# RabbitMQ\_基础入门

# 第一章 MQ基本概念

## 一、MQ概述

MQ：

MQ全称 Message Queue（消息队列），是在消息的传输过程中保存消息的容器（中间件）。多用于分布式系统之间进行通信。

图解：

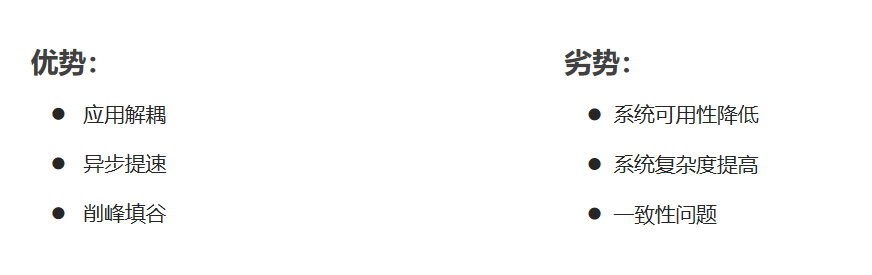


小结：

* MQ，消息队列，存储消息的**中间件**
* 分布式系统通信两种方式：**直接远程调用** 和 **借助第三方** 完成间接通信
* 消息发送方称为**生产者**，消息接收方称为**消费者**

## 二、MQ的优势&劣势

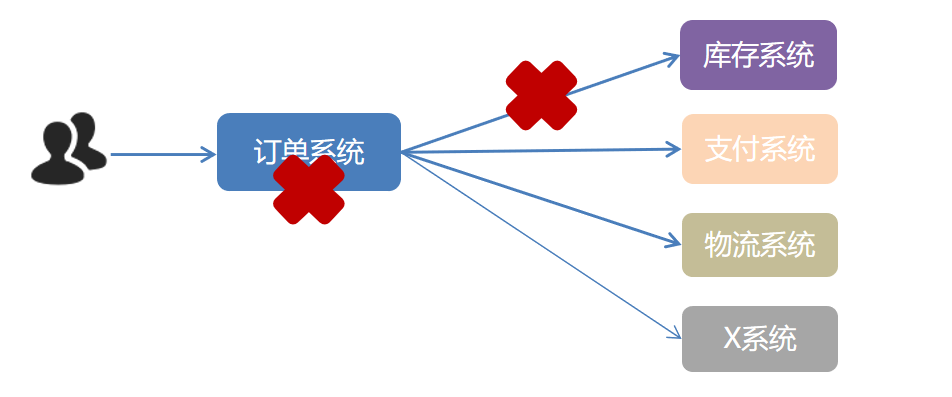
总述：



## 三、MQ的优势

### 1、应用解耦

如下图：

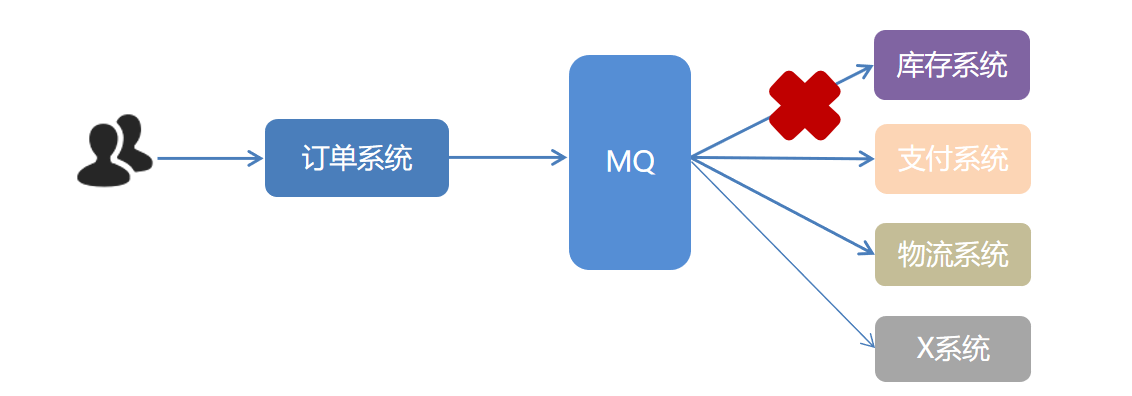


(1) 库存系统异常，导致整个订单系统异常。但是其他系统，比如支付系统、物流系统等，本来是正常的，却也因此被认为是异常的了。

(2) 当每次需要添加或删除一个X系统时，都需要修改订单系统的代码，使得订单系统的代码修改非常频繁

可见：系统的耦合性越高，容错性就越低，可维护性就越低。

引入MQ：



(1) 当库存系统异常时，订单系统不会整个崩溃，其中的支付系统、物流系统依然是正常的

(2) 当需要往订单系统中添加或删除一个X系统时，也不需要去修改订单系统的代码

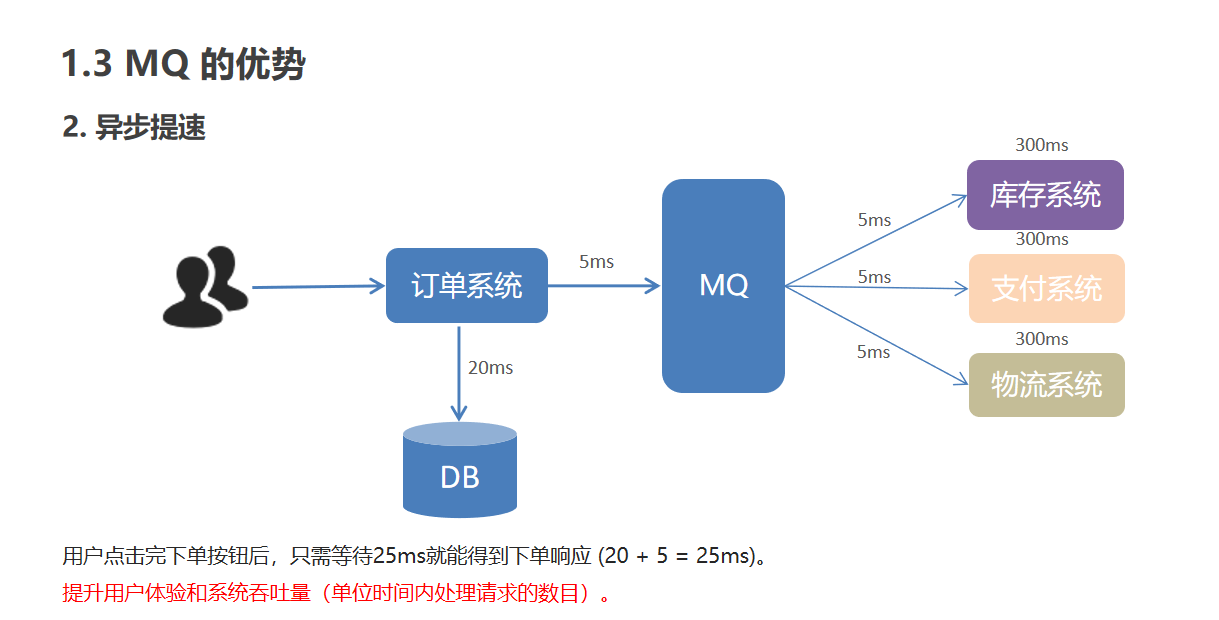
可见：使用 MQ 使得应用间解耦，提升容错性和可维护性。

### 2、异步提速

没有使用MQ时：



使用MQ后：



系统吞吐量：

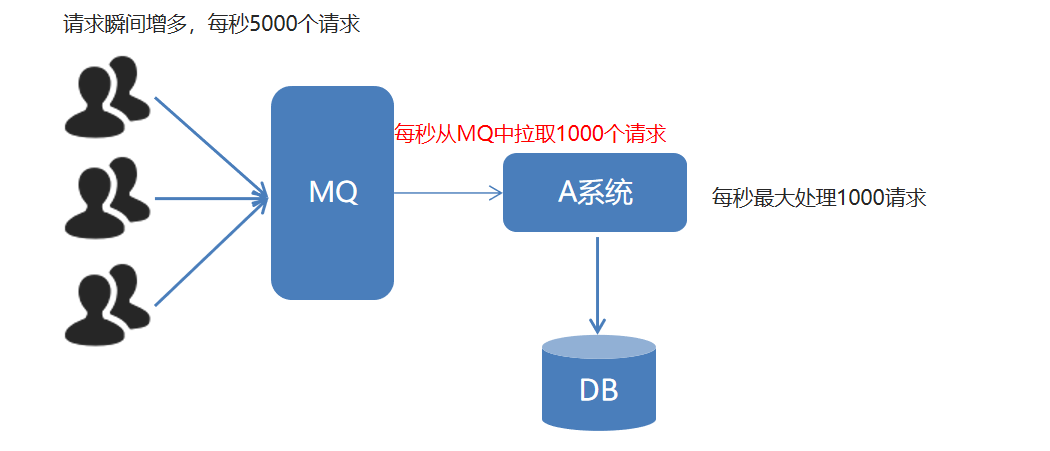
单位时间处理请求的数目

### 3、削峰填谷

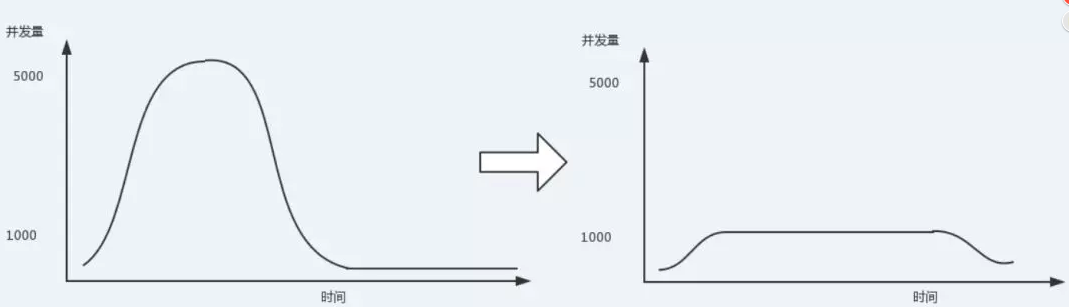
没有使用MQ时：



使用MQ后：



对于MQ来说，每秒5000个请求只是小意思。



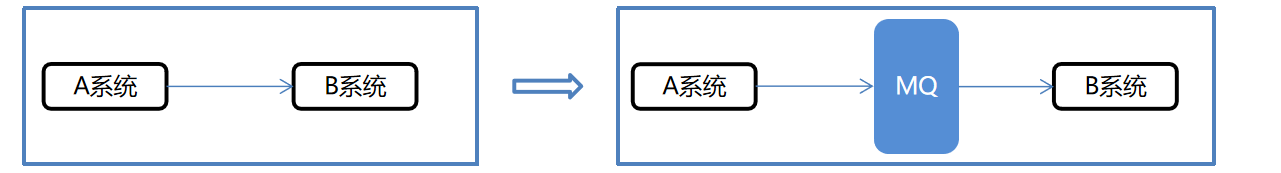
使用了 MQ 之后，限制消费消息的速度为1000，这样一来，高峰期产生的数据势必会被积压在 MQ 中，高峰就被“削”掉了，但是因为消息积压，在高峰期过后的一段时间内，消费消息的速度还是会维持在1000，直到消费完积压的消息，这就叫做“填谷”。

使用MQ后，可以提高系统稳定性。

### 4、优势小结

* 应用解耦：提高系统容错性和可维护性
* 异步提速：提升用户体验和系统吞吐量
* 削峰填谷：提高系统稳定性

## 四、MQ的劣势



1、系统可用性降低

以前，只需要保证A、B两个系统正常就可以了，现在，需要保证A、B、MQ这三个都是正常的。

系统引入的外部依赖越多，系统稳定性越差。一旦 MQ 宕机，就会对业务造成影响。如何保证MQ的高可用？

2、系统复杂度提高

MQ 的加入大大增加了系统的复杂度，以前系统间是同步的远程调用，现在是通过 MQ 进行异步调用。如何保证消息没有被重复消费？怎么处理消息丢失情况？那么保证消息传递的顺序性？

3、一致性问题

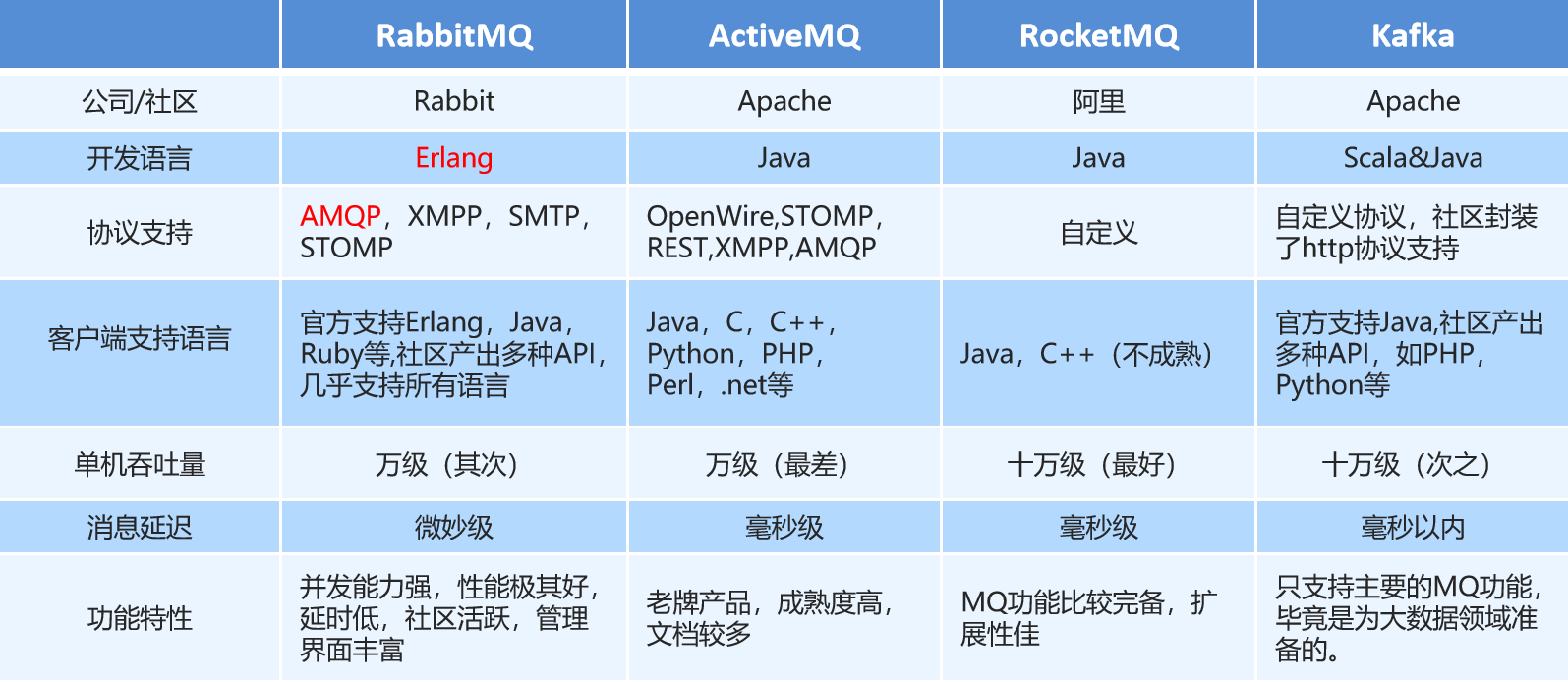
A 系统处理完业务，通过 MQ 给B、C、D三个系统发消息数据，如果 B 系统、C 系统处理成功，D 系统处理失败。如何保证消息数据处理的一致性？

使用MQ的条件（依据）：

* 生产者不需要从消费者处获得反馈。引入消息队列之前的直接调用，其接口的返回值应该为空，这才让明明下层的动作还没做，上层却当成动作做完了继续往后走，即所谓异步成为了可能。
* 容许短暂的不一致性。
* 确实是用了有效果。即解耦、提速、削峰这些方面的收益，超过加入MQ，管理MQ这些成本。

## 五、常见的MQ产品

目前业界有很多的 MQ 产品，例如 RabbitMQ、RocketMQ、ActiveMQ、Kafka、ZeroMQ、MetaMq等，也有直接使用 Redis 充当消息队列的案例，而这些消息队列产品，各有侧重，在实际选型时，需要结合自身需求及 MQ 产品特征，综合考虑。

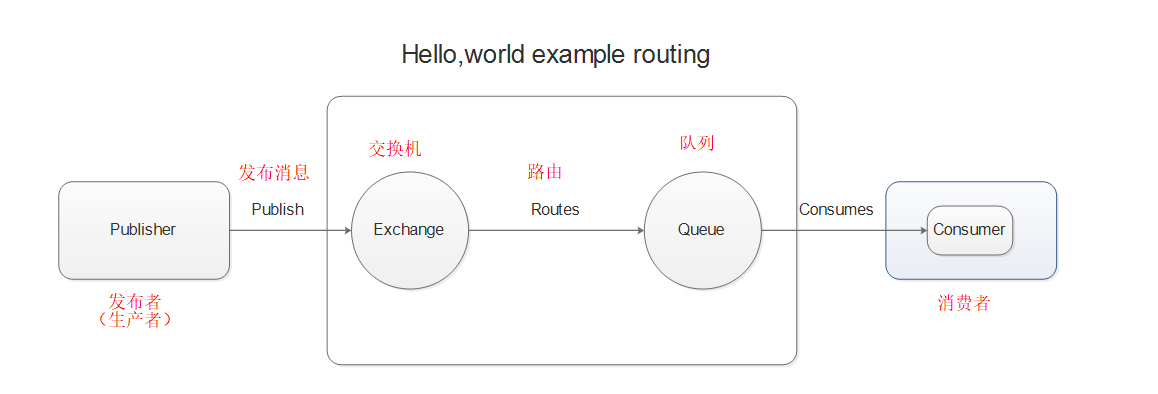


由于 RabbitMQ 综合能力强劲，所以接下来的课程中，我们将主要学习 RabbitMQ。

## 六、RabbitMQ简介

AMQP：

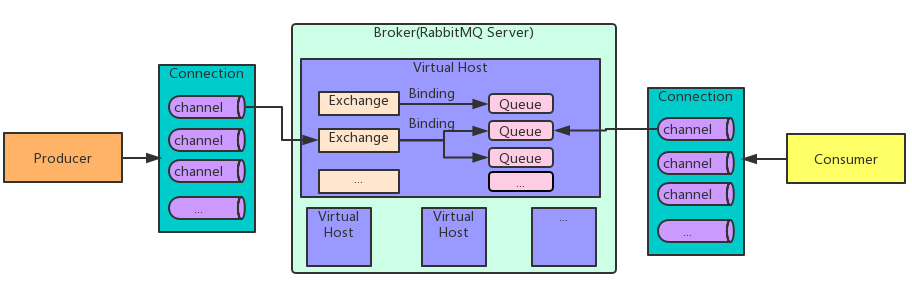
AMQP，即 Advanced Message Queuing Protocol（高级消息队列协议），是一个网络协议，是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受客户端/中间件不同产品，不同的开发语言等条件的限制。2006年，AMQP 规范发布。类比HTTP。



RabbitMQ：

RabbitMQ 采用 Erlang 语言开发。Erlang 语言由 Ericson 设计，专门为开发高并发和分布式系统的一种语言，在电信领域使用广泛。

RabbitMQ 基础架构图：



RabbitMQ 中的相关概念：

* Broker：中间件。接收和分发消息的应用，RabbitMQ Server就是 Message Broker
* Virtual host：虚拟机。出于多租户和安全因素设计的，把 AMQP 的基本组件划分到一个虚拟的分组中，类似于网络中的 namespace 概念。当多个不同的用户使用同一个 RabbitMQ server 提供的服务时，可以划分出多个vhost，每个用户在自己的 vhost 创建 exchange／queue 等。

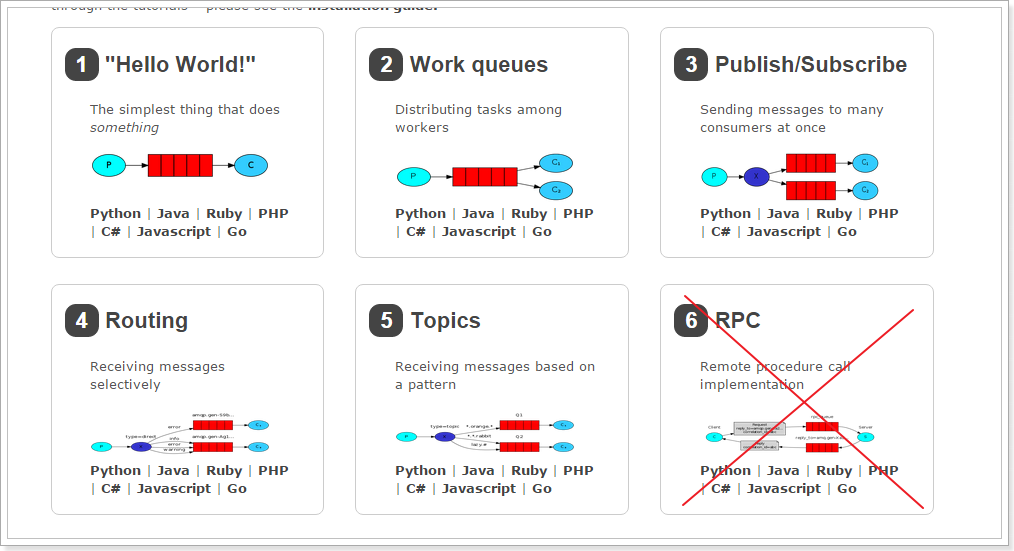
简单理解，类似于mysql中的数据库。

* Connection：连接。publisher／consumer 和 broker 之间的 TCP 连接
* Channel：如果每一次访问 RabbitMQ 都建立一个 Connection，在消息量大的时候建立 TCP Connection的开销将是巨大的，效率也较低。Channel 是在 connection 内部建立的逻辑连接，如果应用程序支持多线程，通常每个thread创建单独的 channel 进行通讯，AMQP method 包含了channel id 帮助客户端和message broker 识别 channel，所以 channel 之间是完全隔离的。Channel 作为轻量级的 Connection 极大减少了操作系统建立 TCP connection 的开销。
* Exchange：交换机。message 到达 broker 的第一站，根据分发规则，匹配查询表中的 routing key，分发消息到queue 中去。常用的类型有：direct (point-to-point), topic (publish-subscribe) and fanout (multicast)
* Queue：队列。消息最终被送到这里等待 consumer 取走
* Binding：绑定。exchange 和 queue 之间的虚拟连接，binding 中可以包含 routing key。Binding 信息被保存到 exchange 中的查询表中，用于 message 的分发依据

RabbitMQ六种工作模式：

简单模式、work queues、Publish/Subscribe 发布与订阅模式、Routing 路由模式、Topics 主题模式、RPC 远程调用模式（远程调用，不太算 MQ；暂不作介绍）。

官网对应模式介绍：<https://www.rabbitmq.com/getstarted.html>



## 七、JMS

* JMS 即 Java 消息服务（JavaMessage Service）应用程序接口，是一个 Java 平台中关于面向消息中间件的API
* JMS 是 JavaEE 规范中的一种，类比JDBC
* 很多消息中间件都实现了JMS规范，例如：ActiveMQ。RabbitMQ 官方没有提供 JMS 的实现包，但是开源社区有

## 八、小结

* RabbitMQ 是基于 AMQP 协议使用 Erlang 语言开发的一款消息队列产品。
* RabbitMQ提供了6种工作模式，我们学习5种。这是今天的重点。
* AMQP 是协议，类比HTTP。
* JMS 是 API 规范接口，类比 JDBC。

# 第二章 RabbitMQ的安装和配置

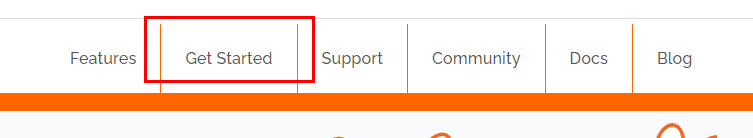
安装可以参考讲义：《RabbitMQ安装说明文档.md》

RabbitMQ 官方地址：<http://www.rabbitmq.com/>

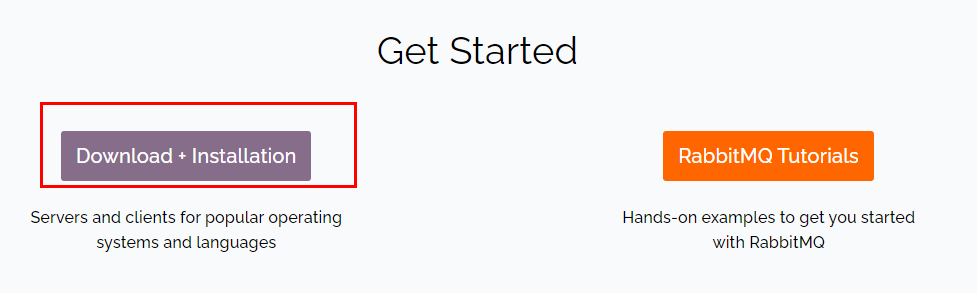
## 一、了解官网

要下载的话：

先点击首页上面的：Get Started



然后Download，右边的是手册



然后选择要下载的RabbitMQ的版本：

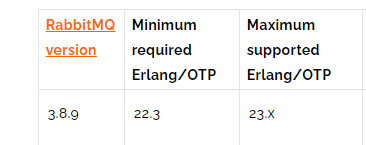


同时需要注意该页面右边的erlang version：



点进去以后，可以查看与RabbitMQ匹配的erlang的版本范围

比如说：



这个就表示，如果RabbitMQ的版本是3.8.9，那么erlang的版本应该介于22.3~23.x之间，否则不兼容。

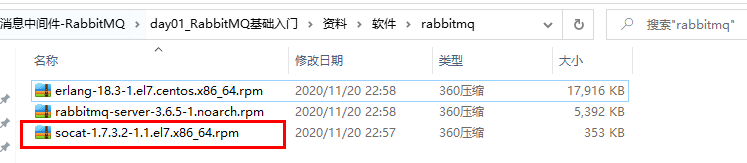
然后在该表格的下面，点击Erlang Solution可以下载Erlang安装包：



点击后，会跳转到erlang的下载网站：

<https://www.erlang-solutions.com/resources/download.html>

这里我们就用老师给的资料在虚拟机上安装RabbitMQ，就不去官网下载了，因为有一个安装包我不知道在哪下载：

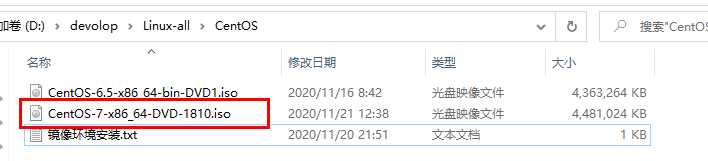


## 二、安装依赖环境

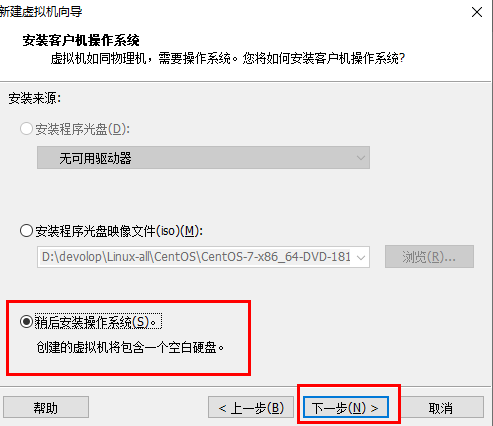
### 1、安装CentOS7

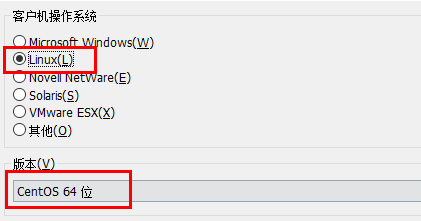
注意:

请使用资料里提供的CentOS-7-x86\_64-DVD-1810.iso 安装虚拟机.

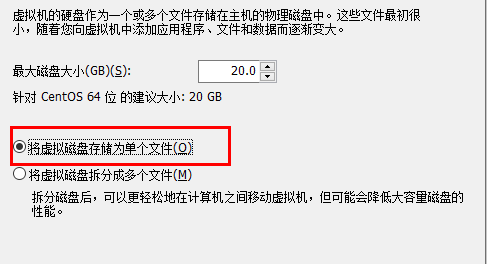


#### 1.1安装虚拟机

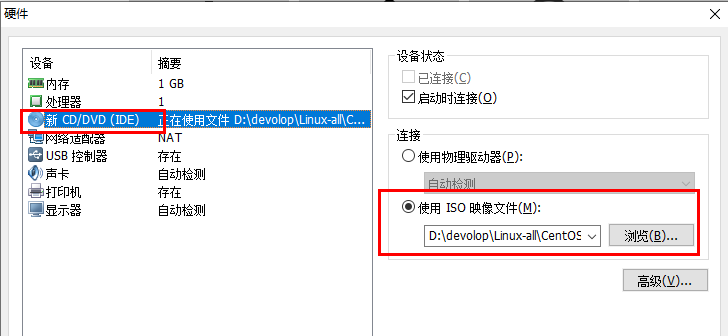








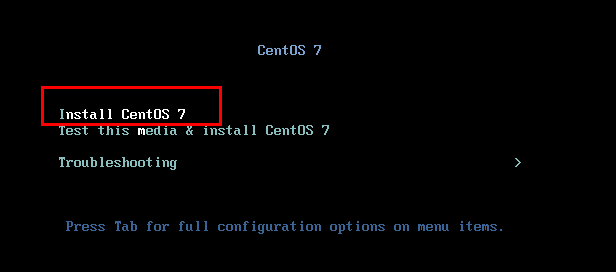
然后自定义硬件，选择CentOS7镜像



关闭以后，虚拟机就创建好了

#### 1.2安装CentOS7

选第一个Install Cent OS 7回车





拉动右侧滚动条 。点击Installation Destination选项，选择磁盘及分区类型



设置网络和主机名，右侧启用网卡后变为“打开”，点击“完成”

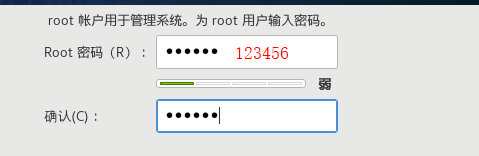


开始安装：

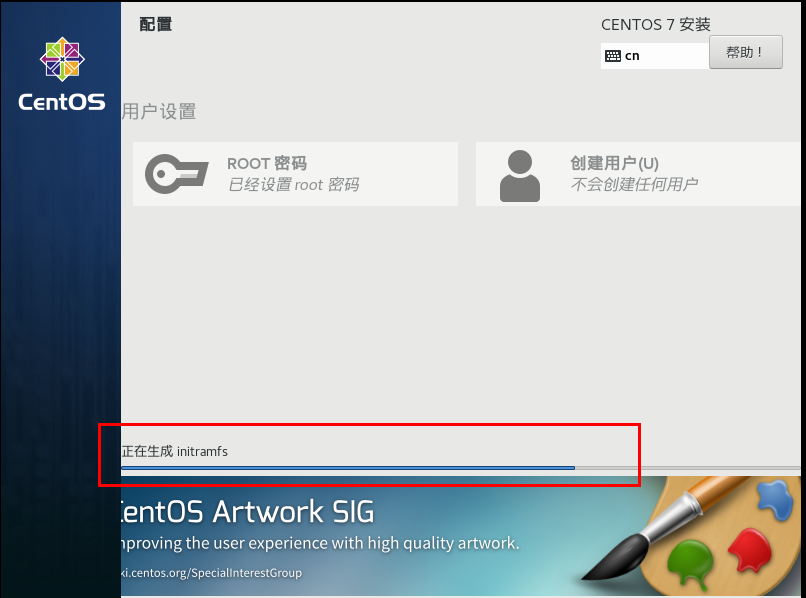


点击ROOT PASSWORD 设置密码。如果密码较弱。点击两次“完成”。

我这里将密码设置为：123456

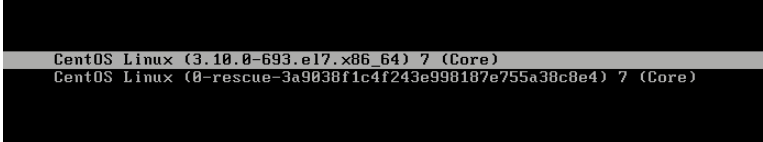


然后等待安装完成

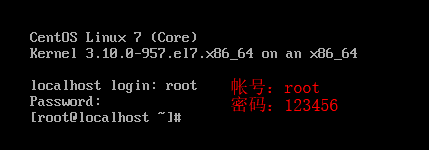


安装完成 点击“重启”



