

# 消息队列中间件-RabbitMQ

---

课程目标：

- 了解消息中间件的概念、使用场景
- 了解AMQP的协议模型和相关概念
- 可以实现RabbitMQ的Windows下安装
- 掌握RabbitMQ的生产者和消费者的代码编写。
- 理解RabbitMQ的五种工作模式
- 掌握Spring整合RabbitMQ ( Xml传统方式 )

## 1. RabbitMQ概述

---

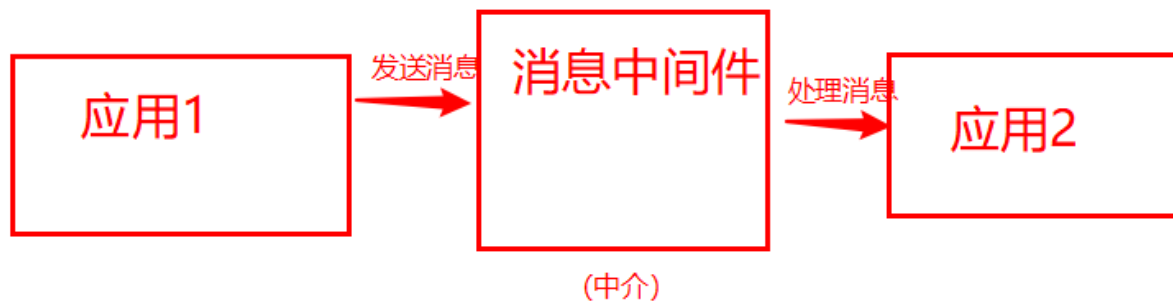
### 1.1 消息队列中间件是什么？

---

消息队列 ( Message Queue ) 是一种进程间或者线程间的异步通信方式，通常用来作为应用程序和应用程序之间的通信方法。

中间件(英语:Middleware),是提供系统软件和应用软件之间连接的软件，以便于软件各部件之间的沟通，特别是应用软件对于系统软件的集中的逻辑，在现代信息技术应用框架如Web服务、面向服务的体系结构等中应用比较广泛。如数据库、Apache的Tomcat。

本文的消息队列中间件指的是用来处理消息的消息队列服务,简称MQ **是分布式系统中非常重要的组件**



### 1.2 MQ常见的应用场景

---

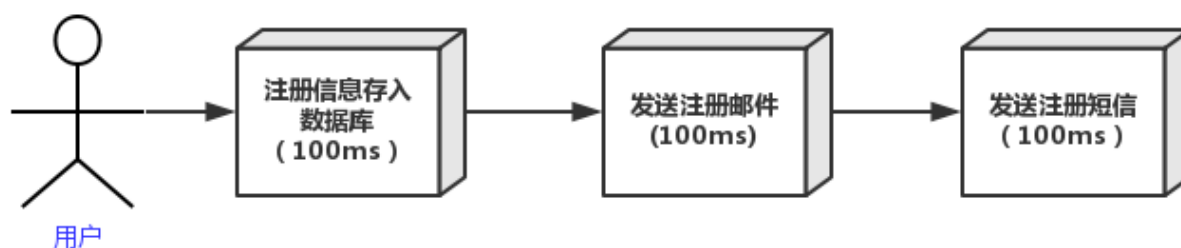
在项目中，可将一些无需即时返回且耗时的操作提取出来，进行了异步处理，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

开发中消息队列通常有如下应用场景：

以下介绍消息队列在实际应用中常用的使用场景：**异步处理，应用解耦，流量削峰和消息通知**四个场景。

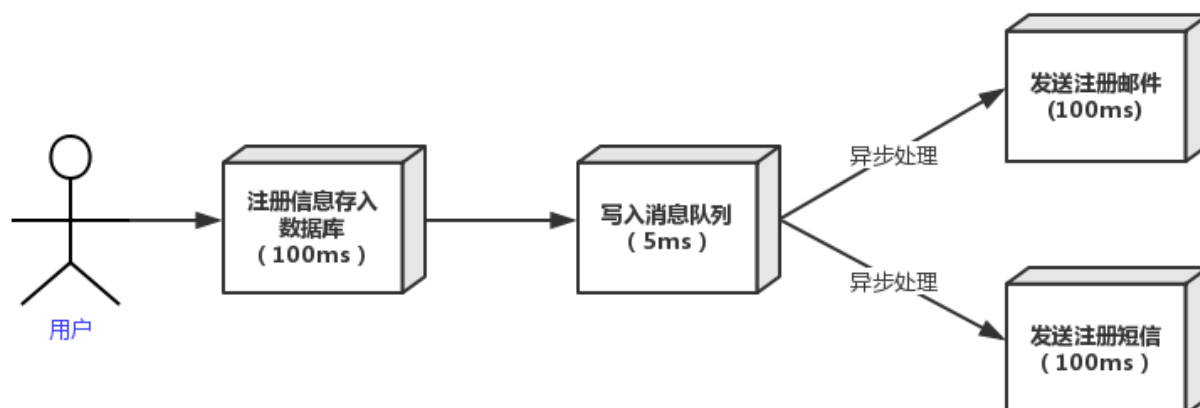
1) 异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法 同步串行方式：将注册信息持久化后，发送注册邮件，再发送注册短信。三个业务全部完成后，返回给客户端。



假设三个业务节点每个使用100毫秒钟，不考虑其他开销，则串行方式的时间是300ms，并行的时间可能是200毫秒。则串行的方式1秒内可处理3次请求，并行方式1秒内可处理5次请求，综上所述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？

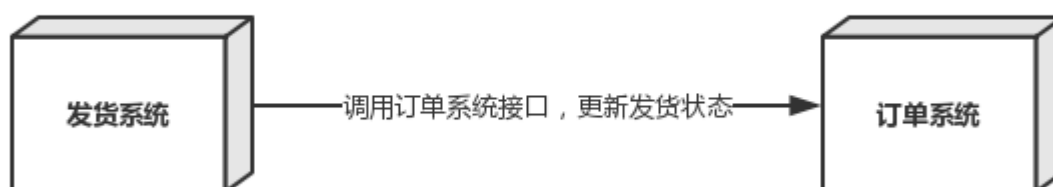
引入消息队列，将不是必须的业务逻辑，异步处理。如下图所示



按照上图，用户的响应时间相当于是注册信息写入数据库的时间和将消息插入消息队列，也就是105毫秒。注册邮件，发送短信消息写入队列后，直接返回。如此消息队列异步处理后，1秒内可处理9次请求。大大提高了系统的性能。简单的说，异步消息就是将不需要同步处理的、并且耗时长的操作由消息队列通知消息接收方进行异步处理。提高了应用程序的响应时间。

## 2) 应用解耦

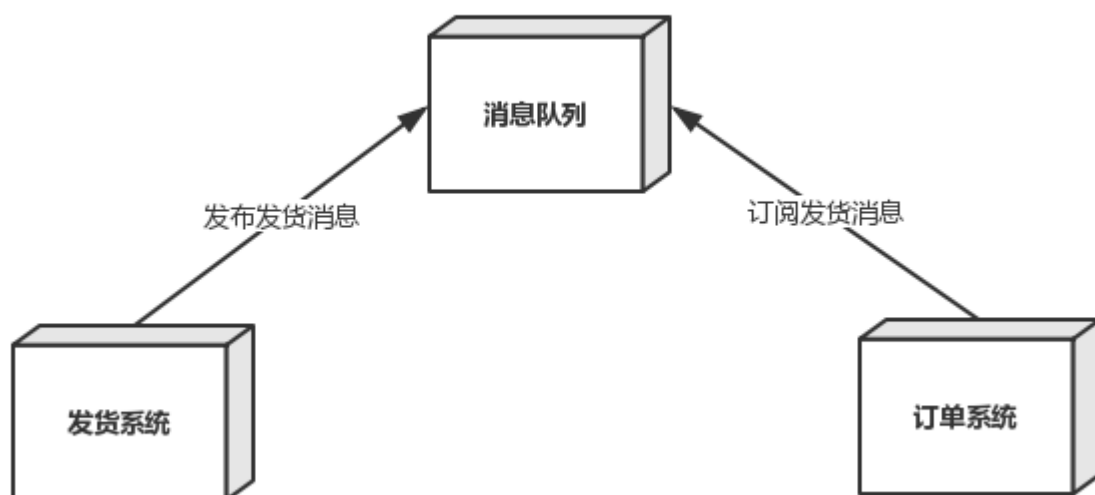
场景说明：后台发货系统，发货后快递发货系统需要通知订单系统，该订单已发货。如果我们用传统的做法是，快递发货系统调用订单系统的接口，更新订单为已发货。如下图



传统模式的缺点：

- 1) 假如订单系统无法访问，则订单更新为已发货失败，从而导致发货失败；
- 2) 发货系统与订单系统耦合；

如何解决以上问题呢？引入应用消息队列后的方案，如下图：



发货系统：发货后，发货系统完成持久化处理，将消息写入消息队列，返回发货成功。

订单系统：订阅发货的消息，获取发货信息，订单系统根据信息，进行更新操作。

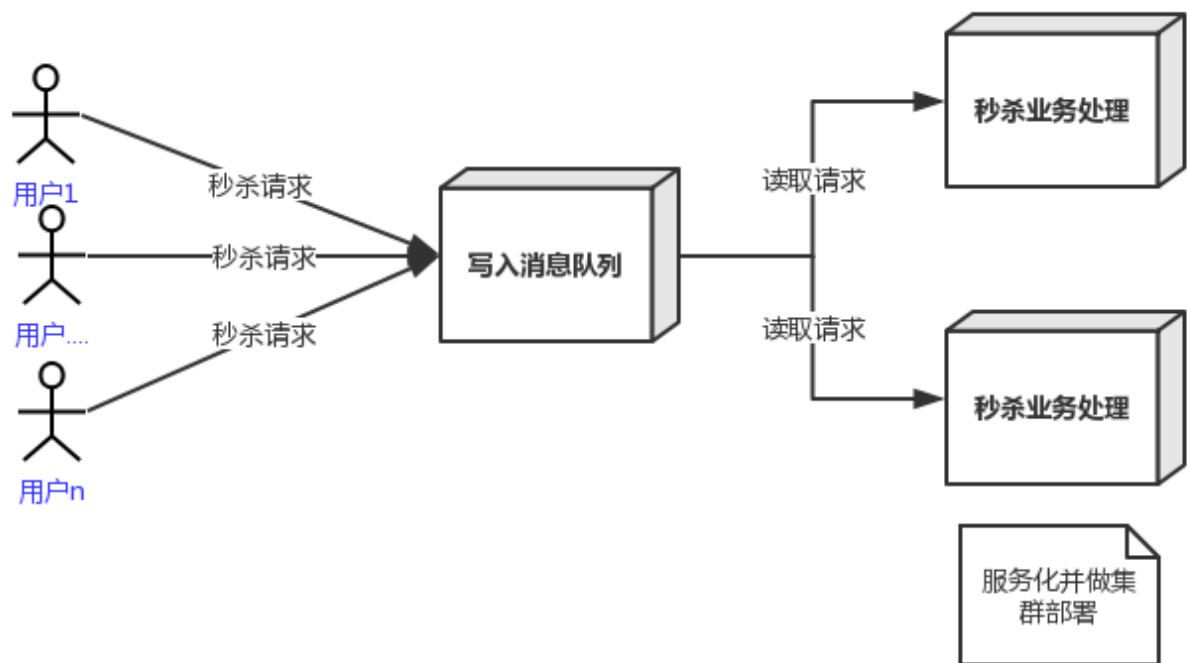
如上，发货系统在发货的时候不用关心后续操作了，如果订单系统不能正常使用。也不影响正常发货，实现订单系统与发货系统的应用解耦。

简单的说，MQ相当于一个中介，它将应用程序（服务）进行解耦合，提高系统的可靠性以及可扩展性。

### 3) 流量削峰

流量削峰也是消息队列中的常用场景，一般在秒杀或抢购活动中使用广泛。

应用场景：秒杀活动，一般会因为流量过大，应用系统配置承载不了这股瞬间流量，导致系统直接挂掉，即传说中的“宕机”现象。为解决这个问题，我们会将那股巨大的流量拒在系统的上层，即将其转移至 MQ 而不直接涌入我们的接口，此时MQ起到了缓冲作用。



简单的说，消息中间件可以临时缓存要处理的任务消息，该缓冲有助于控制和优化数据流经过系统的速度。

消息队列就像“水库”一样，拦蓄上游的洪水，削减进入下游河道的洪峰流量，从而达到减免洪水灾害的目的。

提示：MQ只是流量削峰的一种方式而已。

#### 4) 消息通讯（通知）

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在实现消息通讯。如点对点消息队列，或者聊天室等。点对点通讯：客户端A和客户端B使用同一队列，进行消息通讯。



聊天室通讯：客户端A，客户端B 订阅同一主题，进行消息发布和接收。



简单的说，MQ可以作为数据的中转站。

小结：消息队列中间件是分布式系统中重要的组件，主要解决**异步消息**，**应用耦合**，**流量削锋**，**消息通讯**等问题实现高性能，高可用，可伸缩和最终一致性[架构-分布式事务]。

## 1.3 MQ常见消息通信协议

消息通信协议是用于实现消息队列功能时所涉及的协议

常见的消息通信协议: JMS、AMQP、MQTT

### JMS (Java平台上的专业技术规范) 锁定

JMS即Java消息服务（Java Message Service）应用程序接口，是一个Java平台中关于面向**消息中间件**（MOM）的API，用于在两个应用程序之间，或**分布式系统**中发送消息，进行**异步通信**。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

