactory*.setVirtualHost(**"/v-sxt"**);  }   */\*\*  \* 提供一个获取连接的方法  \*/* **public static** Connection getConnection(){  **try** {  *//从连接工厂里面创建一个连接* Connection connection = *connectionFactory*.newConnection();  **return** connection;  }**catch** (Exception e){  e.printStackTrace();  }  **return null**;  }   */\*\*  \* 提供一个可以关闭通道和连接的方法  \*/* **public static void** closeChannelAndConnection(Channel channel,Connection connection){  **try**{  **if**(**null**!=channel)channel.close();  **if**(**null**!=connection) connection.close();  }**catch** (Exception e){  e.printStackTrace();  }  } } |

## 修改生产者

|  |
| --- |
| **package** a\_direct;  **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; **import** java.util.concurrent.TimeoutException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息生产者  \*/* **public class** Producer {  @Test  **public void** sendMessage() **throws** IOException, TimeoutException {   *//从连接工厂里面创建一个连接* Connection connection = RabbitMQUtils.*getConnection*();  *//创建通道* Channel channel = connection.createChannel();  */\*\*  \* 绑定队列  \* 参数1：队列名 如果发送消息时，队列在mq里面不存在，它会创建一个新的队列  \* 参数2：是否持久化 如果为false 当MQ重启时，消息会丢失  \* 参数3：是否独享。true代表只有当前的connection可以访问这个队列  \* 参数4：是否自动删除 是否用完之后就删除  \* 参数5：其它参数  \*/* channel.queueDeclare(**"hello"**,**true**,**false**,**false**,**null**);   */\*\*  \* 发送消息  \* 参数1 交换机名称 暂时用不到  \* 参数2 队列名  \* 参数3 基础参数 是否持久  \* 参数4 消息的内容  \*/* channel.basicPublish(**""**,**"hello"**,**null**,**"hello rabbitmq"**.getBytes());   *//关闭通道和连接* RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  System.***out***.println(**"消息发送成功"**);  } } |

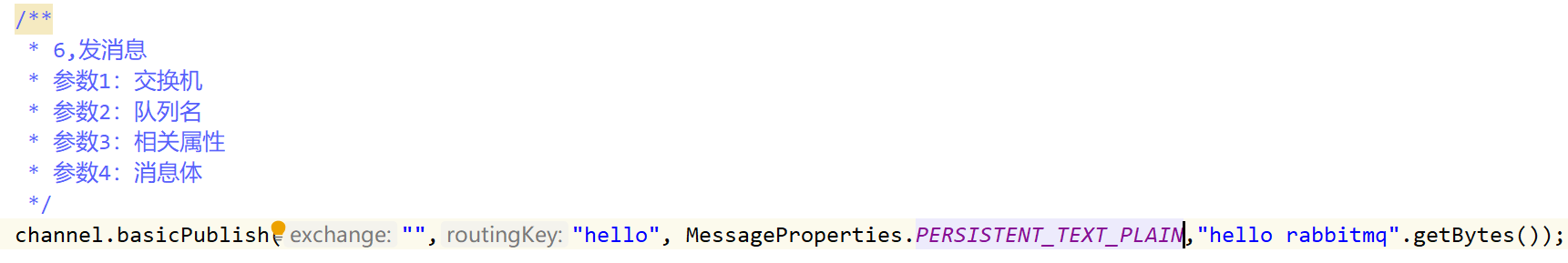
## 修改消费者

|  |
| --- |
| **package** a\_direct; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; **import** java.util.concurrent.TimeoutException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息消费者  \*/* **public class** Consumer {  @Test  **public void** receiveMessage() **throws** IOException, TimeoutException {  *//从连接工厂里面创建一个连接* Connection connection = RabbitMQUtils.*getConnection*();  *//创建通道* Channel channel = connection.createChannel();  */\*\*  \* 绑定队列  \* 参数1：队列名 如果发送消息时，队列在mq里面不存在，它会创建一个新的队列  \* 参数2：是否持久化 如果为false 当MQ重启时，消息会丢失  \* 参数3：是否独享。true代表只有当前的connection可以访问这个队列  \* 参数4：是否自动删除 是否用完之后就删除  \* 参数5：其它参数  \*/* channel.queueDeclare(**"hello"**,**true**,**false**,**false**,**null**);  */\*\*  \* 接收消息  \* 参数1：队列名  \* 参数2：是否自动签收  \* 参数3：回调接口  \*  \*/* channel.basicConsume(**"hello"**,**true**,**new** DefaultConsumer(channel){  */\*\*  \** ***@param consumerTag*** *\** ***@param envelope*** *\** ***@param properties*** *\** ***@param body*** *消息体  \** ***@throws*** *IOException  \*/* @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者收到消息："**+**new** String(body));  }  });  *//不能让程序结束* System.***in***.read();  *//关闭* RabbitMQUtils.*closeChannelAndConnection*(channel,connection); *// channel.close(); // connection.close(); // System.out.println("消息消费成功");* } } |

## 相关参数详解的演示

如果只设置队列的持久化，消息默认的是不会持久化的

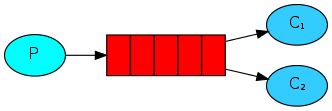




# **【掌握】****第二种模型(work quene)**

## 概述

Work queues，也被称为（Task queues），任务模型。当消息处理比较耗时的时候，可能生产消息的速度会远远大于消息的消费速度。长此以往，消息就会堆积越来越多，无法及时处理。此时就可以使用work 模型：让多个消费者绑定到一个队列，共同消费队列中的消息。队列中的消息一旦消费，就会消失，因此任务是不会被重复执行的。



角色：

P：生产者：任务的发布者

C1：消费者-1，领取任务并且完成任务，假设完成速度较慢

C2：消费者-2：领取任务并完成任务，假设完成速度快

## 创建生产者

|  |
| --- |
| **package** b\_workqueue; **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.MessageProperties; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; **import** java.util.concurrent.TimeoutException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息生产者  \*/* **public class** Producer {  @Test  **public void** sendMessage() **throws** IOException, TimeoutException {  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  channel.queueDeclare(**"hello"**,**false**,**false**,**false**,**null**);  **for** (**int** i = 1; i <=100 ; i++) {  channel.basicPublish(**""**,**"hello"**, **null**,(**"hello rabbitmq workqueue--"**+i).getBytes());  }  *//关闭通道和连接* RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  System.***out***.println(**"消息发送成功"**);  } } |

## 创建消费者1

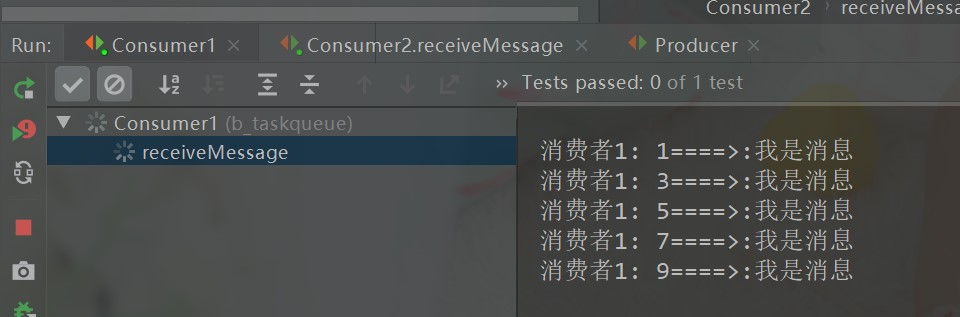
|  |
| --- |
| **package** b\_workqueue; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; **import** java.util.concurrent.TimeoutException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息消费者  \*/* **public class** Consumer1 {   @Test  **public void** receiveMessage() **throws** IOException, TimeoutException {  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  channel.queueDeclare(**"hello"**,**false**,**false**,**false**,**null**);  channel.basicConsume(**"hello"**,**true**,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【1】收到消息："**+**new** String(body));  }  });  System.***out***.println(**"消费者【1】启动成功"**);  *//不能让程序结束* System.***in***.read();  *//关闭* RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  } } |

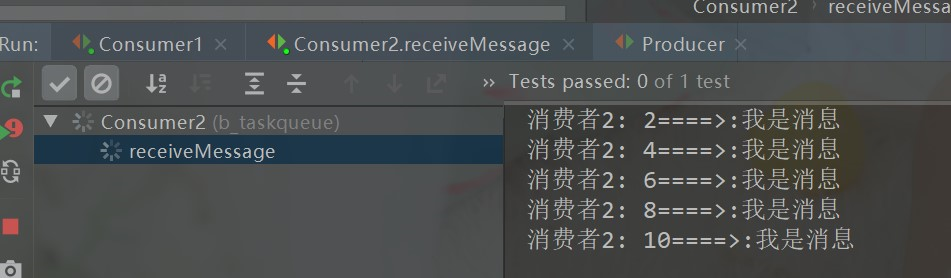
## 创建消费者2

|  |
| --- |
| **package** b\_workqueue;  **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; **import** java.util.concurrent.TimeoutException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* 消息消费者  \*/* **public class** Consumer2 {  @Test  **public void** receiveMessage() **throws** IOException, TimeoutException {  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel(); *// channel.basicQos(1);//一次只处理一条消息* channel.queueDeclare(**"hello"**,**false**,**false**,**false**,**null**);  *//把签收模式变成false* channel.basicConsume(**"hello"**,**false**,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException { *// try { // Thread.sleep(500); // } catch (InterruptedException e) { // e.printStackTrace(); // }  //手动签收  //channel.basicAck(envelope.getDeliveryTag(),false);* System.***out***.println(**"消费者【2】收到消息："**+**new** String(body));  }  });  System.***out***.println(**"消费者【2】启动成功"**);  *//不能让程序结束* System.***in***.read();  *//关闭* RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  } } |

## 测试

先启动消费者1和消费者2，再发送消息，发现结果如下





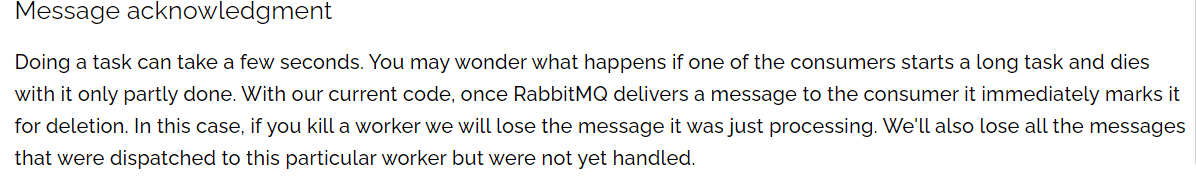
他们是平均消费的 官网有说明 <https://www.rabbitmq.com/tutorials/tutorial-two-java.html>



那么实际开发中可能有消费者处理的慢，有的处理的快，那么如何配置呢  引入自动确认机制

## 消息的自动确认机制【重点】

<https://www.rabbitmq.com/tutorials/tutorial-two-java.html>



完成一项任务可能需要几秒钟。您可能想知道，如果一个消费者开始了一项很长的任务，但只完成了一部分就去世了，会发生什么。在我们当前的代码中，一旦RabbitMQ向使用者传递了一条消息，它就会立即将其标记为删除。在这种情况下，如果你杀死一个工人，我们就会丢失它正在处理的信息。我们还将丢失所有发送给这个特定worker但尚未处理的消息。

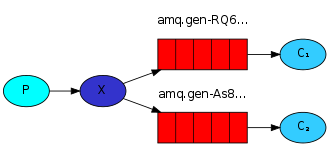
但我们不想失去任何任务。如果一个工人死亡，我们希望任务被交付给另一个工人。

|  |
| --- |
| *//设置一次只接收一条未确认的消息* channel.basicQos(2);  *//接收消息 参数2:关闭自动确认消息* channel.basicConsume(**"hello"**,**false**,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties,  **byte**[] body) **throws** IOException { *// try { // Thread.sleep(500); // } catch (InterruptedException e) { // e.printStackTrace(); // } // i++; // if(i<=2){ // int a=10/0; // }* System.***out***.println(**"消息者【1】接收到消息："**+**new** String(body));  *//手动签收消息* channel.basicAck(envelope.getDeliveryTag(),**false**);  }  }); |

# **【掌握】第三种模型(fanout)**

## 概述

fanout 扇出 也称为广播



在广播模式下，消息发送流程是这样的：

* 可以有多个消费者
* 每个消费者有自己的queue（队列）
* 每个队列都要绑定到Exchange（交换机）
* 生产者发送的消息，只能发送到交换机，交换机来决定要发给哪个队列，生产者无法决定。
* 交换机把消息发送给绑定过的所有队列
* 队列的消费者都能拿到消息。实现一条消息被多个消费者消费

## 创建生产者

|  |
| --- |
| **package** c\_fanout; **import** com.rabbitmq.client.BuiltinExchangeType; **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* fanout发送者测试  \*/* **public class** Producer {  @Test  **public void** sendMessage() **throws** IOException {  Connection connection = RabbitMQUtils.*getConnection*();  *//创建通道* Channel channel = connection.createChannel();  *//设置交换机 // channel.exchangeDeclare("logs", "fanout");* channel.exchangeDeclare(**"logs"**, BuiltinExchangeType.***FANOUT***);  *//向交换机发消息* channel.basicPublish(**"logs"**,**""**,**null**,(**"我是个fanout类型的消息"**).getBytes());  RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  System.***out***.println(**"消费发送成功"**);  } } |

## 创建消费者1

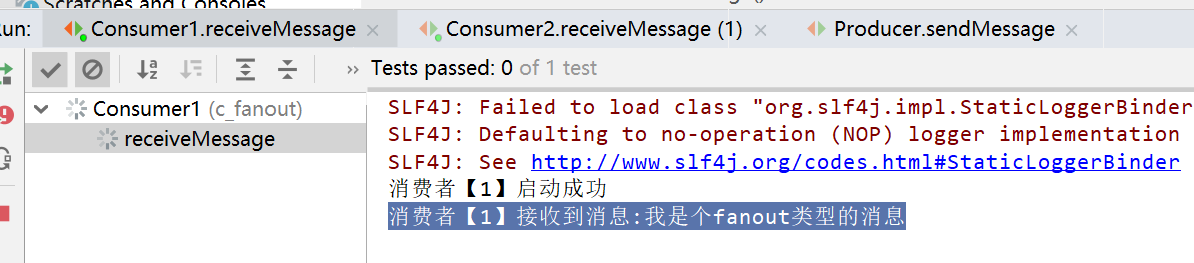
|  |
| --- |
| **package** c\_fanout; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* fanout类型交换机的消费者类  \*/* **public class** Consumer1 {  @Test  **public void** receiveMessage() **throws** IOException {  Connection connection = RabbitMQUtils.*getConnection*();  *//得到通道* Channel channel = connection.createChannel();  *//绑定交换机* channel.exchangeDeclare(**"logs"**, BuiltinExchangeType.***FANOUT***);  *//从通道里面得到一个临时队列* String queue = channel.queueDeclare().getQueue();  *//把临时队列和交换机绑定* channel.queueBind(queue,**"logs"**,**""**);  *//接收消息* channel.basicConsume(queue,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【1】接收到消息:"**+**new** String(body));  }  });  System.***out***.println(**"消费者【1】启动成功"**);  System.***in***.read();  } } |

## 创建消费者2

|  |
| --- |
| **package** c\_fanout; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* fanout类型交换机的消费者类  \*/* **public class** Consumer2 {  @Test  **public void** receiveMessage() **throws** IOException {  Connection connection = RabbitMQUtils.*getConnection*();  *//得到通道* Channel channel = connection.createChannel();  *//绑定交换机* channel.exchangeDeclare(**"logs"**, BuiltinExchangeType.***FANOUT***);  *//从通道里面得到一个临时队列* String queue = channel.queueDeclare().getQueue();  *//把临时队列和交换机绑定* channel.queueBind(queue,**"logs"**,**""**);  *//接收消息* channel.basicConsume(queue,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【2】接收到消息:"**+**new** String(body));  }  });  System.***out***.println(**"消费者【2】启动成功"**);  System.***in***.read();  } } |

## 测试

先启动两个消费者，再发消息





# **【掌握】第四种模型(Routing)-Direct**

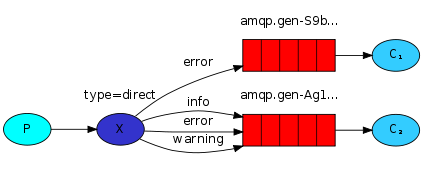
## 概述

在Fanout模式中，一条消息，会被所有订阅的队列都消费。但是，在某些场景下，我们希望不同的消息被不同的队列消费。这时就要用到Direct类型的Exchange。

在Direct模型下：

* 队列与交换机的绑定，不能是任意绑定了，而是要指定一个RoutingKey（路由key）
* 消息的发送方在 向 Exchange发送消息时，也必须指定消息的 RoutingKey。
* Exchange不再把消息交给每一个绑定的队列，而是根据消息的Routing Key进行判断，只有队列的Routingkey与消息的 Routing key完全一致，才会接收到消息

流程:



图解：

P：生产者，向Exchange发送消息，发送消息时，会指定一个routing key。

X：Exchange（交换机），接收生产者的消息，然后把消息递交给 与routing key完全匹配的队列

C1：消费者，其所在队列指定了需要routing key 为 error 的消息

C2：消费者，其所在队列指定了需要routing key 为 info、error、warning 的消息

## 创建生产者

|  |
| --- |
| **package** d\_routing\_direct;  **import** com.rabbitmq.client.BuiltinExchangeType; **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** org.junit.Test; **import** utils.RabbitMQUtils; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \*  \* 交换机直连模式  \*/* **public class** Producer {  @Test  **public void** sendMessage() **throws** Exception{  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  String exchangeName = **"logs\_direct"**;  channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***DIRECT***);  *//声明一个路由key* String routingKey=**"error"**;  channel.basicPublish(exchangeName,routingKey,**null**,  (**"我是一个直连类型的交换机消息-routingKey:"**+routingKey).getBytes());  System.***out***.println(**"消息发送成功"**);  RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  } } |

## 创建消费者1

|  |
| --- |
| **package** d\_routing\_direct; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException;  */\*\*  \** ***@Author:*** *尚学堂 雷哥  \*  \* 交换机直连模式  \*/* **public class** Consumer1 {   @Test  **public void** receiveMessage() **throws** Exception{  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  String exchangeName = **"logs\_direct"**;  channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***DIRECT***);  *//得到一个临时队列* String queue = channel.queueDeclare().getQueue();  *//绑定队列到交换机* channel.queueBind(queue,exchangeName,**"error"**);  *//消费消息* channel.basicConsume(queue,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【1】收到消息:"**+**new** String(body));  }  });  System.***in***.read();  } } |

## 创建消费者2

|  |
| --- |
| **package** d\_routing\_direct;  **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils;  **import** java.io.IOException;  */\*\*  \** ***@Author:*** *尚学堂 雷哥  \*  \* 交换机直连模式  \*/* **public class** Consumer2 {   @Test  **public void** receiveMessage() **throws** Exception{  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  String exchangeName = **"logs\_direct"**;  channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***DIRECT***);  *//得到一个临时队列* String queue = channel.queueDeclare().getQueue();  *//绑定队列到交换机* channel.queueBind(queue,exchangeName,**"info"**);  channel.queueBind(queue,exchangeName,**"error"**);  channel.queueBind(queue,exchangeName,**"warm"**);  *//消费消息* channel.basicConsume(queue,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【1】收到消息:"**+**new** String(body));  }  });  System.***in***.read();  } } |

