

0. 学习目标

- 能够说出什么是消息中间件
- 能够安装RabbitMQ
- 能够编写RabbitMQ的入门程序
- 能够说出RabbitMQ的5种模式特征
- 能够使用Spring整合RabbitMQ

1. 消息中间件概述

1.1. 什么是消息中间件

先来说一个故事：

周末无聊刷着手机，某宝网APP突然蹦出来一条消息“为了回馈老客户，女朋友买一送一，活动仅限今天！”。买一送一还有这种好事，那我可不能错过！忍不住立马点了去。于是选了两个最新款，下单、支付一气呵成！满足的躺在床上，想着马上有女朋友了，竟然幸福的失眠了.....

第二天正常上着班，突然接到快递小哥的电话：

小哥：“你是xx吗？你的女朋友到了，我现在在你楼下，你来拿一下吧！”。

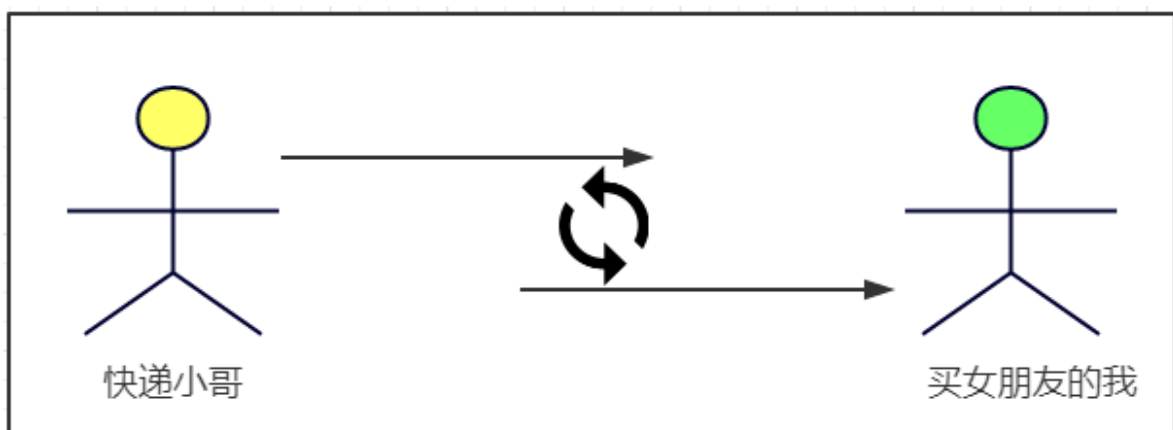
我：“这.....我在上班呢，可以晚上送过来吗？”。

小哥：“晚上可不行哦，晚上我也下班了呢！”。

于是两个人僵持了很久.....

最后小哥说，要不我帮你放到楼下小芳便利店吧，你晚上下班了过来拿，尴尬的局面这才得以缓解！

回到正题，如果没有小芳便利店，那快递小哥和我的交互图就应该如下：



会出现什么情况呢？

- 1、为了这个女朋友，我请假回去拿（老板不批）。
- 2、小哥一直在你楼下等（小哥还有其他的快递要送）。
- 3、周末再送（显然等不及）。
- 4、这个女朋友我不要了（绝对不可能）！

小芳便利店出现后，交互图就应如下：



在上面例子中，“快递小哥”和“买女朋友的我”就是需要交互的两个系统，小芳便利店就是我们本文要讲的“消息中间件”。总结下来小芳便利店（消息中间件）出现后有如下好处：

1、解耦

快递小哥手上有许多快递需要送，他每次都需要先电话——确认收货人是否有空、哪个时间段有空，然后再确定好送货的方案。这样完全依赖收货人了！如果快递一多，快递小哥估计的忙疯了……如果有了便利店，快递小哥只需要将同一个小区的快递放在同一个便利店，然后通知收货人来取货就可以了，这时候快递小哥和收货人就实现了解耦！

2、异步

快递小哥打电话给我后需要一直在你楼下等着，直到我拿走你的快递他才能去送其他人的。快递小哥将快递放在小芳便利店后，又可以干其他的活儿去了，不需要等待你到来而一直处于等待状态。提高了工作的效率。

3、削峰

假设双十一我买了不同店里的各种商品，而恰巧这些店发货的快递都不一样，有中通、圆通、申通、各种通等……更巧的是他们都同时到货了！中通的小哥打来电话叫我去北门取快递、圆通小哥叫我去南门、申通小哥叫我去东门。我一时手忙脚乱……

我们能看到在系统需要交互的场景中，使用消息队列中间件真的是好处多多，基于这种思路，就有了丰巢、菜鸟驿站等比小芳便利店更专业的“中间件”了。

最后，上面的故事纯属虚构……

MQ全称为Message Queue，消息队列是应用程序和应用程序之间的通信方法。

- 为什么使用MQ

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**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：

1、任务**异步**处理

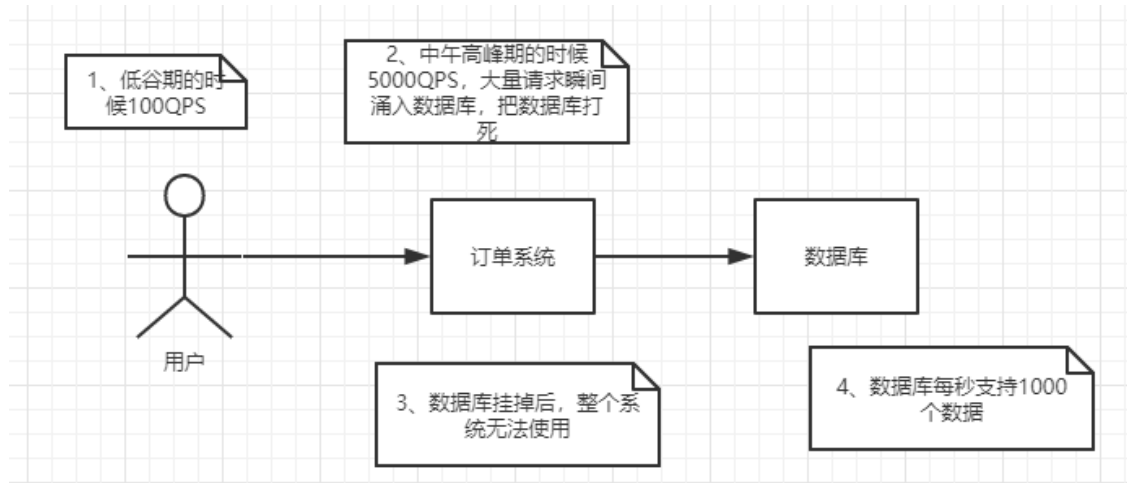
将不需要同步处理的并且耗时长操作由消息队列通知消息接收方进行异步处理。提高了应用程序的响应时间。

2、应用程序**解耦合**

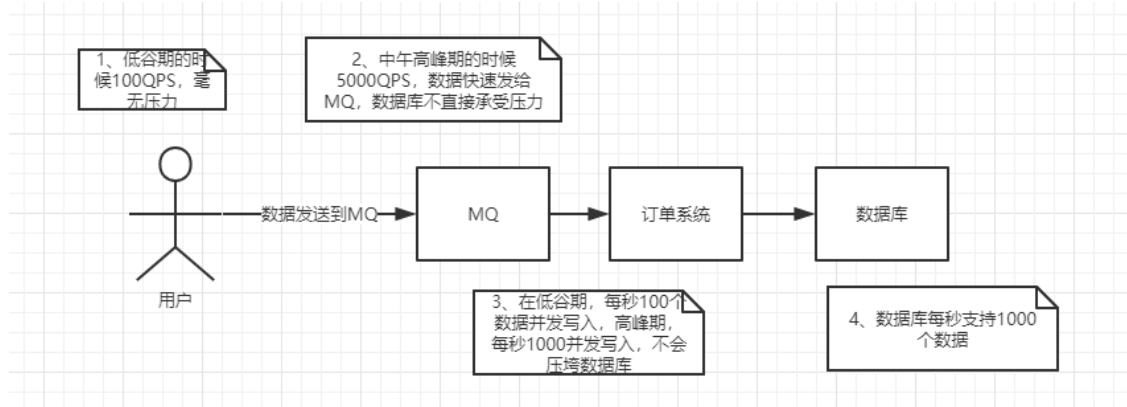
MQ相当于一个中介，生产方通过MQ与消费方交互，它将应用程序进行解耦合。

3、**削峰填谷**

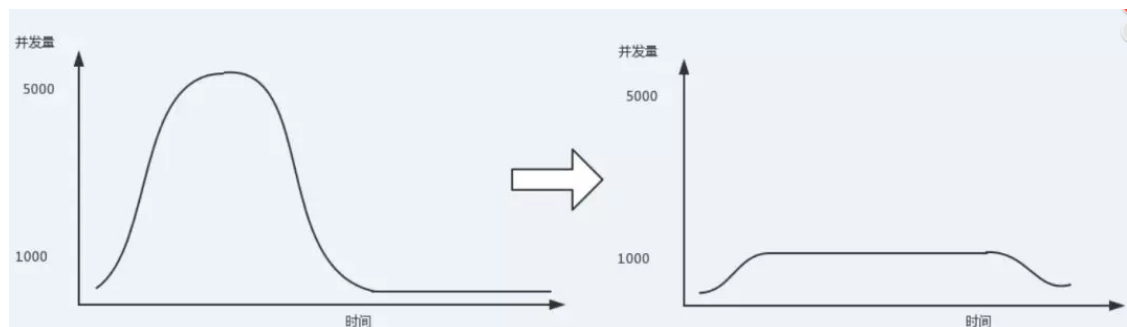
如订单系统，在下单的时候就会往数据库写数据。但是数据库只能支撑每秒1000左右的并发写入，并发量再高就容易宕机。低峰期的时候并发也就100多个，但是在高峰期时候，并发量会突然激增到5000以上，这个时候数据库肯定卡死了。



