

RabbitMQ入门进阶学习

学习目标

- 消息队列介绍
- 安装RabbitMQ

RabbitMQ的使用操作

- ==编写RabbitMQ的入门程序[简单模式]==

消息模式6种

- ==RabbitMQ的5种模式讲解==

应用场景
实现方式

- ==SpringBoot整合RabbitMQ==

1. 消息队列概述

1.1. 消息队列MQ

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。

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高级消息队列协议，是一个进程间传递异步消息的网络协议，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

JMS

- ①订阅模式
- ②点对点消息模式

1.3. 消息队列产品

市场上常见的消息队列有如下：

目前市面上成熟主流的MQ有Kafka、RocketMQ、RabbitMQ，我们这里对每款MQ做一个简单介绍。

Kafka

Apache下的一个子项目，使用scala实现的一个高性能分布式Publish/Subscribe消息队列系统。

- 1.快速持久化：通过磁盘顺序读写与零拷贝机制，可以在O(1)的系统开销下进行消息持久化；
- 2.高吞吐：在一台普通的服务器上既可以达到10w/s的吞吐速率；
- 3.高堆积：支持topic下消费者较长时间离线，消息堆积量大；
- 4.完全的分布式系统：Broker、Producer、Consumer都原生自动支持分布式，依赖zookeeper自动实现复杂均衡；
- 5.支持Hadoop数据并行加载：对于像Hadoop的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。

RocketMQ

RocketMQ的前身是Metaq，当Metaq3.0发布时，产品名称改为RocketMQ。RocketMQ是一款分布式、队列模型的消息中间件，具有以下特点：

- 1.能够保证严格的消息顺序
- 2.提供丰富的消息拉取模式
- 3.高效的订阅者水平扩展能力
- 4.实时的消息订阅机制
- 5.支持事务消息
- 6.亿级消息堆积能力

RabbitMQ

使用Erlang编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP，SMTP，STOMP，也正是如此，使它变的非常重量级，更适合于企业级的开发。同时实现了Broker架构，核心思想是生产者不会将消息直接发送给队列，消息在发送给客户端时先在中心队列排队。对路由(Routing)，负载均衡(Load balance)、数据持久化都有很好的支持。多用于进行企业级的ESB整合。

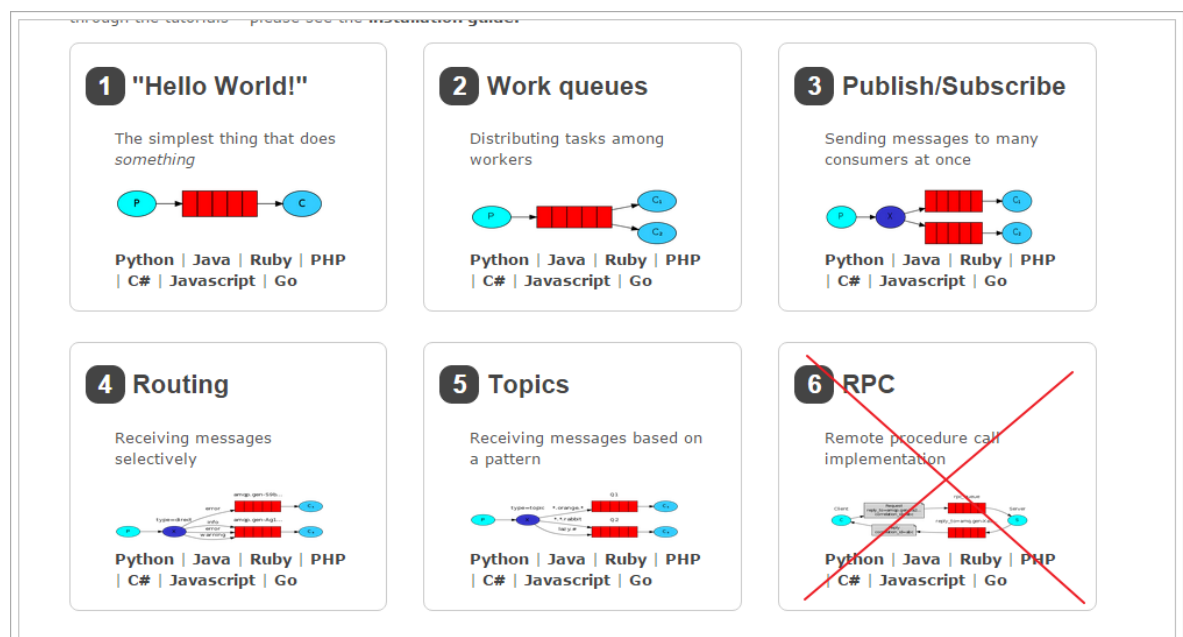
1.4. RabbitMQ

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；不作介绍）；

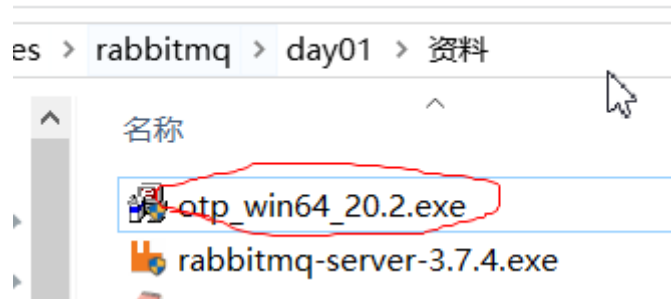
官网对应模式介绍：<https://www.rabbitmq.com/getstarted.html>



2. 安装及配置RabbitMQ

2.1. 安装说明

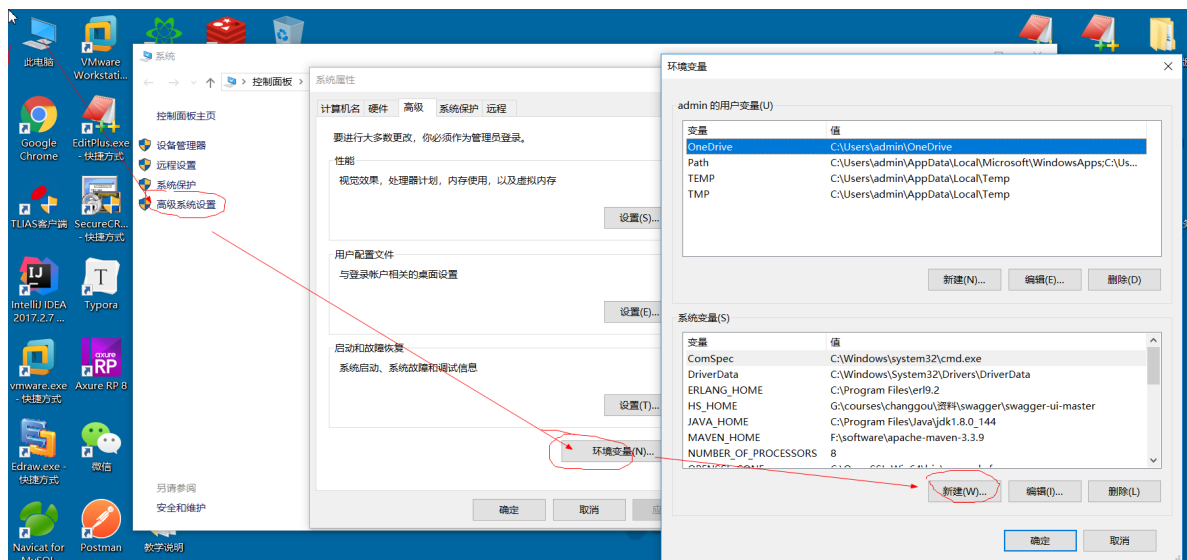
1.安装erlang，如下图：注意使用**管理员身份**打开即可



2.安装rabbitmq-server:

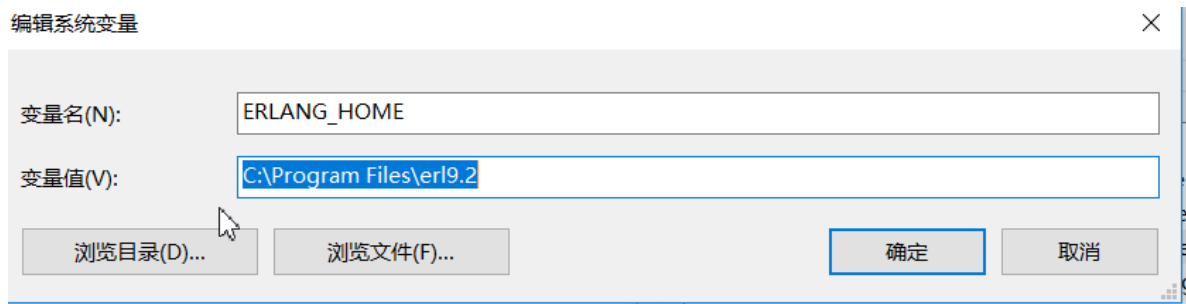
如上图使用**管理员身份**打开那个rabbitmq-server-3.7.4.exe文件。

3.配置环境变量

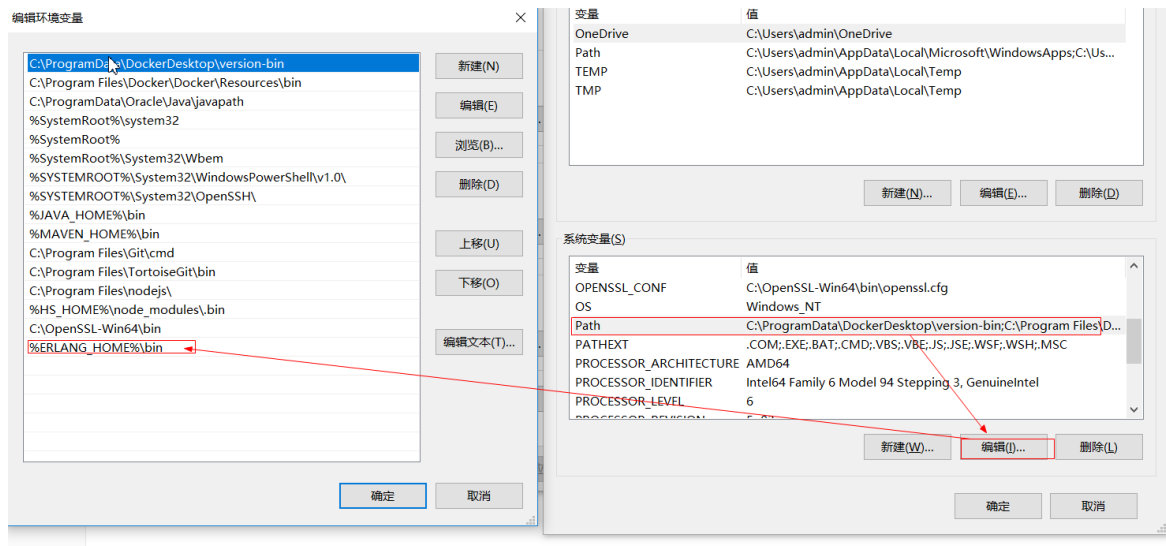


配置：erlang-home;指定你的erlang的安装目录

编辑系统变量



添加到path中:



(3) cmd到rabbitmq-server的 sbin目录下 打开cmd命令行工具

```
cd C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.4\sbin
```

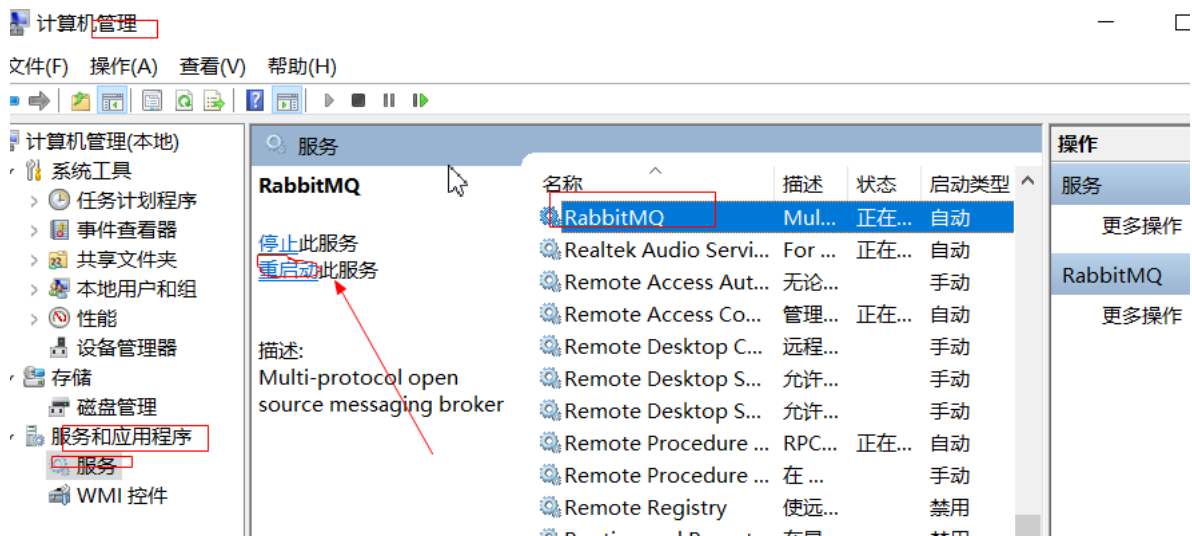
执行命令:

```
rabbitmq-plugins.bat enable rabbitmq_management
```

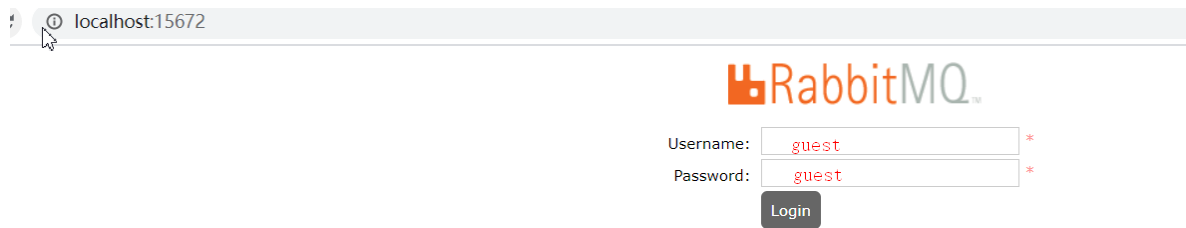
```
C:\Users\admin>cd C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.4\sbin
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.4\sbin>rabbitmq-plugins.bat enable rabbitmq_management
Enabling plugins on node rabbit@DESKTOP-3PTPOUD:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@DESKTOP-3PTPOUD...
Plugin configuration unchanged.
C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.4\sbin>`Z
```

(4)重启rabbitmq服务器:

此电脑--》右击 点击管理界面--》双击服务和应用程序--》双击 服务--》点击重启按钮即可



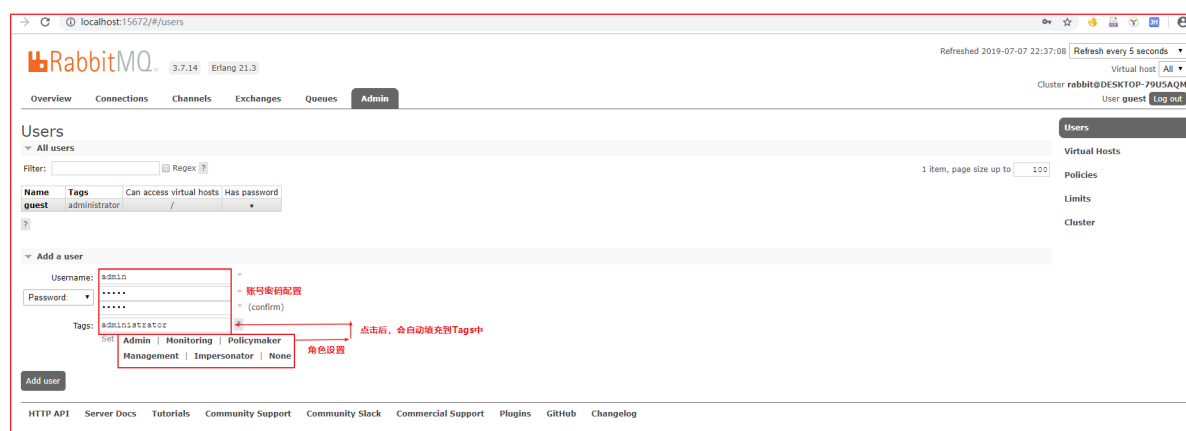
(5) 浏览器中输入localhost:15672 输 guest/guest



2.2. 用户以及Virtual Hosts配置

2.2.1. 用户角色

RabbitMQ在安装好后，可以访问<http://localhost: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。

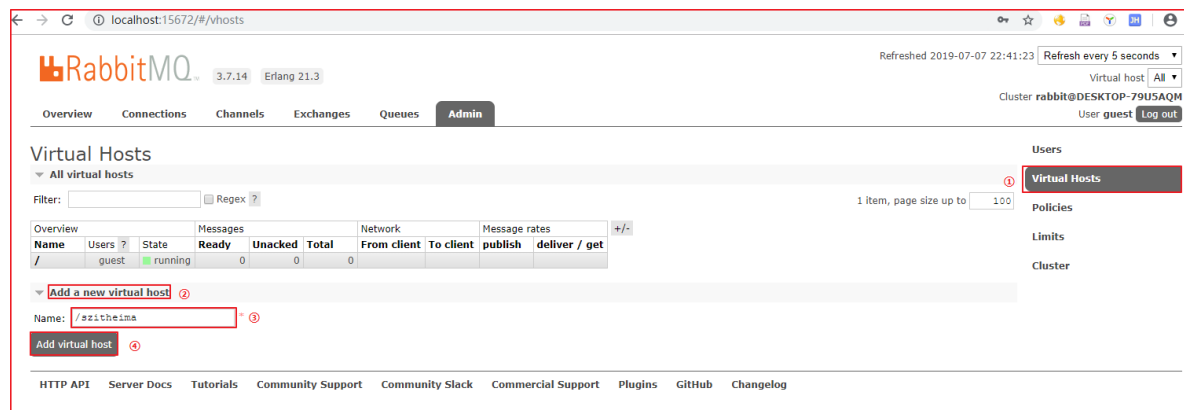
5、其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

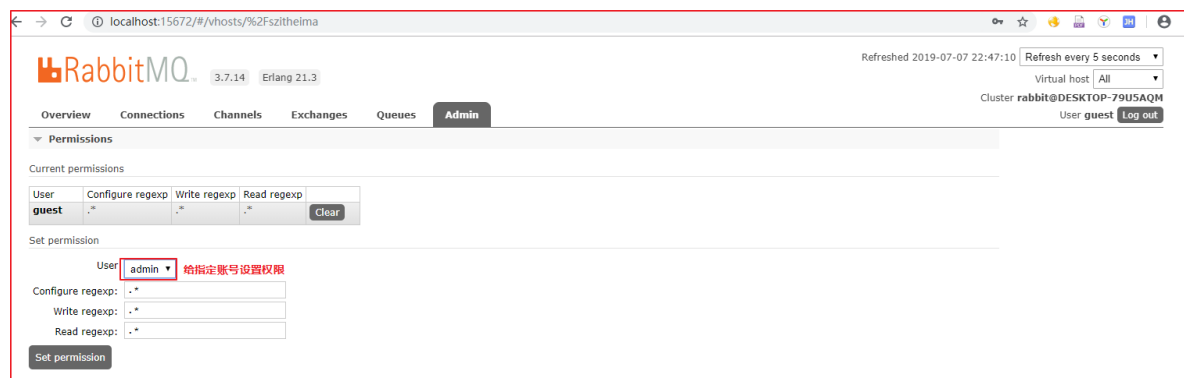
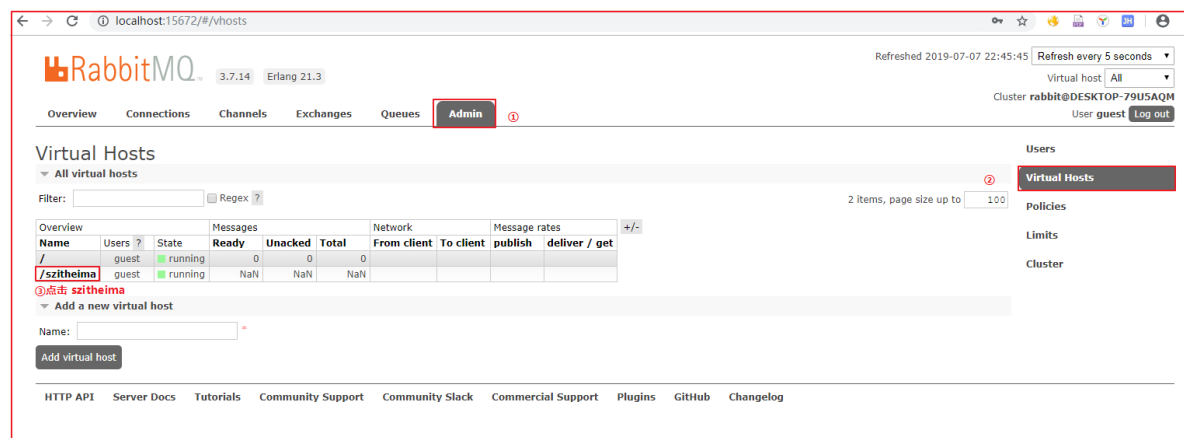
2.2.2. Virtual Hosts配置

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。

(1)创建Virtual Hosts



(2)设置Virtual Hosts权限



参数说明：

user: 用户名

configure : 一个正则表达式，用户对符合该正则表达式的所有资源拥有 **configure** 操作的权限

write: 一个正则表达式，用户对符合该正则表达式的所有资源拥有 **write** 操作的权限

read: 一个正则表达式，用户对符合该正则表达式的所有资源拥有 **read** 操作的权限

3. RabbitMQ入门

入门案例将使用RabbitMQ的简单模式实现。

3.1. 搭建示例工程

3.1.1. 创建工程

工程坐标如下：

```
<groupId>com.itheima</groupId>
<artifactId>rabbitmq-demo</artifactId>
<version>1.0-SNAPSHOT</version>
```

3.1.2. 添加依赖

往heima-rabbitmq的pom.xml文件中添加如下依赖：

```
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.6.0</version>
</dependency>
```

3.2. 生产者

生产者的创建分为如下几个步骤：

```
//创建链接工厂对象
//设置RabbitMQ服务主机地址,默认localhost
//设置RabbitMQ服务端口,默认5672
//设置虚拟主机名字,默认/
//设置用户连接名,默认guest
//设置链接密码,默认guest
//创建链接
//创建频道
//声明队列
//创建消息
//消息发送
//关闭资源
```

按照上面的步骤，我们创建一个消息生产者，创建com.itheima.rabbitmq.simple.Producer类，代码如下：

```
public class Producer {

    /**
     * 消息生产者
     * @param args
     * @throws IOException
     * @throws TimeoutException
     */
}
```



```

public static void main(String[] args) throws IOException, TimeoutException
{
    //创建链接工厂对象
    ConnectionFactory connectionFactory = new ConnectionFactory();

    //设置RabbitMQ服务主机地址,默认localhost
    connectionFactory.setHost("localhost");

    //设置RabbitMQ服务端口,默认5672
    connectionFactory.setPort(5672);

    //设置虚拟主机名字,默认/
    connectionFactory.setVirtualHost("/szitheima");

    //设置用户连接名,默认guest
    connectionFactory.setUsername("admin");

    //设置链接密码,默认guest
    connectionFactory.setPassword("admin");

    //创建链接
    Connection connection = connectionFactory.newConnection();

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

    /**
     * 声明队列
     * 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("simple_queue",true,false,false,null);

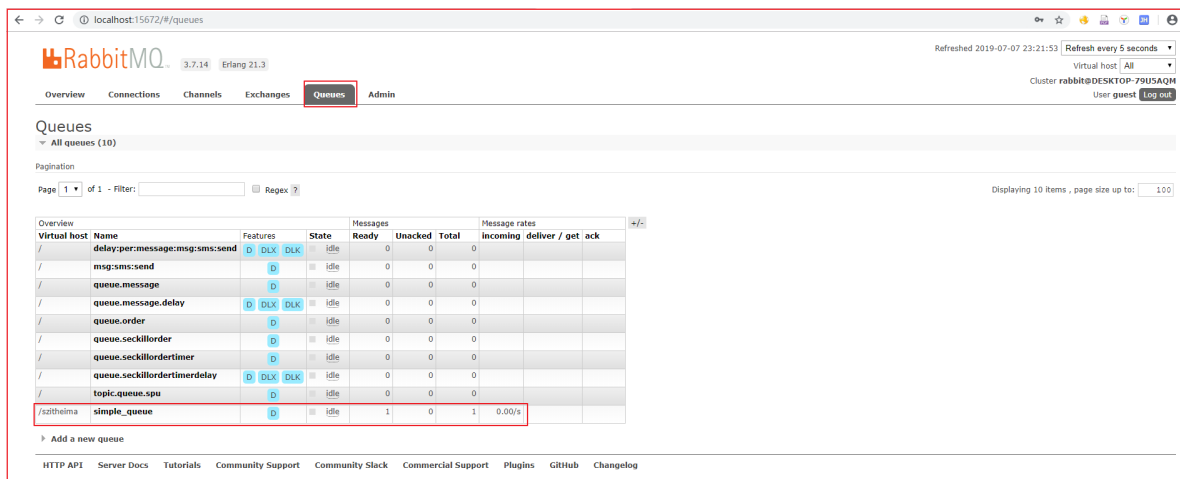
    //创建消息
    String message = "hello!welcome to itheima!";

    /**
     * 消息发送
     * 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("", "simple_queue", null, message.getBytes());

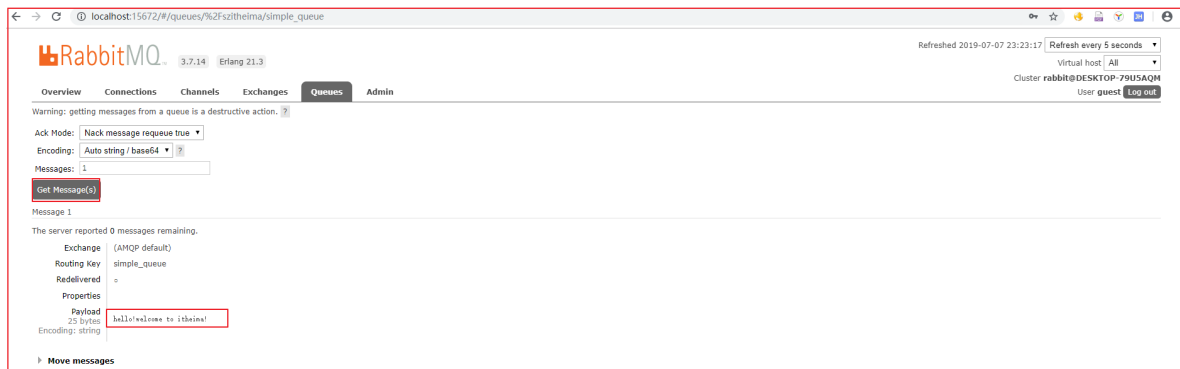
    //关闭资源
    channel.close();
    connection.close();
}
}

```

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:



如果想查看消息，可以点击 队列名称->Get Messages,如下图：



3.3. 消费者

消费者创建可以按照如下步骤实现：

```
//创建链接工厂对象
//设置RabbitMQ服务主机地址,默认localhost
//设置RabbitMQ服务端口,默认5672
//设置虚拟主机名字,默认/
//设置用户连接名,默认guest
//设置链接密码,默认guest
//创建链接
//创建频道
//创建队列
//创建消费者,并设置消息处理
//消息监听
//关闭资源(不建议关闭,建议一直监听消息)
```

按照上面的步骤创建消息消费者com.itheima.rabbitmq.simple.Consumer代码如下：

```
public class Consumer {

    /**
     * 消息消费者
     * @param args
     * @throws IOException
     * @throws TimeoutException
     */
}
```

```

public static void main(String[] args) throws IOException, TimeoutException
{
    //创建链接工厂对象
    ConnectionFactory connectionFactory = new ConnectionFactory();

    //设置RabbitMQ服务主机地址,默认localhost
    connectionFactory.setHost("localhost");

    //设置RabbitMQ服务端口,默认5672
    connectionFactory.setPort(5672);

    //设置虚拟主机名字,默认/
    connectionFactory.setVirtualHost("/szitheima");

    //设置用户连接名,默认guest
    connectionFactory.setUsername("admin");

    //设置链接密码,默认guest
    connectionFactory.setPassword("admin");

    //创建链接
    Connection connection = connectionFactory.newConnection();

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

    //创建队列
    channel.queueDeclare("simple_queue",true,false,false,null);

    //创建消费者,并设置消息处理
    DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
        /**
         * @param consumerTag    消息者标签,在channel.basicConsume时候可以指定
         * @param 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)
         * @param properties        属性信息
         * @param body              消息
         * @throws IOException
         */
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope,
            AMQP.BasicProperties properties, byte[] body) throws IOException {
            //路由的key
            String routingKey = envelope.getRoutingKey();
            //获取交换机信息
            String exchange = envelope.getExchange();
            //获取消息ID
            long deliveryTag = envelope.getDeliveryTag();
            //获取消息信息
            String message = new String(body,"UTF-8");

            System.out.println("routingKey:"+routingKey+",exchange:"+exchange+",deliveryTag:"+deliveryTag+",message:"+message);
        }
    };

    /**
     * 消息监听

```

```

    * 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("simple_queue", true, defaultConsumer);

    //关闭资源(不建议关闭, 建议一直监听消息)
    //channel.close();
    //connection.close();
}
}

```

执行后, 控制台输入如下:

```

Run: Producer Consumer
C:\Program Files\Java\jdk\bin\java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
routingKey:simple_queue,exchange:,deliveryTag:1,message:hello!welcome to itheima!

