**Rabbitmq学习文档**

# 相关网站

官网：<https://www.rabbitmq.com>

学习例子：<https://blog.csdn.net/wqc19920906/article/category/7984962>

MQ详解及四大MQ比较：

本地账号： lys 123456

交换机：<https://www.cnblogs.com/LipeiNet/p/5978276.html>

# MQ详解及四大MQ比较

<https://blog.csdn.net/wqc19920906/article/details/82193316>

# MQ使用场景

<https://blog.csdn.net/wqc19920906/article/details/82193593>

# 介绍

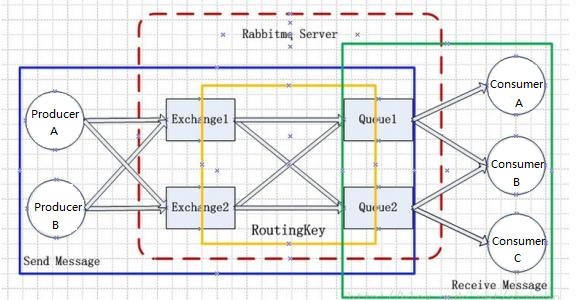
https://blog.csdn.net/wqc19920906/article/details/82193942

1、什么是RabbitMQ？

RabbitMQ是由Erlang语言编写的实现了高级消息队列协议（AMQP）的开源消息代理软件（也可称为 面向消息的中间件）。支持Windows、Linux/Unix、MAC OS X操作系统和包括JAVA在内的多种编程语言。

AMQP，即Advanced Message Queuing Protocol，一个提供统一消息服务的应用层标准高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。基于此协议的客户端与消息中间件可传递消息，并不受 客户端/中间件 不同产品，不同的开发语言等条件的限制。

2、RabbitMQ中的重要概念



（1）Broker：经纪人。提供一种传输服务，维护一条从生产者到消费者的传输线路，保证消息数据能按照指定的方式传输。粗略的可以将图中的RabbitMQ Server当作Broker。

（2）Exchange：消息交换机。指定消息按照什么规则路由到哪个队列Queue。

（3）Queue：消息队列。消息的载体，每条消息都会被投送到一个或多个队列中。

（4）Binding：绑定。作用就是将Exchange和Queue按照某种路由规则绑定起来。

（5）RoutingKey：路由关键字。Exchange根据RoutingKey进行消息投递。

（6）Vhost：虚拟主机。一个Broker可以有多个虚拟主机，用作不同用户的权限分离。一个虚拟主机持有一组Exchange、Queue和Binding。

（7）Producer：消息生产者。主要将消息投递到对应的Exchange上面。一般是独立的程序。

（8）Consumer：消息消费者。消息的接收者，一般是独立的程序。

（9）Channel：消息通道，也称信道。在客户端的每个连接里可以建立多个Channel，每个Channel代表一个会话任务。

3、RabbitMQ的使用流程

AMQP模型中，消息在producer中产生，发送到MQ的exchange上，exchange根据配置的路由方式投递到相应的Queue上，Queue又将消息发送给已经在此Queue上注册的consumer，消息从queue到consumer有push和pull两种方式。

消息队列的使用过程大概如下：

（1）客户端连接到消息队列服务器，打开一个channel。

（2）客户端声明一个exchange，并设置相关属性。

（3）客户端声明一个queue，并设置相关属性。

（4）客户端使用routing key，在exchange和queue之间建立好Binding关系。

（5）生产者客户端投递消息到exchange。

（6）exchange接收到消息后，就根据消息的RoutingKey和已经设置的binding，进行消息路由（投递），将消息投递到一个或多个队列里。

（7）消费者客户端从对应的队列中获取并处理消息。

工作过程：

生产者客户端：

1.客户端连接到RabbitMQ服务器上，打开一个消息通道（channel）；

2.客户端声明一个消息交换机（exchange），并设置相关属性。

3.客户端声明一个消息队列（queue），并设置相关属性。

4.客户端使用routing key在消息交换机（exchange）和消息队列（queue）中建立好绑定关系。

5.客户端投递消息都消息交换机（exchange）上

6.客户端关闭消息通道（channel）以及和服务器的连接。

服务器端：

exchange接收到消息后，根据消息的key（这个key的产生规则暂时没研究，有知道的小伙伴可以留言告诉我）和以及设置的binding，进行消息路由，将消息投递到一个或多个消息队列中。

关于exchange也有几个类型：

(1). Direct交换机：完全根据key进行投递。例如，绑定时设置了routing key为abc，客户端提交信息提交信息时只有设置了key为abc的才会投递到队列；

(2).Topic交换机：在key进行模式匹配后进行投递。例如：符号”#”匹配一个或多个字符，符号”\*”匹配一串连续的字母字符，例如”abc.#”可以匹配”abc.def.ghi”，而”abc.\*”只可以匹配”abc.def”。

(3).Fanout交换机：它采取广播模式，消息进来时，将会被投递到与改交换机绑定的所有队列中。

消费者客户端：待续...