## 测试

先启动两个consumer

再使用Producer发送routingKey=error 发现两个都收到

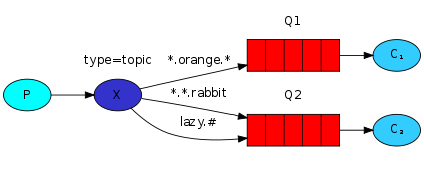
如果发送info 发现只有consumer2收到

说明：它的路由key的匹配规则是等值匹配

# **【掌握】第五种模型(Routing)-Topic**

## 概述

Topic类型的Exchange与Direct相比，都是可以根据RoutingKey把消息路由到不同的队列。只不过Topic类型Exchange可以让队列在绑定Routing key 的时候**使用通配符**！这种模型Routingkey 一般都是由一个或多个单词组成，多个单词之间以”.”分割，例如： item.insert



通配符

\* (star) can substitute for exactly one word. 匹配不多不少恰好1个词

# (hash) can substitute for zero or more words. 匹配0个或多个词

如:

audit.# 匹配audit.irs.corporate或者 audit.irs 等

audit.\* 只能匹配 audit.irs

## 创建生产者

|  |
| --- |
| **package** e\_routing\_topic;  **import** com.rabbitmq.client.BuiltinExchangeType; **import** com.rabbitmq.client.Channel; **import** com.rabbitmq.client.Connection; **import** org.junit.Test; **import** utils.RabbitMQUtils;  */\*\*  \** ***@Author:*** *尚学堂 雷哥  \*  \* 交换机topic模式  \*/* **public class** Producer {  @Test  **public void** sendMessage() **throws** Exception{  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  String exchangeName = **"topic"**;  channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***TOPIC***);  *//声明一个路由key* String routingKey=**"user"**;  channel.basicPublish(exchangeName,routingKey,**null**,  (**"我是一个topic类型的交换机消息-routingKey:"**+routingKey).getBytes());  System.***out***.println(**"消息发送成功"**);  RabbitMQUtils.*closeChannelAndConnection*(channel,connection);  } } |

## 创建消费者1

|  |
| --- |
| **package** e\_routing\_topic; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* \* 交换机直连模式  \*/* **public class** Consumer1 {   @Test  **public void** receiveMessage() **throws** Exception{  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  String exchangeName = **"topic"**;  channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***TOPIC***);  *//得到一个临时队列* String queue = channel.queueDeclare().getQueue();  *//绑定队列到交换机* channel.queueBind(queue,exchangeName,**"user.\*"**);  *//消费消息* channel.basicConsume(queue,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【1】收到消息:"**+**new** String(body));  }  });  System.***in***.read();  } } |

## 创建消费者2

|  |
| --- |
| **package** e\_routing\_topic; **import** com.rabbitmq.client.\*; **import** org.junit.Test; **import** utils.RabbitMQUtils; **import** java.io.IOException; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \* \* 交换机直连模式  \*/*  **public class** Consumer2 {  @Test  **public void** receiveMessage() **throws** Exception{  Connection connection = RabbitMQUtils.*getConnection*();  Channel channel = connection.createChannel();  String exchangeName = **"topic"**;  channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***TOPIC***);  *//得到一个临时队列* String queue = channel.queueDeclare().getQueue();  *//绑定队列到交换机* channel.queueBind(queue,exchangeName,**"user.#"**);  *//消费消息* channel.basicConsume(queue,**new** DefaultConsumer(channel){  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println(**"消费者【2】收到消息:"**+**new** String(body));  }  });  System.***in***.read();  } } |

## 测试

先启动消费者1和消费者2

再分别发送routingKey= user 和user.add 和user.add.insert等类型的消息、

查看结果

# **【掌握】SpringBoot中使用RabbitMQ**

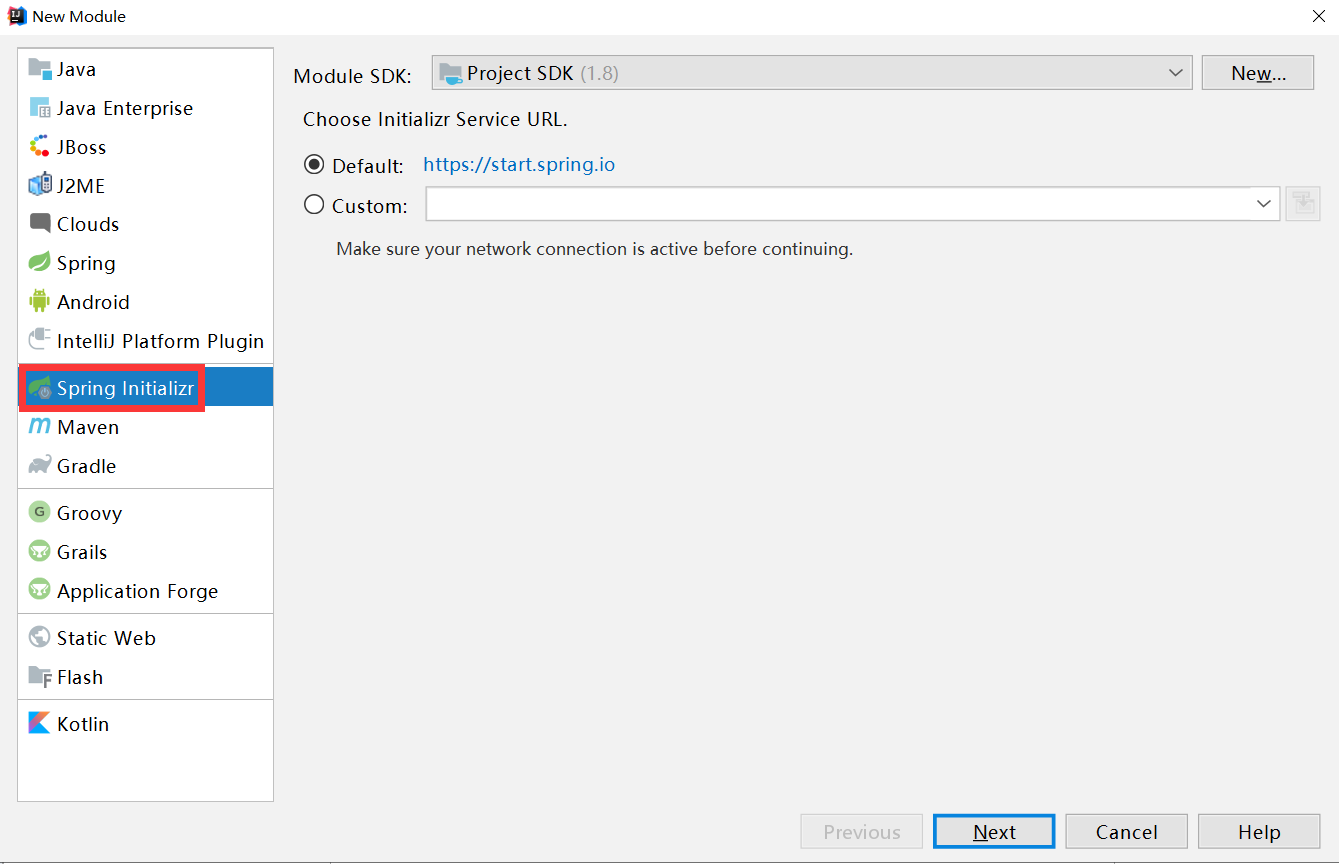
## 概述

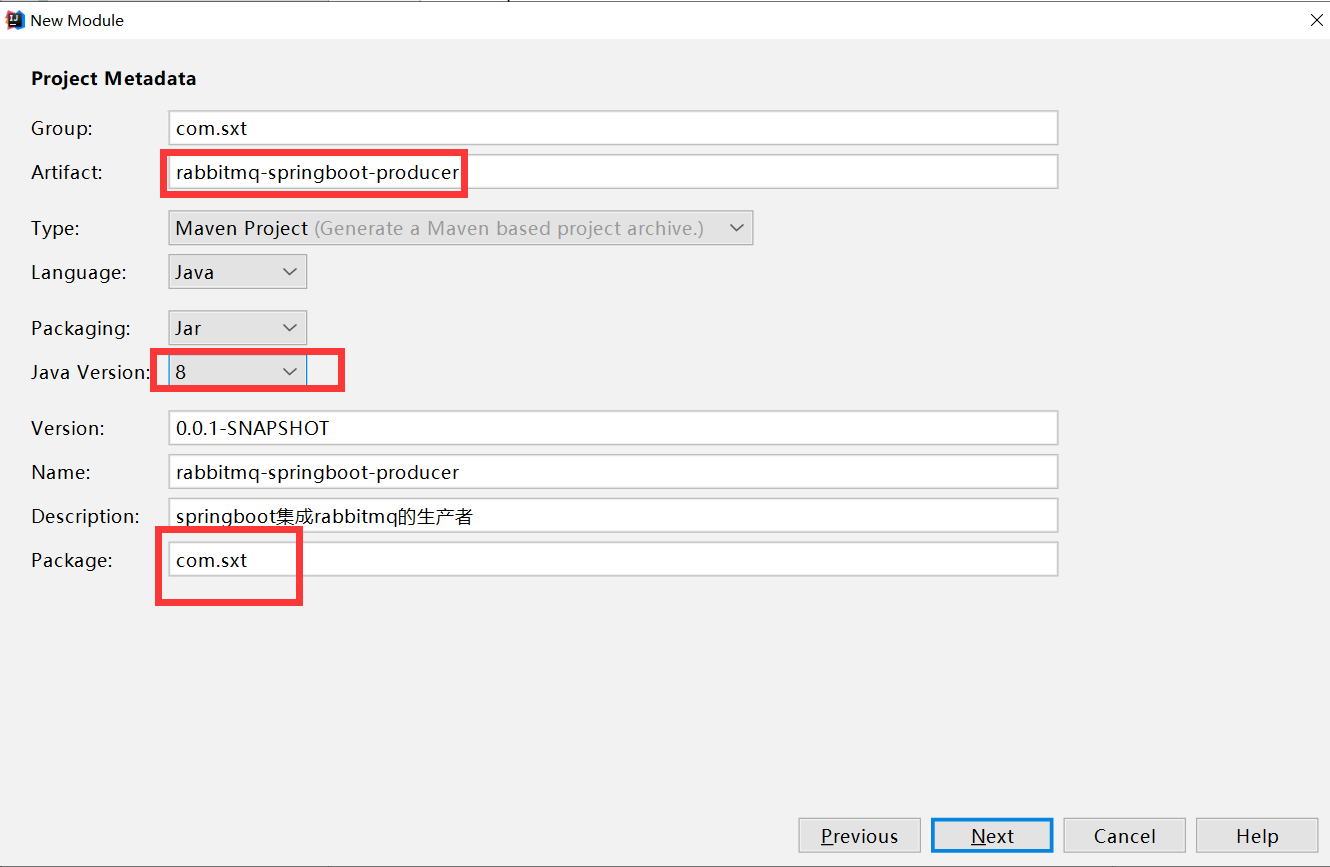
在实际开发中，我们都是使用springboot的技术栈来完成RabbitMQ的开发

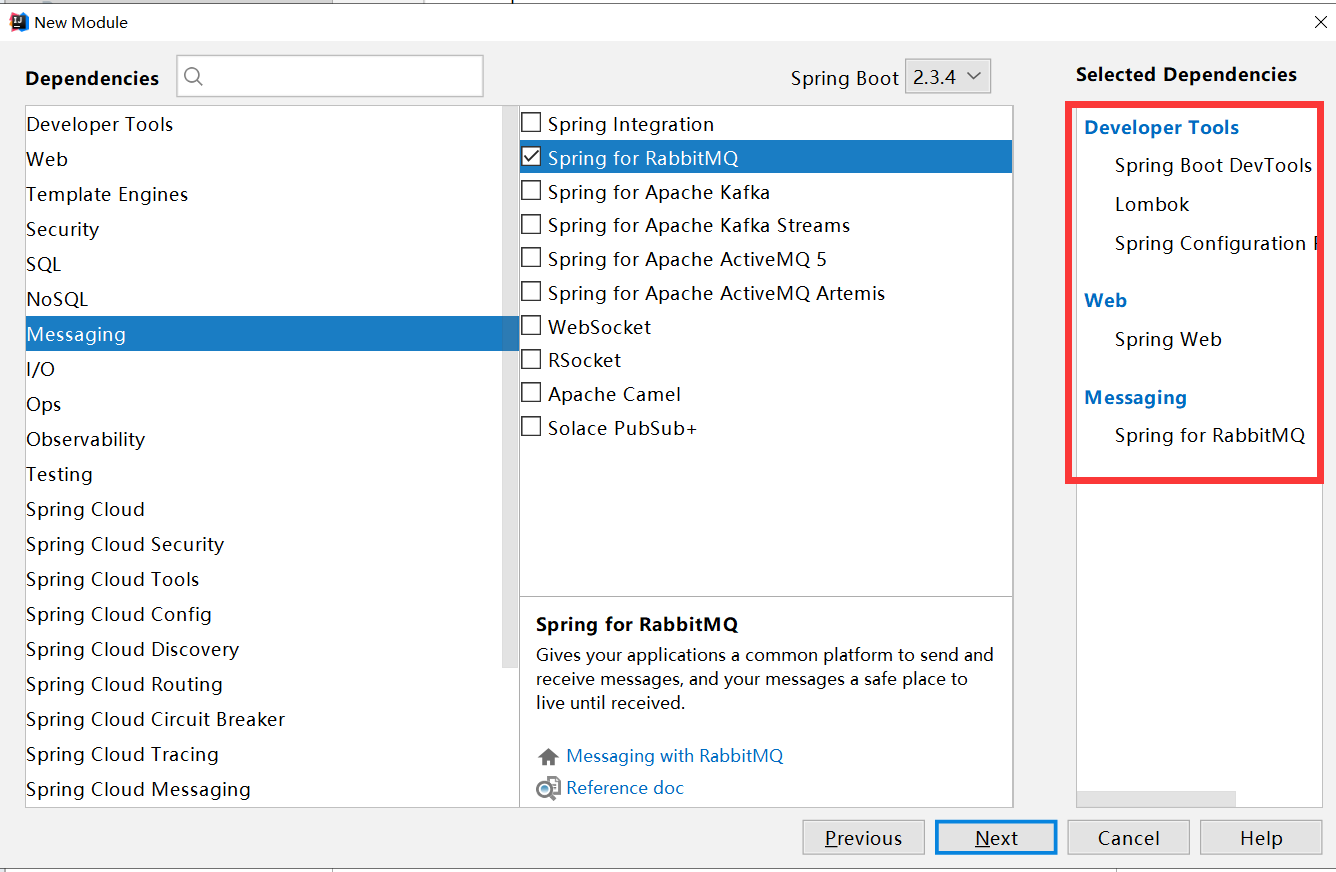
这个测试案例中，我们使用两个项目，一个生产者的项目，一个消费者的项目

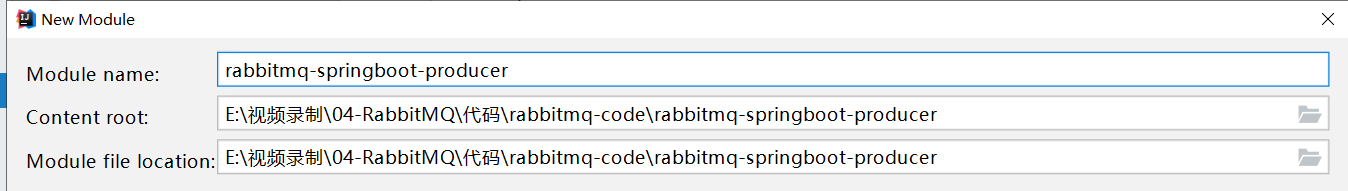
## 生产者准备工作

### 创建项目rabbitmq-springboot-producer









### 导入依赖

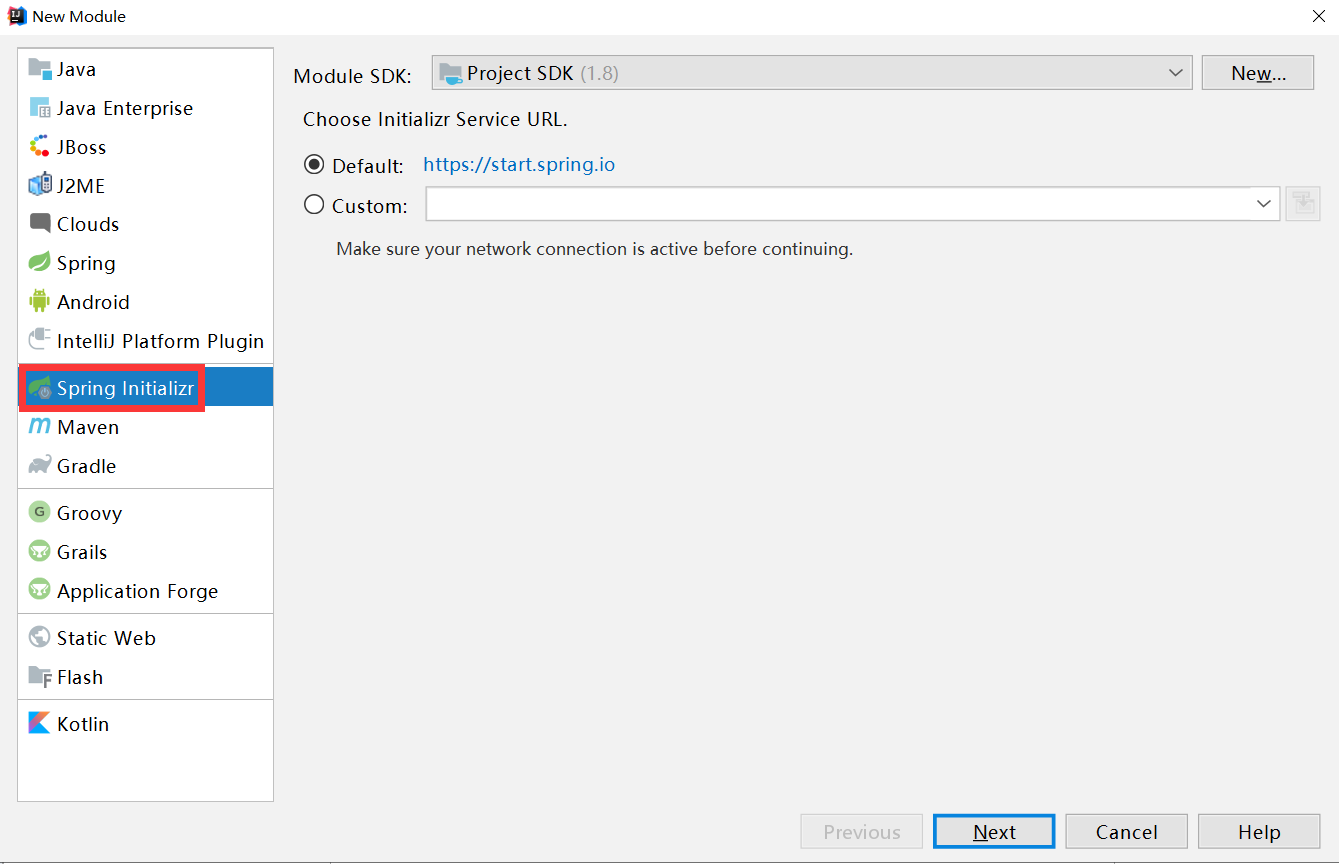
|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-amqp</**artifactId**> </**dependency**> |