消息被MQ保存起来了，然后系统就可以按照自己的消费能力来消费，比如每秒1000个数据，这样慢慢写入数据库，这样就不会卡死数据库了。



但是使用了MQ之后，限制消费消息的速度为1000，但是这样一来，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1000QPS，直到消费完积压的消息，这就叫做“填谷”



1.2. AMQP 和 JMS

MQ是消息通信的模型；实现MQ的大致有两种主流方式：AMQP、JMS。

1.2.1. AMQP

AMQP是一种协议，更准确的说是一种binary wire-level protocol（链接协议）。这是其和JMS的本质差别，AMQP不从API层进行限定，而是直接定义网络交换的数据格式。

1.2.2. 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 (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。

1.2.3. AMQP 与 JMS 区别

- 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规定了两种消息模式；而AMQP的消息模式更加丰富

1.3. 消息队列产品

市场上常见的消息队列有如下：

- ActiveMQ：基于JMS
- ZeroMQ：基于C语言开发
- 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
- Kafka：类似MQ的产品；分布式消息系统，高吞吐量

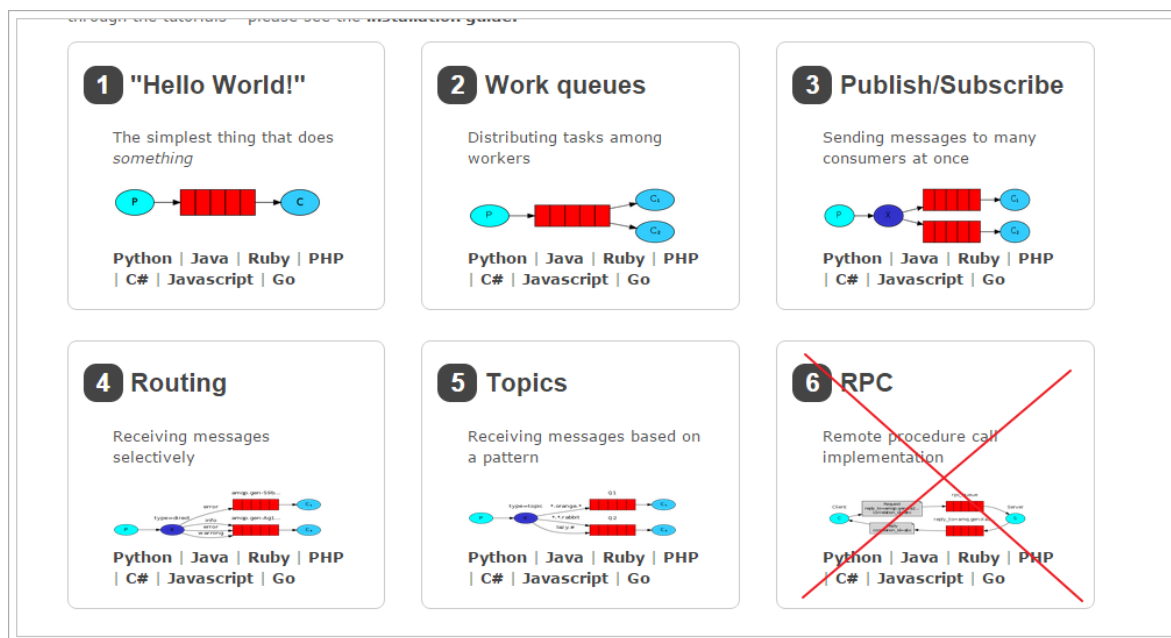
1.4. RabbitMQ

RabbitMQ是由erlang语言开发，基于AMQP (Advanced Message Queue 高级消息队列协议) 协议实现的消息队列，它是一种应用程序之间的通信方法，消息队列在分布式系统开发中应用非常广泛。

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

RabbitMQ提供了6种模式：简单模式，work模式，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>



2. 安装及配置RabbitMQ

2.1. 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
```

2.2. 安装Erlang

上传

```
erlang-18.3-1.el7.centos.x86_64.rpm  
socat-1.7.3.2-5.el7.linux.x86_64.rpm  
rabbitmq-server-3.6.5-1.noarch.rpm
```

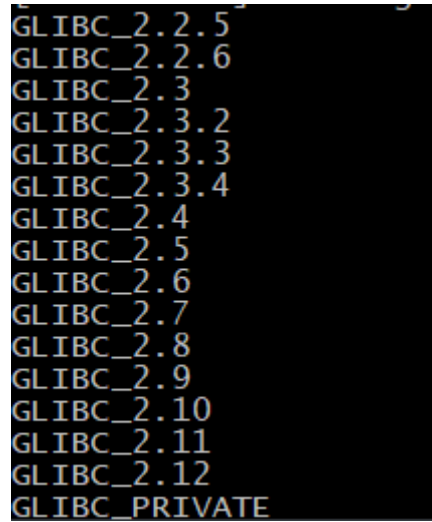
```
# 安装  
rpm -ivh erlang-18.3-1.el7.centos.x86_64.rpm
```

如果出现如下错误

```
warning: erlang-22.0.7-1.el7.x86_64.rpm: Header V4 RSA/SHA1 Signature, key ID 6026dfca: NOKEY  
error: Failed dependencies:  
libc.so.6(GLIBC_2.14)(64bit) is needed by erlang-22.0.7-1.el7.x86_64  
libc.so.6(GLIBC_2.15)(64bit) is needed by erlang-22.0.7-1.el7.x86_64  
libcrypto.so.10(OPENSSL_1.0.2)(64bit) is needed by erlang-22.0.7-1.el7.x86_64
```

说明glibc 版本太低。我们可以查看当前机器的glibc 版本

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



```
GLIBC_2.2.5
GLIBC_2.2.6
GLIBC_2.3
GLIBC_2.3.2
GLIBC_2.3.3
GLIBC_2.3.4
GLIBC_2.4
GLIBC_2.5
GLIBC_2.6
GLIBC_2.7
GLIBC_2.8
GLIBC_2.9
GLIBC_2.10
GLIBC_2.11
GLIBC_2.12
GLIBC_PRIVATE
```

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

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

下载rpm包

```
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/glibc-utils-2.17-55.el6.x86_64.rpm &
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/glibc-static-2.17-55.el6.x86_64.rpm &
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/glibc-2.17-55.el6.x86_64.rpm &
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/glibc-common-2.17-55.el6.x86_64.rpm &
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/glibc-devel-2.17-55.el6.x86_64.rpm &
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/glibc-headers-2.17-55.el6.x86_64.rpm &
wget http://copr-be.cloud.fedoraproject.org/results/mosquito/myrepo-el6/epel-6-
x86_64/glibc-2.17-55.fc20/nscd-2.17-55.el6.x86_64.rpm &
```

安装rpm包

```
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

```
GLIBC_2.2.5
GLIBC_2.2.6
GLIBC_2.3
GLIBC_2.3.2
GLIBC_2.3.3
GLIBC_2.3.4
GLIBC_2.4
GLIBC_2.5
GLIBC_2.6
GLIBC_2.7
GLIBC_2.8
GLIBC_2.9
GLIBC_2.10
GLIBC_2.11
GLIBC_2.12
GLIBC_2.13
GLIBC_2.14
GLIBC_2.15
GLIBC_2.16
GLIBC_2.17
GLIBC_PRIVATE
```

2.3. 安装RabbitMQ

```
# 安装
rpm -ivh socat-1.7.3.2-5.el7.linux.x86_64.rpm

# 安装
rpm -ivh rabbitmq-server-3.6.5-1.noarch.rpm
```

2.4. 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

# 比如修改密码、配置等等，例如：loopback_users 中的 <<"guest">>,只保留guest
```

2.5. 启动

```
service rabbitmq-server start # 启动服务
service rabbitmq-server stop # 停止服务
service rabbitmq-server restart # 重启服务
```

设置配置文件

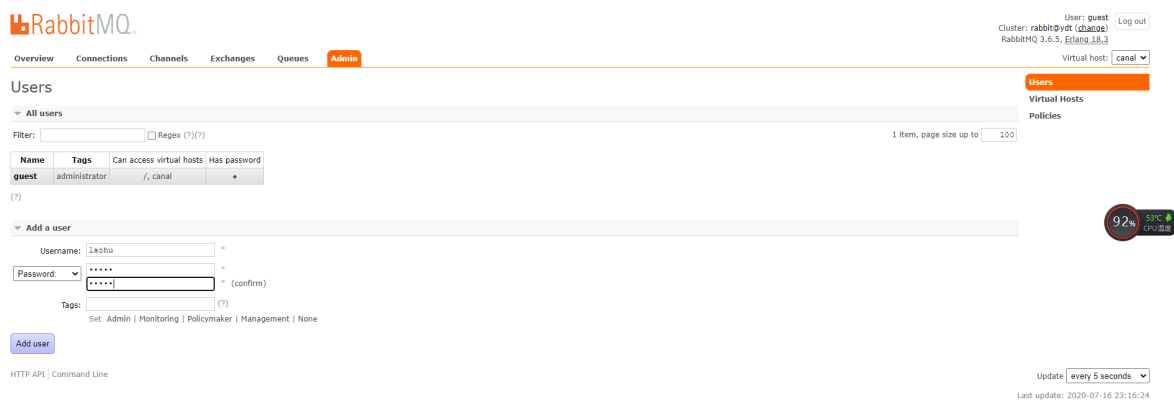
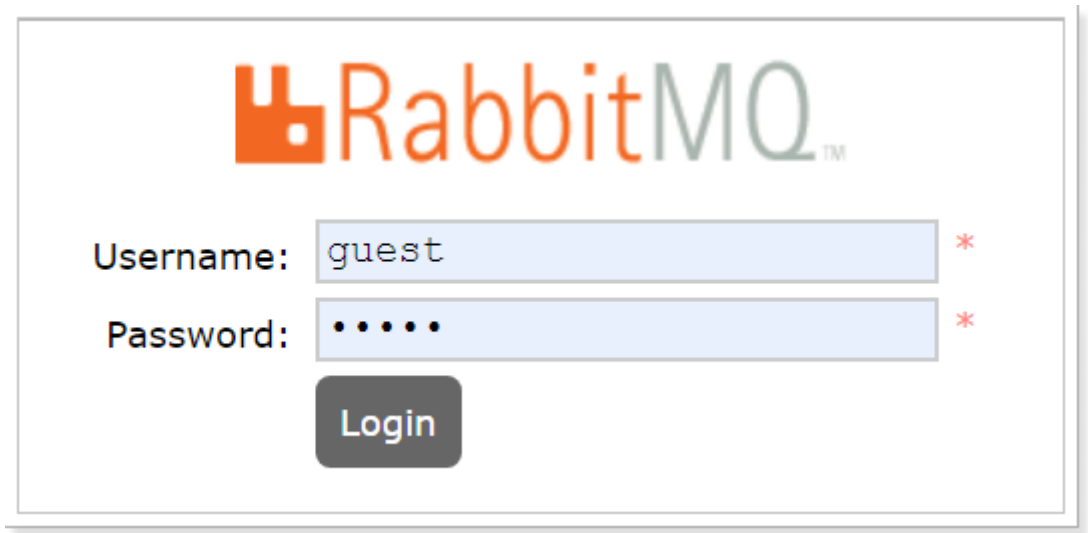
```
cd /usr/share/doc/rabbitmq-server-3.6.5/