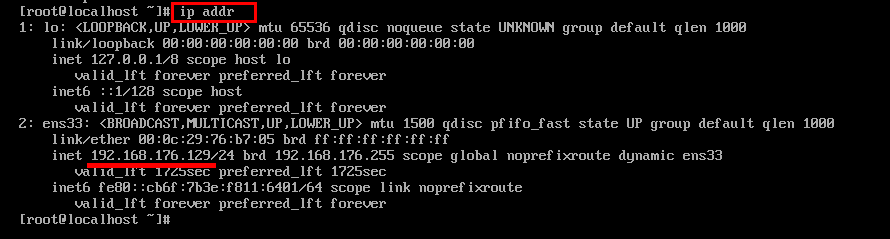


登陆：



CentOS7查看IP的命令：

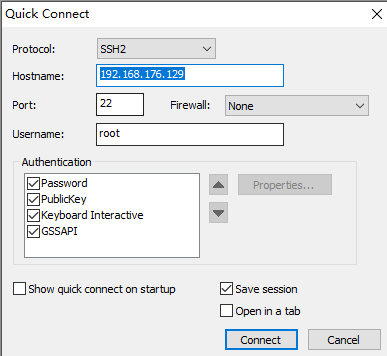
·ip addr



IP：192.168.176.129

#### 1.3使用CRT连接

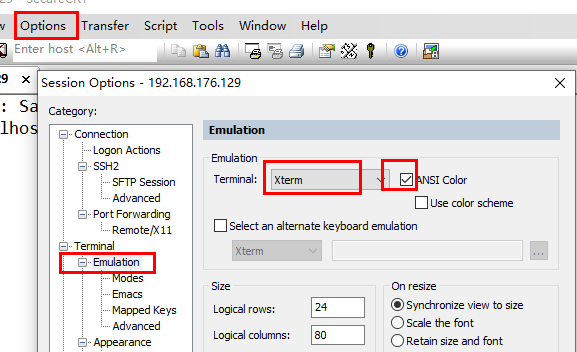
初次使用CRT连接：



调用以前的配色方案：

Options => Sessions options => Terminal => Emulation

在 Terminal下拉列表下选择X-term，勾选 ANSI Color。

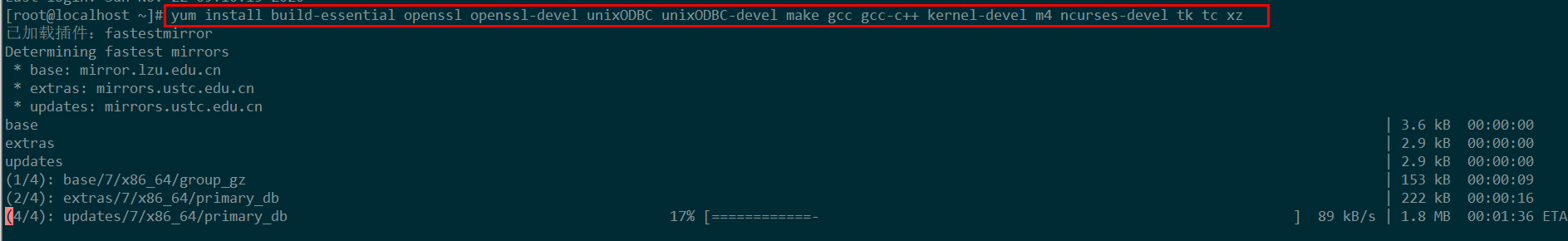


OK！

### 2、安装依赖环境

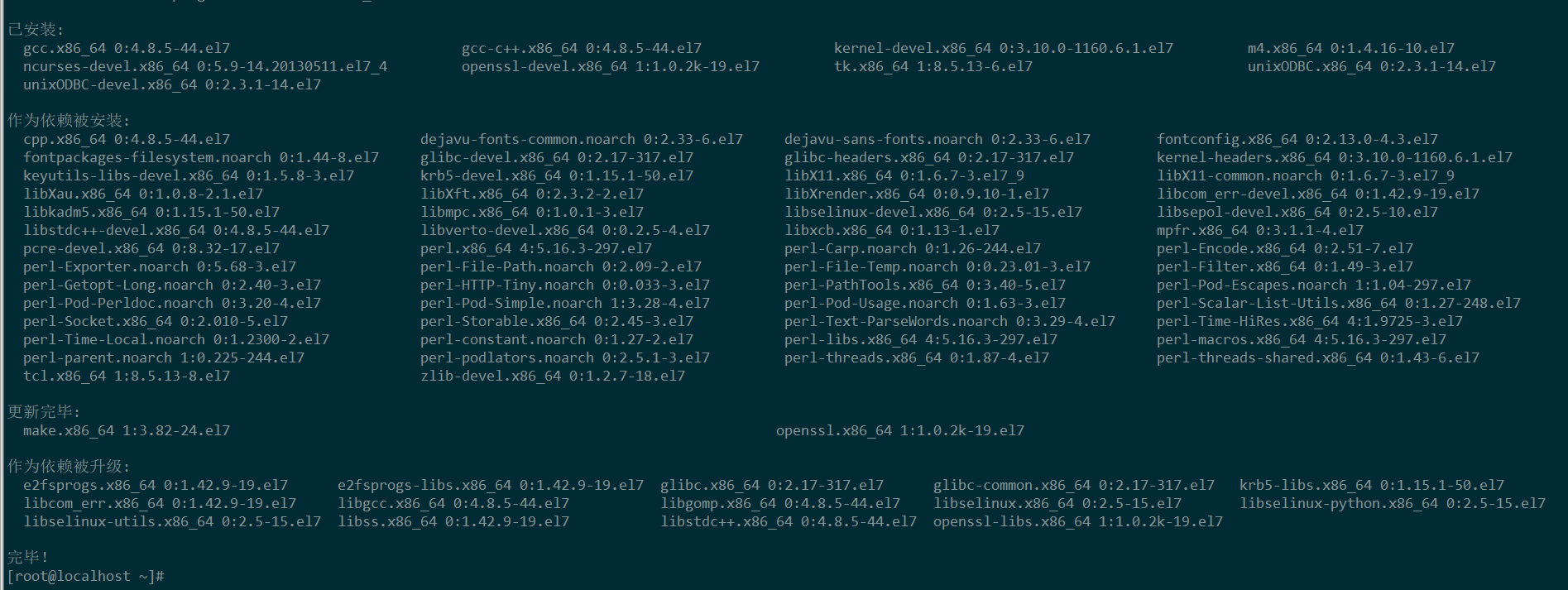
在线安装依赖环境：

yum install build-essential openssl openssl-devel unixODBC unixODBC-devel make gcc gcc-c++ kernel-devel m4 ncurses-devel tk tc xz



这一步必须联网，将会安装许多东西

安装完成截图：



### 3、细节补充

安装后无法识别ifconfig命令

如果是ifconfig命令不存在的话，输入命令安装：

yum upgrade

yum install net-tools

实测有效！

## 三、安装Erlang

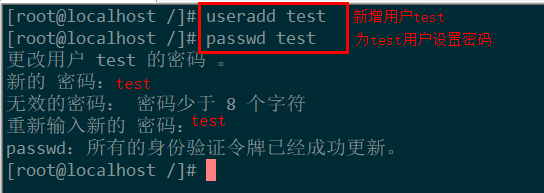
### 1、正常安装

(1) 添加用户test

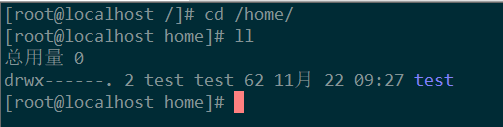
useradd test

(2) 为test用户设置密码，比如也设为test

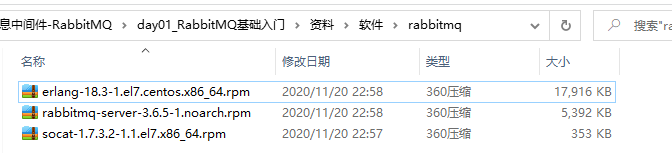
passwd test



然后在/home/目录下就会生成一个新的目录：test



(3) 将下面3个安装包上传到/home/test/rabbit目录下

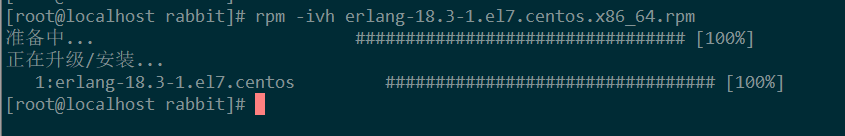


上传好后：



(4) 直接在/rabbit目录下安装

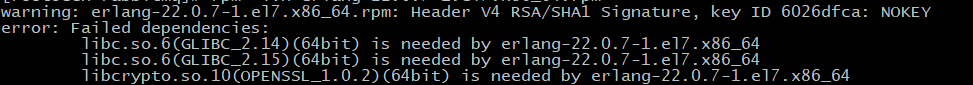
rpm -ivh erlang-18.3-1.el7.centos.x86\_64.rpm



可以看到安装erlang这一步目前我自己是成功了的。

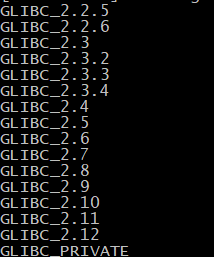
### 2、细节补充

***如果出现如下错误：***



说明gblic 版本太低。我们可以查看当前机器的gblic 版本

strings /lib64/libc.so.6 | grep GLIBC



当前最高版本2.12，需要2.15.所以需要升级glibc

**·使用yum更新安装依赖：**

sudo yum install zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel readline-devel tk-devel gcc make -y

**·下载rpm包**

**·安装rpm包：**

sudo rpm -Uvh \*-2.17-55.el6.x86\_64.rpm --force --nodeps

**·安装完毕后再查看glibc版本，发现glibc版本已经到2.17了**

strings /lib64/libc.so.6 | grep GLIBC

## 四、安装RabbitMQ

### 1、安装

直接在/home/test/rabbit/目录下安装即可

(1) 安装socat

rpm -ivh socat-1.7.3.2-5.el7.lux.x86\_64.rpm

出现报错：

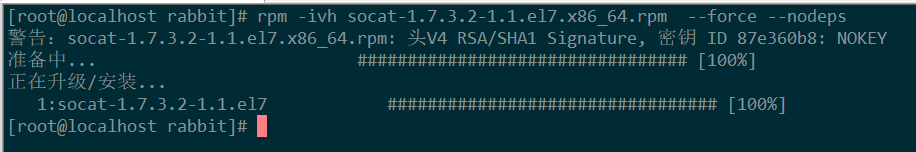


在网上搜了一下，查到的说明如下：  
网上资料说这是由于yum安装了旧版本的GPG keys造成的

解决办法：

在rpm 语句后面加上 --force --nodeps 即原本为 rpm -ivh \*.rpm 现在改成 rpm -ivh \*.rpm --force --nodeps就可以了。 nodeps的意思是忽视依赖关系。因为各个软件之间会有多多少少的联系。有了这两个设置选项就忽略了这些依赖关系，强制安装或者卸载

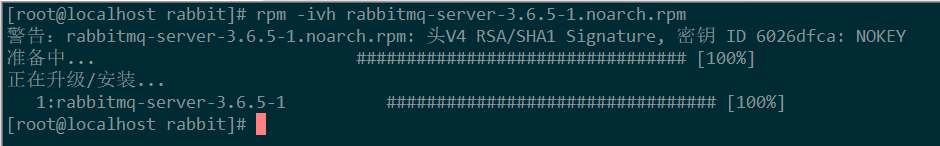
我自己的操作截图：



问题解决

(2) 安装RabbitMQ

rpm -ivh rabbitmq-server-3.6.5-1.noarch.rpm



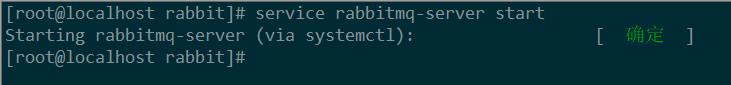
这样RabbitMQ就安装好了

### 2、启动服务

service rabbitmq-server start # 启动服务

service rabbitmq-server stop # 停止服务

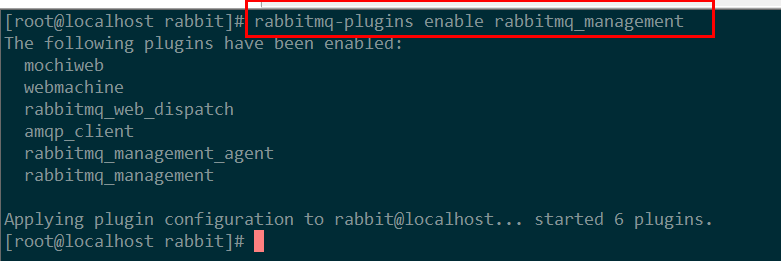
service rabbitmq-server restart # 重启服务



### 3、开启管理界面及配置

# 开启管理界面

rabbitmq-plugins enable rabbitmq\_management



# 修改默认配置信息

vim /usr/lib/rabbitmq/lib/rabbitmq\_server-3.6.5/ebin/rabbit.app

**解决了vim的问题时候，修改配置信息截图：**

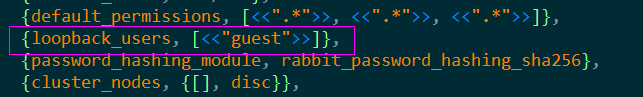
编辑rabbbit.app，在里面可以看到，默认监听端口是5672，这个一般不去修改.

至于管理控制台，则是在端口前面加了一个1，即：15672

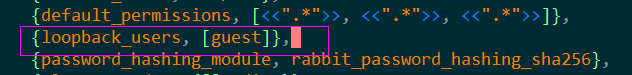


我们将后面的loopback\_users 中的 <<"guest">>改为只保留guest

初始：



修改后：



修改完管理界面的配置之后，记得必须要重启RabbitMQ服务，配置才会生效

Service rabbitmq-server restart

修改完成后，因为RabbitMQ已经启动了，我们去浏览器访问：

虚拟机IP:15672

第一次访问，会访问不到，因为防火墙没有关闭

### 4、设置防火墙

**注意CentOS7与CentOS6中设置防火墙的命令不一样**

**永久关闭防火墙（当时不生效，要重启虚拟机才会生效）：**

·chkconfig iptables off -- centos6

·systemctl disable firewalld -- centos7

**暂时关闭防火墙，（当时立马就会生效）：**

·service iptables stop -- centos6

·systemctl stop firewalld -- centos7

**重启防火墙：**

·service iptables restart -- centos6

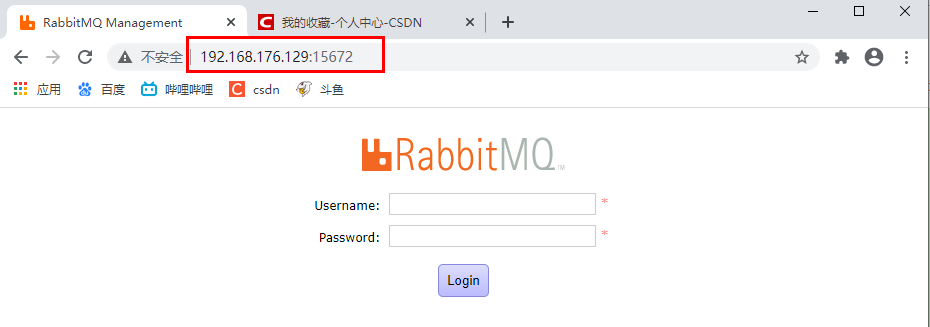
·systemctl enable firewalld -- centos7

**# 查看防火墙状态**

·service iptables status -- centos6

·systemctl status firewalld -- centos7

关闭防火墙后，重新在浏览器访问：虚拟机IP:15672

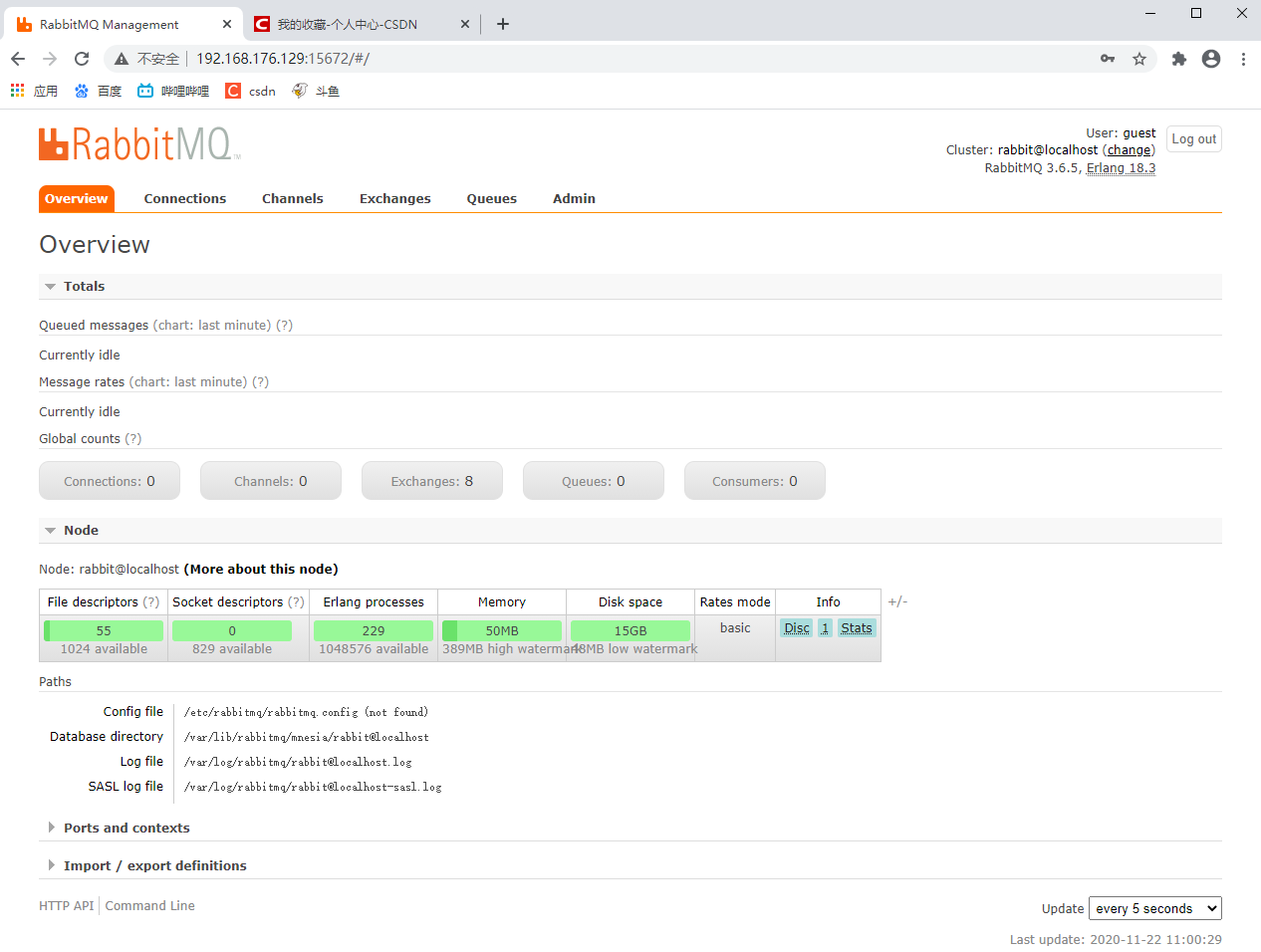


访问成功！

用户名和密码都是guest

这是之前在rabbit.app文件中设置的，参考：[6、开启管理界面及配置](#_6、开启管理界面及配置)

登陆成功页面：



能看到这个页面，就表示RabbitMQ，以及其管理控制台全都安装成功了。

### 5、Vim命令

***解决CentOS7不能使用vim命令的问题***

使用vim编辑是出现报错：

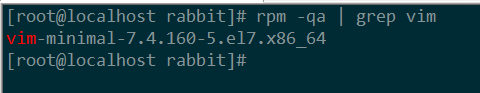


解决办法：

因为CentOS7默认没有安装vim，因此不能使用vim命令。

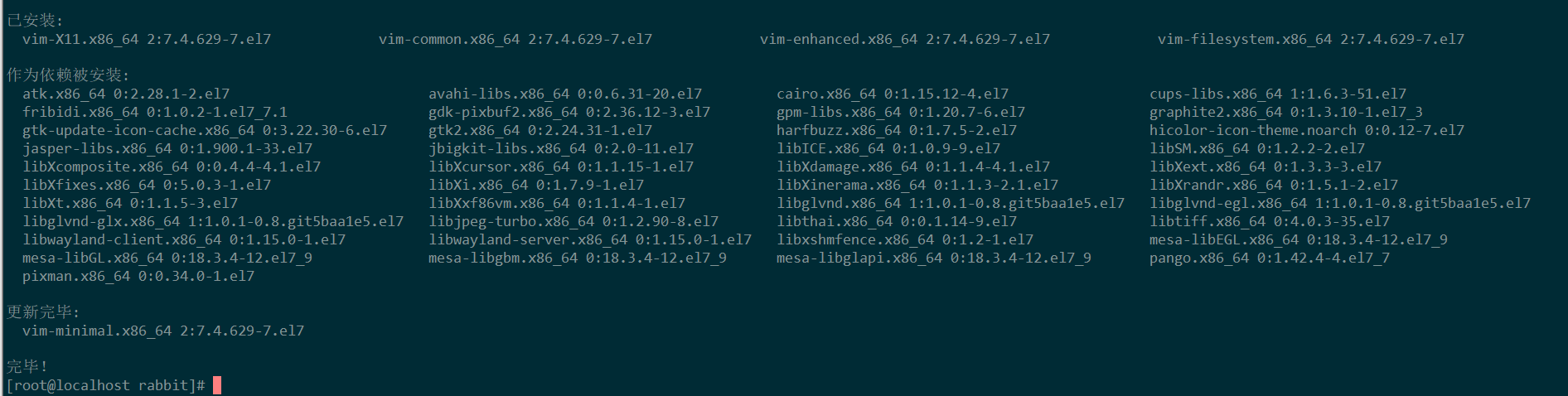
我们可以查看一下

rpm -qa | grep vim



然后输入命令，下载安装（联网）：

yum -y install vim\*



然后就可以使用vim命令了

此方法亲测有效！

### 6、相关目录

(1) 压缩包的目录，之前我们是上传到了这个目录下：

/home/test/rabbit

安装完成后，可以将压缩包全删了

rm -rf 安装包全名

(2) RabbitMQ默认安装目录

/usr/share/doc/rabbitmq-server-3.6.5/



里面的rabbitmq.config.example是一个配置文件，不过已经被我们复制到另一个目录下了，在(3)中进行说明。

复制的原因：根据管理界面的要求，进行复制的

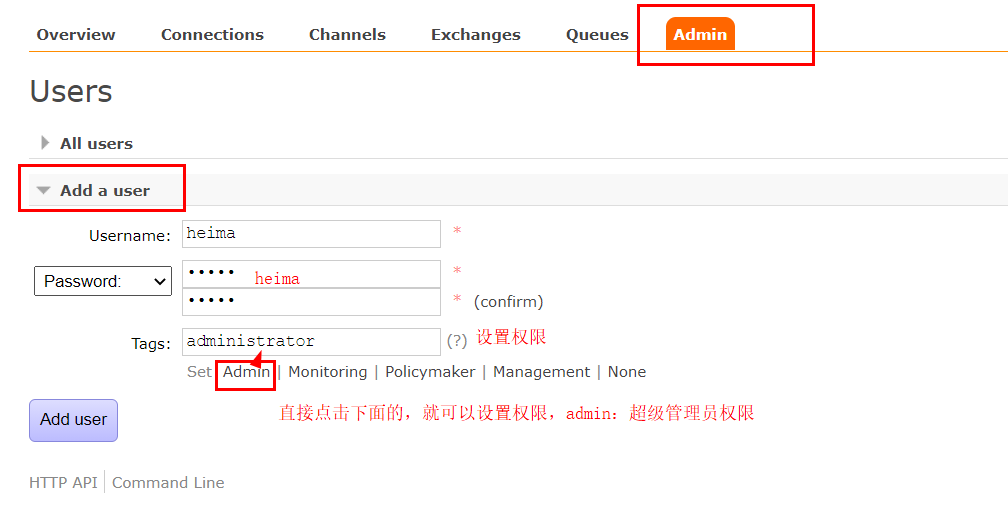
(3) RabbitMQ配置文件

/etc/rabbitmq/rabbitmq.config

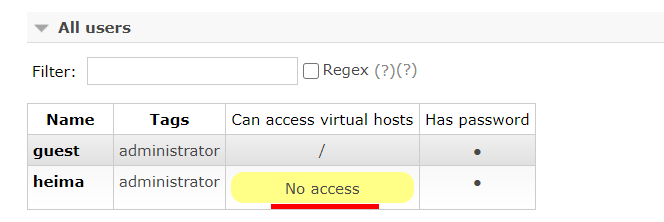
## 五、管控台的使用

***RabbitMQ管控台的使用***

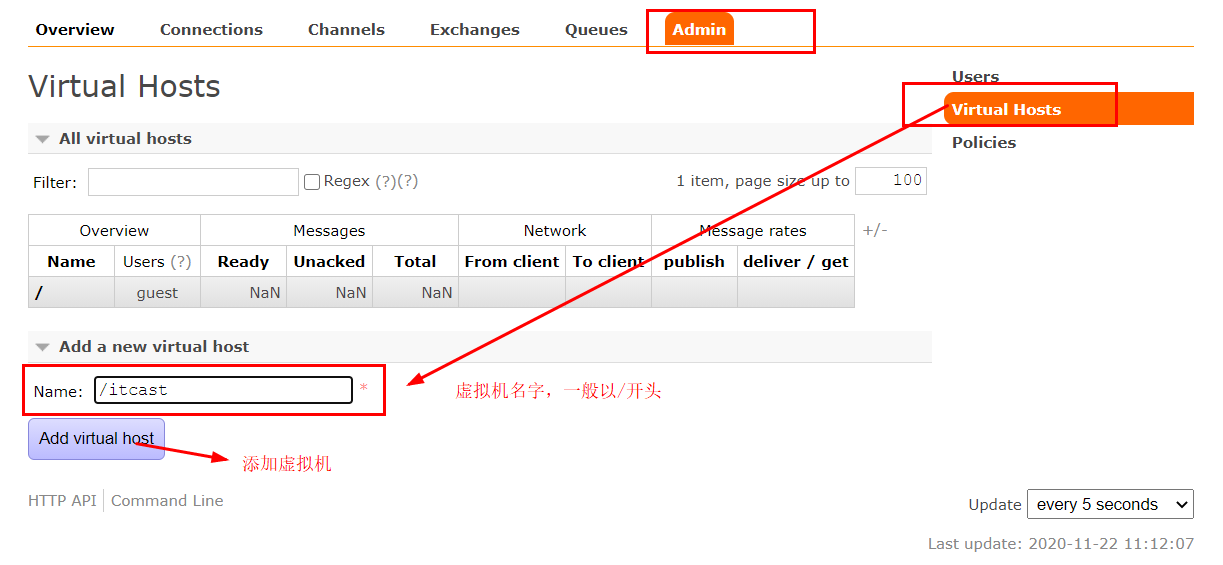
### 1、添加用户



可以看到，目前heima用户可以访问的虚拟机是0个。

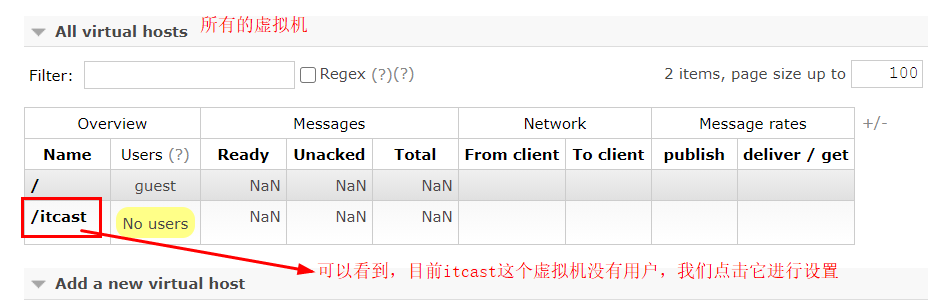


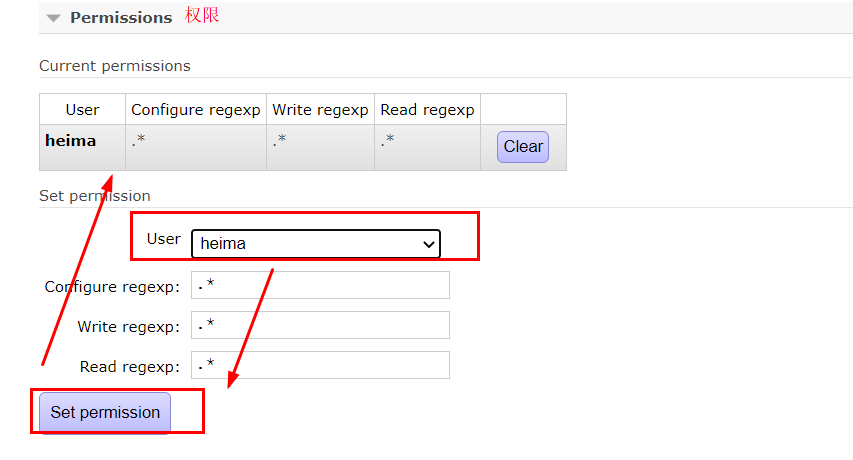
### 2、添加虚拟机



### 3、设置权限

点击“itcast”，对该虚拟机进行设置

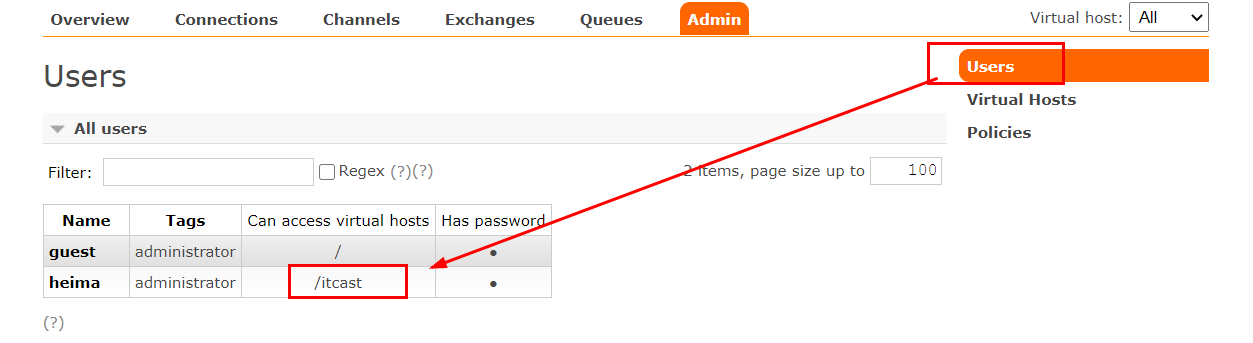




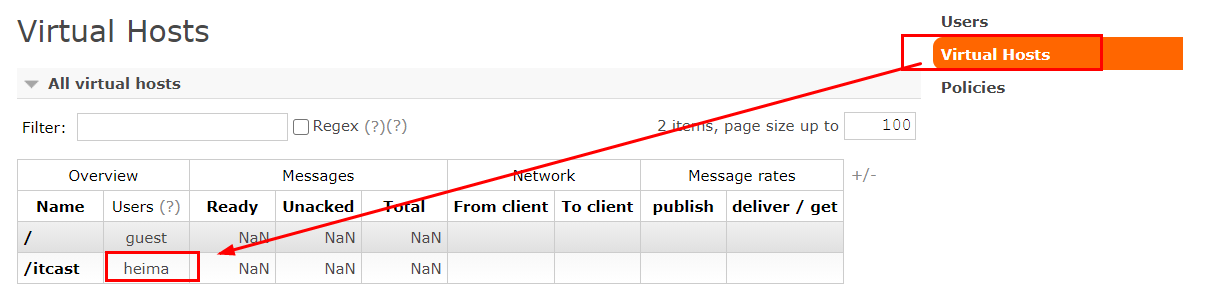
如果希望删除heima的话，在Set permission下面也有一个Delete this vhost

（即删除该虚拟机的用户）

设置好权限后，我们再回到Users，就可以看到heima用户下面可以访问的虚拟机多了一个itcast：



此外在虚拟机中也多了一个用户：heima：

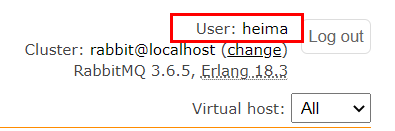


### 4、切换登陆用户

右上角Log out是登出，我们点击后，切换到heima用户登陆

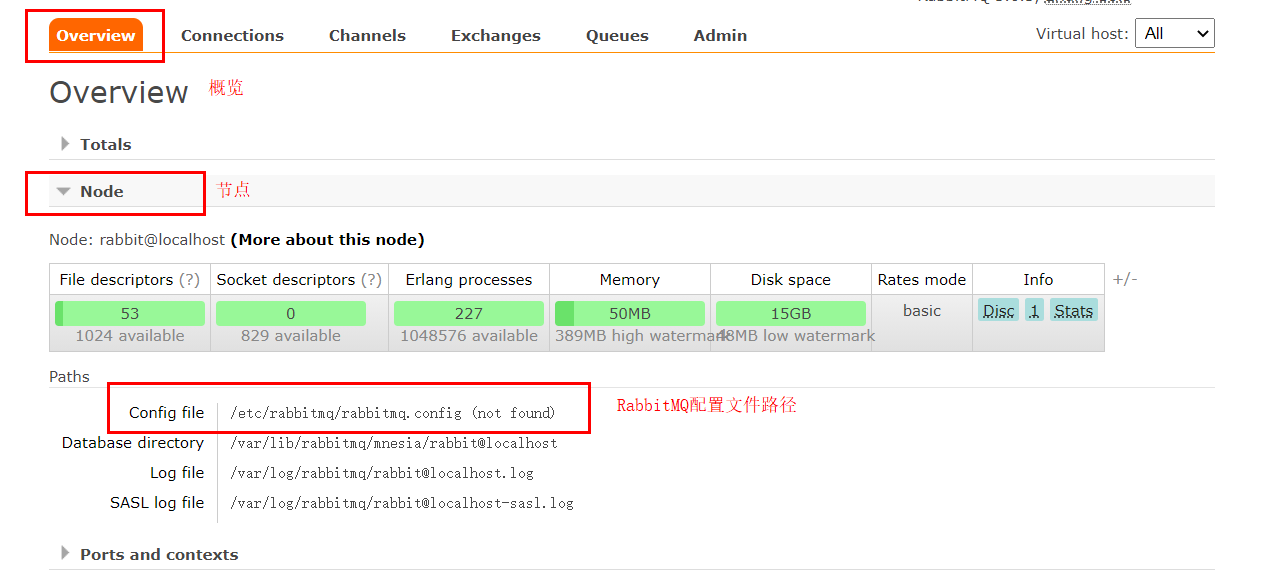


可以看到右上角的user变为了heima：



### 5、修改配置文件路径

在管控台的Overview（概览）——Node（节点）：



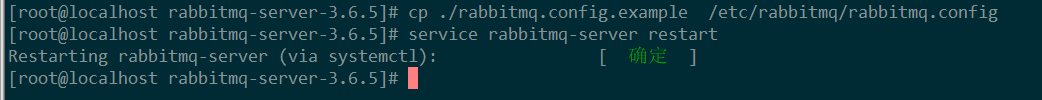
我们需要将RabbitMQ的配置文件，复制一份，粘贴到/etc/rabbitmq/目录下，这样管控台才能看到。

