# RabbitMQ

## 1.RabbitMQ简介

如果从消息组件来讲主要划分为两类：

* JMS组件：ActiveMQ(速度慢)
* AMQP组件（协议组件）：性能是最高的，而AMQP有两个主要开源的

|-RabbitMQ:使用最为广泛，速度也很快。

|-Kafka:是在大数据时代产生，明日之星。

## 2.安装RabbitMQ

RabbitMQ依赖的开发语言为ErLong,所以如果要想使用RabbitMQ服务，首先一定要在系统之中进行此开发环境的配置。

### 安装ErLong环境

1. 将erlong的开发包“otp\_src\_19.2.tar.gz”下载地址：

<http://www.erlang.org/downloads/19.2>上传到liunx系统之中路径：“/usr/local/src”，而后进行解压缩：

|  |
| --- |
| tar xzvf otp\_src\_19.2.tar.gz |

1. 进入erlang的源代码目录之中对其进行编译处理：

\*一定要保证你的主机上已经配置了JDK;

\*如果想编译erlang源代码，则需要有一个类库的支持：apt-get install libncurses5-dev

建立一个erlang编译后的文件目录：

|  |
| --- |
| mkdir -p /use/local/erlang |

进入源代码目录

|  |
| --- |
| cd otp\_src\_19.2 |

进行编译配置：

|  |
| --- |
| ./configure --prefix=/use/local/erlang/ |

进行编译与安装 这个编译的时间会比较的长

|  |
| --- |
| make&&make install |

3.将erlang开发包配置到环境属性之中

打开配置文件

|  |
| --- |
| vim /etc/profile |

追加新的环境配置：

|  |
| --- |
| export ERLANG\_HOME=/use/local/erlang/  export PATH=$PATH:$ERLANG\_HOME/bin: |

是配置立即生效：

|  |
| --- |
| source /etc/profile |

切换终端后每次都需要重新生效 /ect/profile

解决办法 在/root/.bashrc文件尾部添加:   
source /etc/profile保存后执行:./root/.bashrc就ok了

查看erlang是否安装成功。

|  |
| --- |
| 命令：erl 输出如下信息表示安装成功：  Eshell V5.7.2 (abort with ^G)  测试输出“hello world”  1> io:format("hellow world").  hellow worldok  2>  退出编译器：  halt(). |

### 2.安装并配置RabbitMQ

RabbitMQ的配置过程比较容易，但是这里面有一个小小的坑，一般而言，在Liunx之中所有压缩文件都会以“tar.gz”的格式。可是RabbitMQ如果你通过官方网站下载的时候发现可以使用的是“XZ”文件。

1.将RabbitMQ开发包上传到系统之中，而后对其解压缩：

\*将tar.xz解压缩：xz -d rabbitmq-server-generic-unix-3.7.5.tar.xz

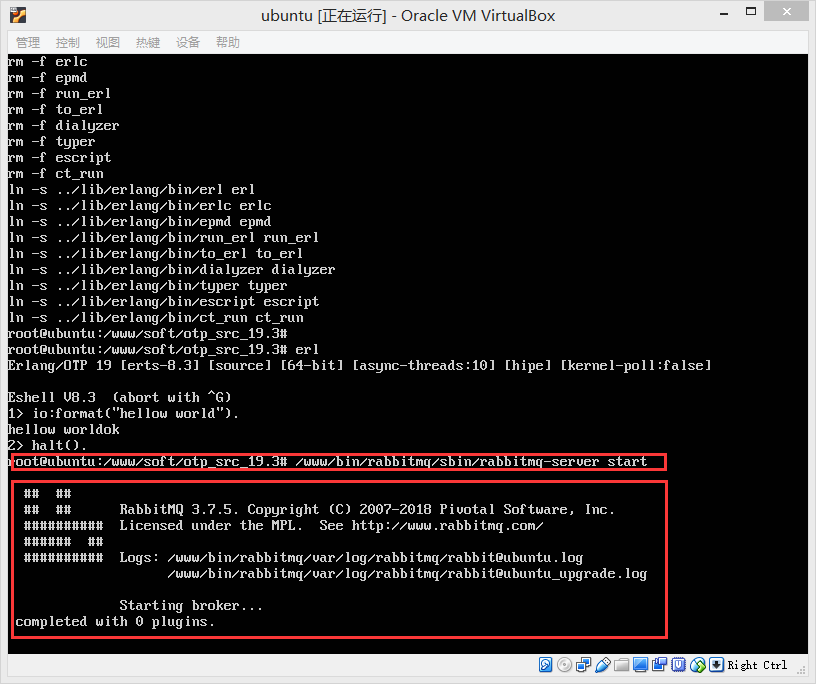
\*解压缩到指定目录：tar -xf rabbitmq-server-generic-unix-3.7.5.tar -C /www/bin/