4、RabbitMQ的消息持久化

RabbitMQ支持数据持久化，也就是把数据写在磁盘上，可以增加数据的安全性。消息队列持久化包括三个部分：

1.消息交换机（exchange）持久化，在声明时指定durable为1

2.消息队列（queue）持久化，在声明时指定durable为1

3.消息持久化，在投递时指定delivery\_mode为2（1是非持久化）

如果消息交换机（exchange）和消息队列（queue）都是持久化的话，那么他们之间的绑定（Binding）也是持久化的。如果消息交换机和消息队列之间一个持久化、一个非持久化，那么就不允许绑定。

5、RabbitMQ的优缺点

适用场景：

比较适合异步传输，这里解释一下什么是异步和同步。

异步：发送方不关心消息有没有发送成功，只发送消息，不去获取消息是否发送成功。

同步：发送方关心消息是否发送成功，发送消息后，会等待接收方返回状态码，根据状态码来判断是否发送成功，然后执行相对于的动作。

下边以Http中的同步和异步为例：

如：普通的B/S架构客户端和服务器端之间的通信就是同步的，即提交请求 ---> 等待服务器处理完毕返回消息 ---> 拿到服务器返回的消息，处理完毕。

如：Ajax技术就是异步的，请求通过事件触发 ---> 服务器处理（浏览器不用等待，仍可以做其他的事情） ---> 处理完毕。

有人可能会好奇说应用场景怎么说到了同步和异步，那说明你还不是很理解技术和应用场景之间的紧密联系。

优点：

（1）由Erlang语言开发，支持大量协议：AMQP、XMPP、SMTP、STOMP。

（2）支持消息的持久化、负载均衡和集群，且集群易扩展。

（3）具有一个Web监控界面，易于管理。

（4）安装部署简单，上手容易，功能丰富，强大的社区支持。

（5）支持消息确认机制、灵活的消息分发机制。

缺点：

（1）由于牺牲了部分性能来换取稳定性，比如消息的持久化功能，使得RabbitMQ在大吞吐量性能方面不及Kafka和ZeroMQ。

（2）由于支持多种协议，使RabbitMQ非常重量级，比较适合企业级开发。

因此当需要一个稳定的、高可靠性的、功能强大且易于管理的消息队列可以选择RabbitMQ。如果对消息吞吐量需求较大，且不在乎消息偶尔丢失的情况可以使用Kafka。

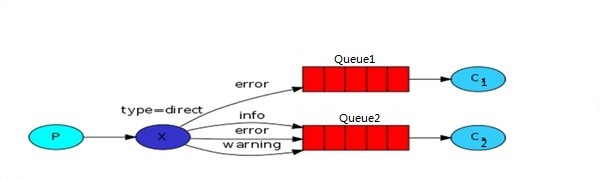
6、Exchange类型

6.1、Direct Exchange

（1）名称：直接交换器类型

（2）默认的预先定义exchange名字：空字符串或者amq.direct

（3）作用描述：根据Binding指定的Routing Key，将符合Key的消息发送到Binding的Queue。可以构建点对点消息传输模型。



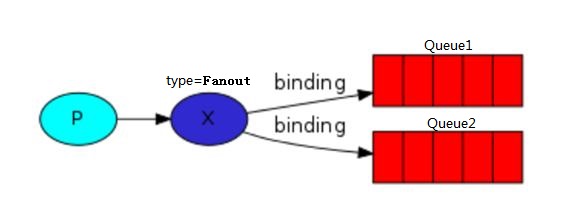
如图中RoutingKey分别是error、info、warning，其中error被Binding（绑定）到queue1和queue2上，info和warning被Binding到queue2上。当消息的RoutingKey是error，这条消息将被投递到queue1和queue2中（相当于消息被复制成两个分别投放到两个queue中），然后分别被Consumer1和Consumer2处理。如果消息的RoutingKey是info或者warning，这条消息只会被投递到queue2中，然后被Consumer2处理。如果消息的RoutingKey是其他的字符串，这条消息则会被丢弃。

6.2、Fanout Exchange

（1）名称：广播式交换器类型

（2）默认的预先定义exchange名字：amq.fanout

（3）作用描述：将同一个message发送到所有同该Exchange 绑定的queue。不论RoutingKey是什么，这条消息都会被投递到所有与此Exchange绑定的queue中。



广播式交换器类型的工作方式：不使用任何参数将queue和Exchange进行Binding，发布者publisher向Exchange发送一条消息（注意：直接交换器类型中的producer变成了publisher，其中隐含了两种交换器的区别），然后这条消息被无条件的投递给所有和这个Exchange绑定的queue中。