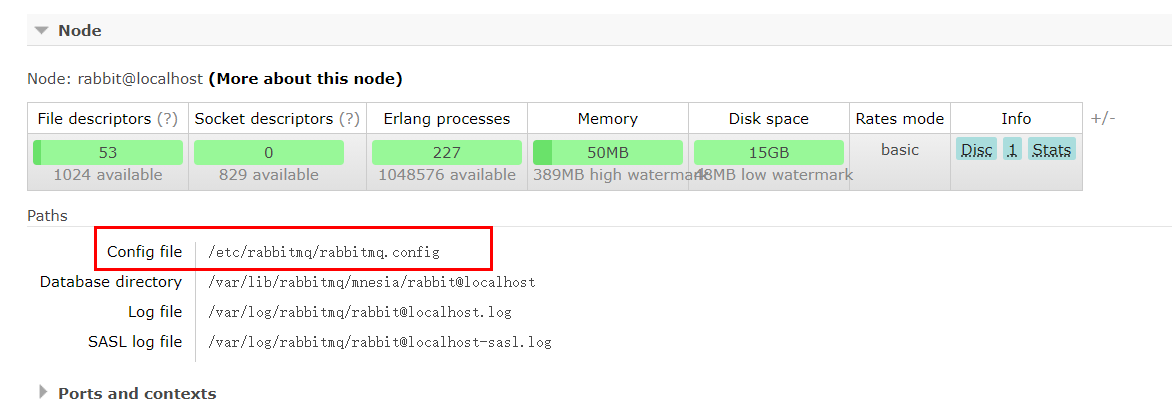
cd /usr/share/doc/rabbitmq-server-3.6.5/

cp ./rabbitmq.config.example /etc/rabbitmq/rabbitmq.config

然后重启RabbitMQ

service rabbitmq-server restart

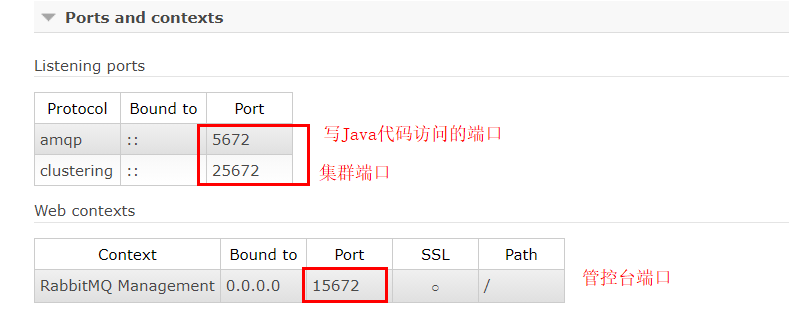


然后我们再来看管控台的节点信息：  


现在管控台就能看到RabbitMQ的配置文件了。

### 6、端口介绍

管控台的Overview——Ports and contexts：



### 7、配置文件的导入/出

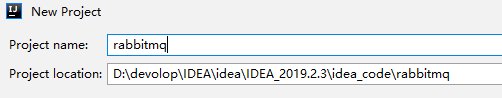
管控台的Overview—— Import/export definations



# 第三章 RabbitMQ快速入门

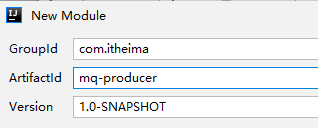
## 一、创建工程

先创建一个空的工程，取名为rabbitmq



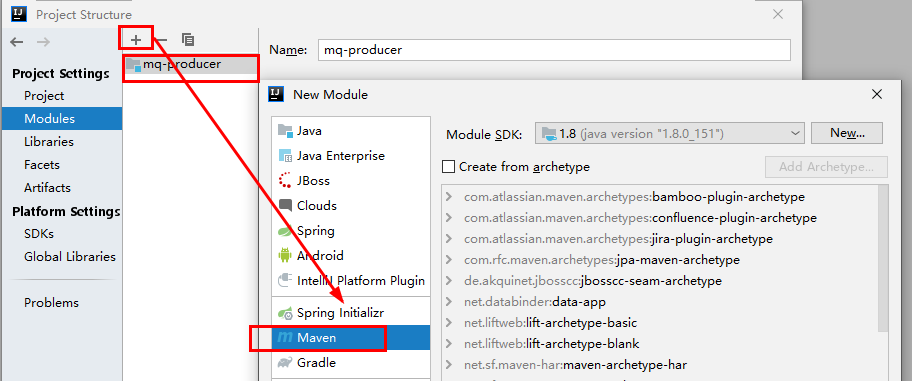
然后在rabbitmq项目里面创建一个模块，maven的SE工程，取名为：

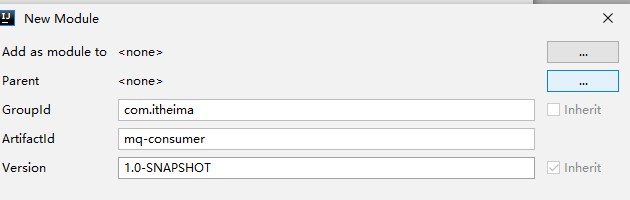
mq-producer，作为mq生产者。

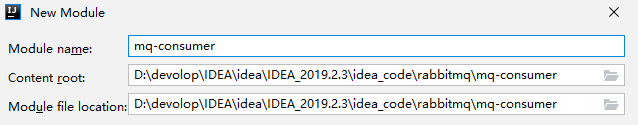




然后直接再创建第二个模块，也是maven的SE工程，取名为：mq-consumer，作为mq消费者。







这样，producer和consumer就相当于是rabbitmq目录下的两个独立的项目了，因为rabbitmq是一个空的项目，因此它是没有依赖的，并不是父工程，只是相当于一个空的目录而已。

## 二、pom

资料中已经准备好了，我们只需要复制过去即可：

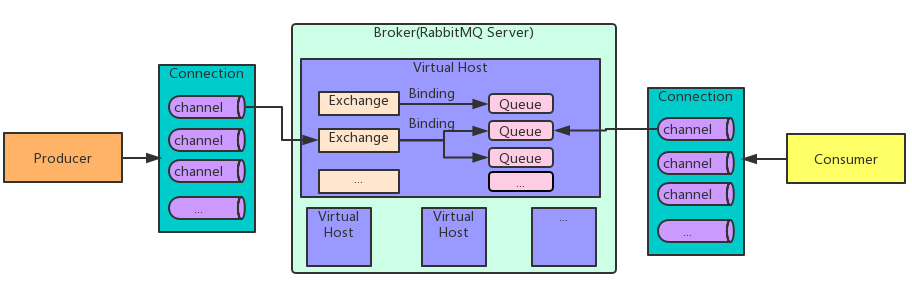


生产者和消费者的pom都是复制下面这段代码

<**dependencies**>  
 <!--rabbitmq java 客户端-->  
 <**dependency**>  
 <**groupId**>com.rabbitmq</**groupId**>  
 <**artifactId**>amqp-client</**artifactId**>  
 <**version**>5.6.0</**version**>  
 </**dependency**>  
</**dependencies**>  
  
<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>3.8.0</**version**>  
 <**configuration**>  
 <**source**>1.8</**source**>  
 <**target**>1.8</**target**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

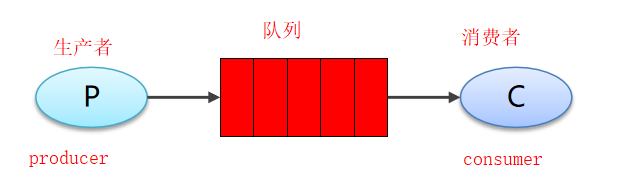
## 三、思路框架

RabbitMQ基础架构：



RabbitMQ有5种工作模式，而在入门程序中要使用的就是简单模式。

简单模式：



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：消息的接收者，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

在简单模式的broker(中间件)中，看似没有设置exchange(交换机)，其实是有的，只不过使用的是默认交换机。

## 四、生产者

将mq-producer模块作为生产者。在com.itheima.producer包下创建一个类，取名为：Producer\_HelloWorld。

### 1、代码

*/\*\*  
 \* mq生产者\_简单模式  
 \*/*

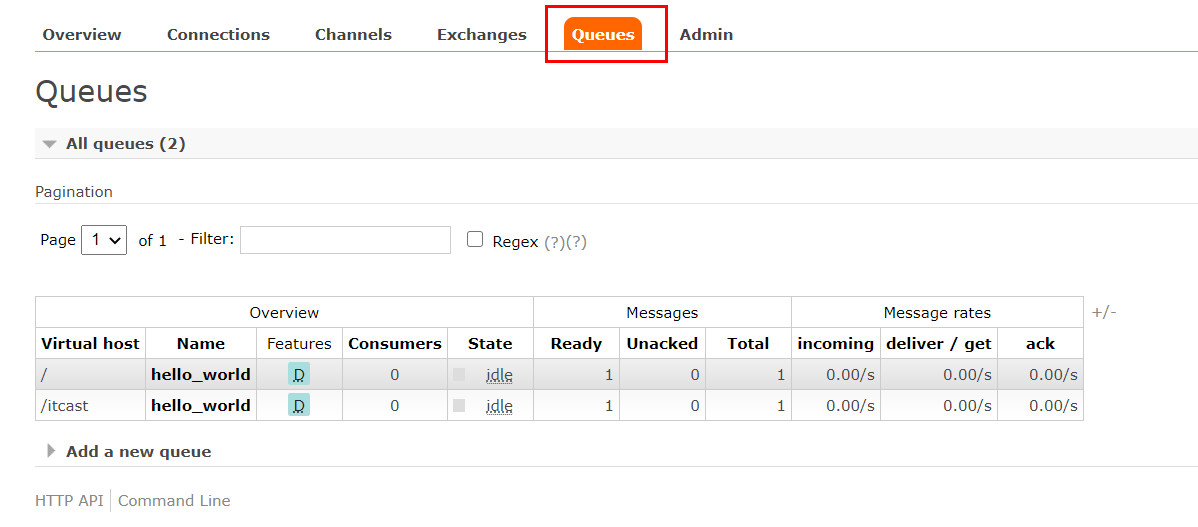
**public class** Producer\_HelloWorld {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel  
 Channel channel = connection.createChannel();  
 //5. 创建队列 Queue  
 /\*  
 queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments)  
 参数：  
 1.queue:队列名称  
 2.durable:是否持久化，当mq重启之后，还存在(持久化)。  
 3.exclusive:  
 \* 是否独占。只能有一个消费者监听这个队列(独占)  
 \* 当Connection关闭时，是否删除队列  
 4.autoDelete:是否自动删除。当没有Consumer时，自动删除掉  
 5.arguments:参数。  
 \*/  
 //如果没有一个名字叫做“hello\_world”的队列，则会创建该队列，如果有则不会创建  
 channel.queueDeclare(**"hello\_world"**, **true**, **false**, **false**, **null**);  
 /\*  
 basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body)  
 参数：  
 1.exchange:交换机名称。简单模式下，交换机会使用默认的""  
 2.routingKey:路由名称  
 3.props:配置信息  
 4.body:发送消息数据  
 \*/  
 //定义一个消息数据  
 String body = **"hello rabbitmq~~~"**;  
 //6.发送消息  
 channel.basicPublish(**""**,**"hello\_world"**,**null**,body.getBytes());  
 //7.释放资源  
 channel.close();  
 connection.close();  
 }  
}

### 2、测试

一开时执行这个main方法时，一直报错，总是无法创建连接。

后来我将连接的虚拟机改为“/”，重启启动main方法，发现执行成功了。

然后又将虚拟机“/itcast”删除，重新创建，并增加管理用户heima，重启启动main方法，发现终于成功了。

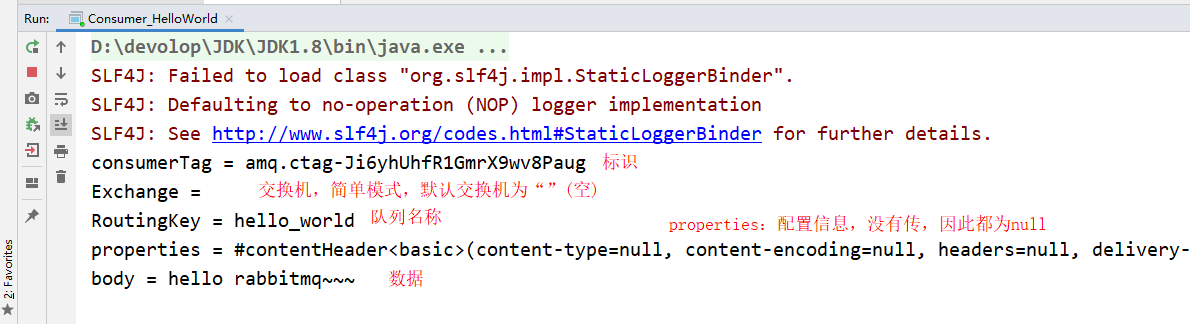


## 五、消费者

### 1、代码

*/\*\*  
 \* mq消费者  
 \*/***public class** Consumer\_HelloWorld {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel，注意消费者与生产者的Channel可不是同一个，看RabbitMQ的架构图就知道  
 Channel channel = connection.createChannel();  
 //5. 创建队列 Queue  
 /\*  
 queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments)  
 参数：  
 1.queue:队列名称  
 2.durable:是否持久化，当mq重启之后，还存在(持久化)。  
 3.exclusive:  
 \* 是否独占。只能有一个消费者监听这个队列(独占)  
 \* 当Connection关闭时，是否删除队列  
 4.autoDelete:是否自动删除。当没有Consumer时，自动删除掉  
 5.arguments:参数。  
 \*/  
 //如果没有一个名字叫做“hello\_world”的队列，则会创建该队列，如果有则不会创建  
 channel.queueDeclare(**"hello\_world"**, **true**, **false**, **false**, **null**);  
  
 //上面部分的代码，与生产者完全一致，其实上面的队列可以不用创建，因为生产者里面已经创建了，不过这里创建也没关系  
  
 //6. 创建Consumer对象  
 Consumer consumer = **new** DefaultConsumer(channel) {  
 /\*  
 回调方法，当收到消息后，会自动执行该方法  
 1.consumerTag:标识  
 2.envelope:获取一些信息，交换机，路由key...  
 3.properties:配置信息  
 4.body:数据  
 \*/  
 @Override  
 **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  
 System.***out***.println(**"consumerTag = "** + consumerTag);  
 System.***out***.println(**"Exchange = "** + envelope.getExchange());  
 System.***out***.println(**"RoutingKey = "** + envelope.getRoutingKey());  
 System.***out***.println(**"properties = "** + properties);  
 System.***out***.println(**"body = "** + **new** String(body));  
 }  
 };  
 /\*  
 basicConsume(String queue, boolean autoAck, Consumer callback)  
 参数:  
 1.queue:队列名称 hello\_world  
 2.autoAck:是否自动确认（确认消息是否收到）  
 3.callback:回调对象  
 \*/  
 //7. 接收消息  
 channel.basicConsume(**"hello\_world"**,**true**,consumer);  
 //8.消费者不要关闭资源，否则无法监听  
 }  
}

### 2、测试



消费者没有关闭资源，因此可以看到方法的一直在运行，没有结束，上图中的红格子一直亮着。

到此，快速入门的代码就算完成了。

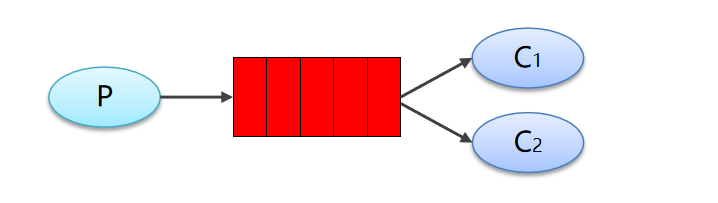
# 第四章 RabbitMQ的工作模式

## 一、工作队列模式

所谓工作模式，其实就是消息路由分发的方式

### 1、模式说明

Work queues 工作队列模式



* Work Queues：

与入门程序的简单模式相比，多了一个或多个消费端，多个消费端共同消费同一个队列中的消息。

* 应用场景：

对于任务过重或任务较多情况使用工作队列可以提高任务处理的速度。例如：短信服务部署多个，只需要有一个节点成功发送即可。

上图中的多个消费者对于消息的接收是竞争关系，即同一条消息只能被其中一个消费者接收到。

至于代码，与简单模式的代码几乎也是一样的，只不过是多加了一个消费者而已。

下面我们来演示一下

### 2、生产者

队列名称：work\_queues

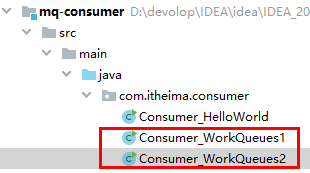
发送10条消息

*/\*\*  
 \*生产者\_工作队列模式  
 \*/*

**public class** Producer\_WorkQueues {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel  
 Channel channel = connection.createChannel();  
 //5. 创建队列 Queue  
 /\*  
 queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments)  
 参数：  
 1.queue:队列名称  
 2.durable:是否持久化，当mq重启之后，还存在(持久化)。  
 3.exclusive:  
 \* 是否独占。只能有一个消费者监听这个队列(独占)  
 \* 当Connection关闭时，是否删除队列  
 4.autoDelete:是否自动删除。当没有Consumer时，自动删除掉  
 5.arguments:参数。  
 \*/  
 //如果没有一个名字叫做“hello\_world”的队列，则会创建该队列，如果有则不会创建  
 channel.queueDeclare(**"work\_queues"**, **true**, **false**, **false**, **null**);  
 /\*  
 basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body)  
 参数：  
 1.exchange:交换机名称。简单模式下，交换机会使用默认的""  
 2.routingKey:路由名称  
 3.props:配置信息  
 4.body:发送消息数据  
 \*/  
 //定义消息数据  
 String body = **null**;  
 //6.发送10条消息  
 **for** (**int** i = 0; i <= 10; i++) {  
 body = i + **"hello rabbitmq~~~"**;  
 channel.basicPublish(**""**,**"work\_queues"**,**null**,body.getBytes());  
 }  
 //7.释放资源  
 channel.close();  
 connection.close();  
 }  
}

### 3、消费者

消费者有两个，除了类名不同以外，main方法代码完全一致

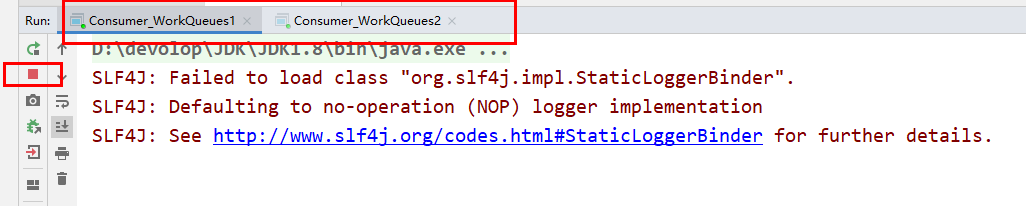


*/\*\*  
 \* mq消费者1  
 \*/***public class** Consumer\_WorkQueues1 {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel，注意消费者与生产者的Channel可不是同一个，看RabbitMQ的架构图就知道  
 Channel channel = connection.createChannel();  
 //5. 创建队列 Queue  
 /\*  
 queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments)  
 参数：  
 1.queue:队列名称  
 2.durable:是否持久化，当mq重启之后，还存在(持久化)。  
 3.exclusive:  
 \* 是否独占。只能有一个消费者监听这个队列(独占)  
 \* 当Connection关闭时，是否删除队列  
 4.autoDelete:是否自动删除。当没有Consumer时，自动删除掉  
 5.arguments:参数。  
 \*/  
 //如果没有一个名字叫做“hello\_world”的队列，则会创建该队列，如果有则不会创建  
 channel.queueDeclare(**"work\_queues"**, **true**, **false**, **false**, **null**);  
  
 //上面部分的代码，与生产者完全一致，其实上面的队列可以不用创建，因为生产者里面已经创建了，不过这里创建也没关系  
  
 //6. 创建Consumer对象  
 Consumer consumer = **new** DefaultConsumer(channel) {  
 /\*  
 回调方法，当收到消息后，会自动执行该方法  
 1.consumerTag:标识  
 2.envelope:获取一些信息，交换机，路由key...  
 3.properties:配置信息  
 4.body:数据  
 \*/  
 @Override  
 **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  
 /\*  
 System.out.println("consumerTag = " + consumerTag);  
 System.out.println("Exchange = " + envelope.getExchange());  
 System.out.println("RoutingKey = " + envelope.getRoutingKey());  
 System.out.println("properties = " + properties);  
 \*/  
 System.***out***.println(**"body = "** + **new** String(body));  
 }  
 };  
 /\*  
 basicConsume(String queue, boolean autoAck, Consumer callback)  
 参数:  
 1.queue:队列名称 hello\_world  
 2.autoAck:是否自动确认（确认消息是否收到）  
 3.callback:回调对象  
 \*/  
 //7. 接收消息  
 channel.basicConsume(**"work\_queues"**,**true**,consumer);  
 //8.消费者不要关闭资源，否则无法监听  
 }  
}

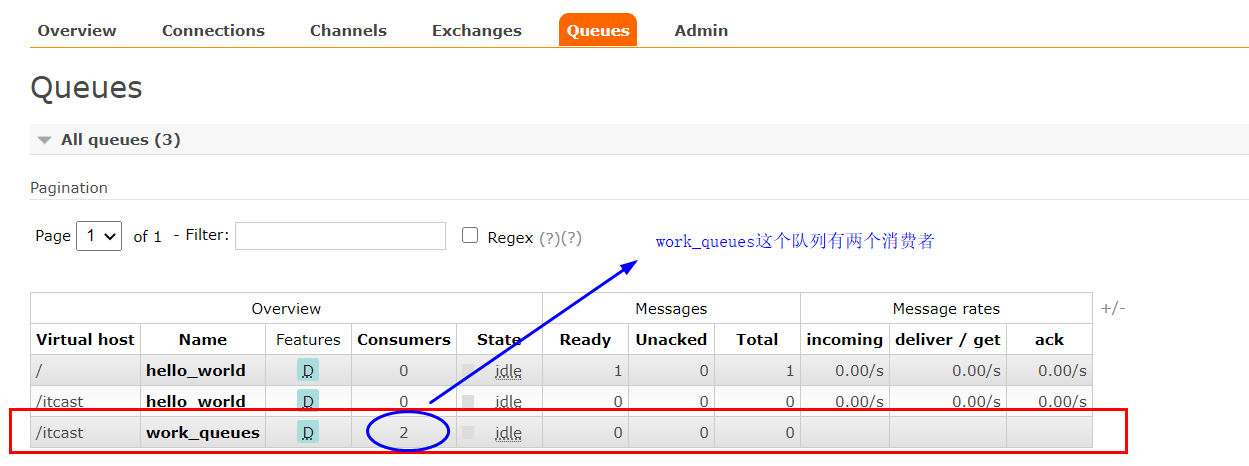
### 4、测试

先启动两个消费者，即执行Consumer\_WorkQueues1、

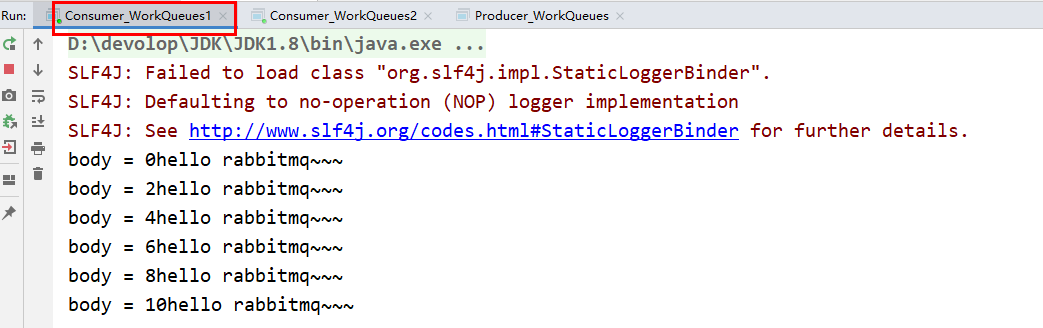
Consumer\_WorkQueues2的main方法：

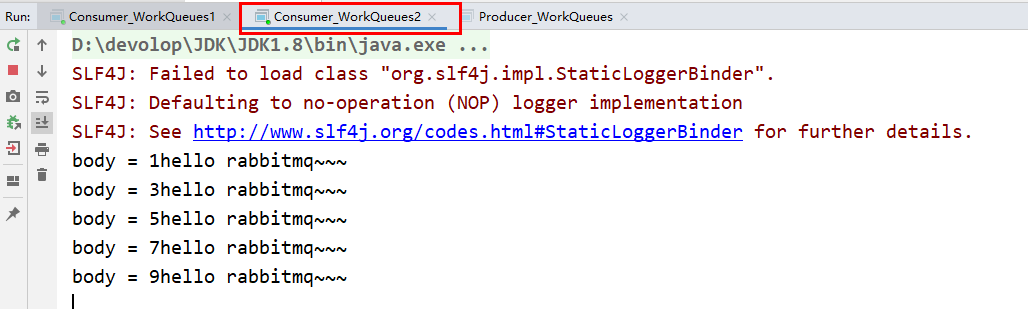


在管控台，可以看到work\_queues队列有两个消费者：



然后启动生产者的main方法，生产10条消息，然后分别看两个消费者接收消息的情况：





不小心发了11条消息，问题不大。

## 二、订阅模式

### 1、模式说明



在订阅模型中，多了一个 Exchange 角色，而且过程略有变化：

* P：生产者。也就是要发送消息的程序，但是不再发送到队列中，而是发给X（交换机）
* C：消费者。消息的接收者，会一直等待消息到来
* Queue：消息队列。接收消息、缓存消息
* Exchange：交换机（X）。一方面，接收生产者发送的消息。另一方面，知道如何处理消息，例如递交给某个特别队列、递交给所有队列、或是将消息丢弃。到底如何操作，取决于Exchange的类型。Exchange有常见以下3种类型：

**① Fanout：广播。**将消息交给所有绑定到交换机的队列

**② Direct：定向。**把消息交给符合指定routing key 的队列

**③ Topic：通配符。**把消息交给符合routing pattern（路由模式） 的队列

Exchange（交换机）只负责转发消息，不具备存储消息的能力，因此如果没有任何队列与 Exchange 绑定，或者没有符合路由规则的队列，那么消息会丢失！

### 2、生产者

我们再创建一个类，作为生产者，取名为：Producer\_PubSub

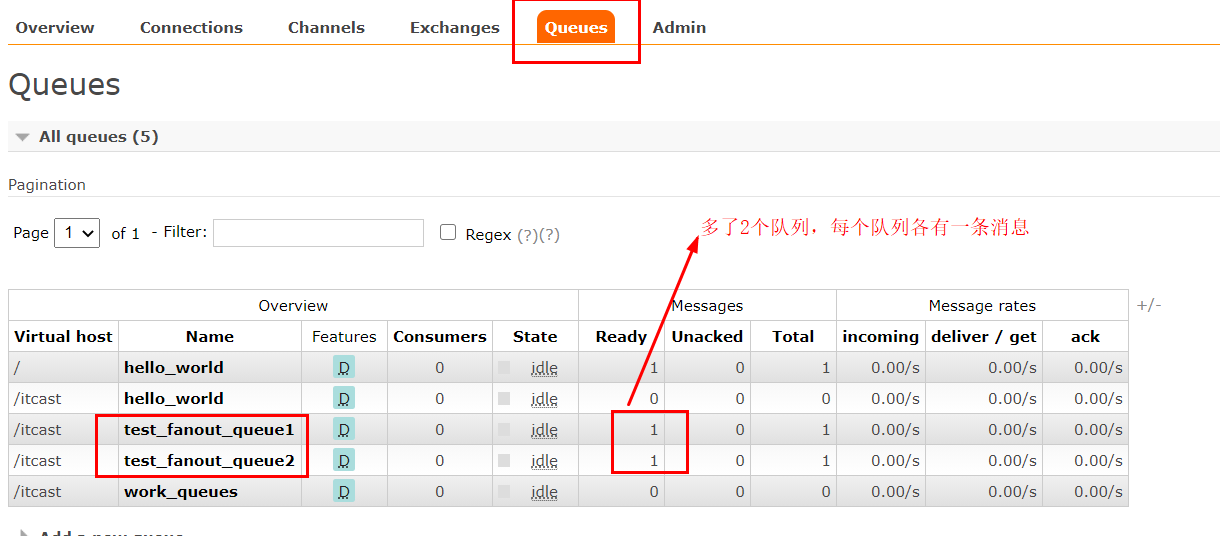
#### 2.1代码

Producer\_PubSub：

*/\*\*  
 \* mq生产者\_订阅模式  
 \*/***public class** Producer\_PubSub {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel  
 Channel channel = connection.createChannel();  
 /\*  
 exchangeDeclare(String exchange, BuiltinExchangeType type, boolean durable  
 , boolean autoDelete, boolean internal, Map<String, Object> arguments)  
 参数：  
 1.exchange:交换机名称  
 2.type:交换机类型  
 (1)DIRECT("direct"), :定向  
 (2)FANOUT("fanout"), :扇形（广播），发送消息到每一个与之绑定的队列  
 (3)TOPIC("topic"), :通配符方式  
 (4)HEADERS("headers"); :参数匹配（用的很少，不做介绍）  
 3.durable:是否持久化  
 4.autoDelete:是否自动删除  
 5.internal:内部使用。一般false  
 6.arguments：参数  
 \*/  
 String exchangeName = **"test\_fanout"**;  
 //5. 创建交换机  
 channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***FANOUT***,**true**,**false**,**false**,**null**);  
 //6. 创建队列  
 String queue1Name = **"test\_fanout\_queue1"**;//队列1名称  
 String queue2Name = **"test\_fanout\_queue2"**;//队列2名称  
 channel.queueDeclare(queue1Name, **true**, **false**, **false**, **null**);  
 channel.queueDeclare(queue2Name, **true**, **false**, **false**, **null**);  
 /\*  
 queueBind(String queue, String exchange, String routingKey)  
 参数:  
 1.queue:队列名称  
 2.exchange:交换机名称  
 3.routingKey:路由键，路由规则  
 如果交换机的类型为fanout，则将routingKey设置为""，表示将消息转发给与之绑定的每一个队列  
 \*/  
 //7. 绑定队列和交换机  
 channel.queueBind(queue1Name, exchangeName, **""**);  
 channel.queueBind(queue2Name, exchangeName, **""**);  
 //定义要发送的数据  
 String body = **"日志信息：张三调用了findAll方法...日志级别：info..."**;  
 //8. 发送消息  
 channel.basicPublish(exchangeName,**""**,**null**,body.getBytes());  
 //9. 释放资源  
 channel.close();  
 connection.close();  
 }  
}

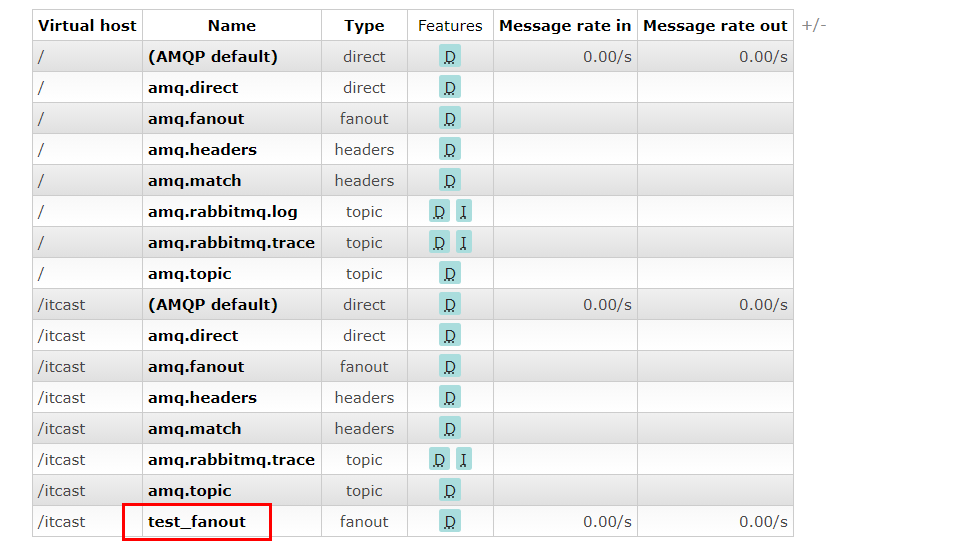
#### 2.2测试

执行生产者的main方法后：



多了2条队列，每条队列各有一条消息。

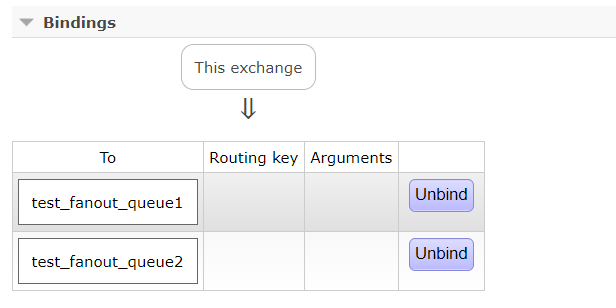
然后看交换机Exchange：



多了一个名为test\_fanout的交换机，类型是fanout。

名称以amq开头的是系统默认的交换机。

我们点击“test\_fannout”：



可以看到，该交换机与两个队列绑定了。

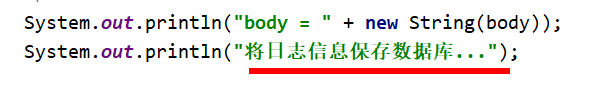
### 3、消费者

#### 3.1消费者1

*/\*\*  
 \* mq消费者1\_订阅模式  
 \*/***public class** Consumer\_PubSub1 {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel，注意消费者与生产者的Channel可不是同一个，看RabbitMQ的架构图就知道  
 Channel channel = connection.createChannel();  
  
 //上面部分的代码，与生产者完全一致  
  
 String queue1Name = **"test\_fanout\_queue1"**;//队列1名称  
 String queue2Name = **"test\_fanout\_queue2"**;//队列2名称  
  
