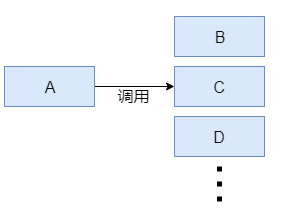
# **RabbitMQ 使用场景**

## **服务解耦**

假设有这样一个场景, 服务A产生数据, 而服务B,C,D需要这些数据, 那么我们可以在A服务中直接调用B,C,D服务,把数据传递到下游服务即可

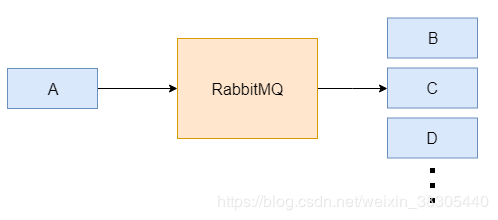
但是,随着我们的应用规模不断扩大,会有更多的服务需要A的数据,如果有几十甚至几百个下游服务,而且会不断变更,再加上还要考虑下游服务出错的情况,那么A服务中调用代码的维护会极为困难

这是由于服务之间耦合度过于紧密



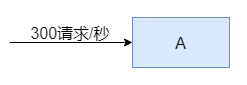
再来考虑用RabbitMQ解耦的情况

A服务只需要向消息服务器发送消息,而不用考虑谁需要这些数据;下游服务如果需要数据,自行从消息服务器订阅消息,不再需要数据时则取消订阅即可



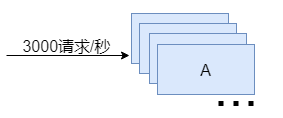
## **流量削峰**

假设我们有一个应用,平时访问量是每秒300请求,我们用一台服务器即可轻松应对



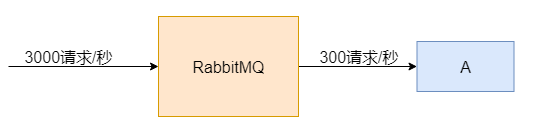
而在高峰期,访问量瞬间翻了十倍,达到每秒3000次请求,那么单台服务器肯定无法应对,这时我们可以考虑增加到10台服务器,来分散访问压力

但如果这种瞬时高峰的情况每天只出现一次,每次只有半小时,那么我们10台服务器在多数时间都只分担每秒几十次请求,这样就有点浪费资源了



这种情况,我们就可以使用RabbitMQ来进行流量削峰,高峰情况下,瞬间出现的大量请求数据,先发送到消息队列服务器,排队等待被处理,而我们的应用,可以慢慢的从消息队列接收请求数据进行处理,这样把数据处理时间拉长,以减轻瞬时压力

这是消息队列服务器非常典型的应用场景

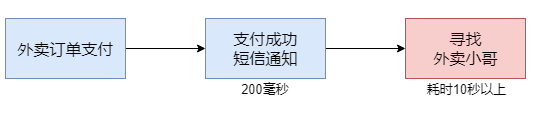


## **异步调用**

考虑定外卖支付成功的情况

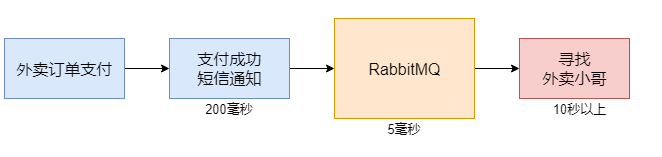
支付后要发送支付成功的通知,再寻找外卖小哥来进行配送,而寻找外卖小哥的过程非常耗时,尤其是高峰期,可能要等待几十秒甚至更长