2.创建软连接，便于管理版本。

ln -s rabbitmq\_server-3.7.5/ rabbitmq

3.启动rabbitmq的服务：

看到如下界面表示RabbitMQ成功启动：



4.RabbitMQ本身自带有系统监控也就是说它存在一个WEB的客户端，但是如果要想自己的操作。就做一个新的管理员账号。

|  |
| --- |
| /www/bin/rabbitmq/sbin/rabbitmqctl add\_user java java |

此时java账号只是一个普通的用户，随后加入到管理员分组之中。

/www/bin/rabbitmq/sbin/rabbitmqctl set\_user\_tags java administrator

5.默认情况下RabbitMQ的服务里面是不带有webServer启动的，需要自己配置启动，执行如下命令：

|  |
| --- |
| rabbitmq-plugins enable rabbitmq\_management |

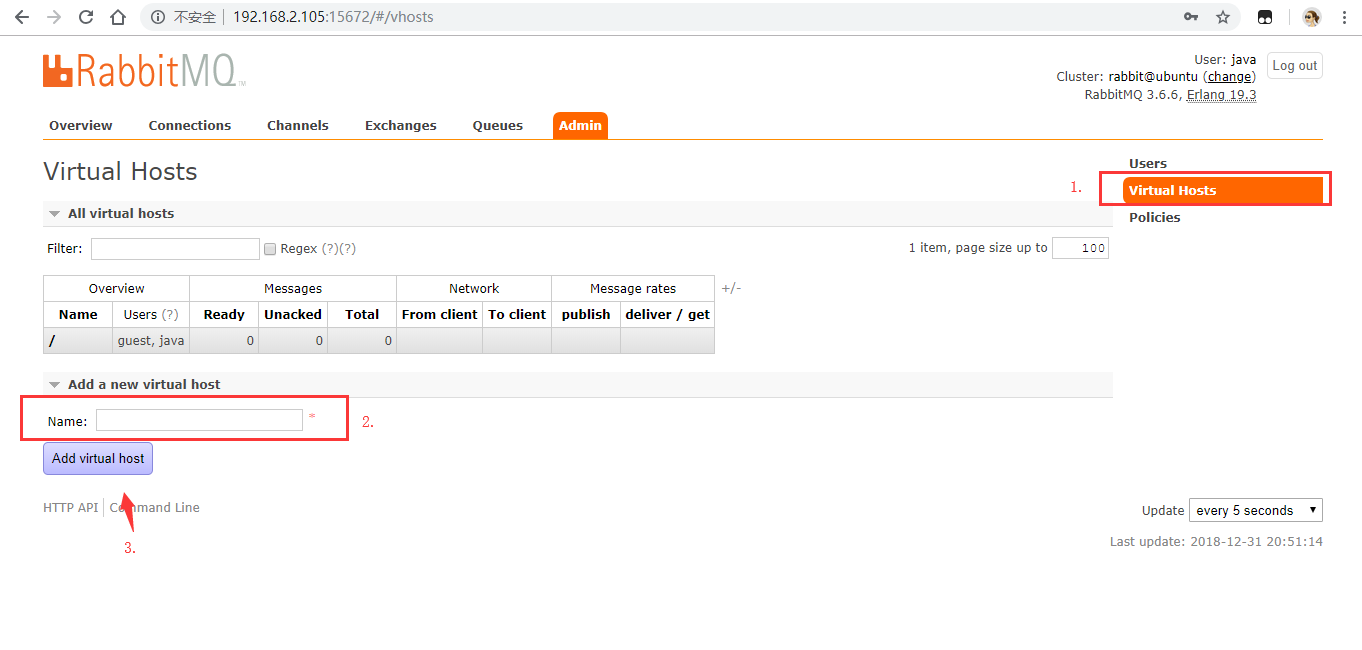
6.在浏览器上访问后台地址：<http://192.168.20.85:15672/>



## Virtual Hosts配置说明

Virtual Hosts简单理解就是对应我们数据库的数据库。

1.通过web界面添加virtual host



2.给用户授权访问



## 4.操作简单队列

### 1.模型

http://www.rabbitmq.com/img/tutorials/python-one.png

### 2.新建项目。引入rabbitmq-cilent jar包

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.rabbitmq</groupId>  <artifactId>amqp-client</artifactId>  <version>4.0.2</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.10</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.7.5</version>  </dependency>  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>1.2.17</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>3.8.1</version>  <scope>test</scope>  </dependency>  </dependencies> |

### 3.创建一个连接RabbitMQ 消息中间的工具类

|  |
| --- |
| **package** com.liuxc.www.rabbitmq.util;  **import** com.rabbitmq.client.Connection; **import** com.rabbitmq.client.ConnectionFactory;  **import** java.io.IOException; **import** java.util.concurrent.TimeoutException;  */\*\*  \** ***@author*** *L  \** ***@date*** *2018/12/31 - 21:29  \*/* **public class** RabbitMQConnectUtil {   **private static** RabbitMQConnectUtil *ourInstance* = **new** RabbitMQConnectUtil();   **private** RabbitMQConnectUtil() {   }   **public static** RabbitMQConnectUtil getInstance() {  **return** *ourInstance*;  }   */\*\*  \* 获取MQ连接  \*/* **public static** Connection getConnection() {  //创建连接工厂  ConnectionFactory factory = **new** ConnectionFactory();  //设置ip  factory.setHost("192.168.2.105");  //设置AMQP协议端口  factory.setPort(5672);  //设置虚拟主机  factory.setVirtualHost("/my-virtual-hosts");  //设置用户名  factory.setUsername("java");  //设置密码  factory.setPassword("java");  **try** {  **return** factory.newConnection();  } **catch** (IOException | TimeoutException e) {  e.printStackTrace();  }  **return null**;  }  } |