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

2.6. 配置虚拟主机及用户

2.6.1. 用户角色

RabbitMQ在安装好后，可以访问 `http://ip地址:15672`；其自带了guest/guest的用户名和密码；如果需要创建自定义用户；那么也可以登录管理界面后，如下操作：



角色说明：

1、超级管理员(administrator)

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

2、监控者(monitoring)

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

3、策略制定者(policymaker)

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

4、普通管理者(management)

仅可登陆管理控制台，无法看到节点信息，也无法对策略进行管理。

5、其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

2.6.2. Virtual Hosts配置

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VirtualHost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。

192.168.223.128:15672/#/vhosts

RabbitMQ

Overview Connections Channels Exchanges Queues Admin

Virtual Hosts

Filter: ☐ Regexp (?)

2 items, page size up to 100

Overview		Messages			Network		Message rates		
Name	Users (?)	Ready	Unacked	Total	From client	To client	publish	deliver	/ get
/	guest	0	0	0					
canal	guest	0	0	0					

Add a new virtual host

Name:

Add virtual host

HTTP API | Command Line

Update: every 5 seconds

Last update: 2020-07-16 23:18:23

Users

Virtual Hosts

Policies

93% 37% 40% 100%

Current permissions

... no permissions ...

Set permission

User

Configure regexp:

Write regexp:

Read regexp:

Set permission

3. RabbitMQ入门

3.1. 添加依赖

```
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.9.0</version>
</dependency>
```

3.2 编写连接工具类

```
package com.ydt.rabbitmq;

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

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class ConnectionUtil {
```

```

    public static Connection getConnection() throws IOException,
TimeoutException {
        //创建连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //主机地址;默认为 localhost
        connectionFactory.setHost("192.168.223.128");
        //连接端口;默认为 5672
        connectionFactory.setPort(5672);
        //虚拟主机名称;默认为 /
        connectionFactory.setVirtualHost("ydt");
        //连接用户名;默认为guest
        connectionFactory.setUsername("ydt");
        //连接密码;默认为guest
        connectionFactory.setPassword("ydt");

        //创建连接
        Connection connection = connectionFactory.newConnection();
        return connection;
    }
}

```

3.3. 编写生产者

```

package com.ydt.rabbitmq.simple;

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

import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class Producer {

    //队列名称
    public static String QUEUE_NAME = "simple_queue";

    public static void main(String[] args) throws IOException, TimeoutException
    {

        Connection connection = ConnectionUtil.getConnection();
        // 创建频道
        Channel channel = connection.createChannel();

        // 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
         */
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);
    }
}

```

```

// 要发送的信息
String message = "你好; 小兔子! ";
/**
 * 参数1: 交换机名称, 如果没有指定则使用默认Default Exchage
 * 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
 */
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
System.out.println("已发送消息: " + message);

// 关闭资源
channel.close();
connection.close();
}
}

```

在执行上述的消息发送之后; 可以登录rabbitMQ的管理控制台, 可以发现队列和其消息:



Overview
Connections
Channels
Exchanges
Queues
Admin

Queues

▼ All queues (1)

Pagination

Page 1 of 1 - Filter: ☐ Regex (?)

Overview				Messages			Message rates		
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
ydt	simple_queue	D	idle	1	0	1	0.00/s	0.00/s	0.00/s

▼ Add a new queue

Virtual host: /
Name:
Durability: Durable
Auto delete: (?) No
Arguments: = String

Add Message TTL (?) | Auto expire (?) | Max length (?) | Max length bytes (?)
Dead letter exchange (?) | Dead letter routing key (?) | Maximum priority (?)

Add queue

Overview
Connections
Channels
Exchanges
Queues
Admin

Warning: getting messages from a queue is a destructive action. ?

Ack Mode:

Encoding: ?

Messages:

Get Message(s)

Message 1

The server reported 0 messages remaining.

Exchange	(AMQP default)
Routing Key	simple_queue
Redelivered	o
Properties	
Payload	你好; 小兔子!
21 bytes	
Encoding: string	

3.4. 编写消费者

```

package com.ydt.rabbitmq.simple;

import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        // 创建频道
        Channel channel = connection.createChannel();
        // 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
         */
        channel.queueDeclare(Producer.QUEUE_NAME, true, false, false, null);

        //创建消费者; 并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签, 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
             */
        }
    }

```

```

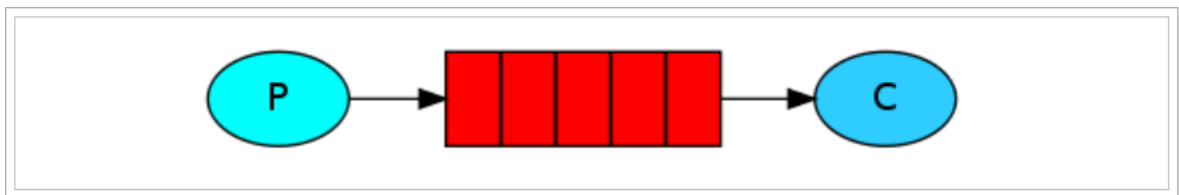
        public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
            //路由key
            System.out.println("路由key为: " + envelope.getRoutingKey());
            //交换机
            System.out.println("交换机为: " + envelope.getExchange());
            //消息id
            System.out.println("消息id为: " + envelope.getDeliveryTag());
            //收到的消息
            System.out.println("接收到的消息为: " + new String(body, "utf-8"));
        }
    };
    //监听消息
    /**
     * 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(Producer.QUEUE_NAME, true, consumer);

    //不关闭资源, 应该一直监听消息
    //channel.close();
    //connection.close();
}
}

```

3.4. 小结

上述的入门案例中其实使用的是如下的简单模式：



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。

4. AMQP

4.1. 相关概念介绍

AMQP 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。

AMQP是一个二进制协议，拥有一些现代化特点：多信道、协商式，异步，安全，扩平台，中立，高效。

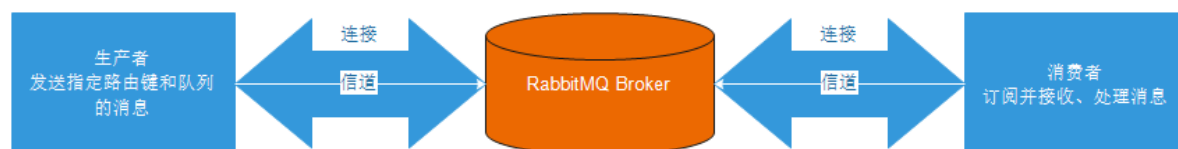
RabbitMQ是AMQP协议的Erlang的实现。

概念	说明
连接 Connection	一个网络连接，比如TCP/IP套接字连接。
会话Session	端点之间的命名对话。在一个会话上下文中，保证“恰好传递一次”。
信道 Channel	多路复用连接中的一条独立的双向数据流通道。为会话提供物理传输介质。
客户端 Client	AMQP连接或者会话的发起者。AMQP是非对称的，客户端生产和消费消息，服务器存储和路由这些消息。
服务节点 Broker	消息中间件的服务节点；一般情况下可以将一个RabbitMQ Broker看作一台RabbitMQ 服务器。
端点	AMQP对话的任意一方。一个AMQP连接包括两个端点（一个是客户端，一个是服务器）。
消费者 Consumer	一个从消息队列里请求消息的客户端程序。
生产者 Producer	一个向交换机发布消息的客户端应用程序。

4.2. RabbitMQ运转流程

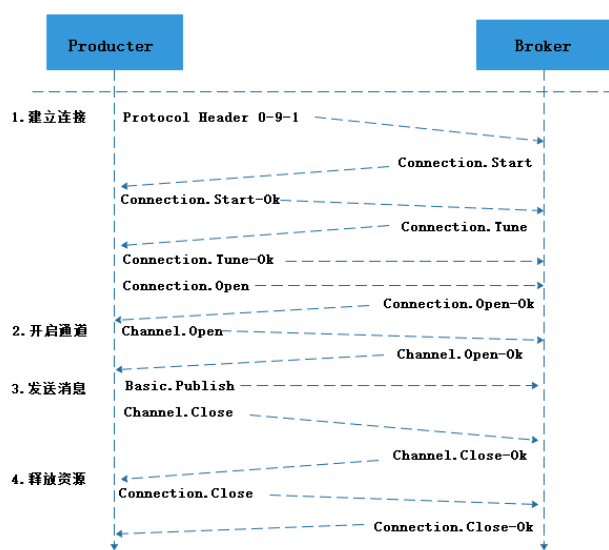
在入门案例中：

- 生产者发送消息
 - 生产者创建连接（Connection），开启一个信道（Channel），连接到RabbitMQ Broker；
 - 声明队列并设置属性；如是否排它，是否持久化，是否自动删除；
 - 将路由键（空字符串）与队列绑定起来；
 - 发送消息至RabbitMQ Broker；
 - 关闭信道；
 - 关闭连接；
- 消费者接收消息
 - 消费者创建连接（Connection），开启一个信道（Channel），连接到RabbitMQ Broker
 - 向Broker 请求消费相应队列中的消息，设置相应的回调函数；
 - 等待Broker回应并投递响应队列中的消息，消费者接收消息；
 - 确认（ack，自动确认）接收到的消息；
 - RabbitMQ从队列中删除相应已经被确认的消息；
 - 关闭信道；
 - 关闭连接；