JMS是一种与厂商无关的API，用来访问收发系统消息，它类似于JDBC(Java Database Connectivity)。这里，JDBC是可以用来访问许多不同关系数据库的API，而JMS则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商都支持JMS，包括IBM的MQSeries、BEA的Weblogic JMS service和Progress的SonicMQ。JMS使您能够通过消息收发服务（有时称为消息中介程序或路由器）从一个JMS客户机向另一个JMS客户机发送消息。消息是JMS中的一种类型对象，由两部分组成：报头和消息主体。报头由路由信息以及有关该消息的元数据组成。消息主体则携带着应用程序的数据或有效负载。根据有效负载的类型来划分，可以将消息分为几种类型，它们分别携带：简单文本(TextMessage)、可序列化的对象(ObjectMessage)、属性集合(MapMessage)、字节流(BytesMessage)、原始值流(StreamMessage)，还有无有效负载的消息(Message)。

### AMQP

 编辑

 讨论

 上传视频

AMQP，即Advanced Message Queuing Protocol，一个提供统一消息服务的应用层标准高级**消息**队列协议，是**应用层**协议的一个开放标准，为面向消息的中间件设计。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。**Erlang**中的实现有**RabbitMQ**等。

中文名	高级消息队列协议	应用领域	计算机
外文名	Advanced Message Queuing Protocol	特点	可传递消息，不受不同开发语言等条件的限制
属性	应用层标准协议	目标	实现一种在全行业广泛使用的标准消息中间件技术

# MQTT

- ✎ 编辑

💬 讨论

📺 上传视频

本词条由“[科普中国](#)”科学百科词条编写与应用工作项目 审核。

**MQTT**(~~消息队列遥测传输~~)是ISO 标准(ISO/IEC PRF 20922)下基于发布/订阅范式的消息协议。它工作在 [TCP/IP协议族](#)上，是为硬件性能低下的远程设备以及网络状况糟糕的情况下而设计的发布/订阅型消息协议，为此，它需要一个[消息中间件](#)。

MQTT是一个基于客户端-服务器的消息发布/订阅传输协议。MQTT协议是轻量、简单、开放和易于实现的，这些特点使它适用范围非常广泛。在很多情况下，包括受限的环境中，如：机器与机器（M2M）通信和物联网（IoT）。其在，通过卫星链路通信传感器、偶尔拨号的医疗设备、智能家居、及一些小型化设备中已广泛使用。

小结：

- JMS是定义了统一的接口，来对消息操作进行统一；AMQP是通过规定协议来统一数据交互的格式
- JMS限定了必须使用Java语言；AMQP只是协议，不规定实现方式，因此是跨语言的。
- JMS规定了两种消息模型；而AMQP的消息模型更加丰富
- MQTT 主要应用在物联网

## 1.4 MQ的相关产品

市场上常见的MQ产品有：Kafka、ActiveMQ、RabbitMQ、RocketMQ，关于他们之间的对比也有各种各样的讨论，下面我们看看关于它们的总结：

特性	ActiveMQ	RabbitMQ	RocketMQ	kafka
开发语言	java	erlang	java	scala
单机吞吐量	万级	万级	10万级	10万级
时效性	ms级	us级	ms级	ms级以内
可用性	高(主从架构)	高(主从架构)	非常高(分布式架构)	非常高(分布式架构)
功能特性	成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好	基于erlang开发，所以并发能力很强，性能极其好，延时很低；管理界面较丰富	MQ功能比较完备，扩展性佳	只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。

(1)ActiveMQ版本更新频率较低，中小型软件公司，建议选RabbitMQ。一方面，erlang语言天生具备高并发的特性，而且他的管理界面用起来十分方便。正所谓，成也萧何，败也萧何！他的弊端也在这里，虽然RabbitMQ是开源的，然而国内有几个能定制化开发erlang的程序员呢？所幸，RabbitMQ的社区十分活跃，可以解决开发过程中遇到的bug，这点对于中小型公司来说十分重要。不考虑rocketmq和kafka的原因是，一方面中小型软件公司不如互联网公司，数据量没那么大，选消息中间件，应首选功能比较完备的，所以kafka排除。不考虑rocketmq的原因是，

rocketmq是阿里出品，如果阿里放弃维护rocketmq，中小型公司一般抽不出人来进行rocketmq的定制化开发，因此不推荐。

(2)大型软件公司，根据具体使用在rocketMQ和kafka之间二选一。一方面，大型软件公司，具备足够的资金搭建分布式环境，也具备足够大的数据量。针对rocketMQ,大型软件公司也可以抽出人手对rocketMQ进行定制化开发，毕竟国内有能力改JAVA源码的人，还是相当多的。至于kafka，根据业务场景选择，如果有日志采集功能，肯定是首选kafka了。具体该选哪个，看使用场景。

小结:

- ActiveMQ：基于JMS，Apache
- RabbitMQ：基于AMQP协议，erlang语言开发，稳定性好
- RocketMQ：基于JMS，阿里巴巴产品，目前交由Apache基金会
- Kafka：分布式消息系统，高吞吐量(自定义基于TCP协议)

## 1.5 什么是RabbitMQ

RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写。

AMQP（Advanced Message Queue Protocol）：高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。就像SMTP、HTTP等协议一样，它是一个线路层协议规范，而不是API规范(例如JMS)，因此它天然就是跨平台的，只要开发者按照规范的格式发送数据，任何平台都可以通过AMQP进行消息交互。基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。像目前流行的StormMQ、RabbitMQ等都实现了AMQP。

Erlang是一种通用的面向并发的编程语言，它由瑞典电信设备制造商爱立信所辖的CS-Lab开发，目的是创造一种可以应对大规模并发活动的编程语言和运行环境。

RabbitMQ支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

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

### RabbitMQ 的主要特点（课后阅读）：

#### 1.可靠性（Reliability）

RabbitMQ使用一些机制来保证可靠性，如持久化、传输确认、发布确认。

#### 2.灵活的路由（Flexible Routing）

在消息进入队列之前，通过Exchange来路由消息的。对于典型的路由功能，RabbitMQ已经提供了一些内置的Exchange来实现。针对更复杂的路由功能，可以将多个Exchange绑定在一起，也通过插件机制实现自己的Exchange。换句话说：路由主要用于转发规则的定义和执行。

#### 3.消息集群（Clustering）

多个RabbitMQ服务器可以组成一个集群，形成一个逻辑Broker。

#### 4.高可用（Highly Available Queues）

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

#### 5.多种协议（Multi-protocol）RabbitMQ支持多种消息队列协议，比如STOMP、MQTT等等。

- 6.多语言客户端（ Many Clients ） RabbitMQ 几乎支持所有常用语言，比如 Java、.NET、Ruby 等等。
- 7.管理界面（ Management UI ） RabbitMQ 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker 的许多方面。
- 8.跟踪机制（ Tracing ） 如果消息异常，RabbitMQ 提供了消息跟踪机制，使用者可以找出发生了什么。
- 9.插件机制（ Plugin System ） RabbitMQ 提供了许多插件，来从多方面进行扩展，也可以编写自己的插件。

## 2. Windows下快速单机部署

### 2.1 RabbitMQ单机安装启动

目标

按照文档在本机安装windows版本RabbitMQ，并配置其用户和Virtual Hosts

步骤

1. 安装erlang；
2. 安装RabbitMQ；
3. 开启图形化管理插件
4. 创建管理RabbitMQ的用户；
5. 创建虚拟主机Virtual Hosts并赋权

（1）下载并安装 Erlang

官网直达：<http://www.erlang.org/downloads>

配套软件中已提供 otp\_win64\_21.0.1.exe （以管理员身份运行安装！注册表会写入信息）

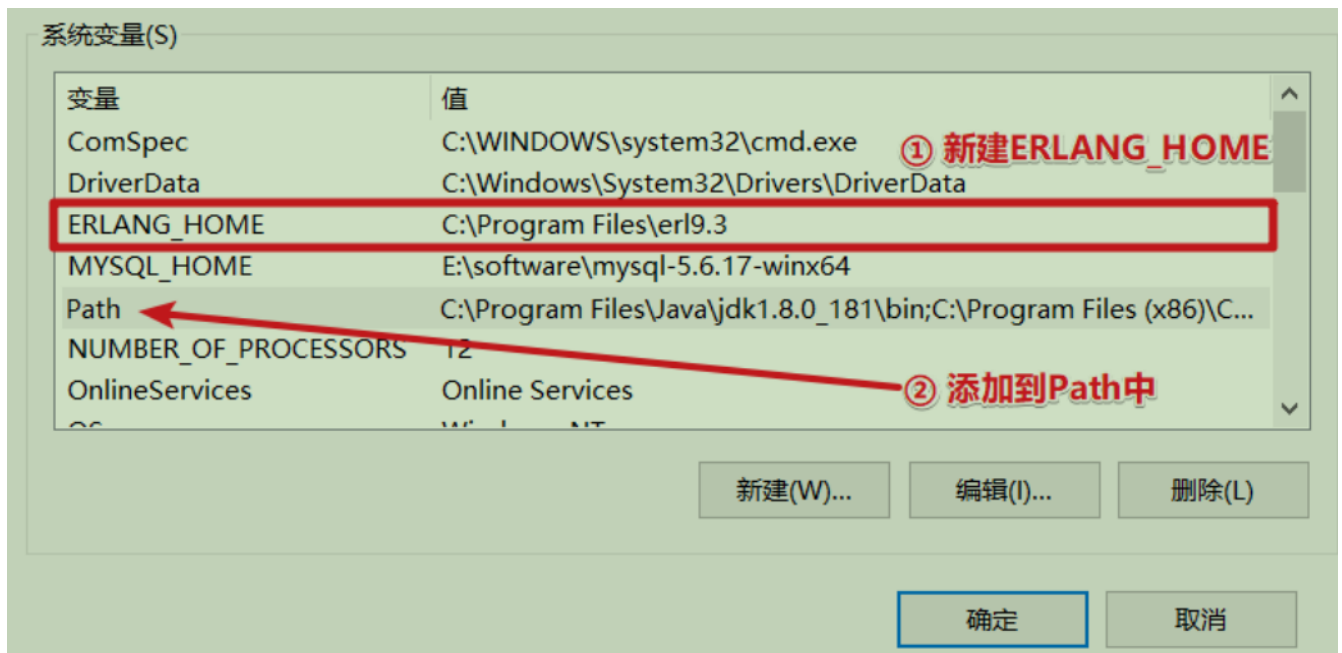
我这里安装到了 C:\11\_SR\erl\erl10.0.1



安装后，系统变量中会有：

- ERLANG\_HOME：C:\11\_SR\erl10.0.1 （安装后自动添加的，若没有则需手动添加）
- Path：%ERLANG\_HOME%\bin （若要编程使用，则需手动添加，否则可省略）