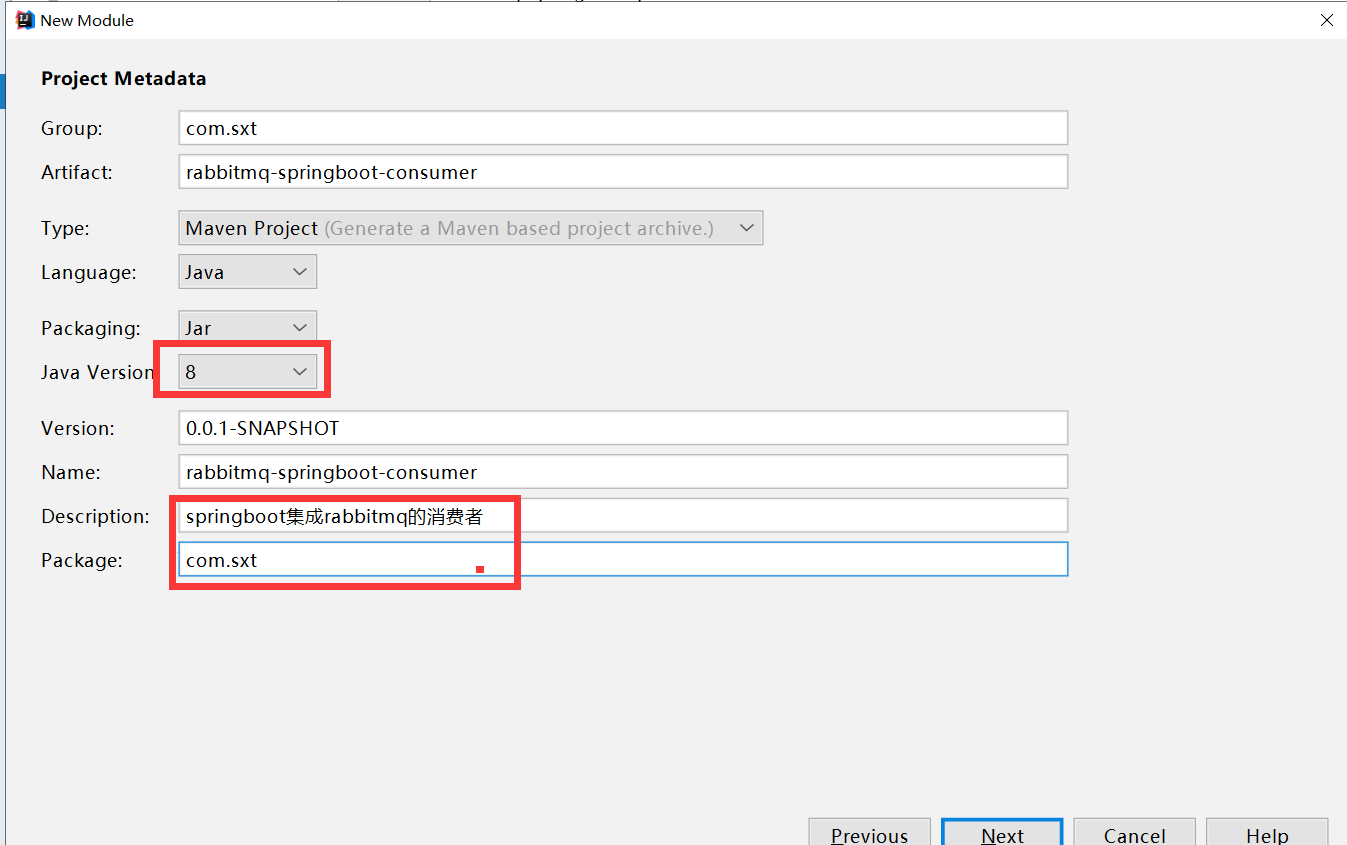
### 修改yml

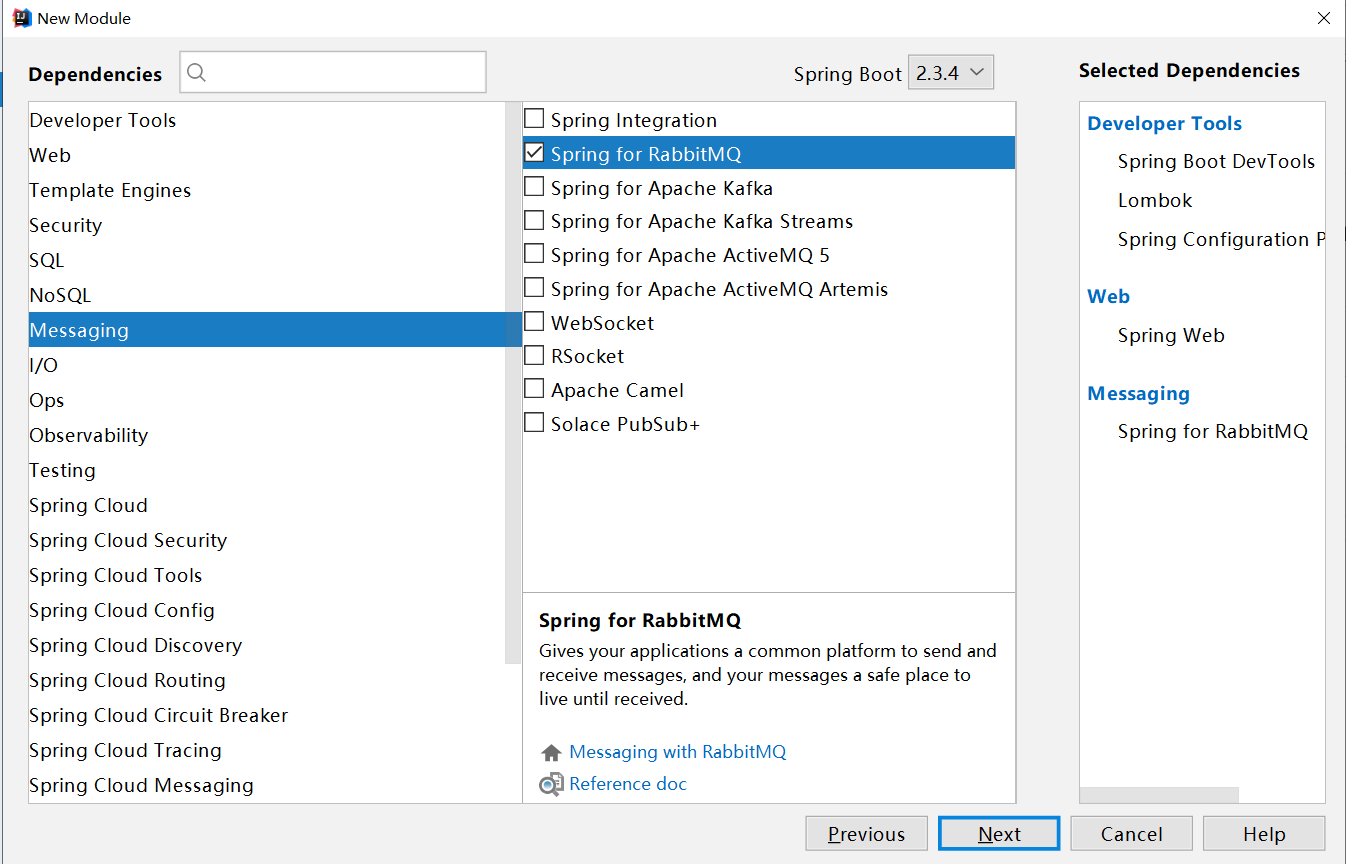
|  |
| --- |
| **server**:  **port**: 8001 **spring**:  **application**:  **name**: producer  **rabbitmq**:  **host**: 81.68.123.67  **port**: 5672  **username**: sxt  **password**: 123456  **virtual-host**: v-sxt |

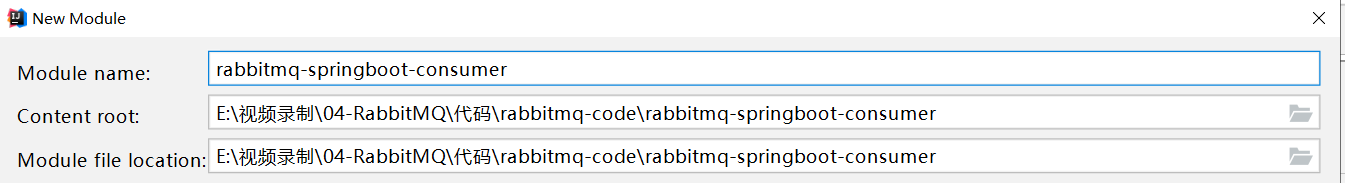
## 消费者准备工作

### 创建项目rabbitmq-springboot-consumer









### 导入依赖

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

### 修改yml

|  |
| --- |
| **server**:  **port**: 8002 **spring**:  **application**:  **name**: producer  **rabbitmq**:  **host**: 81.68.123.67  **port**: 5672  **username**: sxt  **password**: 123456  **virtual-host**: v-sxt |

## 第一种模型(直连)

### 生产者配置

#### 生产者消费发送

|  |
| --- |
| */\*\*  \* 第一种模型的消息发送  \*/* @Test **void** testHello(){  **rabbitTemplate**.convertAndSend(**"hello"**,**"hello world"**);  System.***out***.println(**"消息发成功"**); } |

#### 生产者配置类HelloConfig

|  |
| --- |
| **package** com.sxt.config;  **import** org.springframework.amqp.core.Queue; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/22:21  \** ***@Description:*** *\*/* @Configuration **public class** HelloConfig {  */\*\*  \* 创建一个队列  \*/* @Bean  **public** Queue hello(){  *//这里面可以和之前的Hello项目里面一样，进行5个参数的配置* Queue hello = **new** Queue(**"hello"**);  **return** hello;  } } |

### 消费者配置

#### 消费者接收消息两种方式都可以

|  |
| --- |
| @Component @RabbitListener(queuesToDeclare = {@Queue(**"hello"**)}) **public class** HelloConsumer {   @RabbitHandler  **public void** receive1(String message){  System.***out***.println(**"接收到消息:"**+message);  }  } |
| @Component **public class** HelloConsumer {   @RabbitListener(queuesToDeclare = {@Queue(**"hello"**)})  **public void** receive1(String message){  System.***out***.println(**"接收到消息:"**+message);  }  } |

### 启动测试

先使用生产者发消息，再让消费者上线，再使用生产者发消息

## 第二种模型(Workquene)

### 生产者配置

#### 生产者消费发送

|  |
| --- |
| */\*\*  \* 第二模型的消息发送  \*/* @Test **void** testWork(){  **for** (**int** i = 1; i <=10 ; i++) {  **rabbitTemplate**.convertAndSend(**"work"**,**"hello work----"**+i);  }  System.***out***.println(**"消息发成功"**); } |

#### 生产者配置类WorkConfig

|  |
| --- |
| @Configuration **public class** WorkConfig {   @Bean  **public** Queue work(){  **return new** Queue(**"work"**);  } } |

### 消费者配置

#### 消费者接收消息WorkConsumer

|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/1:08  \** ***@Description:*** *\*/* @Component **public class** WorkConsumer {   */\*  第一个消费者  \*/* @RabbitListener(queuesToDeclare = {@Queue(**"work"**)})  **public void** receive1(String message){  System.***out***.println(**"接收到消息【1】:"**+message);  }   */\*  \*第二个消费者  \*/* @RabbitListener(queuesToDeclare = {@Queue(**"work"**)})  **public void** receive2(String message){  System.***out***.println(**"接收到消息【2】:"**+message);  } } |

### 启动测试

先使用生产者发消息，再让消费者上线

## 第三种模型(fanout)广播

### 生产者配置

#### 生产者消费发送

|  |
| --- |
| */\*\*  \* 第三种模型(fanout)广播  \*/* @Test **public void** testFanout(){  **rabbitTemplate**.convertAndSend(**"logs"**,**""**,**"这是日志广播"**);  System.***out***.println(**"消费发送成功"**); } |

#### 生产者配置类FanoutConfig

|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/22:58  \** ***@Description:*** *\*/* @Configuration **public class** FanoutConfig {   */\*\*  \* 声明交换机  \*/* @Bean  **public** FanoutExchange fanoutExchange(){  FanoutExchange fanoutExchange=**new** FanoutExchange(**"logs"**);  **return** fanoutExchange;  }   */\*\*  \* 声明队列  \*/* @Bean  **public** Queue fanoutQueue1(){  Queue queue=**new** Queue(**"fanout\_queue1"**);  **return** queue;  }   */\*\*  \* 声明队列  \*/* @Bean  **public** Queue fanoutQueue2(){  Queue queue=**new** Queue(**"fanout\_queue2"**);  **return** queue;  }   */\*\*  \*把fanout\_queue1队列绑定到交换机  \*/* @Bean  **public** Binding bindingQ1(){  Binding binding= BindingBuilder.*bind*(fanoutQueue1())  .to(fanoutExchange());  **return** binding;  }   */\*\*  \*把fanout\_queue2队列绑定到交换机  \*/* @Bean  **public** Binding bindingQ2(){  Binding binding= BindingBuilder.*bind*(fanoutQueue2())  .to(fanoutExchange());  **return** binding;  } } |

### 消费者配置

#### 消费者接收消息FanoutCustomer

|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/22:24  \** ***@Description:*** *\*/* @Component **public class** FanoutCustomer {   @RabbitListener(bindings = @QueueBinding( *//绑定队表和交换机* value = @Queue(**"fanout\_queue1"**),*//绑定队列* exchange = @Exchange(name=**"logs"**,type = ExchangeTypes.***FANOUT***)*//绑定交换机和声明交换机类型* ))  **public void** receive1(String message){  System.***out***.println(**"receive1 = "** + message);  }  @RabbitListener(bindings = @QueueBinding(  value = @Queue(**"fanout\_queue2"**),  exchange = @Exchange(name=**"logs"**,type = ExchangeTypes.***FANOUT***)  ))  **public void** receive2(String message){  System.***out***.println(**"receive2 = "** + message);  } } |

### 启动测试

先启动消费者，再让使用生产者发消息，结果是两个方法都接到到消息

## 第四种模型(routing-Direct)

### 生产者配置

#### 生产者消费发送

|  |
| --- |
| */\*\*  \* 第四种模型(routing-Direct)  \*/* @Test **public void** testRoutingDirect() {  **rabbitTemplate**.convertAndSend(**"directs"**, **"info"**, **"info 的日志信息"**);  **rabbitTemplate**.convertAndSend(**"directs"**, **"warm"**, **"warm 的日志信息"**);  **rabbitTemplate**.convertAndSend(**"directs"**, **"debug"**, **"debug 的日志信息"**);  **rabbitTemplate**.convertAndSend(**"directs"**, **"error"**, **"error 的日志信息"**);  System.***out***.println(**"消息发送成功"**); } |

#### 生产者配置类

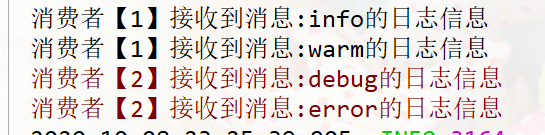
|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/23:27  \** ***@Description:*** *\*/* @Configuration **public class** RoutingDirectConfig {   */\*\*  \* 声明交换机  \*/* @Bean  **public** DirectExchange directExchange(){  DirectExchange directExchange=**new** DirectExchange(**"directs"**);  **return** directExchange;  }   */\*\*  \* 声明队列1绑定info 和warm  \*/* @Bean  **public** Queue queue1(){  **return new** Queue(**"queue1"**);  }   */\*\*  \* 声明队列2  \*/* @Bean  **public** Queue queue2(){  **return new** Queue(**"queue2"**);  }   */\*\*  \* 把队列1绑定到交换机里面指定info的路由key  \*/* @Bean  **public** Binding binding11(){  Binding binding= BindingBuilder.*bind*(queue1())  .to(directExchange()).with(**"info"**);  **return** binding;  }  */\*\*  \* 把队列1绑定到交换机里面指定warm的路由key  \*/* @Bean  **public** Binding binding12(){  Binding binding= BindingBuilder.*bind*(queue1())  .to(directExchange()).with(**"warm"**);  **return** binding;  }   */\*\*  \* 把队列2绑定到交换机里面指定debug的路由key  \*/* @Bean  **public** Binding binding21(){  Binding binding= BindingBuilder.*bind*(queue2())  .to(directExchange()).with(**"debug"**);  **return** binding;  }  */\*\*  \* 把队列2绑定到交换机里面指定error的路由key  \*/* @Bean  **public** Binding binding22(){  Binding binding= BindingBuilder.*bind*(queue2())  .to(directExchange()).with(**"error"**);  **return** binding;  }  } |

### 消费者配置

#### 消费者接收消息RoutingDirectConsumer

|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/23:32  \** ***@Description:*** *\*/* @Component **public class** RoutingDirectConsumer {   @RabbitListener(bindings ={  @QueueBinding(  value = @Queue(**"queue1"**),  key={**"info"**,**"warm"**},  exchange = @Exchange(name=**"directs"**,type = ExchangeTypes.***DIRECT***)  )})  **public void** receive1(String message){  System.***out***.println(**"消费者【1】接收到消息："** + message);  }  @RabbitListener(bindings ={  @QueueBinding(  value =@Queue(**"queue2"**),  key={**"debug"**,**"error"**},  exchange = @Exchange(name=**"directs"**,type = ExchangeTypes.***DIRECT***)  )})  **public void** receive2(String message){  System.***err***.println(**"消费者【2】接收到消息："** + message);  }  } |

### 启动测试



## 第五种模型(routing-Topic)

### 生产者配置

#### 生产者消费发送

|  |
| --- |
| */\*\*  \* 第五种模型(routing-Topic)  \*/* @Test **public void** testTopic(){  **rabbitTemplate**.convertAndSend(**"topics"**,**"user.save"**,**"user.save 的消息"**);  **rabbitTemplate**.convertAndSend(**"topics"**,**"user.save.findAll"**,**"user.save.findAll 的消息"**);  **rabbitTemplate**.convertAndSend(**"topics"**,**"user"**,**"user 的消息"**); } |

#### 生产者配置类RoutingTopicConfig

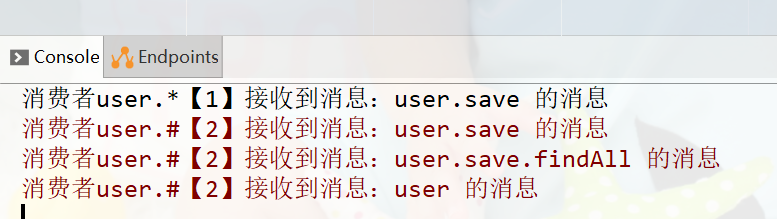
|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/23:27  \** ***@Description:*** *\*/* @Configuration **public class** RoutingTopicConfig {   */\*\*  \* 声明交换机  \*/* @Bean  **public** TopicExchange topicExchange(){  TopicExchange topicExchange=**new** TopicExchange(**"topics"**);  **return** topicExchange;  }   */\*\*  \* 声明队列1绑定info 和warm  \*/* @Bean  **public** Queue topicQueue1(){  **return new** Queue(**"topicQueue1"**);  }   */\*\*  \* 声明队列2  \*/* @Bean  **public** Queue topicQueue2(){  **return new** Queue(**"topicQueue2"**);  }   */\*\*  \* 把队列1绑定到交换机里面指定user.\*的路由key  \*/* @Bean  **public** Binding binding111(){  Binding binding= BindingBuilder.*bind*(topicQueue1())  .to(topicExchange()).with(**"user.\*"**);  **return** binding;  }  */\*\*  \* 把队列2绑定到交换机里面指定user.#的路由key  \*/* @Bean  **public** Binding binding222(){  Binding binding= BindingBuilder.*bind*(topicQueue2())  .to(topicExchange()).with(**"user.#"**);  **return** binding;  }  } |