4.3. 生产者流转过程说明

1. 客户端与代理服务器Broker建立连接。会调用newConnection() 方法,这个方法会进一步封装 Protocol Header 0-9-1 的报文头发送给Broker , 以此通知Broker 本次交互采用的是AMQPO-9-1 协议, 紧接着Broker 返回Connection.Start 来建立连接, 在连接的过程中涉及 Connection.Start/.Start-Ok 、 Connection.Tune/.Tune-Ok , Connection.Open/ .Open-Ok 这6 个命令的交互。
2. 客户端调用connection.createChannel方法。此方法开启信道, 其包装的channel.open命令发送给Broker,等待channel.basicPublish方法, 对应的AMQP命令为Basic.Publish,这个命令包含了 content Header 和content Body()。content Header 包含了消息体的属性, 例如:投递模式, 优先级等, content Body 包含了消息体本身。
3. 客户端发送完消息需要关闭资源时, 涉及到Channel.Close和Channl.Close-Ok 与 Connetion.Close和Connection.Close-Ok的命令交互。

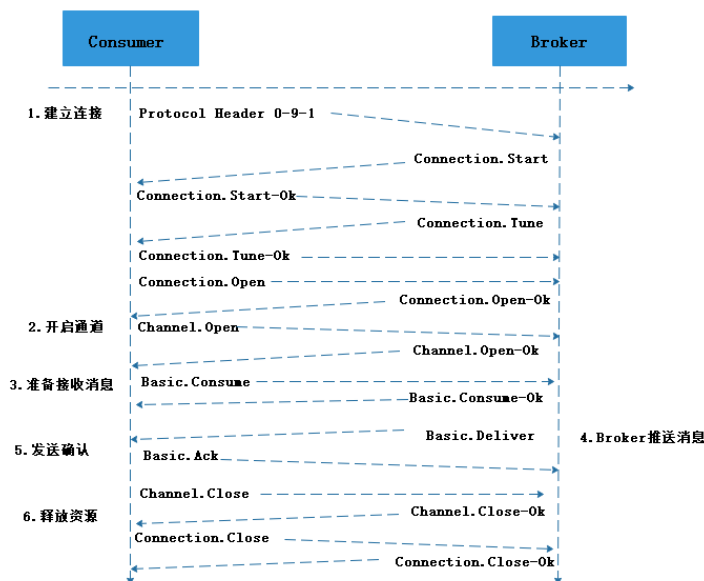


生产者流转过程

4.4. 消费者流转过程说明

1. 消费者客户端与代理服务器Broker建立连接。会调用newConnection() 方法,这个方法会进一步封装 Protocol Header 0-9-1 的报文头发送给Broker , 以此通知Broker 本次交互采用的是AMQPO-9-1 协议, 紧接着Broker 返回Connection.Start 来建立连接, 在连接的过程中涉及 Connection.Start/.Start-Ok 、 Connection.Tune/.Tune-Ok , Connection.Open/ .Open-Ok 这6 个命令的交互。
2. 消费者客户端调用connection.createChannel方法。和生产者客户端一样, 协议涉及Channel . Open/Open-Ok命令。
3. 在真正消费之前, 消费者客户端需要向Broker 发送Basic.Consume 命令(即调用 channel.basicConsume 方法) 将Channel 置为接收模式, 之后Broker 回执Basic . Consume - Ok 以告诉消费者客户端准备好消费消息。
4. Broker 向消费者客户端推送(Push) 消息, 即Basic.Deliver 命令, 这个命令和Basic.Publish 命令一样会携带Content Header 和Content Body。
5. 消费者接收到消息并正确消费之后, 向Broker 发送确认, 即Basic.Ack 命令。

6. 客户端发送完消息需要关闭资源时，涉及到Channel.Close和Channel.Close-Ok 与 Connection.Close和Connection.Close-Ok的命令交互。

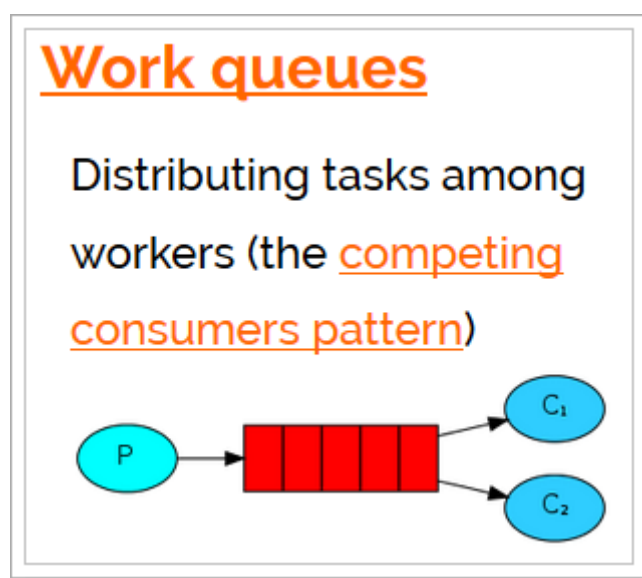


消费者流转过程

5. RabbitMQ工作模式

4.1. Work queues工作队列模式

4.1.1. 模式说明



work queues 与入门程序的简单模式相比，多了一个或一些消费端，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。

4.1.2. 代码

work Queues 与入门程序的简单模式的代码是几乎一样的；可以完全复制，并复制多一个消费者进行多个消费者同时消费消息的测试。

1) 生产者

```
package com.ydt.rabbitmq.work;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.ydt.rabbitmq.util.ConnectionUtil;

public class Producer {

    static final String QUEUE_NAME = "work_queue";

    public static void main(String[] args) throws Exception {

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        // 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
         */
        channel.queueDeclare(QUEUE_NAME, true, false, false, null);

        for (int i = 1; i <= 30; i++) {
            // 发送信息
            String message = "你好：小兔子！work模式--" + i;
            /**
             * 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
             */
            channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
            System.out.println("已发送消息：" + message);
        }

        // 关闭资源
        channel.close();
        connection.close();
    }
}
```

2) 消费者1

```
package com.ydt.rabbitmq.work;

import com.rabbitmq.client.*;
import com.ydt.rabbitmq.util.ConnectionUtil;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        final Channel channel = connection.createChannel();

        // 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
         */
        channel.queueDeclare(Producer.QUEUE_NAME, true, false, false, null);

        //一次只能接收并处理一个消息
        channel.basicQos(1);

        //创建消费者；并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签，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
             */
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                try {
                    //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
                    System.out.println("消费者1-接收到的消息为: " + new String(body,
                        "utf-8"));

                    Thread.sleep(1000);

                    //确认消息
                    channel.basicAck(envelope.getDeliveryTag(), false);
                } catch (InterruptedException e) {
```

```

        e.printStackTrace();
    }
}
};
//监听消息
/**
 * 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(Producer.QUEUE_NAME, false, consumer);
}
}

```

3) 消费者2

```

package com.ydt.rabbitmq.work;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        final Channel channel = connection.createChannel();

        // 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
         */
        channel.queueDeclare(Producer.QUEUE_NAME, true, false, false, null);

        //一次只能接收并处理一个消息
        channel.basicQos(1);

        //创建消费者; 并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签, 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
             */
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {

```

```

try {
    //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
    System.out.println("消费者2-接收到的消息为: " + new String(body,
"utf-8"));

    Thread.sleep(1000);

    //确认消息
    channel.basicAck(envelope.getDeliveryTag(), false);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
};
//监听消息
/**
 * 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(Producer.QUEUE_NAME, false, consumer);
}
}

```

4.1.3. 测试

启动两个消费者, 然后再启动生产者发送消息; 到IDEA的两个消费者对应的控制台查看是否竞争性的接收到消息。

```

Consumer1 x Consumer2 x Producer (1) x
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder
路由key为: work_queue
交换机为:
消息id为: 1
消费者1-接收到的消息为: 你好; 小兔子! work模式--1
路由key为: work_queue
交换机为:
消息id为: 2
消费者1-接收到的消息为: 你好; 小兔子! work模式--3
路由key为: work_queue
交换机为:
消息id为: 3
消费者1-接收到的消息为: 你好; 小兔子! work模式--5
路由key为: work_queue
交换机为:
消息id为: 4

```

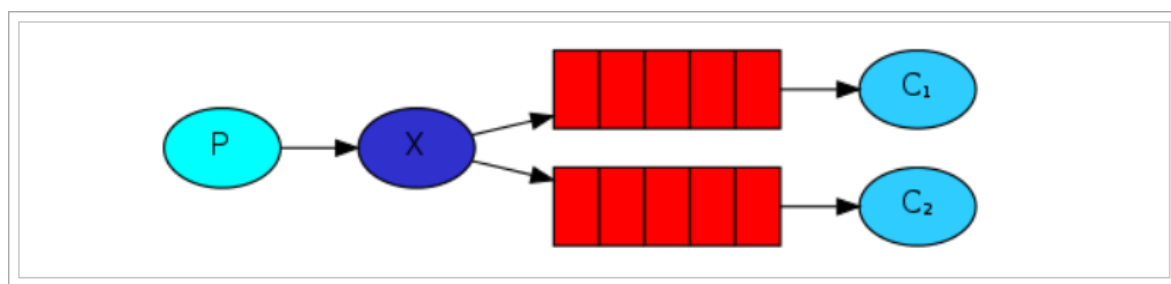
```
Consumer1 x Consumer2 x Producer (1) x
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinde
路由key为: work_queue
交换机为:
消息id为: 1
消费者2-接收到的消息为: 你好; 小兔子! work模式--2
路由key为: work_queue
交换机为:
消息id为: 2
消费者2-接收到的消息为: 你好; 小兔子! work模式--4
路由key为: work_queue
交换机为:
消息id为: 3
消费者2-接收到的消息为: 你好; 小兔子! work模式--6
路由key为: work_queue
```