 //5. 创建Consumer对象  
 Consumer consumer = **new** DefaultConsumer(channel) {  
 /\*  
 回调方法，当收到消息后，会自动执行该方法  
 1.consumerTag:标识  
 2.envelope:获取一些信息，交换机，路由key...  
 3.properties:配置信息  
 4.body:数据  
 \*/  
 @Override  
 **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  
 /\*  
 System.out.println("consumerTag = " + consumerTag);  
 System.out.println("Exchange = " + envelope.getExchange());  
 System.out.println("RoutingKey = " + envelope.getRoutingKey());  
 System.out.println("properties = " + properties);  
 \*/  
 System.***out***.println(**"body = "** + **new** String(body));  
 System.***out***.println(**"将日志信息打印到控制台..."**);  
 }  
 };  
 /\*  
 basicConsume(String queue, boolean autoAck, Consumer callback)  
 参数:  
 1.queue:队列名称 hello\_world  
 2.autoAck:是否自动确认（确认消息是否收到）  
 3.callback:回调对象  
 \*/  
 //6. 接收消息  
 channel.basicConsume(queue1Name,**true**,consumer);//接收队列1的消息  
 //7.消费者不要关闭资源，否则无法监听  
 }  
}

#### 3.2消费者2

与消费者1的区别在于：



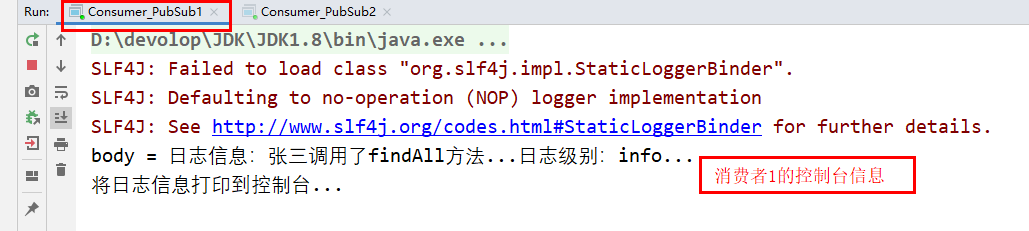
此外，最终要的区别是，消费者2接收队列2的消息：



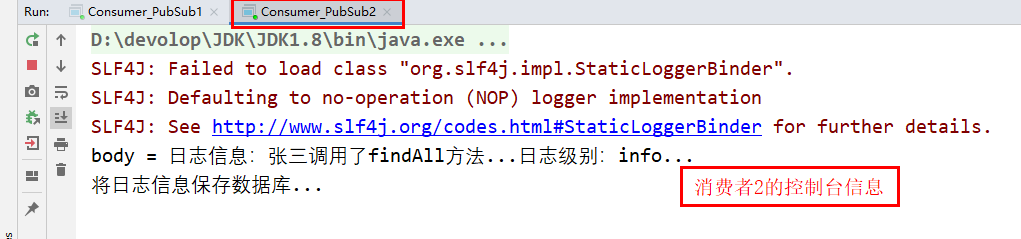
#### 3.3测试

分别启动消费者1和2

消费者1：



消费者2：



可以看到，生产者发送的同一个数据body，经交换机转发后，被两个消费者都接收到了。

### 4、小结

1. 交换机需要与队列进行绑定，绑定之后；一个消息可以被多个消费者都收到。

2. 发布订阅模式与工作队列模式的区别：

* 工作队列模式不用定义交换机，而发布/订阅模式需要定义交换机
* 发布/订阅模式的生产方是面向交换机发送消息，工作队列模式的生产方是面向队列发送消息(底层使用默认交换机)
* 发布/订阅模式需要设置队列和交换机的绑定，工作队列模式不需要设置，实际上工作队列模式会将队列绑定到默认的交换机

Tips：

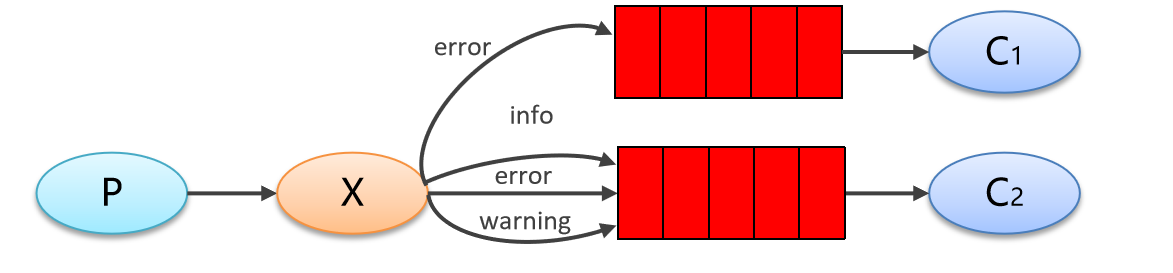
此外，当交换机类型为fanout(广播)时，也决定了路由的routingKey为空字符串。

## 三、路由模式

### 1、模式说明

* 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，该队列才会接收到消息。

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

### 2、生产者

注意：

交换机类型direct、生产者向exchange发消息时，需要执行exchangeName和一个routingKey、exchange与queue绑定时，也需要指定routingKey。只有二者的routingKey完全匹配，exchange才能将消息传给绑定的queue。

#### 2.1代码

*/\*\*  
 \* mq生产者\_路由模式  
 \*/***public class** Producer\_Routing {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel  
 Channel channel = connection.createChannel();  
 /\*  
 exchangeDeclare(String exchange, BuiltinExchangeType type, boolean durable  
 , boolean autoDelete, boolean internal, Map<String, Object> arguments)  
 参数：  
 1.exchange:交换机名称  
 2.type:交换机类型  
 (1)DIRECT("direct"), :定向  
 (2)FANOUT("fanout"), :扇形（广播），发送消息到每一个与之绑定的队列  
 (3)TOPIC("topic"), :通配符方式  
 (4)HEADERS("headers"); :参数匹配（用的很少，不做介绍）  
 3.durable:是否持久化  
 4.autoDelete:是否自动删除  
 5.internal:内部使用。一般false  
 6.arguments：参数  
 \*/  
 String exchangeName = **"test\_direct"**;  
 //5. 创建交换机  
 channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***DIRECT***,**true**,**false**,**false**,**null**);  
 //6. 创建队列  
 String queue1Name = **"test\_direct\_queue1"**;//队列1名称  
 String queue2Name = **"test\_direct\_queue2"**;//队列2名称  
 channel.queueDeclare(queue1Name, **true**, **false**, **false**, **null**);  
 channel.queueDeclare(queue2Name, **true**, **false**, **false**, **null**);  
 /\*  
 queueBind(String queue, String exchange, String routingKey)  
 参数:  
 1.queue:队列名称  
 2.exchange:交换机名称  
 3.routingKey:路由键，路由规则  
 如果交换机的类型为fanout，则将routingKey设置为""，表示将消息转发给与之绑定的每一个队列  
 \*/  
 //7. 绑定队列和交换机  
 /\*队列1绑定 error\*/  
 channel.queueBind(queue1Name, exchangeName, **"error"**);  
 /\*队列2绑定 info error warning\*/  
 channel.queueBind(queue2Name, exchangeName, **"info"**);  
 channel.queueBind(queue2Name, exchangeName, **"error"**);  
 channel.queueBind(queue2Name, exchangeName, **"warning"**);  
 //定义要发送的数据  
 String body = **"日志信息：张三调用了findAll方法...日志级别：info..."**;  
 //8. 发送消息  
 channel.basicPublish(exchangeName,**"info"**,**null**,body.getBytes());  
 //9. 释放资源  
 channel.close();  
 connection.close();  
 }  
}

#### 2.2测试

启动生产者的main方法，生产者只发送了一条routingKey为info的消息，那么应该只有一个队列能够接收到消息：



可以看到，只有test\_direct\_queue2即队列2里面接收到了消息

交换机test\_direct与队列的绑定情况：



### 3、消费者

#### 3.1消费者1

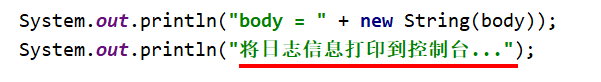
让消费者1接收队列1的消息

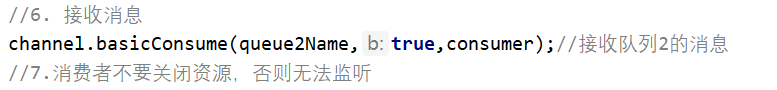
*/\*\*  
 \* mq消费者1\_路由模式  
 \*/***public class** Consumer\_Routing1 {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel，注意消费者与生产者的Channel可不是同一个，看RabbitMQ的架构图就知道  
 Channel channel = connection.createChannel();  
  
 //上面部分的代码，与生产者完全一致  
  
 String queue1Name = **"test\_direct\_queue1"**;//队列1名称  
 String queue2Name = **"test\_direct\_queue2"**;//队列2名称  
  
 //5. 创建Consumer对象  
 Consumer consumer = **new** DefaultConsumer(channel) {  
 /\*  
 回调方法，当收到消息后，会自动执行该方法  
 1.consumerTag:标识  
 2.envelope:获取一些信息，交换机，路由key...  
 3.properties:配置信息  
 4.body:数据  
 \*/  
 @Override  
 **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  
 /\*  
 System.out.println("consumerTag = " + consumerTag);  
 System.out.println("Exchange = " + envelope.getExchange());  
 System.out.println("RoutingKey = " + envelope.getRoutingKey());  
 System.out.println("properties = " + properties);  
 \*/  
 System.***out***.println(**"body = "** + **new** String(body));  
 System.***out***.println(**"将日志信息保存到数据库..."**);  
 }  
 };  
 /\*  
 basicConsume(String queue, boolean autoAck, Consumer callback)  
 参数:  
 1.queue:队列名称 hello\_world  
 2.autoAck:是否自动确认（确认消息是否收到）  
 3.callback:回调对象  
 \*/  
 //6. 接收消息  
 channel.basicConsume(queue1Name,**true**,consumer);//接收队列1的消息  
 //7.消费者不要关闭资源，否则无法监听  
 }  
}

#### 3.2消费者2

让消费者2接收队列2的消息

只展示与消费者1代码不同的地方，其余地方完全一致：





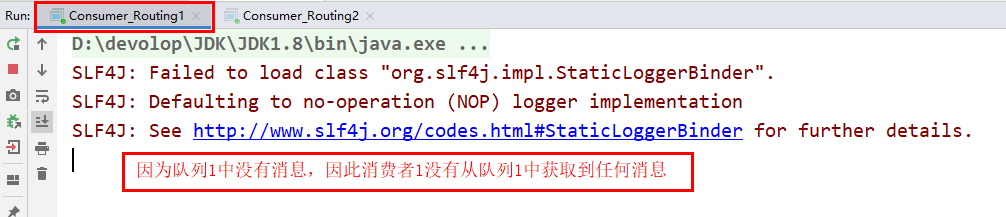
### 4、测试

#### 4.1测试1

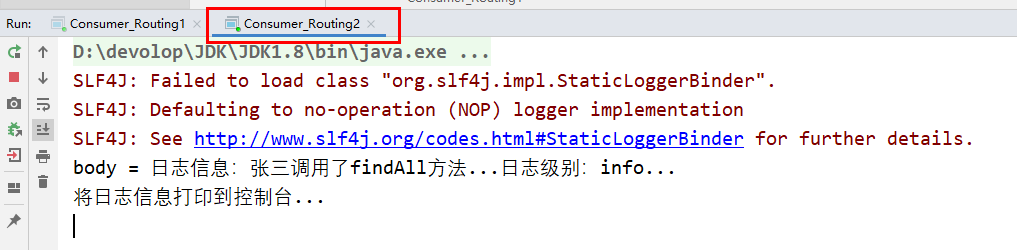
当发送的消息时的routingKey为info时

分别启动消费者1和2，看消息接收的情况

消费者1：没有从队列1接收到消息



消费者2：从队列2接收到了消息

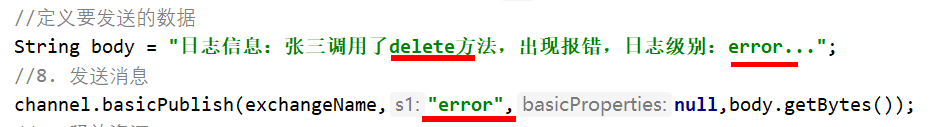


#### 4.2测试2

当发送的消息时的routingKey为error时

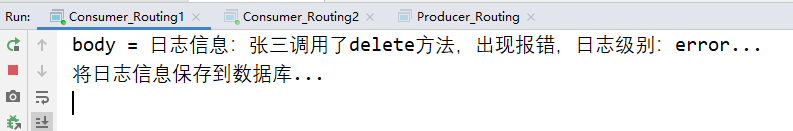
另外我们稍微修改一下生产者的代码，修改内容如下：

生产者：



然后重新运行生产者，发布routingKey为error的消息，查看消费者的情况：

消费者1：



消费者2：



可以看到，两个消费者都接收到了消息。

## 四、通配符模式

### 1、模式说明

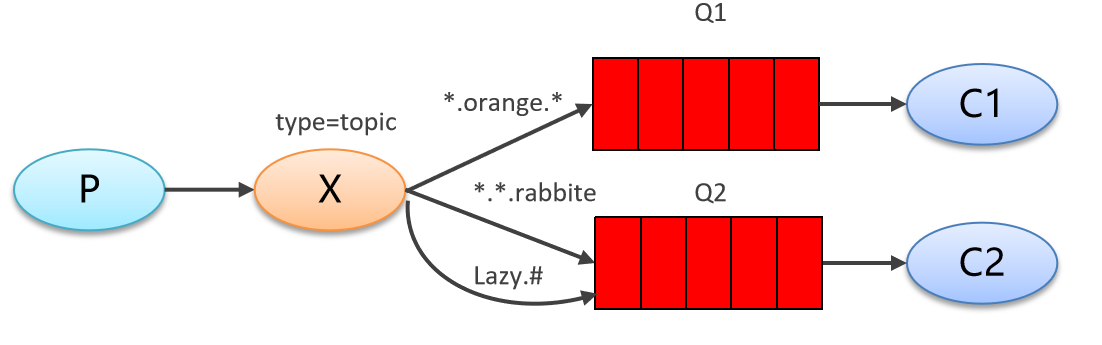
* Topic 类型与 Direct 相比，都是可以根据 RoutingKey 把消息路由到不同的队列。只不过 Topic 类型Exchange 可以让队列在绑定 Routing key 的时候使用通配符！
* Routingkey 一般都是有一个或多个单词组成，多个单词之间以”.”分割，例如： item.insert
* 通配符规则：# 匹配一个或多个词，\* 匹配不多不少恰好1个词。

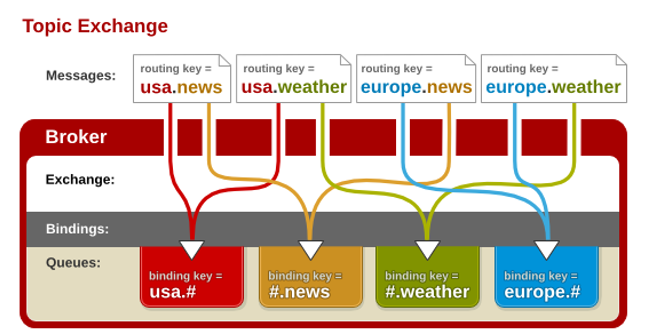
例如：

·item.# 能够匹配 item.insert.abc 或者 item.insert，

·item.\* 只能匹配 item.insert

图解：





* 红色 Queue：绑定的是 usa.# ，因此凡是以 usa. 开头的 routing key 都会被匹配到
* 黄色 Queue：绑定的是 #.news ，因此凡是以 .news 结尾的 routing key 都会被匹配

### 2、生产者

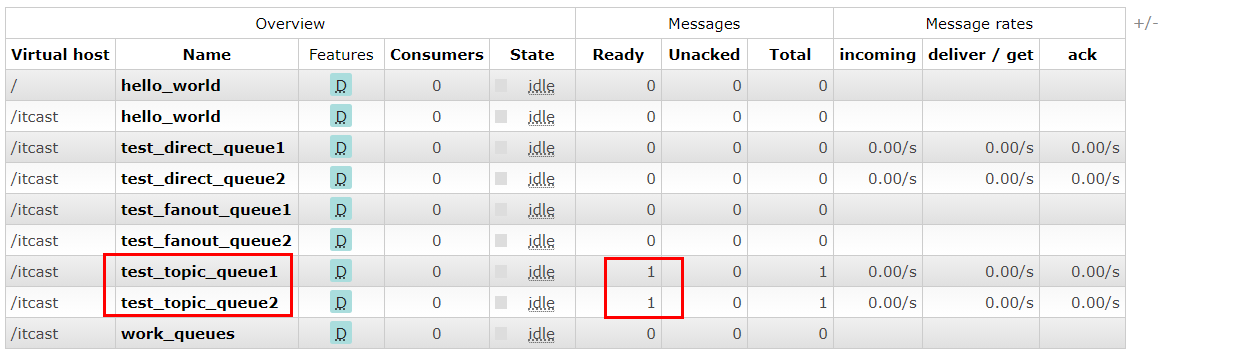
#### 2.1代码

*/\*\*  
 \* mq生产者\_通配符模式  
 \*/***public class** Producer\_Topics {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel  
 Channel channel = connection.createChannel();  
 /\*  
 exchangeDeclare(String exchange, BuiltinExchangeType type, boolean durable  
 , boolean autoDelete, boolean internal, Map<String, Object> arguments)  
 参数：  
 1.exchange:交换机名称  
 2.type:交换机类型  
 (1)DIRECT("direct"), :定向  
 (2)FANOUT("fanout"), :扇形（广播），发送消息到每一个与之绑定的队列  
 (3)TOPIC("topic"), :通配符方式  
 (4)HEADERS("headers"); :参数匹配（用的很少，不做介绍）  
 3.durable:是否持久化  
 4.autoDelete:是否自动删除  
 5.internal:内部使用。一般false  
 6.arguments：参数  
 \*/  
 String exchangeName = **"test\_topic"**;  
 //5. 创建交换机  
 channel.exchangeDeclare(exchangeName, BuiltinExchangeType.***TOPIC***,**true**,**false**,**false**,**null**);  
 //6. 创建队列  
 String queue1Name = **"test\_topic\_queue1"**;//队列1名称  
 String queue2Name = **"test\_topic\_queue2"**;//队列2名称  
 channel.queueDeclare(queue1Name, **true**, **false**, **false**, **null**);  
 channel.queueDeclare(queue2Name, **true**, **false**, **false**, **null**);  
 /\*  
 queueBind(String queue, String exchange, String routingKey)  
 参数:  
 1.queue:队列名称  
 2.exchange:交换机名称  
 3.routingKey:路由键，路由规则  
 如果交换机的类型为fanout，则将routingKey设置为""，表示将消息转发给与之绑定的每一个队列  
 \*/  
 //7. 绑定队列和交换机  
 //routingKey 系统名称.日志级别  
 //需求:所有error级别的日志，存入数据库；所有order系统的日志存入数据库  
 channel.queueBind(queue1Name, exchangeName, **"#.error"**);//队列1  
 channel.queueBind(queue1Name, exchangeName, **"order.\*"**);//队列1  
 channel.queueBind(queue2Name, exchangeName, **"\*.\*"**);//队列2  
 //定义要发送的数据  
 String body = **"日志信息：张三调用了findAll方法...日志级别：info..."**;  
 //8. 发送消息  
 channel.basicPublish(exchangeName,**"order.info"**,**null**,body.getBytes());  
 //9. 释放资源  
 channel.close();  
 connection.close();  
 }  
}

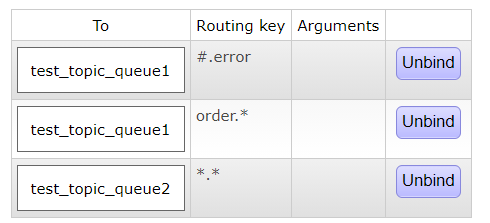
#### 2.2测试

启动生产者，发送消息，消息的routingKey为“order.info”，应该两个队列都能接收到消息

消息接收情况：



队列与交换机绑定情况：



### 3、消费者

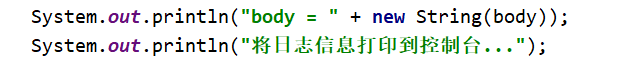
#### 3.1消费者1

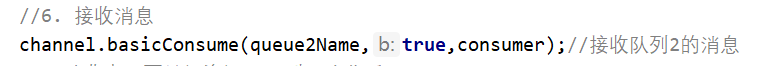
代码：

*/\*\*  
 \* mq消费者1\_通配符模式  
 \*/***public class** Consumer\_Topic1 {  
 **public static void** main(String[] args) **throws** Exception {  
 //1. 创建连接工厂  
 ConnectionFactory factory = **new** ConnectionFactory();  
 //2. 设置参数  
 factory.setHost(**"192.168.176.129"**);//虚拟机的IP，默认值 localhost  
 factory.setPort(5672);//端口，默认值 5672  
 factory.setVirtualHost(**"/itcast"**);//虚拟机名称 默认值是“/”  
 factory.setUsername(**"heima"**);//用户名 默认值 guest  
 factory.setPassword(**"heima"**);//密码 默认值 guest  
 //3. 创建连接 Connection  
 Connection connection = factory.newConnection();  
 //4. 创建Channel，注意消费者与生产者的Channel可不是同一个，看RabbitMQ的架构图就知道  
 Channel channel = connection.createChannel();  
  
 //上面部分的代码，与生产者完全一致  
  
 String queue1Name = **"test\_topic\_queue1"**;//队列1名称  
 String queue2Name = **"test\_topic\_queue2"**;//队列2名称  
  
 //5. 创建Consumer对象  
 Consumer consumer = **new** DefaultConsumer(channel) {  
 /\*  
 回调方法，当收到消息后，会自动执行该方法  
 1.consumerTag:标识  
 2.envelope:获取一些信息，交换机，路由key...  
 3.properties:配置信息  
 4.body:数据  
 \*/  
 @Override  
 **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  
 /\*  
 System.out.println("consumerTag = " + consumerTag);  
 System.out.println("Exchange = " + envelope.getExchange());  
 System.out.println("RoutingKey = " + envelope.getRoutingKey());  
 System.out.println("properties = " + properties);  
 \*/  
 System.***out***.println(**"body = "** + **new** String(body));  
 System.***out***.println(**"将日志信息保存到数据库..."**);  
 }  
 };  
 /\*  
 basicConsume(String queue, boolean autoAck, Consumer callback)  
 参数:  
 1.queue:队列名称 hello\_world  
 2.autoAck:是否自动确认（确认消息是否收到）  
 3.callback:回调对象  
 \*/  
 //6. 接收消息  
 channel.basicConsume(queue1Name,**true**,consumer);//接收队列1的消息  
 //7.消费者不要关闭资源，否则无法监听  
 }  
}

#### 3.2消费者2

只展示与消费者1代码不同的地方





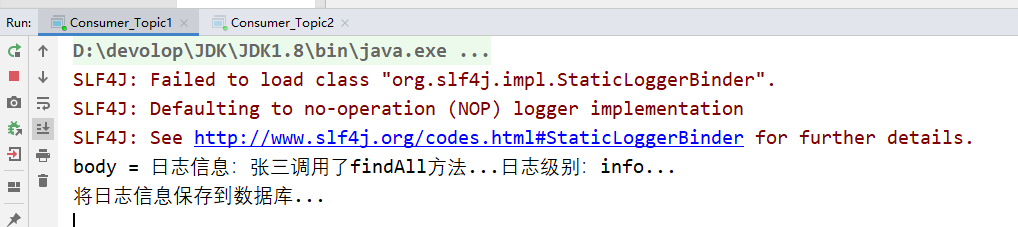
#### 3.3测试

分别启动消费者1和2，查看消息接收情况

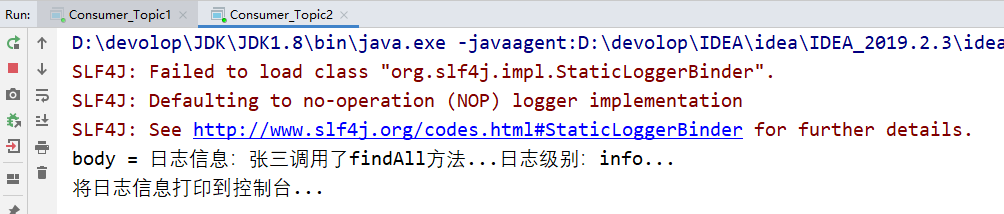
测试1：

生产者发送消息时的routingKey为：order.info

消费者1：



消费者2：



可以看到，两个消费者都接收到了消息。

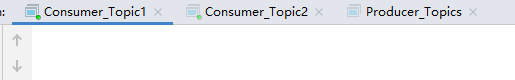
测试2：

生产者发送消息时的routingKey改为：goods.info

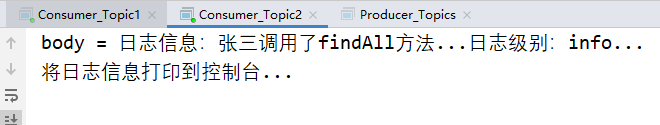


那么应该就只有队列2能够接收到消息了，再来查看消费者的接收消息情况：

消费者1：



消费者2：



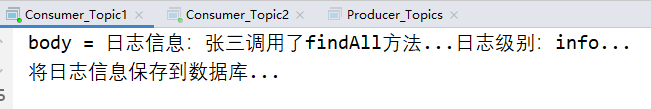
测试3：

生产者发送消息时的routingKey改为：goods.info.error

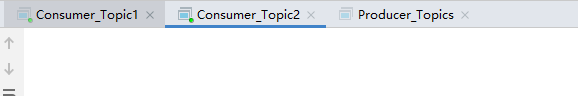
那么应该就只有消费者1能够收到消息了



消费者1：



消费者2：



测试成功！

小结：

Topic 主题模式可以实现 Pub/Sub 发布与订阅模式和 Routing 路由模式的功能，只是 Topic 在配置routing key 的时候可以使用通配符，显得更加灵活。

## 五、工作模式总结

1、简单模式 HelloWorld

一个生产者、一个消费者，不需要设置交换机（**使用默认的交换机**）。

消息与消费者之间的关系：一对一

2、工作队列模式 Work Queue

一个生产者、多个消费者（竞争关系），不需要设置交换机（**使用默认的交换机**）。

多个消费者与同一个消息之间的关系：竞争

3、发布订阅模式 Publish/subscribe

需要设置类型为 fanout 的交换机，并且交换机和队列进行绑定，当发送消息到交换机后，交换机会将消息发送到绑定的队列。

多个消费者与同一个消息之间的关系：共享，routing key为空字符串” ”

4、路由模式 Routing

需要设置类型为 direct 的交换机，交换机和队列进行绑定，并且指定 routing key，当发送消息到交换机后，交换机会根据 routing key 将消息发送到对应的队列。

关系：指定routing key，将消息定向发送给消费者

5、通配符模式 Topic

需要设置类型为 topic 的交换机，交换机和队列进行绑定，并且指定通配符方式的 routing key，当发送消息到交换机后，交换机会根据 routing key 将消息发送到对应的队列。

关系：指定routing key规则。定向发送消息。

一些细节：

1、两个routing key创建的时机：

(1) 生产者发消息的时候，必须指定一个routing key

(2) 交换机绑定队列的时候，指定一个routing key

2、创建队列时：

需要指定队列名称，其他参数。不需要指定routing key

3、创建交换机时：

需要指定交换机的名称、交换机的类型、其他参数。不需要指定routing key

4、交换机绑定队列时：

需要指定交换机名称、队列名称、routing key

5、生产者发消息时：

* 需要指定交换机名称、routing key、消息数据、其他参数。
* 如果交换机的名字写错：那么消息就到不了交换机
* 如果交换机名字写对，但是routing key与交换机绑定队列的routing key不一致，则消息只能发送到交换机，而交换机无法把消息转发给绑定的队列

6、简单模式和工作模式中

没有使用交换机，（或者说用的是默认交换机“”），发送消息时，routing key就直接与队列名称对应。

7、消费者中：

(1) 创建consumer对象：可以接收消息，编写要做的事

(2) 接收消息时，需要的参数：队列名称、consumer对象、其他参数。

注意接收消息时，是不需要routing key的。

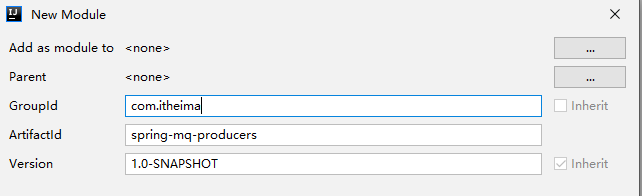
# 第五章 Spring整合RabbitMQ

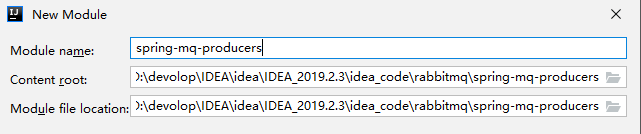
## 一、环境搭建

### 1、创建工程

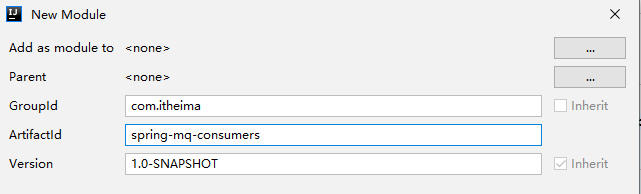
在之前的空项目rabbiitmq下再创建一个模块，maven的SE工程，取名为

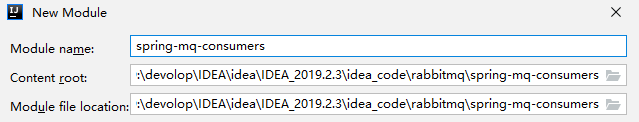
spring-mq-producers，作为生产者模块





然后再创建一个模块，maven的SE工程，取名为spring-mq-consumers，作为消费者模块





### 2、pom

资料里面已经准备好了：



producers和consumers这两个模块要导入的pom一样：

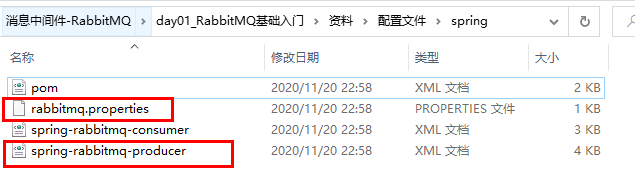
代码：

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>5.1.7.RELEASE</**version**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.amqp</**groupId**>  
 <**artifactId**>spring-rabbit</**artifactId**>  
 <**version**>2.1.8.RELEASE</**version**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**version**>4.12</**version**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-test</**artifactId**>  
 <**version**>5.1.7.RELEASE</**version**>  
 </**dependency**>  
</**dependencies**>  
  
<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>3.8.0</**version**>  
 <**configuration**>  
 <**source**>1.8</**source**>  
 <**target**>1.8</**target**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

## 二、生产者

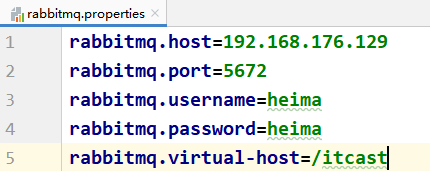
### 1、配置文件

将下面两个文件复制到producers模块的resources目录下：



#### 1.1配置文件1

rabbitmq.properties：

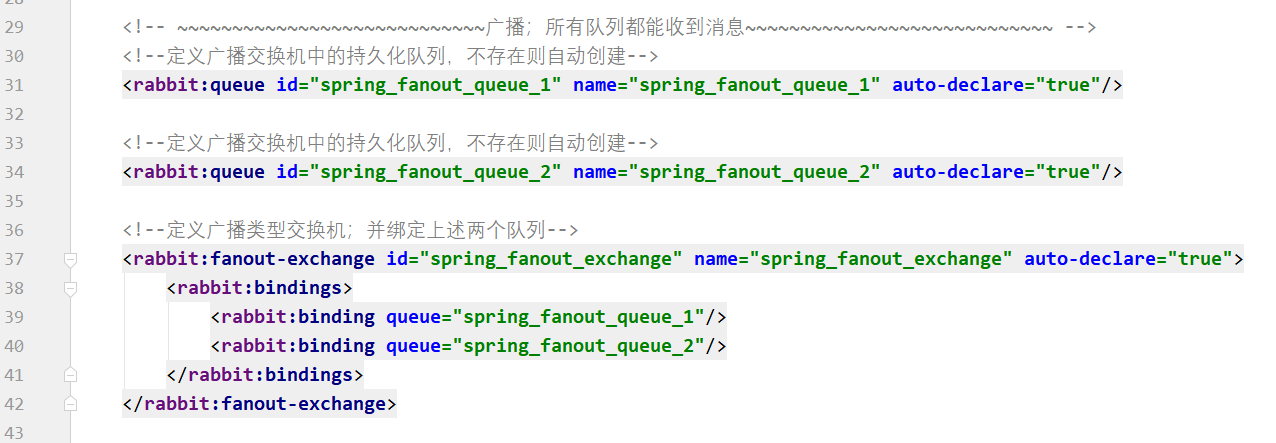


#### 1.2配置文件2

spring-rabbitmq-producer：

图片：







代码：

### 2、简单模式

其实工作队列模式也可以用这个，因为工作队列模式也是没有使用交换机的。

配置文件中已经定义好了很多的队列，交换机，以及队列与交换机的绑定关系，绑定的routing key。

在test目录中创建包：com.itheima，在itheima包下创建一个测试类，比如取名为：ProducerTest

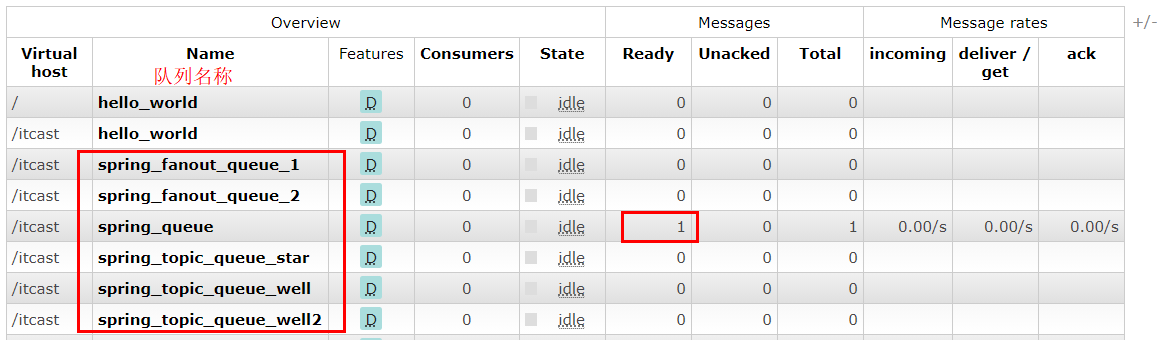
代码：

*/\*\*  
 \* RabbitMQ生产者\_测试类  
 \*/*@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(locations = **"classpath:spring-rabbitmq-producer.xml"**)  
**public class** ProducerTest {  
 //注入RabbitTemplate  
 @Autowired  
 **private** RabbitTemplate **rabbitTemplate**;  
  