这样就造成整条调用链路响应非常缓慢



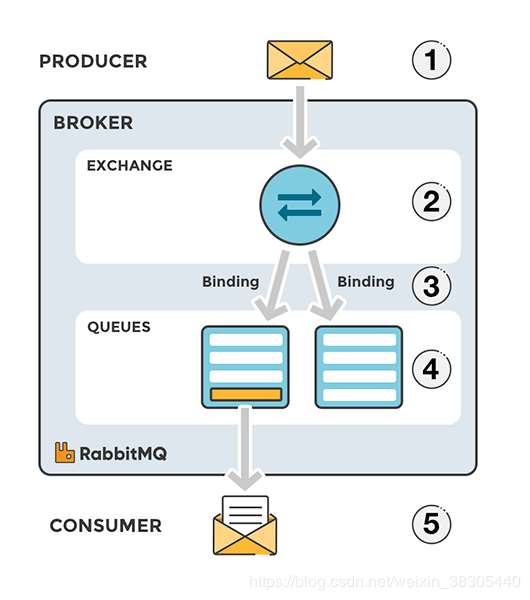
而如果我们引入RabbitMQ消息队列,订单数据可以发送到消息队列服务器,那么调用链路也就可以到此结束,订单系统则可以立即得到响应,整条链路的响应时间只有200毫秒左右

寻找外卖小哥的应用可以以异步的方式从消息队列接收订单消息,再执行耗时的寻找操作



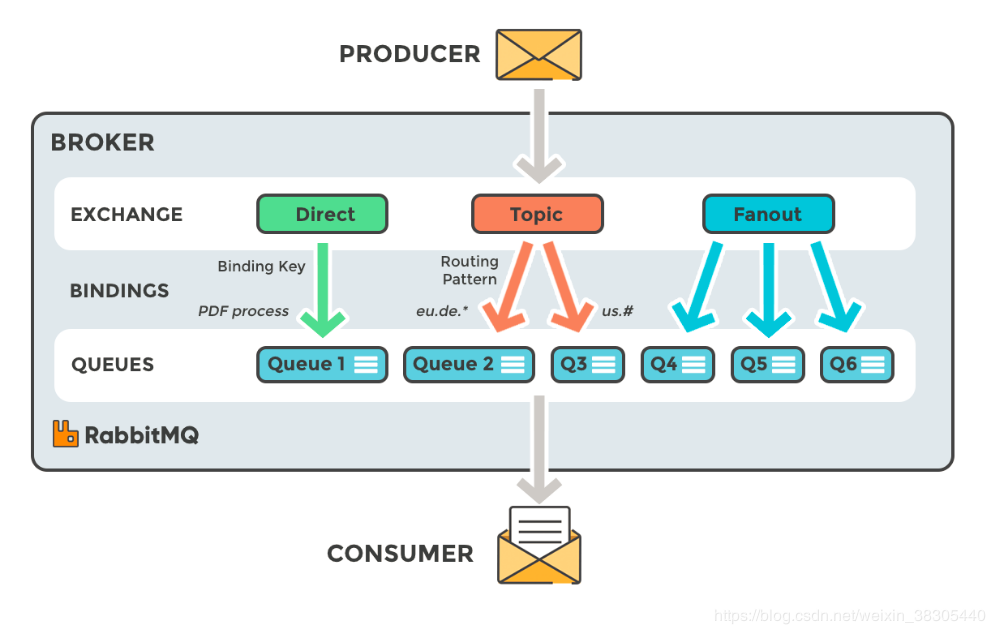
# **rabbitmq 基本概念**

RabbitMQ是一种消息中间件，用于处理来自客户端的异步消息。服务端将要发送的消息放入到队列池中。接收端可以根据RabbitMQ配置的转发机制接收服务端发来的消息。RabbitMQ依据指定的转发规则进行消息的转发、缓冲和持久化操作，主要用在多服务器间或单服务器的子系统间进行通信，是分布式系统标准的配置。



## **Exchange**

接受生产者发送的消息，并根据Binding规则将消息路由给服务器中的队列。ExchangeType决定了Exchange路由消息的行为。在RabbitMQ中，ExchangeType常用的有direct、Fanout和Topic三种。



## **Message Queue**

消息队列。我们发送给RabbitMQ的消息最后都会到达各种queue，并且存储在其中(如果路由找不到相应的queue则数据会丢失)，等待消费者来取。

## **Binding Key**

它表示的是Exchange与Message Queue是通过binding key进行联系的，这个关系是固定。

## **Routing Key**

生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则。这个routing key需要与Exchange Type及binding key联合使用才能生，我们的生产者只需要通过指定routing key来决定消息流向哪里。