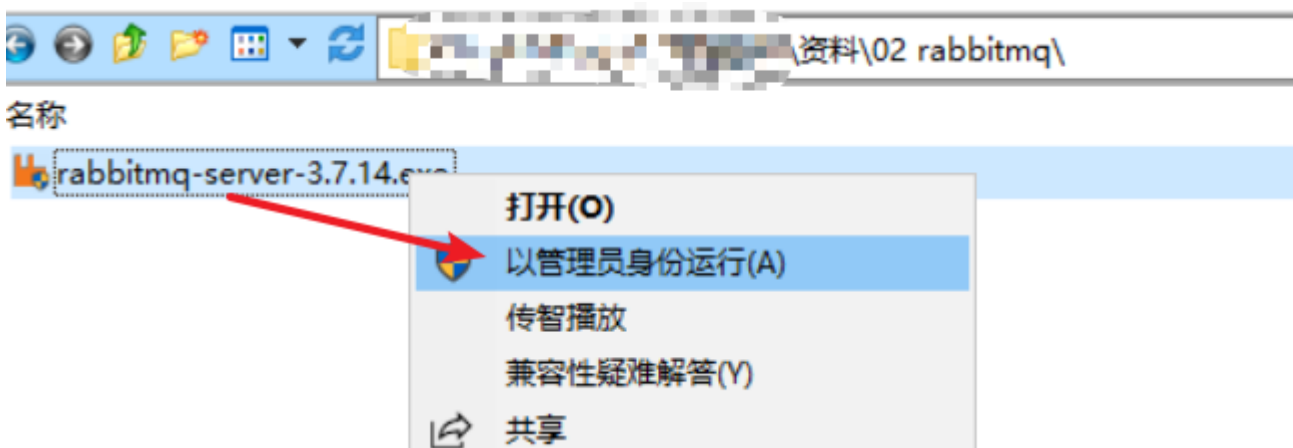
(2) 下载并安装rabbitmq

官网直达：<http://www.rabbitmq.com/download.html>

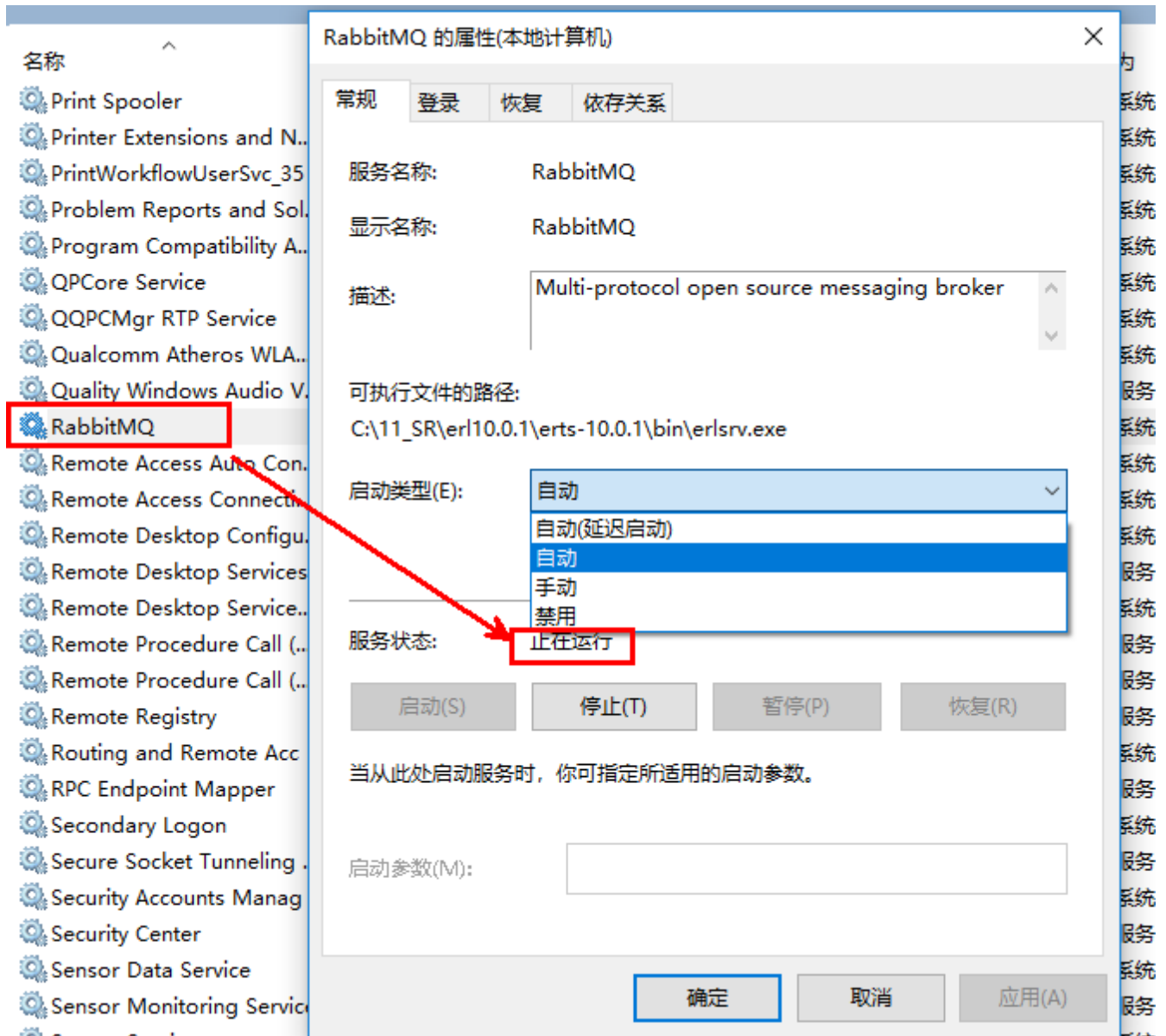
配套软件中已提供 rabbitmq-server-3.7.7.exe。

双击安装，管理员身份运行（会添加系统服务），注意不要安装在包含中文和空格的目录下！

我这里安装到了 C:\02\_Server\MQServer\RabbitMQ Server



安装后window服务中就存在rabbitMQ了，并且是启动状态。



### (3) 管理插件的开启 (图形化管理)

打开cmd，先进入rabbitMQ安装目录的sbin目录，输入命令：

```
rabbitmq-plugins enable rabbitmq_management
```

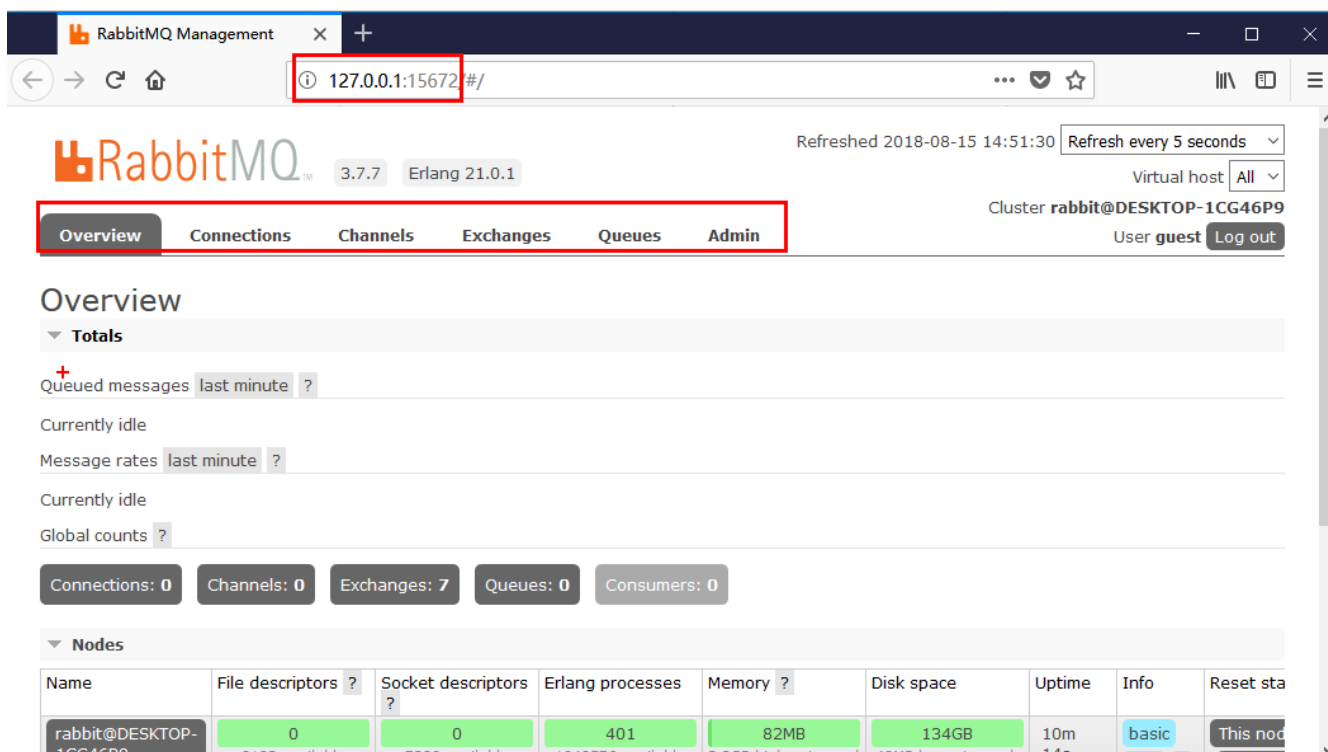
结果：

```
命令提示符
Microsoft Windows [版本 10.0.17134.885]
(c) 2018 Microsoft Corporation. 保留所有权利。

C:\Users\61073>cd C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.14\sbin ← ④ 进入sbin目录
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.14\sbin>rabbitmq-plugins.bat enable rabbitmq_management ← ⑤ 执行命令
Enabling plugins on node rabbit@LAPTOP-94B58GLJ:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@LAPTOP-94B58GLJ...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
started 3 plugins.

C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.14\sbin>
```

(5) 打开浏览器，地址栏输入<http://127.0.0.1:15672>，即可看到管理界面的登陆页，输入用户名和密码（默认都为 guest），进入主界面。



最上侧的导航以此是：概览、连接、信道、交换器、队列、用户管理

提示：

15672是管理插件的web端口

创建新账号：

guest是内置默认的超管的账号，将来应用客户端连接时，不建议使用该账号（测试无所谓），下面手动创建新的RabbitMQ的用户（bobo/bobo），并给用户赋予管理员角色（Admin或Management）：

RabbitMQ 3.7.7 Erlang 21.0.1 Refreshed 2020-02-18 22:08:38 Refresh every 5 seconds Virtual host All Cluster rabbit@DESKTOP-1CG46P9 User guest Log out

Overview Connections Channels Exchanges Queues Admin ①

## Users ②

▼ All users

Filter:  ☐ Regex ? 2 items, page size up to 100

Name	Tags	Can access virtual hosts	Has password
bobo	administrator	No access	•
guest	administrator	/, /bobohost	•

⑥显示用户列表

▼ Add a user ③输入用户名和密码

Username: bobo \*

Password:   (confirm) \*

Tags: administrator ?

④点击Admin, 设置角色为管理员

Set Admin Monitoring Polycmaker Management Impersonator None

Add user ⑤点击添加

创建虚拟主机Virtual Hosts :

像mysql拥有数据库的概念并且可以指定用户对库和表等操作的权限。RabbitMQ也有类似的权限管理；在RabbitMQ中可以虚拟消息服务器Virtual Host，每个Virtual Hosts相当于一个相对独立的RabbitMQ服务器，每个Virtual Host之间是相互隔离的。exchange、queue、message不能互通。相当于mysql的db。Virtual Name一般以/开头。

新建虚拟主机"/bobohost"，注意前面的"/"：

RabbitMQ 3.7.7 Erlang 21.0.1 Refreshed 2020-02-18 21:46:48 Refresh every 5 seconds Virtual host All Cluster rabbit@DESKTOP-1CG46P9 User guest Log out

Overview Connections Channels Exchanges Queues Admin ①

## Virtual Hosts ②

▼ All virtual hosts

Filter:  ☐ Regex ? 2 items, page size up to 100

Overview			Messages			Network		Message rates		
Name	Users ?	State	Ready	Unacked	Total	From client	To client	publish	deliver / get	+/-
/ ⑤	guest	running	NaN	NaN	NaN					
/bobohost	guest	running	NaN	NaN	NaN					

④输入虚拟主机的名字

▼ Add a new virtual host

Name: /bobohost

Add virtual host ④

设置Virtual Hosts权限，选择需要授权访问用户：

Permissions

Current permissions

User	Configure regexp	Write regexp	Read regexp	
bobo	.*	.*	.*	Clear
guest	.*	.*	.*	Clear

Set permission

User bobo ①选择用户

Configure regexp: .\*

Write regexp: .\*

Read regexp: .\*

Set permission ②点击设置

③显示设置好的权限

重新登录：

localhost:15672/#/vhosts/%2Fbobohost

RabbitMQ

Username: bobo

Password: \*\*\*\*

Login

localhost:15672/#/users

RabbitMQ 3.7.7 Erlang 21.0.1

Refreshed 2020-02-18 22:14:14 Refresh every 5 seconds

Virtual host /bobohost

Cluster rabbit@DESKTOP-1CG46P9

User bobo Log out

Overview Connections Channels Exchanges Queues Admin

小结：

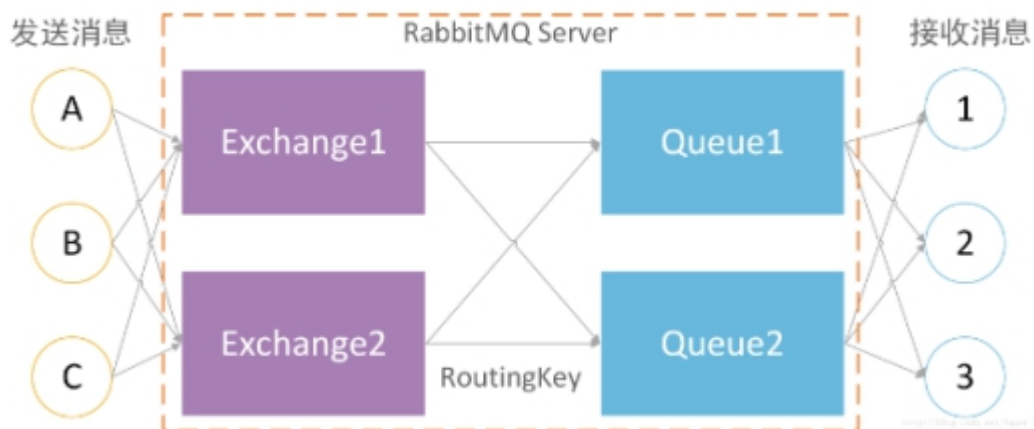
安装流程如下：



注意：安装erlang和RabbitMQ的时候需要使用 管理员身份安装。

## 2.2 AMQP协议模型和相关概念

AMQP协议是一个高级抽象层消息通信协议，RabbitMQ是AMQP协议的实现。架构模型如下：



主要组件的相关概念如下：

- RabbitMQ Server：也叫Broker Server，它是一种传输服务。他的角色就是维护一条从Producer到Consumer的路线，保证数据能够按照指定的方式进行传输。
- Producer：消息生产者，如图A、B、C，数据的发送方。消息生产者连接RabbitMQ服务器然后将消息投递到Exchange。
- Consumer：消息消费者，如图1、2、3，数据的接收方。消息消费者订阅队列，RabbitMQ将Queue中的消息发送到消息消费者。
- Exchange：生产者将消息发送到Exchange（交换器），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。Exchange并不存储消息。RabbitMQ中的Exchange有direct、fanout、topic、headers四种类型，每种类型对应不同的路由规则。
- Queue：（队列）是RabbitMQ的内部对象，用于存储消息。消息消费者就是通过订阅队列来获取消息的，rabbitMQ中的消息都只能存储在Queue中，生产者生产消息并最终投递到Queue中，消费者可以从Queue中获取消息并消费。多个消费者可以订阅同一个Queue，这时Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。
- RoutingKey：生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。在Exchange Type与binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。RabbitMQ为routing key设定的长度限制为255bytes。
- Connection：（连接）：Producer和Consumer都是通过TCP连接到RabbitMQ Server的。以后我们可以看到，程序的起始处就是建立这个TCP连接。
- Channels：（信道）：它建立在上述的TCP连接中。数据流动都是在Channel中进行的。也就是说，一般情况是程序起始建立TCP连接，第二步就是建立这个Channel。
- VirtualHost：权限控制的基本单位，一个VirtualHost里面有若干Exchange和MessageQueue，以及指定被哪些user使用