 //简单模式  
 @Test  
 **public void** testHelloWorld() {  
 //1.定义消息  
 String body = **"hello world spring..."**;  
 //2.发送消息  
 **rabbitTemplate**.convertAndSend(**"spring\_queue"**,body);  
 }  
}

测试：

启动testHelloWorld方法后：

队列：

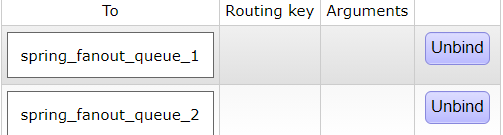


可以看到，方法一启动，配置文件中配置的5个队列就都创建好了，只有名为spring-queue的队列收到了消息。

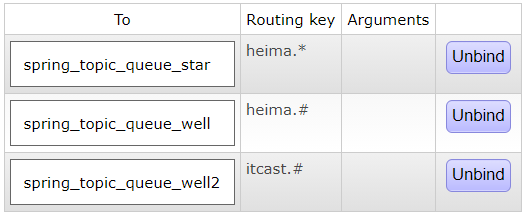
因为简单模式中，没有交换机，生产者直接将消息发送给队列，那么如何知道发送给哪个队列呢？就根据发送消息时的routing key和队列名称直接对应。

交换机：

spring\_fanout\_exchange交换机



spring\_topic\_exchange交换机



### 3、订阅模式

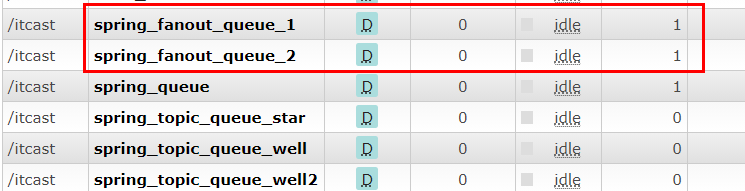
发送广播消息，交换机类型为fanout，routing key为空字符串“”

代码：

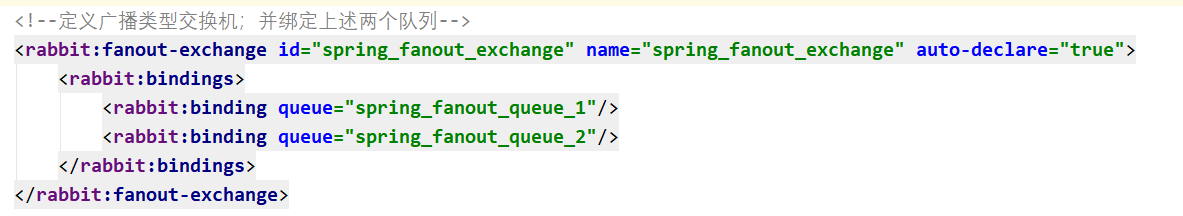
//订阅模式，发送fanout消息  
@Test  
**public void** testFanout() {  
 //1.定义消息  
 String body = **"spring fanout ..."**;  
 //2.发送消息  
 **rabbitTemplate**.convertAndSend(**"spring\_fanout\_exchange"**,**""**,body);  
}

测试：

可以看到，红框中的两个队列都接收到了消息，因为都与这个交换机绑定了，routing key都为空字符串“”，在配置文件中没有指定。



查看配置文件：

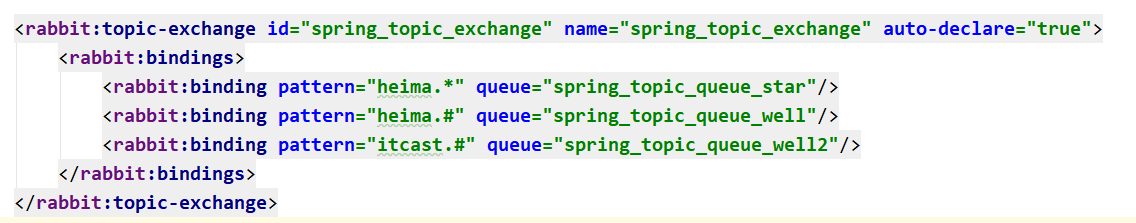


### 4、通配符模式

代码：

//通配符模式  
@Test  
**public void** testTopic() {  
 //1.定义消息  
 String body = **"spring topic ..."**;  
 //2.发送消息  
 **rabbitTemplate**.convertAndSend(**"spring\_topic\_exchange"**,**"heima.hehe.xixi"**,body);  
}

配置文件：



测试：



果然，只有红框中的队列接收到了消息。

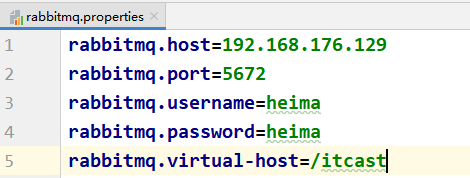
至于路由模式，我们就不展示了，需要在配置文件中定义对应的交换机才行，与通配符模式的使用方法大同小异，只是注意交换机类型，以及routing key的区别而已。

## 三、消费者

**在spring-mq-consumers模块中**

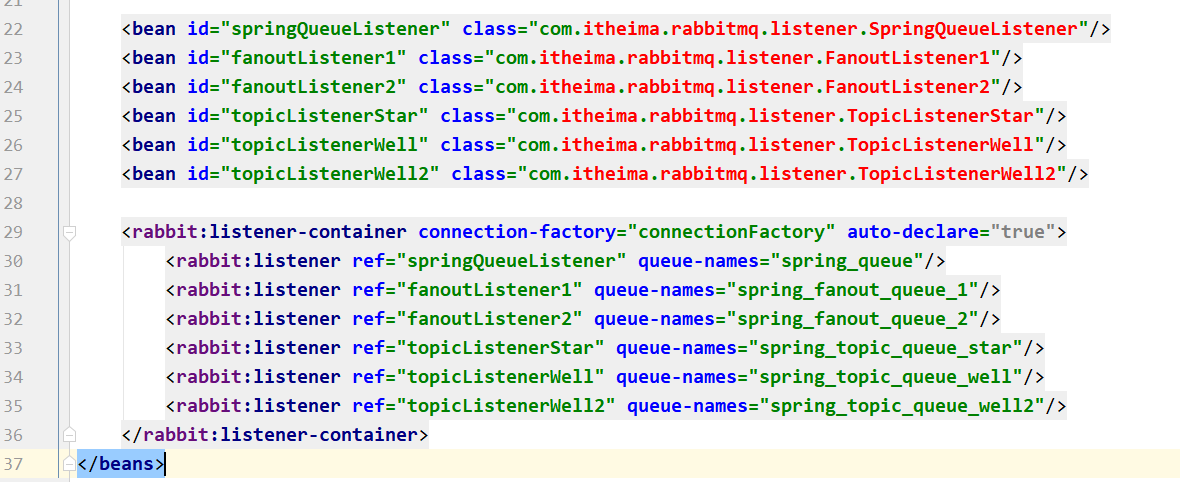
### 1、配置文件

#### 1.1配置文件1



#### 1.2配置文件2



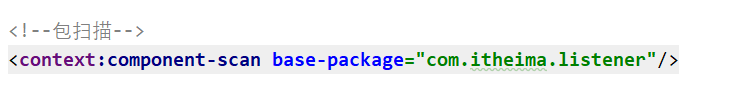


上面定义了很多的<bean>，即监听器，每个监听器监听一个队列。之所以报红，是因为我们还没有创建对应的类。

Tips：

这样定义了很多bean，比较麻烦，我们可以直接用一个报扫描的配置替代，这样以后只要在对应的包下的类上加@Component注解，这些类就能直接被加入Spring IOC中了。

例如：



接下来我们就来创建对应的监听器类。

### 2、监听器1

#### 2.1监听器

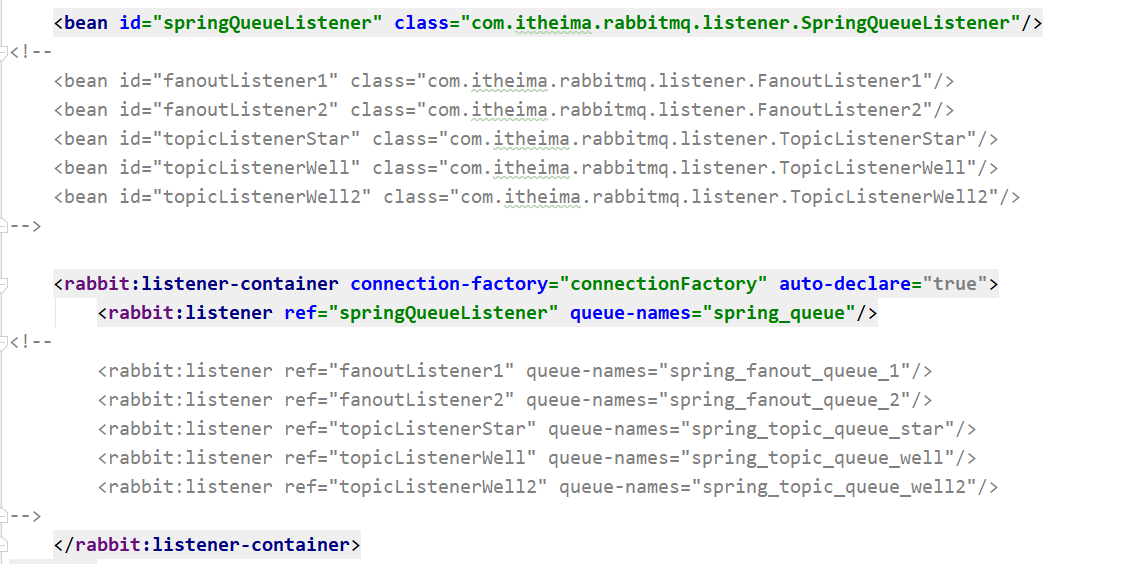
注意监听器的全类名必须与配置文件中的完全一致，否则会报错。我们首先在java目录下创建包：com.itheima.rabbitmq.listener，然后再listener包下创建类：SpringQueueListener。

这个类要想作为监听器，必须实现接口：MessageListener

代码：

**public class** SpringQueueListener **implements** MessageListener {  
 @Override  
 **public void** onMessage(Message message) {  
 String body = **new** String(message.getBody());  
 //打印消息  
 System.***out***.println(**"body = "** + body);  
 }  
}

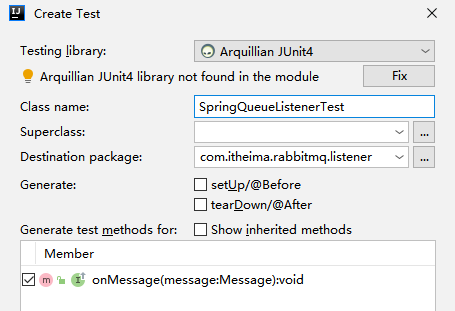
注意，要想不报错，首先必须把其他监听器的配置给注释掉，因为我们目前只写了这一个监听器。



#### 2.2测试类

然后在test目录中创建这个监听器的测试类。

使用IDEA快速创建：选中类名，按alt + enter——Create Test

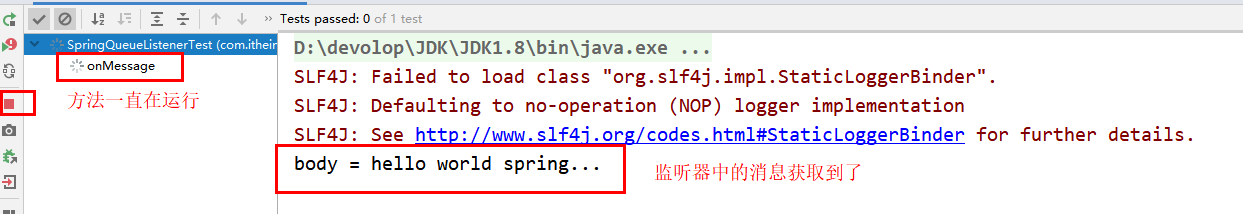


代码：

@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(locations = **"classpath:spring-rabbitmq-consumer.xml"**)  
**public class** SpringQueueListenerTest {  
  
 @Test  
 **public void** onMessage() {  
 **while** (**true**){   
 }  
 }  
}

在onMessage方法中，写了一个死循环，是为了让方法一直运行。

测试：



可以看到，前面定义的监听器中的方法执行了，获取到了消息。

至于其他的监听器，实现方法也是一模一样的，只需实现MessageListener接口，重写onMessage方法，在里面就可以获取到队列中给的消息了。

根据配置文件中，每个监听器都配置了一个队列，就会获取指定的队列中的消息。

其他的就不再演示了。

## 四、配置详解

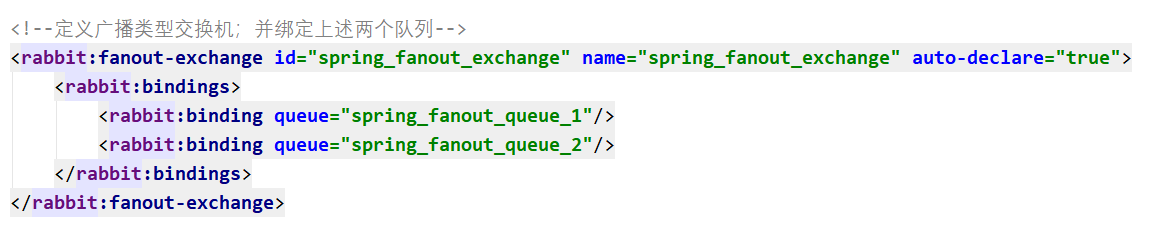
针对生产者和消费者中给的配置文件2，进行详解。

### 1、定义队列：

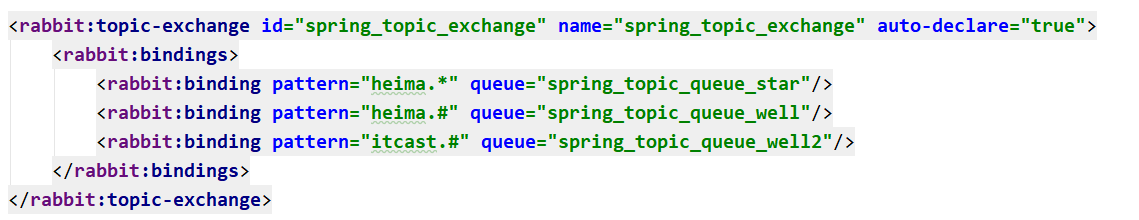
<!--  
定义持久化队列，不存在则自动创建；不绑定到交换机则绑定到默认交换机  
默认交换机类型为direct，名字为：""，路由键为队列的名称  
-->  
<!--  
 id: bean的名称  
 name: queue的名称  
 auto-delcare:自动创建  
 auto-delete: 自动删除。最后一个消费者和队列断开连接后，自动删除队列  
 exclusive:是否独占  
 \* 是否独占。只能有一个消费者监听这个队列(独占)  
 \* 当Connection关闭时，是否删除队列  
 durable: 是否持久化  
-->  
<**rabbit:queue id="spring\_queue" name="spring\_queue" auto-declare="true"**/>

### 2、定义交换机

2.1广播类型交换机（fanout）



2.2通配符类型交换机，需要指定路由key



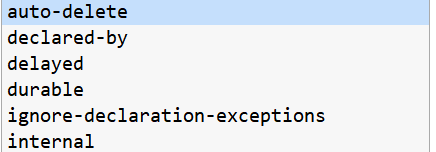
属性：

rabbit：fanout —— 交换机的类型

id：bean的id

name：交换机的名称

其他参数：



其他交换机类型：



2.3路由类型交换机（direct）



# 第六章 SpringBoot整合RabbitMQ

## 一、思路

生产者：

1. 创建生产者SpringBoot工程

2. 引入start，依赖坐标

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

3. 编写yml配置，基本信息配置

4. 定义交换机，队列以及绑定关系的配置类

5. 注入RabbitTemplate，调用方法，完成消息发送

消费者

1. 创建消费者SpringBoot工程

2. 引入start，依赖坐标

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

3. 编写yml配置，基本信息配置

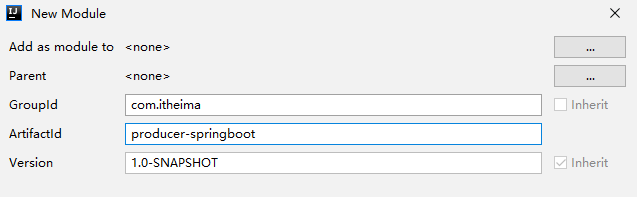
4. 定义监听类，使用@RabbitListener注解完成队列监听。

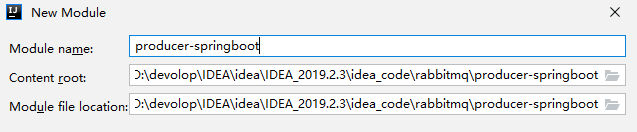
## 二、生产者

### 1、环境搭建

#### 1.1创建工程

依然是在rabbitmq这个空项目下创建模块，maven的SE工程，比如取名为：producer-springboot

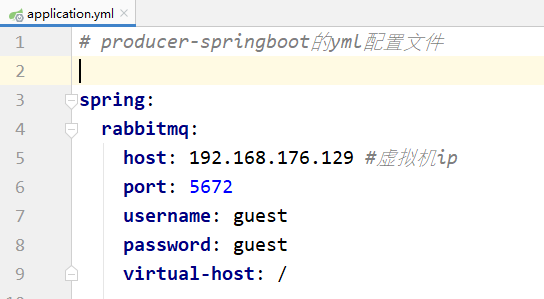




#### 1.2 pom

<!--spring-boot父工程依赖-->  
<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.4.RELEASE</**version**>  
</**parent**>  
  
<**dependencies**>  
 <!--rabbitmq启动器-->  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-amqp</**artifactId**>  
 </**dependency**>  
 <!--junit单元测试启动器-->  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 </**dependency**>  
</**dependencies**>

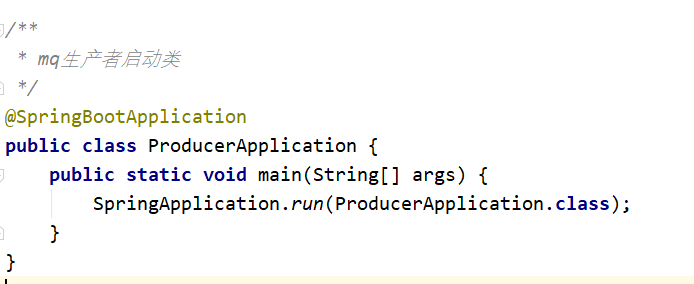
#### 1.3 yml文件



#### 1.4启动类

在com.itheima包下创建启动类

ProducerApplication：



启动类的意义：

虽然我们没有启动这个启动类，但是后面要写测试类。测试类的包必须要能被启动类扫描到。否则无法使用@Runwith等注解。

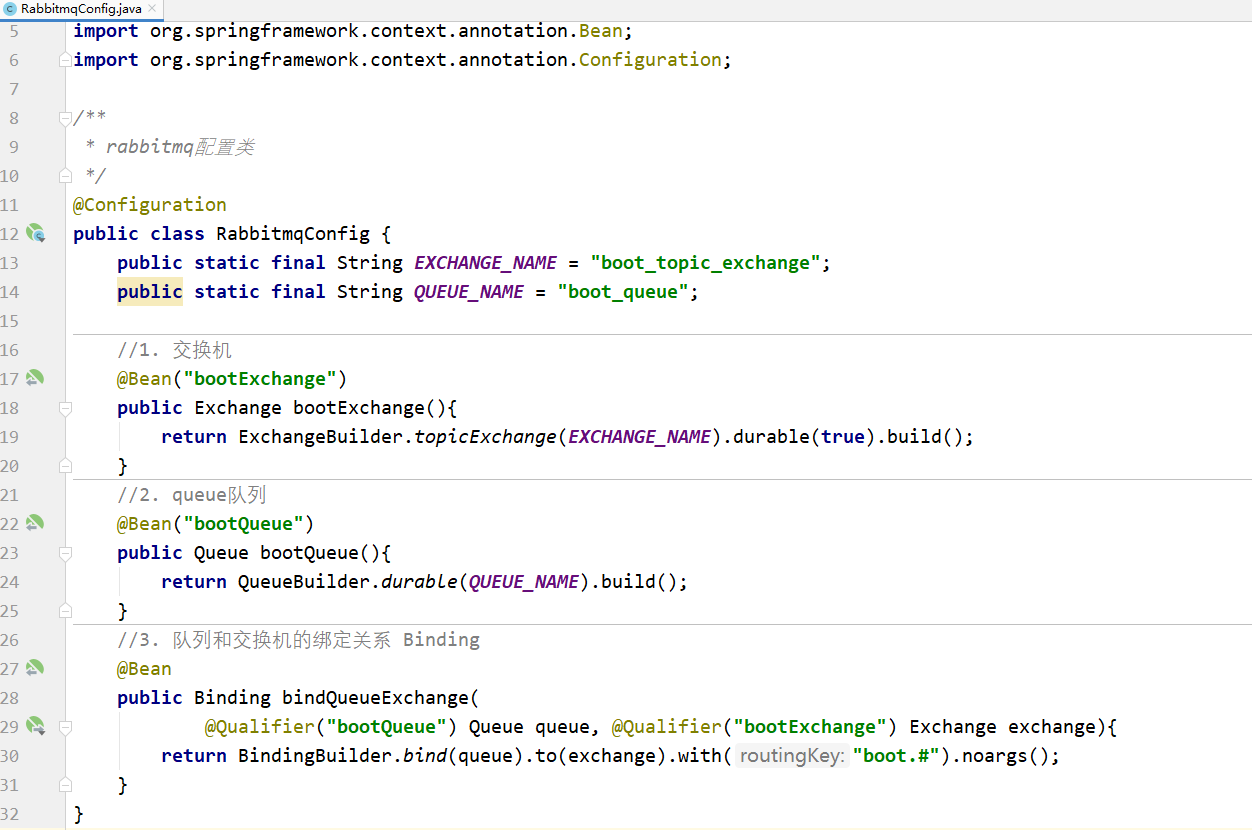
因此启动类虽然我们这里不启动，但是必须要写。

### 2、配置类

在com.itheima.rabbitmq.config包下创建一个配置类：RabbitmqConfig

我们在配置类中定义：交换机、队列、绑定关系等。

RabbitmqConfig：



### 3、测试类

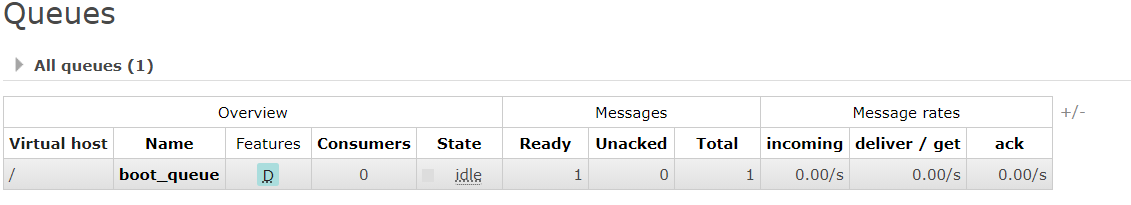
在test目录的com.itheima.test包下，创建测试类ProducerTest

ProducerTest：

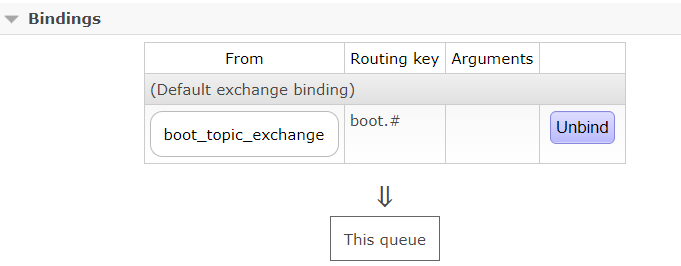
*/\*\*  
 \* 生产者测试类  
 \*/*@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** ProducerTest {  
 @Autowired  
 **private** RabbitTemplate **rabbitTemplate**;  
  
 //发送topic消息  
 @Test  
 **public void** testSend() {  
 String body = **"boot mq hello~~~~"**;  
 **rabbitTemplate**.convertAndSend(  
 RabbitmqConfig.***EXCHANGE\_NAME***,**"boot.haha"**,body);  
 }  
}

启动测试方法：

队列：



绑定关系：



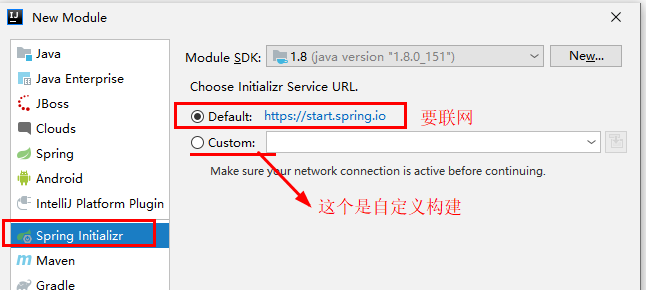
## 三、消费者

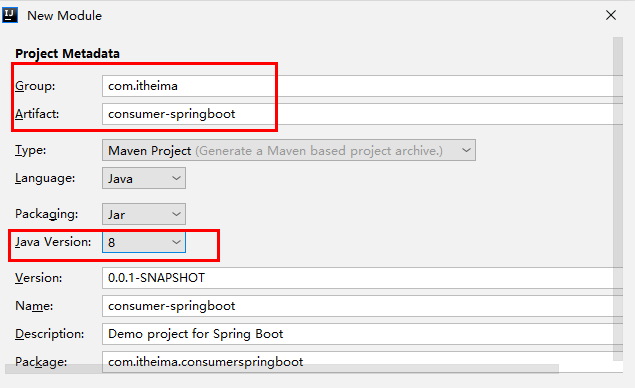
### 1、环境搭建

#### 1.1创建工程

在rabbitmq项目下再创建一个模块，这次我们直接加一个springboot的项目，取名为：consumer-springboot

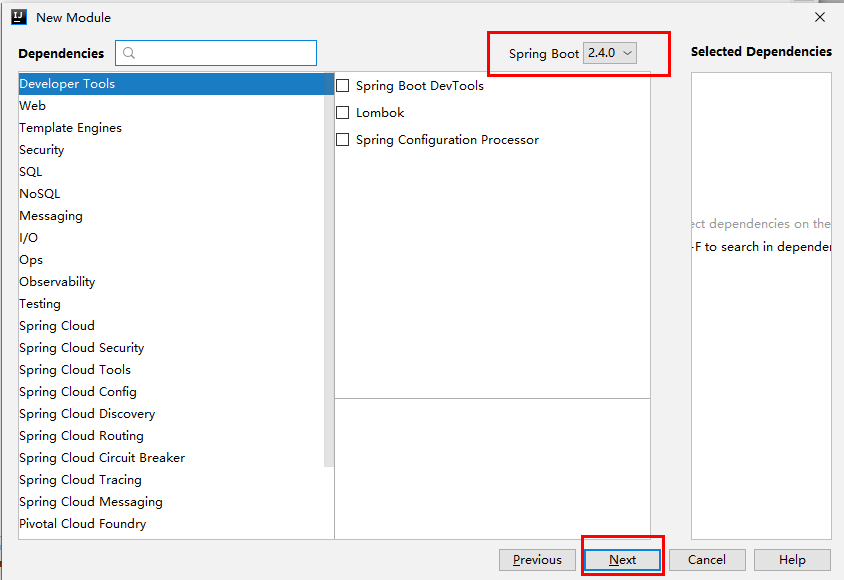
也可以来看一下IDEA帮我们初始化的SpringBoot项目是怎样的。

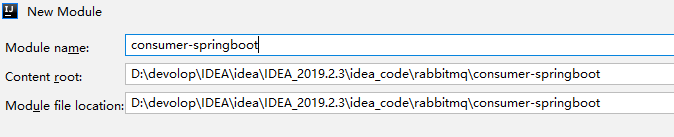




添加依赖

没有找到amqp的依赖，我们直接下一步





之后需要修改SpringBoot父工程的版本，为了和producer的版本统一，我们稍后需要修改成2.1.4。

#### 1.2 pom

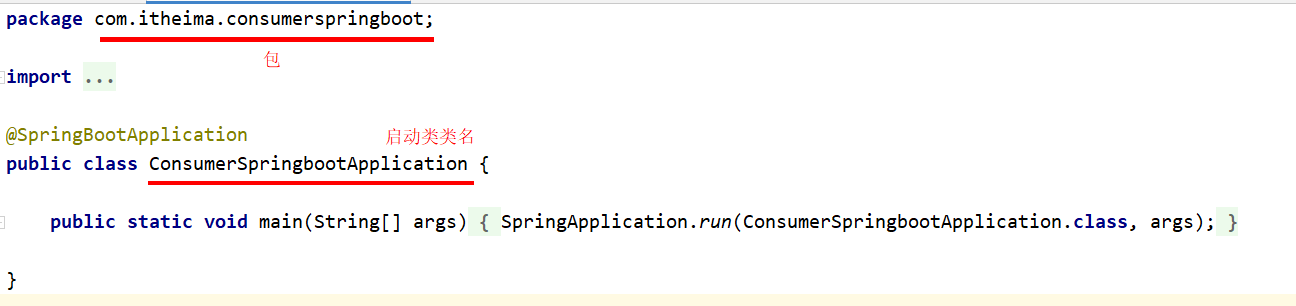
IDEA帮我们生成的pom里面，默认有SpringBoot启动器、单元测试启动器、maven插件、以及SpringBoot的父工程依赖。

我们需要引入rabbitmq依赖，最终重要的内容如下：

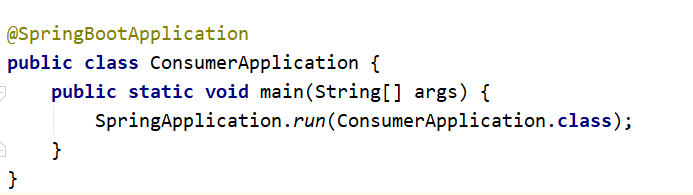
<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.4.RELEASE</**version**>  
 <**relativePath**/> <!-- lookup parent from repository -->  
</**parent**>  
<**groupId**>com.itheima</**groupId**>  
<**artifactId**>consumer-springboot</**artifactId**>  
<**version**>0.0.1-SNAPSHOT</**version**>  
  
<**properties**>  
 <**java.version**>1.8</**java.version**>  
</**properties**>  
  
<**dependencies**>  
 <!--rabbitmq-->  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-amqp</**artifactId**>  
 </**dependency**>  
</**dependencies**>  
  
<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

#### 1.3启动类

IDEA会自动帮我们创建一个启动类



我们可以删除，然后再itheima包下创建启动类ConsumerApplication



注意：

IDEA同时还会帮我们生成一个测试类，包名与初始启动类一致。也可以将其删去

#### 1.4 yml文件

IDEA自动帮我们创建的是application.properties。我们可以改成yml文件

IDEA改名快捷键：shift + F6

消费者的application.yml：

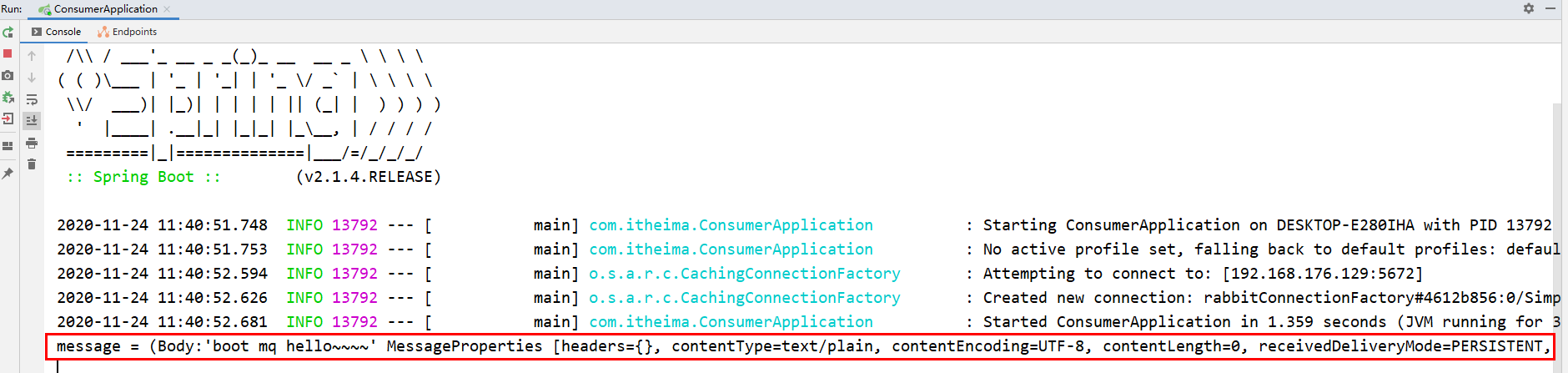
*# consumer-springboot的yml配置文件***spring**:  
 **rabbitmq**:  
 **host**: 192.168.176.129 *#虚拟机ip* **port**: 5672  
 **username**: guest  
 **password**: guest  
 **virtual-host**: /

### 2、监听器

*/\*\*  
 \* 消费者监听器  
 \*/*@Component  
**public class** RabbitmqListener {  
 @RabbitListener(queues = **"boot\_queue"**)  
 **public void** ListenerQueue(Message message){  
 System.***out***.println(**"message = "** + message);  
 }  
}

### 3、测试

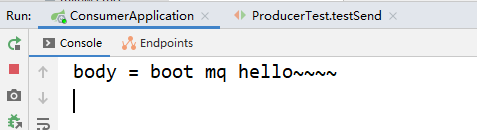
打开启动类，观察监听器中方法是否执行，是否成功打印出message



打印的是整个message，如果指向获取消息数据的话，可以对监听器代码进行修改：

@Component  
**public class** RabbitmqListener {  
 @RabbitListener(queues = **"boot\_queue"**)  
 **public void** ListenerQueue(Message message){  
 //System.out.println("message = " + message);  
 String body = **new** String(message.getBody());  
 System.***out***.println(**"body = "** + body);  
 }  
}