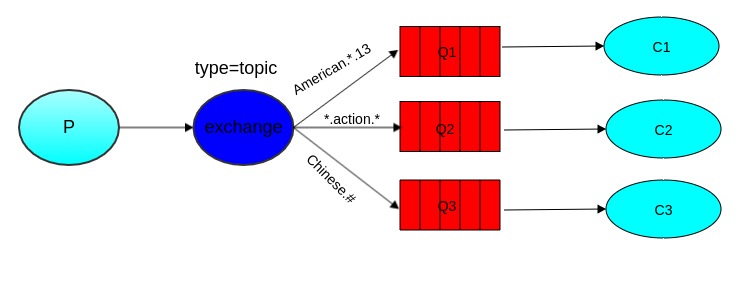
如图中，没有RoutingKey的限制，只要消息到达Exchange，都会被投递到queue1和queue2中，然后被对应的Consumer处理。

6.3、Topic Exchange

（1）名称：主题交换器类型

（2）默认的预先定义exchange名字：amq.topic

（3）作用描述：根据Binding指定的RoutingKey，Exchange对key进行模式匹配后投递到相应的Queue，模式匹配时符号“#”匹配一个或多个词，符号“\*”匹配正好一个词，而且单词与单词之间必须要用“.”符号进行分隔。此模式可以用来支持经典的发布/订阅消息传输模型-使用主题名字空间作为消息寻址模式，将消息传递给那些部分或者全部匹配主题模式的queue。

    如图中，假如消息的RoutingKey是American.action.13，这条消息将被投递到Q1和Q2中。假如RoutingKey是American.action.13.test（注意：此处是四个词），这条消息将会被丢弃，因为没有routingkey与之匹配。假如RoutingKey是Chinese.action.13，这条消息将被投递到Q2和Q3中。假如RoutingKey是Chinese.action.13.test，这条消息只会被投递到Q3中，#可以匹配一个或者多个单词，而\*只能匹配一个词。

6.4、Headers Exchange

（1）名称：标题交换器类型

（2）默认的预先定义exchange名字：amq.match和amq.headers

（3）作用描述：同direct exchange类似，不同之处是不再使用Routing Key路由，而是使用headers（Message attributes）进行匹配路由到指定Queue。

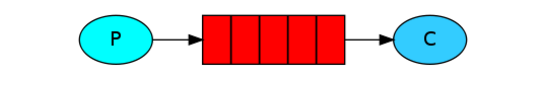
Headers类型的exchange使用的比较少，它也是忽略routingKey的一种路由方式。是使用Headers来匹配的。Headers是一个键值对，可以定义成HashTable。发送者在发送的时候定义一些键值对，接收者也可以再绑定时候传入一些键值对，两者匹配的话，则对应的队列就可以收到消息。匹配有两种方式all和any。这两种方式是在接收端必须要用键值"x-mactch"来定义。all代表定义的多个键值对都要满足，而any则代码只要满足一个就可以了。fanout，direct，topic exchange的routingKey都需要要字符串形式的，而headers exchange则没有这个要求，因为键值对的值可以是任何类型。

# [java简单的实现RabbitMQ](https://www.cnblogs.com/LipeiNet/p/5977028.html)

https://www.cnblogs.com/LipeiNet/p/5977028.html

## ****"Hello RabbitMQ"****

下面有一幅图，其中P表示生产者，C表示消费者，红色部分为消息队列



## ****项目开始****

1：首先引入rabbitMQ jar包

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>3.6.5</version>

</dependency>

2：创建消费者Producer

/\*\*

\* 消息生成者

\*/

public class Producer {

public final static String QUEUE\_NAME="rabbitMQ.test";

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

//创建连接工厂

ConnectionFactory factory = new ConnectionFactory();

//设置RabbitMQ相关信息

factory.setHost("localhost");

//factory.setUsername("lp");

//factory.setPassword("");

// factory.setPort(2088);

//创建一个新的连接

Connection connection = factory.newConnection();

//创建一个通道

Channel channel = connection.createChannel();

// 声明一个队列 channel.queueDeclare(QUEUE\_NAME, false, false, false, null);

String message = "Hello RabbitMQ";

//发送消息到队列中

channel.basicPublish("", QUEUE\_NAME, null, message.getBytes("UTF-8"));

System.out.println("Producer Send +'" + message + "'");

//关闭通道和连接

channel.close();

connection.close();

}

}

注1：queueDeclare第一个参数表示队列名称、第二个参数为是否持久化（true表示是，队列将在服务器重启时生存）、第三个参数为是否是独占队列（创建者可以使用的私有队列，断开后自动删除）、第四个参数为当所有消费者客户端连接断开时是否自动删除队列、第五个参数为队列的其他参数

注2：basicPublish第一个参数为交换机名称、第二个参数为队列映射的路由key、第三个参数为消息的其他属性、第四个参数为发送信息的主体

2.3：创建消费者

public class Customer {

private final static String QUEUE\_NAME = "rabbitMQ.test";