```

RabbitMQ控制台如下:

Overview				Messages			Message rates			+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/	delay:per:message:msg:sms:send	D DLX DLK	idle	0	0	0				
/	msg:sms:send	D	idle	0	0	0				
/	queue.message	D	idle	0	0	0				
/	queue.message.delay	D DLX DLK	idle	0	0	0				
/	queue.order	D	idle	0	0	0				
/	queue.seckillorder	D	idle	0	0	0				
/	queue.seckillordertimer	D	idle	0	0	0				
/	queue.seckillordertimerdelay	D DLX DLK	idle	0	0	0				
/	topic.queue.spu	D	idle	0	0	0				
/szitheima	simple_queue	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	

3.4 工具抽取

(1)工具类抽取

无论是消费者, 还是生产者, 我们发现前面的几个步骤几乎一模一样, 所以可以抽取一个工具类,将下面这段代码抽取出去。

```

//创建链接工厂对象
ConnectionFactory connectionFactory = new ConnectionFactory();

//设置RabbitMQ服务主机地址,默认localhost
connectionFactory.setHost("localhost");

//设置RabbitMQ服务端口,默认5672
connectionFactory.setPort(5672);

//设置虚拟主机名字,默认/
connectionFactory.setVirtualHost("/szitheima");

//设置用户连接名,默认guest
connectionFactory.setUsername("admin");

//设置链接密码,默认guest
connectionFactory.setPassword("admin");

//创建链接
Connection connection = connectionFactory.newConnection();

```

创建com.itheima.rabbitmq.util.ConnectionUtil工具类对象,用于创建Connection,代码如下:

```

public class ConnectionUtil {

    /**
     * 创建链接对象
     * @return
     * @throws IOException
     * @throws TimeoutException
     */
    public static Connection getConnection() throws IOException,
    TimeoutException {
        //创建链接工厂对象
        ConnectionFactory connectionFactory = new ConnectionFactory();

        //设置RabbitMQ服务主机地址,默认localhost
        connectionFactory.setHost("localhost");

        //设置RabbitMQ服务端口,默认5672
        connectionFactory.setPort(5672);

        //设置虚拟主机名字,默认/
        connectionFactory.setVirtualHost("/szitheima");

        //设置用户连接名,默认guest
        connectionFactory.setUsername("admin");

        //设置链接密码,默认guest
        connectionFactory.setPassword("admin");

        //创建链接
        Connection connection = connectionFactory.newConnection();
        return connection;
    }
}

```

(2)生产者优化

修改com.itheima.rabbitmq.simple.Producer，链接对象使用上面的ConnectionUtil工具类创建，代码如下：

```
//创建链接
Connection connection = ConnectionUtil.getConnection();
```

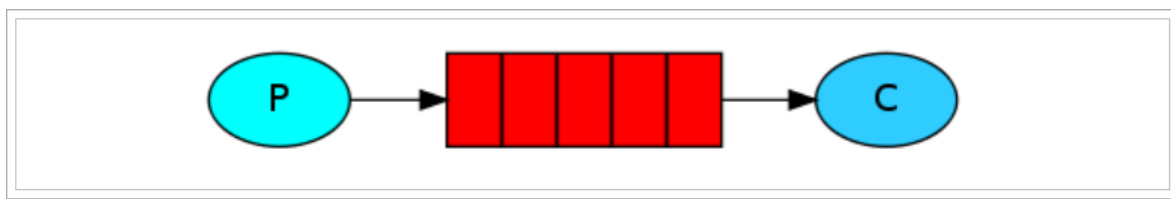
(3)消费者优化

修改com.itheima.rabbitmq.simple.Consumer，链接对象使用上面的ConnectionUtil工具类创建，代码如下：

```
//创建链接
Connection connection = ConnectionUtil.getConnection();
```

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。

在RabbitMQ中消息者是一定要到一个消息队列中去获取消息的

4. RabbitMQ工作模式

4.1. Work queues工作队列模式

4.1.1. 模式说明

2 **Work queues**

Distributing tasks among workers (the [competing consumers pattern](#))

```
graph LR; P((P)) --> Queue[ ]; Queue --> C1((C1)); Queue --> C2((C2))
```

- [Python](#)
- [Java](#)
- [Ruby](#)
- [PHP](#)
- [C#](#)
- [JavaScript](#)

- [Go](#)
- [Elixir](#)
- [Objective-C](#)
- [Swift](#)
- [Spring AMQP](#)

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

创建`com.itheima.rabbitmq.work.WorkProducer`消息生产者对象，代码如下：

```
public class WorkProducer {

    /**
     * 消息生产者
     * @param args
     * @throws IOException
     * @throws TimeoutException
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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("work_queue",true,false,false,null);

        for (int i = 0; i < 10; i++) {
            String message = "hello!welcome to itheima!" + i;
            channel.basicPublish("", "work_queue", null, message.getBytes());
        }

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

(2)消费者One

创建第1个Work消费者`com.itheima.rabbitmq.work.WorkConsumerOne`,代码如下：

```

public class WorkConsumerOne {

    /**
     * 消息消费者
     * @param args
     * @throws IOException
     * @throws TimeoutException
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {

        //创建链接
        Connection connection = ConnectionUtil.getConnection();

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

        //创建队列
        channel.queueDeclare("work_queue", true, false, false, null);

        //创建消费者，并设置消息处理
        DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
            /**
             * @param consumerTag 消息者标签，在channel.basicConsume时候可以指定
             * @param 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)
             * @param properties 属性信息
             * @param body 消息
             * @throws IOException
             */
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                //路由的key
                String routingKey = envelope.getRoutingKey();
                //获取交换机信息
                String exchange = envelope.getExchange();
                //获取消息ID
                long deliveryTag = envelope.getDeliveryTag();
                //获取消息信息
                String message = new String(body, "UTF-8");
                System.out.println("Work-
One:routingKey:"+routingKey+",exchange:"+exchange+",deliveryTag:"+deliveryTag+",
message:"+message);
            }
        };

        /**
         * 消息监听
         * 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("work_queue", true, defaultConsumer);

        //关闭资源(不建议关闭，建议一直监听消息)
        //channel.close();
        //connection.close();
    }
}

```



```
}  
}
```

(3)消费者Two

创建第2个Work消费者com.itheima.rabbitmq.work.WorkConsumerTwo, 代码如下:

```
public class WorkConsumerTwo {  
  
    /**  
     * 消息消费者  
     * @param args  
     * @throws IOException  
     * @throws TimeoutException  
     */  
    public static void main(String[] args) throws IOException, TimeoutException  
    {  
  
        //创建链接  
        Connection connection = ConnectionUtil.getConnection();  
  
        //创建频道  
        Channel channel = connection.createChannel();  
  
        //创建队列  
        channel.queueDeclare("work_queue", true, false, false, null);  
  
        //创建消费者, 并设置消息处理  
        DefaultConsumer defaultConsumer = new DefaultConsumer(channel){  
            /**  
             * @param consumerTag 消息者标签, 在channel.basicConsume时候可以指定  
             * @param 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)  
             * @param properties 属性信息  
             * @param body 消息  
             * @throws IOException  
             */  
            @Override  
            public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {  
                //路由的key  
                String routingKey = envelope.getRoutingKey();  
                //获取交换机信息  
                String exchange = envelope.getExchange();  
                //获取消息ID  
                long deliveryTag = envelope.getDeliveryTag();  
                //获取消息信息  
                String message = new String(body, "UTF-8");  
                System.out.println("Work-  
Two:routingKey:"+routingKey+",exchange:"+exchange+",deliveryTag:"+deliveryTag+",  
message:"+message);  
            }  
        };  
  
        /**
```

```

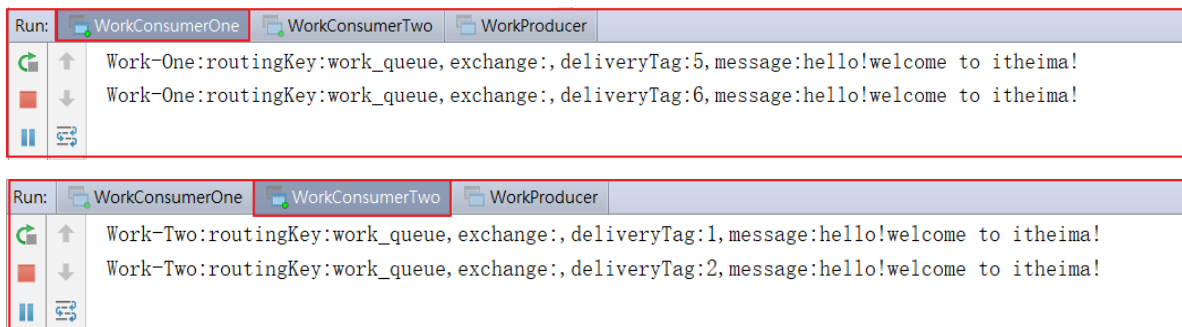
    * 消息监听
    * 参数1: 队列名称
    * 参数2: 是否自动确认, 设置为true为表示消息接收到自动向mq回复接收到了, mq接收到回复
    会删除消息, 设置为false则需要手动确认
    * 参数3: 消息接收到后回调
    */
    channel.basicConsume("work_queue", true, defaultConsumer);

    //关闭资源(不建议关闭, 建议一直监听消息)
    //channel.close();
    //connection.close();
}
}

```

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。

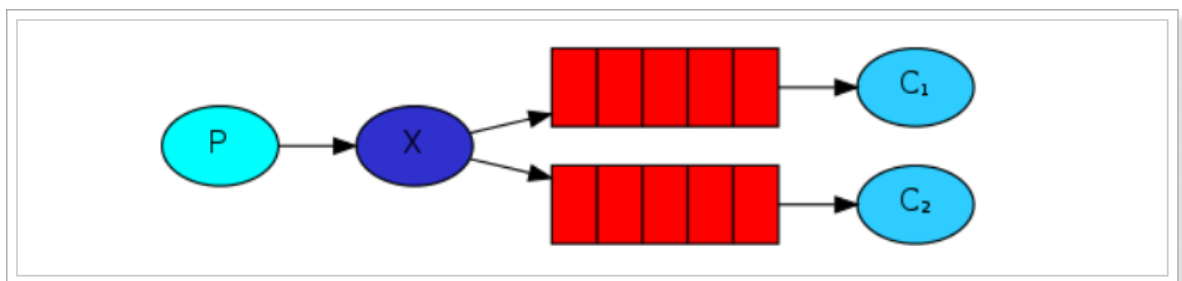


4.1.4. 小结

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

生产者需要注意如下3点：

1. 声明交换机

2. 声明队列

3. 队列需要绑定指定的交换机

创建com.itheima.rabbitmq.ps.PublishSubscribeProducer消息生产者，代码如下：

```
public class PublishSubscribeProducer {  
  
    /**
```

```

    * 订阅模式
    * @param args
    */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        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", "");

        //消息
        String message = "发布订阅模式:欢迎来到传深圳黑马训练营程序员中心! ";
        /**
         * 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("fanout_exchange", "", null, message.getBytes());

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

```

(2)消费者One

创建com.itheima.rabbitmq.ps.PublishSubscribeConsumerOne消息消费对象, 代码如下:

```

public class PublishSubscribeConsumerOne {

    /**

```

```

    * 订阅模式消息消费者
    * @param args
    */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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);

        //创建消费者; 并设置消息处理
        DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
            @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
                System.out.println("消费者One-接收到的消息为: " + new String(body,
                    "utf-8"));
            }
        };

        //消息监听
        channel.basicConsume("fanout_queue_1", true, defaultConsumer);

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

```

(3)消费者Two

创建com.itheima.rabbitmq.ps.PublishSubscribeConsumerTwo消息消费对象, 代码如下:

```

public class PublishSubscribeConsumerTwo {

    /**
     * 订阅模式消息消费者
     * @param args

```

```

    */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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_2", true, false, false, null);

        //创建消费者; 并设置消息处理
        DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
            @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
                System.out.println("消费者Two-接收到的消息为: " + new String(body,
                    "utf-8"));
            }
        };

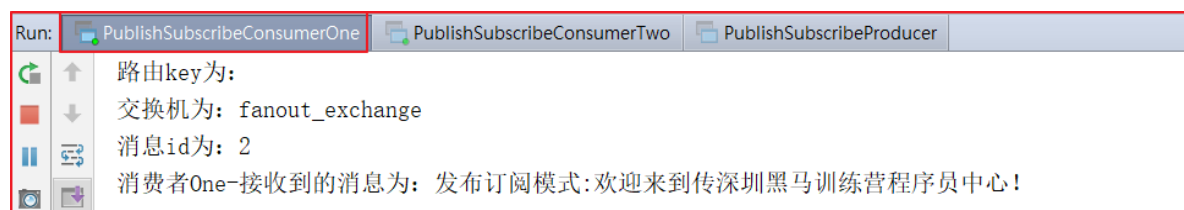
        //消息监听
        channel.basicConsume("fanout_queue_2", true, defaultConsumer);

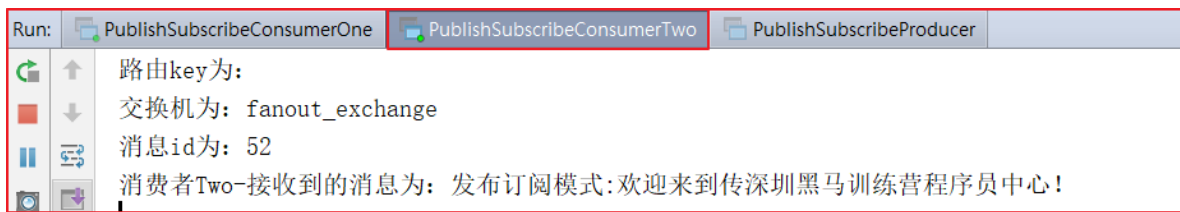
        //关闭资源
        //channel.close();
        //connection.close();
    }
}

```

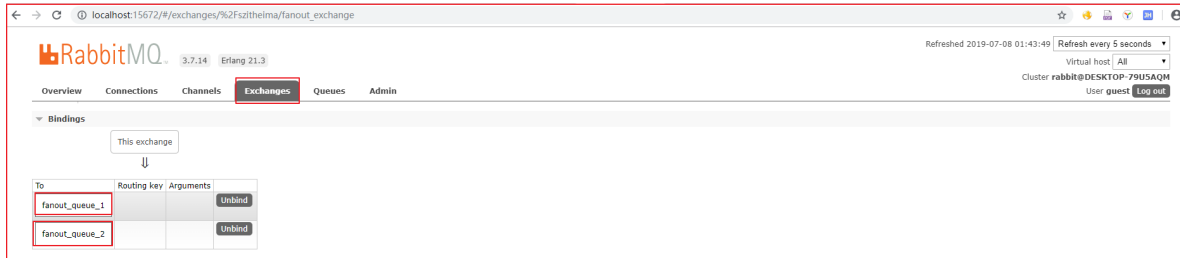
4.3.3. 测试

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：



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work队列模式的区别

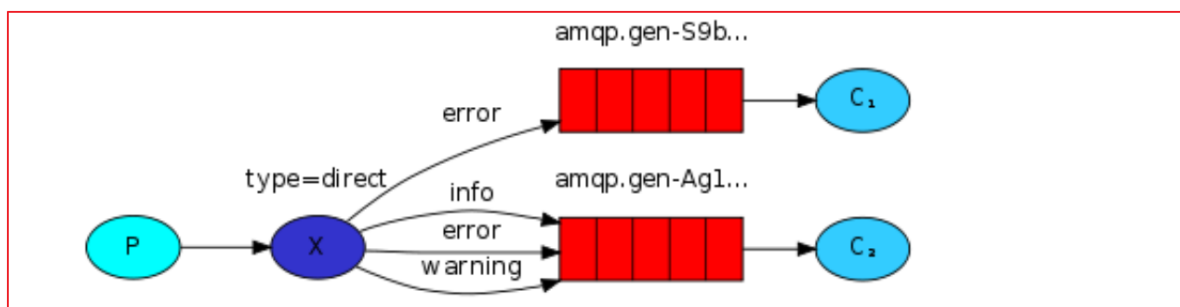
- 1、**work**队列模式不用定义交换机，而发布/订阅模式需要定义交换机。
- 2、发布/订阅模式的生产方是面向交换机发送消息，**work**队列模式的生产方是面向队列发送消息（底层使用默认交换机）。
- 3、发布/订阅模式需要设置队列和交换机的绑定，**work**队列模式不需要设置，实际上**work**队列模式会将队列绑定到默认的交换机。

4.4. Routing路由模式

4.4.1. 模式说明

路由模式特点：

1. 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）
2. 消息的发送方在 向 **Exchange**发送消息时，也必须指定消息的 **RoutingKey**。
3. **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

创建com.itheima.rabbitmq.toutekey.RouteKeyProducer消息生产者，代码如下：

```
public class RouteKeyProducer {

    /**
     * 订阅模式-RouteKey
     * @param args
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        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");
    }
}
```