然后重新启动消费者的启动类，目前是没有消息的，因为刚才的消息已经被接受了，我们需要重新启动生产者发送消息。观察消费者的接收情况。



成功接收到消息

## 四、小结

* SpringBoot提供了快速整合RabbitMQ的方式
* 基本信息再yml中配置，队列交互机以及绑定关系在配置类中使用Bean的方式配置
* 生产端直接注入RabbitTemplate完成消息发送
* 消费端直接使用@RabbitListener完成消息接收

# RabbitMQ\_高级特性

# 第一章 RabbitMQ的高级特性

## 一、消息的可靠投递

### 1、概述

在使用 RabbitMQ 的时候，作为消息发送方希望杜绝任何消息丢失或者投递失败场景。RabbitMQ 为我们提供了两种方式用来控制消息的投递可靠性模式。

* confirm 确认模式
* return 退回模式

rabbitmq 整个消息投递的路径为：

producer--->rabbitmq broker--->exchange--->queue--->consumer

* 消息从 producer 到 exchange 则会返回一个 confirmCallback 。
* 消息从 exchange-->queue 投递失败则会返回一个 returnCallback 。

我们将利用这两个 callback 控制消息的可靠性投递

下面我们就使用Spring整合RabbitMQ的方式来学习。具体初始代码参考第五章。

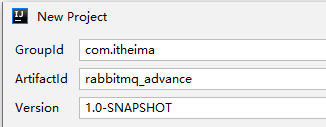
·spring-mq-producers

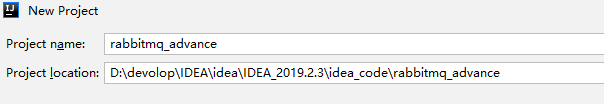
·spring-mq-consumers

这一部分知识，全都是针对生产者的

### 2、创建工程

我们再创建一个maven的工程，取名为：rabbitmq-advance，作为父工程





在父工程的pom中导入插件：

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>3.8.0</**version**>  
 <**configuration**>  
 <**source**>1.8</**source**>  
 <**target**>1.8</**target**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

然后创建子模块

·mq-consumer-spring 消费者模块

·mq-producer-spring 生产者模块

都引入相同的依赖：

都引入相同的配置文件1：

**rabbitmq.host**=**192.168.176.129  
rabbitmq.port**=**5672  
rabbitmq.username**=**guest  
rabbitmq.password**=**guest  
rabbitmq.virtual-host**=**/**

都用的是默认用户guest，和默认的虚拟机“/”。

### 3、确认模式

***下面展示的都是确认模式下的生产者代码***

#### 3.1配置文件

配置文件1生产者与消费者一样，上面已经介绍了，就不再重复叙述，这里介绍配置文件2

与以往不同的是，需要在该配置文件中开启确认模式

代码：

*<?***xml version="1.0" encoding="UTF-8"***?>*<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:rabbit="http://www.springframework.org/schema/rabbit"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/context  
 https://www.springframework.org/schema/context/spring-context.xsd  
 http://www.springframework.org/schema/rabbit  
 http://www.springframework.org/schema/rabbit/spring-rabbit.xsd"**>  
 <!--加载配置文件-->  
 <**context:property-placeholder location="classpath:rabbitmq.properties"**/>  
  
 <!-- 定义rabbitmq connectionFactory -->  
 <**rabbit:connection-factory id="connectionFactory" host="${rabbitmq.host}"  
 port="${rabbitmq.port}"  
 username="${rabbitmq.username}"  
 password="${rabbitmq.password}"  
 virtual-host="${rabbitmq.virtual-host}"  
 publisher-confirms="true"** />  
 <!--定义管理交换机、队列-->  
 <**rabbit:admin connection-factory="connectionFactory"**/>  
  
 <!--定义rabbitTemplate对象操作可以在代码中方便发送消息-->  
 <**rabbit:template id="rabbitTemplate" connection-factory="connectionFactory"**/>  
  
 <!--消息可靠性投递（生产者）-->  
 <!--定义队列-->  
 <**rabbit:queue id="test\_queue\_confirm" name="test\_queue\_confirm"**></**rabbit:queue**>  
 <!--定义交换机，绑定队列-->  
 <**rabbit:direct-exchange name="test\_exchange\_confirm"**>  
 <**rabbit:bindings**>  
 <!--routing key设为“confirm”-->  
 <**rabbit:binding queue="test\_queue\_confirm" key="confirm"**></**rabbit:binding**>  
 </**rabbit:bindings**>  
 </**rabbit:direct-exchange**>  
</**beans**>

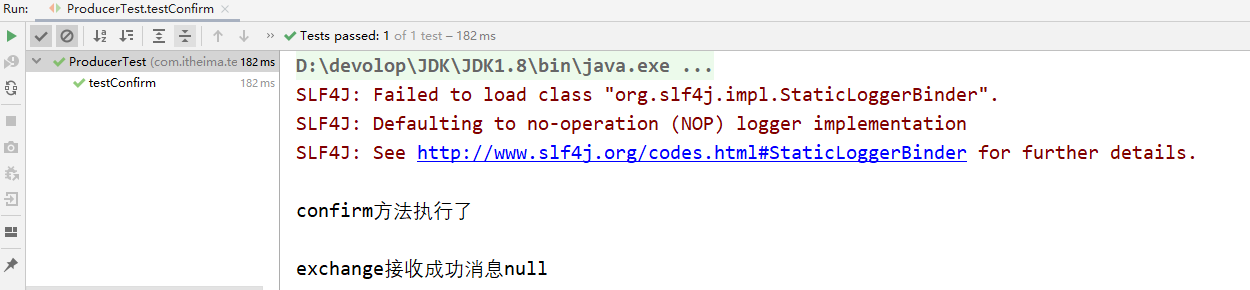
#### 3.2测试类

*/\*\*  
 \* 生产者测试类  
 \*/*@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(locations = **"classpath:mq.producer-spring.xml"**)  
**public class** ProducerTest {  
   
 @Autowired  
 **private** RabbitTemplate **rabbitTemplate**;  
  
 /\*  
 确认模式：  
 步骤：  
 1.确认模式开启：修改配置文件，在<rabbit:connection-factory>标签中，  
 增加一条属性：publisher-confirms="true"  
 2.使用rabbitTemplate定义ConfirmCallBack回调函数  
 \*/  
 @Test  
 **public void** testConfirm() {  
 //可以使用Lambda表达式代替  
 //定义回调函数  
 **rabbitTemplate**.setConfirmCallback(**new** RabbitTemplate.ConfirmCallback() {  
 */\*\*  
 \*  
 \** ***@param correlationData*** *相关配置信息  
 \** ***@param ack*** *exchange交换机是否成功收到消息。true代表成功  
 \** ***@param cause*** *失败原因  
 \*/* @Override  
 **public void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  
 System.***out***.println(**"confirm方法执行了"**);  
 **if**(ack){  
 //exchange成功接收到消息  
 System.***out***.println(**"exchange接收成功消息"** + cause);  
 }**else** {  
 System.***out***.println(**"exchange接收失败消息"** + cause);  
 //做一些处理，让消息再次发送  
 }  
 }  
 });  
 //定义消息  
 String body = **"message confirm..."**;  
 //发送消息  
 **rabbitTemplate**.convertAndSend(**"test\_exchange\_confirm"**,**"confirm"**,body);  
 }   
}

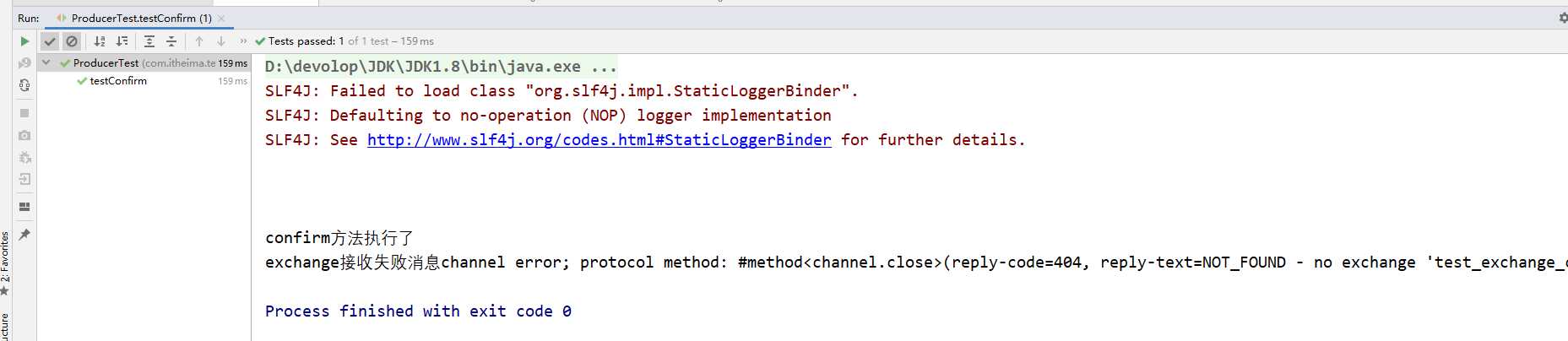
#### 3.3测试

**注意回调函数中的逻辑。**

当消息发送成功时：



当消息发送失败时，比如我们将交换机的名字后面加一个1，就会发送失败了



### 4、退回模式

***下面展示的是退回模式下的生产者代码。***

只展示测试类的代码了。因为其余部分与确认模式一样。

#### 4.1代码

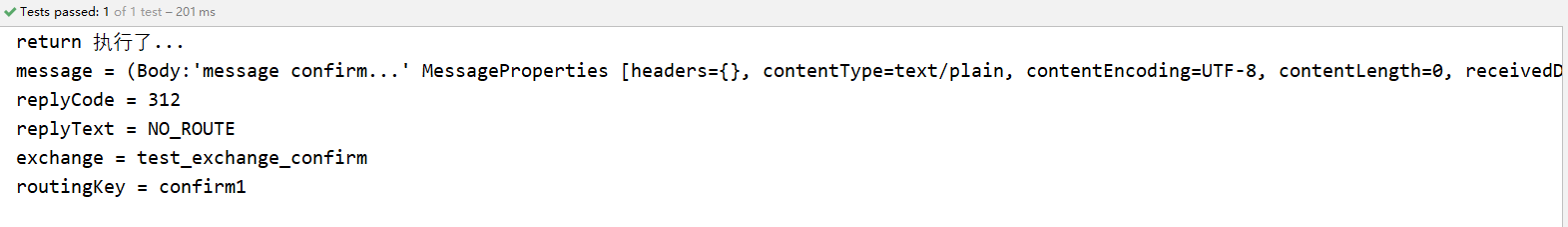
/\*  
退回模式:当消息发送给exchange后，exchange路由到queue失败时，才会执行ReturnCallBack  
步骤:  
 1.回退模式开启：publisher-returns="true"  
 2.设置ReturnCallBack  
 3.设置exchange处理消息的模式：  
 1.如果消息没有路由到queue，则丢弃消息（默认）  
 2.如果消息没有路由到queue，则返回给消息发送方ReturnCallBack  
 \*/  
@Test  
**public void** testReturn() {  
  
 //设置交换机处理失败消息的模式  
 **rabbitTemplate**.setMandatory(**true**);  
 //设置ReturnCallBack  
 **rabbitTemplate**.setReturnCallback(**new** RabbitTemplate.ReturnCallback() {  
 */\*\*  
 \*  
 \** ***@param message*** *消息对象  
 \** ***@param replyCode*** *错误码  
 \** ***@param replyText*** *错误信息  
 \** ***@param exchange*** *交换机  
 \** ***@param routingKey*** *路由键  
 \*/* @Override  
 **public void** returnedMessage(Message message, **int** replyCode, String replyText, String exchange, String routingKey) {  
 System.***out***.println(**"return 执行了..."**);  
  
 System.***out***.println(**"message = "** + message);  
 System.***out***.println(**"replyCode = "** + replyCode);  
 System.***out***.println(**"replyText = "** + replyText);  
 System.***out***.println(**"exchange = "** + exchange);  
 System.***out***.println(**"routingKey = "** + routingKey);  
 }  
 });  
 //定义消息  
 String body = **"message confirm..."**;  
 //发送消息  
 **rabbitTemplate**.convertAndSend(**"test\_exchange\_confirm"**,**"confirm1"**,body);  
}

#### 4.2测试

在发送消息时，我们故意将routing key写错，在confirm后面加了一个1，但是交换机名字写的是正确的。

这样消息可以发送到交换机，但是无法从交换机发送到队列。这样

returnCallBack方法才会执行。



此外，设置开启回退模式的位置与确认模式一样，都是在配置文件2的

<rabbit:connection-factory>标签中进行设置，加一条属性：

publisher-returns="true"。



### 5、小结

* 设置ConnectionFactory的publisher-confirms="true" 开启 确认模式。
* 使用rabbitTemplate.setConfirmCallback设置回调函数。当消息发送到exchange后回调confirm方法。在方法中判断ack，如果为true，则发送成功，如果为false，则发送失败，需要处理。
* 设置ConnectionFactory的publisher-returns="true" 开启 退回模式。
* 使用rabbitTemplate.setReturnCallback设置退回函数，当消息从exchange路由到queue失败后，如果设置了rabbitTemplate.setMandatory(true)参数，则会将消息退回给producer。并执行回调函数returnedMessage。
* 在RabbitMQ中也提供了事务机制，但是性能较差，此处不做讲解。

使用channel下列方法，完成事务控制：

txSelect(), 用于将当前channel设置成transaction模式

txCommit()，用于提交事务

txRollback(),用于回滚事务

## 二、Consumer ACK

### 1、概述

ack指Acknowledge，确认。 表示消费端收到消息后的确认方式。

有三种确认方式：

* 自动确认：acknowledge="none"
* 手动确认：acknowledge="manual"
* 根据异常情况确认：acknowledge="auto"，（这种方式使用麻烦，不作讲解）

其中自动确认是指，当消息一旦被Consumer接收到，则自动确认收到，并将相应 message 从 RabbitMQ 的消息缓存中移除。

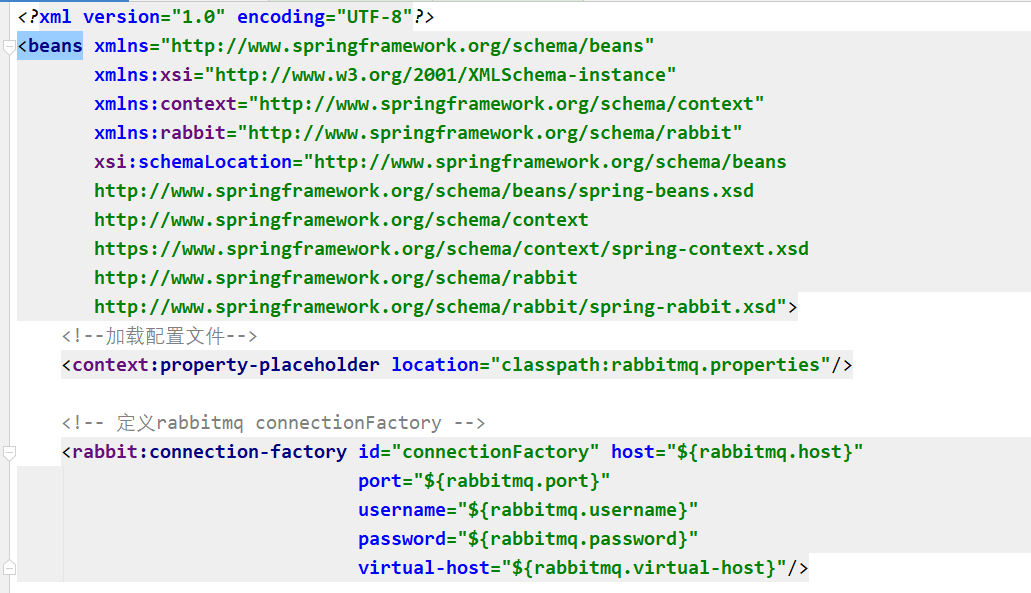
但是在实际业务处理中，很可能消息接收到，业务处理出现异常，那么该消息就会丢失。

如果设置了手动确认方式，则需要在业务处理成功后，调用

channel.basicAck()，手动签收，如果出现异常，则调用channel.basicNack()方法，让其自动重新发送消息。

这一部分知识，全都是针对消费者的

### 2、配置文件





### 3、监听器

注意监听器所在的包，要与配置文件中的包扫描对应，否则扫描不到注解，就无法将其加入ioc容器。

另外监听器的类名，必须与配置文件中的完全一致。

代码：

/\*  
Consumer Ack机制:  
 1.设置手动签收。acknowledge="manual"  
 2.让监听器实现ChannelAwareMessageListener接口  
 3.如果消息处理成功，则调用channel的basicAck()签收  
 4.如果消息处理失败，则调用channel的basicNack()拒绝签收，broker重新发送给consumer  
 \*/  
@Component  
**public class** AckListener **implements** ChannelAwareMessageListener {  
 @Override  
 **public void** onMessage(Message message, Channel channel) **throws** Exception {  
 Thread.*sleep*(1000);//执行该方法时，先睡1秒  
 //获取消息的投递标签  
 **long** deliveryTag = message.getMessageProperties().getDeliveryTag();  
 **try** {  
 //1.获取消息数据  
 String body = **new** String(message.getBody());  
 System.***out***.println(**"body = "** + body);//打印消息  
 //2.处理业务逻辑  
 System.***out***.println(**"处理业务逻辑..."**);  
 //int i = 3 / 0; //出现错误  
  
 //3.手动签收，第二个参数multiple为true时，表示可以签收多个消息  
 channel.basicAck(deliveryTag,**true**);  
 } **catch** (Exception e) {  
 //4.处理消息失败，拒绝签收  
 //第三个参数：requeue——重回队列。如果设置为true，  
 //则消息重新回到queue，broker会重新发送该消息给消费端  
 channel.basicNack(deliveryTag,**true**,**true**);  
 //channel.basicReject(deliveryTag,true); //处理1个  
 }  
 }  
}

### 4、测试

我们写一个测试类：

@RunWith(SpringJUnit4ClassRunner.**class**)  
@ContextConfiguration(locations = **"classpath:mq.consumer.spring.xml"**)  
**public class** AckListenerTest {  
  
 @Test  
 **public void** test() {  
 **while** (**true**){}  
 }  
}

该方法的主要目的是读取消费者的配置文件，就会自动执行监听器里的方法。另外就是死循环。

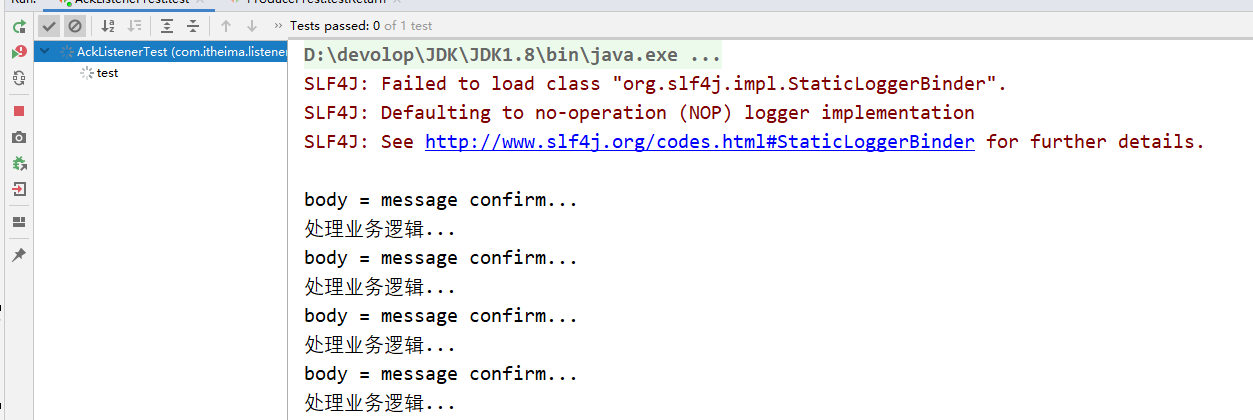
测试：

当我们打开报错：int i = 3 / 0时，代表处理业务逻辑时报错，进入catch，执行basicNack方法，这样消息就会重新回到queue，broker会重新发送消息给消费端：

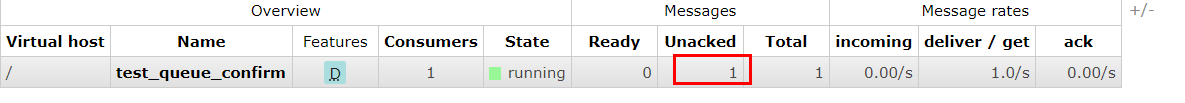
比如我们先让生产者发送消息：  


ready：1.表示该队列中有1条消息，准备发送给消费者，但是还未发送

然后我们启动消费者（打开了报错的）



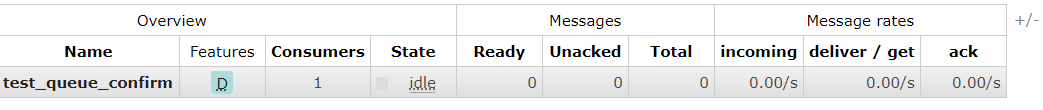
每个1秒，就重复发送1个消息，无线循环



unacked：1.表示有一条消息正在从broker发送个消费者，但是未被确认签收。

然后我们将报错注释掉，重启启动消费者的测试类：

发现终于不再重复发送消息：



### 5、小结

Consumer Ack小结：

* 在rabbit:listener-container标签中设置acknowledge属性，设置ack方式 none：自动确认，manual：手动确认
* 如果在消费端没有出现异常，则调用channel.basicAck(deliveryTag,false);方法确认签收消息
* 如果出现异常，则在catch中调用 basicNack或 basicReject，拒绝消息，让MQ重新发送消息。

消息可靠性小结：

1、持久化

* + exchange要持久化
  + queue要持久化
  + message要持久化

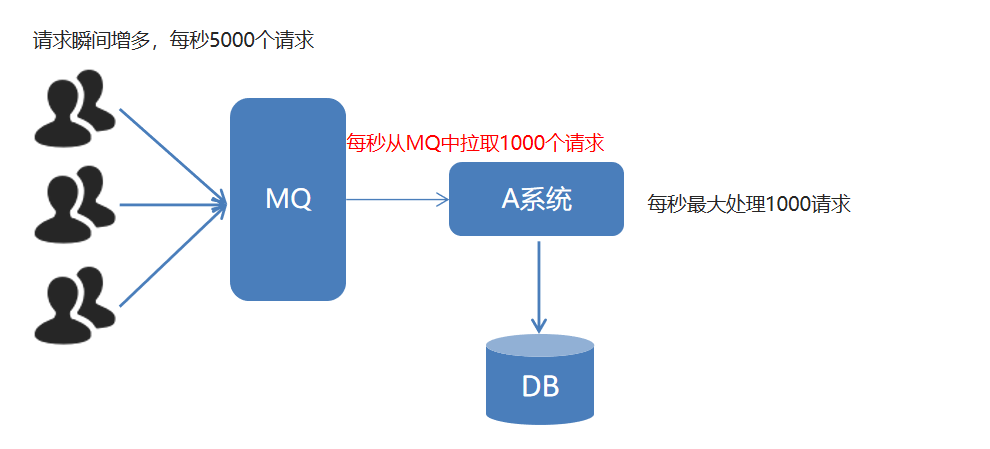
2、生产方确认Confirm

3、消费方确认Ack

4、Broker高可用

## 三、消费端限流

### 1、概述



在<rabbit:listener-container> 中配置 prefetch属性设置消费端一次拉取多少消息

消费端的确认模式一定为手动确认。acknowledge="manual"

### 2、配置文件

只展示一部分



### 3、监听器

QosListener：

/\*  
consumer限流机制  
 1.确保ack机制为手动确认manual  
 2.listener-container配置属性  
 prefetch = "1"，表示消费者每次从mq拉取一条消息进行消费，知道手动确认消费完毕后，才会继续拉取下一条消息  
 \*/  
@Component  
**public class** QosListener **implements** ChannelAwareMessageListener {  
 @Override  
 **public void** onMessage(Message message, Channel channel) **throws** Exception {  
 Thread.*sleep*(1000);//睡1秒  
 //1.获取消息并打印  
 String body = **new** String(message.getBody());  
 System.***out***.println(**"body = "** + body);  
 //2.获取投递标签  
 **long** deliveryTag = message.getMessageProperties().getDeliveryTag();  
 //3.处理业务逻辑  
  
 //4.确认签收  
 channel.basicAck(deliveryTag,**true**);  
 }  
}

### 4、发送消息

在服务端测试类中发送消息：

代码：

//发送消息  
@Test  
**public void** testSend() {  
 String body = **null**;  
 **for** (**int** i = 1; i <= 10; i++) {  
 //定义消息  
 body = i + **"\_message confirm..."**;  
 //发送消息  
 **rabbitTemplate**.convertAndSend(**"test\_exchange\_confirm"**,**"confirm"**,body);  
 }  
}

一次性发送10条消息

### 5、测试

#### 5.1测试1

现在我们在消费者的配置文件中设置的是手动签收，并且每次只能签收1条消息：



然后我们将监听器中的确认签收给注释掉：

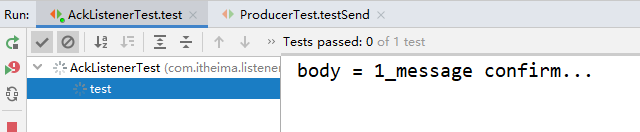
//4.确认签收  
//channel.basicAck(deliveryTag,true);

然后一次性发送10条消息，再开启消费者的测试类中的方法：

我们预计的效果应该是：

因为没有签收消息，并且一次性又只能拉取1条消息，那么队列中的10条消息应该只能获取到1条，并且这一条还应该是处于未确认unacked状态，而另外9条则应该是出于准备被消费状态ready

消费者只获取到1条消息，并且这条消息未被确认签收





#### 5.2测试2

我们重新测试，生产者重新发送10条消息。

我们打开监听器的确认签收代码

//4.确认签收  
channel.basicAck(deliveryTag,**true**);

还是保持prefetch=1，每次只能拉取一条消息。另外线程睡1秒。

那么预计效果应该是：

一次拉取1条消息进行消费，消费完就被确认，然后拉取下一条。



差不多每隔1秒就打印一次

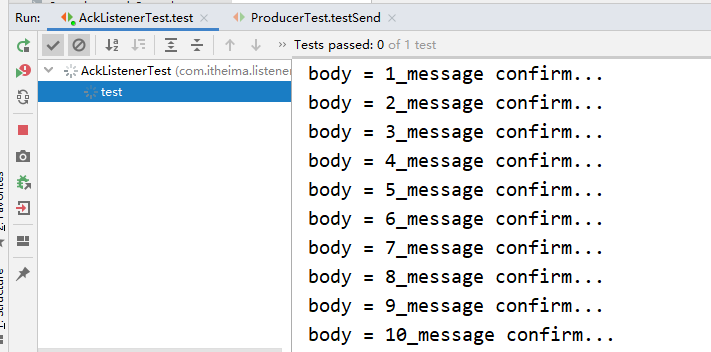
#### 5.3测试3

我们关闭确认消息代码，但是将配置文件中的prefetch=“1”删掉

那么就应该是一次性拉取10条了：

将测试3与测试1进行对比，就能发现prefetch的作用了。

测试结果：





每秒打印一次，我们明明没有确认，但是却将10条消息都获取到了。说明没有设置prefetch属性时，就会直接一次性将队列中的消息都给拉取过来了。

### 6、小结

在<rabbit:listener-container> 中配置 prefetch属性设置消费端一次拉取多少消息

消费端的确认模式一定为手动确认。acknowledge="manual"

## 四、TTL

### 1、概述

* TTL 全称 Time To Live（存活时间/过期时间）。
* 当消息到达存活时间后，还没有被消费，会被自动清除。
* RabbitMQ可以对消息设置过期时间，也可以对整个队列（Queue）设置过期时间。
* 如果设置了消息的过期时间，也设置了队列的过期时间，将以时间短的为准。
* 队列过期后，会将队列中的所有消息全部移除，但是队列自己还是存在的。
* 消息过期后，只有消息在队列顶端（如果这条消息即将被消费，那么说明这条消息在队列的顶端），才会判断是否过期（过期就会被移除掉）。

我们先来看一下再管控台怎么设置消息/队列的过期时间，然后再学习如何使用Java设置过期时间。

### 2、管控台演示

*在管理控制台演示消息存活时间*

(1) 新增队列

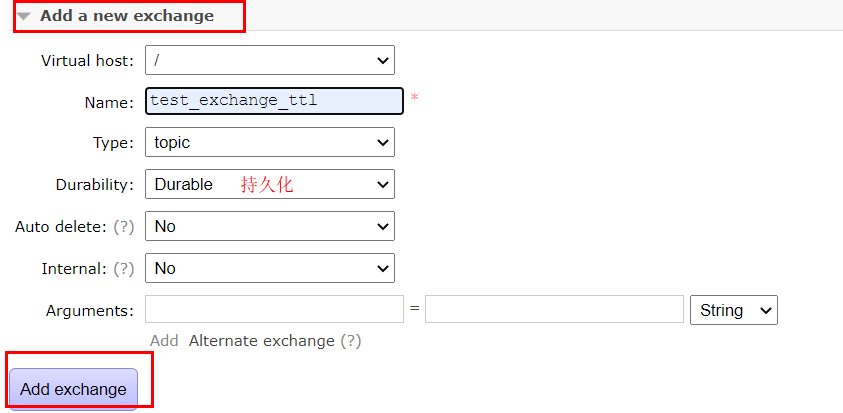
我们在Queues部分：

点击“Add a new queue”，增加队列



x-message-ttl：消息存活时间，比如我们设置为10000，就是10秒

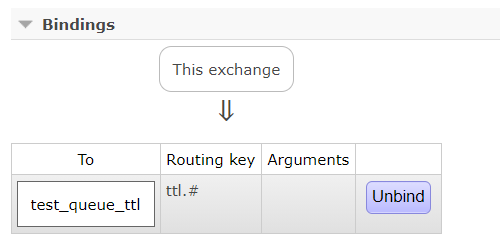
(2) 新增交换机



(3) 绑定交换机与队列



绑定好之后效果：



(4 )发布消息：

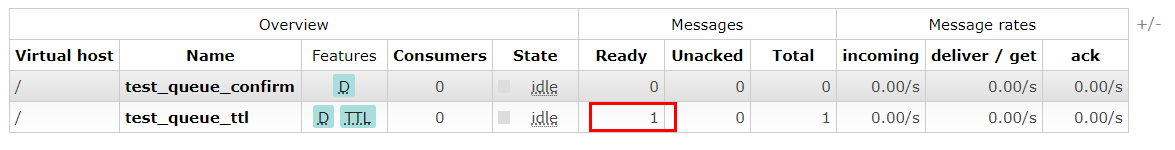
就在bind的下面有一个Publish message，可以发布消息



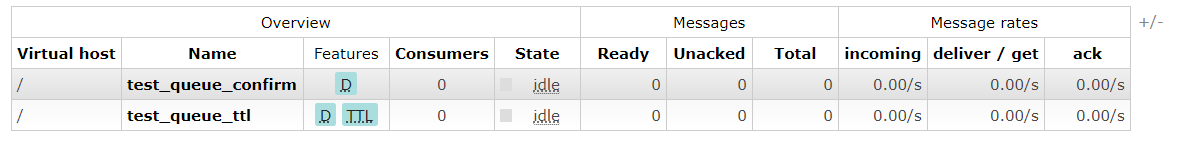
点击Publish message发布消息



然后观察队列，可以看到有一条消息出于ready状态



然后过了10秒，这条消息就消失了：



这样一来，我们就成功实现了如何在管控台演示消息的存活时间，接下来我们将使用Java来演示

### 3、队列统一过期

设置消息存活时间是在生产者这边操作的。

#### 3.1配置文件

生产者的配置文件：

图片：



代码：

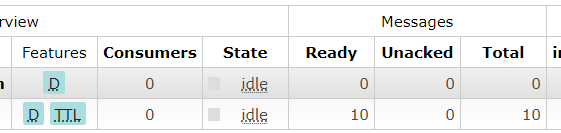
<!--TTL-->  
<**rabbit:queue id="test\_queue\_ttl" name="test\_queue\_ttl"**>  
 <!--设置queue的参数-->  
 <**rabbit:queue-arguments**>  
 <!--x-message-ttl:队列过期时间，单位毫秒-->  
 <!--值的类型默认是String类型，需要设置为Integer类型，否则会报错-->  
 <**entry key="x-message-ttl" value="10000" value-type="java.lang.Integer"**></**entry**>  
 </**rabbit:queue-arguments**>  
</**rabbit:queue**>  
  
<**rabbit:topic-exchange name="test\_exchange\_ttl"**>  
 <**rabbit:bindings**>  
 <**rabbit:binding pattern="ttl.#" queue="test\_queue\_ttl"**></**rabbit:binding**>  
 </**rabbit:bindings**>  
</**rabbit:topic-exchange**>

#### 3.2测试代码

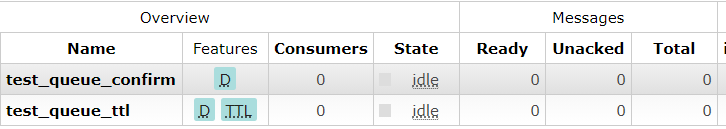
代码：

//发送10条消息，ttl  
@Test  
**public void** testTtl() {  
 **for** (**int** i = 1; i <= 10; i++) {  
 //定义消息  
 String body = i + **"\_hello ttl..."**;  
 //发送消息 **rabbitTemplate**.convertAndSend(**"test\_exchange\_ttl"**,**"ttl.hehe"**,body);  
 }  
}