4.1.4. 小结

在一个队列中如果有多个消费者，那么消费者之间对于同一个消息的关系是**竞争**的关系。

4.2. 订阅模式类型

订阅模式示例图：



前面2个案例中，只有3个角色：

- 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

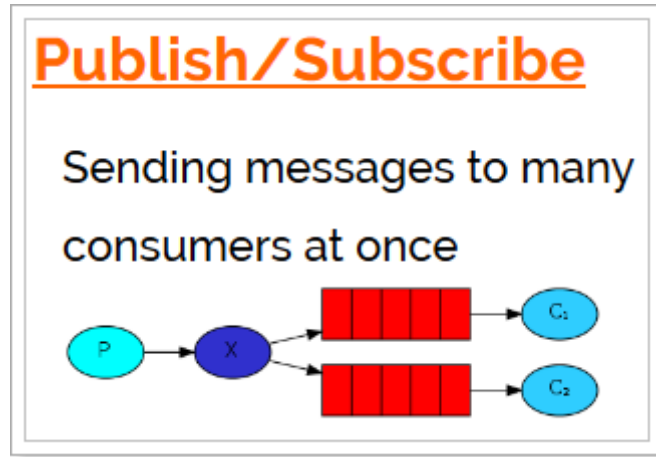
而在订阅模型中，多了一个exchange角色，而且过程略有变化：

- 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，消息的接受者，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，图中的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：
 - Fanout：广播，将消息交给所有绑定到交换机的队列
 - Direct：定向，把消息交给符合指定routing key 的队列
 - 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！

4.3. Publish/Subscribe发布与订阅模式

4.3.1. 模式说明



发布订阅模式：

- 1、每个消费者监听自己的队列。
- 2、生产者将消息发给broker，由交换机将消息转发到绑定此交换机的每个队列，每个绑定交换机的队列都将接收到消息

4.3.2. 代码

1) 生产者

```
package com.ydt.rabbitmq.ps;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;

/**
 * 发布与订阅使用的交换机类型为：fanout
 */
public class Producer {

    //交换机名称
    static final String FANOUT_EXCHANGE = "fanout_exchange";
    //队列名称
    static final String FANOUT_QUEUE_1 = "fanout_queue_1";
    //队列名称
    static final String FANOUT_QUEUE_2 = "fanout_queue_2";

    public static void main(String[] args) throws Exception {

        //创建连接
```

```

Connection connection = ConnectionUtil.getConnection();

// 创建频道
Channel channel = connection.createChannel();

/**
 * 声明交换机
 * 参数1: 交换机名称
 * 参数2: 交换机类型, fanout、topic、direct、headers
 */
channel.exchangeDeclare(FANOUT_EXCHANGE, BuiltinExchangeType.FANOUT);

// 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
 */
channel.queueDeclare(FANOUT_QUEUE_1, true, false, false, null);
channel.queueDeclare(FANOUT_QUEUE_2, true, false, false, null);

//队列绑定交换机
channel.queueBind(FANOUT_QUEUE_1, FANOUT_EXCHANGE, "");
channel.queueBind(FANOUT_QUEUE_2, FANOUT_EXCHANGE, "");

for (int i = 1; i <= 10; i++) {
    // 发送信息
    String message = "你好：小兔子！发布订阅模式--" + i;
    /**
     * 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
     */
    channel.basicPublish(FANOUT_EXCHANGE, "", null, message.getBytes());
    System.out.println("已发送消息: " + message);
}

// 关闭资源
channel.close();
connection.close();
}
}

```

2) 消费者1

```

package com.ydt.rabbitmq.ps;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

```

```

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.FANOUT_EXCHANGE,
        BuiltinExchangeType.FANOUT);

        // 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
         */
        channel.queueDeclare(Producer.FANOUT_QUEUE_1, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(Producer.FANOUT_QUEUE_1, Producer.FANOUT_EXCHANGE, "");

        //创建消费者；并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签，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
             */
            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
                System.out.println("消费者1-接收到的消息为: " + new String(body,
                "utf-8"));
            }
        };

        //监听消息
        /**
         * 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(Producer.FANOUT_QUEUE_1, true, consumer);
    }
}

```


3) 消费者2

```
package com.ydt.rabbitmq.ps;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.FANOUT_EXCHANGE,
        BuiltinExchangeType.FANOUT);

        // 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
         */
        channel.queueDeclare(Producer.FANOUT_QUEUE_2, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(Producer.FANOUT_QUEUE_2, Producer.FANOUT_EXCHANGE, "");

        //创建消费者；并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签，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
             */
            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
                System.out.println("消费者2-接收到的消息为: " + new String(body,
                "utf-8"));
            }
        };
    }
}
```

```

    }
};
//监听消息
/**
 * 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(Producer.FANOUT_QUEUE_2, true, consumer);
}
}

```

4.3.3. 测试

启动所有消费者, 然后使用生产者发送消息; 在每个消费者对应的控制台可以查看到生产者发送的所有消息; 到达广播的效果。

在执行完测试代码后, 其实到RabbitMQ的管理后台找到 Exchanges 选项卡, 点击 fanout_exchange 的交换机, 可以查看到如下的绑定:

The screenshot shows the RabbitMQ management UI. The 'Exchanges' tab is selected and highlighted with a red box. Below the tabs, the 'Details' section for 'fanout_exchange' is visible. The 'Bindings' section is circled in red and shows two bindings:

To	Routing key	Arguments	
fanout_queue_1			Unbind
fanout_queue_2			Unbind

4.3.4. 小结

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。

发布订阅模式与工作队列模式的区别

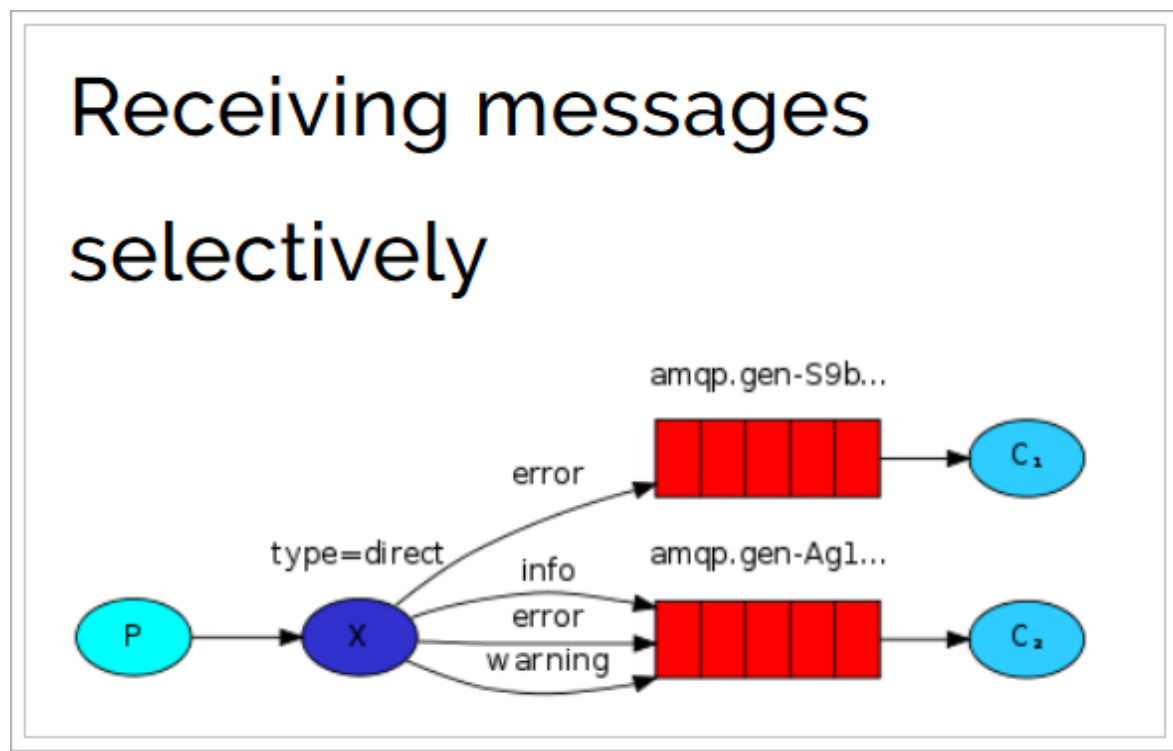
- 1、工作队列模式不用定义交换机, 而发布/订阅模式需要定义交换机。
- 2、发布/订阅模式的生产方是面向交换机发送消息, 工作队列模式的生产方是面向队列发送消息(底层使用默认交换机)。
- 3、发布/订阅模式需要设置队列和交换机的绑定, 工作队列模式不需要设置, 实际上工作队列模式会将队列绑定到默认的交换机。

4.4. Routing路由模式

4.4.1. 模式说明

路由模式特点：

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



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

4.4.2. 代码

在编码上与 `Publish/Subscribe` 发布与订阅模式 的区别是交换机的类型为：Direct，还有队列绑定交换机的时候需要指定routing key。