```

        //消息-direct_queue_insert
        String message_insert = "发布订阅模式-RouteKey-Insert:欢迎来到传深圳黑马训练营
程序员中心! ";
        /**
         * 消息发送
         * 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("direct_exchange","insert",null,message_insert.getBytes())
        ;

        //消息-direct_queue_update
        String message_update = "发布订阅模式-RouteKey-Update:欢迎来到传深圳黑马训练营
程序员中心! ";

        channel.basicPublish("direct_exchange","update",null,message_update.getBytes())
        ;

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

```

(2)消费者RouteKey-Insert

创建 `direct_queue_insert` 队列的消费者`com.itheima.rabbitmq.toutekey.ConsumerInsert`, 代码如下:

```

public class ConsumerInsert {

    /**
     * 订阅模式消息消费者-RouteKey-insert
     * @param args
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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);

        //创建消费者; 并设置消息处理
    }
}

```

```

        DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
            @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
                System.out.println("消费者Insert-接收到的消息为: " + new
String(body, "utf-8"));
            }
        };

        //消息监听
        channel.basicConsume("direct_queue_insert",true,defaultConsumer);

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

```

(3)消费者-RouteKey-Update

创建 `direct_queue_update` 队列的消费者 `com.itheima.rabbitmq.toutekey.ConsumerUpdate`, 代码如下:

```

public class ConsumerUpdate {

    /**
     * 订阅模式消息消费者-RouteKey-update
     * @param args
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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_update",true,false,false,null);

        //创建消费者; 并设置消息处理
        DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
            @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
            System.out.println("消费者Two-接收到的消息为: " + new String(body,
            "utf-8"));
        }
    };

    //消息监听
    channel.basicConsume("direct_queue_update", true, defaultConsumer);

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

```

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。

The first screenshot shows the following output:

```

Run: ConsumerUpdate ConsumerInsert RouteKeyProducer
路由key为: insert
交换机为: direct_exchange
消息id为: 3
消费者Insert-接收到的消息为: 发布订阅模式-RouteKey-Insert:欢迎来到传深圳黑马训练营程序员中心!

```

The second screenshot shows the following output:

```

Run: ConsumerUpdate ConsumerInsert RouteKeyProducer
路由key为: update
交换机为: direct_exchange
消息id为: 3
消费者Two-接收到的消息为: 发布订阅模式-RouteKey-Update:欢迎来到传深圳黑马训练营程序员中心!

```

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：

The screenshot shows the RabbitMQ management interface with the 'Exchanges' tab selected. The 'direct_exchange' is highlighted. Below it, the bindings are listed:

To	Routing key	Arguments	Unbind
direct_queue_insert	insert		Unbind
direct_queue_update	update		Unbind

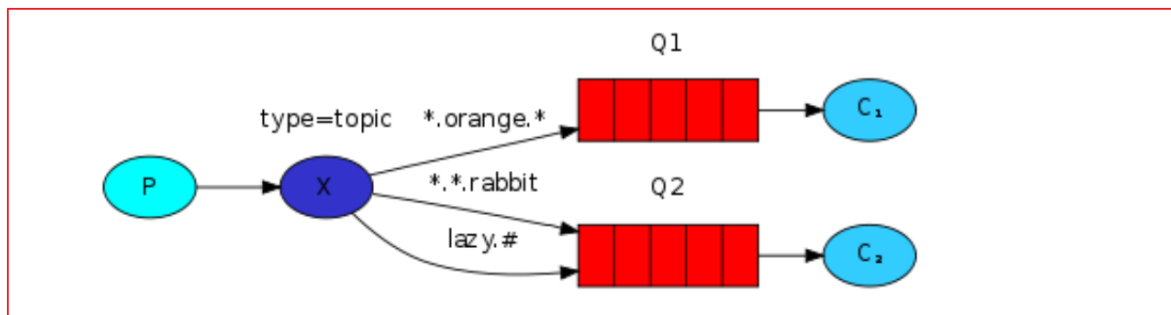
Below the table, there is a section 'Add binding from this exchange' with fields for 'To queue', 'Routing key', and 'Arguments'.

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：

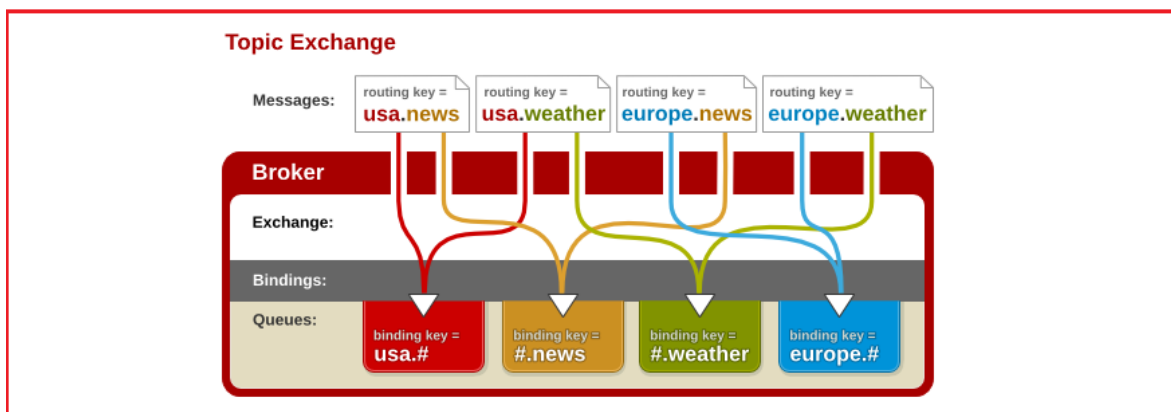
#：匹配一个或多个词

*：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`



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`：

创建com.itheima.rabbitmq.topic.TopicProducer实现消息生产，代码如下：

```
public class TopicProducer {

    /**
     * 订阅模式-Topic
     * @param args
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        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("topic_exchange", BuiltinExchangeType.TOPIC);

        /**
         * 声明队列
         * 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("topic_queue_1", true, false, false, null);
        channel.queueDeclare("topic_queue_2", true, false, false, null);

        //队列绑定交换机，同时添加routekey过滤
        channel.queueBind("topic_queue_1", "topic_exchange", "item.update");
        channel.queueBind("topic_queue_1", "topic_exchange", "item.delete");
        channel.queueBind("topic_queue_2", "topic_exchange", "item.*");

        //消息-item.insert
        String message_insert = "发布订阅模式-Topic-item.insert:欢迎来到传深圳黑马训练营程序员中心! ";

        /**
         * 消息发送
         * 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("topic_exchange", "item.insert", null, message_insert.getBytes());
    }
}
```

```

        //消息-item.update
        String message_update = "发布订阅模式-Topic-item.update:欢迎来到传深圳黑马训练营程序员中心! ";

        channel.basicPublish("topic_exchange", "item.update", null, message_update.getBytes());

        //消息-item.delete
        String message_delete = "发布订阅模式-Topic-item.delete:欢迎来到传深圳黑马训练营程序员中心! ";

        channel.basicPublish("topic_exchange", "item.delete", null, message_delete.getBytes());

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

```

(2)消费者one

上面配置了路由绑定过滤的规则，如下图：

```

//队列绑定交换机
channel.queueBind( queue: "topic_queue_1", exchange: "topic_exchange", routingKey: "item.update");
channel.queueBind( queue: "topic_queue_1", exchange: "topic_exchange", routingKey: "item.delete");
channel.queueBind( queue: "topic_queue_2", exchange: "topic_exchange", routingKey: "item.*");

```

创建com.itheima.rabbitmq.topic.ConsumerOne实现对topic_queue_1队列数据的消费，代码如下：

```

public class ConsumerOne {

    /**
     * 订阅模式消息消费者-Topic
     * @param args
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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("topic_queue_1", true, false, false, null);

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

    //消息监听
    channel.basicConsume("topic_queue_1",true,defaultConsumer);

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

```

(3)消费者Two

```

//队列绑定交换机
channel.queueBind( queue: "topic_queue_1", exchange: "topic_exchange", routingKey: "item.update");
channel.queueBind( queue: "topic_queue_1", exchange: "topic_exchange", routingKey: "item.delete");
channel.queueBind( queue: "topic_queue_2", exchange: "topic_exchange", routingKey: "item.*");

```

item.update, item.delete, item.insert

接收所有类型的消息：新增商品，更新商品和删除商品。

创建com.itheima.rabbitmq.topic.ConsumerTwo实现消息消费，代码如下：

```

public class ConsumerTwo {

    /**
     * 订阅模式消息消费者-Topic
     * @param args
     */
    public static void main(String[] args) throws IOException, TimeoutException
    {
        //创建链接对象
        Connection connection = ConnectionUtil.getConnection();

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

        /**
         * 声明队列
         * 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("topic_queue_2",true,false,false,null);

        //创建消费者；并设置消息处理
    }
}

```

```

DefaultConsumer defaultConsumer = new DefaultConsumer(channel){
    @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
        System.out.println("消费者Topic-Queue-2-接收到的消息为: " + new
String(body, "utf-8"));
    }
};

//消息监听
channel.basicConsume("topic_queue_2", true, defaultConsumer);

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

```

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。

Run: ConsumerTwo ConsumerOne RouteKeyProducer

```

C:\Program Files\Java\jdk\bin\java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
路由key为: item.update
交换机为: topic_exchange
消息id为: 1
消费者Topic-Queue-1-接收到的消息为: 发布订阅模式-Topic-item.update:欢迎来到传深圳黑马训练营程序员中心!
路由key为: item.update
交换机为: topic_exchange
消息id为: 2
消费者Topic-Queue-1-接收到的消息为: 发布订阅模式-Topic-item.delete:欢迎来到传深圳黑马训练营程序员中心!

```

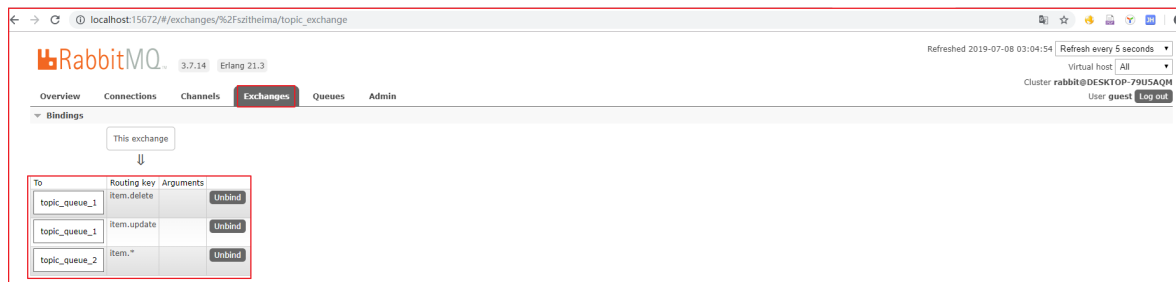
Run: ConsumerTwo ConsumerOne RouteKeyProducer

```

SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
路由key为: item.insert
交换机为: topic_exchange
消息id为: 1
消费者Topic-Queue-2-接收到的消息为: 发布订阅模式-Topic-item.insert:欢迎来到传深圳黑马训练营程序员中心!
路由key为: item.update
交换机为: topic_exchange
消息id为: 2
消费者Topic-Queue-2-接收到的消息为: 发布订阅模式-Topic-item.update:欢迎来到传深圳黑马训练营程序员中心!
路由key为: item.update
交换机为: topic_exchange
消息id为: 3
消费者Topic-Queue-2-接收到的消息为: 发布订阅模式-Topic-item.delete:欢迎来到传深圳黑马训练营程序员中心!

```


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 Boot整合RabbitMQ

5.1 简介

在Spring项目中，可以使用Spring-Rabbit去操作RabbitMQ

<https://github.com/spring-projects/spring-amqp>

尤其是在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 创建工程

创建生产者工程springboot-rabbitmq-producer, 工程坐标如下:

```
<groupId>com.itheima</groupId>
<artifactId>springboot-rabbitmq-producer</artifactId>
<version>1.0-SNAPSHOT</version>
```

5.2.2 添加依赖

修改pom.xml文件内容为如下:

```
<?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>

    <!--父工程-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

    <groupId>com.itheima</groupId>
    <artifactId>springboot-rabbitmq-producer</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!--依赖-->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-amqp</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
        </dependency>
    </dependencies>
```

```
</project>
```

5.2.3 启动类

创建启动类com.itheima.ProducerApplication，代码如下：

```
@SpringBootApplication
public class ProducerApplication {

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

5.2.4 配置RabbitMQ

(1)application.yml配置文件

创建application.yml，内容如下：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    virtual-host: /szitheima
    username: admin
    password: admin
```

(2)绑定交换机和队列

创建RabbitMQ队列与交换机绑定的配置类com.itheima.config.RabbitMQConfig,代码如下：

```
@Configuration
public class RabbitMQConfig {

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

    /**
     * 声明队列
     */
    @Bean(name = "itemQueue")
    public Queue itemQueue(){
        return QueueBuilder.durable("item_queue").build();
    }
}
```

```

    /**
     * 队列绑定到交换机上
     */
    @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. 创建工程

创建消费者工程springboot-rabbitmq-consumer,工程坐标如下:

```

<groupId>com.itheima</groupId>
<artifactId>springboot-rabbitmq-consumer</artifactId>
<version>1.0-SNAPSHOT</version>

```

5.3.2. 添加依赖

修改pom.xml文件内容为如下:

```

<?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>

    <!--父工程-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.4.RELEASE</version>
    </parent>

    <groupId>com.itheima</groupId>
    <artifactId>springboot-rabbitmq-consumer</artifactId>
    <version>1.0-SNAPSHOT</version>

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

```

5.3.3. 启动类

创建启动类com.itheima.ConsumerApplication，代码如下：

```
@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: localhost
    port: 5672
    virtual-host: /szitheima
    username: admin
    password: admin
```

5.3.5. 消息监听处理类

编写消息监听器com.itheima.listener.MessageListener，代码如下：

```
@Component
public class MessageListener {

    /**
     * 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springboot-rabbitmq-producer中创建测试类com.itheima.test.RabbitMQTest，发送消息：

```
@RunWith(SpringRunner.class)
@SpringBootTest
```

```

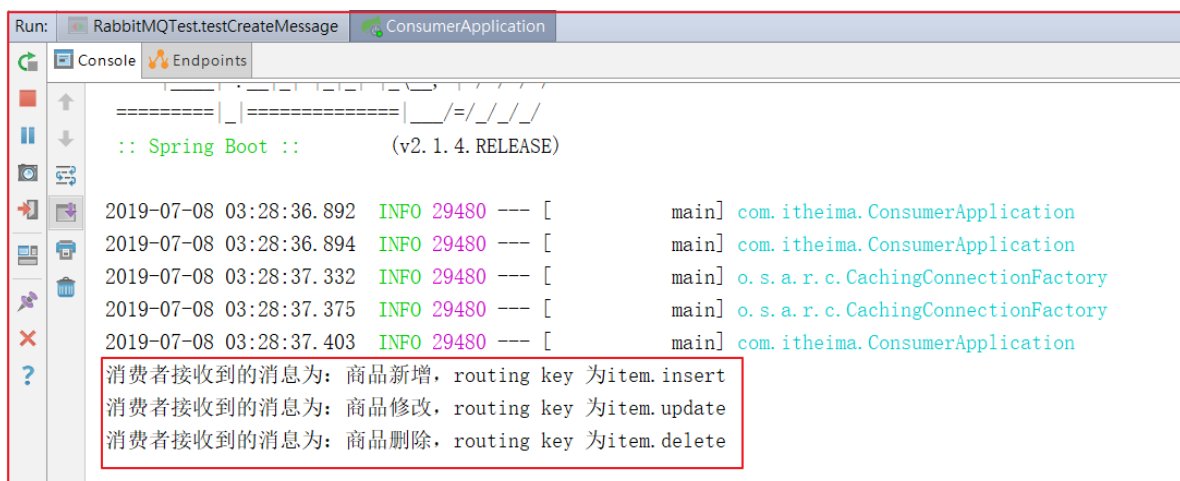
public class RabbitMQTest {

    //用于发送MQ消息
    @Autowired
    private RabbitTemplate rabbitTemplate;