### 创建消息-生产者

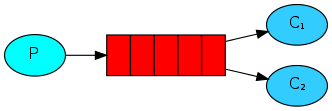
|  |
| --- |
| **public class** Producer {   **private static final** String ***SIMPLE\_QUEUE*** = "SIMPLE\_QUEUE";   **public static void** main(String[] args) **throws** IOException, TimeoutException {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***SIMPLE\_QUEUE***, **false**, **false**, **false**, **null**);  //绑定队列  String msg = "hello world!";  channel.basicPublish("", ***SIMPLE\_QUEUE***, **null**, msg.getBytes());  System.***out***.println("send massage" + msg);  channel.close();  connection.close();  } } |

### 创建消息-消费者

|  |
| --- |
| **public class** Consumer {  **private static final** String ***SIMPLE\_QUEUE*** = "SIMPLE\_QUEUE";   **public static void** main(String[] args) **throws** IOException {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***SIMPLE\_QUEUE***, **false**, **false**, **false**, **null**);  //创建消费者  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  //监听消息  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("consumer msg:" + **new** String(body, "utf-8"));  }  };  //监听队列  channel.basicConsume(***SIMPLE\_QUEUE***, **true**,consumer);  } } |

## 5.工作队列（Work queues）

### 1.模型



### 创建消息-生产者

|  |
| --- |
| **public class** Producer {  **private static final** String ***QUEUE\_NAME*** = "WORK\_QUEUE";   **public static void** main(String[] args) **throws** Exception {  //创建连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //发送消息  **for** (**int** i = 0; i < 50; i++) {  String msg = "work queue " + i;  System.***out***.println("Producer send => " + msg);  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  }  channel.close();  connection.close();  } } |

### 创建消息-消费者

**消费者1**

|  |
| --- |
| **public class** Consumer1 {   **private static final** String ***QUEUE\_NAME*** = "WORK\_QUEUE";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建通道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  //创建消费者，监听消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[Consumer1]接受到消息==> " + **new** String(body, "utf-8"));  //延时一秒，模拟不同服务器的响应时间  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  };  //加入监听  channel.basicConsume(***QUEUE\_NAME***,**true**,consumer);  } } |

**消费者2**

|  |
| --- |
| **public class** Consumer2 {   **private static final** String ***QUEUE\_NAME*** = "WORK\_QUEUE";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建通道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  //创建消费者，监听消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[Consumer2]接受到消息==> " + **new** String(body, "utf-8"));  //延时一秒，模拟不同服务器的响应时间  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  };  //加入监听  channel.basicConsume(***QUEUE\_NAME***,**true**,consumer);  } } |

### 现象

通过以上一个生产者生产消息，多个消费消费同一个队列消息会发现，不管消费者的执行速度快慢，他们消费消息的数量是一样的。

**这种方式叫做轮询分发（round-robin）**

## 6.手动应答（公平分发）

### 前言

由于轮询分发的缺陷，导致不同的服务接受的消息是均分的。所以在轮询分发基础上进行修改实现公平的分发

### 修改生产者

|  |
| --- |
| **public class** Producer {  **private static final** String ***QUEUE\_NAME*** = "WORK\_QUEUE";   **public static void** main(String[] args) **throws** Exception {  //创建连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  /\*  \*在每个消费者发送确认消息之前，消息队列不发送下一个消息到消费者端。  \* 限制队列每次只能发送一个消息到消费者  \*/  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //发送消息  **for** (**int** i = 0; i < 50; i++) {  String msg = "work queue " + i;  System.***out***.println("Producer send => " + msg);  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  }  channel.close();  connection.close();  } } |

### 3.修改消费者

|  |
| --- |
| **public class** Consumer1 {   **private static final** String ***QUEUE\_NAME*** = "WORK\_QUEUE";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建通道  **final** Channel channel = connection.createChannel();  //1次只处理一个消息。处理完成后在接受新的消息  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //创建消费者，监听消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[Consumer1]接受到消息==> " + **new** String(body, "utf-8"));  //延时一秒，模拟不同服务器的响应时间  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  //手动应道  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  }  };  **boolean** autoAck = **false**;//关闭自动应答，使用手动应答.确保每条消息成功执行  //加入监听  channel.basicConsume(***QUEUE\_NAME***, autoAck, consumer);  } } |