# **rabbitmq安装**

在centos7上安装rabbitmq

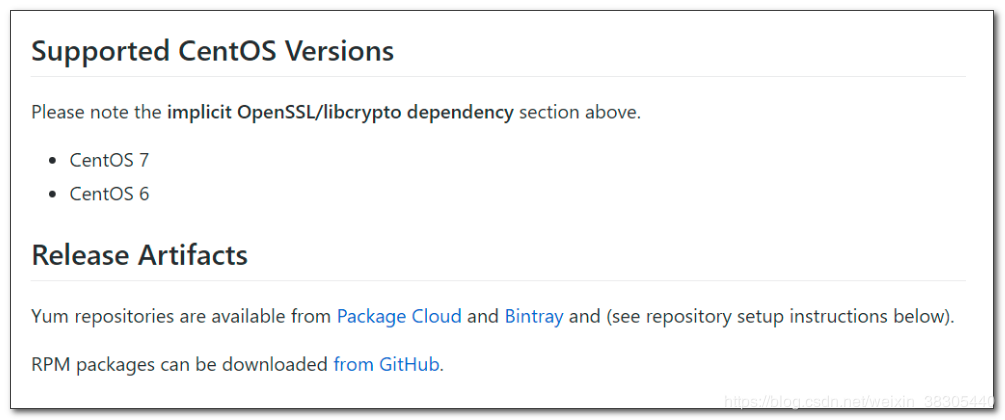
## **安装erlang语言库**

RabbitMQ使用了Erlang开发语言，Erlang是为电话交换机开发的语言，天生自带高并发光环，和高可用特性

### **rabbitmq官方精简的Erlang语言包**

0依赖rpm安装包

<https://github.com/rabbitmq/erlang-rpm>



### **下载和安装**

# 下载Erlang语言包

wget https://github.com/rabbitmq/erlang-rpm/releases/download/v21.2.6/erlang-21.2.6-1.el7.x86\_64.rpm