### 消费者配置

#### 消费者接收消息RoutingTopicConsumer

|  |
| --- |
| */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/07/23:32  \** ***@Description:*** *\*/* @Component **public class** RoutingTopicConsumer {   @RabbitListener(bindings ={  @QueueBinding(  value = @Queue(**"topicQueue1"**),  key={**"user.\*"**},  exchange = @Exchange(name=**"topics"**,type = ExchangeTypes.***TOPIC***)  )})  **public void** receive1(String message){  System.***out***.println(**"消费者user.\*【1】接收到消息："** + message);  }  @RabbitListener(bindings ={  @QueueBinding(  value =@Queue(**"topicQueue2"**),  key={**"user.#"**},  exchange = @Exchange(name=**"topics"**,type = ExchangeTypes.***TOPIC***)  )})  **public void** receive2(String message){  System.***err***.println(**"消费者user.#【2】接收到消息："** + message);  }  } |

### 启动测试



# **【掌握】boot中发送消息的监视**

## 概述及准备

    刚才我们发送消息，不管成功还是失败，都不报错，结果看效果时，发现有的没有发进去，那么如何知道消息是否发送成功呢，RabbitMQ提供了一个消费监视功能

注意：RabbitMQ发送消息分为2 个阶段

* 消息发送到交互机里面，可以监视
* 消息由交互机到队列里面，也可以监视

### 修改yaml

|  |
| --- |
| **server**:  **port**: 8080 **spring**:  **application**:  **name**: springboot\_rabbitmq  **rabbitmq**:  **host**: 81.68.123.67  **port**: 5672  **username**: sxt  **password**: 123456  **virtual-host**: /v-sxt  **publisher-confirm-type**: *correlated #开启消息到达交互机的确认机制* **publisher-returns**: **true** *#消息由交互机到达队列时失败触发* |

## 消息到交互机和未到达队列里面的监视

|  |
| --- |
| */\*\*  \* 第三种模型的消息发送 fanout  \*/* @Test **void** testFanout(){  **rabbitTemplate**.setConfirmCallback(**new** ConfirmCallback() {  */\*\*  \* 当消息到达交换机之后该方法会被回调  \** ***@param correlationData*** *相关的数据  \** ***@param ack*** *交换机接收消息是否成功  \** ***@param cause*** *如果没有成功，返回原因  \*/* @Override  **public void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  **if**(ack){  System.***out***.println(**"交换机接收消息成功"**);  }**else**{  System.***out***.println(**"交换机接收消息失败：原因:"**+cause);  }  }  });  **rabbitTemplate**.setReturnCallback(**new** ReturnCallback() {  */\*\*  \* 消息如未正常到达队列里面会回调  \** ***@param message*** *消息内容  \** ***@param replyCode*** *回调的响应码  \** ***@param replyText*** *响应文本  \** ***@param exchange*** *交换机  \** ***@param routingKey*** *路由key  \*/* @Override  **public void** returnedMessage(Message message, **int** replyCode, String replyText, String exchange,  String routingKey) {  System.***out***.println(message);  System.***out***.println(**"消息未正常到达队列-replyCode:"**+replyCode);  System.***out***.println(**"消息未正常到达队列-replyText:"**+replyText);  System.***out***.println(**"消息未正常到达队列-exchange:"**+exchange);  System.***out***.println(**"消息未正常到达队列-routingKey:"**+routingKey);  }  });  **rabbitTemplate**.convertAndSend(**"logs"**,**""**,**"这是一个fanout的日志消息"**);  System.***out***.println(**"消息发送成功"**);  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  } } |

## 测试

先去掉交换机的声明 队列的声明，队列和交换机的绑定【注释FantoutConfig里面的所有内容】

发消息测试

打开交换机的声明

发消息测试

打开队列的声明

发消息测试

打开队列和交换机绑定的声明

发消息测试

## 优化消息监视和到达队列的监视

|  |
| --- |
| **package** com.sxt.config;  **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.connection.CorrelationData; **import** org.springframework.amqp.rabbit.core.RabbitTemplate; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Component; **import** javax.annotation.PostConstruct; */\*\*  \** ***@Author:*** *尚学堂 雷哥  \*/* @Component **public class** WatchMessageImpl **implements** RabbitTemplate.ConfirmCallback,RabbitTemplate.ReturnCallback {   @Autowired  **private** RabbitTemplate **rabbitTemplate**;   @PostConstruct  **public void** initRabbitTemplate(){  **rabbitTemplate**.setConfirmCallback(**this**);  **rabbitTemplate**.setReturnCallback(**this**);  }   */\*\*  \* 消息到达交换机的回调  \** ***@param correlationData*** *\** ***@param ack*** *是否到达交换机  \** ***@param cause*** *\*/* @Override  **public void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  **if**(ack){  System.***out***.println(**"消息正常到达交换机"**);  }**else**{  System.***out***.println(**"消息没有到达交换机原因："**+cause);  }  }   @Override  **public void** returnedMessage(Message message, **int** replyCode, String replyText, String exchange, String routingKey) {  System.***out***.println(**new** String(message.getBody())+**"到达队列失败"**+replyCode+**" "**+replyText+**" 交换机为:"**+exchange+**" 路由key:"**+routingKey);  *//处理重新发送的问题* } } |

## 消息转换参数的问题



# **【掌握】SpringBoot中消息的消费及签收机制**

## 消息的签收机制说明

消息消费成功后，我们在客户端签收后，消息就从MQ 服务器里面删除了

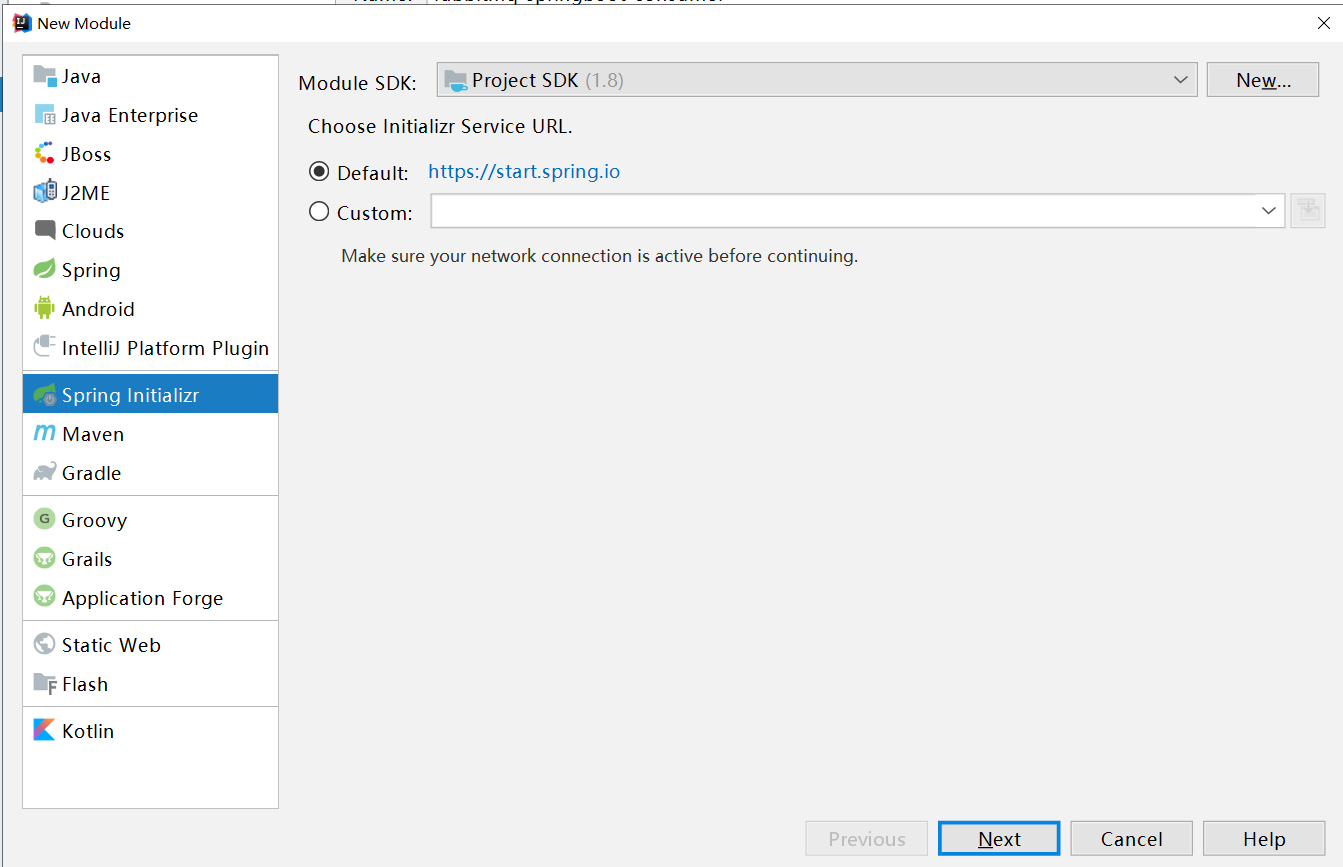
若消息没有消费成功，我们让他回到MQ 里面，让别人再次重试消费

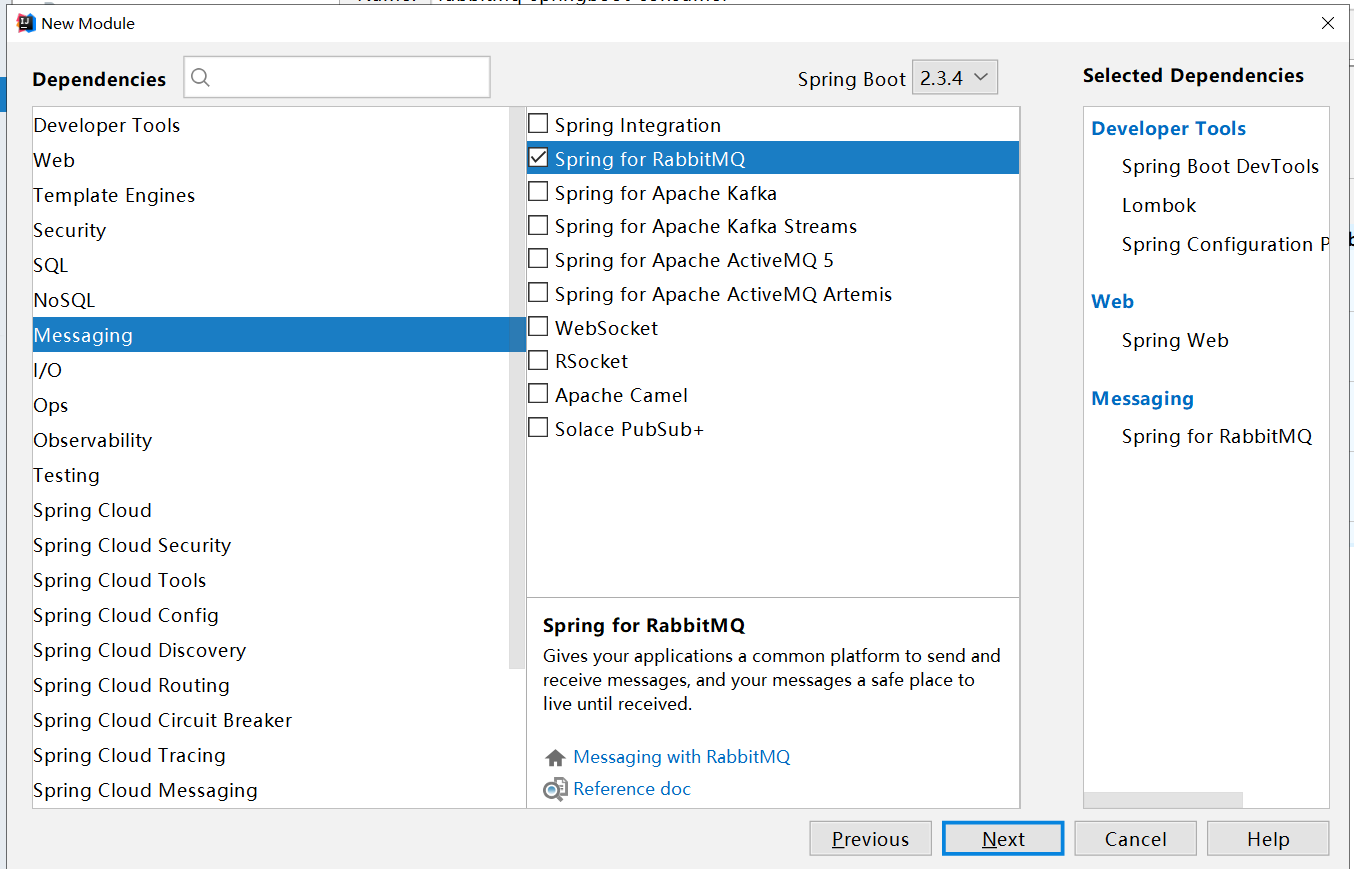
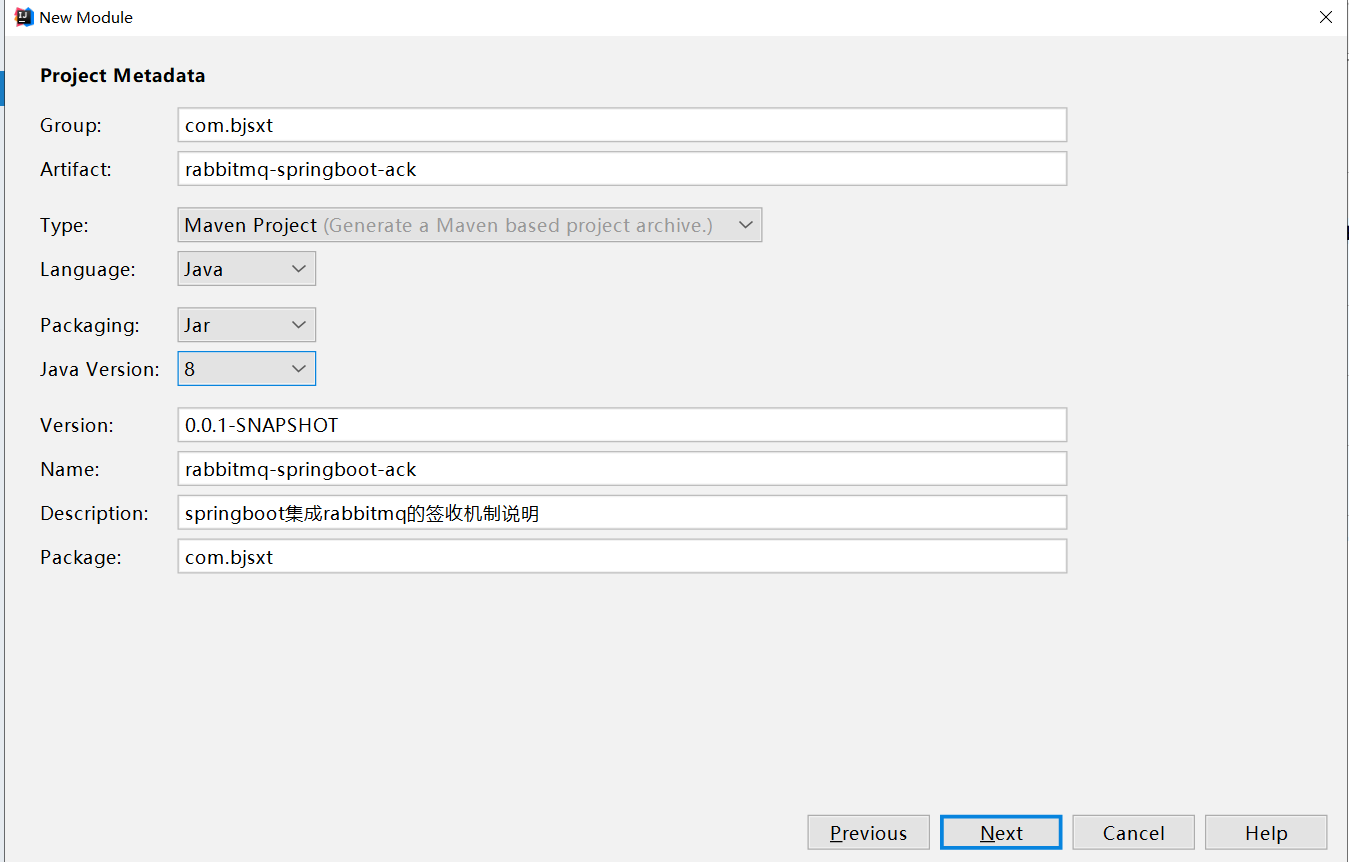
## 自动签收

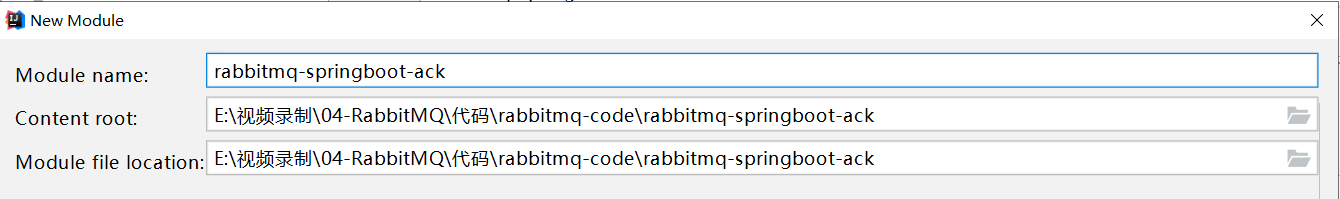
消息只要被客户端接收到，无论你客户端发生了什么，我们服务器都不管你了，直接把你删除了，这是它是默认的行为

## 手动签收

### 创建项目rabbitmq-springboot-ack



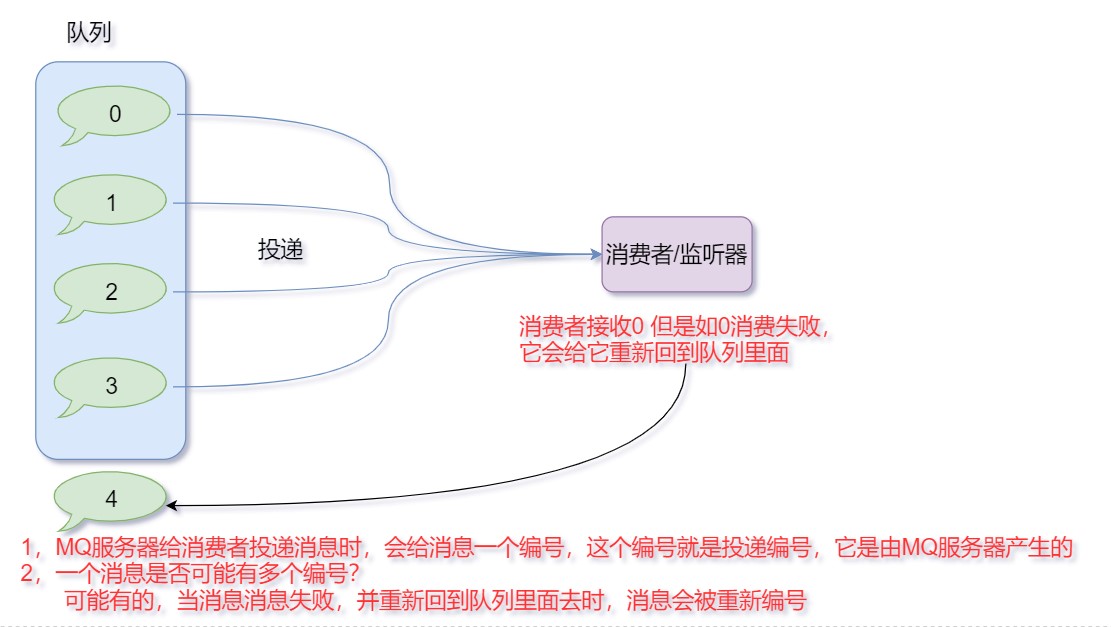




### 创建yml配置文件

|  |
| --- |
| **server**:  **port**: 8001 **spring**:  **application**:  **name**: producer  **rabbitmq**:  **host**: 81.68.123.67  **port**: 5672  **username**: sxt  **password**: 123456  **virtual-host**: /v-sxt  *# publisher-confirms: true 老版本里面的用法* **publisher-confirm-type**: *correlated #开启消息到达交换机的确认机制* **publisher-returns**: **true** *#消息由交互机到达队列时失败触发* **listener**:  **simple**:  **acknowledge-mode**: *manual #手动签收  #acknowledge-mode: auto #自动签收 这个是默认行为* **direct**:  **acknowledge-mode**: *manual #设置直连交互机的签收类型* |