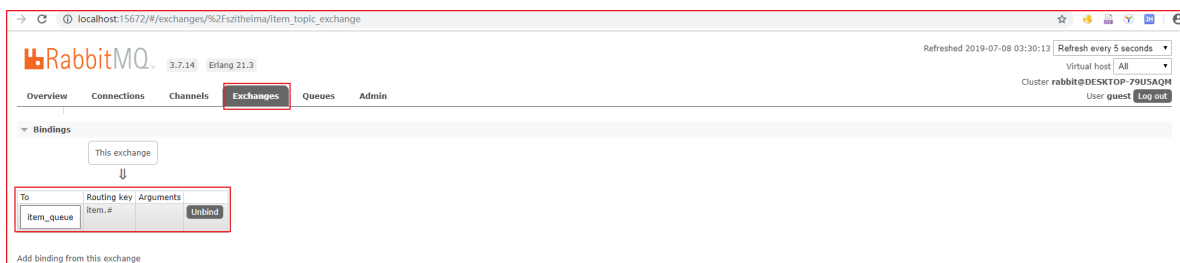
    /**
     * 消息生产测试
     */
    @Test
    public void testCreateMessage(){
        rabbitTemplate.convertAndSend("item_topic_exchange", "item.insert", "商品
        新增, routing key 为item.insert");
        rabbitTemplate.convertAndSend("item_topic_exchange", "item.update", "商品
        修改, routing key 为item.update");
        rabbitTemplate.convertAndSend("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springboot-rabbitmq-consumer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：



Rabbitmq高级特性

学习目标

- 掌握常见的高级特性
- 高级特性生产者可靠性消息投递
- 高级特性消费者ACK确认机制

- 理解相关应用性的解决方案
- 了解相关集群的搭建

1 RabbitMq高级特性

在消息的使用过程当中存在一些问题。比如发送消息我们如何确保消息的投递的可靠性呢？如何保证消费消息可靠性呢？如果不能保证在某些情况下可能会出现损失。比如当我们发送消息的时候和接收消息的时候能否根据消息的特性来实现某一些业务场景的模拟呢？订单30分钟过期等等，系统通信的确认等等。

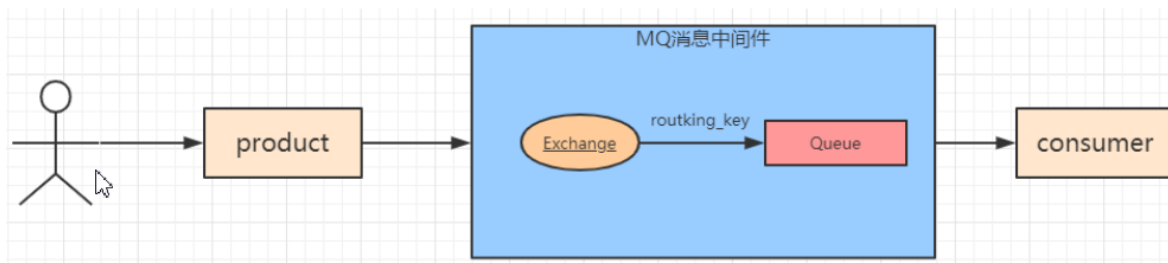
1.1 生产者可靠性消息投递

可靠性消息

在使用 RabbitMQ 的时候，作为消息发送方希望杜绝任何消息丢失或者投递失败场景。RabbitMQ 为我们提供了两种方式用来控制消息的投递可靠性模式，mq提供了如下两种模式：

- + **confirm**模式
生产者发送消息到交换机的时机
- + **return**模式
交换机转发消息给queue的时机

MQ投递消息的流程如下：



1. 生产者发送消息到交换机
2. 交换机根据**routingkey** 转发消息给队列
3. 消费者监控队列，获取队列中信息
4. 消费成功删除队列中的消息

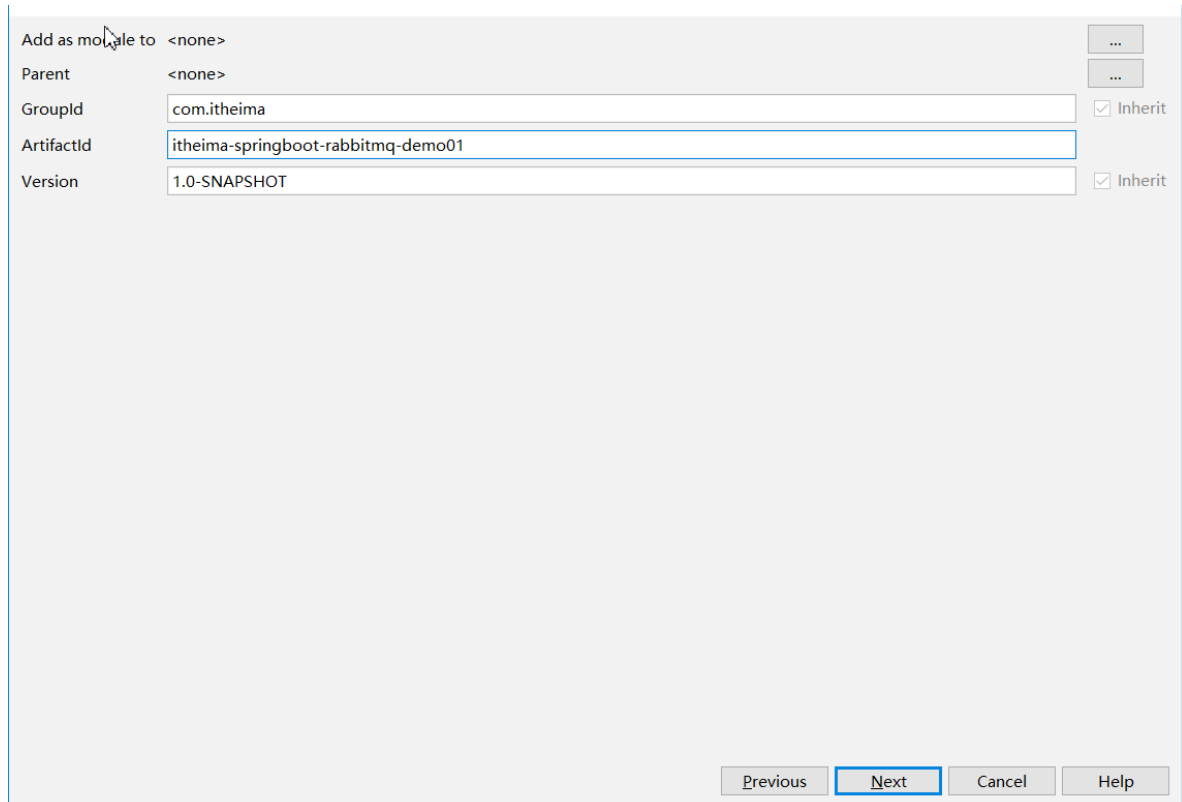
- 消息从 product 到 exchange 则会返回一个 `confirmCallback` 。
- 消息从 exchange 到 queue 投递失败则会返回一个 `returnCallback` 。

1.1.1 `confirmcallback`代码实现

执行的步骤：

1. 创建springboot工程
2. 添加起步依赖
3. 设置`configrm`回调函数
4. 发送消息

(1) 创建springboot工程



Add as module to <none>

Parent <none>

GroupId com.itheima

ArtifactId itheima-springboot-rabbitmq-demo01

Version 1.0-SNAPSHOT

☒ Inherit

☒ Inherit

Previous Next Cancel Help

(2) 添加依赖

```
<?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>com.itheima</groupId>
  <artifactId>itheima-springboot-rabbitmq-demo01</artifactId>
  <version>1.0-SNAPSHOT</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
  </parent>

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

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



```
</dependencies>
</project>
```

(3)在com.itheima下创建启动类,并创建配置交换机队列和绑定

```
package com.itheima;

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

@SpringBootApplication
public class RabbitmqDemo01Application {
    public static void main(String[] args) {
        SpringApplication.run(RabbitmqDemo01Application.class,args);
    }
    //创建队列
    @Bean
    public Queue createqueue(){
        return new Queue("queue_demo01");
    }

    //创建交换机
    @Bean
    public DirectExchange createExchange(){
        return new DirectExchange("exchange_direct_demo01");
    }

    //创建绑定
    @Bean
    public Binding createBinding(){
        return
        BindingBuilder.bind(createqueue()).to(createExchange()).with("item.insert");
    }
}
```

(4) 创建application.yml文件, 配置如下, 配置开启confirms模式, 默认为false

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
    publisher-confirms: true
  server:
    port: 8080
```

(5)创建controller 发送消息

```
@RestController
```

```

@RequestMapping("/test")
public class TestController {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private RabbitTemplate.ConfirmCallback myConfirmCallback;

    /**
     * 发送消息
     *
     * @return
     */
    @RequestMapping("/send1")
    public String send1() {
        //设置回调函数
        rabbitTemplate.setConfirmCallback(myConfirmCallback);
        //发送消息
        rabbitTemplate.convertAndSend("exchange_direct_demo01", "item.insert",
"hello insert");
        return "ok";
    }
}

```

(6)创建回调函数

```

package com.itheima.confirm;

import org.springframework.amqp.rabbit.connection.CorrelationData;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.lang.Nullable;
import org.springframework.stereotype.Component;

@Component
public class MyConfirmCallback implements RabbitTemplate.ConfirmCallback {

    /**
     *
     * @param correlationData 消息信息
     * @param ack 确认标识: true,MQ服务器exchange表示已经确认收到消息 false 表示没有收到消息
     * @param cause 如果没有收到消息,则指定为MQ服务器exchange消息没有收到的原因,如果已经收到则指定为null
     */
    @Override
    public void confirm(@Nullable CorrelationData correlationData, boolean ack,
@Nullable String cause) {
        if(ack){
            System.out.println("发送消息到交换机成功,"+cause);
        }else{
            System.out.println("发送消息到交换机失败,原因是: "+cause);
        }
    }
}

```

(7)测试发送消息:

启动应用，浏览器发送请求：<http://localhost:8080/test/send1>

打印如下：

```
abbiitmApplication
Console Endpoints
2020-03-01 10:03:27.970 INFO 16656 --- [main] com.itheima.RabbitmqDemo01Application : Starting RabbitmqDemo01Application on DESKTOP
2020-03-01 10:03:27.974 INFO 16656 --- [main] com.itheima.RabbitmqDemo01Application : No active profile set, falling back to default
2020-03-01 10:03:29.051 INFO 16656 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-03-01 10:03:29.075 INFO 16656 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-03-01 10:03:29.075 INFO 16656 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.1
2020-03-01 10:03:29.180 INFO 16656 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationCo
2020-03-01 10:03:29.180 INFO 16656 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization co
2020-03-01 10:03:29.442 INFO 16656 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTask
2020-03-01 10:03:29.682 INFO 16656 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with c
2020-03-01 10:03:29.685 INFO 16656 --- [main] com.itheima.RabbitmqDemo01Application : Started RabbitmqDemo01Application in 2.142 se
2020-03-01 10:04:13.213 INFO 16656 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispat
2020-03-01 10:04:13.213 INFO 16656 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-03-01 10:04:13.218 INFO 16656 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 5 ms
2020-03-01 10:04:13.239 INFO 16656 --- [nio-8080-exec-1] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2020-03-01 10:04:13.267 INFO 16656 --- [nio-8080-exec-1] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFacto
发送消息到交换机成功, null
```

稍微注意下：此时我们没有监听消息，那么只表示发送消息到交换机成功。

此时如果我们将交换机名称换掉，会出现失败的案例，如下：

```
@RequestMapping("/send1")
public String send1() {
    //设置回调函数
    rabbitTemplate.setConfirmCallback(myConfirmCallback);
    //发送消息
    rabbitTemplate.convertAndSend(exchange: "exchange_direct_demo02", routingKey: "item.insert", object: "hello insert");
    return "ok";
}
```

再次重启测试如下：

```
2020-03-01 10:06:43.597 INFO 16392 --- [main] com.itheima.RabbitmqDemo01Application : Starting RabbitmqDemo01Application on DESKTOP-3PTPOUD with PID 16392 (C:\User
2020-03-01 10:06:43.601 INFO 16392 --- [main] com.itheima.RabbitmqDemo01Application : No active profile set, falling back to default profiles: default
2020-03-01 10:06:44.779 INFO 16392 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-03-01 10:06:44.801 INFO 16392 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-03-01 10:06:44.801 INFO 16392 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.17]
2020-03-01 10:06:44.909 INFO 16392 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-03-01 10:06:44.909 INFO 16392 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1247 ms
2020-03-01 10:06:45.170 INFO 16392 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-03-01 10:06:45.399 INFO 16392 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2020-03-01 10:06:45.401 INFO 16392 --- [main] com.itheima.RabbitmqDemo01Application : Started RabbitmqDemo01Application in 2.353 seconds (JVM running for 3.287)
2020-03-01 10:07:16.373 INFO 16392 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-03-01 10:07:16.374 INFO 16392 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-03-01 10:07:16.381 INFO 16392 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 7 ms
2020-03-01 10:07:16.400 INFO 16392 --- [nio-8080-exec-1] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2020-03-01 10:07:16.428 INFO 16392 --- [nio-8080-exec-1] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#3be8821f:0/SimpleConnection@
2020-03-01 10:07:16.470 ERROR 16392 --- [ 127.0.0.1:5672] o.s.a.r.c.CachingConnectionFactory : Channel shutdown: channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT_FOUND - no exchange 'exchange_direct_demo02' in vhost '/', clas
发送消息到交换机失败, 原因是: channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT_FOUND - no exchange 'exchange_direct_demo02' in vhost '/', clas
```

总结：

1. 发送放可以根据confirm机制来确保是否消息已经发送到交换机
2. confirm机制能保证消息发送到交换机有回调，不能保证消息转发到queue有回调

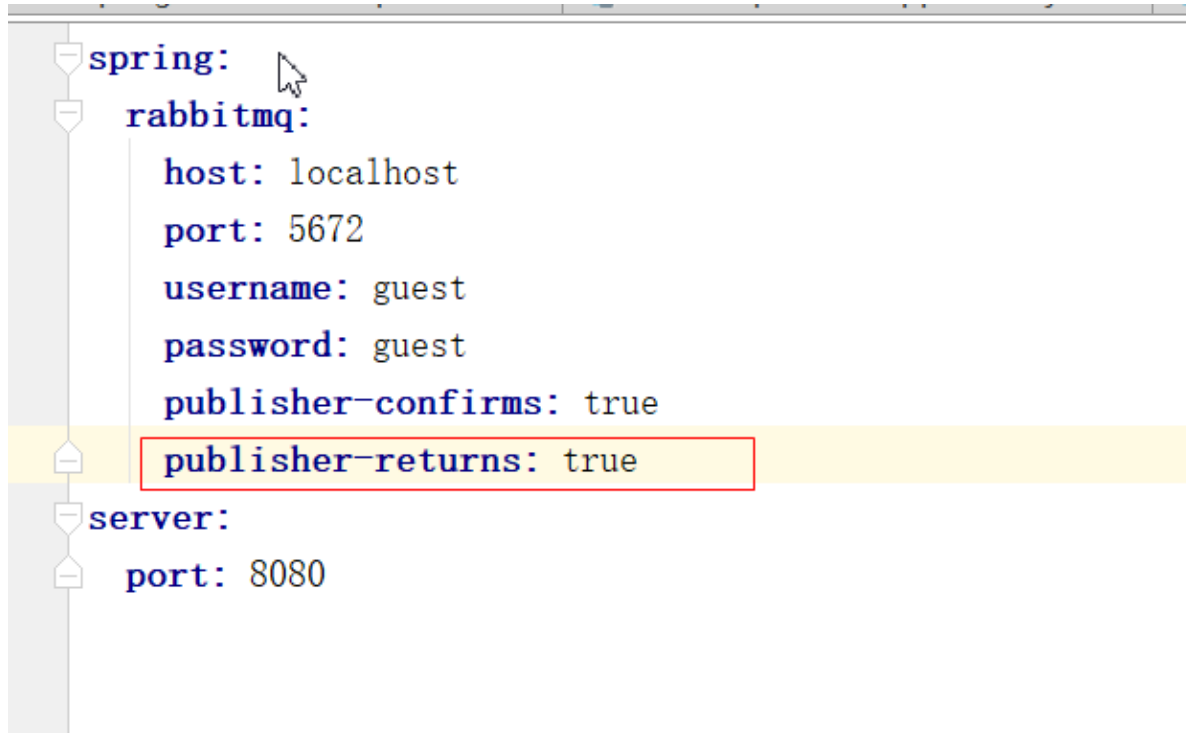
1.1.2 returncallback代码实现

如上，已经实现了消息发送到交换机上的内容，但是如果是，交换机发送成功，但是在路由转发到队列的时候，发送错误，此时就需要用到returncallback模式了。接下来我们实现下。