## 3. 快速入门

### 3.1 模式说明

RabbitMQ官方建议和提供了多种使用模式（<https://www.rabbitmq.com/getstarted.html>），不同工作模式主要是由Exchange的类型来控制 and 决定的，后面专题讲解，我们这里首先使用最简单的模式：“Hello World”，即简单模式。

# 1 "Hello World!"

The simplest thing that does  
*something*



上图的P是指Producer，即生产者；C指的是Consumer，即消费者。中间红色的方形是Queue，即队列。

生产者是用来发送消息的；消费者是等待接收消息的。

队列只受主机的内存和磁盘限制的约束，它本质上是一个大的消息缓冲区，队列在使用时必须指定一个名字。

许多生产者可以将消息发送到一个队列，而许多消费者可以尝试从一个队列接收数据。

## 3.2 测试环境准备

创建工程rabbitmqdemo，类型为Maven项目，起步依赖参考如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

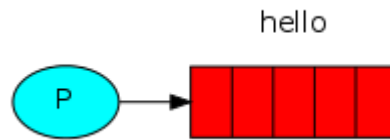
    <groupId>cn.itcast.demo</groupId>
    <artifactId>rabbitmqdemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <!--RabbitMQ客户端-->
        <dependency>
            <groupId>com.rabbitmq</groupId>
            <artifactId>amqp-client</artifactId>
            <version>5.6.0</version>
        </dependency>
    </dependencies>
</project>
```

## 3.3 生产者

## 目标

编写消息生产者代码，发送消息到队列



生产者发送消息到RabbitMQ的某个队列，消费者从队列中获取消息。可以使用rabbitMQ的简单模式（simple）；生产者发送消息步骤：

1. 创建连接工厂（设置连接相关参数）；
2. 创建连接；
3. 创建频道；
4. 声明队列；
5. 发送消息；
6. 关闭资源；

cn.itcast.rabbitmq.helloworld.Producer

```
package cn.itcast.rabbitmq.helloworld;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

/**
 * 生产者：发送消息到MQ。
 * 模式：一个生产者、一个消费者，不需要设置交换机
 */
public class Producer {

    //队列名字
    static final String QUEUE_NAME = "hello";

    //入口方法
    public static void main(String[] args) throws Exception {
        // 1. 创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //设置连接参数 - 主机地址;默认为 localhost
        connectionFactory.setHost("localhost");
        //设置连接参数 - 连接端口;默认为 5672
        connectionFactory.setPort(5672);
        //设置连接参数 - 虚拟主机名称;默认为 /
        connectionFactory.setVirtualHost("/bobohost");
        //设置连接参数 - 连接用户名;默认为guest
        connectionFactory.setUsername("bobo");
        //设置连接参数 - 连接密码;默认为guest
        connectionFactory.setPassword("bobo");
        // 2. 创建连接
        Connection connection = connectionFactory.newConnection();
```



```

// 3. 创建频道
Channel channel = connection.createChannel();
/**
 * 4. 声明 (创建) 队列
 * 参数1: 队列名称
 * 参数2: 是否定义持久化队列
 * 参数3: 是否独占本次连接
 * 参数4: 是否在不使用的时候自动删除队列
 * 参数5: 队列其它参数
 * 更多说明: 声明的队列是幂等的, 它只在不存在时才被创建。
 */
channel.queueDeclare(QUEUE_NAME, true, false, false, null);
// 5. 发送消息
// 5.1 要发送的信息
String message = "Hello world!";
/**
 * 5.2 发布消息
 * 参数1: 交换机名称, 如果没有指定则使用默认Default Exchange
 * 参数2: 路由key, 简单模式可以传递队列名称
 * 参数3: 消息其它属性
 * 参数4: 消息内容。消息内容是字节数组, 因此您可以在那里编码任何您喜欢的内容。
 */
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
//控制台显示发送的消息 (测试用)
System.out.println("已发送消息:" + message);
// 6. 关闭释放资源
channel.close();
connection.close();
}

}

```

提示：

在设置连接工厂的时候，如果没有指定连接的一系列参数的话；那么会有默认值。主要去设置虚拟主机。

#### 【API说明】

##### 1) 声明队列

```

/**
 * Declare a queue
 * @see com.rabbitmq.client.AMQP.Queue.Declare
 * @see com.rabbitmq.client.AMQP.Queue.DeclareOk
 * @param queue the name of the queue
 * @param durable true if we are declaring a durable queue (the queue will survive a
server restart)
 * @param exclusive true if we are declaring an exclusive queue (restricted to this
connection)
 * @param autoDelete true if we are declaring an autodelete queue (server will delete
it when no longer in use)
 * @param arguments other properties (construction arguments) for the queue
 * @return a declaration-confirm method to indicate the queue was successfully declared

```

```

    * @throws java.io.IOException if an error is encountered
    */
    Queue.DeclareOk queueDeclare(String queue, boolean durable, boolean exclusive, boolean
autoDelete,
                                Map<String, Object> arguments) throws IOException;

```

参数说明：

- queue：队列的名称。
- durable：是否持久化，即服务器重启后，数据是否还能存活。
  - 值为false时，数据存放到内存中，重启服务会丢失。
  - 值为true时，数据存放到自带的Mnesia数据库中，重启服务后会自动读取该数据库中的数据。
- exclusive：是否排外，即是否为当前连接的专用队列，是否私有，是否允许多个消费者都访问同一个队列。
  - 值为false时，非排外，公有队列，允许多个消费者使用同一个队列。
  - 值为true时，排外，私有队列，当前队列会加锁，其他通道的消费者无法访问该队列
- autoDelete：是否自动删除，即当没有任何消费者使用时，是否自动删除该队列。
  - 值为false时，所有的消费者都断开连接后仍然保留该队列。
  - 值为true时，当最后一个消费者断开连接之后，该队列会自动被删除。
- arguments：队列消息的其他策略。

## 2) 发布消息

```

channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
// @m basicPublish(String s, String s1, BasicProperties basicProperties, byte[] bytes) void
Sys @m basicPublish(String s, String s1, boolean b, BasicProperties basicProperties, byte[] bytes) void
// @m basicPublish(String s, String s1, boolean b, boolean b1, BasicProperties basicProperties, byte[] bytes) void

```

```

/**
 * Publish a message.
 *
 * Publishing to a non-existent exchange will result in a channel-level
 * protocol exception, which closes the channel.
 *
 * Invocations of <code>Channel#basicPublish</code> will eventually block if a
 * <a href="http://www.rabbitmq.com/alarms.html">resource-driven alarm</a> is in
effect.
 *
 * @see com.rabbitmq.client.AMQP.Basic.Publish
 * @see <a href="http://www.rabbitmq.com/alarms.html">Resource-driven alarms</a>.
 * @param exchange the exchange to publish the message to
 * @param routingKey the routing key
 * @param mandatory true if the 'mandatory' flag is to be set
 * @param immediate true if the 'immediate' flag is to be
 * set. Note that the RabbitMQ server does not support this flag.
 * @param props other properties for the message - routing headers etc
 * @param body the message body
 * @throws java.io.IOException if an error is encountered
 */
void basicPublish(String exchange, String routingKey, boolean mandatory, boolean
immediate, BasicProperties props, byte[] body)

```

```
throws IOException;
```

参数说明：

- exchange：交换机名称，如果为空，则使用RabbitMQ的默认交换机。默认的Exchange隐式绑定到每个队列，routingKey等于队列名称。无法与默认交换机显式绑定或解除绑定，它也不能被删除。
- routingKey：路由键，#匹配0个或多个单词，\*匹配一个单词，在topic exchange做消息转发用
- mandatory：true：如果exchange根据自身类型和消息routeKey无法找到一个符合条件的queue，那么会调用basic.return方法将消息返还给生产者。false：出现上述情形broker会直接将消息扔掉（默认）。

简单来说：mandatory标志告诉服务器至少将该消息route到一个队列中，否则将消息返还给生产者；

- immediate：true：如果exchange在将消息route到queue(s)时发现对应的queue上没有消费者，那么这条消息不会放入队列中。当与消息routeKey关联的所有queue(一个或多个)都没有消费者时，该消息会通过basic.return方法返还给生产者。

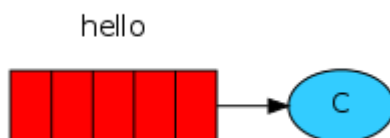
简单来说：immediate标志告诉服务器如果该消息关联的queue上有消费者，则马上将消息投递给它，如果所有queue都没有消费者，直接把消息返还给生产者，不用将消息入队列等待消费者了。

- BasicProperties：需要注意的是BasicProperties.deliveryMode，1:不持久化 2：持久化 这里指的是消息的持久化，配合channel(durable=true),queue(durable)可以实现，即使服务器宕机，消息仍然保留。该参数的值可以直接使用MessageProperties类中的常量。
- Body：要发送的消息体

## 3.4 消费者

目标

编写消息消费者代码，从队列中接收消息并消费



分析

从RabbitMQ的队列中接受消息；实现消息消费者：

1. 创建连接工厂；
2. 创建连接；
3. 创建频道；
4. 声明队列；
5. 创建消费者（接受消息并处理消息）；
6. 监听消息队列

cn.itcast.rabbitmq.helloworld.Consumer

```
package cn.itcast.rabbitmq.helloworld;
```

```

import com.rabbitmq.client.*;

import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * 消费者：从MQ获取消费消息。
 * 模式：一个生产者、一个消费者，不需要设置交换机
 */
public class Consumer {
    //队列名字
    static final String QUEUE_NAME = "hello";
    //入口方法
    public static void main(String[] args) throws Exception {
        // 1. 创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //设置连接参数 - 主机地址;默认为 localhost
        connectionFactory.setHost("localhost");
        //设置连接参数 - 连接端口;默认为 5672
        connectionFactory.setPort(5672);
        //设置连接参数 - 虚拟主机名称;默认为 /
        connectionFactory.setVirtualHost("/shhm091host");
        //设置连接参数 - 连接用户名;默认为guest
        connectionFactory.setUsername("shhm091");
        //设置连接参数 - 连接密码;默认为guest
        connectionFactory.setPassword("shhm091");
        // 2. 创建连接
        Connection connection = connectionFactory.newConnection();
        // 3. 创建频道
        Channel channel = connection.createChannel();

        //4声明队列;(经常是消费者创建的多)
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        //5创建消费者对象,(设置消息处理),用来接收(打印)消息
        // com.rabbitmq.client.Consumer consumer=new DefaultConsumer(channel);
        // DefaultConsumer consumer=new DefaultConsumer();
        DefaultConsumer consumer = new DefaultConsumer(channel){
            /**
             * consumerTag 消费者标签,在channel.basicConsume时候可以指定
             * envelope 消息包的内容,可从中获取消息id,
             * 消息routingkey,交换机,消息和重传标志(收到消息失败后是否需
             要重新发送)
             * properties 属性信息
             * body 消息
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                //路由key
                System.out.println("路由key为:" +envelope.getRoutingKey());
                //交换机

```

```

        System.out.println("交换机为：" + envelope.getExchange());
        //消息id
        System.out.println("消息id为：" + envelope.getDeliveryTag());
        //收到的消息
        String message=new String(body, "UTF-8");
        String message=new String(body, StandardCharsets.UTF_8);
        //打印消息
        System.out.println("接收到的消息为：" + message);

    }

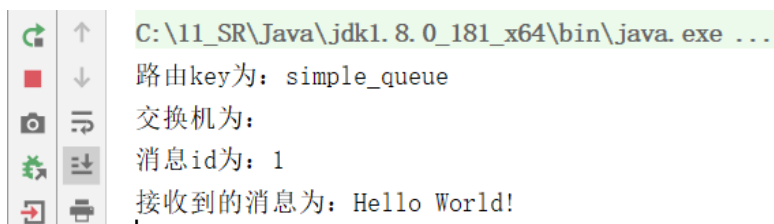
};

///消费队列中的消息，会自动开启监听。
/**
 * 参数1：队列名称
 * 参数2：是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，
 *      mq接收到回复会删除消息，设置为false则需要手动确认
 * 参数3：消息接收到后回调
 */
channel.basicConsume(QUEUE_NAME, true, consumer);
//不关闭资源，应该一直监听消息
}

}

```

执行结果：



```

C:\11_SR\Java\jdk1.8.0_181_x64\bin\java.exe ...
路由key为: simple_queue
交换机为:
消息id为: 1
接收到的消息为: Hello World!

```

注意：

在启动消息监听之后不要把资源关闭。

在消费者监听状态下，执行生产者发送消息，则可实现一边生产一边消费的效果。

### 【API说明】

```

/**
 * Start a non-nolocal, non-exclusive consumer, with
 * a server-generated consumerTag.
 * @param queue the name of the queue
 * @param autoAck true if the server should consider messages
 *      acknowledged once delivered; false if the server should expect
 *      explicit acknowledgements
 * @param callback an interface to the consumer object
 * @return the consumerTag generated by the server
 * @throws java.io.IOException if an error is encountered

```

```

* @see com.rabbitmq.client.AMQP.Basic.Consume
* @see com.rabbitmq.client.AMQP.Basic.ConsumeOk
* @see #basicConsume(String, boolean, String, boolean, boolean, Map, Consumer)
*/
String basicConsume(String queue, boolean autoAck, Consumer callback) throws
IOException;

```

参数说明：

- queue：通道或队列名字
- autoAck：是否自动确认ack，即在接收到消息后，是否自动反馈一个消息给服务器。消息确认功能，就是在消费者处理完任务之后，就给服务器一个回馈，服务器就会将该消息删除，如果消费者超时不回馈，那么服务器将就将该消息重新发送给其他消费者。
  - 值为false时，消费者接收到消息后不会自动反馈自己已经消费了该消息的情况给服务器。这种情况下，需要手动使用channel.ack、channel.nack、channel.basicReject 进行消息应答。
  - 值为true时，消费者在接收到消息后，就会自动反馈一个消息给服务器。
- consumer：消费者对象。

小结

简单模式：

1. 生产者发送消息到指定队列中，消费者从队列中接收消息；
2. 在rabbitMQ中消费者只能从队列中接收到消息。
3. 思考：如果接收消息的消费者在同一个队列中有多个时，消息是如何分配的？

## (面试点) 消息发送确认机制

**ConfirmListener** 用于确认MQ是否接收到消息

消息的确认，是指生产者投递消息后，如果 Broker 收到消息，则会给我们生产者一个应答。生产者进行接收应答，用来确定这条消息是否正常的发送到 Broker，这种方式也是消息的可靠性投递的核心保障！

第一步:在 channel 上开启确认模式:  
channel.confirmSelect()

第二步:在 channel 上添加监听:  
channel.addConfirmListener(ConfirmListener listener);

监听成功和失败的返回结果，根据具体的结果对消息进行重新发送、或记录日志等后续处理！

**Return Listener** 用于处理一些不可路由的消息!

- 消息生产者，通过指定一个 `Exchange` 和 `Routingkey`，把消息送达到某一个队列中去，然后我们的消费者监听队列，进行消费处理操作!
- 但是在某些情况下，如果我们在发送消息的时候，当前的 exchange 不存在或者指定的路由 key 路由不到，这个时候如果我们需要监听这种不可达的消息，就要使用 `Return Listener` !
- 在基础API中有一个关键的配置项: `Mandatory`：如果为 `true`，则监听器会接收到路由不可达的消息，然后进行后续处理，如果为 `false`，那么 broker 端自动删除该消息!