测试结果：

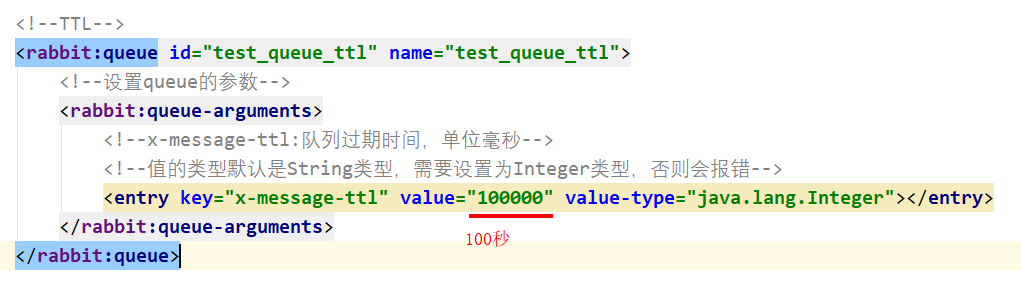


10秒后



### 4、消息单独过期

首先我们将队列的存活时间设置为100秒



#### 4.1测试1

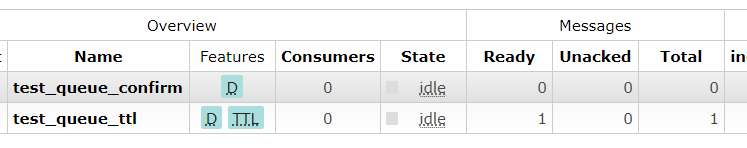
基本用法，设置消息的存活时间为5秒

测试类代码：

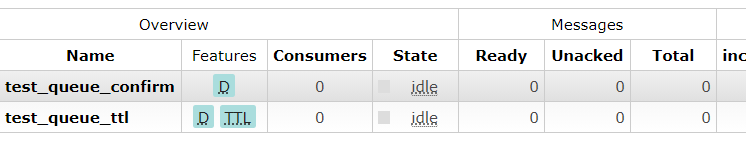
//ttl  
@Test  
**public void** testTtl() {  
 //消息后处理对象，可以设置一些消息的参数信息,可用Lambda  
 MessagePostProcessor messagePostProcessor = **new** MessagePostProcessor() {  
 @Override  
 **public** Message postProcessMessage(Message message) **throws** AmqpException {  
 //1.设置message的信息  
 message.getMessageProperties().setExpiration(**"5000"**);//消息的过期时间，5秒  
 //2.返回消息  
 **return** message;  
 }  
 };  
 //定义消息  
 String body = **"hello ttl..."**;  
 //发送消息  
 **rabbitTemplate**.convertAndSend(**"test\_exchange\_ttl"**,**"ttl.hehe"**,body,messagePostProcessor);  
}

效果：

消息的存活时间设置的是5秒，而队列的是100秒。我们先将队列删除，然后执行这个方法：



5秒后：

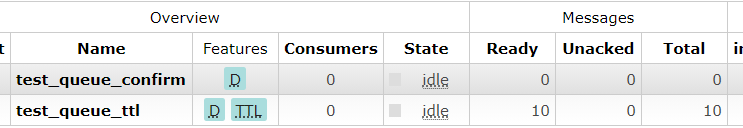


#### 4.2测试2

发送10条消息，当i==5时，发送1条过期消息（即存活时间只有5秒的消息），其余9条都是不过期的消息：

测试方法：  


效果：



发现消息一直是10条，那条过期消息过了5秒也没有消失。

因为i等于5时，才发送的那条过期消息，也就是说过期消息是在中间，而不是在队列的顶端。因此不会被移除掉（不是不会过期，只是不会被移除，只有当它出于队列顶端时，才会被移除）。

正因如此，我们一般都是对队列整体来设置过期时间，而不是单独对某条消息设置过期时间。

### 5、小结

* 设置队列过期时间使用参数：x-message-ttl，单位：ms(毫秒)，会对整个队列消息统一过期。
* 设置消息过期时间使用参数：expiration。单位：ms(毫秒)，当该消息在队列头部时（消费时），会单独判断这一消息是否过期。
* 如果两者都进行了设置，以时间短的为准。

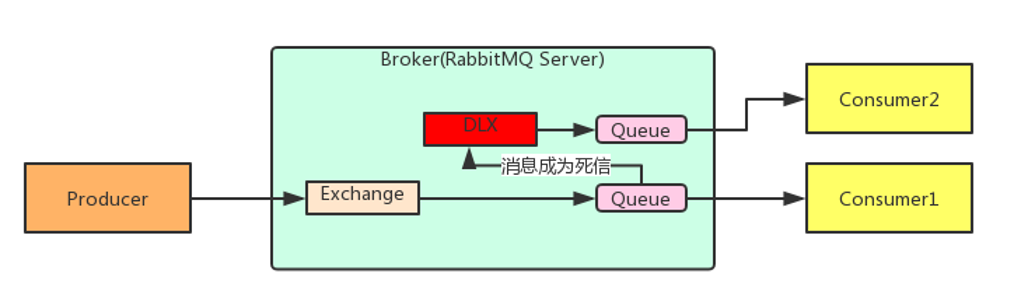
## 五、死信队列

### 1、概念

死信队列：

死信队列，英文缩写：DLX 。Dead Letter Exchange（死信交换机），当消息成为Dead message后，可以被重新发送到另一个交换机，这个交换机就是DLX。

各种MQ产品中基本都有死信队列的概念，只不过只有RabbitMQ中才有交换机的概念。在RabbitMQ中，死信队列指的不是队列，而是一种特殊的交换机。

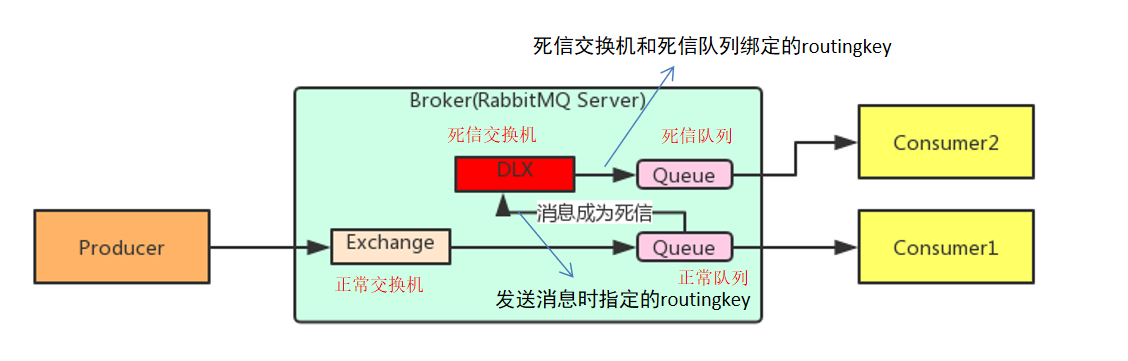
图解：  


消息成为死信的三个种情况：

* 队列消息长度到达限制；
* 消费者拒接消费消息，basicNack/basicReject，并且不把消息重新放入原目标队列，requeue=false；
* 原队列存在消息过期设置，消息到达超时时间未被消费；

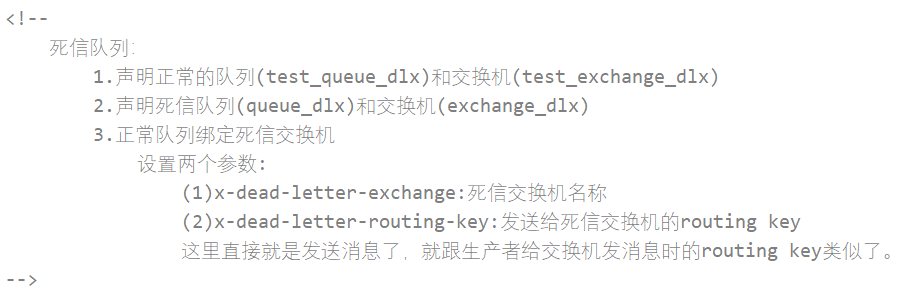
队列绑定死信交换机：

给队列设置参数： x-dead-letter-exchange 和 x-dead-letter-routing-key



### 2、配置文件

***生产者的配置文件***





### 3、发送消息测试

#### 3.1测试1

***测试消息过期之后，是否成为死信***

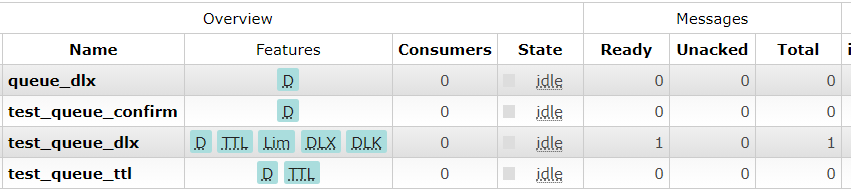
代码：

//发送死信消息测试1:过期时间  
@Test  
**public void** testDlx1() {  
 //定义消息  
 String body = **"我是一条消息，我会成为死信吗？"**;  
 //给正常交换机发送消息  
**rabbitTemplate**.convertAndSend(**"test\_exchange\_dlx"**,**"test.dlx.zhengchang"**,body);  
}

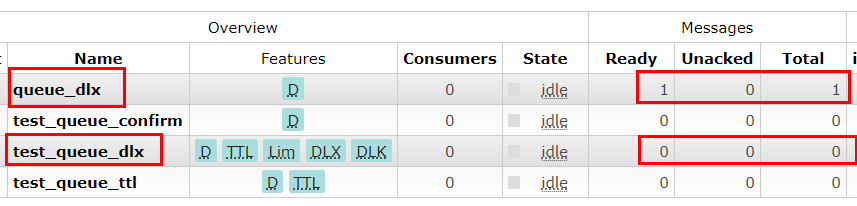
消息分析：

生产者——>正常交换机——>正常队列——>10秒后正常队列里消息全部过期，消息成为死信——>死信交换机——>死信队列

效果：



10秒后：



我们点击队列“queue\_dlx”，然后“Get Messages”：



可以看到，最终转发到死信队列的消息数据，果然就是我们之前发给正常交换机的消息数据。

这说明消息过期后成为了死信，并且成功转发给了死信交换机。

#### 3.2测试2

***测试超过长度限制，消息是否成为死信***

代码：

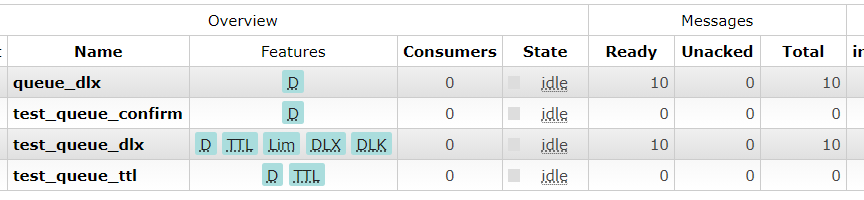
//发送死信消息测试2:长度限制  
@Test  
**public void** name() {  
 String body = **null**;  
 **for** (**int** i = 1; i <= 20; i++) {  
 body = i + **"\_超过队列长度限制的消息，会成为死信吗？"**;  
 **rabbitTemplate**.convertAndSend(**"test\_exchange\_dlx"**,**"test.dlx.zc"**,body);  
 }  
}

分析：

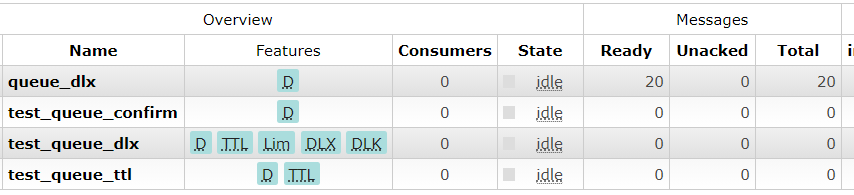
一次性发送了20条消息，但是该正常队列我们设置的最多只能容纳10条消息，那么一开始超出的10条消息就会直接成为死信，转发给死信交换机，最终发送到死信队列。而10秒过后，正常队列的消息全部过期，也成为死信，最终也会发送到死信队列。

效果：

一开始，正常队列、死信队列各有10条消息



10秒后，正常队列10条消息，死信队列20条消息



查看消息



至于第三种情况，消息拒收，因为还涉及到消费者，要写的内容比较多，因此放在第4部分进行介绍。

### 4、消息拒收

***演示消息成为死信的第三种情况：消息拒收***

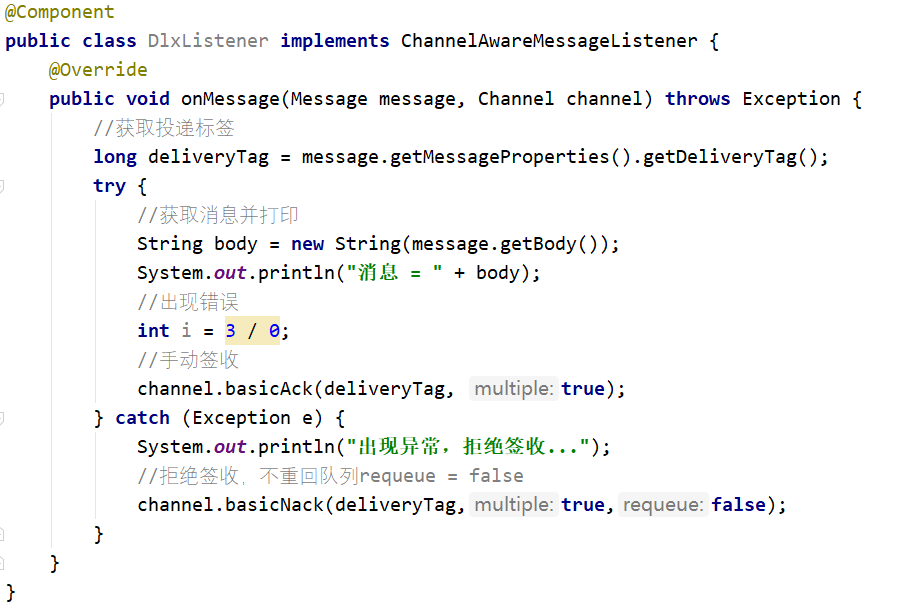
#### 4.1发送消息

在发送端的测试类中，给正常队列test\_queue\_dlx发送一条消息

@Test  
**public void** testDlx1() {  
 //定义消息  
 String body = **"我是一条消息，我会成为死信吗？"**;  
 //给正常交换机发送消息  
 **rabbitTemplate**.convertAndSend(**"test\_exchange\_dlx"**,**"test.dlx.zhengchang"**,body);  
}

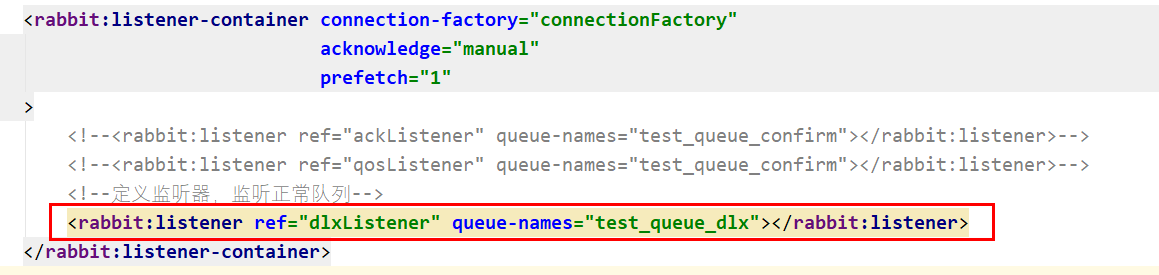
#### 4.2监听器

消费者的监听器，签收消息之前出现异常，进入catch，拒绝签收，并且让消息不重回原来的正常队列，这样才能作为死信转发到死信交换机，然后进入死信队列。



#### 4.3消费者配置

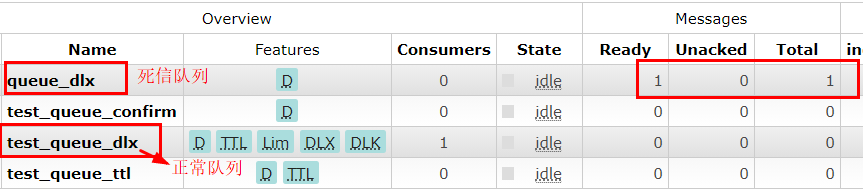
在消费端配置监听器，监听正常队列test\_queue\_dlx



#### 4.4测试

启动消费端，然后启动生产端发送给正常交换机发送一条消息。

因为消息在签收之前出现报错，进入catch，消息被拒收，并且不能重返正常队列，而是转发到死信交换机，进而发送到死信队列。

效果：  


看到的效果是消息直接进入死信队列，而没有进入正常队列。最多是在正常队列阻塞unacked一小会，然后进入死信队列。

### 5、小结

1. 死信交换机和死信队列与普通的没有区别，关键的设置就在正常队列中进行配置

2. 当消息成为死信后，如果该队列绑定了死信交换机，则消息会被死信交换机重新路由到死信队列

3. 消息成为死信的三种情况：

* 队列消息长度到达限制；
* 消费者拒接消费消息，并且不重回队列；
* 原队列存在消息过期设置，消息到达超时时间未被消费；

## 六、延迟队列

### 1、概念

延迟队列：

即消息进入队列后不会立即被消费，只有到达指定时间后，才会被消费。

需求：

1. 下单后，30分钟未支付，取消订单，回滚库存。

2. 新用户注册成功7天后，发送短信问候。

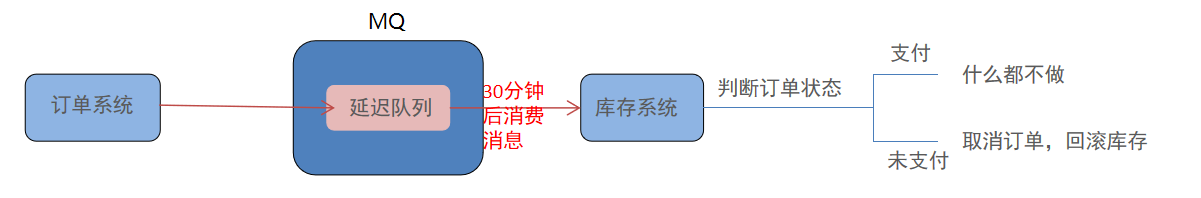
实现方式：

1. 定时器

2. 延迟队列

定时器并不优雅，既然学了MQ，我们就用MQ中延迟队列的思想来实现。

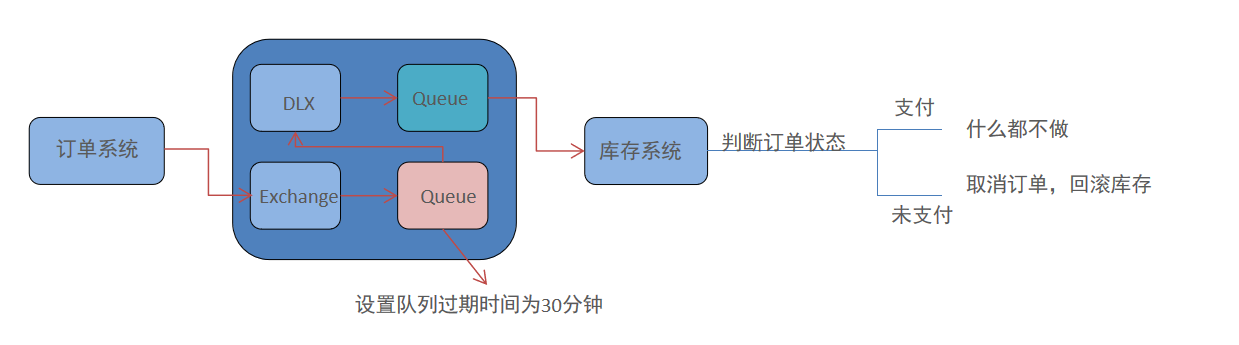
思路：



不过很可惜，在RabbitMQ中并没有提供延迟队列的功能。

但是可以用TTL + DLX（死信队列）组合实现延迟队列的效果

例如：



接下来我们就在代码中来实现延迟队列。

关键就是生产者的配置信息

### 2、配置文件

#### 2.1生产者

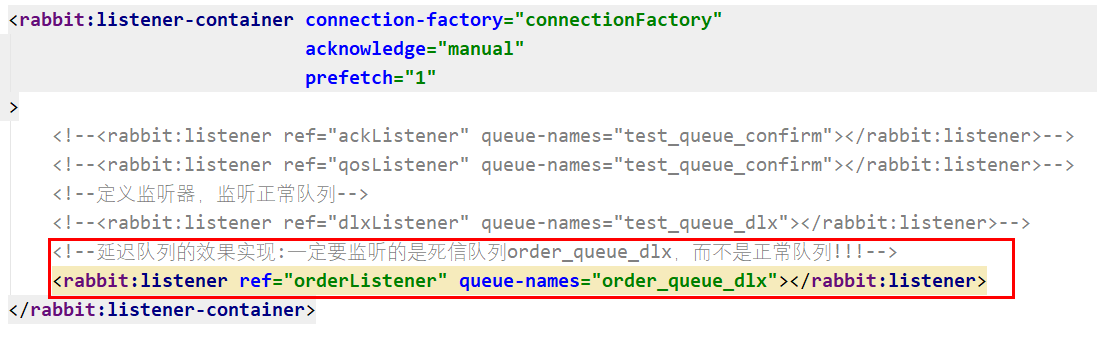
*生产者的配置文件2*

延迟队列:  
 1.定义正常交换机(order\_exchange)和正常队列(order\_queue)  
 2.定义死信交换机(order\_exchange\_dlx)和死信队列(order\_queue\_dlx)  
 3.在正常队列中绑定死信交换机，设置正常队列的过期时间为10秒钟



#### 2.2消费者

*消费者的配置文件2*



千万注意：

监听器监听的是死信队列，而不是正常队列，否则无法实现延迟效果。

因为生产者一发送消息，正常队列就收到消息了，而死信队列要等10秒才能收到消息，从而实现延迟效果。

### 3、监听器



### 4、测试

#### 4.1发送消息

在生产端的测试类中给正常交换机发送消息：

代码：

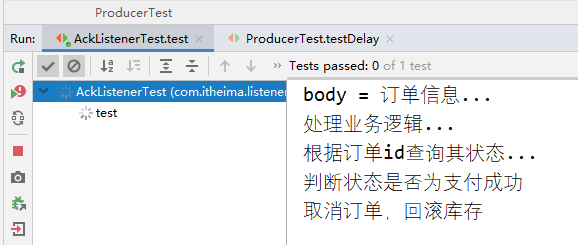
//延迟队列  
@Test  
**public void** testDelay() **throws** InterruptedException {  
 //定义消息  
 String body = **"订单信息..."**;  
 //发送消息  
 **rabbitTemplate**.convertAndSend(**"order\_exchange"**,**"order.msg"**,body);  
 //打印倒计时10秒  
 **for** (**int** i = 10; i > 0; i--) {  
 System.***out***.println(**"i = "** + i);  
 Thread.*sleep*(1000);//每打印1次，睡1秒  
 }  
}

#### 4.2测试效果

每个10秒打印一次



10秒过后，消费端才接收到消息，从而消费消息：



### 5、小结

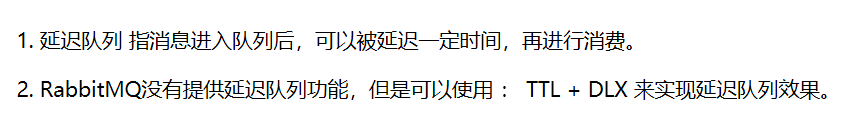
我自己的小结：

学习延迟队列与学习死信队列差不多

关键在于：

* 延迟队列中，消费端监听器所监听的是死信队列，而不是正常队列
* 正常队列中设置过期时间DDL，也就是延迟时间

老师的小结：

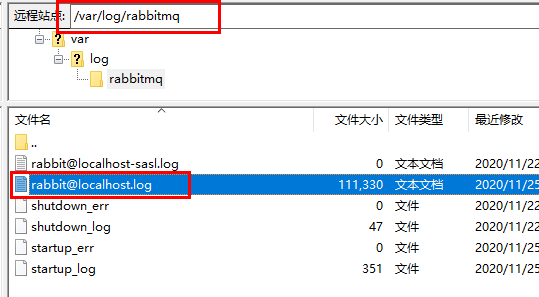


## 七、日志与监控

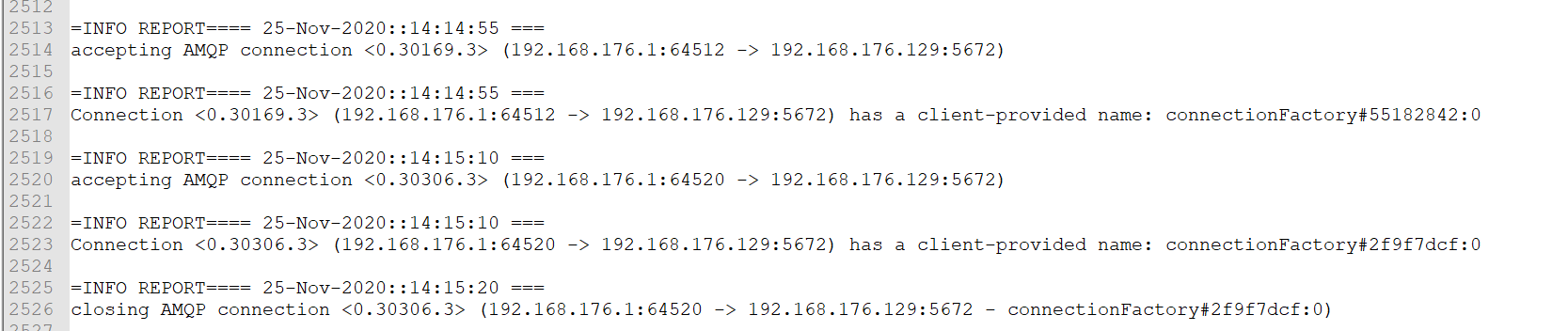
**这部分内容了解即可**

### 1、RabbitMQ日志

* RabbitMQ默认日志存放路径： /var/log/rabbitmq/rabbit@xxx.log
* 日志包含了RabbitMQ的版本号、Erlang的版本号、RabbitMQ服务节点名称、cookie的hash值、RabbitMQ配置文件地址、内存限制、磁盘限制、默认账户guest的创建以及权限配置等等。

我们可以打开看一下：  


我们主要就看rabbit@localhost.log这个文件



最下面记录的是RabbitMQ的最新日志信息

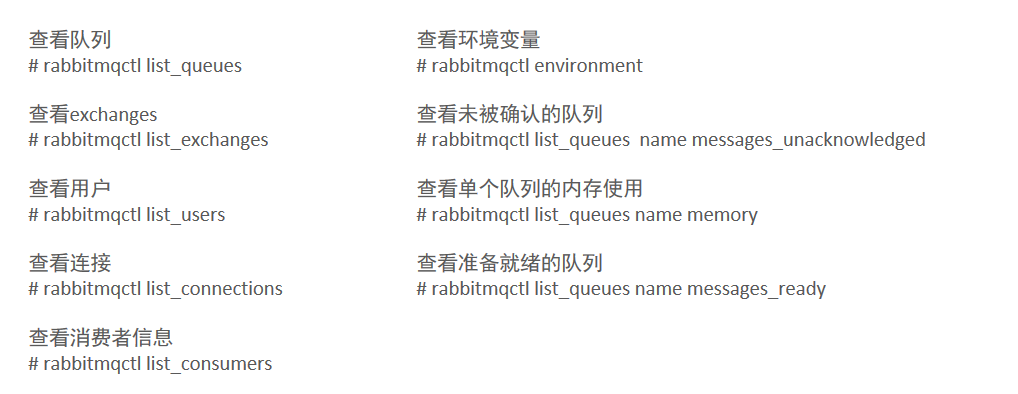
### 2、web管控台监控

具体看视频：《11\_高级特性-日志与监控.mp4》

除了可以通过管控台监控以外，还可以通过一些命令。

### 3、rabbitmqctl管理和监控

命令图片：



这些命令都是在CentOS7上使用的

查看队列

# rabbitmqctl list\_queues

查看exchanges

# rabbitmqctl list\_exchanges

查看用户

# rabbitmqctl list\_users

查看连接

# rabbitmqctl list\_connections

查看消费者信息

# rabbitmqctl list\_consumers

查看环境变量

# rabbitmqctl environment

查看未被确认的队列

# rabbitmqctl list\_queues name messages\_unacknowledged

查看单个队列的内存使用

# rabbitmqctl list\_queues name memory

查看准备就绪的队列

# rabbitmqctl list\_queues name messages\_ready

## 八、消息追踪

### 1、概述

目前暂时了解即可，以后有需要可再重新看视频学习。

这个开启之后会影响性能，一般用于排查bug。

消息追踪：

在使用任何消息中间件的过程中，难免会出现某条消息异常丢失的情况。对于RabbitMQ而言，可能是因为生产者或消费者与RabbitMQ断开了连接，而它们与RabbitMQ又采用了不同的确认机制；也有可能是因为交换器与队列之间不同的转发策略；甚至是交换器并没有与任何队列进行绑定，生产者又不感知或者没有采取相应的措施；另外RabbitMQ本身的集群策略也可能导致消息的丢失。这个时候就需要有一个较好的机制跟踪记录消息的投递过程，以此协助开发和运维人员进行问题的定位。

在RabbitMQ中可以使用Firehose和rabbitmq\_tracing插件功能来实现消息追踪。

### 2、Firehose

firehose的机制是将生产者投递给rabbitmq的消息，rabbitmq投递给消费者的消息按照指定的格式发送到默认的exchange上。这个默认的exchange的名称为amq.rabbitmq.trace，它是一个topic类型的exchange。发送到这个exchange上的消息的routing key为 publish.exchangename 和 deliver.queuename。其中exchangename和queuename为实际exchange和queue的名称，分别对应生产者投递到exchange的消息，和消费者从queue上获取的消息。

注意：

打开 trace 会影响消息写入功能，适当打开后请关闭。

* rabbitmqctl trace\_on：开启Firehose命令
* rabbitmqctl trace\_off：关闭Firehose命令

### 3、rabbitmq\_tracing

rabbitmq\_tracing和Firehose在实现上如出一辙，只不过rabbitmq\_tracing的方式比Firehose多了一层GUI的包装，更容易使用和管理。

启用插件：rabbitmq-plugins enable rabbitmq\_tracing

# 第二章 RabbitMQ应用问题

## 一、消息可靠性保障

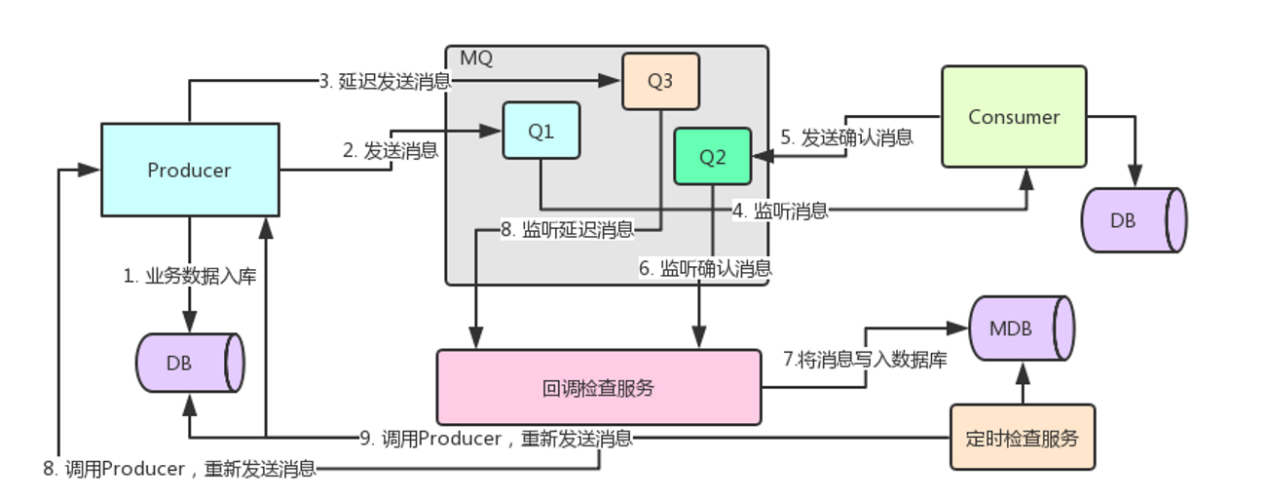
需求：

100%确保消息发送成功

方案：

采用消息补偿

消息补偿架构图：



老师花了接近7分钟来讲解这个架构图，暂时没有完全了理解。

## 二、消息幂等性保障

幂等性：

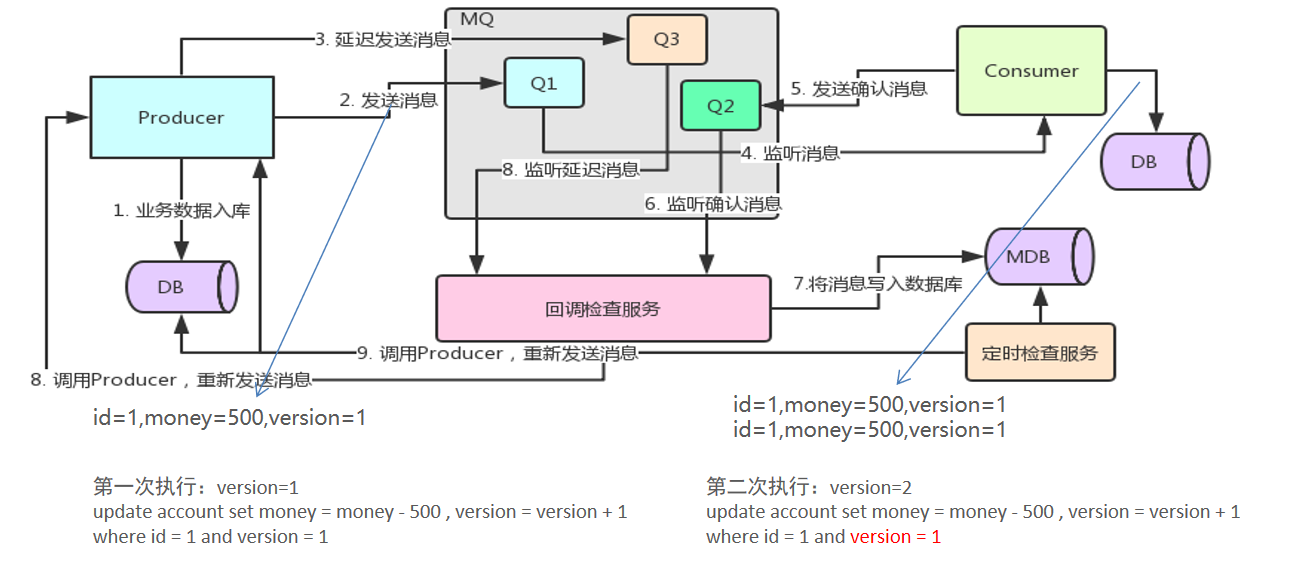
指一次和多次请求某一个资源，对于资源本身应该具有同样的结果。也就是说，其任意多次执行对资源本身所产生的影响均与一次执行的影响相同。