## 7.消息应答与持久化

### 1.消息自动应答模式

**boolean** autoAck = **true**;  
channel.basicConsume(***QUEUE\_NAME***, autoAck, consumer);

优点:执行效率高，不需要管消费者应答。一旦RabbitMQ将消息分发给消费者，就会从内存中删除。

缺点：服务一旦宕机，那么发送给消费者的消息就会丢失。

### 2.消息手动应答模式

**boolean** autoAck = false;  
channel.basicConsume(***QUEUE\_NAME***, autoAck, consumer);

优点：队列需要收到消费者的确认后，才会从内存中删除掉该条消息。如果没有收取到消费者的确认，则会继续重发给其他消费者。直到该消息被消费者并确认。这样就保证的就算服务器宕机了，数据也不会丢失。

缺点：执行效率会降低，当队列没有收到消费者确认就会一直重发的发送给消费者。

消息应答默认是false;

### 消息持久化

**boolean** durable=**false**;  
channel.queueDeclare(***QUEUE\_NAME***,durable,**false**,**false**,**null**);

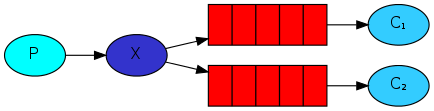
**false：表示队列消息不会持久化到磁盘**

**true:表示队列消息持久化到磁盘**

细节问题：当我们已经生成好的队列，我们将改其参数重新生成时会报错。只能将原本生成好的队列删除，重新创建队列。RabbitMQ 不允许重新定义一个已经存在的队列。

## 8.发布/订阅模式

### 模型



解读：

1. 一个生成者，多个消费者
2. 每一个消费者都有自己的队列
3. 生产者没有直接把消息发送到队列，而是发送到了交换机中（exchange）
4. 每个队列都需要绑定到交换机上
5. 生产者发送消息经过交换机到达不同的队列，一个消息被多个不同的队列的消费者消费。

应该场景：

用户注册 🡪 发送邮件通知 🡪 发送短信通知

### 2.创建消息-生产者

|  |
| --- |
| **public class** Producer {  **private static final** String ***EXCHANGE\_NAME*** = "EXCHANGE\_PUBLISH\_SUBSCRIBE\_FANOUT";   **public static void** main(String[] args) **throws** Exception {  //创建连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明交换空间  channel.exchangeDeclare(***EXCHANGE\_NAME***, BuiltinExchangeType.***FANOUT***);  //发送消息  String msg = "publish/subscribe";  System.***out***.println("Producer send => " + msg);  channel.basicPublish(***EXCHANGE\_NAME***,"", **null**, msg.getBytes());  channel.close();  connection.close();  } } |

### 3.创建消息-消费者

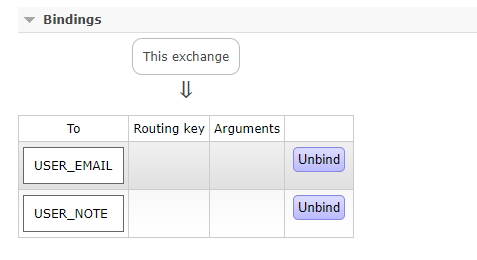
**消费者 Email**

|  |
| --- |
| **public class** ConsumerEmail {   **private static final** String ***QUEUR\_NAME*** = "USER\_EMAIL";  **private static final** String ***EXCHANGE\_NAME*** = "EXCHANGE\_PUBLISH\_SUBSCRIBE\_FANOUT";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建通信管道  Channel channel = connection.createChannel();  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUR\_NAME***, **false**, **false**, **false**, **null**);  //绑定到交换空间  channel.queueBind(***QUEUR\_NAME***, ***EXCHANGE\_NAME***, "");  //监听消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[Email Cosumer] ===> " + **new** String(body, "UTF-8"));  //手动应答  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  };  channel.basicConsume(***QUEUR\_NAME***,**false**,consumer);  } } |

**消费者note**

|  |
| --- |
| **public class** ConsumerNote {   **private static final** String ***QUEUR\_NAME*** = "USER\_NOTE";  **private static final** String ***EXCHANGE\_NAME*** = "EXCHANGE\_PUBLISH\_SUBSCRIBE\_FANOUT";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建通信管道  Channel channel = connection.createChannel();  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUR\_NAME***, **false**, **false**, **false**, **null**);  //绑定到交换空间  channel.queueBind(***QUEUR\_NAME***, ***EXCHANGE\_NAME***, "");  //监听消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[Note Cosumer] ===> " + **new** String(body, "UTF-8"));  //手动应答  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  };  channel.basicConsume(***QUEUR\_NAME***,**false**,consumer);  } } |