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

// 创建连接工厂

ConnectionFactory factory = new ConnectionFactory();

//设置RabbitMQ地址

factory.setHost("localhost");

//创建一个新的连接

Connection connection = factory.newConnection();

//创建一个通道

Channel channel = connection.createChannel();

//声明要关注的队列

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

System.out.println("Customer Waiting Received messages");

//DefaultConsumer类实现了Consumer接口，通过传入一个频道，

// 告诉服务器我们需要那个频道的消息，如果频道中有消息，就会执行回调函数handleDelivery

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope,

AMQP.BasicProperties properties, byte[] body)

throws IOException {

String message = new String(body, "UTF-8");

System.out.println("Customer Received '" + message + "'");

}

};

//自动回复队列应答 -- RabbitMQ中的消息确认机制

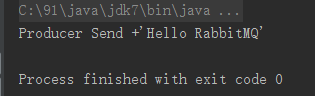
channel.basicConsume(QUEUE\_NAME, true, consumer);

}

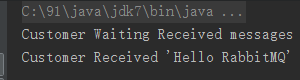
前面代码我们可以看出和生成者一样的，后面的是获取生产者发送的信息，其中envelope主要存放生产者相关信息（比如交换机、路由key等）body是消息实体。

2.4：运行结果

生产者：

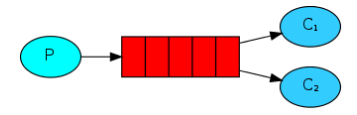


消费者：



## ****三：实现任务分发****

工作队列



一个队列的优点就是很容易处理并行化的工作能力，但是如果我们积累了大量的工作，我们就需要更多的工作者来处理，这里就要采用分布机制了。

我们新创建一个生产者NewTask

public class NewTask {

private static final String TASK\_QUEUE\_NAME="task\_queue";

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

ConnectionFactory factory=new ConnectionFactory();

factory.setHost("localhost");

Connection connection=factory.newConnection();

Channel channel=connection.createChannel();

channel.queueDeclare(TASK\_QUEUE\_NAME,true,false,false,null);

//分发信息

for (int i=0;i<10;i++){

String message="Hello RabbitMQ"+i;

channel.basicPublish("",TASK\_QUEUE\_NAME,

MessageProperties.PERSISTENT\_TEXT\_PLAIN,message.getBytes());

System.out.println("NewTask send '"+message+"'");

}

channel.close();

connection.close();

}

}

然后创建2个工作者Work1和Work2代码一样

public class Work1 {

private static final String TASK\_QUEUE\_NAME = "task\_queue";

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

final ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

final Channel channel = connection.createChannel();

channel.queueDeclare(TASK\_QUEUE\_NAME, true, false, false, null);

System.out.println("Worker1 Waiting for messages");

//每次从队列获取的数量

channel.basicQos(1);

final Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag,

Envelope envelope,

AMQP.BasicProperties properties,

byte[] body) throws IOException {

String message = new String(body, "UTF-8");

System.out.println("Worker1 Received '" + message + "'");

try {

throw new Exception();

//doWork(message);

}catch (Exception e){

channel.abort();

}finally {

System.out.println("Worker1 Done");

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

}

}

};

boolean autoAck=false;

//消息消费完成确认

channel.basicConsume(TASK\_QUEUE\_NAME, autoAck, consumer);

}

private static void doWork(String task) {

try {

Thread.sleep(1000); // 暂停1秒钟

} catch (InterruptedException \_ignored) {

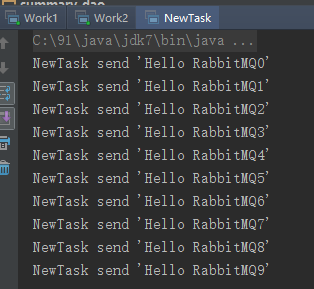
Thread.currentThread().interrupt();

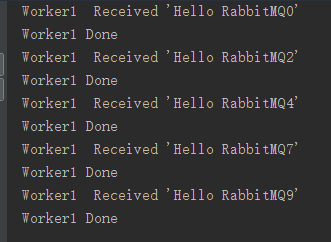
}

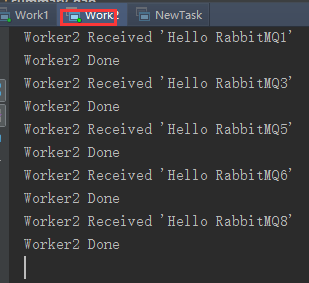
}

}

注：channel.basicQos(1);保证一次只分发一个 。autoAck是否自动回复，如果为true的话，每次生产者只要发送信息就会从内存中删除，那么如果消费者程序异常退出，那么就无法获取数据，我们当然是不希望出现这样的情况，所以才去手动回复，每当消费者收到并处理信息然后在通知生成者。最后从队列中删除这条信息。如果消费者异常退出，如果还有其他消费者，那么就会把队列中的消息发送给其他消费者，如果没有，等消费者启动时候再次发送。