### 消息投送的ID的说明【重点】



### 怎么获取投递的ID【重点】

|  |
| --- |
| **package** com.bjsxt.receive;  **import** com.rabbitmq.client.Channel; **import** java.io.IOException; **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.annotation.Exchange; **import** org.springframework.amqp.rabbit.annotation.Queue; **import** org.springframework.amqp.rabbit.annotation.QueueBinding; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/14:49  \** ***@Description:*** *\*/* @Component **public class** MessageReceive1 {  @RabbitListener(bindings = {  @QueueBinding(value = @Queue(**"queue"**),  key = **"error"**,  exchange = @Exchange(value = **"directs"**))*//默认的直连交换机* })  **public void** receiveMessage(String content, Message message, Channel channel) **throws** IOException {  System.***out***.println(**"消费者收到消息-内容为:"**+content);  System.***out***.println(**"消费者收到消息-消息对象:"**+message);  System.***out***.println(**"消费者收到消息-信道:"**+channel);  **long** deliveryTag = message.getMessageProperties().getDeliveryTag();*//消息投递ID* System.***out***.println(**"消息投递ID:"**+deliveryTag);  String messageId = message.getMessageProperties().getMessageId();  System.***out***.println(**"消息自定义ID:"**+messageId);  */\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \*/* channel.basicAck(deliveryTag,**false**);  System.***out***.println(**"消息签收成功"**);  System.***out***.println(**"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"**);  }  } |

### 投递ID存在的问题及消息的永久ID设置的问题【重点】

什么能代表消息的唯一的标识，显然投送的id 不行，因为一个消息可能会有多个投送的id，我们就需要给消息一个唯一的值，这个伴随消息终身，不会变化！

我们需要发生消息时，给消息设置一个Id，然后保证该Id唯一就可以！

|  |
| --- |
| @Test  **public void** sendMessage() **throws** Exception {  **for** (**int** i = 1; i <=5; i++) { *// rabbitTemplate.convertAndSend("directs","error","我是一个日志消息-"+i);* **rabbitTemplate**.convertAndSend(**"directs"**, **"error"**, **"我是一个日志消息-"** + i,  **new** MessagePostProcessor() {  @Override  **public** Message postProcessMessage(Message message) **throws** AmqpException {  *//自己给消息设置自定义的ID* String messageId= UUID.*randomUUID*().toString().replace(**"-"**,**""**);  message.getMessageProperties().setMessageId(messageId);  **return** message;  }  });  }  System.***out***.println(**"消息发送成功"**);  System.***in***.read();  } |

### 关于批量的签收

若我们此时签收4 了，但是前面0，1，2，3 都没有签收，则MQ 若是批量的签收，它会把0，1，2，3 都签收，因为MQ 认为，比他晚投递的已经签收，前面的肯定已经消费成功了

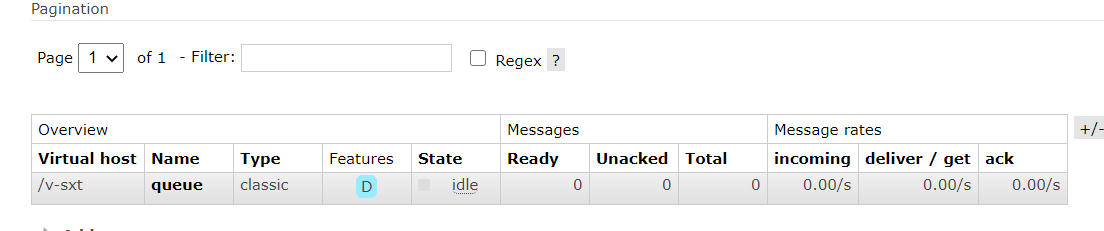
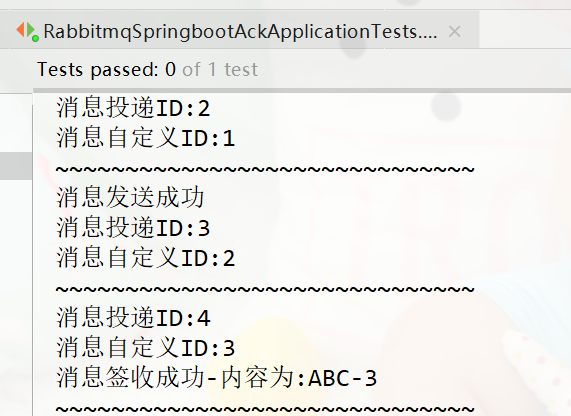
发送者

|  |
| --- |
| **static int** *a*=1; @Test **public void** sendMessage2() **throws** Exception {  **for** (**int** i = 1; i <=3; i++) {  **rabbitTemplate**.convertAndSend(**"directs"**, **"error"**, **"ABC-"** + i,  **new** MessagePostProcessor() {  @Override  **public** Message postProcessMessage(Message message) **throws** AmqpException {  *//自己给消息设置自定义的ID*  message.getMessageProperties().setMessageId((*a*++)+**""**);  **return** message;  }  });  }  System.***out***.println(**"消息发送成功"**);  System.***in***.read(); } |

消费者

|  |
| --- |
| **package** com.bjsxt.receive;  **import** com.rabbitmq.client.Channel; **import** java.io.IOException; **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.annotation.Exchange; **import** org.springframework.amqp.rabbit.annotation.Queue; **import** org.springframework.amqp.rabbit.annotation.QueueBinding; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/14:49  \** ***@Description:*** *\*/* @Component **public class** MessageReceive2 {  @RabbitListener(bindings = {  @QueueBinding(value = @Queue(**"queue"**),  key = **"error"**,  exchange = @Exchange(value = **"directs"**))*//默认的直连交换机* })  **public void** receiveMessage(String content, Message message, Channel channel) **throws** IOException {  **long** deliveryTag = message.getMessageProperties().getDeliveryTag();*//消息投递ID* System.***out***.println(**"消息投递ID:"**+deliveryTag);  String messageId = message.getMessageProperties().getMessageId();  System.***out***.println(**"消息自定义ID:"**+messageId);  **if**(content.equals(**"ABC-3"**)){  */\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \*/* channel.basicAck(deliveryTag,**true**);  System.***out***.println(**"消息签收成功-内容为:"**+content);  }  System.***out***.println(**"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"**);  }  } |

效果



可以发现只签收了ABC-3 但是队列里面没有消息了，说明前面的12都被批量签收了

## 不签收

当我们认为消息不合格时，或不是我们要的消息时，我们可以选择不签收它

### 生产者

|  |
| --- |
| @Test **public void** sendMessage2() **throws** Exception {  **rabbitTemplate**.convertAndSend(**"directs"**, **"error"**, **"1234567"**,  **new** MessagePostProcessor() {  @Override  **public** Message postProcessMessage(Message message) **throws** AmqpException {  *//自己给消息设置自定义的ID* String messageId= UUID.*randomUUID*().toString().replace(**"-"**,**""**);  message.getMessageProperties().setMessageId(messageId);  **return** message;  }  });  System.***out***.println(**"消息发送成功"**);  System.***in***.read(); } |

### 消费者

|  |
| --- |
| **package** com.bjsxt.receive;  **import** com.rabbitmq.client.Channel; **import** java.io.IOException; **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.annotation.Exchange; **import** org.springframework.amqp.rabbit.annotation.Queue; **import** org.springframework.amqp.rabbit.annotation.QueueBinding; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/14:49  \** ***@Description:*** *\*/* @Component **public class** MessageReceive3 {  @RabbitListener(bindings = {  @QueueBinding(value = @Queue(**"queue"**),  key = **"error"**,  exchange = @Exchange(value = **"directs"**))*//默认的直连交换机* })  **public void** receiveMessage(String content, Message message, Channel channel) **throws** IOException {  **long** deliveryTag = message.getMessageProperties().getDeliveryTag();*//消息投递ID* System.***out***.println(**"消息投递ID:"**+deliveryTag);  String messageId = message.getMessageProperties().getMessageId();  System.***out***.println(**"消息自定义ID:"**+messageId);  **if**(content.equals(**"123456"**)){*//如果消息内容为123456就签收它  /\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \*/* channel.basicAck(deliveryTag,**false**);  System.***out***.println(**"消息签收成功"**);  }**else**{  *//如果不是123456就拒绝签收  /\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \* requeue: true 代表拒绝签收并把消息重新放回到队列里面 false就直接拒绝  \*/* channel.basicNack(deliveryTag,**false**,**true**);  System.***out***.println(**"消息被拒绝签收"**);  }  System.***out***.println(**"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"**);  } } |

我们选择不签收，其实是为了保护消息，当消费消息发送异常时，我们可以把消息放在队列里面，让它重新投递，重新让别人消费！而不是丢了它！

## 不去签收消息的死循环怎么解决

不签收，并且让它回到队列里面，想法很好，但是很容易造成死循环，因为没有任何人能消费她！

我们设计一个机制，当一个消息被消费3次还没有消费成功，我们就直接把它记录下来，人工处理！

消息消费3次（消息的标识，消息的计数）

我们引入Redis ，使用Redis 计数，若超过3次，直接拒绝消息，并且不回到队列里面

### 引入redis并使用docker运行

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**> </**dependency**> |

docker run -d --name myredis -p 6390:6379 redis --requirepass "123456"

### 配置文件

|  |
| --- |
| *#redis的配置* **redis**:  **host**: 81.68.123.67  **port**: 6390  **password**: 123456 |

### 代码

|  |
| --- |
| **package** com.bjsxt.receive;  **import** com.rabbitmq.client.Channel; **import** java.io.IOException; **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.annotation.Exchange; **import** org.springframework.amqp.rabbit.annotation.Queue; **import** org.springframework.amqp.rabbit.annotation.QueueBinding; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.data.redis.core.StringRedisTemplate; **import** org.springframework.stereotype.Component;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/14:49  \** ***@Description:*** *\*/* @Component **public class** MessageReceive3 {   @Autowired  **private** StringRedisTemplate **redisTemplate**;   *//消息的前缀* **private** String **MESSAGE**=**"message:"**;   @RabbitListener(bindings = {  @QueueBinding(value = @Queue(**"queue"**),  key = **"error"**,  exchange = @Exchange(value = **"directs"**))*//默认的直连交换机* })  **public void** receiveMessage(String content, Message message, Channel channel) **throws** IOException {  **long** deliveryTag = message.getMessageProperties().getDeliveryTag();*//消息投递ID* System.***out***.println(**"消息投递ID:"**+deliveryTag);  String messageId = message.getMessageProperties().getMessageId();  System.***out***.println(**"消息自定义ID:"**+messageId);  **if**(content.equals(**"123456"**)){*//如果消息内容为123456就签收它  /\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \*/* channel.basicAck(deliveryTag,**false**);  System.***out***.println(**"消息签收成功"**);  }**else**{  String count=**redisTemplate**.opsForValue().get(**MESSAGE**+messageId);  **if**(count!=**null**&&Long.*valueOf*(count)>=3){  channel.basicNack(deliveryTag,**false**,**false**);  System.***out***.println(**"该消息消费3失败，我们记录它，人工处理:"**+content);  }**else** {  *//如果不是123456就拒绝签收  /\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \* requeue: true 代表拒绝签收并把消息重新放回到队列里面 false就直接拒绝  \*/  //处理业务逻辑的处理[可能逻辑出现问题]* channel.basicNack(deliveryTag, **false**, **true**);  System.***out***.println(**"消息被拒绝签收"**);  *//因现被拒绝了，我们把消息ID放到redis里面* **redisTemplate**.opsForValue().increment(**MESSAGE** + messageId);  }  }  System.***out***.println(**"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"**);  }  } |

### 测试注意

因为统计计数时，消息的次数，是通过消息的Id 来计数的，我们在发送消息时，要设置消息的头：



# **【掌握】消息的重复消费及处理【面试】**

## 重复消费概述

当消息回退到队列里面后，会被再次消费，但是我们不能让消息消费成功2次

其实,MQ 自己就可以保证消息不被重复消费，因为MQ 可以把消息投递给消费者时，是阻塞的，不会把一个消息投递给多个消费者！

 但是面试时，有人问你，消息怎么保证不被重复消费！

无论在RabbitMQ ,或者Activemq 里面，解决思路都是一样的！！

对于重复消费--> 去重操作--->

接口的幂等操作（->找到该操作的有个唯一标识-> 具体的情况，具体讨论）

Eg: 微信里面，若有人关注我了，我给他发红包，我的红包只能发送一次，我们可以使用用户的openId 做一个去重的操作

订单不重复（订单的编号）（先按订单编号查询，再操作）

数据库不能重复（数据库的id）（先查询是否存在，再进行操作）

总结：去重操作就是找到一个该操作的唯一标识，把该标识，放在一个空间里面，在此操作时，我们可以先判断该空间是否已经操作过它了

## 利用BloomFilter 实现去重的操作

https://www.hutool.cn/docs/#/bloomFilter/%E6%A6%82%E8%BF%B0



### 添加Hutool的依赖

|  |
| --- |
| <**dependency**>  <**groupId**>cn.hutool</**groupId**>  <**artifactId**>hutool-all</**artifactId**>  <**version**>5.1.5</**version**> </**dependency**> |

### 配置BitMapBloomFilter

|  |
| --- |
| **package** com.bjsxt.config;  **import** cn.hutool.bloomfilter.BitMapBloomFilter; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/16:34  \** ***@Description:*** *\*/* @Configuration **public class** BloomFilterConfig {   @Bean  **public** BitMapBloomFilter bitMapBloomFilter(){  **return new** BitMapBloomFilter(Integer.***MAX\_VALUE***);  }  } |

### 使用Hutool -BloomFilter去重

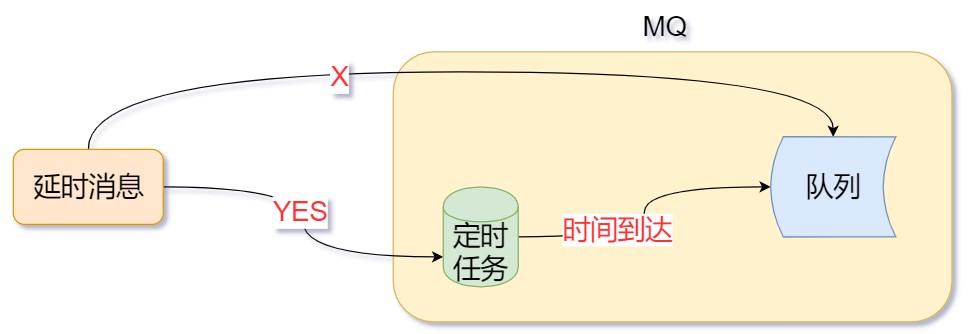
|  |
| --- |
| **package** com.bjsxt.receive;  **import** cn.hutool.bloomfilter.BitMapBloomFilter; **import** com.rabbitmq.client.Channel; **import** java.io.IOException; **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.annotation.Exchange; **import** org.springframework.amqp.rabbit.annotation.Queue; **import** org.springframework.amqp.rabbit.annotation.QueueBinding; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.data.redis.core.StringRedisTemplate; **import** org.springframework.stereotype.Component;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/14:49  \** ***@Description:*** *\*/* @Component **public class** MessageReceive4 {   @Autowired  **private** StringRedisTemplate **redisTemplate**;   @Autowired  **private** BitMapBloomFilter **bitMapBloomFilter**;   *//消息的前缀* **private** String **MESSAGE**=**"message:"**;   @RabbitListener(bindings = {  @QueueBinding(value = @Queue(**"queue"**),  key = **"error"**,  exchange = @Exchange(value = **"directs"**))*//默认的直连交换机* })  **public void** receiveMessage(String content, Message message, Channel channel) **throws** IOException {  **long** deliveryTag = message.getMessageProperties().getDeliveryTag();*//消息投递ID* System.***out***.println(**"消息投递ID:"**+deliveryTag);  String messageId = message.getMessageProperties().getMessageId();  System.***out***.println(**"消息自定义ID:"**+messageId);   **if**(**bitMapBloomFilter**.contains(messageId)){  System.***out***.println(**"这个消息被消费过，不能重复消费"**);  **try** {  *//如果进入到这里面，说明这个消息之前被消费过，但是MQ认为你没有消费，所以我们要签收这条消息* channel.basicAck(deliveryTag,**false**);  **return**;  }**catch** (Exception e){  System.***out***.println(e);  }  }   **if**(content.equals(**"123456"**)){*//如果消息内容为123456就签收它  /\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \*/* channel.basicAck(deliveryTag,**false**);  System.***out***.println(**"消息签收成功"**);  *//消费成功之后放到布隆过滤器里面* **bitMapBloomFilter**.add(messageId);  }**else**{  String count=**redisTemplate**.opsForValue().get(**MESSAGE**+messageId);  **if**(count!=**null**&&Long.*valueOf*(count)>=3){  channel.basicNack(deliveryTag,**false**,**false**);  System.***out***.println(**"该消息消费3失败，我们记录它，人工处理:"**+content);  }**else** {  *//如果不是123456就拒绝签收  /\*\*  \* 参数说明:  \* deliveryTag 消息投递ID，要签收的投递ID是多少  \* multiple:是否批量签收  \* requeue: true 代表拒绝签收并把消息重新放回到队列里面 false就直接拒绝  \*/  //处理业务逻辑的处理[可能逻辑出现问题]* channel.basicNack(deliveryTag, **false**, **true**);  System.***out***.println(**"消息被拒绝签收"**);  *//因现被拒绝了，我们把消息ID放到redis里面* **redisTemplate**.opsForValue().increment(**MESSAGE** + messageId);  }  }  System.***out***.println(**"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"**);  }  } |

### 发消息的代码和上面一样

# **【掌握】延迟消息+死信消息（常考）**

## 什么是延迟消息及使用场景

当消息发送到服务器时，该消息不能直接被放在队列里面，而是在mq服务器里面建立一个定时任务，在服务器到达延时时间后再执行投递的操作



### 延迟消息的使用场景

    淘宝七天自动确认收货。在我们签收商品后，物流系统会在七天后延时发送一个消息给支付系统，通知支付系统将款打给商家，这个过程持续七天，就是使用了消息中间件的延迟推送功能。

    12306 购票支付确认页面。我们在选好票点击确定跳转的页面中往往都会有倒计时，代表着 30 分钟内订单不确认的话将会自动取消订单。其实在下订单那一刻开始购票业务系统就会发送一个延时消息给订单系统，延时30分钟，告诉订单系统订单未完成，如果我们在30分钟内完成了订单，则可以通过逻辑代码判断来忽略掉收到的消息。