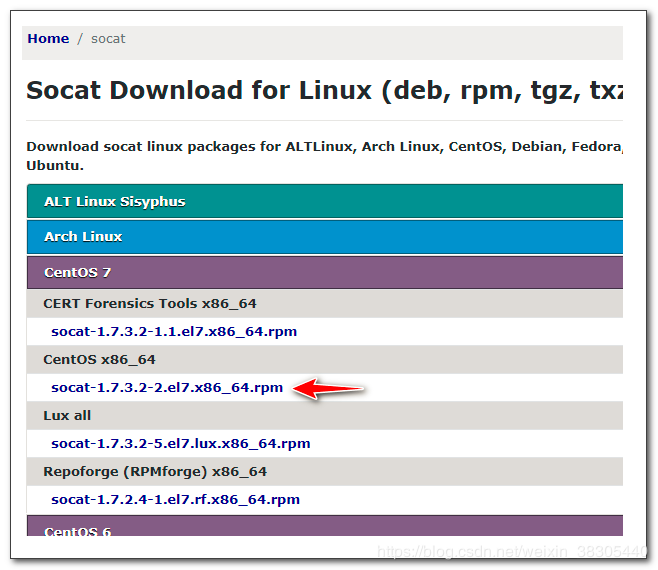
# 安装Erlang

rpm -ivh erlang-21.2.6-1.el7.x86\_64.rpm --force --nodeps

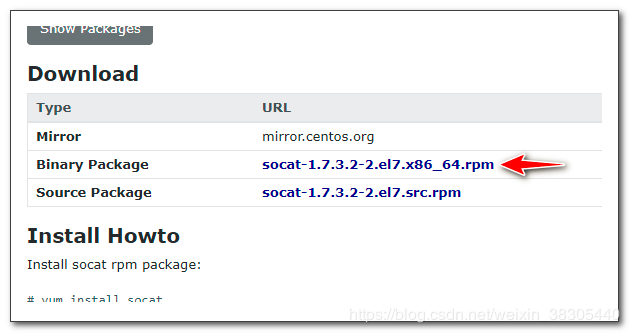
## **安装socat依赖**

### **socat依赖包**

<https://pkgs.org/download/socat>



<https://centos.pkgs.org/7/centos-x86_64/socat-1.7.3.2-2.el7.x86_64.rpm.html>



### **下载和安装**

# 下载 socat rpm

wget http://mirror.centos.org/centos/7/os/x86\_64/Packages/socat-1.7.3.2-2.el7.x86\_64.rpm

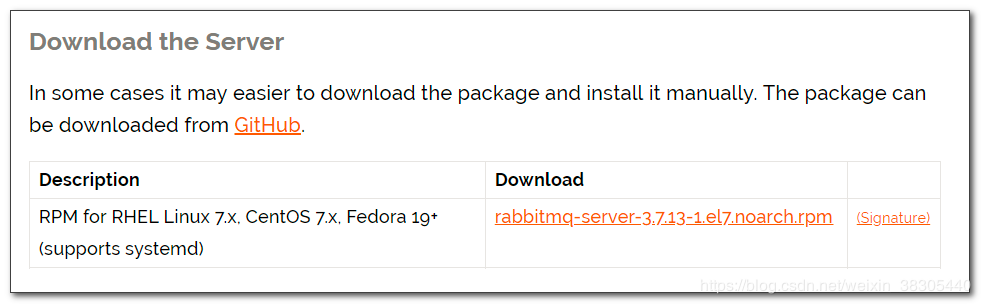
# 安装 socat 依赖包

rpm -ivh socat-1.7.3.2-2.el7.x86\_64.rpm

## **安装rabbitmq**

### **rabbitmq安装包**

* [http://www.rabbitmq.com/install-rpm.html#downloads](http://www.rabbitmq.com/install-rpm.html" \l "downloads)



### **下载和安装**

# 下载 rpm 包

wget https://github.com/rabbitmq/rabbitmq-server/releases/download/v3.7.13/rabbitmq-server-3.7.13-1.el7.noarch.rpm

# 安装 rpm 包

rpm -ivh rabbitmq-server-3.7.13-1.el7.noarch.rpm

## **rabbitmq启动和停止命令**

# 设置服务,开机自动启动

chkconfig rabbitmq-server on

# 启动服务

service rabbitmq-server start

# 停止服务

service rabbitmq-server stop

## **rabbitmq管理界面**

### **启用管理界面**

# 开启管理界面插件

rabbitmq-plugins enable rabbitmq\_management

# 防火墙打开 15672 管理端口

firewall-cmd --zone=public --add-port=15672/tcp --permanent

firewall-cmd --reload

### **访问**

访问服务器的15672端口,例如:  
http://192.168.64.140:15672

## **添加用户**

## **添加用户**

# 添加用户

rabbitmqctl add\_user admin admin

# 新用户设置用户为超级管理员

rabbitmqctl set\_user\_tags admin administrator

### **设置访问权限**



用户管理参考

<https://www.cnblogs.com/AloneSword/p/4200051.html>

## **开放客户端连接端口**

# 打开客户端连接端口

firewall-cmd --zone=public --add-port=5672/tcp --permanent

firewall-cmd --reload

主要端口介绍

4369 – erlang发现口

5672 – client端通信口

15672 – 管理界面ui端口

25672 – server间内部通信口

# **rabbitmq六种工作模式**

## **简单模式**

RabbitMQ是一个消息中间件，你可以想象它是一个邮局。当你把信件放到邮箱里时，能够确信邮递员会正确地递送你的信件。RabbitMq就是一个邮箱、一个邮局和一个邮递员。

发送消息的程序是生产者

队列就代表一个邮箱。虽然消息会流经RbbitMQ和你的应用程序，但消息只能被存储在队列里。队列存储空间只受服务器内存和磁盘限制，它本质上是一个大的消息缓冲区。多个生产者可以向同一个队列发送消息，多个消费者也可以从同一个队列接收消息