1) 生产者

```
package com.ydt.rabbitmq.routing;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.BuiltinExchangeType;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;

/**
```

```

* 路由模式的交换机类型为: direct
*/
public class Producer {

    //交换机名称
    static final String DIRECT_EXCHANGE = "direct_exchange";
    //队列名称
    static final String DIRECT_QUEUE_INSERT = "direct_queue_insert";
    //队列名称
    static final String DIRECT_QUEUE_UPDATE = "direct_queue_update";

    public static void main(String[] args) throws Exception {

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        /**
         * 声明交换机
         * 参数1: 交换机名称
         * 参数2: 交换机类型, fanout、topic、direct、headers
         */
        channel.exchangeDeclare(DIRECT_EXCHANGE, BuiltinExchangeType.DIRECT);

        // 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
         */
        channel.queueDeclare(DIRECT_QUEUE_INSERT, true, false, false, null);
        channel.queueDeclare(DIRECT_QUEUE_UPDATE, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(DIRECT_QUEUE_INSERT, DIRECT_EXCHANGE, "insert");
        channel.queueBind(DIRECT_QUEUE_UPDATE, DIRECT_EXCHANGE, "update");

        // 发送信息
        String message = "新增了商品。路由模式: routing key 为 insert ";
        /**
         * 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
         */
        channel.basicPublish(DIRECT_EXCHANGE, "insert", null,
            message.getBytes());
        System.out.println("已发送消息: " + message);

        // 发送信息
        message = "修改了商品。路由模式: routing key 为 update ";
        /**
         * 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
        */
        channel.basicPublish(DIRECT_EXCHANGE, "update", null,
message.getBytes());
        System.out.println("已发送消息: " + message);

        // 关闭资源
        channel.close();
        connection.close();
    }
}

```

2) 消费者1

```

package com.ydt.rabbitmq.routing;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.DIRECT_EXCHANGE,
BuiltinExchangeType.DIRECT);

        // 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
        */
        channel.queueDeclare(Producer.DIRECT_QUEUE_INSERT, true, false, false,
null);

        //队列绑定交换机
        channel.queueBind(Producer.DIRECT_QUEUE_INSERT, Producer.DIRECT_EXCHANGE,
"insert");

        //创建消费者; 并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
            * consumerTag 消息者标签, 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
        */
        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
            System.out.println("消费者1-接收到的消息为: " + new String(body,
"utf-8"));
        }
    };
    //监听消息
    /**
     * 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(Producer.DIRECT_QUEUE_INSERT, true, consumer);
}
}

```

3) 消费者2

```

package com.ydt.rabbitmq.routing;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.DIRECT_EXCHANGE,
BuiltinExchangeType.DIRECT);

        // 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
         */
    }
}

```

```

channel.queueDeclare(Producer.DIRECT_QUEUE_UPDATE, true, false, false,
null);

//队列绑定交换机
channel.queueBind(Producer.DIRECT_QUEUE_UPDATE, Producer.DIRECT_EXCHANGE,
"update");

//创建消费者；并设置消息处理
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override
    /**
     * consumerTag 消息者标签，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
     */
    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
        System.out.println("消费者2-接收到的消息为: " + new String(body,
"utf-8"));
    }
};

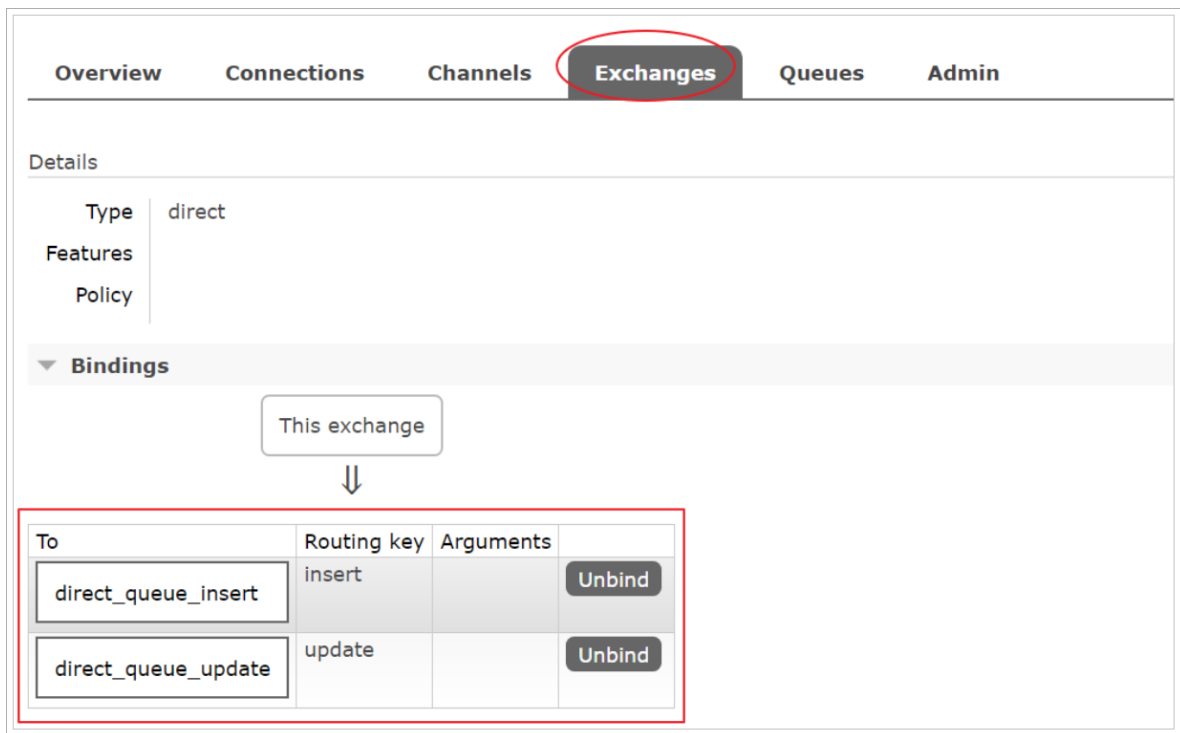
//监听消息
/**
 * 参数1: 队列名称
 * 参数2: 是否自动确认，设置为true为表示消息接收到自动向mq回复接收到了，mq接收到回复
会删除消息，设置为false则需要手动确认
 * 参数3: 消息接收到后回调
 */
channel.basicConsume(Producer.DIRECT_QUEUE_UPDATE, true, consumer);
}
}

```

4.4.3. 测试

启动所有消费者，然后使用生产者发送消息；在消费者对应的控制台可以查看到生产者发送对应 routing key 对应队列的消息；到达**按照需要接收**的效果。

在执行完测试代码后，其实到RabbitMQ的管理后台找到 `Exchanges` 选项卡，点击 `direct_exchange` 的交换机，可以查看到如下的绑定：



4.4.4. 小结

Routing模式要求队列在绑定交换机时要指定routing key，消息会转发到符合routing key的队列。

4.5. Topics通配符模式

4.5.1. 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

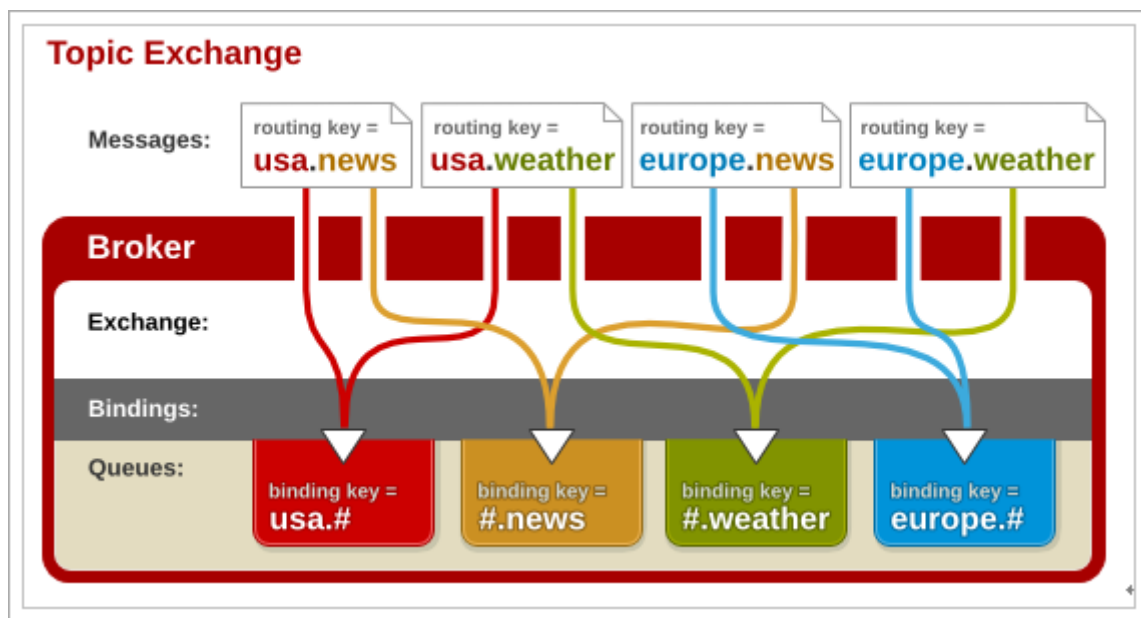
*：匹配不多不少恰好1个词

举例：

`item.#`：能够匹配`item.insert.abc`或者`item.insert`

`item.*`：只能匹配`item.insert`

Receiving messages based on a pattern (topics)



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

4.5.2. 代码

1) 生产者

使用topic类型的Exchange，发送消息的routing key有3种：`item.insert`、`item.update`、`item.delete`：

```
package com.ydt.rabbitmq.topic;

import com.ydt.rabbitmq.util.ConnectionUtil;
```

```

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

/**
 * 通配符Topic的交换机类型为: topic
 */
public class Producer {

    //交换机名称
    static final String TOPIC_EXCHANGE = "topic_exchange";
    //队列名称
    static final String TOPIC_QUEUE_1 = "topic_queue_1";
    //队列名称
    static final String TOPIC_QUEUE_2 = "topic_queue_2";

    public static void main(String[] args) throws Exception {

        //创建连接
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        /**
         * 声明交换机
         * 参数1: 交换机名称
         * 参数2: 交换机类型, fanout、topic、topic、headers
         */
        channel.exchangeDeclare(TOPIC_EXCHANGE, BuiltinExchangeType.TOPIC);

        // 发送信息
        String message = "新增了商品。Topic模式; routing key 为 item.insert ";
        channel.basicPublish(TOPIC_EXCHANGE, "item.insert", null,
message.getBytes());
        System.out.println("已发送消息: " + message);

        // 发送信息
        message = "修改了商品。Topic模式; routing key 为 item.update" ;
        channel.basicPublish(TOPIC_EXCHANGE, "item.update", null,
message.getBytes());
        System.out.println("已发送消息: " + message);

        // 发送信息
        message = "删除了商品。Topic模式; routing key 为 item.delete" ;
        channel.basicPublish(TOPIC_EXCHANGE, "item.delete", null,
message.getBytes());
        System.out.println("已发送消息: " + message);

        // 关闭资源
        channel.close();
        connection.close();
    }
}