如果两个不同的队列绑定到了交换空间



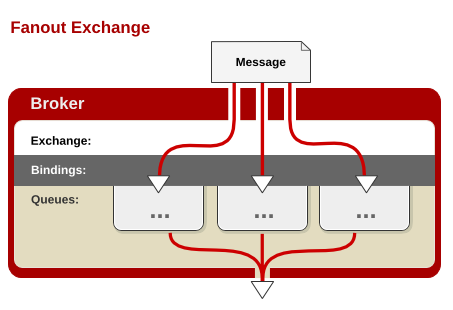
### 细节问题

1. 在RabbitMQ中只有队列有存储能力。
2. 队列绑定交换空间时，需要交换空间存在才能绑定成功，不然会报错。

## 9.Exchange(交换空间)

RabbitMQ提供了四种Exchange模式：fanout,direct,topic,header 。 header模式在实际使用中较少。

### 1. fanout Exchange

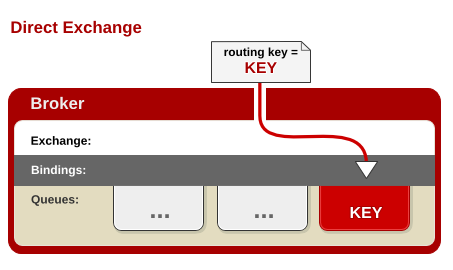


所有发送到Fanout Exchange的消息都会被转发到与该Exchange 绑定(Binding)的所有Queue上。

　　Fanout Exchange 不需要处理RouteKey 。只需要简单的将队列绑定到exchange 上。这样发送到exchange的消息都会被转发到与该交换机绑定的所有队列上。类似子网广播，每台子网内的主机都获得了一份复制的消息。

所以，Fanout Exchange 转发消息是最快的。

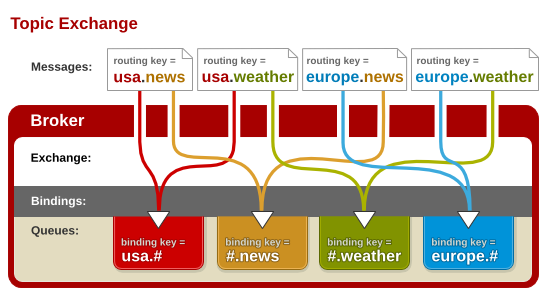
### 2. Direct Exchange



所有发送到Direct Exchange的消息被转发到RouteKey中指定的Queue。

Direct模式,可以使用rabbitMQ自带的Exchange：default Exchange 。所以不需要将Exchange进行任何绑定(binding)操作 。消息传递时，RouteKey必须完全匹配，才会被队列接收，否则该消息会被抛弃。

### 3.Topic Exchange



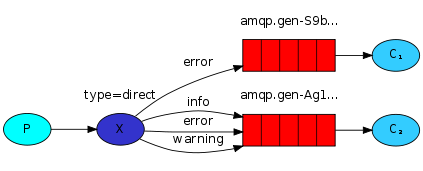
所有发送到Topic Exchange的消息被转发到所有关心RouteKey中指定Topic的Queue上，

　　Exchange 将RouteKey 和某Topic 进行模糊匹配。此时队列需要绑定一个Topic。可以使用通配符进行模糊匹配，符号“#”匹配一个或多个词，符号“\*”匹配不多不少一个词。因此“log.#”能够匹配到“log.info.oa”，但是“log.\*” 只会匹配到“log.error”。

所以，Topic Exchange 使用非常灵活

## 10. Routing（路由key）

### 1.模型



### 2.创建消息-生产者

|  |
| --- |
| **public class** Producer {  **private static final** String ***EXCHANGE\_NAME*** = "exchange\_routing\_direct";  **private static final** String ***ROUTING\_KEY*** = "error";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  //声明交换空间  channel.exchangeDeclare(***EXCHANGE\_NAME***, BuiltinExchangeType.***DIRECT***);  //发送消息  String msg = "routing key!";  System.***out***.println("Producer send msg ==> " + msg );  channel.basicPublish(***EXCHANGE\_NAME***, ***ROUTING\_KEY***, **null**, msg.getBytes());  //关闭连接  channel.close();  connection.close();  } } |

### 创建消息-消费者

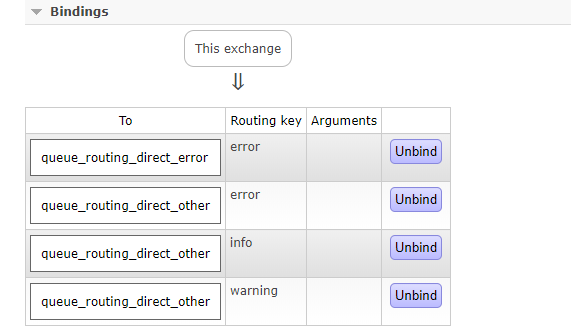
**消费key=error**

|  |
| --- |
| **public class** Consumer1 {  **private static final** String ***QUEUE\_NAME*** = "queue\_routing\_direct\_error";  **private static final** String ***EXCHANGE\_NAME*** = "exchange\_routing\_direct";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  //绑定交换空间  channel.queueBind(***QUEUE\_NAME***,***EXCHANGE\_NAME***,"error");  //接受消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[consumer 1 ]接受到消息 ===> " + **new** String(body, "UTF-8"));  //手动确认消息  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  };  channel.basicConsume(***QUEUE\_NAME***,consumer);  } } |