实现步骤如下：

1. 开启returncallback模式
2. 设置回调函数
3. 发送消息

(1) 配置yaml开启returncallback



(2) 编写returncallback代码:

```
@Component
public class MyReturnCallback implements RabbitTemplate.ReturnCallback {
    /**
     *
     * @param message 消息信息
     * @param replyCode 退回的状态码
     * @param replyText 退回的信息
     * @param exchange 交换机
     * @param routingKey 路由key
     */
    @Override
    public void returnedMessage(Message message, int replyCode, String
replyText, String exchange, String routingKey) {
        System.out.println("退回的消息是: "+new String(message.getBody()));
        System.out.println("退回的replyCode是: "+replyCode);
        System.out.println("退回的replyText是: "+replyText);
        System.out.println("退回的exchange是: "+exchange);
        System.out.println("退回的routingKey是: "+routingKey);
    }
}
```

(3) 发送消息

我们发送正确的交换机，但是发送错误的routingkey测试下

```
@Autowired
private RabbitTemplate.ReturnCallback myReturnCallback;

I
@RequestMapping("/send2")
public String send2() {
    //设置回调函数
    rabbitTemplate.setReturnCallback(myReturnCallback);
    //发送消息
    rabbitTemplate.convertAndSend(exchange: "exchange_direct_demo01", routingKey: "item.insert1234", object: "hello insert");
    return "ok";
}
```

```
@Autowired
private RabbitTemplate.ReturnCallback myReturnCallback;

@RequestMapping("/send2")
public String send2() {
    //设置回调函数
    rabbitTemplate.setReturnCallback(myReturnCallback);
    //发送消息
    rabbitTemplate.convertAndSend("exchange_direct_demo01", "item.insert1234",
    "hello insert");
    return "ok";
}
```

(4) 测试发送请求出现如下，说明测试成功。

```
2020-03-01 10:22:34.759 INFO 5168 --- [nio-8080-exec-1] o.s.web.servlet
2020-03-01 10:22:34.763 INFO 5168 --- [nio-8080-exec-1] o.s.web.servlet
2020-03-01 10:22:34.783 INFO 5168 --- [nio-8080-exec-1] o.s.a.r.c.Cachi
2020-03-01 10:22:34.811 INFO 5168 --- [nio-8080-exec-1] o.s.a.r.c.Cachi
```

退回的消息是: hello insert

退回的replyCode是: 312

退回的replyText是: NO_ROUTE

退回的exchange是: exchange_direct_demo01

退回的routingKey是: item.insert1234

如果我们两个都设置了那么就变成这样：

```
30 public String send2() {
31     rabbitTemplate.setConfirmCallback(myConfirmCallback);
32     //设置回调函数
33     rabbitTemplate.setReturnCallback(myReturnCallback);
34     //发送消息
35     rabbitTemplate.convertAndSend(exchange: "exchange_direct_demo01", routingKey: "item.insert1234", object: "hell
36     return "ok";
37 }
38
39
TestController > send2()
Run RabbitmqDemo01Application
Console Endpoints
2020-03-01 10:23:59.929 INFO 7860 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext
2020-03-01 10:24:00.193 INFO 7860 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorServ
2020-03-01 10:24:00.441 INFO 7860 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s)
2020-03-01 10:24:00.444 INFO 7860 --- [main] com.itheima.RabbitmqDemo01Application : Started RabbitmqDemo01App
2020-03-01 10:24:31.677 INFO 7860 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring Dispe
2020-03-01 10:24:31.677 INFO 7860 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dis
2020-03-01 10:24:31.682 INFO 7860 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization
2020-03-01 10:24:31.702 INFO 7860 --- [nio-8080-exec-1] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to:
2020-03-01 10:24:31.732 INFO 7860 --- [nio-8080-exec-1] o.s.a.r.c.CachingConnectionFactory : Created new connection: r
退回的消息是: hello insert
退回的replyCode是: 312
退回的replyText是: NO_ROUTE
退回的exchange是: exchange_direct_demo01
退回的routingKey是: item.insert1234
发送消息到交换机成功, null
```

(5) 小结

- + returncallback模式，需要手动设置开启
- + 该模式 指定 在路由的时候发送错误的时候调用回调函数，不影响消息发送到交换机

1.1.2 两种模式的总结

confirm模式用于在消息发送到交换机时机使用，return模式用于在消息被交换机路由到队列中发送错误时使用。

但是一般情况下我们使用confirm即可，因为路由key 由开发人员指定，一般不会出现错误。如果要保证消息在交换机和routingkey的时候那么需要结合两者的方式来进行设置。

1.2 消费者确认机制（ACK）

上边我们学习了发送方的可靠性投递，但是在消费方也有可能出现问题，比如没有接受消息，比如接受到消息之后，在代码执行过程中出现了异常，这种情况下我们需要额外的处理，那么就需要手动进行确认签收消息。rabbitmq给我们提供了一个机制：ACK机制。

ACK机制：有三种方式

- 自动确认 acknowledge="none"
- 手动确认 acknowledge="manual"
- 根据异常情况来确认（暂时不怎么用） acknowledge="auto"

解释：

其中自动确认是指：

当消息一旦被Consumer接收到，则自动确认收到，并将相应 message 从 RabbitMQ 的消息缓存中移除。但是在实际业务处理中，很可能消息接收到，业务处理出现异常，那么该消息就会丢失。

其中手动确认方式是指：

则需要在业务处理成功后，调用channel.basicAck()，手动签收，如果出现异常，则调用channel.basicNack()等方法，让其按照业务功能进行处理，比如：重新发送，比如拒绝签收进入死信队列等等。

1.2.1 ACK代码实现

实现的步骤：

1. 创建普通消息监听器监听消息
2. 修改controller 发送正确消息测试
3. 设置配置文件开启ack手动确认，默认是自动确认
4. 修改消息监听器进行手动确认业务判断逻辑

(1) 创建普通消息监听器

```
package com.itheima.listener;

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

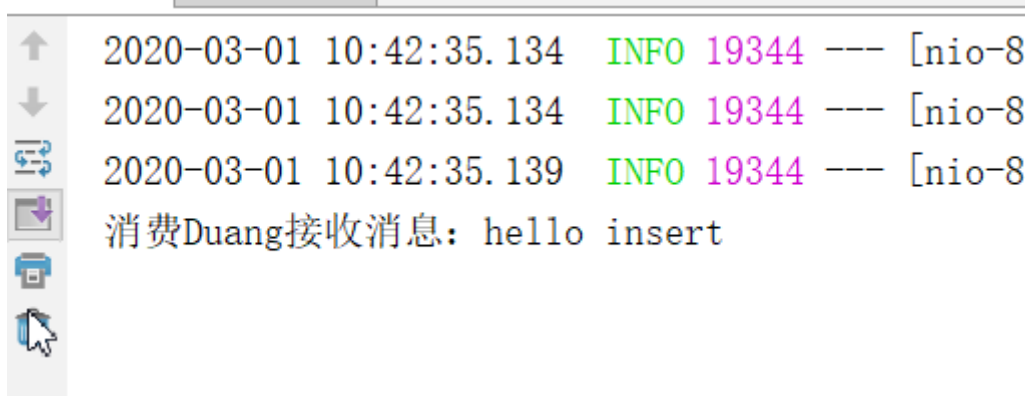
@Component
@RabbitListener(queues = "queue_demo01")
public class MyRabbitListener {

    @RabbitHandler
    public void msg(String message) {
        System.out.println("消费Duang接收消息: " + message);
    }
}
```

(2) 修改Testcontroller方法用于测试发送正确消息：

```
/**
 * 发送正确消息
 * @return
 */
@RequestMapping("/send3")
public String send3() {
    //设置回调函数
    //发送消息
    rabbitTemplate.convertAndSend("exchange_direct_demo01", "item.insert",
    "hello insert");
    return "ok";
}
```

测试OK，如下图：



(3) 设置yml设置为手动确认模式



(4) 修改监听类

如下，此时我们并没有手动签收，就是不签收消息

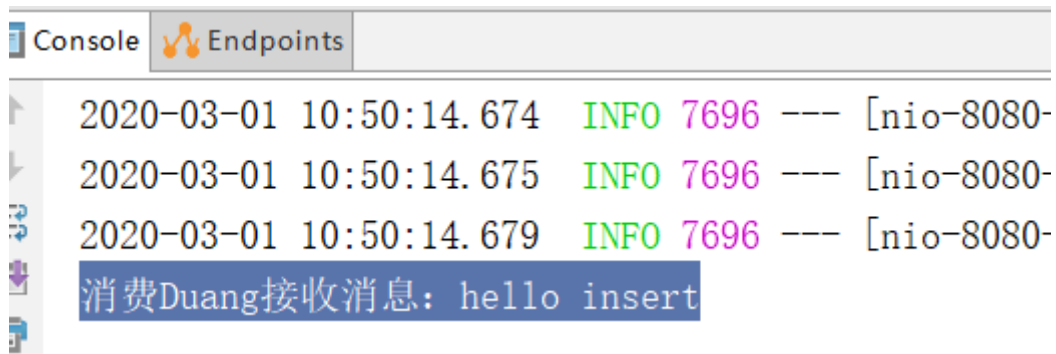


(5)测试：

发送消息之后，队列中出现：

Overview				Messages			Message rates				+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
/	queue_demo01	D	idle	0	1	1	0.00/s	0.00/s	0.00/s		
/szihteima	ps_direct_queue1	D	idle	0	0	0	0.00/s	0.00/s	0.00/s		

控制台打印：



说明一直没有被签收。

1.2.2 ACK确认的方式

ack确认方式有几种：

- 签收消息
- 拒绝消息 批量处理/单个处理

以上可以根据不同的业务进行不同的选择。需要注意的是，如果拒绝签收，下一次启动又会自动的进行消费。

监听代码的业务实现步骤：

```
//接收消息
//处理本地业务
//签收消息
//如果出现异常，则拒绝消息 可以重回队列 也可以丢弃 可以根据业务场景来
```

```
@RabbitHandler
public void msg(Message message, Channel channel, String msg) {
    //接收消息
    System.out.println("消费Duang接收消息: " + msg);
    try {
        //处理本地业务
        System.out.println("处理本地业务开始=====start=====");
        Thread.sleep(2000);
        System.out.println("处理本地业务结束=====end=====");
        //签收消息

    } catch (Exception e) {
        e.printStackTrace();
        //如果出现异常，则拒绝消息 可以重回队列 也可以丢弃 可以根据业务场景来
    }
}
```

第一种：签收

`channel.basicAck()`

第二种：拒绝签收 批量处理

`channel.basicNack()`

第三种：拒绝签收 不批量处理

`channel.basicReject()`

正常则签收，不正常则进行丢弃处理。

```
@RabbitHandler
public void msg(Message message, Channel channel, String msg) {
    //接收消息
    System.out.println("消费Duang接收消息: " + msg);
    try {
        //处理本地业务
        System.out.println("处理本地业务开始=====start=====");
        Thread.sleep(2000);
        int i=1/0;
        System.out.println("处理本地业务结束=====end=====");
        //签收消息
        channel.basicAck(message.getMessageProperties().getDeliveryTag(),
false);
    } catch (Exception e) {
        e.printStackTrace();
        //如果出现异常，则拒绝消息 可以重回队列 也可以丢弃 可以根据业务场景来
        try {

channel.basicNack(message.getMessageProperties().getDeliveryTag(),false,false);

//channel.basicReject(message.getMessageProperties().getDeliveryTag(),false);
        } catch (Exception e1) {
            e1.printStackTrace();
        }
    }
}
```

消息丢弃，则没有消息存在。

Overview				Messages			Message rates		
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	queue_demo01	D	idle	0	0	0	0.00/s	0.00/s	0.00/s
/	no_direct_queue1		idle	0	0	0	0.00/s	0.00/s	0.00/s

正常则签收，不正常再重回队列进行再次投递：

`channel.basicNack („true)` 第三个参数设置为重回队列进行再次投递。

```
@RabbitHandler
public void msg(Message message, Channel channel, String msg) {
    //接收消息
    System.out.println("消费Duang接收消息: " + msg);
```

```

try {
    //处理本地业务
    System.out.println("处理本地业务开始=====start=====");
    Thread.sleep(2000);
    int i = 1 / 0;
    System.out.println("处理本地业务结束=====end=====");
    //签收消息
    channel.basicAck(message.getMessageProperties().getDeliveryTag(),
false);
} catch (Exception e) {
    e.printStackTrace();
    //如果出现异常，则拒绝消息 可以重回队列 也可以丢弃 可以根据业务场景来
    try {
        if (message.getMessageProperties().getRedelivered()) {
            //消息已经重新投递，不需要再次投递
            System.out.println("已经投递一次了");
        } else {
            //第三个参数：设置是否重回队列

            channel.basicNack(message.getMessageProperties().getDeliveryTag(), false,
true);
        }

        //channel.basicReject(message.getMessageProperties().getDeliveryTag(),false);
    } catch (Exception e1) {
        e1.printStackTrace();
    }
}
}

```

如图：还有没确认的消息，下次继续可以消费。

Overview				Messages			Message rates		
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	queue_demo01	D	idle	0	1	1	0.00/s	0.00/s	0.00/s