在上面两种场景中，如果我们使用下面两种传统解决

1 对于单机版而已，我们的策略有3 种

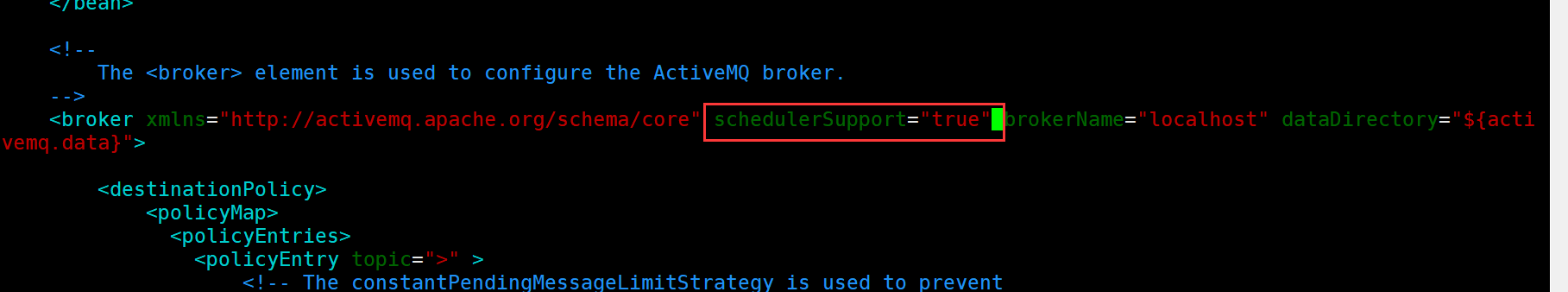
定时删除、定期删除、惰性删除

**2 对于分布式项目而言呢，我们最好选择MQ**

**插件安装：https://www.cnblogs.com/geekdc/p/13549613.html**

## 对于Activemq 实现延迟消息【可以回看雷哥的ActiveMQ】

### 修改配置文件



### 发消息时，指定时间

MessageProducer producer = session.createProducer(destination);

TextMessage message = session.createTextMessage("test msg");

long time = 60 \* 1000;

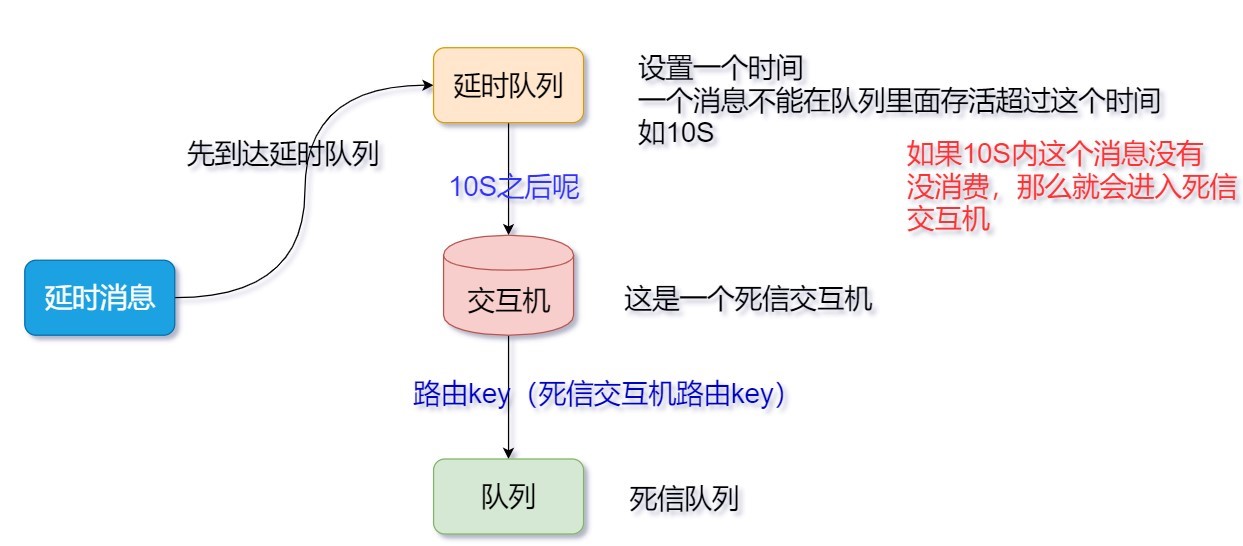
message.setLongProperty(ScheduledMessage.AMQ\_SCHEDULED\_DELAY, time);

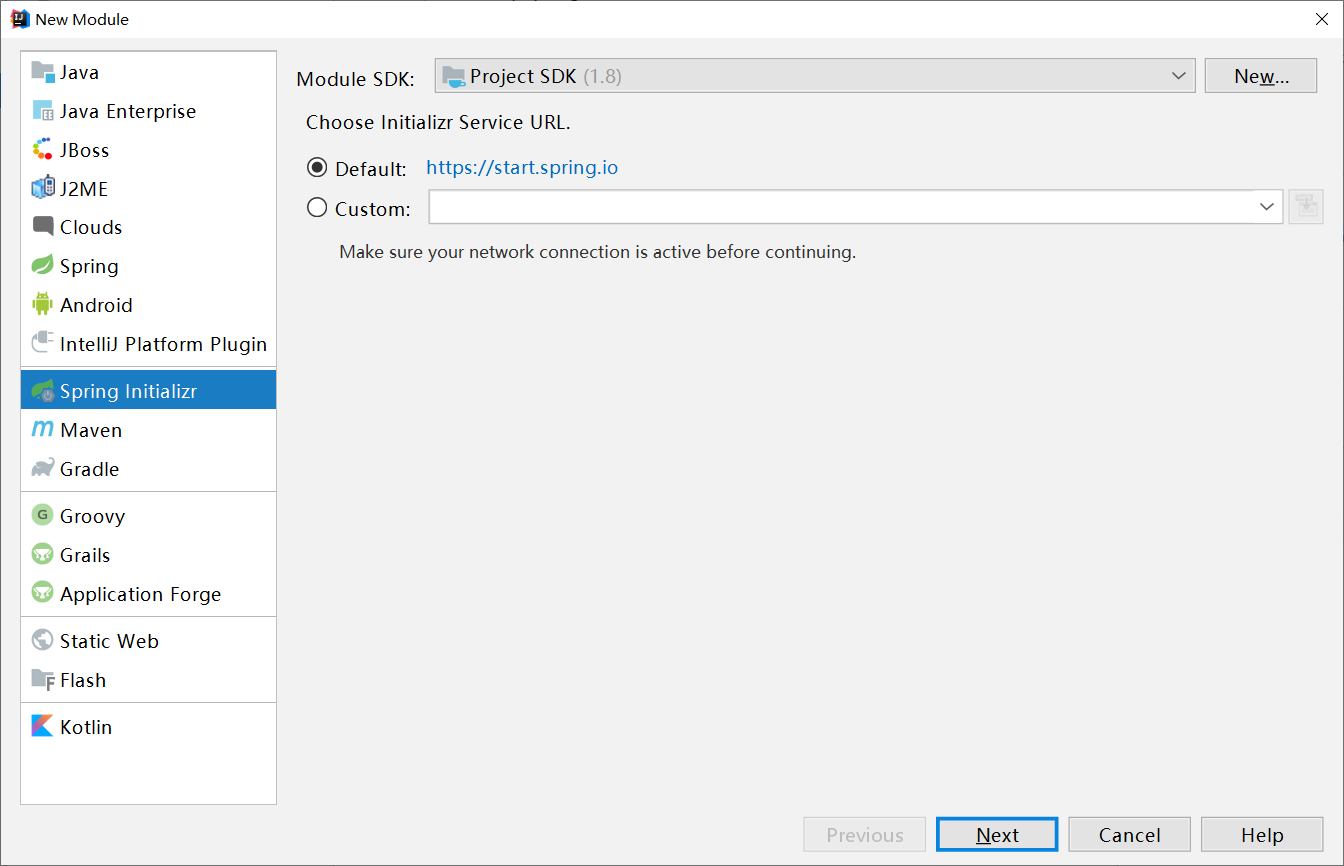
producer.send(message);

## Rabbitmq的延时消息实现

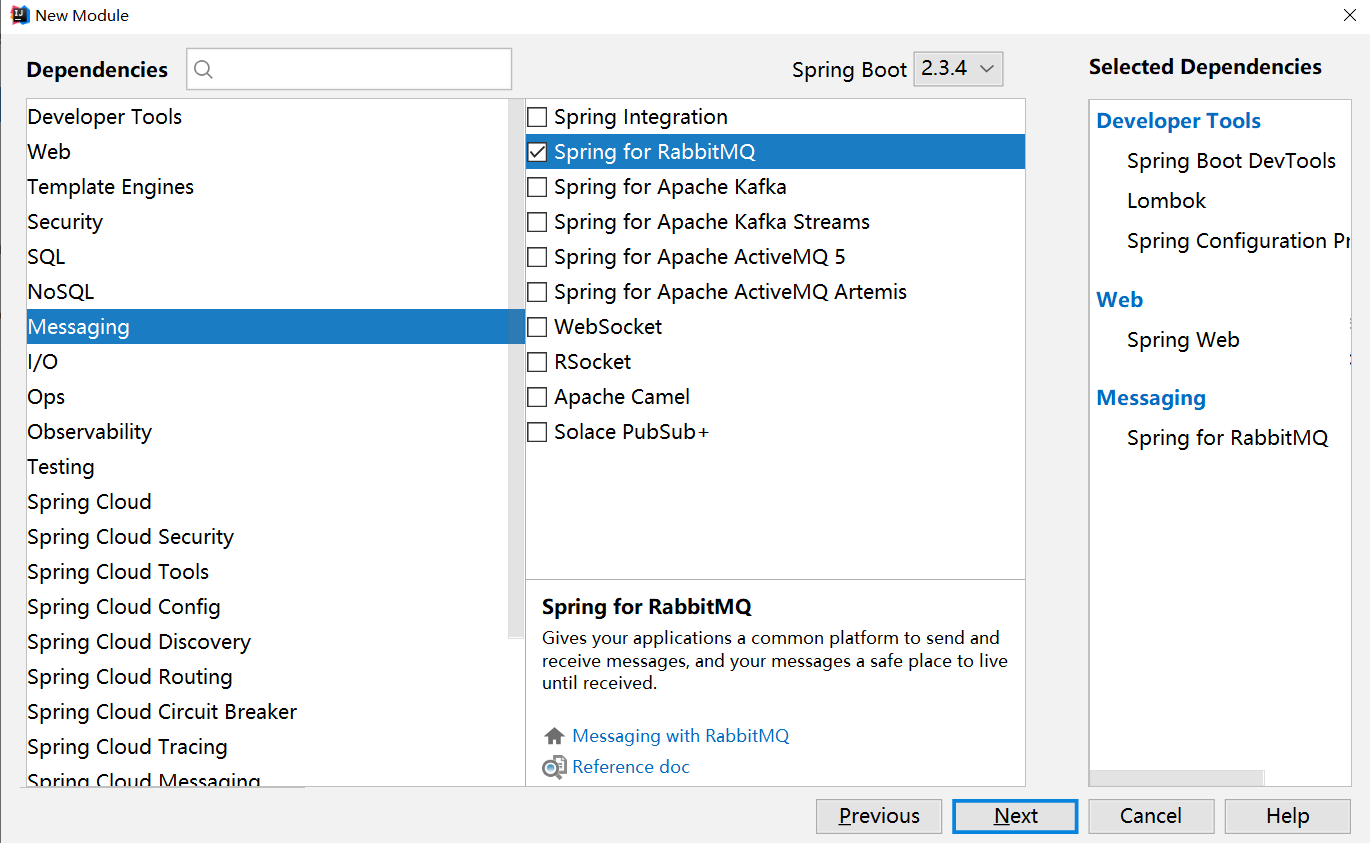
Rabbitmq 本身没有提供一个队列的机制，但是它里面有个延迟队列的机制

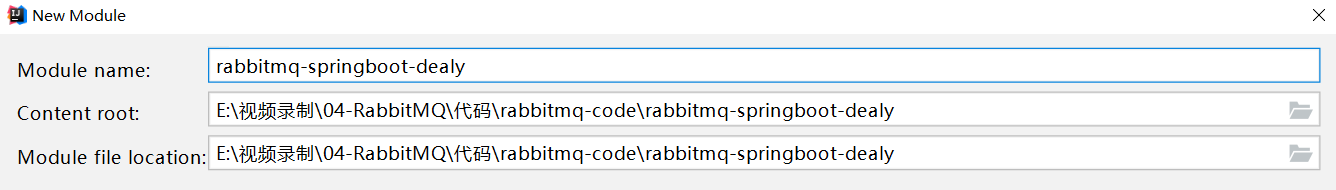
### 延迟队列的机制及创建项目rabbitmq-springboot-dealy











### 创建yml文件

|  |
| --- |
| **server**:  **port**: 8001 **spring**:  **application**:  **name**: rabbitmq-springboot-dealy  **rabbitmq**:  **host**: 81.68.123.67  **port**: 5672  **username**: sxt  **password**: 123456  **virtual-host**: /v-sxt  *# publisher-confirms: true 老版本里面的用法* **publisher-confirm-type**: *correlated #开启消息到达交换机的确认机制* **publisher-returns**: **true** *#消息由交互机到达队列时失败触发* **listener**:  **simple**:  **acknowledge-mode**: *manual #手动签收  #acknowledge-mode: auto #自动签收 这个是默认行为* **direct**:  **acknowledge-mode**: *manual #设置直连交互机的签收类型* |

### 代码实现延迟消息

|  |
| --- |
| **package** com.bjsxt.config;  **import** java.util.HashMap; **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;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/16:43  \** ***@Description:*** *\*/* **public class** DealyMessageConfig {  @Bean  **public** Queue dealyQueue(){  HashMap<String, Object> args = **new** HashMap<>();  *// 把一个队列修改为延迟队列* args.put(**"x-message-ttl"**,10\*1000) ; *// 消息的最大存活时间* args.put(**"x-dead-letter-exchange"**,**"DeadLetter.exc"**) ; *// 该队列里面的消息死了，去那个交换机* args.put(**"x-dead-letter-routing-key"**,**"DeadLetter.key"**) ; *// 该队列里面的消息死了，去那个交换机, 由那个路由key 路由他* Queue dealy = **new** Queue(**"dealy"**,**true**,**false**,**false**,args);  **return** dealy ;  }   */\*\*  \* 死信交互就  \** ***@return*** *\*/* @Bean  **public** DirectExchange deadLetterExchange() {  **return new** DirectExchange(**"DeadLetter.exc"**);  }    */\*\*  \* 绑定  \** ***@return*** *\*/* @Bean  **public** Binding newAndDeadLetterExchange(){  **return** BindingBuilder.*bind*(newQueue()).to(deadLetterExchange()).  with(**"DeadLetter.key"**); *// 死信路由key* }   */\*\*  \* 新生队列  \** ***@return*** *\*/* @Bean  **public** Queue newQueue(){  **return new** Queue(**"new.queue"**) ;  } } |

### 监听新生的世界

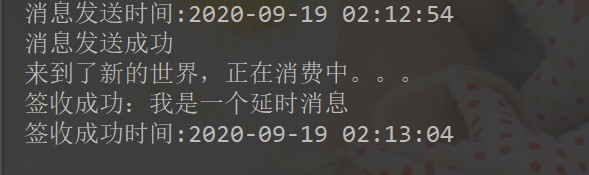
|  |
| --- |
| **package** com.bjsxt.receive;  **import** com.rabbitmq.client.Channel; **import** java.io.IOException; **import** java.text.SimpleDateFormat; **import** java.util.Date; **import** org.springframework.amqp.core.Message; **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created with IntelliJ IDEA.  \*  \** ***@Auther:*** *雷哥  \** ***@Date:*** *2020/10/11/14:49  \** ***@Description:*** *\*/* @Component @RabbitListener(queues = {**"new.queue"**}) **public class** MessageReceive5 {  @RabbitHandler  **public void** onMessage(String content, Message message, Channel channel) {  System.***out***.println(**"来到了新的世界，正在消费中。。。"**);  **long** deliveryTag = message.getMessageProperties().getDeliveryTag();   **try** {  channel.basicAck(deliveryTag, **false**);  System.***out***.println(**"签收成功："** + content);  SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  System.***out***.println(**"签收成功时间:"** + sdf.format(**new** Date()));  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

### 测试

我们往延迟队列发送消息

启动项目

|  |
| --- |
| @Test **public void** sendDealy() **throws** IOException {  SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  System.***out***.println(**"消息发送时间:"**+sdf.format(**new** Date()));  **rabbitTemplate**.convertAndSend(**"dealy"**,**"我是一个延时消息"**);  System.***out***.println(**"消息发送成功"**);  System.***in***.read(); } |



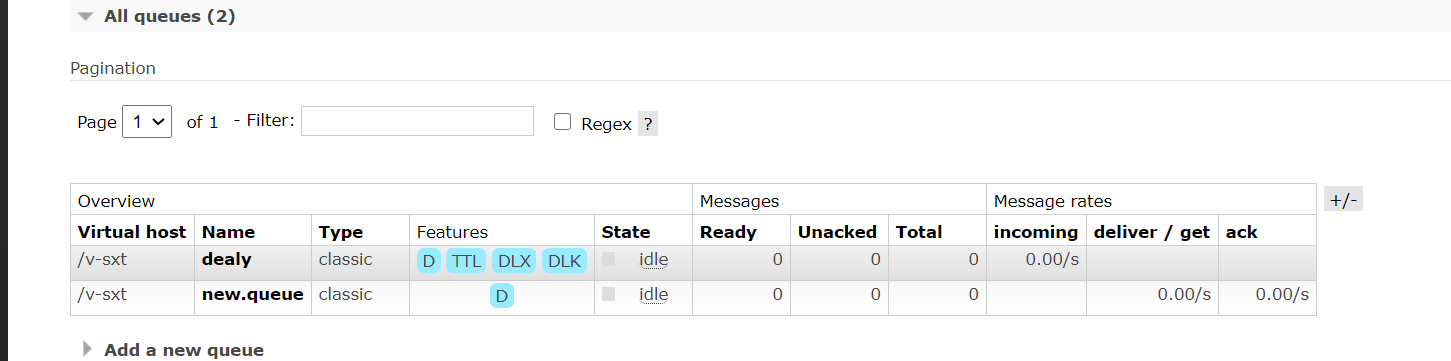
### 发消息时，也可以指定时间

我们的这个队列里面让你活10s ，你想活11s 不行

但是你活5s 是可以的

它是以那个最小为主的！

|  |
| --- |
| @Test **public void** sendDealy2() **throws** IOException {  SimpleDateFormat sdf = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  System.***out***.println(**"消息发送时间:"**+sdf.format(**new** Date()));  **rabbitTemplate**.convertAndSend(**"dealy"**, (Object) **"我是一个延时消息"**, **new** MessagePostProcessor() {  @Override  **public** Message postProcessMessage(Message message) **throws** AmqpException {  message.getMessageProperties().setExpiration(**"5000"**);*//我们延时队列里面设置的为10秒，但这个消息它只想活5秒* **return** message;  }  });  System.***out***.println(**"消息发送成功"**);  System.***in***.read(); } |

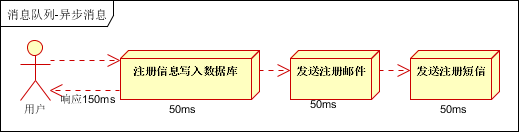


# **【实战】MQ的应用场景**

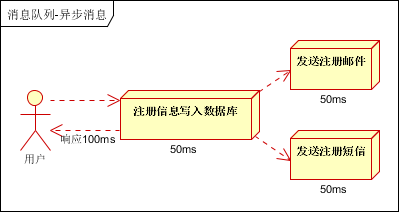
## 异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种 1.串行的方式；2.并行方式

a、串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。



b、并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间

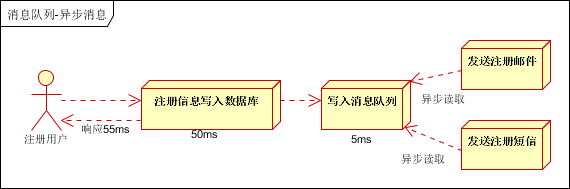


假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。

因为CPU在单位时间内处理的请求数是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次（1000/150）。并行方式处理的请求量是10次（1000/100）