****

****

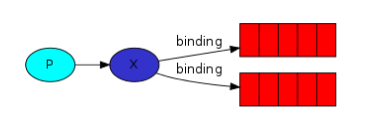


# [采用不同的交换机规则](https://www.cnblogs.com/LipeiNet/p/5978276.html)

在上一篇我们都是采用发送信息到队列然后队列把信息在发送到消费者，其实实际情况并非如此，rabbitMQ其实真正的思想是生产者不发送任何信息到队列，甚至不知道信息将发送到哪个队列。相反生产者只能发送信息到交换机，交换机接收到生产者的信息，然后按照规则把它推送到对列中，交换机是如何做处理他接收到的信息，并怎么样发送到特定的队列，那么这一篇主要是讲解交换机的规则。

## 发布/订阅-广播fanout

在上一篇说到的队列都指定了名称，但是现在我们不需要这么做，我们需要所有的日志信息，而不只是其中的一个。如果要做这样的队列，我们需要2件事，一个就是获取一个新的空的队列，这样我就需要创建一个随机名称的队列，最好让服务器帮我们做出选择，第一个就是我们断开用户的队列，应该自动进行删除。ok下面是一副工作图。



信息发送端代码

public class EmitLog {

private static final String EXCHANGE\_NAME = "logs";

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

ConnectionFactory factory=new ConnectionFactory();

factory.setHost("localhost");

Connection connection=factory.newConnection();

Channel channel=connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME,"fanout");//fanout表示分发，所有的消费者得到同样的队列信息

//分发信息

for (int i=0;i<5;i++){

String message="Hello World"+i;

channel.basicPublish(EXCHANGE\_NAME,"",null,message.getBytes());

System.out.println("EmitLog Sent '" + message + "'");

}

channel.close();

connection.close();

}

消费者代码

public class ReceiveLogs1 {

private static final String EXCHANGE\_NAME = "logs";

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE\_NAME, "fanout");

//产生一个随机的队列名称

String queueName = channel.queueDeclare().getQueue();

channel.queueBind(queueName, EXCHANGE\_NAME, "");//对队列进行绑定

System.out.println("ReceiveLogs1 Waiting for messages");

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {

String message = new String(body, "UTF-8");

System.out.println("ReceiveLogs1 Received '" + message + "'");

}

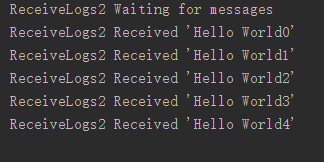
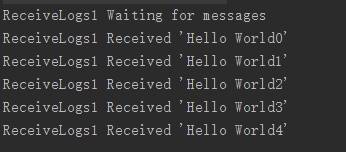
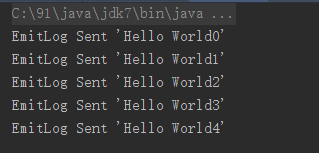
};

channel.basicConsume(queueName, true, consumer);//队列会自动删除

}

}

上面就完成了一个发布/订阅模式的消息队列 看看结果

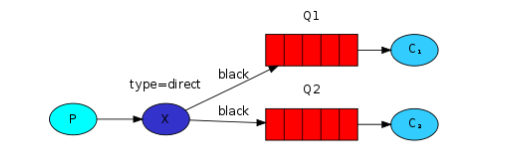


先启动接收客户端，开启接收通道（exchange=fanout）

再启动发布端，往对应的交换机发送信息

## [Routing](http://www.rabbitmq.com/tutorials/tutorial-four-python.html)-路由direct

上面我用采用了广播的模式进行消息的发送，现在我们采用路由的方式对不同的消息进行过滤



发送端代码

public class RoutingSendDirect {

private static final String EXCHANGE\_NAME = "direct\_logs";

// 路由关键字

private static final String[] routingKeys = new String[]{"info" ,"warning", "error"};

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

//声明交换机

channel.exchangeDeclare(EXCHANGE\_NAME,"direct");//注意是direct

//发送信息

for (String routingKey:routingKeys){

String message = "RoutingSendDirect Send the message level:" + routingKey;

channel.basicPublish(EXCHANGE\_NAME,routingKey,null,message.getBytes());

System.out.println("RoutingSendDirect Send"+routingKey +"':'" + message);

}

channel.close();

connection.close();

}

}

ReceiveLogsDirect1 消费者代码

public class ReceiveLogsDirect1 {

// 交换器名称

private static final String EXCHANGE\_NAME = "direct\_logs";

// 路由关键字

private static final String[] routingKeys = new String[]{"info" ,"warning"};

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

//声明交换器

channel.exchangeDeclare(EXCHANGE\_NAME, "direct");