**消费者 key=info,warning,error**

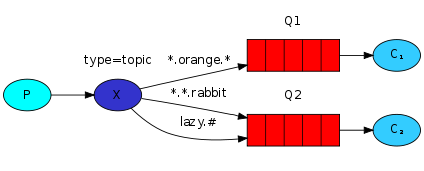
|  |
| --- |
| **public class** Consumer2 {  **private static final** String ***QUEUE\_NAME*** = "queue\_routing\_direct\_other";  **private static final** String ***EXCHANGE\_NAME*** = "exchange\_routing\_direct";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  //绑定交换空间  channel.queueBind(***QUEUE\_NAME***,***EXCHANGE\_NAME***,"info");  channel.queueBind(***QUEUE\_NAME***,***EXCHANGE\_NAME***,"warning");  channel.queueBind(***QUEUE\_NAME***,***EXCHANGE\_NAME***,"error");  //接受消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[consumer 2 ]接受到消息 ===> " + **new** String(body, StandardCharsets.***UTF\_8***));  //手动确认消息  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  };  channel.basicConsume(***QUEUE\_NAME***,consumer);  } } |

### 4.队列绑定图



## 11. [Topics](http://www.rabbitmq.com/tutorials/tutorial-five-python.html)

### 1.模型



**#：**匹配一个或多个

\*：匹配一个

### 2.创建消息-生产者

|  |
| --- |
| **public class** Producer {  **private static final** String ***EXCHANGE\_NAME*** = "exchange\_routing\_topics";  **private static final** String ***ROUTING\_KEY*** = "goods.add";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  //声明交换空间  channel.exchangeDeclare(***EXCHANGE\_NAME***, BuiltinExchangeType.***TOPIC***);  //发送消息  String msg = "topics !";  System.***out***.println("Producer send msg ==> " + msg);  channel.basicPublish(***EXCHANGE\_NAME***, ***ROUTING\_KEY***, **null**, msg.getBytes());  //关闭连接  channel.close();  connection.close();  } } |

### 3.创建消息-消费者

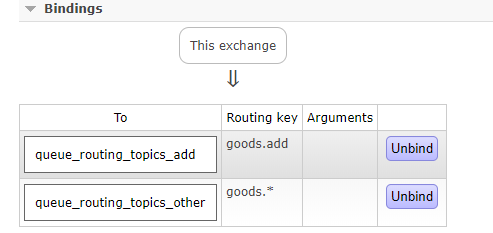
**消费者1**

|  |
| --- |
| **public class** Consumer1 {  **private static final** String ***QUEUE\_NAME*** = "queue\_routing\_topics\_add";  **private static final** String ***EXCHANGE\_NAME*** = "exchange\_routing\_topics";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  //绑定交换空间  channel.queueBind(***QUEUE\_NAME***,***EXCHANGE\_NAME***,"goods.add");  //接受消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[consumer 1 ]接受到消息 ===> " + **new** String(body, "UTF-8"));  //手动确认消息  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  };  channel.basicConsume(***QUEUE\_NAME***,**false**,consumer);  } } |

**消费者2**

|  |
| --- |
| **public class** Consumer2 {  **private static final** String ***QUEUE\_NAME*** = "queue\_routing\_topics\_other";  **private static final** String ***EXCHANGE\_NAME*** = "exchange\_routing\_topics";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  channel.basicQos(1);  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  //绑定交换空间  channel.queueBind(***QUEUE\_NAME***,***EXCHANGE\_NAME***,"goods.\*");   //接受消息  DefaultConsumer consumer = **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[consumer 2 ]接受到消息 ===> " + **new** String(body, StandardCharsets.***UTF\_8***));  //手动确认消息  channel.basicAck(envelope.getDeliveryTag(), **false**);  }  };  channel.basicConsume(***QUEUE\_NAME***,**false**,consumer);  } } |

### 队列绑定图



## 12.RabbitMQ 消息确认机制

在RabbitMq中我们可以通过之持久化数据来解决RabbitMQ服务器异常的数据丢失问题。消息确认机制是保证生产者将消息发送出去之后，消息到底有没有到达RabbitMQ服务器。

### AMPQ事物机制

事物的三个方法

txSelect:用户将当前channel设置成transation 模式。

TxCommit:用于提交事物

TxRollback:回滚事物

#### 1.1消息生产者

|  |
| --- |
| **public class** ProducerTx {  **private static final** String ***QUEUE\_NAME*** = "transaction\_queue";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***,**false**,**false**,**false**,**null**);  **try** {  //开启事物  channel.txSelect();  //int i = 1/0;  //发送消息  String msg = "transation !";  System.***out***.println("Producer send msg ==> " + msg);  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  //提交事物  channel.txCommit();  } **catch** (Exception e) {  e.printStackTrace();  channel.txRollback();  }  //关闭连接  channel.close();  connection.close();  } } |