```

2) 消费者1

接收两种类型的消息：更新商品和删除商品

```
package com.ydt.rabbitmq.topic;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer1 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.TOPIC_EXCHANGE,
        BuiltinExchangeType.TOPIC);

        // 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
         */
        channel.queueDeclare(Producer.TOPIC_QUEUE_1, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(Producer.TOPIC_QUEUE_1, Producer.TOPIC_EXCHANGE,
        "item.update");
        channel.queueBind(Producer.TOPIC_QUEUE_1, Producer.TOPIC_EXCHANGE,
        "item.delete");

        //创建消费者：并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            /**
             * consumerTag 消息者标签，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
             */
            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
        System.out.println("消费者1-接收到的消息为: " + new String(body,
"utf-8"));
    }
};
//监听消息
/**
 * 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(Producer.TOPIC_QUEUE_1, true, consumer);
}
}

```

3) 消费者2

接收所有类型的消息：新增商品，更新商品和删除商品。

```

package com.ydt.rabbitmq.topic;

import com.ydt.rabbitmq.util.ConnectionUtil;
import com.rabbitmq.client.*;

import java.io.IOException;

public class Consumer2 {

    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();

        // 创建频道
        Channel channel = connection.createChannel();

        //声明交换机
        channel.exchangeDeclare(Producer.TOPIC_EXCHANGE,
BuiltinExchangeType.TOPIC);

        // 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
         */
        channel.queueDeclare(Producer.TOPIC_QUEUE_2, true, false, false, null);

        //队列绑定交换机
        channel.queueBind(Producer.TOPIC_QUEUE_2, Producer.TOPIC_EXCHANGE,
"item.*");

        //创建消费者；并设置消息处理
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override

```

```

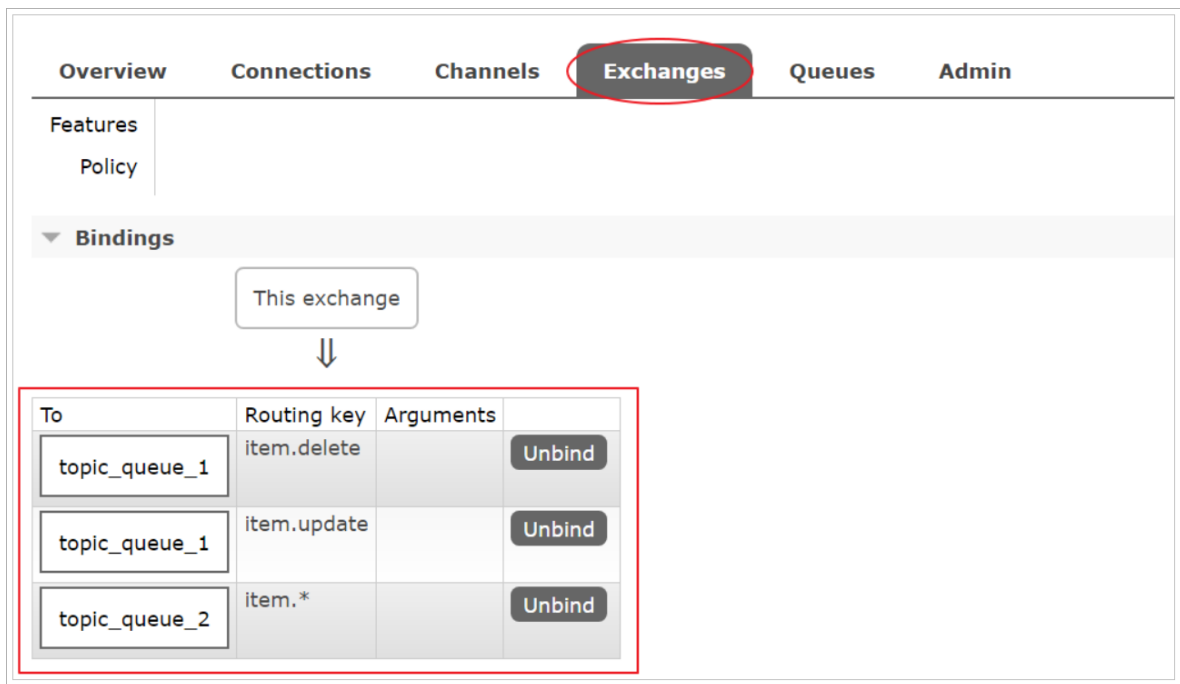
        /**
         * consumerTag 消息者标签, 在channel.basicConsume时候可以指定
         * envelope 消息包的内容, 可从中获取消息id, 消息routingkey, 交换机, 消息和重
        传标志(收到消息失败后是否需要重新发送)
         * properties 属性信息
         * body 消息
        */
        public void handleDelivery(String consumerTag, Envelope envelope,
        AMQP.BasicProperties 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
            System.out.println("消费者2-接收到的消息为: " + new String(body,
            "utf-8"));
        }
    };
    //监听消息
    /**
     * 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(Producer.TOPIC_QUEUE_2, true, consumer);
}
}

```

4.5.3. 测试

启动所有消费者, 然后使用生产者发送消息; 在消费者对应的控制台可以查看到生产者发送对应 routing key对应队列的消息; 到达**按照需要接收**的效果; 并且这些routing key可以使用通配符。

在执行完测试代码后, 其实到RabbitMQ的管理后台找到 Exchanges 选项卡, 点击 `topic_exchange` 的交换机, 可以查看到如下的绑定:



4.5.4. 小结

Topic主题模式可以实现 **Publish/Subscribe**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。

4.6. 模式总结

RabbitMQ工作模式：

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）

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）

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

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

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

5. Spring 整合RabbitMQ

5.1. 搭建生产者工程

5.1.1. 添加依赖

修改pom.xml文件内容为如下：

```

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>2.2.6.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

5.1.2. 配置整合

1. 创建rabbitmq.properties`连接参数等配置文件;

```

rabbitmq.host=192.168.223.128
rabbitmq.port=5672
rabbitmq.username=ydt
rabbitmq.password=ydt
rabbitmq.virtual-host=ydt

```

2. 创建 spring-rabbitmq.xml整合配置文件

```

<?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: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}"/>

<!--定义管理交换机、队列-->
<rabbit:admin connection-factory="connectionFactory"/>

<!--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
-->
<rabbit:queue id="spring_queue" name="spring_queue" auto-declare="true"/>

<!-- ~~~~~广播：所有队列都能收到消息
~~~~~ -->
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_fanout_queue_1" name="spring_fanout_queue_1" auto-
declare="true"/>

<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_fanout_queue_2" name="spring_fanout_queue_2" auto-
declare="true"/>

<!--定义广播类型交换机；并绑定上述两个队列-->
<rabbit:fanout-exchange id="spring_fanout_exchange"
name="spring_fanout_exchange" auto-declare="true">
    <rabbit:bindings>
        <rabbit:binding queue="spring_fanout_queue_1"/>
        <rabbit:binding queue="spring_fanout_queue_2"/>
    </rabbit:bindings>
</rabbit:fanout-exchange>

<!-- ~~~~~通配符：*匹配一个单词，#匹配多个单词
~~~~~ -->
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_topic_queue_star" name="spring_topic_queue_star"
auto-declare="true"/>
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_topic_queue_well" name="spring_topic_queue_well"
auto-declare="true"/>
<!--定义广播交换机中的持久化队列，不存在则自动创建-->
<rabbit:queue id="spring_topic_queue_well2" name="spring_topic_queue_well2"
auto-declare="true"/>

<rabbit:topic-exchange id="spring_topic_exchange"
name="spring_topic_exchange" auto-declare="true">
    <rabbit:bindings>
        <rabbit:binding pattern="ydt.*" queue="spring_topic_queue_star"/>
        <rabbit:binding pattern="ydt.#" queue="spring_topic_queue_well"/>
        <rabbit:binding pattern="ydt.#" queue="spring_topic_queue_well2"/>
    </rabbit:bindings>
</rabbit:topic-exchange>

<!--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"/>

```



```
</beans>
```

5.1.3. 发送消息

创建测试文件 ProducerTest.java`

```
package com.ydt.rabbitmq;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:spring-rabbitmq.xml")
public class ProducerTest {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    /**
     * 只发队列消息
     * 默认交换机类型为 direct
     * 交换机的名称为空，路由键为队列的名称
     */
    @Test
    public void queueTest(){
        //路由键与队列同名
        rabbitTemplate.convertAndSend("spring_queue", "只发队列spring_queue的消息。");
    }

    /**
     * 发送广播
     * 交换机类型为 fanout
     * 绑定到该交换机的所有队列都能够收到消息
     */
    @Test
    public void fanoutTest(){
        /**
         * 参数1: 交换机名称
         * 参数2: 路由键名（广播设置为空）
         * 参数3: 发送的消息内容
         */
        rabbitTemplate.convertAndSend("spring_fanout_exchange", "", "发送到spring_fanout_exchange交换机的广播消息");
    }

    /**
     * 通配符
     * 交换机类型为 topic
     * 匹配路由键的通配符，*表示一个单词，#表示多个单词
     * 绑定到该交换机的匹配队列能够收到对应消息
     */
}
```