//获取匿名队列名称

String queueName=channel.queueDeclare().getQueue();

//根据路由关键字进行绑定

for (String routingKey:routingKeys){

channel.queueBind(queueName,EXCHANGE\_NAME,routingKey);

System.out.println("ReceiveLogsDirect1 exchange:"+EXCHANGE\_NAME+"," +

" queue:"+queueName+", BindRoutingKey:" + routingKey);

}

System.out.println("ReceiveLogsDirect1 Waiting for messages");

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {

String message = new String(body, "UTF-8");

System.out.println("ReceiveLogsDirect1 Received '" + envelope.getRoutingKey() + "':'" + message + "'");

}

};

channel.basicConsume(queueName, true, consumer);

}

ReceiveLogsDirect2消费者代码

public class ReceiveLogsDirect2 {

// 交换器名称

private static final String EXCHANGE\_NAME = "direct\_logs";

// 路由关键字

private static final String[] routingKeys = new String[]{"error"};

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

//声明交换器

channel.exchangeDeclare(EXCHANGE\_NAME, "direct");

//获取匿名队列名称

String queueName = channel.queueDeclare().getQueue();

//根据路由关键字进行多重绑定

for (String severity : routingKeys) {

channel.queueBind(queueName, EXCHANGE\_NAME, severity);

System.out.println("ReceiveLogsDirect2 exchange:"+EXCHANGE\_NAME+", queue:"+queueName+", BindRoutingKey:" + severity);

}

System.out.println("ReceiveLogsDirect2 Waiting for messages");

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws UnsupportedEncodingException {

String message = new String(body, "UTF-8");

System.out.println("ReceiveLogsDirect2 Received '" + envelope.getRoutingKey() + "':'" + message + "'");

}

};

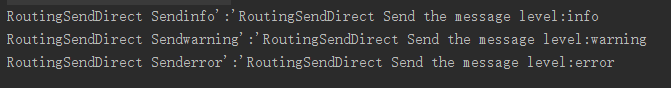
channel.basicConsume(queueName, true, consumer);

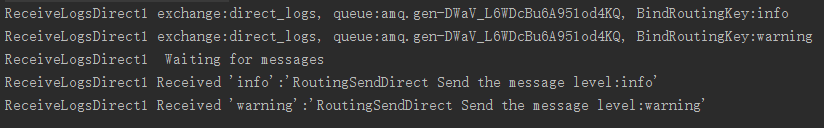
}

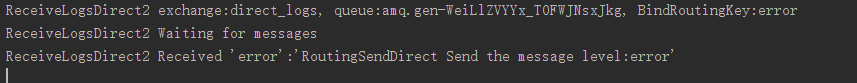
}

[复制代码](javascript:void(0);)

上面代码可以看出这里是通过路由来找个这个对列的。我们看下结果







## Topics-模糊匹配topic

这种应该属于模糊匹配

\* ：可以替代一个词

#：可以替代0或者更多的词

现在我们继续看看代码来理解

发送端

public class TopicSend {

private static final String EXCHANGE\_NAME = "topic\_logs";

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

Connection connection = null;

Channel channel = null;

try{

ConnectionFactory factory=new ConnectionFactory();

factory.setHost("localhost");

connection=factory.newConnection();

channel=connection.createChannel();

//声明一个匹配模式的交换机

channel.exchangeDeclare(EXCHANGE\_NAME,"topic");

//待发送的消息

String[] routingKeys=new String[]{

"quick.orange.rabbit",

"lazy.orange.elephant",

"quick.orange.fox",

"lazy.brown.fox",

"quick.brown.fox",

"quick.orange.male.rabbit",

"lazy.orange.male.rabbit"

};

//发送消息

for(String severity :routingKeys){

String message = "From "+severity+" routingKey' s message!";

channel.basicPublish(EXCHANGE\_NAME, severity, null, message.getBytes());

System.out.println("TopicSend Sent '" + severity + "':'" + message + "'");

}

}catch (Exception e){

e.printStackTrace();

if (connection!=null){

channel.close();

connection.close();

}

}finally {

if (connection!=null){

channel.close();

connection.close();

}

}

}

}

[复制代码](javascript:void(0);)

消费者1：

[复制代码](javascript:void(0);)

public class ReceiveLogsTopic1 {

private static final String EXCHANGE\_NAME = "topic\_logs";

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

//声明一个匹配模式的交换机

channel.exchangeDeclare(EXCHANGE\_NAME, "topic");

String queueName = channel.queueDeclare().getQueue();

//路由关键字

String[] routingKeys = new String[]{"\*.orange.\*"};

//绑定路由

for (String routingKey : routingKeys) {

channel.queueBind(queueName, EXCHANGE\_NAME, routingKey);

System.out.println("ReceiveLogsTopic1 exchange:" + EXCHANGE\_NAME + ", queue:" + queueName + ", BindRoutingKey:" + routingKey);