小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

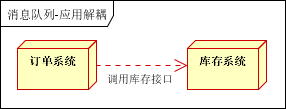
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。因此架构改变后，系统的吞吐量提高到每秒20 QPS。比串行提高了3倍，比并行提高了两倍。

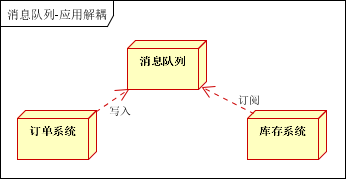
## 应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图：



传统模式的缺点：假如库存系统无法访问，则订单减库存将失败，从而导致订单失败，订单系统与库存系统耦合

如何解决以上问题呢？引入应用消息队列后的方案，如下图：



订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功

库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦

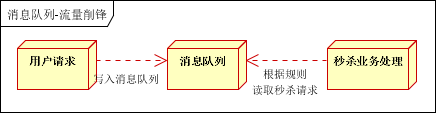
## 流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。

应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

a、可以控制活动的人数

b、可以缓解短时间内高流量压垮应用



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

秒杀业务根据消息队列中的请求信息，再做后续处理

## 日志处理

日志处理是指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。架构简化如下



日志采集客户端，负责日志数据采集，定时写受写入Kafka队列

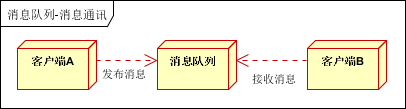
Kafka消息队列，负责日志数据的接收，存储和转发

日志处理应用：订阅并消费kafka队列中的日志数据

## 消息通讯

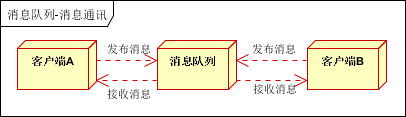
消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等

点对点通讯：



客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：



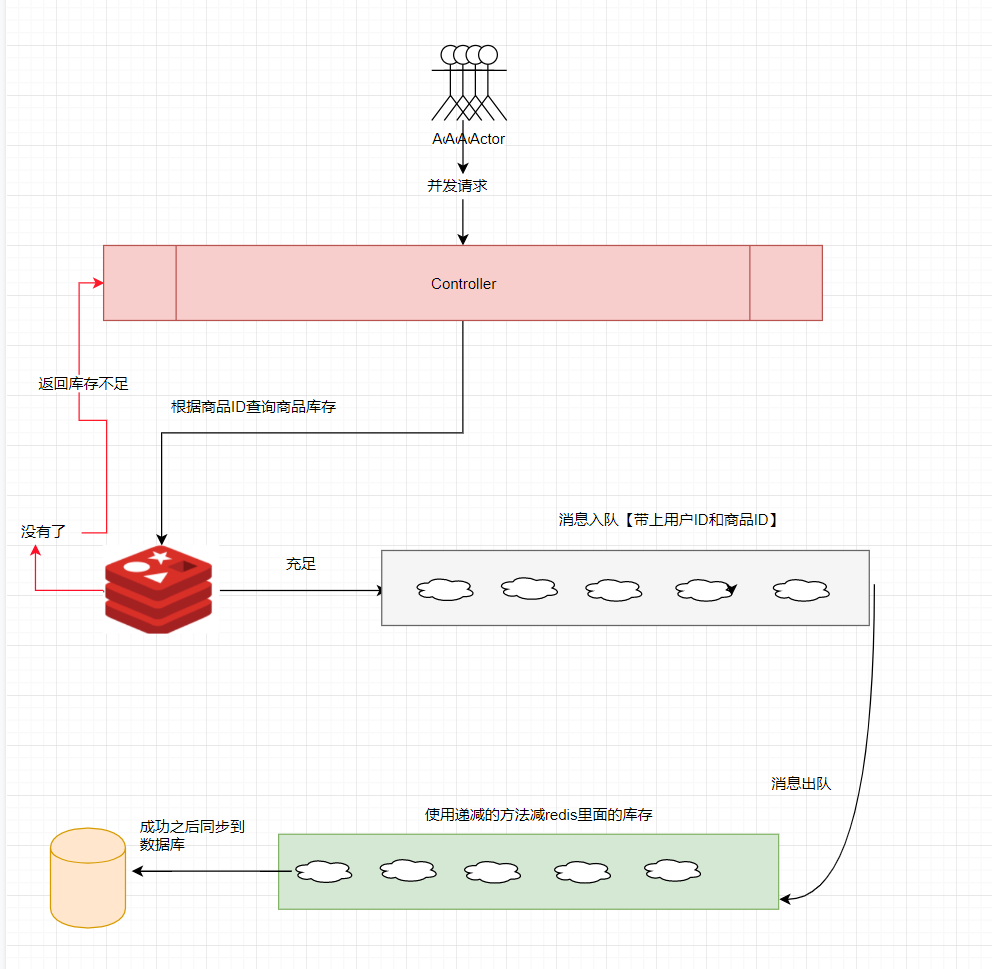
客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

# **【实战】秒杀**

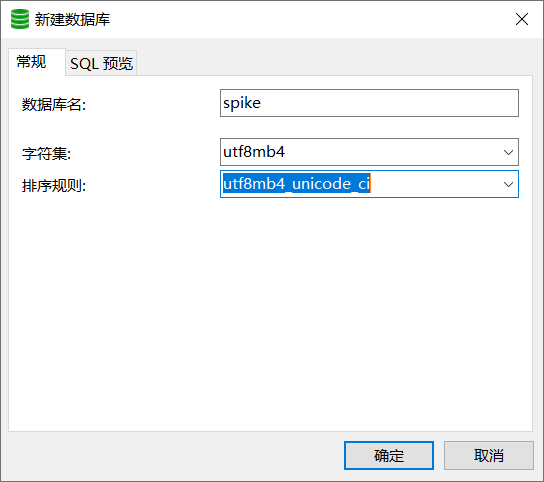
## 技术选择型

* Springboot
* redis
* rabbitmq
* mysql

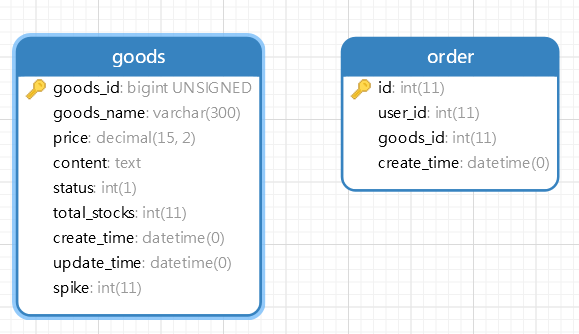
## 架构图



## 准备工作-数据库

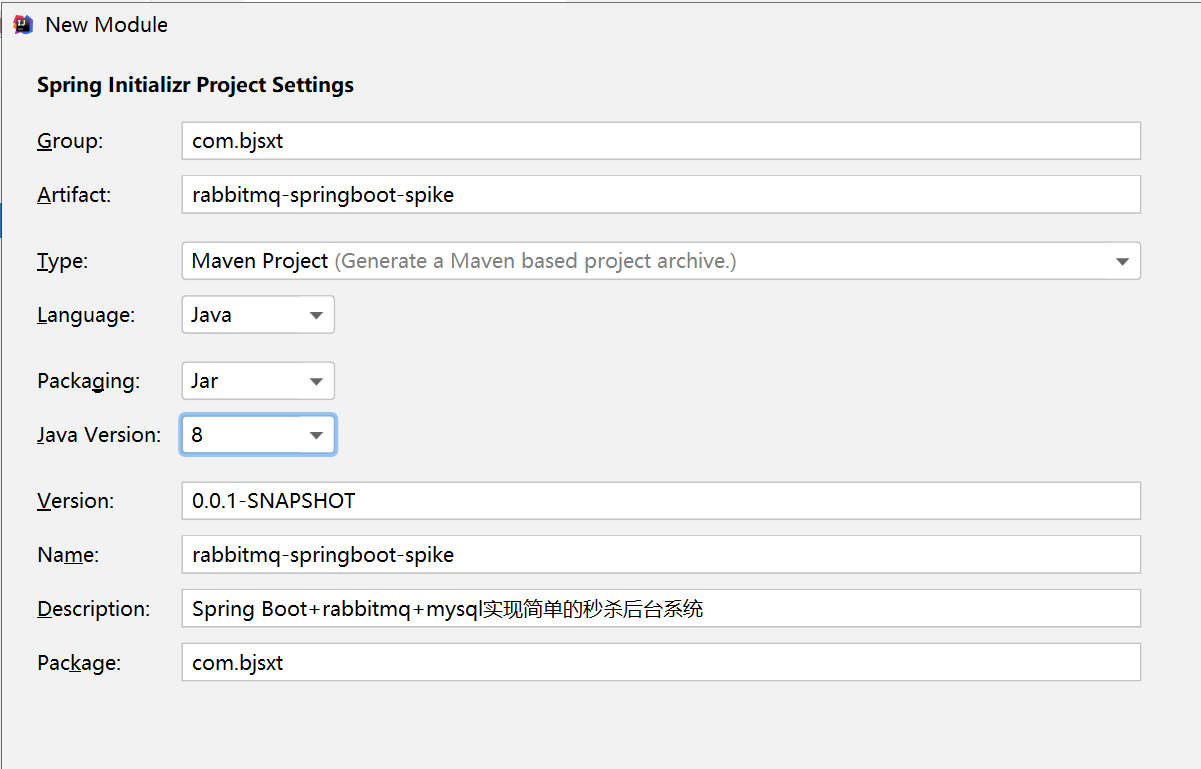


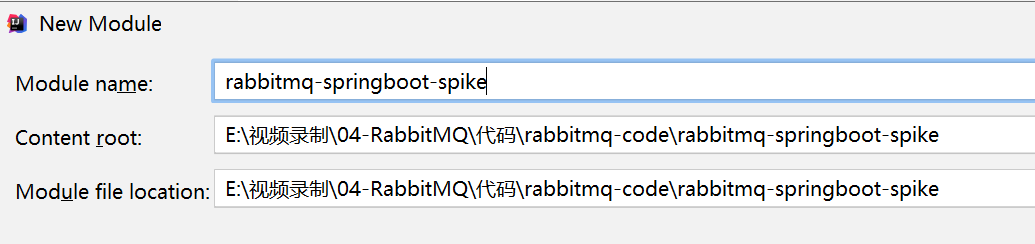
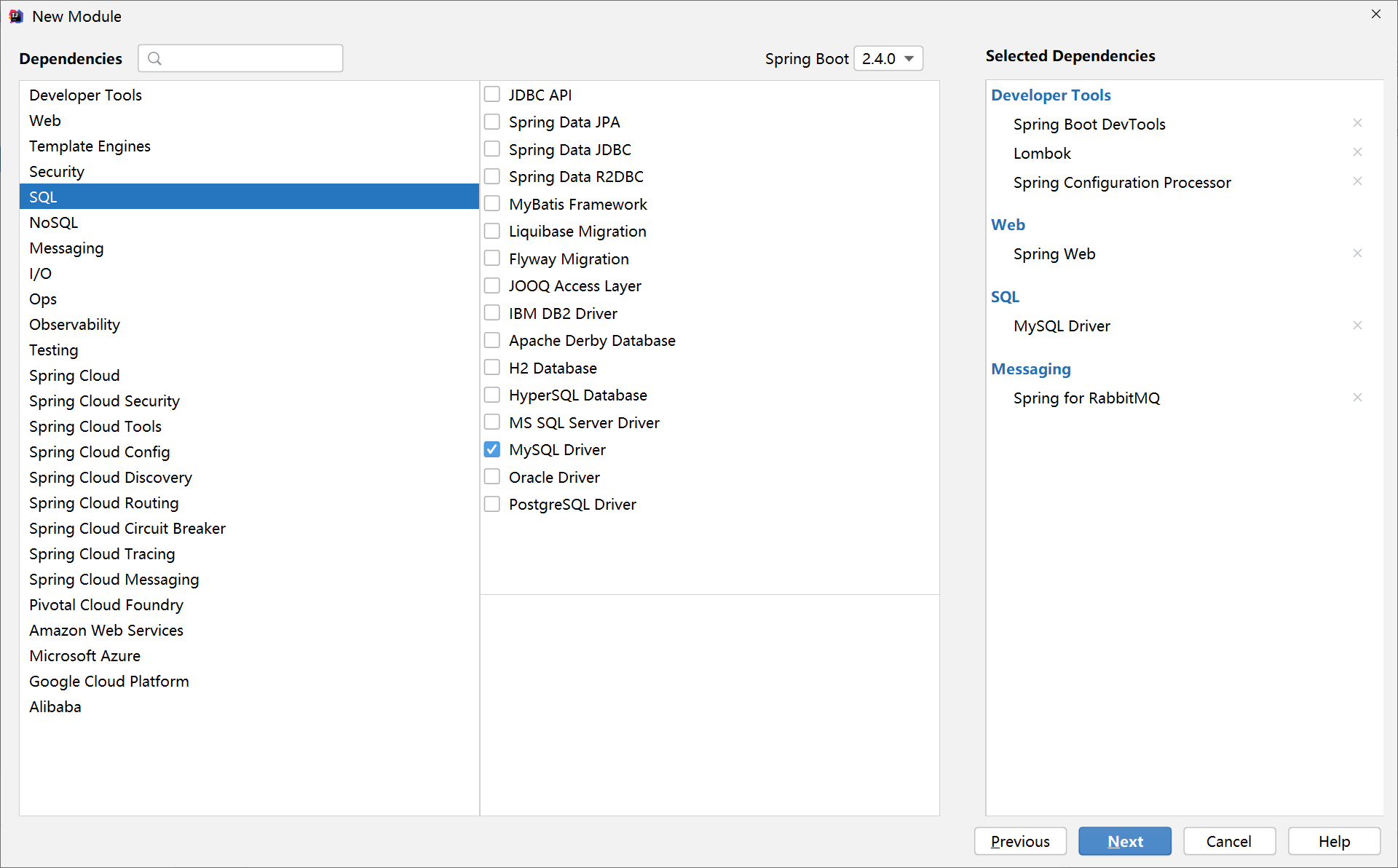
|  |
| --- |
| SET NAMES utf8mb4;  SET FOREIGN\_KEY\_CHECKS = 0;  -- ----------------------------  -- Table structure for goods  -- ----------------------------  DROP TABLE IF EXISTS `goods`;  CREATE TABLE `goods` (  `goods\_id` bigint(20) UNSIGNED NOT NULL AUTO\_INCREMENT COMMENT '商品ID',  `goods\_name` varchar(300) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL DEFAULT '' COMMENT '商品名称',  `price` decimal(15, 2) DEFAULT NULL COMMENT '现价',  `content` text CHARACTER SET utf8 COLLATE utf8\_general\_ci COMMENT '详细描述',  `status` int(1) DEFAULT 0 COMMENT '默认是1，表示正常状态, -1表示删除, 0下架',  `total\_stocks` int(11) DEFAULT 0 COMMENT '总库存',  `create\_time` datetime(0) DEFAULT NULL COMMENT '录入时间',  `update\_time` datetime(0) DEFAULT NULL COMMENT '修改时间',  `spike` int(11) DEFAULT 0 COMMENT '是否参与秒杀1是0否',  PRIMARY KEY (`goods\_id`) USING BTREE  ) ENGINE = InnoDB AUTO\_INCREMENT = 95 CHARACTER SET = utf8 COLLATE = utf8\_general\_ci COMMENT = '商品' ROW\_FORMAT = Dynamic;  -- ----------------------------  -- Records of goods  -- ----------------------------  INSERT INTO `goods` VALUES (18, 'Apple iPhone XS Max 移动联通电信4G手机 ', 1.01, '<div style=\"margin: 0px; padding: 0px; color: #666666; font-family: tahoma, arial, \'Microsoft YaHei\', \'Hiragino Sans GB\', u5b8bu4f53, sans-serif; font-size: 12px; font-style: normal; font-variant-ligatures: normal; font-variant-caps: normal; font-weight: 400; letter-spacing: normal; orphans: 2; text-indent: 0px; text-transform: none; white-space: normal; widows: 2; word-spacing: 0px; -webkit-text-stroke-width: 0px; background-color: #ffffff; text-decoration-style: initial; text-decoration-color: initial;\" align=\"center\">\n<table id=\"\_\_01\" style=\"text-align: center;\" border=\"0\" width=\"750\" cellspacing=\"0\" cellpadding=\"0\">\n<tbody>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img30.360buyimg.com/cms/jfs/t1/4626/32/3475/220504/5b997365E80a1373f/279c244f12161cb3.jpg\" alt=\"\" width=\"750\" height=\"1991\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img12.360buyimg.com/cms/jfs/t1/3397/21/3533/236322/5b99759aE73795787/f782e04a140c8f16.jpg\" alt=\"\" width=\"750\" height=\"2052\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img11.360buyimg.com/cms/jfs/t1/5274/3/3465/245167/5b997365E16b81bc9/93e07e40f3af5e62.jpg\" alt=\"\" width=\"750\" height=\"2250\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img30.360buyimg.com/cms/jfs/t1/2322/11/3524/269574/5b997365E26f81a7a/e01fc9486da9eda1.jpg\" alt=\"\" width=\"750\" height=\"2327\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img13.360buyimg.com/cms/jfs/t1/5074/21/3432/296470/5b997364Ee966f7a0/7f424d41479db45d.jpg\" alt=\"\" width=\"750\" height=\"2561\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img13.360buyimg.com/cms/jfs/t1/5770/18/3580/288371/5b997365Ea2c58cb4/176b9a40ccd4e56b.jpg\" alt=\"\" width=\"750\" height=\"2668\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img11.360buyimg.com/cms/jfs/t1/227/21/3811/268132/5b997364E3d6c51b2/92d2a3a559e3baa8.jpg\" alt=\"\" width=\"750\" height=\"2850\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img20.360buyimg.com/cms/jfs/t1/3787/5/3493/125020/5b997363E3c9f5910/ddbd08a556744630.jpg\" alt=\"\" width=\"750\" height=\"1486\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img30.360buyimg.com/cms/jfs/t1/1687/5/3327/266718/5b997366E9cc80e69/9e40ceae1fef4466.jpg\" alt=\"\" width=\"750\" height=\"3376\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img30.360buyimg.com/cms/jfs/t1/457/6/3849/283318/5b997363E0c5ab7a9/6f636f0a286bc87c.jpg\" alt=\"\" width=\"750\" height=\"2455\" /></td>\n</tr>\n<tr>\n<td><img class=\"\" style=\"margin: 0px; padding: 0px; border: 0px; vertical-align: middle;\" src=\"https://img10.360buyimg.com/cms/jfs/t1/397/25/3796/217624/5b9975a8E5ee578af/4d8f05a606fa5c4a.jpg\" alt=\"\" width=\"750\" height=\"2703\" /></td>\n</tr>\n</tbody>\n</table>\n</div>', 1, 10, '2019-03-29 14:40:00', '2019-06-22 18:28:32', 1);  INSERT INTO `goods` VALUES (59, '兰蔻粉水清滢柔肤水400ml 爽肤水女保湿舒缓滋润嫩肤', 420.00, '<p><img src=\"http://img-test.gz-yami.com/2019/04/71f54ee20ef34872b1e0aa53cb75b7b6.jpg\" alt=\"\" width=\"790\" height=\"1110\" /></p>', 1, 10, '2019-04-21 19:15:34', '2019-04-29 14:30:44', 1);  INSERT INTO `goods` VALUES (68, '【Dole都乐】菲律宾都乐非转基因木瓜1只 单只约410g', 26.00, '<p style=\"text-align: justify;\"><img src=\"http://img-test.gz-yami.com/2019/04/e7536a53a83d450e8635ce1e9819faf6.jpg\" alt=\"\" width=\"790\" height=\"350\" /></p>', 1, 10, '2019-04-21 21:56:38', '2019-05-22 10:30:37', 0);  INSERT INTO `goods` VALUES (69, '阿迪达斯官方 adidas 三叶草 NITE JOGGER 男子经典鞋BD7956', 1199.00, '<p><img src=\"http://img-test.gz-yami.com/2019/04/6d0bea4a0be54423999136bcd1158897.jpg\" alt=\"\" width=\"790\" height=\"2232\" /></p>', 1, 10, '2019-04-21 22:10:04', '2019-05-23 20:17:03', 0);  INSERT INTO `goods` VALUES (70, '【Dole都乐】比利时Truval啤梨12只 进口水果新鲜梨 单果120g左右', 38.00, '<p><img src=\"http://img-test.gz-yami.com/2019/04/67ce2251e9b14ea08b87752ef7b30207.jpg\" alt=\"\" width=\"760\" height=\"488\" /></p>', 1, 10, '2019-04-22 16:43:33', '2019-06-22 09:40:24', 0);  INSERT INTO `goods` VALUES (71, '旗舰店官网 自拍神器 梵高定制', 6998.00, '<p><img src=\"http://img-test.gz-yami.com/2019/04/fa35b300102e45f3a57d7c5c775ebf6d.jpg\" alt=\"\" width=\"790\" height=\"853\" /></p>\n<p><img src=\"http://img-test.gz-yami.com/2019/04/f8fd168ddb8a437dbb5b742691bd1d02.jpg\" alt=\"\" width=\"800\" height=\"800\" /><img src=\"http://img-test.gz-yami.com/2019/04/db46108466264b48841b18437940e0b3.jpg\" alt=\"\" width=\"800\" height=\"800\" /></p>', 1, 10, '2019-04-23 15:43:26', '2019-05-21 11:01:59', 0);  -- ----------------------------  -- Table structure for order  -- ----------------------------  DROP TABLE IF EXISTS `order`;  CREATE TABLE `order` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `user\_id` int(11) DEFAULT NULL,  `goods\_id` int(11) DEFAULT NULL,  `create\_time` datetime(0) DEFAULT NULL,  PRIMARY KEY (`id`) USING BTREE  ) ENGINE = InnoDB AUTO\_INCREMENT = 23 CHARACTER SET = utf8mb4 COLLATE = utf8mb4\_unicode\_ci ROW\_FORMAT = Dynamic;  SET FOREIGN\_KEY\_CHECKS = 1; |