1.2.3 小结

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

如何保证消息的高可靠性传输？

1. 持久化

- `exchange`要持久化
- `queue`要持久化
- `message`要持久化

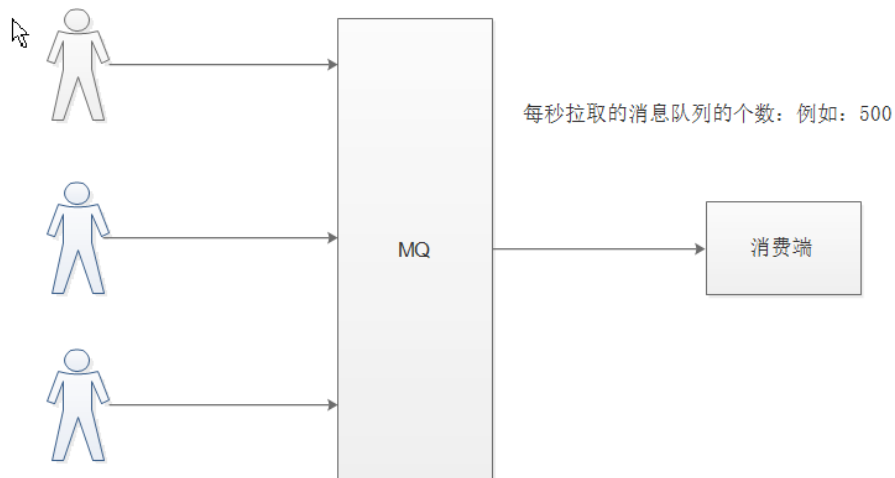
2. 生产方确认Confirm、Return

3. 消费方确认Ack

1.3 消费端限流

1.3.1 消费端限流说明

如果并发量大的情况下，生产方不停的发送消息，可能处理不了那么多消息，此时消息在队列中堆积很多，当消费端启动，瞬间就会涌入很多消息，消费端有可能瞬间垮掉，这时我们可以在消费端进行限流操作，每秒钟放行多少个消息。这样就可以进行并发量的控制，减轻系统的负载，提供系统的可用性，这种效果往往可以在秒杀和抢购中进行使用。在rabbitmq中也有限流的一些配置。

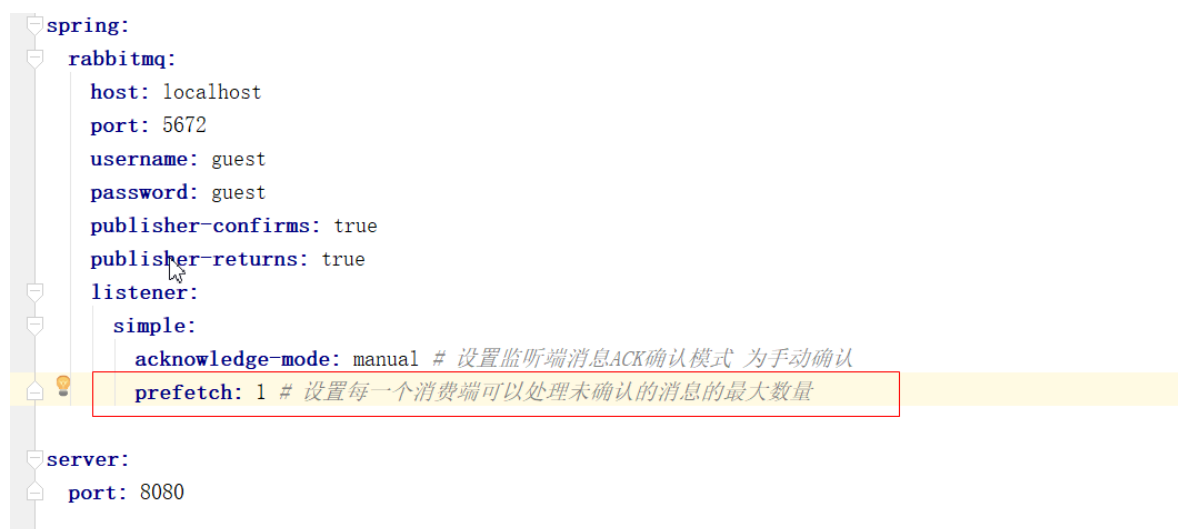


1.3.2 代码实现测试

实现步骤：

1. 设置限流的量
2. 进行测试即可

配置如下：



默认是250个。

消费端代码，模拟每隔一秒钟处理一个消息



测试：并发送10个消息，此时，如下图所示，每一个都是一个处理一个只有等处理完成之后，才能继续处理。

OverviewConnectionsChannelsExchangesQueuesAdmin											
Channels											
All channels (11)											
Pagination											
Page 1 of 1 - Filter: <input type="text"/> <input type="checkbox"/> Regex ?											
Overview				Details				Message rates			
Channel	Virtual host	User name	Mode ?	State	Unconfirmed	Prefetch ?	Unacked	publish	confirm	deliver / get	ack
127.0.0.1:54234 (1)	/	guest	C	running	0	1	1			1.0/s	1.0/s

1.4 TTL

TTL 全称 Time To Live（存活时间/过期时间）。当消息到达存活时间后，还没有被消费，会被自动清除。

RabbitMQ设置过期时间有两种：

- 针对某一个队列设置过期时间；队列中的所有消息在过期时间到之后，如果没有被消费则被全部清除
- 针对某一个特定的消息设置过期时间；队列中的消息设置过期时间之后，如果这个消息没有被消息则被清除。

需要注意一点的是：

针对某一个特定的消息设置过期时间时，一定是消息在队列中在队头的时候进行计算，如果某一个消息A设置过期时间5秒，消息B在队头，消息B没有设置过期时间，B此时过了已经5秒钟了还没被消费。注意，此时A消息并不会被删除，因为它并没有再队头。

一般在工作当中，单独使用TTL的情况较少。我们后面会讲到延时队列。在这里有用处。

演示TTL 代码步骤：

- 1.创建配置类配置 过期队列 交换机 和绑定
- 2.创建controller 测试发送消息

(1) 创建配置类：

```
package com.itheima.config;

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

@Configuration
public class TtlConfig {

    //创建过期队列
    @Bean
    public Queue createqueuettl1(){
        //设置队列过期时间为10000 10s钟
        return QueueBuilder.durable("queue_demo02").withArgument("x-message-ttl",10000).build();
    }

    //创建交换机
    @Bean
    public DirectExchange createExchangettl1(){
        return new DirectExchange("exchange_direct_demo02");
    }

    //创建绑定
    @Bean
    public Binding createBindingttl1(){
        return
        BindingBuilder.bind(createqueuettl1()).to(createExchangettl1()).with("item.ttl");
    }
}
```

(2)创建controller测试

```

/**
 * 发送 ttl测试相关的消息
 * @return
 */
@RequestMapping("/send4")
public String send4() {
    //设置回调函数
    //发送消息
    rabbitTemplate.convertAndSend("exchange_direct_demo02", "item.ttl",
    "hello ttl哈哈");
    return "ok";
}
}

```

(3)测试

过10S钟之后，该数据就都被清0

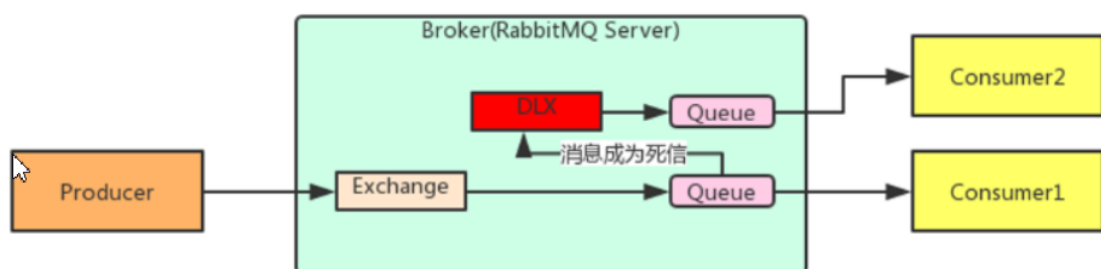
/	queue_demo02	D	TTL	idle	2	0	2	0.00/s		
/	exchange_demo01			idle	0	0	0	0.00/s	0.00/s	0.00/s

1.5 死信队列

1.5.1 死信队列的介绍

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

如下图的过程：



成为死信的三种条件：

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

1.5.2 死信的处理过程

DLX也是一个正常的Exchange，和一般的Exchange没有区别，它能在任何的队列上被指定，实际上就是设置某个队列的属性。

当这个队列中有死信时，RabbitMQ就会自动的将这个消息重新发布到设置的Exchange上去，进而被路由到另一个队列。

可以监听这个队列中的消息做相应的处理。

1.5.3 死信队列的设置

刚才说到死信队列也是一个正常的exchange.只需要设置一些参数即可。

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

如上图所示：

- 1.创建queue1 正常队列 用于接收死信队列过期之后转发过来的消息
- 2.创建queue2 可以针对他进行参数设置 死信队列
- 3.创建交换机 死信交换机
- 4.绑定正常队列到交换机

(1)创建配置类用于配置死信队列 死信交换机 死信路由 和正常队列

```
package com.itheima.config;

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

@Configuration
public class DlxConfig {

    //正常的队列      接收死信队列转移过来的消息
    @Bean
    public Queue createQueueDlq(){
        return QueueBuilder.durable("queue_demo03").build();
    }

    //死信队列      ---->将来消息发送到这里
    @Bean
    public Queue createQueueDdlq(){
        return QueueBuilder
            .durable("queue_demo03_deq")
            .withArgument("x-message-ttl",10000)//设置队列的消息过期时间
            .withArgument("x-dead-letter-
exchange","exchange_direct_demo03_dlx")//设置死信交换机
            .withArgument("x-dead-letter-routing-key","item.dlx")//设置死信路由key
            .build();
    }

    //创建交换机

    @Bean
    public DirectExchange createExchangedel(){
        return new DirectExchange("exchange_direct_demo03_dlx");
    }
}
```



```

//创建绑定 将正常队列绑定到死信交换机上
@Bean
public Binding createBindingdel(){
    return
    BindingBuilder.bind(createqueuettlq()).to(createExchangedel()).with("item.dlx");
}
}

```

(2)添加controller的方法用于测试

```

/**
 * 测试发送死信队列
 * @return
 */
@RequestMapping("/send5")
public String send5() {
    //发送消息到死信队列 可以使用默认的交换机 指定outingkey为死信队列名即可

    rabbitTemplate.convertAndSend("queue_demo03_deq", "hello dlx哈哈");
    return "ok";
}

```

1.5.3.1 测试超时进入死信

测试：

浏览器中输入：

<http://localhost:8080/test/send5>

查看控制台：

队列数据为1

Overview				Messages			Message rates				
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver	/ get	ack	
/	queue_demo01	D	idle	0	0	0	0.00/s		0.00/s	0.00/s	
/	queue_demo02	D TTL	idle	0	0	0	0.00/s				
/	queue_demo03	D	idle	2	0	2			0.00/s	0.00/s	
/	queue_demo03_deq	D TTL DLX DLK	idle	1	0	1	0.20/s				
/szitheima	ps_direct_queue1	D	idle	0	0	0	0.00/s		0.00/s	0.00/s	

经过10S钟之后： 变成0 由此上边的demo03正常队列中多了一个消息。

Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver	/ get	ack
/	queue_demo01	D	idle	0	0	0	0.00/s		0.00/s	0.00/s
/	queue_demo02	D TTL	idle	0	0	0	0.00/s			
/	queue_demo03	D	idle	3	0	3			0.00/s	0.00/s
/	queue_demo03_deq	D TTL DLX DLK	idle	0	0	0	0.00/s			
/szitheima	ps_direct_queue1	D	idle	0	0	0	0.00/s		0.00/s	0.00/s

至此，我们测试了过期进入死信队列。

1.5.3.2 测试拒绝签收进入死信

(1) 创建监听器

```
@Component
@RabbitListener(queues = "queue_demo03_deq")
public class DLXListner {
    @RabbitHandler
    public void lis(Message message, Channel channel, String msg){
        System.out.println("消息是:"+msg);
        try {
            System.out.println("我拒绝签收");

            channel.basicAck(message.getMessageProperties().getDeliveryTag(), false, false);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

(2) 发送消息测试如下:

```
2020-03-01 15:49:08.824 INFO 21128 --- [nio-8080-exec-1] o.s.
2020-03-01 15:49:08.829 INFO 21128 --- [nio-8080-exec-1] o.s.
消息是:hello dlx哈哈
我拒绝签收
```

控制台由原来的3立即变成4 不需要等待10S

	queue_demo03	D	TTL	DLX	DLK	idle	4	0	4	
/	queue_demo03_deq	D	TTL	DLX	DLK	idle	0	0	0	0.00/s

1.5.3.3 测试设置长度进入死信

(1) 修改配置, 添加队列长度参数

```
@Bean
public Queue createqueueetdelq2() {
    return QueueBuilder
        .durable("queue_demo03_deq")
        .withArgument("x-max-length", 1) //设置队列的最大消息条数
        .withArgument("x-message-ttl", 10000) //设置队列的消息过期时间
        .withArgument("x-dead-letter-exchange", "exchange_direct_demo03_dlx") //设置死信交换
        .withArgument("x-dead-letter-routing-key", "item.dlx") //设置死信路由key
        .build();
}
```

//创建交换机

(2) 再控制台中删除交换机 队列 重新启动系统, 才能生效。

(3) 测试:

点击发送4次消息，如图立即有3条进入死信，还有一条在队列中，等10S钟之后，也会进入死信。

Overview				Messages				Message
Virtual host	Name	Features	State	Ready	Unacked	Total		incoming
/	queue_demo01	D	idle	0	0	0	0	0.0
/	queue_demo02	D TTL	idle	0	0	0	0	0.0
/	queue_demo03	D	idle	3	0	3		
/	queue_demo03_deq	D TTL Lim DLX DLK	idle	1	0	1	0.0	
/szihteima	ps_direct_queue1	D	idle	0	0	0	0.0	

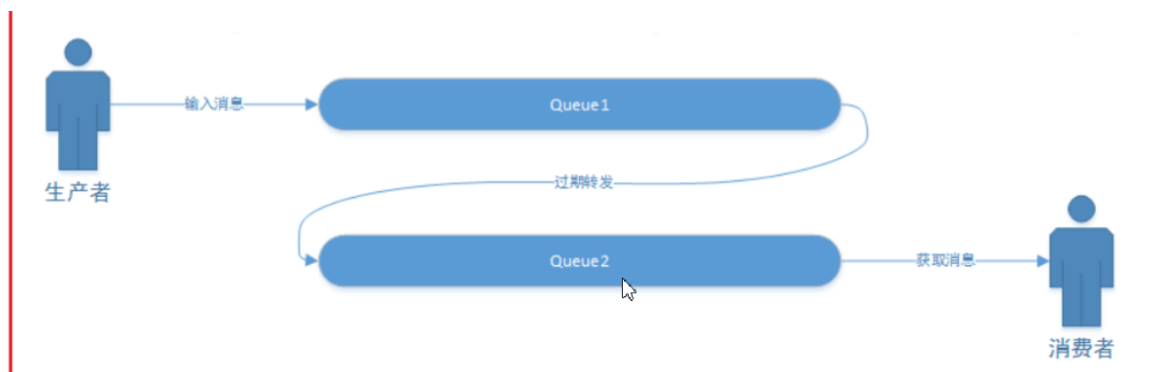
10S钟之后：

Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	Unacked	get	ack
/	queue_demo01	D	idle	0	0	0	0.00/s	0.00/s	0.00/s	
/	queue_demo02	D TTL	idle	0	0	0	0.00/s			
/	queue_demo03	D	idle	4	0	4				
/	queue_demo03_deq	D TTL Lim DLX DLK	idle	0	0	0	0.00/s			

1.6 延迟队列

延迟队列，即消息进入队列后不会立即被消费，只有到达指定时间后，才会被消费。在rabbitmq中，并没有延迟队列概念，但是我们可以使用ttl和死信队列的方式进行达到延迟的效果。这种需求往往在某些应用场景中出现。当然还可以使用插件。

如图所示：



1. 生产者产生一个消息发送到queue1
2. queue1中的消息过期则转发到queue2
3. 消费者在queue2中获取消息进行消费

如上场景中 典型的案例：下订单之后，30分钟如果还未支付则，取消订单回滚库存。我们来模拟下需求：

(1) 创建配置类

```
@Configuration
public class DelayConfig {
    //正常的队列    接收死信队列转移过来的消息
    @Bean
    public Queue createQueue2(){
        return QueueBuilder.durable("queue_order_queue2").build();
    }
}
```