}

System.out.println("ReceiveLogsTopic1 Waiting for messages");

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws IOException {

String message = new String(body, "UTF-8");

System.out.println("ReceiveLogsTopic1 Received '" + envelope.getRoutingKey() + "':'" + message + "'");

}

};

channel.basicConsume(queueName, true, consumer);

}

}

[复制代码](javascript:void(0);)

消费者2：

[复制代码](javascript:void(0);)

ublic class ReceiveLogsTopic2 {

private static final String EXCHANGE\_NAME = "topic\_logs";

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

// 声明一个匹配模式的交换器

channel.exchangeDeclare(EXCHANGE\_NAME, "topic");

String queueName = channel.queueDeclare().getQueue();

// 路由关键字

String[] routingKeys = new String[]{"\*.\*.rabbit", "lazy.#"};

// 绑定路由关键字

for (String bindingKey : routingKeys) {

channel.queueBind(queueName, EXCHANGE\_NAME, bindingKey);

System.out.println("ReceiveLogsTopic2 exchange:"+EXCHANGE\_NAME+", queue:"+queueName+", BindRoutingKey:" + bindingKey);

}

System.out.println("ReceiveLogsTopic2 Waiting for messages");

Consumer consumer = new DefaultConsumer(channel) {

@Override

public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, byte[] body) throws UnsupportedEncodingException {

String message = new String(body, "UTF-8");

System.out.println("ReceiveLogsTopic2 Received '" + envelope.getRoutingKey() + "':'" + message + "'");

}

};

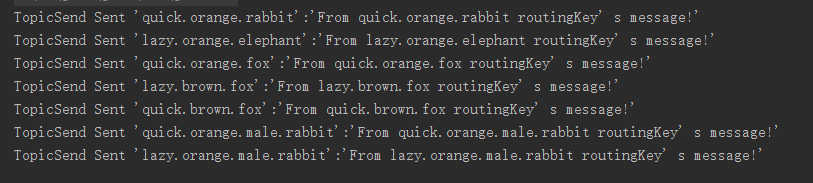
channel.basicConsume(queueName, true, consumer);

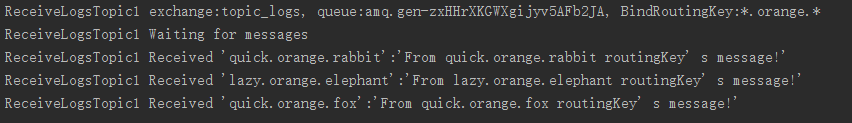
}

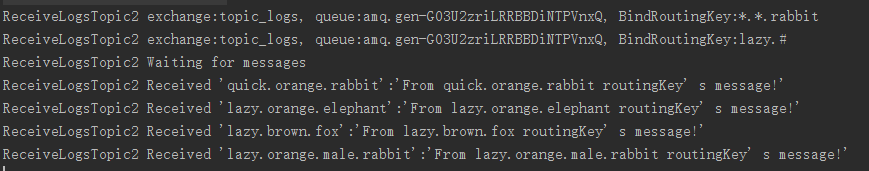
}

[复制代码](javascript:void(0);)

运行后结果







# [远程调用](https://www.cnblogs.com/LipeiNet/p/5980802.html)

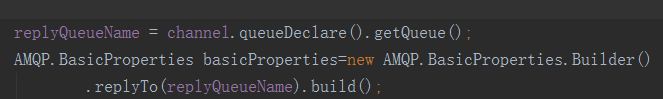
<https://www.cnblogs.com/LipeiNet/p/5980802.html>