```
// 第三个参数 Mandatory 如果为true return监听可以接收不可达消息
channel.basicPublish("", "hello", true, MessageProperties.PERSISTENT_TEXT_PLAIN, new String("消息内容").getBytes());

channel.addReturnListener(new ReturnListener() {
    @Override
    public void handleReturn(int replyCode, String replyText, String exchange,
        String routingKey, AMQP.BasicProperties properties, byte[] body) throws IOException {
        //此处便是执行Basic.Return之后回调的地方
        String message = new String(body);
        System.out.println("Basic.Return返回的结果:  "+message);
    }
});
```

## (面试点) 消息的持久化机制

在Rabbitmq中 交换器、队列、消息都可以进行持久化的设置，如果没设置持久化，那么对应的数据会存储在内存中，如果RabbitMQ宕机可能会出现消息丢失。要想保证消息不丢失，至少要设置队列 和 消息的持久化

### 队列设置持久化

```
channel.queueDeclare(QueueName, true, false, false, null);
```

- queue：队列的名称。
- durable：是否持久化，即服务器重启后，数据是否还能存活。
  - 值为false时，数据存放到内存中，重启服务会丢失。
  - 值为true时，数据存放到自带的Mnesia数据库中，重启服务后会自动读取该数据库中的数据。

### 消息设置持久化

BasicProperties：需要注意的是BasicProperties.deliveryMode, 1:不持久化 2:持久化 这里指的是消息的持久化，配合channel(durable=true), queue(durable)可以实现，即使服务器宕机，消息仍然保留。该参数的值可以直接使用MessageProperties类中的常量。

## (面试点) 消费者的消息确认机制

通过刚才的案例可以看出，消息一旦被消费者接收，队列中的消息就会被删除。

那么问题来了：RabbitMQ怎么知道消息被接收了呢？

这就要通过消息确认机制（Acknowledge）来实现了。当消费者获取消息后，会向RabbitMQ发送回执ACK，告知消息已经被接收。不过这种回执ACK分两种情况：

- 自动ACK：消息一旦被接收，消费者自动发送ACK
- 手动ACK：消息接收后，不会发送ACK，需要手动调用

大家觉得哪种更好呢？

这需要看消息的重要性：

- 如果消息不太重要，丢失也没有影响，那么自动ACK会比较方便
- 如果消息非常重要，不容丢失。那么最好在消费完成后手动ACK，否则接收消息后就自动ACK，RabbitMQ就会把消息从队列中删除。如果此时消费者宕机，那么消息就丢失了。

```
/**
```

```
* queue: 要监听的队列
* autoAck: 是否自动确认
* consumer: 处理消息的消费者
*/
```

```
channel.basicConsume(queue: "hello", autoAck: false, consumerTag: "我是消费者", consumer);
// 不用关闭channel 监听会持续进行
```

```
// 第二个参数 是否自动确认 false为手动确认
```

```
channel.basicConsume(QUEUE, false, consumer);
```

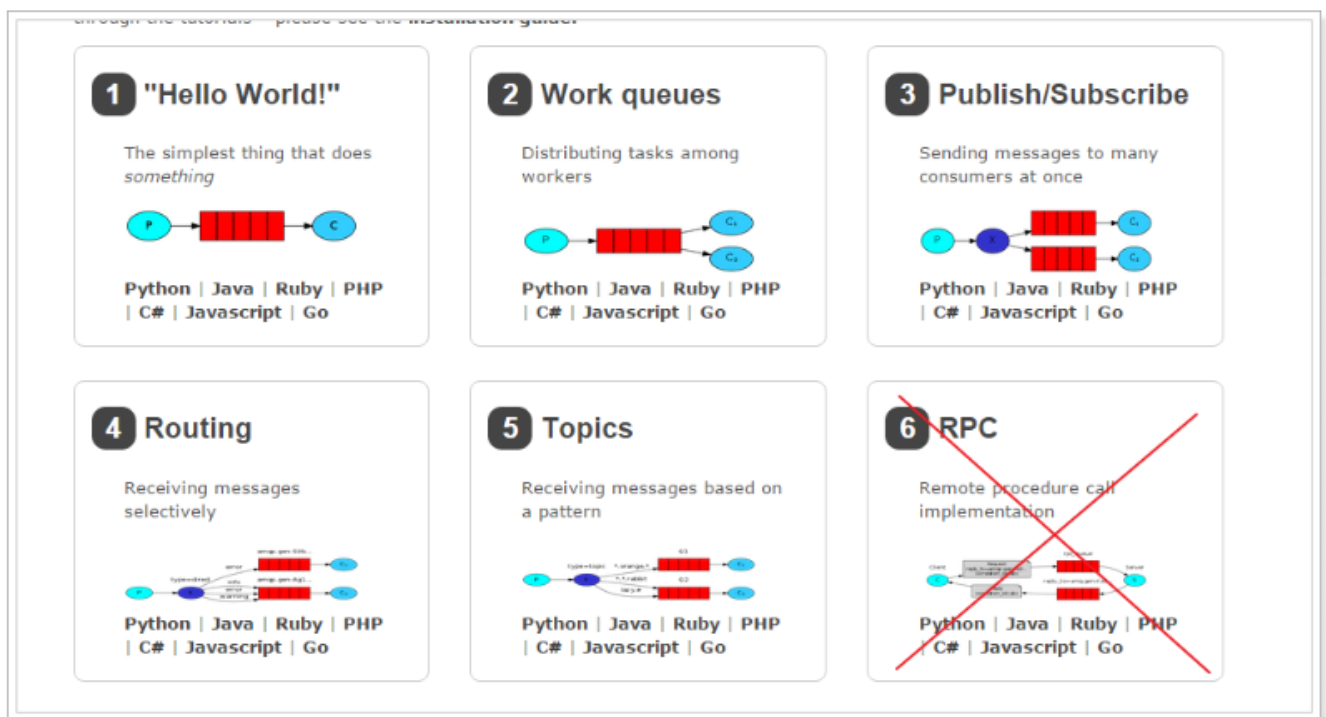
```
// 手动确认消息 消息ID 是否批量确认
```

```
channel.basicAck(envelope.getDeliveryTag(), false);
```

## 4. 五种工作模式演示

RabbitMQ提供了6种消息模型，但是第6种其实是RPC，并不是MQ，因此不予学习。那么也就剩下5种。

参考链接:<https://www.rabbitmq.com/getstarted.html>





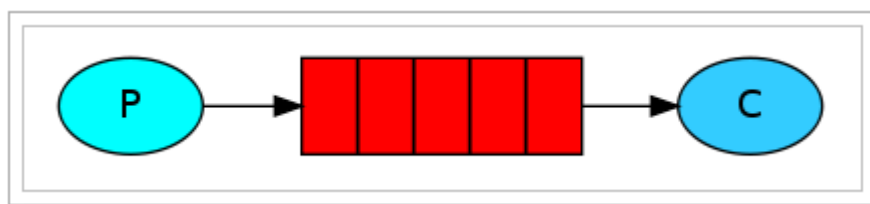
这几种消息模型的实现和Exchange交换机类型有直接关系。交换器（Exchange）说到底是一个名称与队列绑定的列表。当消息发布到交换器时，实际上是由你所连接的信道，将消息路由键同交换器上绑定的列表进行比较，最后路由消息。

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

下面我们会结合案例演示下不同的消息模型是如何发送和消费消息的

## 4.1 简单模式 (默认交换器)

基本消息模型图：



在上图的模型中，有以下概念：

- P：生产者，也就是要发送消息的程序
- C：消费者：消息的接受者，会一直等待消息到来。
- queue：消息队列，图中红色部分。类似一个邮箱，可以缓存消息；生产者向其中投递消息，消费者从其中取出消息。

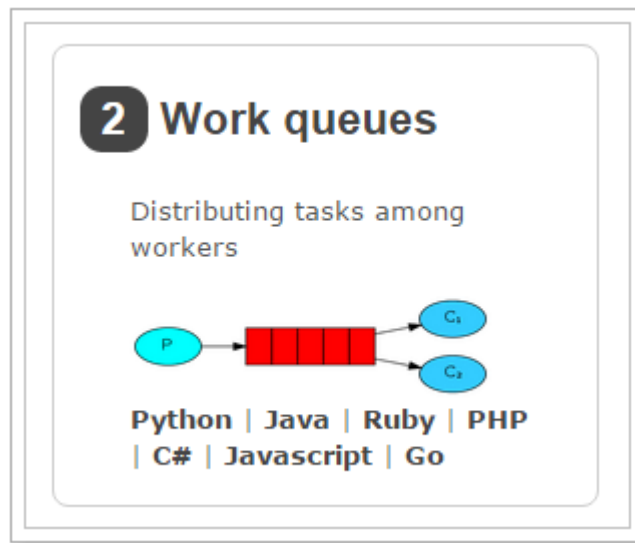
我们在快速入门的案例中使用的就是简单案例，不用自己创建交换器 直接使用默认交换器即可，默认交换器会把指定的 route\_key 直接作为队列的名称找到指定的队列

## 4.2 工作队列模式(默认交换器)

在刚才的基本模型中，一个生产者，一个消费者，生产的消息直接被消费者消费。比较简单。

Work queues，也被称为（Task queues），任务模型。

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



角色：

- P：生产者：任务的发布者
- C1：消费者，领取任务并且完成任务，假设完成速度较慢
- C2：消费者2：领取任务并完成任务，假设完成速度快

更改生产者代码 循环发送50条消息

```
/**
 * MQ的发送者示例
 **/
public class MqProducer {
    public static void main(String[] args) throws IOException, TimeoutException {
        // 1. 创建连接工厂 并设置连接参数
        ConnectionFactory connectionFactory = new ConnectionFactory();
        // 设置 虚拟目录
        connectionFactory.setVirtualHost("/");
        // 设置 host和端口
        connectionFactory.setHost("localhost");
        connectionFactory.setPort(5672);
        // 设置 用户名和密码
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        // 2. 根据工厂对象获取 连接对象
        Connection connection = connectionFactory.newConnection();
        // 3. 根据连接对象获取 信道对象
        Channel channel = connection.createChannel();
        // 4. 通过信道对象声明一个队列
        /**
         * queue: 队列名称
         * durable: 是否持久化
         * exclusive: 是否独占
         * autoDelete: 是否自动删除
         * arguments: 其它参数
         */
        channel.queueDeclare("hello", true, false, false, null);
        // 5. 通过信道对象发送消息
        for (int i = 0; i < 50; i++) {
```

```

        String message = "消息内容==>" + i;
        // 推送消息
        /**
         * exchange: 交换器 *
         * routingKey: 路由 *
         * mandatory: 没有投递队列时 是否返还消息
         * props: 消息的属性
         * body: 消息内容 *
         */
        channel.basicPublish("", "hello", false,
MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes());
    }
    // 6. 关闭信道对象和连接对象
    channel.close();
    connection.close();
}
}

```

复制两个消费者类

一个名称消费者1号 一个名称消费者2号

```

public class MqConsumer1 {
    public static void main(String[] args) throws IOException, TimeoutException {
        // 1. 创建连接工厂 并设置连接参数
        ConnectionFactory connectionFactory = new ConnectionFactory();
        // 设置 虚拟目录
        connectionFactory.setVirtualHost("/");
        // 设置 host和端口
        connectionFactory.setHost("localhost");
        connectionFactory.setPort(5672);
        // 设置 用户名和密码
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        // 2. 根据工厂对象获取 连接对象
        Connection connection = connectionFactory.newConnection();
        // 3. 根据连接对象获取 信道对象
        Channel channel = connection.createChannel();
        // 4. 通过信道对象声明一个队列
        /**
         * queue: 队列名称 *
         * durable: 是否持久化 *
         * exclusive: 是否独占
         * autoDelete: 是否自动删除
         * arguments: 其它参数
         */
        channel.queueDeclare("hello", true, false, false, null);
        // 5. 创建一个消费者, 并重写消费方法
        //
        channel.basicQos(1);