```

//死信队列 --->将来消息发送到这里 这里不设置过期时间，我们应该在发送消息时设置某一个消息（某一个用户下单的）的过期时间
@Bean
public Queue createQueue1(){
    return QueueBuilder
        .durable("queue_order_queue1")
        .withArgument("x-dead-letter-exchange", "exchange_order_delay")//
设置死信交换机
        .withArgument("x-dead-letter-routing-key", "item.order")//设置死信
路由key
        .build();
}

//创建交换机
@Bean
public DirectExchange createOrderExchangeDelay(){
    return new DirectExchange("exchange_order_delay");
}

//创建绑定 将正常队列绑定到死信交换机上
@Bean
public Binding createBindingDelay(){
    return
BindingBuilder.bind(createQueue2()).to(createOrderExchangeDelay()).with("item.order");
}
}

```

(2)修改controller

```

/**
 * @RequestMapping("/send6")
 * public String send6() {
 *     //发送消息到死信队列 可以使用默认的交换机 指定outingkey为死信队列名即可
 *     System.out.println("用户下单成功，10秒钟之后如果没有支付，则过期，回滚订单");
 *
 *     rabbitTemplate.convertAndSend("queue_order_queue1", (Object) "哈哈我要检查你是否有支付", new MessagePostProcessor() {
 *         @Override
 *         public Message postProcessMessage(Message message) throws AmqpException {
 *             message.getMessageProperties().setExpiration("10000");//设置该消息的过期时间 该消息为设置过期时间
 *             return message;
 *         }
 *     });
 *     return "用户下单成功，10秒钟之后如果没有支付，则过期，回滚订单";
 * }
 */

```

注意：发送消息要发送到queue1,监听消息要监听queue2

```

/**
 * 发送下单
 *
 * @return
 */
@RequestMapping("/send6")
public String send6() {
    //发送消息到死信队列 可以使用默认的交换机 指定outingkey为死信队列名即可
    System.out.println("用户下单成功，10秒钟之后如果没有支付，则过期，回滚订单");
}

```

```

        System.out.println("时间: "+new Date());
        rabbitTemplate.convertAndSend("queue_order_queue1", (Object) "哈哈我要检查你是
        否有支付", new MessagePostProcessor() {
            @Override
            public Message postProcessMessage(Message message) throws AmqpException
            {
                message.getMessageProperties().setExpiration("10000");//设置该消息的过
                期时间
                return message;
            }
        });
        return "用户下单成功, 10秒钟之后如果没有支付, 则过期, 回滚订单";
    }
}

```

(3) 设置监听类

注意, 监听消息要监听queue2, 发送消息要发送queue1

```

@Component
@RabbitListener(queues = "queue_order_queue2")
public class OrderListener {

    @RabbitHandler
    public void orderhandler(Message message, Channel channel, String msg) {
        System.out.println("获取到消息:" + msg + ":时间为:" + new Date());
        try {
            System.out.println("模拟检查开始=====start");
            Thread.sleep(1000);
            System.out.println("模拟检查结束=====end");
            System.out.println("用户没付款, 检查没通过, 进入回滚库存处理");
            channel.basicAck(message.getMessageProperties().getDeliveryTag(),
            false);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

(4)测试,浏览器发送路径: <http://localhost:8080/test/send6>

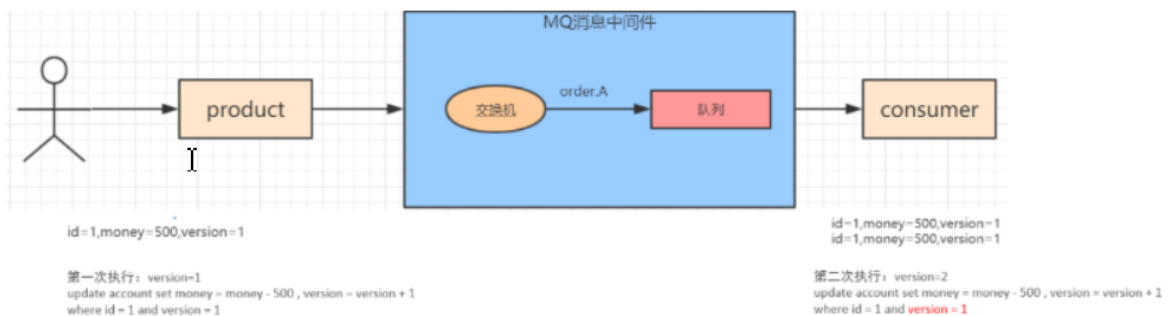
用户下单成功, 10秒钟之后如果没有支付, 则过期, 回滚订单

测试如下图

2020-03-01 16:20:29.101 INFO 10000 INFO 0000 EXEC 1] O.S. Web Service Dispatching
用户下单成功，10秒钟之后如果没有支付，则过期，回滚订单
时间：Sun Mar 01 16:20:29 CST 2020
获取到消息：哈哈我要检查你是否有支付：时间为：Sun Mar 01 16:20:39 CST 2020
模拟检查开始=====start
模拟检查结束=====end
用户没付款，检查没通过，进入回滚库存处理

2 rabbitmq应用的问题

幂等性指一次和多次请求某一个资源，对于资源本身应该具有同样的结果。也就是说，其任意多次执行对资源本身所产生的影响均与一次执行的影响相同。在MQ中指，消费多条相同的消息，得到与消费该消息一次相同的结果。



以转账为例：

1. 发送消息
2. 消息内容包含了id 和 版本和 金额
3. 消费者接收到消息，则根据ID 和版本执行sql语句，
`update account set money=money-?,version=version+1 where id=? and version=?`
4. 如果消费第二次，那么同一个消息内容是修改不成功的。

3.RabbitMQ集群(了解)

实际生产应用中都会采用消息队列的集群方案，如果选择RabbitMQ那么有必要了解下它的集群方案。一般来说，如果只是为了学习RabbitMQ或者验证业务工程的正确性那么在本地环境或者测试环境上使用其单实例部署就可以了，但是出于MQ中间件本身的可靠性、并发性、吞吐量和消息堆积能力等问题的考虑，在生产环境上一般都会考虑使用RabbitMQ的集群方案。

3.1 rabbitmq集群通信原理

RabbitMQ这款消息队列中间件产品本身是基于Erlang编写，Erlang语言天生具备分布式特性（通过同步Erlang集群各节点的magic cookie来实现）。因此，RabbitMQ天然支持Clustering。集群是保证可靠性的一种方式，同时可以通过水平扩展以达到增加消息吞吐量能力的目的，这里只需要保证erlang_cookie的参数一致集群即可通信。

rabbitmq集群包括两种：普通集群和镜像集群。

普通集群有缺点也有优点，镜像集群有缺点也有优点。

大致上,

如果是普通集群: 那么每一个节点的数据, 存储了另外一个节点的元数据, 当需要使用消息时候, 从另外一台节点拉取数据, 这样性能很高, 但是性能瓶颈发生在单台服务器上。而且宕机有可能出现消息丢失。

如果镜像集群, 那么在使用时候, 每个节点都相互通信互为备份, 数据共享。那么这样一来使用消息时候, 就直接获取, 不再零时获取, 但是缺点就是消耗很大的性能和带宽。

3.2 rabbitmq集群搭建

rabbitmq集群搭建, 这里我们采用docker的方式来进行搭建, 由于docker还没学习, 那么我们了解即可。

准备一个虚拟机 里面安装docker引擎。这里为了测试我们采用2台rabbitmq的实例, 也就是两个docker容器来模拟2个rabbitmq服务器。

准备一个虚拟机 里面安装docker引擎。这里为了测试我们采用2台rabbitmq的实例, 也就是两个docker容器来模拟2个rabbitmq服务器。

- 准备一台虚拟机 我的机器ip为192.168.211.128 .也可以使用畅购的虚拟机。

```
192.168.211.128 x
[root@A ~]#
[root@A ~]#
[root@A ~]#
[root@A ~]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:1ff:fe00:c566 prefixlen 64 scopeid 0x20<link>
    ether 02:42:01:00:c5:66 txqueuelen 0 (Ethernet)
    RX packets 5731 bytes 5367554 (5.1 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 6939 bytes 2473531 (2.3 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.211.128 netmask 255.255.255.0 broadcast 192.168.211.255
    inet6 fe80::20c:29ff:fea8:4b2a prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:a8:4b:2a txqueuelen 1000 (Ethernet)
    RX packets 123520 bytes 169383776 (161.5 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 57620 bytes 11075006 (10.5 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

- 安装docker引擎

这个不再演示

3.2.1 拉取镜像

执行命令:

```
docker pull rabbitmq:3.6.15-management
```

3.2.2 创建rabbitmq容器

- 创建rabbitmq容器1:

```
docker run -d --hostname rabbit1 --name myrabbit1 -p 15672:15672 -p 5672:5672 -e RABBITMQ_ERLANG_COOKIE='rabbitcookie' rabbitmq:3.6.15-management
```

- 创建rabbitmq容器2:

```
docker run -d --hostname rabbit2 --name myrabbit2 -p 15673:15672 -p 5673:5672 --link=myrabbit1:rabbit1 -e RABBITMQ_ERLANG_COOKIE='rabbitcookie' rabbitmq:3.6.15-management
```

解释:

`--link <name or id>:alias`

其中, `name`和`id`是源容器的`name`和`id`, `alias`是源容器在`link`下的别名。

`--link` 用于在容器中进行通信的时候需要使用到的。

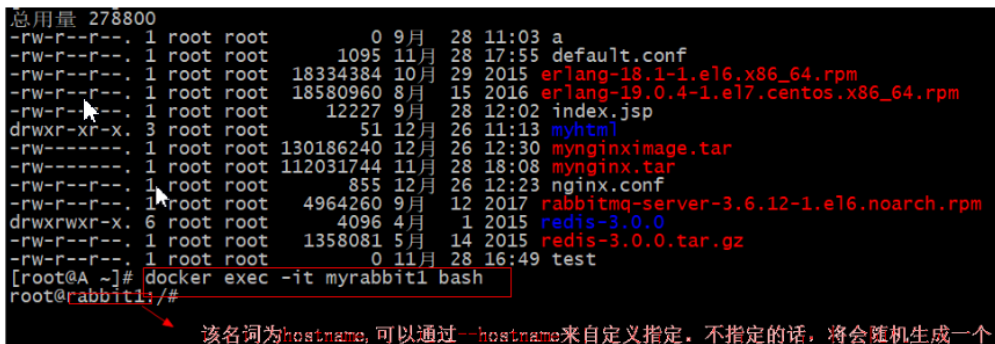
`-e RABBITMQ_ERLANG_COOKIE='rabbitcookie'`

其中 `-e` 设置环境变量 变量名为: `RABBITMQ_ERLANG_COOKIE` 值为: `rabbitcookie` 该值可以任意。但是一定要注意, 两个容器的`cookie`值一定要一样才行。他的作用用于发现不同的节点, 并通过该`cookie`进行自动校验和通信使用。

`--hostname rabbit2`

其中: `--hostname` 用于设置容器内部的`hostname`名称, 如果不设置, 那就会自动随机生成一个`hostname`字, 如下图。

这里一定要设置。因为`rabbitmq`的节点数据进行通信加入集群的时候需要用`hostname`作为集群名称。



```
总用量 278800
-rw-r--r--. 1 root root      0 9月 28 11:03 a
-rw-r--r--. 1 root root    1095 11月 28 17:55 default.conf
-rw-r--r--. 1 root root 18334384 10月 29 2015 erlang-18.1-1.el6.x86_64.rpm
-rw-r--r--. 1 root root 18580960 8月 15 2016 erlang-19.0.4-1.el7.centos.x86_64.rpm
-rw-r--r--. 1 root root    12227 9月 28 12:02 index.jsp
drwxr-xr-x. 3 root root      51 12月 26 11:13 myhtml
-rw-r--r--. 1 root root 130186240 12月 26 12:30 mynginximage.tar
-rw-r--r--. 1 root root 112031744 11月 28 18:08 mynginx.tar
-rw-r--r--. 1 root root      855 12月 26 12:23 nginx.conf
drwxrwxr-x. 6 root root    4096 4月 1 2015 rabbitmq-server-3.6.12-1.el6.noarch.rpm
-rw-r--r--. 1 root root    1358081 5月 14 2015 redis-3.0.0.tar.gz
-rw-r--r--. 1 root root      0 11月 28 16:49 test
[root@A ~]# docker exec -it myrabbit1 bash
root@rabbit1:/#
```

该名词为`hostname`, 可以通过`--hostname`来自定义指定。不指定的话, 将会随机生成一个

3.4 配置rabbitmq集群

这里我们使用 集群名 `rabbit@rabbit1` ,将节点2 加入到节点1号中。

3.4.1 配置rabbit1

- 进入到myrabbit1容器内部

```
docker exec -it myrabbit1 bash
```

- 配置节点

```
rabbitmqctl stop_app  
rabbitmqctl reset  
rabbitmqctl start_app  
exit
```

解释:

```
rabbitmqctl stop_app --- 表示关闭节点  
rabbitmqctl reset --- 重新设置节点配置  
rabbitmqctl start_app --- 重新启动 （此处不需要设置，将该节点作为集群master,其他节点加入到该节点中）  
exit ---退出容器
```

3.4.2 配置rabbitmq2

- 进入到myrabbit2容器内部

```
docker exec -it myrabbit2 bash
```

- 配置节点

```
rabbitmqctl stop_app  
rabbitmqctl reset  
rabbitmqctl join_cluster --ram rabbit@rabbit1  
rabbitmqctl start_app  
exit
```

解释:


```
rabbitmqctl join_cluster --ram rabbit@rabbit1
```

-- 用于将该节点加入到集群中

-- **ram** 设置为内存存储，默认为 **disc** 磁盘存储，如果为磁盘存储可以不用配置**ram**

-- **rabbit@rabbit1** 该配置为节点集群名称：集群名称为：**rabbit@server** 而**server**指定就是hostname的名称。

配置完成，打开web管理界面，如下图所示：



Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory	Disk space	Uptime	Info	Reset stats	+/-
rabbit@rabbit1	23 1048576 available	1 943626 available	336 1048576 available	78MB 1.5GB high watermark	11GB 40MB low watermark	1h 1m	basic disc 1	This node All nodes	
rabbit@rabbit2	20	0	320	79MB	11GB	38m 48s	basic disc 1	This node All nodes	

3.5 配置镜像队列(可选)

如上，我们已经搭建好了集群，但是并不能做到高可用，所以需要配置升级为镜像队列。

在任意的节点（A或者B）中执行如下命令：

```
rabbitmqctl set_policy ha-all "^" '{"ha-mode":"all"}'
```

解释

```
rabbitmqctl set_policy
```

用于设置策略

ha-all

表示设置为镜像队列并策略为所有节点可用，意味着队列会被（同步）到所有的节点，当一个节点被加入到集群中时，也会同步到新的节点中，此策略比较保守，性能相对低，对接使用半数原则方式设置（ $N/2+1$ ），例如：有3个结点 此时可以设置为：**ha-two** 表示同步到2个结点即可。

"^" 表示针对的队列的名称的正则表达式，此处表示匹配所有的队列名称

'{"ha-mode":"all"}' 设置一组key/value的JSON 设置为高可用模式 匹配所有exchange

此时查看web管理界面：添加一个队列itcast111,如下图已经可以出现结果为有一个结点，并且是ha-all模式（镜像队列模式）

▼ All queues (1)

Pagination

Page

1

 of 1 - Filter: ☐ Regexp (?) (?)

Overview				Messages			Message rates			+/-
Name	Node	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
itcast111	A <div>+1</div>	<div>D</div> <div>ha-all</div>	<div>idle</div>	0	0	0				

▼ Add a new queue