 前言：前面我们讲解的都是本地服务器，现在如果需要远程计算机上运行一个函数，等待结果。这就是一个不同的故事了，这种模式通常被称为远程过程调用或者RPC。

本章教程我们使用RabbitMQ搭建一个RPC系统，一个客户端和一个可扩展的RPC服务器，现在我们开始吧。

## Callback queue

一般做rpc在RabbitMQ是比较容易的，一个客户端发送一个请求信息和一个响应信息的服务器回复，为了得到一个响应，我们需要发送一个回调队列地址请求。如下



Message属性：

AMQP协议一共预定义了14个属性，但是大多数属性很少使用，下面几个可能用的比较多

deliveryMode：有2个值，一个是持久，另一个表示短暂（第二篇说过）

contentType：内容类型：用来描述编码的MIME类型。例如，经常使用JSON编码是将此属性设置为一个很好的做法：application/json。

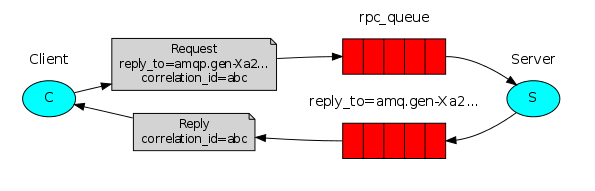
replyTo：经常使用的是回调队列的名字

correlationid：RPC响应请求的相关应用

## Correlation Id

在队列上接收到一个响应，但它并不清楚响应属于哪一个，当我们使用CorrelationId属性的时候，我们就可以将它设置为每个请求的唯一值，稍后当我们在回调队列中接收消息的时候，我们会看到这个属性，如果我们看到一个未知的CorrelationId，我们就可以安全地忽略信息-它不属于我们的请求。为什么我们应该忽略未知的消息在回调队列中，而不是失败的错误？这是由于服务器端的一个竞争条件的可能性。比如还未发送了一个确认信息给请求，但是此时RPC服务器挂了。如果这种情况发生，将再次重启RPC服务器处理请求。这就是为什么在客户端必须处理重复的反应。

## 需求



 我们的rpc工作方式如下：

1：当客户端启动时，它创建一个匿名的独占回调队列。

2：对于rpc请求，客户端发送2个属性，一个是replyTo设置回调队列，另一是correlationId为每个队列设置唯一值

3：请求被发送到一个rpc\_queue队列中

4：rpc服务器是等待队列的请求，当收到一个请求的时候，他就把消息返回的结果返回给客户端，使请求结束。

5：客户端等待回调队列上的数据，当消息出现的时候，他检查correlationId，如果它和从请求返回的值匹配，就进行响应。

## 编码

RPCServer.Java

public class RPCServer {

private static final String RPC\_QUEUE\_NAME = "rpc\_queue";

private static int fib(int n) {

if (n == 0) {

return 0;

}

if (n == 1) {

return 1;

}

return fib(n - 1) + fib(n - 1);

}

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

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

Connection connection = factory.newConnection();

Channel channel = connection.createChannel();

channel.queueDeclare(RPC\_QUEUE\_NAME, false, false, false, null);

channel.basicQos(1);

QueueingConsumer consumer = new QueueingConsumer(channel);

channel.basicConsume(RPC\_QUEUE\_NAME, false, consumer);

System.out.println("RPCServer Awating RPC request");

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

BasicProperties props = delivery.getProperties();

BasicProperties replyProps = new AMQP.BasicProperties.Builder().

correlationId(props.getCorrelationId()).build();

String message = new String(delivery.getBody(), "UTF-8");

int n = Integer.parseInt(message);

System.out.println("RPCServer fib(" + message + ")");

String response = "" + fib(n);

channel.basicPublish( "", props.getReplyTo(), replyProps, response.getBytes());

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

}

}

}

服务器代码比较简单

1：建立连接，通道，队列

2：我们可能运行多个服务器进程，为了分散负载服务器压力，我们设置channel.basicQos(1);

3：我们用basicconsume访问队列。然后进入循环，在其中我们等待请求消息并处理消息然后发送响应。

RPCClient.java

public class RPCClient {

private Connection connection;

private Channel channel;

private String requestQueueName = "rpc\_queue";

private String replyQueueName;

private QueueingConsumer consumer;

public RPCClient() throws IOException, TimeoutException {

ConnectionFactory factory = new ConnectionFactory();

factory.setHost("localhost");

connection = factory.newConnection();

channel = connection.createChannel();

replyQueueName = channel.queueDeclare().getQueue();

consumer = new QueueingConsumer(channel);

channel.basicConsume(replyQueueName, true, consumer);

}

public String call(String message) throws IOException, InterruptedException {

String response;

String corrID = UUID.randomUUID().toString();

AMQP.BasicProperties props = new AMQP.BasicProperties().builder()

.correlationId(corrID).replyTo(replyQueueName).build();

channel.basicPublish("", requestQueueName, props, message.getBytes("UTF-8"));

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

if (delivery.getProperties().getCorrelationId().equals(corrID)) {

response = new String(delivery.getBody(), "UTF-8");

break;

}

}

return response;

}

public void close() throws Exception {

connection.close();

}

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

RPCClient rpcClient = null;

String response;

try {

rpcClient = new RPCClient();

System.out.println("RPCClient Requesting fib(20)");

response = rpcClient.call("20");

System.out.println("RPCClient Got '" + response + "'");

} catch (Exception e) {

e.printStackTrace();

} finally {

if (rpcClient != null) {

rpcClient.close();

}

}

}

}

客户端代码解读

1：建立一个连接和通道，并声明了一个唯一的“回调”队列的答复

2：我们订阅回调队列，这样就可以得到RPC的响应

3：定义一个call方法用于发送当前的回调请求

4：生成一个唯一的correlationid，然后通过while循环来捕获合适的回应

5：我们请求信息，发送2个属性，replyTo 和correlationId

6：然后就是等待直到有合适的回应到达

7：while循环是做一个非常简单的工作，对于每一个响应消息，它检查是否有correlationid然后进行匹配。然后是就进行响应。

8：最后把响应返回到客户端。