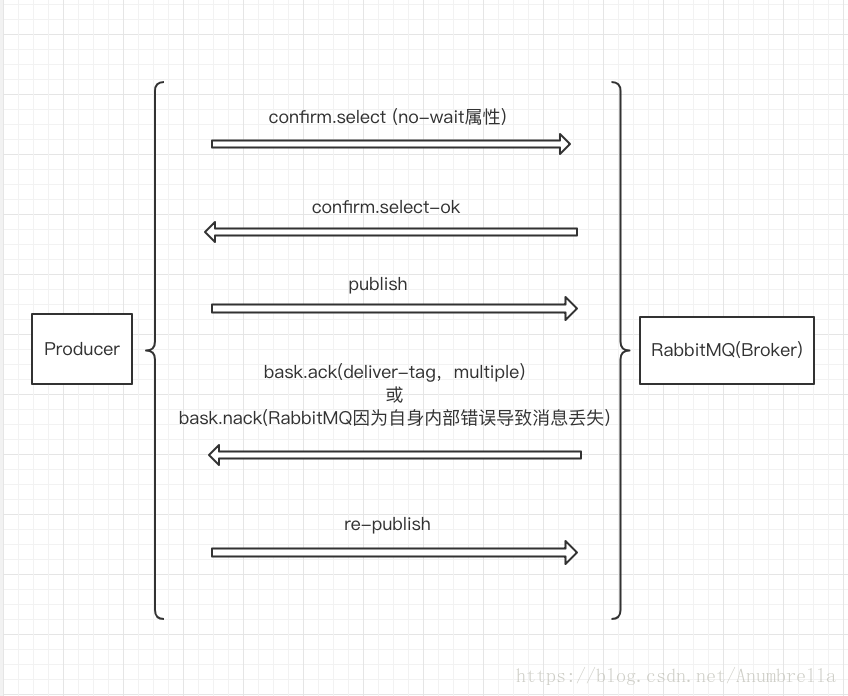
#### 1.2消息消费者

|  |
| --- |
| **public class** ConsumerTx {  **private static final** String ***QUEUE\_NAME*** = "transaction\_queue";   **public static void** main(String[] args) **throws** Exception {  //获取连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //获取通信管道  Channel channel = Objects.*requireNonNull*(connection).createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //接受消息  channel.basicConsume(***QUEUE\_NAME***, **true**, **new** DefaultConsumer(channel) {  @Override  **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body) **throws** IOException {  System.***out***.println("[consumer] 接受到消息 ==> " + **new** String(body, StandardCharsets.***UTF\_8***));  }  });  } } |

这种方式的缺点也很明显，需要多次和服务器进行通信。这样大大降低了RabbitMQ的吞吐量。

### 2.confirm 模式

#### 2.1 producer端confirm模式的实现原理



为了使用Confirm模式，client会发送confirm.select方法帧。通过是否设置了no-wait属性，来决定Broker端是否会以confirm.select-ok来进行应答。一旦在channel上使用confirm.select方法，channel就将处于Confirm模式。处于 transactional模式的channel不能再被设置成Confirm模式，反之亦然。

在生产者将信道设置成Confirm模式，一旦信道进入Confirm模式，所有在该信道上面发布的消息都会被指派一个唯一的ID(以confirm.select为基础从1开始计数)，一旦消息被投递到所有匹配的队列之后，Broker就会发送一个确认给生产者（包含消息的唯一ID）,这就使得生产者知道消息已经正确到达目的队列了，如果消息和队列是可持久化的，那么确认消息会将消息写入磁盘之后发出，Broker回传给生产者的确认消息中deliver-tag域包含了确认消息的序列号，此外Broker也可以设置basic.ack的multiple域，表示到这个序列号之前的所有消息都已经得到了处理。

Confirm模式最大的好处在于它是异步的，一旦发布一条消息，生产者应用程序就可以在等信道返回确认的同时继续发送下一条消息，当消息最终得到确认之后，生产者应用便可以通过回调方法来处理该确认消息，如果RabbitMQ因为自身内部错误导致消息丢失，就会发送一条basic.nack来代替basic.ack的消息，在这个情形下，basic.nack中各域值的含义与basic.ack中相应各域含义是相同的，同时requeue域的值应该被忽略。通过nack一条或多条消息， Broker表明自身无法对相应消息完成处理，并拒绝为这些消息的处理负责。在这种情况下，client可以选择将消息re-publish。

在channel 被设置成Confirm模式之后，所有被publish的后续消息都将被Confirm（即 ack）或者被nack一次。但是没有对消息被Confirm的快慢做任何保证，并且同一条消息不会既被Confirm又被nack。