        Consumer consumer = new DefaultConsumer(channel){
            /**

```

```

        * 处理接收到消息的方法
        * @param consumerTag 消费者标签(自定义的字符串)
        * @param envelope 信封(记录消息的路由 交换器 ID 等信息)
        * @param properties 消息的属性信息
        * @param body 消息内容
        * @throws IOException
        */
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
            String s = new String(body);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(consumerTag + " 消费消息 ==>" + s);
            channel.basicAck(envelope.getDeliveryTag(), true);
        }
    };
    // 6. 使用消费者 监听指定消息队列
    /**
     * queue: 要监听的队列
     * autoAck: 是否自动确认
     * consumer: 处理消息的消费者
     */
    channel.basicConsume("hello", false, "消费者1号", consumer);
    // 不用关闭channel 监听会持续进行
}
}

```

```

public class MqConsumer2 {
    public static void main(String[] args) throws IOException, TimeoutException {
        // 1. 创建连接工厂 并设置连接参数
        ConnectionFactory connectionFactory = new ConnectionFactory();
        // 设置 虚拟目录
        connectionFactory.setVirtualHost("/");
        // 设置 host和端口
        connectionFactory.setHost("localhost");
        connectionFactory.setPort(5672);
        // 设置 用户名和密码
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        // 2. 根据工厂对象获取 连接对象
        Connection connection = connectionFactory.newConnection();
        // 3. 根据连接对象获取 信道对象
        Channel channel = connection.createChannel();
        // 4. 通过信道对象声明一个队列
        /**
         * queue: 队列名称
         * durable: 是否持久化
         * exclusive: 是否独占
         * autoDelete: 是否自动删除

```

```

        * arguments: 其它参数
    */
    channel.queueDeclare("hello",true,false,false,null);
    // 5. 创建一个消费者,并重写消费方法
    //
    channel.basicQos(1);
    Consumer consumer = new DefaultConsumer(channel){
        /**
         * 处理接收到消息的方法
         * @param consumerTag 消费者标签(自定义的字符串)
         * @param envelope 信封(记录消息的路由 交换器 ID 等信息)
         * @param properties 消息的属性信息
         * @param body 消息内容
         * @throws IOException
         */
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
            String s = new String(body);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(consumerTag + " 消费消息 ==>" + s);
            channel.basicAck(envelope.getDeliveryTag(),true);
        }
    };
    // 6. 使用消费者 监听指定消息队列
    /**
     * queue: 要监听的队列
     * autoAck: 是否自动确认
     * consumer: 处理消息的消费者
     */
    channel.basicConsume("hello",false,"消费者2号",consumer);
    // 不用关闭channel 监听会持续进行
    }
}

```

查看结果, 会发现两个队列公平的消耗了 队列中的所有消息。可以看到 当多个消费者监听同一队列时 消息只会被消费一次, 并且消息会被轮询分配到每个消费者中。 如果我们的队列数据比较多时 可以通过多个消费者快速消费消息。

```
消费者1号 消费消息 ==>消息内容==>0
消费者1号 消费消息 ==>消息内容==>2
消费者1号 消费消息 ==>消息内容==>4
消费者1号 消费消息 ==>消息内容==>6
消费者1号 消费消息 ==>消息内容==>8
消费者1号 消费消息 ==>消息内容==>10
消费者1号 消费消息 ==>消息内容==>12
消费者1号 消费消息 ==>消息内容==>14
消费者1号 消费消息 ==>消息内容==>16
消费者1号 消费消息 ==>消息内容==>18
消费者1号 消费消息 ==>消息内容==>20
消费者1号 消费消息 ==>消息内容==>22
消费者1号 消费消息 ==>消息内容==>24
消费者1号 消费消息 ==>消息内容==>26
消费者1号 消费消息 ==>消息内容==>28
消费者1号 消费消息 ==>消息内容==>30
消费者1号 消费消息 ==>消息内容==>32
消费者1号 消费消息 ==>消息内容==>34
消费者1号 消费消息 ==>消息内容==>36
消费者1号 消费消息 ==>消息内容==>38
消费者1号 消费消息 ==>消息内容==>40
消费者1号 消费消息 ==>消息内容==>42
消费者1号 消费消息 ==>消息内容==>44
消费者1号 消费消息 ==>消息内容==>46
消费者1号 消费消息 ==>消息内容==>48
```

```
消费者2号 消费消息 ==>消息内容==>1
消费者2号 消费消息 ==>消息内容==>3
消费者2号 消费消息 ==>消息内容==>5
消费者2号 消费消息 ==>消息内容==>7
消费者2号 消费消息 ==>消息内容==>9
消费者2号 消费消息 ==>消息内容==>11
消费者2号 消费消息 ==>消息内容==>13
消费者2号 消费消息 ==>消息内容==>15
消费者2号 消费消息 ==>消息内容==>17
消费者2号 消费消息 ==>消息内容==>19
消费者2号 消费消息 ==>消息内容==>21
消费者2号 消费消息 ==>消息内容==>23
消费者2号 消费消息 ==>消息内容==>25
消费者2号 消费消息 ==>消息内容==>27
消费者2号 消费消息 ==>消息内容==>29
消费者2号 消费消息 ==>消息内容==>31
消费者2号 消费消息 ==>消息内容==>33
消费者2号 消费消息 ==>消息内容==>35
消费者2号 消费消息 ==>消息内容==>37
消费者2号 消费消息 ==>消息内容==>39
消费者2号 消费消息 ==>消息内容==>41
消费者2号 消费消息 ==>消息内容==>43
消费者2号 消费消息 ==>消息内容==>45
消费者2号 消费消息 ==>消息内容==>47
消费者2号 消费消息 ==>消息内容==>49
```

## 能者多劳

刚才的实现有问题吗？

- 消费者1比消费者2的效率要低，一次任务的耗时较长
- 然而两人最终消费的消息数量是一样的
- 消费者2大量时间处于空闲状态，消费者1一直忙碌

现在的状态属于是把任务平均分配，正确的做法应该是消费越快的人，消费的越多。

怎么实现呢？

我们可以修改设置，让消费者同一时间只接收一条消息，这样处理完成之前，就不会接收更多消息，就可以让处理快的人，接收更多消息：

```
// 在消费者的代码中添加设置
channel.basicQos(1);
// 注意：该设置只在手动确认时有效
```

## 4.3 演示前的准备

拷贝资料中的工具类到项目中

### 准备工具类

```
/**
 * 创建RabbitMQ的工具类
 * 快捷的获取连接
 * @作者 itcast
 * @创建日期 2020/4/8 15:05
 **/
public class RabbitUtils {
    //连接对象
    private static Connection connection=null;
    //频道 ( 通道 )
    private static Channel channel=null;

    static{
        try {
            //初始化连接
            init();
            // 虚拟机退出时关闭
            Runtime.getRuntime().addShutdownHook(new Thread(){
                @Override
                public void run() {
                    System.out.println("-----释放关闭资源中....");
                    //关闭释放资源
                    destory();
                    System.out.println("-----释放关闭资源成功....");
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //初始化
    public static void init() throws Exception {
        //创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //设置连接参数 - 主机地址;默认为 localhost
        connectionFactory.setHost("localhost");
        //设置连接参数 - 连接端口;默认为 5672
        connectionFactory.setPort(5672);
        //设置连接参数 - 虚拟主机名称;默认为 /
        connectionFactory.setVirtualHost("/");
        //设置连接参数 - 连接用户名;默认为guest
        connectionFactory.setUsername("guest");
        //设置连接参数 - 连接密码;默认为guest
    }
}
```

```

        connectionFactory.setPassword("guest");
        //创建连接
        connection = connectionFactory.newConnection();
        //创建频道
        channel = connection.createChannel();
    }

    //释放资源
    public static void destory(){
        //关闭频道
        if(null !=channel && channel.isOpen()){
            try {
                channel.close();
            } catch (IOException e) {
                e.printStackTrace();
            } catch (TimeoutException e) {
                e.printStackTrace();
            }
        }

        //关闭连接
        if(null !=connection && connection.isOpen()){
            try {
                connection.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    //获取连接
    public static Connection getConnection(){
        return connection;
    }

    //获取连接
    public static Channel getChannel(){
        return channel;
    }
}

```

## 准备发送者类

```

/**
 * Exchange交换器的4种类型
 * @作者 itcast
 * @创建日期 2020/3/20 14:25
 */
public class MessageProducerExchange {
    Channel channel = null;
    /**
     * test前会执行

```



```

    * 初始化客户端连接
    */
@Before
public void init(){
    channel = RabbitUtils.getChannel();
}
/**
 * test后执行
 * 释放连接
 */
@After
public void close(){
    RabbitUtils.destory();
}
/**
 * 创建消息队列
 * 声明队列：红色队列 绿色队列 黄色队列 颜色队列
 * @throws Exception
 */
@Test
public void createQueue() throws Exception {

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

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

    channel.queueDeclare("yellow_queue", true, false, false, null);
    channel.queueDeclare("color_queue", true, false, false, null);
}
}

```

准备几个队列: red、green、yellow、color

```

/**
 * 红色队列的消息监听
 * @作者 itcast
 * @创建日期 2020/3/20 11:26
 */
public class RedConsumer {
    //队列名字
    static final String QUEUE_NAME = "red_queue";
    //入口方法
    public static void main(String[] args) throws Exception {
        Channel channel = RabbitUtils.getChannel();
        DefaultConsumer consumer = new DefaultConsumer(channel){
            /**
             * consumerTag 消费者标签，在channel.basicConsume时候可以指定
             * envelope 消息包的内容，可从中获取消息id，
             * 消息routingkey，交换机，消息和重传标志(收到消息失败后是否需要重新发送)
             * properties 属性信息
             * body 消息内容

```

```

        */
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
            String message=new String(body, StandardCharsets.UTF_8);
            //打印消息
            System.out.println("当前队列 == "+ consumerTag +" 接收到的消息为：" + message);
        }
    };
    //消费队列中的消息，会自动开启监听。
    /**
     * 参数1：队列名称
     * 参数2：是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，
     *      mq接收到回复会删除消息，设置为false则需要手动确认
     * 参数3：消息接收到后回调
     */
    channel.basicConsume(QUEUE_NAME, true, QUEUE_NAME, consumer);
    //不关闭资源，应该一直监听消息
}
}

```

```

/**
 * 绿色队列监听
 * @作者 itcast
 * @创建日期 2020/3/20 11:26
 */
public class GreenConsumer {
    //队列名字
    static final String QUEUE_NAME = "green_queue";
    //入口方法
    public static void main(String[] args) throws Exception {
        Channel channel = RabbitUtils.getChannel();

        DefaultConsumer consumer = new DefaultConsumer(channel){
            /**
             * consumerTag 消费者标签，在channel.basicConsume时候可以指定
             * envelope 消息包的内容，可从中获取消息id，
             *      消息routingkey，交换机，消息和重传标志(收到消息失败后是否需
             *      要重新发送)
             * properties 属性信息
             * body 消息内容
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message=new String(body, StandardCharsets.UTF_8);
                //打印消息
                System.out.println("当前队列 == "+ consumerTag +" 接收到的消息为：" + message);
            }
        };
        //消费队列中的消息，会自动开启监听。
        /**
         * 参数1：队列名称

```

```

        * 参数2：是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，
        *      mq接收到回复会删除消息，设置为false则需要手动确认
        * 参数3：消息接收到后回调
        */
channel.basicConsume(QUEUE_NAME, true, QUEUE_NAME, consumer);
//不关闭资源，应该一直监听消息
    }
}

```

```

/**
 * 黄色队列监听
 * @作者 itcast
 * @创建日期 2020/3/20 11:26
 */
public class yellowConsumer {
    //队列名字
    static final String QUEUE_NAME = "yellow_queue";
    //入口方法
    public static void main(String[] args) throws Exception {
        Channel channel = RabbitUtils.getChannel();
        DefaultConsumer consumer = new DefaultConsumer(channel){
            /**
             * consumerTag 消费者标签，在channel.basicConsume时候可以指定
             * envelope 消息包的内容，可从中获取消息id，
             *      消息routingkey，交换机，消息和重传标志(收到消息失败后是否需要重新发送)
             * properties 属性信息
             * body 消息内容
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                //路由key
//                System.out.println("路由key为：" + envelope.getRoutingKey());
                //交换机
//                System.out.println("交换机为：" + envelope.getExchange());
                //消息id
//                System.out.println("消息id为：" + envelope.getDeliveryTag());
                //收到的消息
//                String message=new String(body, "UTF-8");
                String message=new String(body, StandardCharsets.UTF_8);
                //打印消息
                System.out.println("当前队列 == "+ consumerTag +" 接收到的消息为：" + message);
            }
        };
        //消费队列中的消息，会自动开启监听。
    /**
     * 参数1：队列名称
     * 参数2：是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，
     *      mq接收到回复会删除消息，设置为false则需要手动确认
     * 参数3：消息接收到后回调
     */
    channel.basicConsume(QUEUE_NAME, true, QUEUE_NAME, consumer);
}

```