在MQ中指，消费多条相同的消息，得到与消费该消息一次相同的结果。

简单理解：

比如买了个东西，应该要花费500元。然后发送一条消息到服务端，中间比如说因为网络或者其其他问题，消息发送了多次，比如发送了2次。我们需要保证最终只扣款500元，而不是1000元。

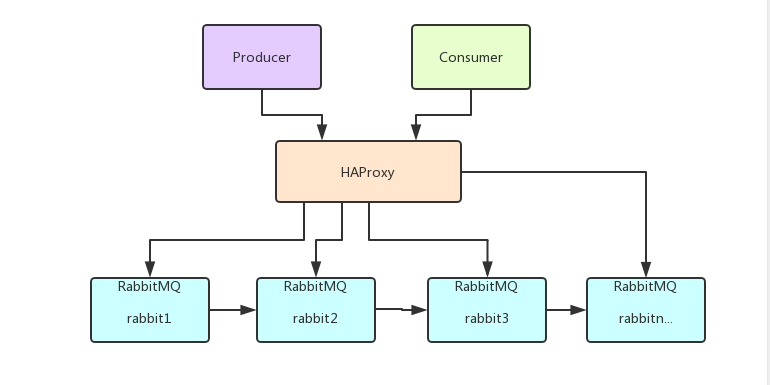
乐观锁机制实现幂等性：



# 第三章 RabbitMQ集群搭建

## 一、集群方案原理

RabbitMQ这款消息队列中间件产品本身是基于Erlang编写，Erlang语言天生具备分布式特性（通过同步Erlang集群各节点的magic cookie来实现）。因此，RabbitMQ天然支持Clustering。这使得RabbitMQ本身不需要像ActiveMQ、Kafka那样通过ZooKeeper分别来实现HA方案和保存集群的元数据。集群是保证可靠性的一种方式，同时可以通过水平扩展以达到增加消息吞吐量能力的目的。



## 二、单机多实例部署

由于某些因素的限制，有时候你不得不在一台机器上去搭建一个rabbitmq集群，这个有点类似zookeeper的单机版。真实生成环境还是要配成多机集群的。有关怎么配置多机集群的可以参考其他的资料，这里主要论述如何在单机中配置多个rabbitmq实例。

主要参考官方文档：<https://www.rabbitmq.com/clustering.html>

建议操作虚拟机之前，拍摄快照

单机多实例：

通过不同的端口来区分不同的rabbitmq。

以后在不同机器上搭建的时候，就是通过不同的IP来区分。

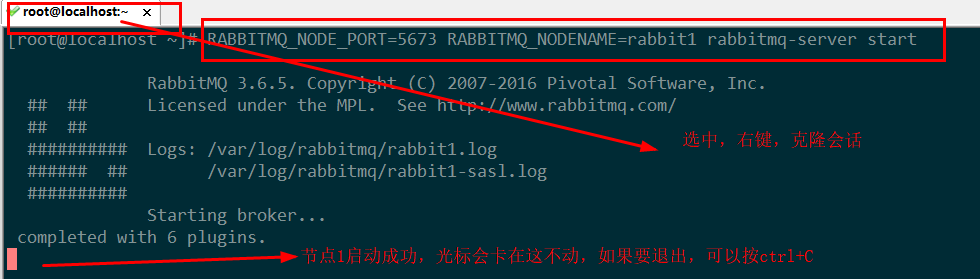
首先要确保RabbitMQ运行没有问题，在启动rabbitmq之后，输入下面的命令：

rabbitmqctl status

然后停止rabbitmq服务：

service rabbitmq-server stop

启动第一个节点：命令

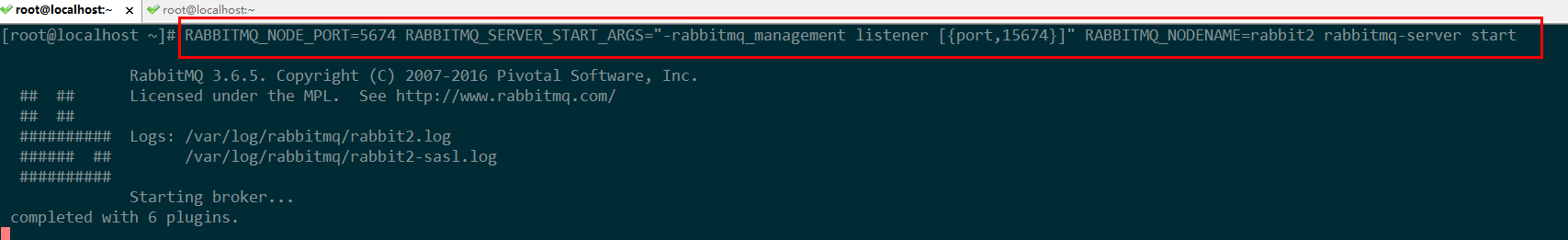


上面的5673表示的是Java访问该节点时，所使用的端口。上面我们没有写管控台的端口，那么该节点的管控台就使用默认端口15672

然后克隆一个会话窗口clone session

启动第二个节点：命令

（web管理插件端口占用,所以还要指定其web插件占用的端口号。）



Tips：

结束命令：

rabbitmqctl -n rabbit1 stop

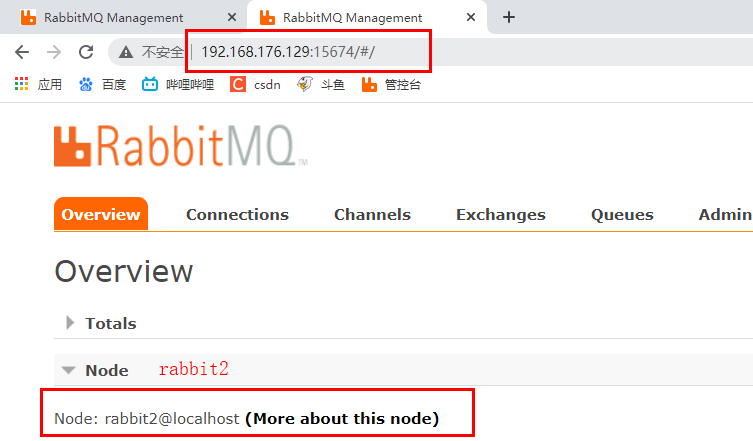
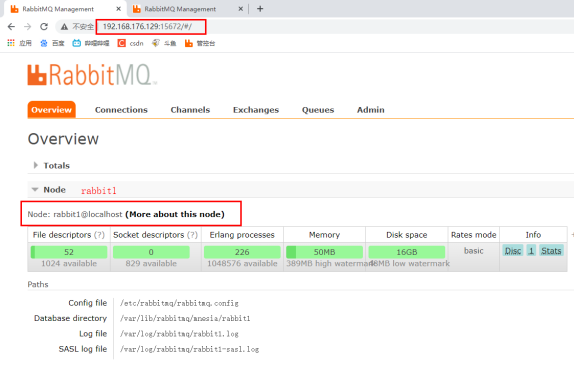
rabbitmqctl -n rabbit2 stop

如果后续操作不小心遇到bug，可以重启虚拟机。

现在我们有两个管控台：

虚拟机IP：15672

虚拟机IP：15674

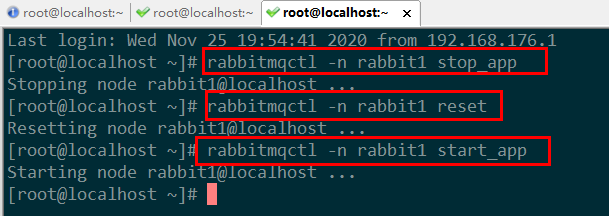


rabbit1操作作为主节点：

rabbitmqctl -n rabbit1 stop\_app

rabbitmqctl -n rabbit1 reset

rabbitmqctl -n rabbit1 start\_app



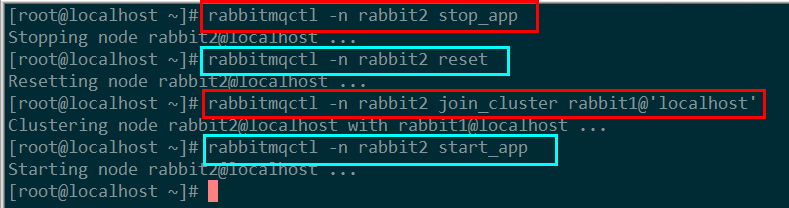
rabbit2操作为从节点：

rabbitmqctl -n rabbit2 stop\_app

rabbitmqctl -n rabbit2 reset

rabbitmqctl -n rabbit2 join\_cluster rabbit1@'localhost' //‘’内是主机名，将localhost换成自己的，比如我的就是localhost

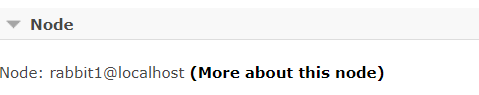
rabbitmqctl -n rabbit2 start\_app



Tips：

查看自己的主机名，在管控台可以看到：

Overview——Node



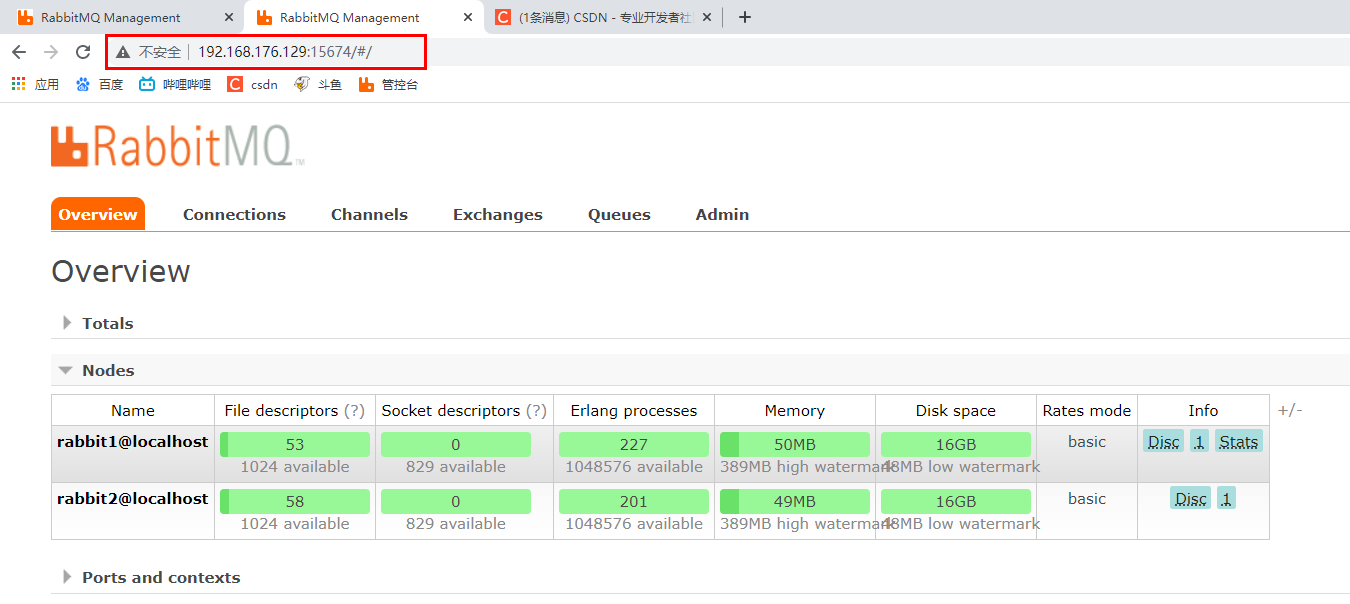
或者在CRT上，每个命令的@后面都有显示：



@后面是“localhost”，也就是该虚拟机的主机名

然后我们来看一下管控台：





现在，集群虽然搭建好了，但是还存在一个问题：数据不同步。

比如说：

我们在rabbit1中创建一个队列queue1，我们往queue1里面发布一条消息。虽然这条消息在rabbit2中也能看到，但是当rabbit这个节点挂了以后。rabbit2中的这条消息也会消失，这就失去了搭建集群的意义了。

Tips：

如何让rabbit1挂掉呢？很简单，我们之前是在CRT的一个session里面开启的rabbit1节点，只需要在该session里面按ctrl + C，那么这个节点就会挂掉，对应的虚拟机IP：15672端口的管控台也会无法访问了。

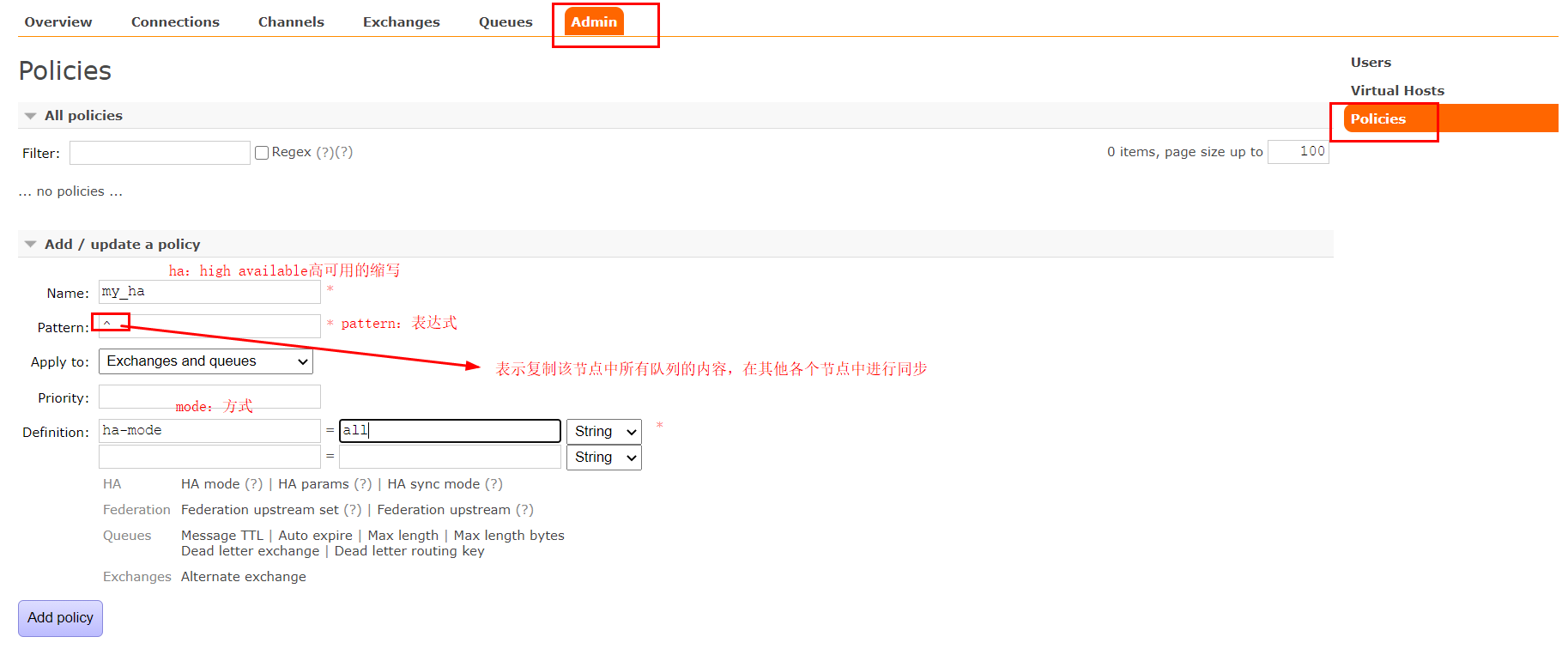
我们要解决这个问题，也很简单，可以使用镜像队列。

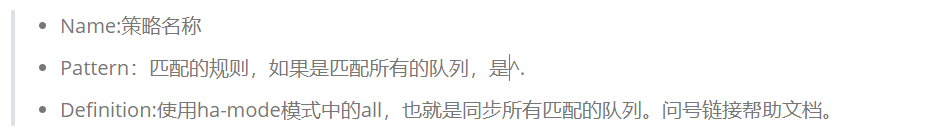
## 三、镜像队列

上面已经完成RabbitMQ默认集群模式，但并不保证队列的高可用性，尽管交换机、绑定这些可以复制到集群里的任何一个节点，但是队列内容不会复制。虽然该模式解决一项目组节点压力，但队列节点宕机直接导致该队列无法应用，只能等待重启，所以要想在队列节点宕机或故障也能正常应用，就要复制队列内容到集群里的每个节点，必须要创建镜像队列。

镜像队列是基于普通的集群模式的，然后再添加一些策略，所以你还是得先配置普通集群，然后才能设置镜像队列，我们就以上面的集群接着做。

我们可以在管控台设置镜像队列：Admin——Policies





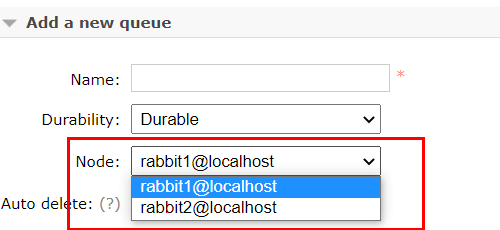
然后直接点击：Add policy，添加这个策略

然后我们会发现两个节点rabbit1和rabbit2的管控台都会有这个策略

现在我们在rabbit1节点中创建一个队列，取名为aa，往里面发布一条消息。

然后再rabbit2节点中创建一个队列，取名为bb。

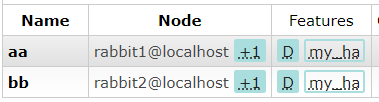
Tips：



创建队列时，在node这一栏，可以选在将队列创建在哪个节点。

我们观察：

rabbit1的管控台：



上面的“+1”，表示同步到另1个节点中去，如果集群中一共有5个节点的话，那么这里就会是“+4”。

另外，如果我们将鼠标放在aa队列的“+1”上面，就会显示：

Synchronized mirrors：rabbit2@localhost

表示将该队列同步镜像到rabbit2这个节点中去

同理，如果将鼠标放在bb队列的“+1”上面，就会显示：

Synchronized mirrors：rabbit1@localhost

这样一来，即使rabbit1这个节点宕机了，在rabbit2节点中，消息依然会存在。

这样就解决了消息同步的问题。

但是，虽然消息同步的问题解决了，还存在另一个问题：

就是Java应该访问哪个端口的问题。

我们在启动节点rabbit1和rabbit2的时候，分别设置的Java访问的端口为5673和5674。那么Java应该访问哪个端口呢？

其实访问哪个端口都一样，都能访问，并且都存在同一个问题。比如说我们访问5673端口，万一5673这个端口对应的节点rabbit1宕机了，那么Java就访问不到这个rabbit集群了。

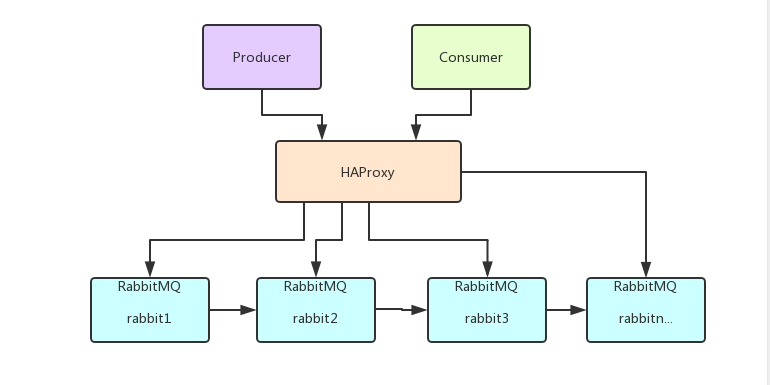
要解决这个问题，就需要用到负载均衡的思想。我们应该是访问服务，而不是访问端口。通过访问RabbitMQ服务，然后给我们分配一个rabbit节点实例。

就跟之前微服务中学习路由时差不多。路由到服务，而不是路由到实例。

## 四、负载均衡-HAProxy

HAProxy提供高可用性、负载均衡以及基于TCP和HTTP应用的代理，支持虚拟主机，它是免费、快速并且可靠的一种解决方案,包括Twitter，Reddit，StackOverflow，GitHub在内的多家知名互联网公司在使用。HAProxy实现了一种事件驱动、单一进程模型，此模型支持非常大的并发连接数。

图解：

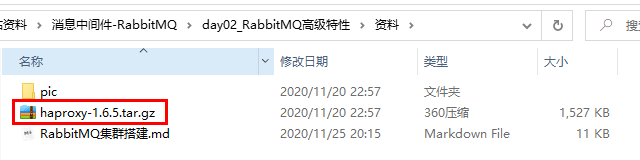


### 1、安装HAProxy

下载依赖包

yum install gcc vim wget

上传haproxy源码包



上图中的haproxy-1.6.5.tar.gz就是源码包，比如我们就上传到

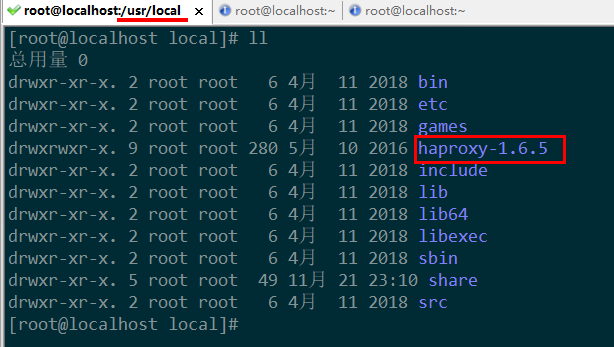
/home/test/rabbit/目录下

然后进入该源码包所在的目录

cd /home/test/rabbit/

解压到/usr/local/目录下：

tar -zxvf haproxy-1.6.5.tar.gz -C /usr/local/



进入/usr/local/haproxy-1.6.5/目录，进行编译、安装：

make TARGET=linux31 PREFIX=/usr/local/haproxy

make install PREFIX=/usr/local/haproxy

mkdir /etc/haproxy

//赋权

groupadd -r -g 149 haproxy

useradd -g haproxy -r -s /sbin/nologin -u 149 haproxy

//创建并编辑haproxy配置文件(以为该文件不存在，所以vim成了创建+编辑)

vim /etc/haproxy/haproxy.cfg

### 2、配置HAProxy

*配置文件路径：/etc/haproxy/haproxy.cfg*

配置文件内容：

我们需要注意的内容如下：

第一个：Java访问的端口

listen rabbitmq\_cluster

bind 0.0.0.0:5672

表示该集群绑定的端口，以后我们用Java访问虚拟机的5672端口，就能访问到虚拟机上的rabbitmq集群了。

第二个：负载均衡

balance roundrobin

server node1 127.0.0.1:5673 check inter 5000 rise 2 fall 2

server node2 127.0.0.1:5674 check inter 5000 rise 2 fall 2

第三个：监听统计（管控台）

listen stats

bind 192.168.176.129:8100

mode http

option httplog

stats enable

stats uri /rabbitmq-stats

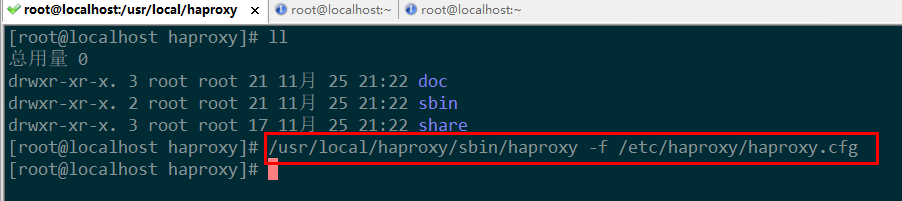
stats refresh 5s

以后在本机浏览器上访问：虚拟机IP：8100，就能监控这个集群了。

我们将这段配置内容全都复制到/etc/haproxy/haproxy.cfg中保存

然后启动HAproxy负载，-f后面是指定配置文件路径：

/usr/local/haproxy/sbin/haproxy -f /etc/haproxy/haproxy.cfg

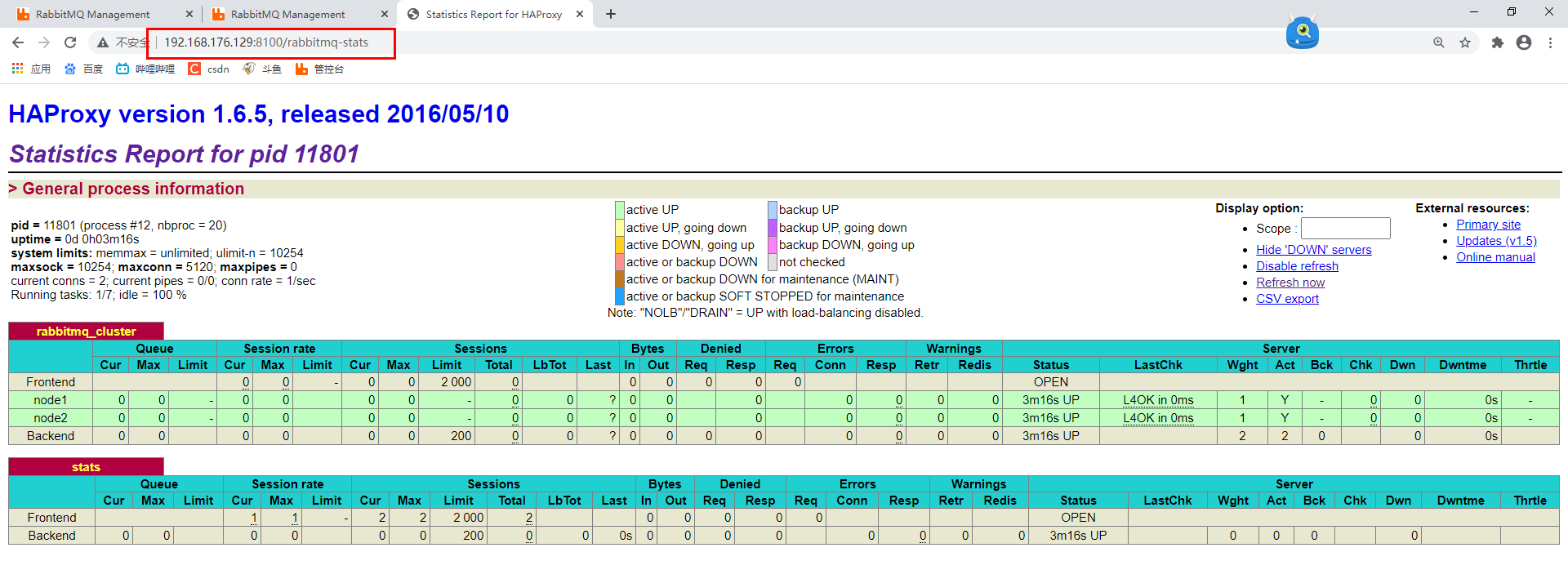


//查看haproxy进程状态

ps -ef | grep haproxy

访问如下地址对mq节点进行监控

<http://192.168.176.129:8100/rabbitmq-stats>



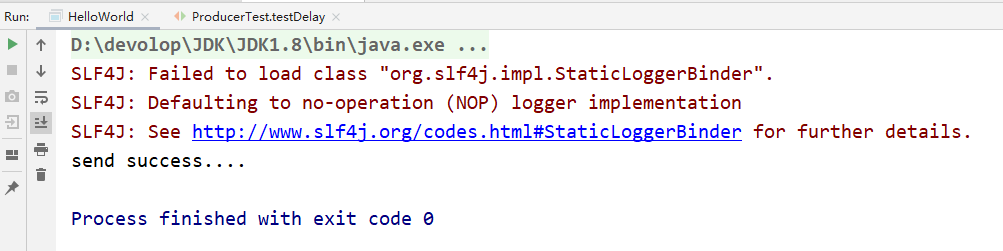
能访问到这个界面，就说明HAProxy安装成功了。

### 3、Java访问

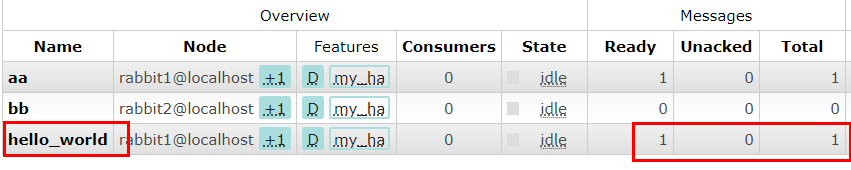
代码：



效果：



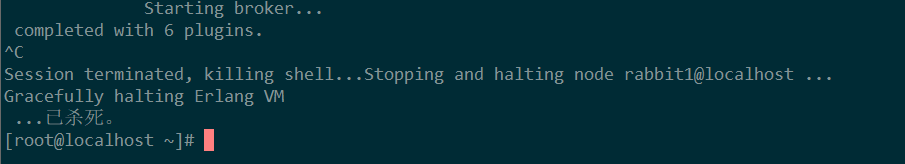
队列创建成功，消息发送成功，两个节点都能看到这个队列

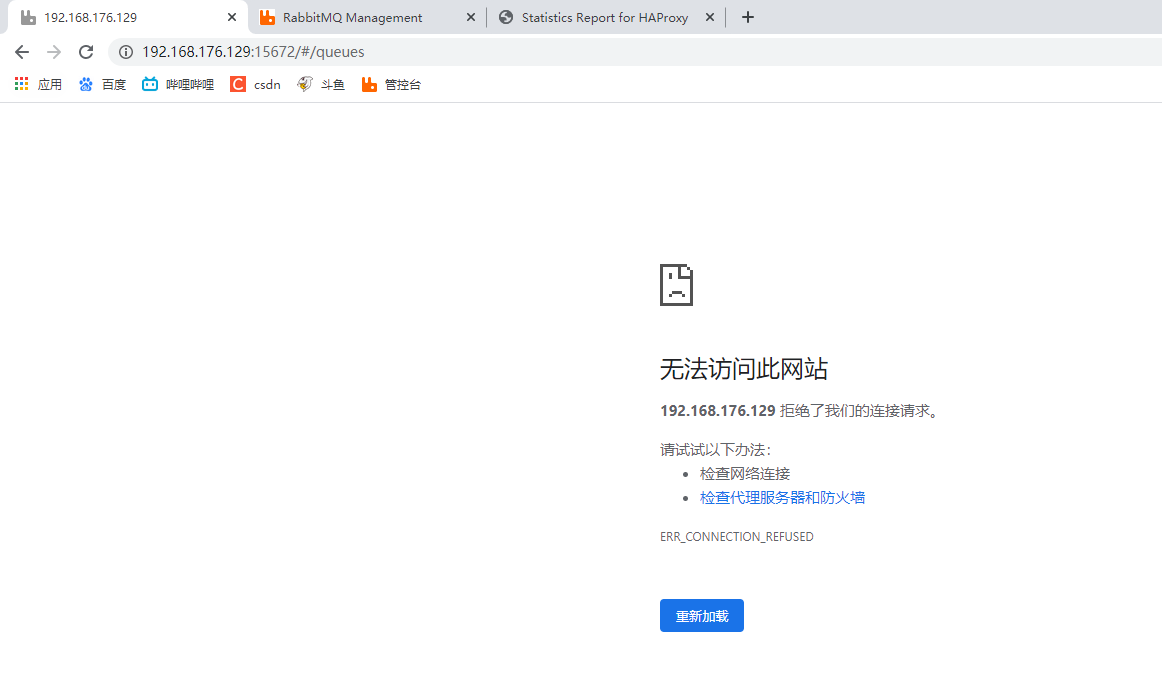




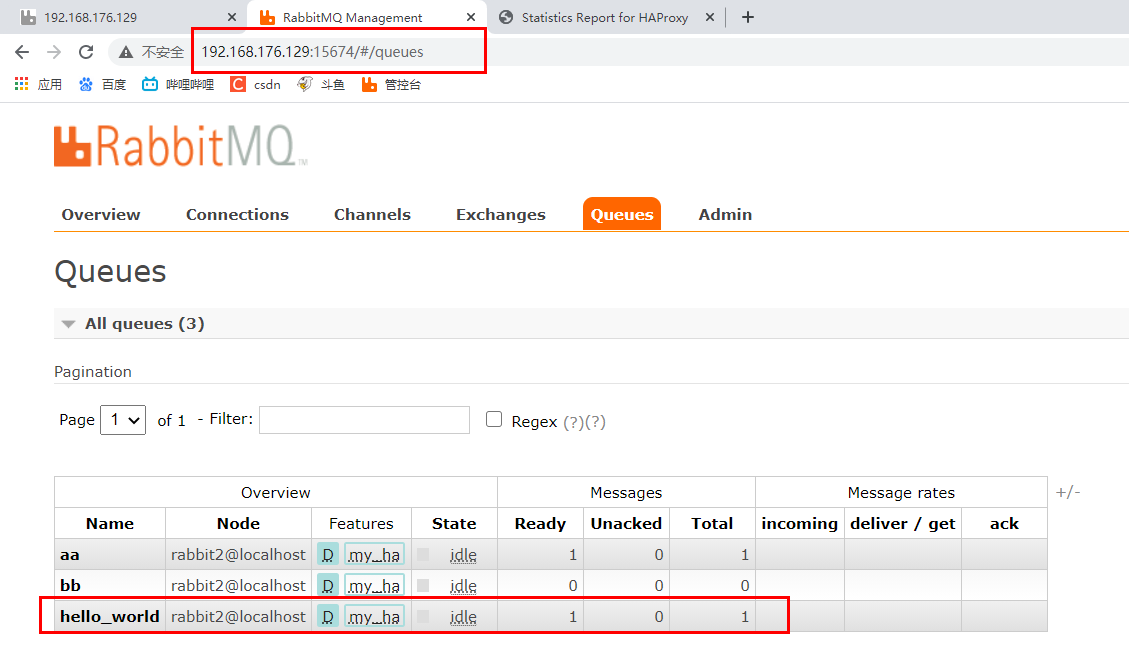
然后我们关闭任意一个节点，另一个节点中，该队列仍然存在：

比如我们关闭rabbit1的节点，在CRT中在rabbit节点的session按ctrl+C即可关闭





然后我们看rabbit2节点：



可以看到，即使rabbit1节点宕机了，但是rabbit2节点中的队列、信息仍然存在。

如果要重新启动节点的话，也很简单，在前面也提到过：[二、单机多实例部署](#_二、单机多实例部署)

启动节点rabbit1

启动节点rabbit2

rabbit1节点没有指定管控台的端口号，就使用默认的端口号15672

到此，RabbitMQ就学完了！