#### 2.2编程模式

客户端实现生产者confirm有三种编程方式：

1.普通confirm模式：每发送一条消息后，调用waitForConfirms()方法，等待服务器端confirm。实际上是一种串行confirm了。

2.批量confirm模式：每发送一批消息后，调用waitForConfirms()方法，等待服务器端confirm。

3.异步confirm模式：提供一个回调方法，服务端confirm了一条或者多条消息后Client端会回调这个方法。

#### 2.3第一种普通Confirm模式

普通Confirm模式最简单，publish一条消息后，等待服务器端Confirm，如果服务端返回false或者超时时间内未返回，客户端就进行消息重传。

|  |
| --- |
| **public class** ProducerSingle {  **private static final** String ***QUEUE\_NAME*** = "queue\_confirm";   **public static void** main(String[] args) **throws** Exception {  //创建连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //开始confirm 模式  channel.confirmSelect();  //发送消息  String msg = "confirm !";  System.***out***.println("producer single send msg " + msg);  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  //消息确认  **if** (channel.waitForConfirms()) {  System.***out***.println("消息发送成功！");  } **else** {  System.***out***.println("消息发送失败！");  }  //关闭连接  channel.close();  connection.close();  } } |

#### 2.4第二种批量Confirm模式

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception {  //创建连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //开始confirm 模式  channel.confirmSelect();  //发送消息  **for** (**int** i = 0; i < 10; i++) {  String msg = "confirm !" + i;  System.***out***.println("producer single send msg " + msg);  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  }  //消息确认  **if** (channel.waitForConfirms()) {  System.***out***.println("消息发送成功！");  } **else** {  System.***out***.println("消息发送失败！");  }  //关闭连接  channel.close();  connection.close(); } |

#### 2.4第三中异步Confirm模式

|  |
| --- |
| **public class** ProducerAsynchronization {  **private static final** String ***QUEUE\_NAME*** = "queue\_confirm";   **public static void** main(String[] args) **throws** Exception {  //创建连接  Connection connection = RabbitMQConnectUtil.*getConnection*();  //创建管道  Channel channel = connection.createChannel();  //声明队列  channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);  //存储未确认的信息主键  SortedSet<Long> confirmSet = Collections.*synchronizedSortedSet*(**new** TreeSet<Long>());  //开始confirm 模式  channel.confirmSelect();  channel.addConfirmListener(**new** ConfirmListener() {  **public void** handleAck(**long** deliveryTag, **boolean** multiple) **throws** IOException {  **if** (multiple) {  System.***out***.println("多条一起确认");  confirmSet.headSet(deliveryTag + 1L).clear();  } **else** {  System.***out***.println("一条一起确认");  confirmSet.remove(deliveryTag);  }  }   **public void** handleNack(**long** deliveryTag, **boolean** multiple) **throws** IOException {  System.***out***.println("Nack, SeqNo: " + deliveryTag + ", multiple: " + multiple);  **if** (multiple) {  confirmSet.headSet(deliveryTag + 1L).clear();  } **else** {  confirmSet.remove(deliveryTag);  }  }  });  //发送消息  **for** (**int** i = 0; i < 10; i++) {  **long** nextSeqNo = channel.getNextPublishSeqNo();  String msg = "confirm !" + i;  System.***out***.println("producer single send msg " + msg);  channel.basicPublish("", ***QUEUE\_NAME***, **null**, msg.getBytes());  confirmSet.add(nextSeqNo);  }  //关闭连接  channel.close();  connection.close();  } } |

## 13.参数说明（补充）

参数说明详情：<http://www.cnblogs.com/piaolingzxh/p/5448927.html>

Arguments详情：

<https://blog.csdn.net/farmwang/article/details/80750554>

/3.声明队列。-将队列参数传到队列 （队列名字，是否持久化，是否排外，是否自动清理，参数）

channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, **null**);

durable：是否持久化，那么问题来了，这是什么意思？持久化，指的是队列持久化到数据库中。在之前的博文中也说过，如果RabbitMQ服务挂了怎么办，队列丢失了自然是不希望发生的。持久化设置为true的话，即使服务崩溃也不会丢失队列

exclusive：是否排外，what？ 这又是什么呢。设置了排外为true的队列只可以在本次的连接中被访问，也就是说在当前连接创建多少个channel访问都没有关系，但是如果是一个新的连接来访问，对不起，不可以，下面是我尝试访问了一个排外的queue报的错。还有一个需要说一下的是，排外的queue在当前连接被断开的时候会自动消失（清除）无论是否设置了持久化

autoDelete：这个就很简单了，是否自动删除。也就是说queue会清理自己。但是是在最后一个connection断开的时候

arguments：这个值得拿出来单讲一次，暂时不说

不同方式的生明queue

在queuedeclare的时候会发现还有很多这样的方法，下面图中就可以看到，不要以为其它两个都是吃瓜群众，它们也是有着其特殊功效的

QueueDeclareNoWait：相当于一个async版的声明队列，可以看到是没有返回的。调完方法就结束。也不等队列创建结果

QueueDeclarePassive：消极的声明创建？这是什么鬼，事实上它没有去声明队列，所谓消极，去看看有没有名为xxx的queue，如果有我就把名字什么的信息告诉你，没有就直接报错也不管。那么问题来了，这个方法如此鸡肋，我要它有何用？ 。。。其实你可以用来确认queue是否存在嘛