消费者等待从队列接收消息



### **pom.xml**

添加 slf4j 依赖, 和 rabbitmq amqp 依赖

<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.tedu</groupId>

<artifactId>rabbitmq</artifactId>

<version>0.0.1-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>5.4.3</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.8.0-alpha2</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.8.0-alpha2</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

### **生产者发送消息**

package rabbitmq.simple;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

public class Test1 {

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

//创建连接工厂,并设置连接信息

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setPort(5672);//可选,5672是默认端口

f.setUsername("admin");

f.setPassword("admin");

/\*

\* 与rabbitmq服务器建立连接,

\* rabbitmq服务器端使用的是nio,会复用tcp连接,

\* 并开辟多个信道与客户端通信

\* 以减轻服务器端建立连接的开销

\*/

Connection c = f.newConnection();

//建立信道

Channel ch = c.createChannel();

/\*

\* 声明队列,会在rabbitmq中创建一个队列

\* 如果已经创建过该队列，就不能再使用其他参数来创建

\*

\* 参数含义:

\* -queue: 队列名称

\* -durable: 队列持久化,true表示RabbitMQ重启后队列仍存在

\* -exclusive: 排他,true表示限制仅当前连接可用

\* -autoDelete: 当最后一个消费者断开后,是否删除队列

\* -arguments: 其他参数

\*/

ch.queueDeclare("helloworld", false,false,false,null);

/\*

\* 发布消息

\* 这里把消息向默认交换机发送.

\* 默认交换机隐含与所有队列绑定,routing key即为队列名称

\*

\* 参数含义:

\* -exchange: 交换机名称,空串表示默认交换机"(AMQP default)",不能用 null

\* -routingKey: 对于默认交换机,路由键就是目标队列名称

\* -props: 其他参数,例如头信息

\* -body: 消息内容byte[]数组

\*/

ch.basicPublish("", "helloworld", null, "Hello world!".getBytes());

System.out.println("消息已发送");

c.close();

}

}

### **消费者接收消息**

package rabbitmq.simple;

import java.io.IOException;

import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.DeliverCallback;

import com.rabbitmq.client.Delivery;

public class Test2 {

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

//连接工厂

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setUsername("admin");

f.setPassword("admin");

//建立连接

Connection c = f.newConnection();

//建立信道

Channel ch = c.createChannel();

//声明队列,如果该队列已经创建过,则不会重复创建

ch.queueDeclare("helloworld",false,false,false,null);

System.out.println("等待接收数据");

//收到消息后用来处理消息的回调对象

DeliverCallback callback = new DeliverCallback() {

@Override

public void handle(String consumerTag, Delivery message) throws IOException {

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

System.out.println("收到: "+msg);

}

};

//消费者取消时的回调对象

CancelCallback cancel = new CancelCallback() {

@Override

public void handle(String consumerTag) throws IOException {

}

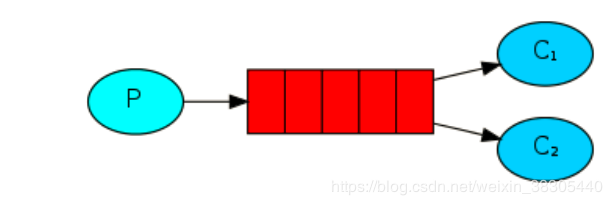
};

ch.basicConsume("helloworld", true, callback, cancel);

}

}

## **工作模式**



工作队列(即任务队列)背后的主要思想是避免立即执行资源密集型任务，并且必须等待它完成。相反，我们将任务安排在稍后完成。

我们将任务封装为消息并将其发送到队列。后台运行的工作进程将获取任务并最终执行任务。当运行多个消费者时，任务将在它们之间分发。

使用任务队列的一个优点是能够轻松地并行工作。如果我们正在积压工作任务，我们可以添加更多工作进程，这样就可以轻松扩展。

### **生产者发送消息**

这里模拟耗时任务,发送的消息中,每个点使工作进程暂停一秒钟,例如"Hello…"将花费3秒钟来处理

package rabbitmq.workqueue;

import java.util.Scanner;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

public class Test1 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setPort(5672);