```

        //不关闭资源，应该一直监听消息
    }
}

```

```

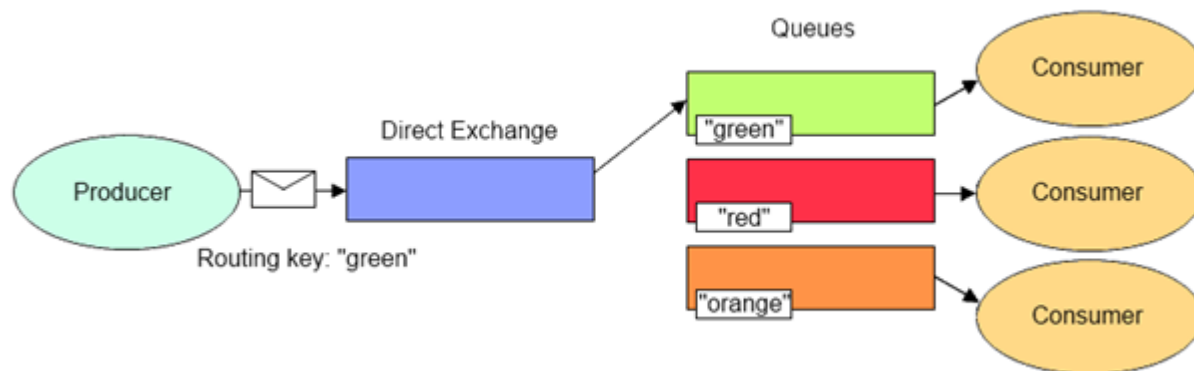
/**
 * color队列监听
 * @作者 itcast
 * @创建日期 2020/3/20 11:26
 **/
public class ColorConsumer {
    //队列名字
    static final String QUEUE_NAME = "color_queue";
    //入口方法
    public static void main(String[] args) throws Exception {
        Channel channel = RabbitUtils.getChannel();

        DefaultConsumer consumer = new DefaultConsumer(channel){
            /**
             * consumerTag 消费者标签，在channel.basicConsume时候可以指定
             * envelope 消息包的内容，可从中获取消息id，
             * 消息routingkey，交换机，消息和重传标志(收到消息失败后是否需要重新发送)
             * properties 属性信息
             * body 消息内容
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                String message=new String(body, StandardCharsets.UTF_8);
                //打印消息
                System.out.println("当前队列 == "+ consumerTag +" 接收到的消息为：" + message);
            }
        };
        //消费队列中的消息，会自动开启监听。
        /**
         * 参数1：队列名称
         * 参数2：是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，
         * mq接收到回复会删除消息，设置为false则需要手动确认
         * 参数3：消息接收到后回调
         */
        channel.basicConsume(QUEUE_NAME, true, QUEUE_NAME, consumer);
        //不关闭资源，应该一直监听消息
    }
}

```

## 4.4 路由模式演示 (Direct)

### 说明和机制



直连型交换机（direct exchange）是根据消息携带的路由键（routing key）将消息投递给对应队列的。设置 Exchange 和 Queue 的 Binding 时需指定 RoutingKey（一般为 Queue Name），发消息时也指定一样的 RoutingKey，消息就会被路由到对应的 Queue。直连交换机用来处理消息的单播路由（unicast routing，尽管它也可以处理多播路由）（一对一模式）

当我们需要将一个消息只能发送到一个队列的时候，需要使用这种模式。

工作机制：

1. 将一个队列绑定到某个交换机上，同时赋予该绑定一个路由键（routing key）。
2. 当一个携带着路由键为 green 的消息被发送给直连交换机时，交换机会把它路由给绑定值同样为 green 的队列。

说明：

- 消息传递时需要一个“RouteKey”，可以简单的理解为要发送到的队列名字。
- 这种模式需要提前将 Exchange 与 Queue 进行绑定，一个 Exchange 可以绑定多个 Queue，一个 Queue 可以同多个 Exchange 进行绑定。在绑定 Queue 的同时，需要绑定 routingKey。一般 Queue 和 routingKey 的名字一样。
- 如果接受到消息的 Exchange 没有与任何 Queue 绑定，则消息会被抛弃。
- 一般情况下，可以直接使用 Default Exchange，因为不需要将 Exchange 进行任何绑定(binding)操作。
- 如果使用 Default Exchange，vhost 中不存在 RouteKey 中指定的队列名，则该消息会被抛弃。

Default Exchange 是一种特殊的 Direct Exchange，是 RabbitMQ 内置、默认的。当你手动创建一个队列时，后台会自动将这个队列绑定到一个名称为空的 Direct Exchange 上，绑定 RoutingKey 与队列名称相同。有了这个默认的交换机和绑定，使我们只关心队列这一层即可，这个比较适合做一些简单的应用，比如上面的 Hello World 示例。

提示：对于直连交换机的相关需求，一般使用默认交换机即可解决。

## 生产者代码

```

/**
 * 创建 直接模式交换机
 *
 * 1. 交换器: direct_exchange
 *
 * 2. 队列: red_queue
 *          green_queue
 *          yellow_queue
 *
 * 3. 绑定关系:
 *      交换器      路由key      队列
 *      direct_exchange      red      red_queue
 *      direct_exchange      green     green_queue

```

```

*    direct_exchange    yellow    yellow_queue
* 4. 发送消息
*/
@Test
public void sendDirect() throws Exception {
    // 声明一个 直接模式的交换机
    channel.exchangeDeclare("direct_exchange",BuiltinExchangeType.DIRECT);
    // 通过路由key 绑定队列和交换器
    channel.queueBind("red_queue","direct_exchange","red");
    channel.queueBind("green_queue","direct_exchange","green");
    channel.queueBind("yellow_queue","direct_exchange","yellow");
    // 发送消息
    String message = "发送的消息";
    channel.basicPublish("direct_exchange","yellow",null,message.getBytes());
}

```

测试结果：

消息经过交换机后，会根据绑定队列列表 匹配路由key 将消息发送到满足的队列中

使用图形化界面操作（参考）

创建交换器

Overview	Connections	Channels	Exchanges	Queues	Admin
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
direct_ex	direct		0.00/s	0.00/s	
fanout_ex	fanout		0.00/s	0.00/s	
spring_topic_exchange	topic	D	0.00/s	0.00/s	
topic_ex	topic		0.00/s	0.00/s	

#### ▼ Add a new exchange

Name:  \*

Type:  ▼

Durability:  ▼

Auto delete: ?  ▼

Internal: ?  ▼

Arguments:  =  String ▼

Add Alternate exchange ?

Add exchange

配置交换机

依次绑定需要走该交换机的队列名字和路由key：

Overview

Connections

Channels

Exchanges

Queues

Admin

Policy

▼ Bindings

This exchange

⇓

... no bindings ...

Add binding from this exchange

To queue ▼ : red\_queue \*

Routing key: red

Arguments: = String ▼

Bind

▼ Publish message

Routing key:

Delivery mode: 1 - Non-persistent ▼

Headers: ? = String ▼

绑定结果：

Overview

Connections

Channels

Exchanges

Queues

Admin

This exchange

⇓

To	Routing key	Arguments	
color_queue	color		Unbind
green_queue	green		Unbind
red_queue	red		Unbind
yellow_queue	yellow		Unbind

测试发送消息

Overview

Connections

Channels

Exchanges

Queues

Admin

Routing key:

Arguments:

=

String

Bind

▼ Publish message

Routing key:

red

Delivery mode:

2 - Persistent

Headers: ?

=

String

Properties: ?

=

Payload:

使用后台管理页面发送消息

Publish message

Overview

Messages

Message rates

Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
color_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
email_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
green_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
hello	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
red_queue	D	idle	1	0	1	0.00/s	0.00/s	0.00/s
sms_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
yellow_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s

进入对应队列 可以通过get message查看对应消息



Encoding:  ?

Messages:

**Get Message(s)**

---

Message 1

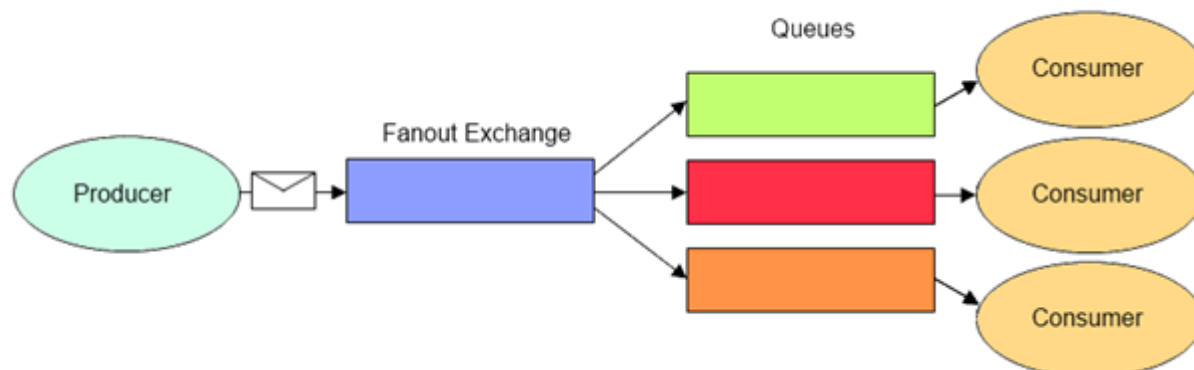
The server reported 0 messages remaining.

Exchange	direct_exchange
Routing Key	red
Redelivered	<input type="radio"/>
Properties	delivery_mode: 1 headers:
Payload	使用后台管理页面发送的消息

39 bytes  
Encoding: string

## 4.4 发布订阅模式演示 (Fanout)

### 说明和机制



扇出交换机 ( fanout exchange ) 将消息路由给绑定到它身上的所有队列，而不会理会绑定的路由键。如果N个队列绑定到某个扇型交换机上，当有消息发送给此扇出交换机时，交换机会将消息的拷贝分别发送给这所有的N个队列。扇出用来交换机处理消息的广播路由 ( broadcast routing )。简单的说，Fanout Exchange 会忽略RoutingKey 的设置，直接将 Message 广播到所有绑定的 Queue 中。因此，你也可以将该交换机称之为广播交换机。

当我们需要将消息一次发给多个队列时，需要使用这种模式。

工作机制：

1. n个队列绑定到一个扇出交换机上。
2. 当消息发送到这个交换机上，这个消息会被分发给与之绑定的所有队列里。

说明：

- 可以理解为路由表的模式。
- 这种模式需要提前将Exchange与Queue进行绑定，一个Exchange可以绑定多个Queue，一个Queue可以同多个Exchange进行绑定。
- 这种模式不需要RouteKey，即使绑定也会被忽略。

- 如果接受到消息的Exchange没有与任何Queue绑定，则消息会被抛弃。

## 生产者代码

```
/**
 * 创建 分发模式交换器
 *
 * 交换器: fanout_exchange
 *
 * 队列: red_queue
 *       green_queue
 *       yellow_queue
 * 绑定关系:
 *      交换器      路由key      队列
 *      fanout_exchange      red_queue
 *                          green_queue
 *                          yellow_queue
 */
@Test
public void sendFanout() throws Exception {
    // 绑定 交换器和队列 路由为"" 即可，因为分发模式不通过路由
    channel.queueBind("red_queue", "fanout_exchange", "");
    channel.queueBind("green_queue", "fanout_exchange", "");
    channel.queueBind("yellow_queue", "fanout_exchange", "");

    // 发送消息
    String message = "发送的消息";
    channel.basicPublish("fanout_exchange", "sadsad", null, message.getBytes());
}
```

测试结果：

消息经过交换机后，所有的绑定队列都收到了消息

### 使用图形化界面操作（参考）

#### 创建队列

创建两个fanout的队列：fanout-queue1和fanout-queue2

Add a new queue

Name:
fanout-queue1

Durability:
Durable

Auto delete:
?
No

Arguments:
=
String

Add
Message TTL
?
|
Auto expire
?
|
Max length
?
|
Max length bytes
?
|
Over
Dead letter exchange
?
|
Dead letter routing key
?
|
Maximum priority
?
|
Lazy mode
?
|
Master locator
?

Add queue

创建结果：

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
direct-queue2	D	idle	1	0	1	0.00/s		
fanout-queue1	D	idle	0	0	0			
fanout-queue2	D	idle	0	0	0			
hello		idle	0	0	0	0.00/s	0.00/s	0.00/s

## 创建和配置交换机

创建一个fanout类型的交换机：fanout-exchange

Add a new exchange

Name:
fanout-exchange

Type:
fanout

Durability:
Durable

Auto delete:
?
No

Internal:
?
No

Arguments:
=
String

Add
Alternate exchange
?

Add exchange

依次绑定需要走该交换机的队列名字：

Add binding from this exchange

To queue  \*

Routing key:

Arguments:  =

绑定结果：

▼ Bindings

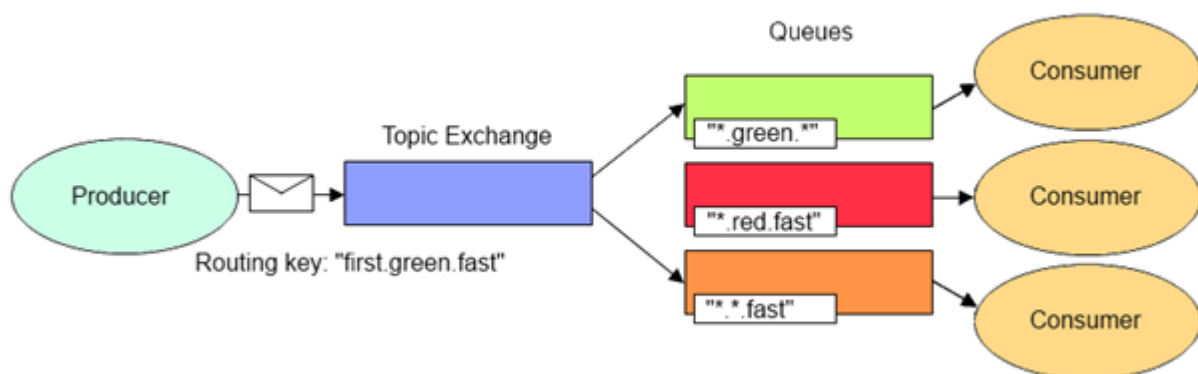
This exchange

⇕

To	Routing key	Arguments	
fanout-queue1			<input type="button" value="Unbind"/>
fanout-queue2			<input type="button" value="Unbind"/>

## 4.5 通配符模式演示 (Topic)

### 说明和机制



主题交换机 (topic exchanges) 通过对消息的路由键和队列到交换机的绑定模式之间的匹配，将消息路由给一个或多个队列。主题交换机经常用来实现各种分发/订阅模式及其变种。主题交换机通常用来实现消息的多播路由 (multicast routing)。Topic Exchange 配置上和 Direct Exchange 类似，也需要通过 RoutingKey 来路由消息，区别在于 Direct Exchange 对 RoutingKey 是精确匹配，而 Topic Exchange 支持模糊匹配。关于通配符，分别支持 \* 和 # 通配符，\* 表示匹配一个单词，# 则表示匹配没有或者多个单词。因此 first.# 能够匹配到 first.green 或 first.green.fast，但是 first.\* 只会匹配到 first.green，不能匹配 first.green.fast。

说明：

- 1.这种模式较为复杂，简单来说，就是每个队列都有其关心的主题，所有的消息都带有一个“标题”(RouteKey)，Exchange会将消息转发到所有关注主题能与RouteKey模糊匹配的队列。
- 2.这种模式需要RouteKey，也许要提前绑定Exchange与Queue。
- 3.在进行绑定时，要提供一个该队列关心的主题，如“#.log.#”表示该队列关心所有涉及log的消息(一个RouteKey为“MQ.log.error”的消息会被转发到该队列)。
- 4.同样，如果Exchange没有发现能够与RouteKey匹配的Queue，则会抛弃此消息

## 生产者代码