## 创建项目

### 创建项目





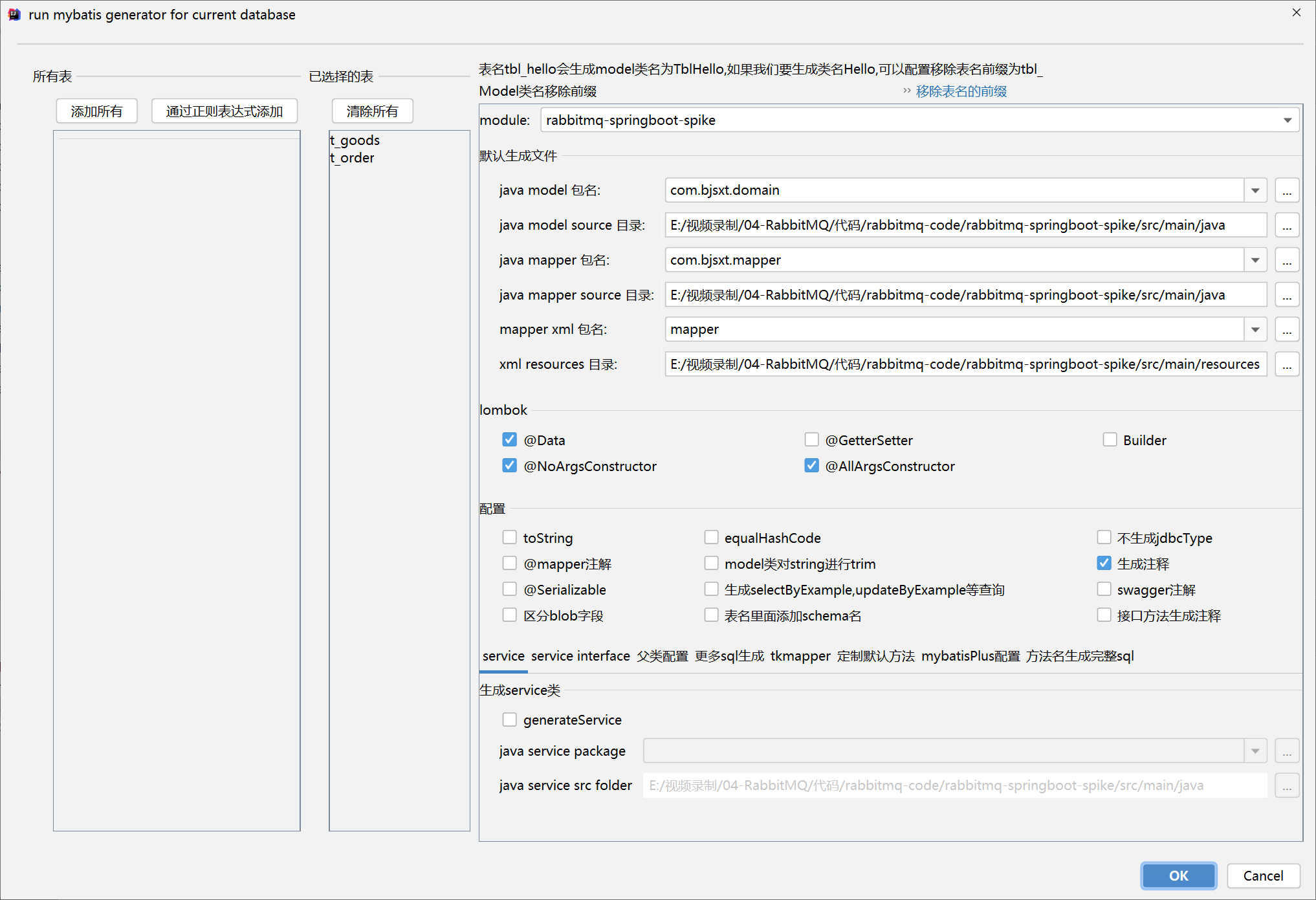
### 修改pom.xml

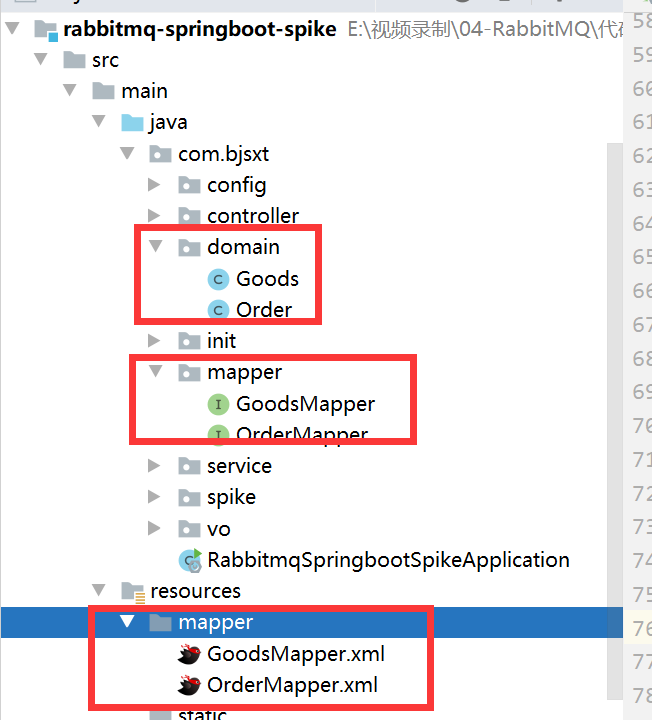
|  |
| --- |
| <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  <version>1.1.21</version> </dependency> <dependency>  <groupId>com.baomidou</groupId>  <artifactId>mybatis-plus-boot-starter</artifactId>  <version>3.4.1</version> </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId> </dependency> |

### 创建yml

|  |
| --- |
| server:  port: 8080 #数据源的配置 spring:  datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://127.0.0.1:3306/spike?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  username: root  password: 123456  #注入数据源的类型 默认的为HikariDataSource  type: com.alibaba.druid.pool.DruidDataSource  druid:  max-active: 20  min-idle: 5  validation-query: select x  initial-size: 3  max-wait: 5000  redis:  host: 81.68.123.67  password: 123456  port: 6390  rabbitmq:  host: 81.68.123.67  port: 5672  username: sxt  password: 123456  virtual-host: v-sxt  #mybatis的配置 mybatis-plus:  mapper-locations: classpath:mapper/\*Mapper.xml #配置mapper.xml的扫描  configuration:  log-impl: org.apache.ibatis.logging.stdout.StdOutImpl |

## 生成Goods和Order相关





## 创建RabbitMQConfig

|  |
| --- |
| package com.bjsxt.config;  import org.springframework.amqp.core.Queue; import org.springframework.context.annotation.Bean;  /\*\*  \* @Author 武汉尚学堂  \*/ public class RabbitMQConfig {   public static final String *QUEUE\_NAME*="spike-queue";   @Bean  public Queue queue(){  return new Queue(*QUEUE\_NAME*);  } } |

## 创建Sender发送消息

|  |
| --- |
| package com.bjsxt.spike;  import com.bjsxt.config.RabbitMQConfig; import lombok.extern.log4j.Log4j2; import org.springframework.amqp.rabbit.core.RabbitTemplate; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Component;  import java.util.HashMap; import java.util.Map;  /\*\*  \* @Author 武汉尚学堂  \*/ @Component @Log4j2 public class Sender {   @Autowired  private RabbitTemplate rabbitTemplate;   private int i=0;   /\*\*  \* 如查用户抢到商品就发送到队列，信息包含商品ID和用户ID  \* @param goodsId  \* @param userId  \*/  public void sendQueue(String goodsId,String userId){  *log*.info(">>>>>>>>>>>>>>秒杀请求已发送，商品ID为："+goodsId+"--用户ID："+userId);  try {  Map<String,String> map=new HashMap<>();  map.put("goodsId",goodsId);  map.put("userId",userId);  //第一个参数是指要发送到哪个队列里面， 第二个参数是指要发送的内容  rabbitTemplate.convertAndSend(RabbitMQConfig.*QUEUE\_NAME*, map);  //此处为了记录并发请求下，请求的次数及消息传递的次数  *log*.info("发送请求>>>>>>>>>>>>>"+i++);  } catch (Exception e) {  *log*.error("请求发送异常："+e.getMessage());  e.printStackTrace();  }  } } |

## 创建Receiver异步接收消息

|  |
| --- |
| package com.bjsxt.spike;  import com.bjsxt.config.RabbitMQConfig; import com.bjsxt.domain.Order; import com.bjsxt.service.GoodsService; import com.bjsxt.service.OrderService; import lombok.extern.log4j.Log4j2; import org.springframework.amqp.rabbit.annotation.Queue; import org.springframework.amqp.rabbit.annotation.RabbitListener; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.data.redis.core.StringRedisTemplate; import org.springframework.data.redis.core.ValueOperations; import org.springframework.stereotype.Component;  import java.util.Date; import java.util.Map;  /\*\*  \* @Author 武汉尚学堂  \*/ @Component @Log4j2 public class Receiver {  @Autowired  private StringRedisTemplate redisTemplate;   @Autowired  private GoodsService goodsService;   @Autowired  private OrderService orderService;    private int i;   @RabbitListener(queuesToDeclare = @Queue(RabbitMQConfig.*QUEUE\_NAME*))  public void receive(Map<String,String> map){  try {  //取到商品ID判断redis里面的该商品库存是否为0  String goodsId = map.get("goodsId");  ValueOperations<String, String> opsForValue = redisTemplate.opsForValue();  long num = opsForValue.decrement(goodsId).longValue();//递减  if(num<0){  /\*\*  \* 此处不能判断等于0，因为当商品库存为1时，Redis执行递减返回为0  \* 如果判断为0商品最后不能卖完也就是当库存为1时此处就抛异常了  \*/  throw new RuntimeException("库存不足啦，不能再抢了");  }  *log*.info("接收时>>>>>>>>>>>"+i++);  //根据商品的id和库存同步数据到MySQL  if(!goodsService.deductionGoodsStock(Long.*valueOf*(goodsId),1)){  throw new RuntimeException("同步到商品表异常！");  }  ////生成订单  String uid=map.get("userId");  *log*.info("成功了>>>>>>>>>>>"+uid+" 抢到了商品 O(∩\_∩)O哈哈~");  Order order=new Order();  order.setGoodsid(Integer.*valueOf*(goodsId));  order.setUserid(Integer.*valueOf*(uid));  order.setCreatetime(new Date());  orderService.addOrder(order);  }catch (Exception e){  e.printStackTrace();  }  } } |

## 创建GoodsService

|  |
| --- |
| package com.bjsxt.service;  import com.bjsxt.domain.Goods;  import java.util.List;  /\*\*  \* @Author 武汉尚学堂  \*/ public interface GoodsService {  /\*\*  \* 扣减库存  \* @param goodsId 商品ID  \* @param num 秒杀数量  \* @return  \*/  boolean deductionGoodsStock(Long goodsId,Integer num); } |

## 创建GoodsServiceImpl

|  |
| --- |
| package com.bjsxt.service.impl;  import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper; import com.bjsxt.domain.Goods; import com.bjsxt.mapper.GoodsMapper; import com.bjsxt.service.GoodsService; import com.bjsxt.spike.Sender; import lombok.extern.log4j.Log4j2; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.data.redis.core.StringRedisTemplate; import org.springframework.data.redis.core.ValueOperations; import org.springframework.stereotype.Service;  import java.util.List;  /\*\*  \* @Author 武汉尚学堂  \*/ @Service @Log4j2 public class GoodsServiceImpl implements GoodsService {   @Autowired  private GoodsMapper goodsMapper;  @Override  public boolean deductionGoodsStock(Long goodsId,Integer num) {  return goodsMapper.deductionGoodsStock(goodsId,num)>0;  } } |

## 创建OrderService

|  |
| --- |
| package com.bjsxt.service;  import com.bjsxt.domain.Order;  /\*\*  \* @Author 武汉尚学堂  \*/ public interface OrderService {  /\*\*  \* 保存订单  \* @param order  \*/  void addOrder(Order order); } |

## 创建OrderServiceImpl

|  |
| --- |
| package com.bjsxt.service.impl;  import com.bjsxt.domain.Order; import com.bjsxt.mapper.OrderMapper; import com.bjsxt.service.OrderService; import lombok.extern.log4j.Log4j2; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service;  /\*\*  \* @Author 武汉尚学堂  \*/ @Service @Log4j2 public class OrderServiceImpl implements OrderService {   @Autowired  private OrderMapper orderMapper;   @Override  public void addOrder(Order order) {  this.orderMapper.insert(order);  } } |

## 启动程序加载要加入秒杀的商品到redis

### 创建ApplicationListener

|  |
| --- |
| package com.bjsxt.init;  import com.bjsxt.domain.Goods; import com.bjsxt.service.GoodsService; import lombok.extern.log4j.Log4j2; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.data.redis.core.StringRedisTemplate; import org.springframework.data.redis.core.ValueOperations; import org.springframework.stereotype.Component;  import javax.annotation.PostConstruct; import java.util.Iterator; import java.util.List;  /\*\*  \* @Author 武汉尚学堂  \*/ @Component @Log4j2 public class ApplicationInitListener {   @Autowired  private GoodsService goodsService;   @Autowired  private StringRedisTemplate redisTemplate;    @PostConstruct  public void initGoodsToRedis() {  ValueOperations<String, String> opsForValue = redisTemplate.opsForValue();  *log*.info(">>>>>>>>>>>>项目初始化完成，执行监听器中逻辑");  //mapper中的sql，返回全部上架（支持秒杀）的商品集合  List<Goods> list = goodsService.selectGoodsToSpike();  Iterator<Goods> it = list.iterator();  while (it.hasNext()) {  Goods p = it.next();  try {  opsForValue.set(String.*valueOf*(p.getGoodsId()), String.*valueOf*(p.getTotalStocks()));  *log*.info("商品放成Redis成功ID：" + p.getGoodsId() + "商品库存：" + p.getTotalStocks());  } catch (Exception e) {  *log*.error("当前商品ID：" + p.getGoodsId() + "库存：" + p.getTotalStocks() + "放入Redis缓存异常<<<<<<<<<<<<<<<<<<<<");  e.printStackTrace();  }  }  } } |

### 修改GoodsService

|  |
| --- |
| /\*\*  \* 查询需要秒杀的商品  \* @return  \*/ List<Goods> selectGoodsToSpike(); |

### 修改GoodsServiceImpl

|  |
| --- |
| @Override public List<Goods> selectGoodsToSpike() {  QueryWrapper<Goods> qw=new QueryWrapper<>();  qw.eq(Goods.*COL\_SPIKE*,1);  return this.goodsMapper.selectList(qw); } |

## 创建ResultObj

|  |
| --- |
| package com.bjsxt.vo;  import lombok.AllArgsConstructor; import lombok.Data; import lombok.NoArgsConstructor;  /\*\*  \* @Author 武汉尚学堂  \*/ @Data @AllArgsConstructor @NoArgsConstructor public class ResultObj {   private Integer code=0;  private String msg=""; } |

## 创建SpikeController

|  |
| --- |
| package com.bjsxt.controller;  import com.bjsxt.service.GoodsService; import com.bjsxt.vo.ResultObj; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController;  /\*\*  \* @Author 武汉尚学堂  \*/ @RestController public class SpikeController {   @Autowired  private GoodsService goodsService;   /\*\*  \* 秒杀入口  \* @param pid -商品id，做检查库存使用  \* @param userId -用户id，做订单和用户关联使用（比如生成成功秒杀商品的用户订单表）  \* 我这里没做多余的逻辑，只看了相关情况的返回结果，有需要的可以自己去实现  \*/  @RequestMapping(value = "doSpike")  public ResultObj secondsKill(String pid, String userId) {  try {  boolean result = goodsService.doSpike(pid, userId);  if(result) {  return new ResultObj(1,"秒杀成功，请稍后去订单查询");  }  } catch (Exception e) {  e.printStackTrace();  return new ResultObj(-1,"秒杀失败，原因:"+e.getMessage());  }  return null;  } } |

## 修改GoodsService

|  |
| --- |
| /\*\*  \* 秒杀  \* @param pid  \* @param userId  \* @return  \*/ boolean doSpike(String pid, String userId); |

## 修改GoodsServiceImpl

|  |
| --- |
| @Autowired private StringRedisTemplate redisTemplate;  @Autowired private Sender sender;  private int i=0; @Override public boolean doSpike(String pid, String userId) {  ValueOperations<String, String> opsForValue = redisTemplate.opsForValue();  boolean result=true;//默认成功  //根据商品ID获取redis中的库存数量  String num=opsForValue.get(pid);  *log*.info("redis>>>>>>{0}",num);  if(Long.*valueOf*(num)<=0){  result=false;  throw new RuntimeException("库存不足");  }  //消息入队，调用相关的方法  sender.sendQueue(pid,userId);  //只是验证请求及发送消息次数  *log*.info("请求次数>>>>>>{0}",i++);  return result;  } |

## 测试

<http://127.0.0.1:8080/doSpike?pid=18&userId=1>