```

    */
    @Test
    public void topicTest(){
        /**
         * 参数1: 交换机名称
         * 参数2: 路由键名
         * 参数3: 发送的消息内容
         */
        rabbitTemplate.convertAndSend("spring_topic_exchange", "ydt.bj", "发送到spring_topic_exchange交换机ydt.bj的消息");
        rabbitTemplate.convertAndSend("spring_topic_exchange", "ydt.bj.1", "发送到spring_topic_exchange交换机ydt.bj.1的消息");
        rabbitTemplate.convertAndSend("spring_topic_exchange", "ydt.bj.2", "发送到spring_topic_exchange交换机ydt.bj.2的消息");
        rabbitTemplate.convertAndSend("spring_topic_exchange", "ydt.cn", "发送到spring_topic_exchange交换机ydt.cn的消息");
    }
}

```

5.2. 搭建消费者工程

5.2.2. 添加依赖

修改pom.xml文件内容为如下:

```

<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.7.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.amqp</groupId>
        <artifactId>spring-rabbit</artifactId>
        <version>2.2.6.RELEASE</version>
    </dependency>
</dependencies>

```

5.2.3. 配置整合

1. 创建rabbitmq.properties`连接参数等配置文件;

```

rabbitmq.host=192.168.223.128
rabbitmq.port=5672
rabbitmq.username=ydt
rabbitmq.password=ydt
rabbitmq.virtual-host=ydt

```

2. 创建 spring-rabbitmq.xml` 整合配置文件;

```
<?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: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:properties/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}"/>

    <bean id="springQueueListener"
class="com.ydt.rabbitmq.listener.SpringQueueListener"/>
    <bean id="fanoutListener1"
class="com.ydt.rabbitmq.listener.FanoutListener1"/>
    <bean id="fanoutListener2"
class="com.ydt.rabbitmq.listener.FanoutListener2"/>
    <bean id="topicListenerStar"
class="com.ydt.rabbitmq.listener.TopicListenerStar"/>
    <bean id="topicListenerWell"
class="com.ydt.rabbitmq.listener.TopicListenerWell"/>
    <bean id="topicListenerWell2"
class="com.ydt.rabbitmq.listener.TopicListenerWell2"/>

    <rabbit:listener-container connection-factory="connectionFactory" auto-
declare="true">
        <rabbit:listener ref="springQueueListener" queue-names="spring_queue"/>
        <rabbit:listener ref="fanoutListener1" queue-
names="spring_fanout_queue_1"/>
        <rabbit:listener ref="fanoutListener2" queue-
names="spring_fanout_queue_2"/>
        <rabbit:listener ref="topicListenerStar" queue-
names="spring_topic_queue_star"/>
        <rabbit:listener ref="topicListenerWell" queue-
names="spring_topic_queue_well"/>
        <rabbit:listener ref="topicListenerWell2" queue-
names="spring_topic_queue_well2"/>
    </rabbit:listener-container>
</beans>
```

5.2.4. 消息监听器

1) 队列监听器

创建 SpringQueueListener.java`

```
public class SpringQueueListener implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("接收路由名称为: %s, 路由键为: %s, 队列名为: %s的消息: %s\n",
                               message.getMessageProperties().getReceivedExchange(),
                               message.getMessageProperties().getReceivedRoutingKey(),
                               message.getMessageProperties().getConsumerQueue(),
                               msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2) 广播监听器1

创建 FanoutListener1.java`

```
public class FanoutListener1 implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("广播监听器1: 接收路由名称为: %s, 路由键为: %s, 队列名为: %s的消息: %s\n",
                               message.getMessageProperties().getReceivedExchange(),
                               message.getMessageProperties().getReceivedRoutingKey(),
                               message.getMessageProperties().getConsumerQueue(),
                               msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3) 广播监听器2

创建 FanoutListener2.java`

```
public class FanoutListener2 implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("广播监听器2: 接收路由名称为: %s, 路由键为: %s, 队列名为: %s的消息: %s\n",
                               message.getMessageProperties().getReceivedExchange(),
                               message.getMessageProperties().getReceivedRoutingKey(),
```

```

        message.getMessageProperties().getConsumerQueue(),
        msg);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

4) 星号通配符监听器

创建 TopicListenerStar.java

```

public class TopicListenerStar implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("通配符*监听器: 接收路由名称为: %s, 路由键为: %s, 队列名  
为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

5) 井号通配符监听器

创建 TopicListenerWell.java`

```

public class TopicListenerWell implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("通配符#监听器: 接收路由名称为: %s, 路由键为: %s, 队列名  
为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

6) 井号通配符监听器2

创建 TopicListenerWell2.java`

```

public class TopicListenerWell2 implements MessageListener {
    public void onMessage(Message message) {
        try {
            String msg = new String(message.getBody(), "utf-8");

            System.out.printf("通配符#监听器2: 接收路由名称为: %s, 路由键为: %s, 队列名  
为: %s的消息: %s \n",
                message.getMessageProperties().getReceivedExchange(),
                message.getMessageProperties().getReceivedRoutingKey(),
                message.getMessageProperties().getConsumerQueue(),
                msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

7) 消费者测试类

```

package com.ydt.rabbitmq;

import org.junit.Test;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ConsumerTest {

    @Test
    public void test(){
        new ClassPathXmlApplicationContext("spring-rabbitmq.xml");
        while (true){

        }
    }
}

```

5.2.5. 开启测试

先运行生产者，生成交换机，队列，并且进行相应的routingkey路由绑定，最后发送消息

再运行消费者，对生产者发送的消息进行消费

注意：因为消费者只是配置了监听类，所以需要先运行生产者，否则会找不到对应的交换机队列等

6. Spring Boot整合RabbitMQ

6.1. 简介

在spring boot项目中只需要引入对应的amqp启动器依赖即可，方便的使用RabbitTemplate发送消息，使用注解接收消息。

一般在开发过程中:

生产者工程：

1. application.yml文件配置RabbitMQ相关信息；
2. 在生产者工程中编写配置类，用于创建交换机和队列，并进行绑定
3. 注入RabbitTemplate对象，通过RabbitTemplate对象发送消息到交换机

消费者工程：

1. application.yml文件配置RabbitMQ相关信息
2. 创建消息处理类，用于接收队列中的消息并进行处理

5.2. 搭建生产者工程

5.2.1. 添加依赖

修改pom.xml文件内容为如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>

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

5.2.3. 启动类

```

package com.ydt.rabbitmq;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ProducerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProducerApplication.class);
    }
}

```

5.2.4. 配置RabbitMQ

1) 配置文件

创建application.yml，内容如下：

```

spring:
  rabbitmq:
    host: 192.168.223.128
    port: 5672
    virtual-host: ydt
    username: ydt
    password: ydt

```

2) 绑定交换机和队列

创建RabbitMQ队列与交换机绑定的配置类RabbitMQConfig

```

package com.ydt.rabbitmq.config;

import org.springframework.amqp.core.*;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {
    //交换机名称
    public static final String ITEM_TOPIC_EXCHANGE = "item_topic_exchange";
    //队列名称
    public static final String ITEM_QUEUE = "item_queue";

    //声明交换机
    @Bean("itemTopicExchange")
    public Exchange topicExchange(){
        return
        ExchangeBuilder.topicExchange(ITEM_TOPIC_EXCHANGE).durable(true).build();
    }
}

```



```

//声明队列
@Bean("itemQueue")
public Queue itemQueue(){
    return QueueBuilder.durable(ITEM_QUEUE).build();
}

//绑定队列和交换机
@Bean
public Binding itemQueueExchange(@Qualifier("itemQueue") Queue queue,
    @Qualifier("itemTopicExchange") Exchange
exchange){
    return BindingBuilder.bind(queue).to(exchange).with("item.#").noargs();
}
}

```

5.3. 搭建消费者工程

5.3.1. 添加依赖

修改pom.xml文件内容为如下：

```

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

```

5.3.3. 启动类

```

package com.ydt.rabbitmq;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class);
    }
}

```

5.3.4. 配置RabbitMQ

创建application.yml，内容如下：

```
spring:
  rabbitmq:
    host: 192.168.223.128
    port: 5672
    virtual-host: ydt
    username: ydt
    password: ydt
```

5.3.5. 消息监听处理类

编写消息监听器MyListener

```
package com.ydt.rabbitmq.listener;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class MyListener {

    /**
     * 监听某个队列的消息
     * @param message 接收到的消息
     */
    @RabbitListener(queues = "item_queue")
    public void myListener1(String message){
        System.out.println("消费者接收到的消息为: " + message);
    }
}
```

5.4. 测试

在**生产者工程**中创建测试类，发送消息：

```
package com.ydt.rabbitmq;

import com.ydt.rabbitmq.config.RabbitMQConfig;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitMQTest {

    @Autowired
    private RabbitTemplate rabbitTemplate;
```

```

@Test
public void test(){
    rabbitTemplate.convertAndSend(RabbitMQConfig.ITEM_TOPIC_EXCHANGE,
        "item.insert", "商品新增, routing key 为item.insert");
    rabbitTemplate.convertAndSend(RabbitMQConfig.ITEM_TOPIC_EXCHANGE,
        "item.update", "商品修改, routing key 为item.update");
    rabbitTemplate.convertAndSend(RabbitMQConfig.ITEM_TOPIC_EXCHANGE,
        "item.delete", "商品删除, routing key 为item.delete");
    }
}

```

先运行上述测试程序（交换机和队列才能先被声明和绑定）

然后启动消费者，在消费者工程中控制台查看是否接收到对应消息。

另外；也可以在RabbitMQ的管理控制台中查看到交换机与队列的绑定：

The screenshot shows the RabbitMQ Management UI with the 'Exchanges' tab selected. The 'Details' section displays the following information:

- Type: topic
- Features: durable: true
- Policy: (empty)

The 'Bindings' section shows a binding from 'This exchange' to 'item_queue' with the routing key 'item.#'. The 'To' column contains 'item_queue', the 'Routing key' column contains 'item.#', and the 'Arguments' column is empty. An 'Unbind' button is visible next to the binding.

To	Routing key	Arguments	
item_queue	item.#		Unbind