```
/**
 * 创建 主题交换器
 *
 * 交换器: topic_exchange
 *
 * 队列: red_queue
 *       green_queue
 *       yellow_queue
 *
 * 绑定关系:
 *      交换器          路由key          队列
 *      topic_exchange  *.red.color    red_queue
 *      topic_exchange  *.green.color  green_queue
 *      topic_exchange  *.yellow.color yellow_queue
 *      topic_exchange  #.color       color_queue
 */
@Test
public void sendTopic() throws Exception {
    // 声明交换器
    channel.exchangeDeclare("topic_exchange", BuiltinExchangeType.TOPIC);
    // 通过路由key 绑定交换器和队列
    channel.queueBind("red_queue", "topic_exchange", "*.red.color");
    channel.queueBind("green_queue", "topic_exchange", "*.green.color");
    channel.queueBind("yellow_queue", "topic_exchange", "*.yellow.color");
    channel.queueBind("color_queue", "topic_exchange", "#.color");

    String message = "发送的消息";
    channel.basicPublish("topic_exchange", "color.yellow.red", null, message.getBytes());
}
```

### 使用图形化界面操作（参考）

#### 创建和配置交换机

创建一个topic类型的交换机：topic-exchange

▼ Add a new exchange

Name:  \*

Type:  ▼

Durability:  ▼ +

Auto delete: ?  ▼

Internal: ?  ▼

Arguments:  =

Add Alternate exchange ?

Add exchange

依次绑定需要走该交换机的队列规则：queuedemo1

Add binding from this exchange

To queue ▼:  \*

Routing key:

Arguments:  =  S

Bind

绑定结果：

▼ Bindings

This exchange

⇓

To	Routing key	Arguments	
queuedemo1	first.#		Unbind
queuedemo2	first.*		Unbind
queuedemo2	first.red.fast		Unbind

## 4.6 工作模式和交换机的关系小结

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

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）

## 1 "Hello World!"

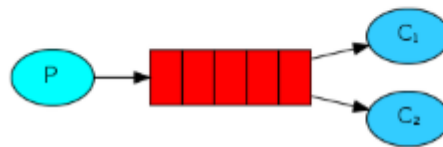
The simplest thing that does  
*something*



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）

## 2 Work queues

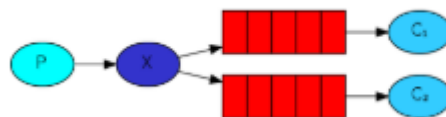
Distributing tasks among  
workers (the competing  
consumers pattern)



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

## 3 Publish/Subscribe

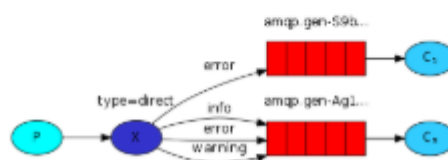
Sending messages to many  
consumers at once



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

## 4 Routing

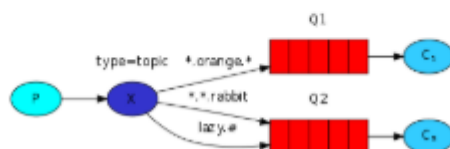
Receiving messages  
selectively



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 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

## 5 Topics

Receiving messages based on a pattern (topics)



工作模式和交换机的关系：

(1) 不配置Exchange交换机（默认交换机）

1) simple简单模式：一个生产者发送消息到队列中由一个消费者接收。

2) work工作队列模式：一个生产者发送消息到队列中可由多个消费者接收；多个消费者之间消息是竞争接收。

(2) 自己配置Exchange交换机；订阅模式（广播fanout，定向direct，通配符topic）

1) 发布与订阅模式：使用了fanout类型的交换机，可以将一个消息发送到所有与交换机绑定的队列并被消费者接收。

2) 路由模式：使用了direct类型的交换机，可以将一个消息发送到routing key相关的队列并被消费者接收。

3) 通配符模式：使用了topic类型的交换机，可以将一个消息发送到routing key（\*,#）相关的队列并被消费者接收。

## 5. Spring整合RabbitMQ

在订阅模式消息通讯中，需要配置交换机，常用的交换机有三种：direct定向、fanout广播、topic通配符。我们这里使用topic通配符模式，功能最强大，这种在企业应用中最为广泛。

### 5.1 消息生产者

需求

模拟发送短信内容到MQ。

步骤

1. 创建消息生产者项目：spring\_rabbitmq\_producer，并添加依赖
2. 编写spring整合rabbitmq配置文件
3. 发消息测试代码



## 实现

1. Maven方式创建消息生产者项目 `spring_rabbitmq_producer` , pom.xml如下：

```
<dependencies>
    <!-- Spring整合RabbitMQ包 -->
    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>2.2.3.RELEASE</version>
    </dependency>
    <!-- Spring 单元测试包 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.2.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
    <!-- Jackson -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.10.2</version>
    </dependency>
</dependencies>
```

2. 在resources中添加编写spring整合rabbitmq配置文件：

```
<?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: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/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">
    <!-- 配置连接工厂 -->
    <rabbit:connection-factory
        id="connectionFactory"
        host="127.0.0.1"
        port="5672"
        username="guest"
        password="guest"
        virtual-host="/" />
    <!-- 配置rabbitMQ admin -->
    <rabbit:admin connection-factory="connectionFactory"></rabbit:admin>

    <!-- 配置消息队列 接收发送短信通知的队列 -->
    <rabbit:queue name="queue_msg_sms" auto-declare="true" />
```

```

<!-- 配置消息队列 接收发送邮件通知的队列 -->
<rabbit:queue name="queue_msg_mail" auto-declare="true"/>

<!-- 配置主题模式的交换机 -->
<rabbit:topic-exchange name="exchange_msg_topic">
    <!-- 配置绑定信息 -->
    <rabbit:bindings>
        <!-- pattern:绑定的路由表达式 queue:绑定的队列名称 -->
        <rabbit:binding pattern="#.sms#.msg" queue="queue_msg_sms"></rabbit:binding>
        <rabbit:binding pattern="#.email#.msg" queue="queue_msg_mail"></rabbit:binding>
    </rabbit:bindings>
</rabbit:topic-exchange>
<!-- 配置操作mq的模板类 -->
<!-- exchange: 默认发送消息的交换机 message-converter: 消息的json转换器 -->
<rabbit:template id="rabbitTemplate" connection-factory="connectionFactory"
exchange="exchange_msg_topic" message-converter="jsonMessageConverter"/>

<!-- 消息对象json转换类 -->
<bean id="jsonMessageConverter"
class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter"/>
</beans>

```

### 3. 在main/java中编写发消息的代码

#### MessageProducer

```

/**
 * @作者 itcast
 * @创建日期 2020/3/20 21:26
 **/
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class MessageProducer {
    @Autowired
    RabbitTemplate rabbitTemplate;

    @Test
    public void sendMsg(){
        System.out.println("用户注册成功，通知MQ发送短信或邮件");

        Map<String,String> map = new HashMap();
        map.put("phone","17701012020");
        map.put("email","171@163.com");
        map.put("content","恭喜您，您的账号已经注册成功");
        // rabbitTemplate.convertAndSend("msg.sms.user",map);
        rabbitTemplate.convertAndSend("msg.email.user",map);
    }
}

```

执行test方法，通过图形化界面查看交换机、队列、消息的情况。

## 5.2 消息消费者

### 需求

模拟从MQ读取短信内容，发送到短信网关服务器上（本例就控制台打印）。

### 步骤

1. 创建消费者项目：spring\_rabbitmq\_consumer，并添加依赖。
2. **编写消息监听器**
3. 编写spring整合rabbitmq配置文件：applicationContext-rabbitmq-consumer.xml
4. 启动运行消费者项目

### 实现

1. Maven方式创建消息生产者项目 `springmq_consumer`

#### 依赖信息

```
<dependencies>
    <!-- Spring整合RabbitMQ包 -->
    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>2.2.3.RELEASE</version>
    </dependency>
    <!-- Jackson -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.10.2</version>
    </dependency>
</dependencies>
```

2. 编写消息监听器

#### SmsListener

```
/**
 * 监听短信消息队列
 * 接收到消息后
 * 进行发送短信的操作
 * @作者 itcast
 * @创建日期 2020/3/20 21:38
 */
@Component
public class SmsListener implements MessageListener {
    @Autowired
    ObjectMapper objectMapper;
```

```

public void onMessage(Message message) {
    try {
        //目标：处理消息
        //获取短信相关内容
        byte[] smsBody = message.getBody();
        //使用Jackson转换为Java ( json ) 对象
        JsonNode jsonNode = objectMapper.readTree(smsBody);
        //获取手机号
        String mobile = jsonNode.get("phone").asText();
        //获取短信内容
        String msgContent = jsonNode.get("content").asText();
        //发送短信(打印消息)
        System.out.println("获取队列中消息, 手机号：" + mobile + ", 短信内容：" + msgContent);

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

## EmailListener

```

/**
 * 监听邮件队列消息
 * 接收到消息后发送邮件
 * @作者 itcast
 * @创建日期 2020/3/20 21:38
 */
@Component
public class EmailListener implements MessageListener {
    @Autowired
    ObjectMapper objectMapper;

    public void onMessage(Message message) {
        try {
            //目标：处理消息
            //获取邮件相关内容
            byte[] emailBody = message.getBody();
            //使用Jackson转换为Java ( json ) 对象
            JsonNode jsonNode = objectMapper.readTree(emailBody);
            //获取手机号
            String email = jsonNode.get("email").asText();
            //获取短信内容
            String msgContent = jsonNode.get("content").asText();
            //发送短信(打印消息)
            System.out.println("获取队列中消息, 邮箱号：" + email + ", 邮件内容：" + msgContent);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

### 3. 在resources中添加编写spring整合rabbitmq配置文件：

```
<?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:rabbit="http://www.springframework.org/schema/rabbit"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/rabbit
http://www.springframework.org/schema/rabbit/spring-rabbit.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <!--组件扫描，需要添加pom依赖 spring-context -->
    <context:component-scan base-package="com.itcast.rabbitmq.listener"/>
    <!--配置连接-->
    <rabbit:connection-factory
        id="connectionFactory"
        host="127.0.0.1"
        port="5672"
        username="guest"
        password="guest"
        virtual-host="/" />
    <!-- 配置 rabbitmq管理端 -->
    <rabbit:admin connection-factory="connectionFactory"></rabbit:admin>
    <!--配置队列名（如果不存在则创建）-->
    <!-- 配置消息队列 接收发送短信通知的队列 -->
    <rabbit:queue name="queue_msg_sms" />
    <!-- 配置消息队列 接收发送邮件通知的队列 -->
    <rabbit:queue name="queue_msg_mail" />
    <!--配置监听-->
    <rabbit:listener-container connection-factory="connectionFactory">
        <!-- 配置监听器对象和要监听的队列的绑定 -->
        <rabbit:listener ref="smsListener" queue-names="queue_msg_sms" />
        <rabbit:listener ref="emailListener" queue-names="queue_msg_mail"/>
    </rabbit:listener-container>
    <!-- Jackson -->
    <bean id="objectMapper" class="com.fasterxml.jackson.databind.ObjectMapper"/>
</beans>
```

### 4. 编写代码，启动运行消费者项目

#### ConsumerTest

```
/**
 * 启动SpringIOC容器
 * System.in.read();让程序不停止
 * 这样监听器可以一直监听消息队列
 * @作者 itcast
 * @创建日期 2020/3/20 21:44
 */
public class ConsumerTest {
    public static void main(String[] args) {
```

```
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

执行main方法，启动监听，查看控制台打印，并通过图形化界面查看交换机、队列、消息的情况。