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//参数:queue,durable,exclusive,autoDelete,arguments

ch.queueDeclare("helloworld", false,false,false,null);

while (true) {

//控制台输入的消息发送到rabbitmq

System.out.print("输入消息: ");

String msg = new Scanner(System.in).nextLine();

//如果输入的是"exit"则结束生产者进程

if ("exit".equals(msg)) {

break;

}

//参数:exchage,routingKey,props,body

ch.basicPublish("", "helloworld", null, msg.getBytes());

System.out.println("消息已发送: "+msg);

}

c.close();

}

}

### **消费者接收消息**

package rabbitmq.workqueue;

import java.io.IOException;

import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.DeliverCallback;

import com.rabbitmq.client.Delivery;

public class Test2 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

ch.queueDeclare("helloworld",false,false,false,null);

System.out.println("等待接收数据");

//收到消息后用来处理消息的回调对象

DeliverCallback callback = new DeliverCallback() {

@Override

public void handle(String consumerTag, Delivery message) throws IOException {

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

System.out.println("收到: "+msg);

//遍历字符串中的字符,每个点使进程暂停一秒

for (int i = 0; i < msg.length(); i++) {

if (msg.charAt(i)=='.') {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

}

}

System.out.println("处理结束");

}

};

//消费者取消时的回调对象

CancelCallback cancel = new CancelCallback() {

@Override

public void handle(String consumerTag) throws IOException {

}

};

ch.basicConsume("helloworld", true, callback, cancel);

}

}

### **运行测试**

运行:

一个生产者

* 两个消费者

生产者发送多条消息,  
如: 1,2,3,4,5. 两个消费者分别收到:

* 消费者一: 1,3,5
* 消费者二: 2,4

rabbitmq在所有消费者中轮询分发消息,把消息均匀地发送给所有消费者

### **消息确认**

一个消费者接收消息后,在消息没有完全处理完时就挂掉了,那么这时会发生什么呢?

就现在的代码来说,rabbitmq把消息发送给消费者后,会立即删除消息,那么消费者挂掉后,它没来得及处理的消息就会丢失

如果生产者发送以下消息:  
  
1…  
  
2  
  
3  
  
4  
  
5

两个消费者分别收到:

消费者一: 1…, 3, 5

消费者二: 2, 4

当消费者一收到所有消息后,要话费7秒时间来处理第一条消息,这期间如果关闭该消费者,那么1未处理完成,3,5则没有被处理

我们并不想丢失任何消息, 如果一个消费者挂掉,我们想把它的任务消息派发给其他消费者

为了确保消息不会丢失，rabbitmq支持消息确认(回执)。当一个消息被消费者接收到并且执行完成后，消费者会发送一个ack (acknowledgment) 给rabbitmq服务器, 告诉他我已经执行完成了，你可以把这条消息删除了。

如果一个消费者没有返回消息确认就挂掉了（信道关闭，连接关闭或者TCP链接丢失），rabbitmq就会明白，这个消息没有被处理完成，rebbitmq就会把这条消息重新放入队列，如果在这时有其他的消费者在线，那么rabbitmq就会迅速的把这条消息传递给其他的消费者，这样就确保了没有消息会丢失。

这里不存在消息超时, rabbitmq只在消费者挂掉时重新分派消息, 即使消费者花非常久的时间来处理消息也可以

手动消息确认默认是开启的，前面的例子我们通过autoAck=ture把它关闭了。我们现在要把它设置为false，然后工作进程处理完意向任务时,发送一个消息确认(回执)。

package rabbitmq.workqueue;

import java.io.IOException;

import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.DeliverCallback;

import com.rabbitmq.client.Delivery;

public class Test2 {

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

//连接工厂

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setUsername("admin");

f.setPassword("admin");

//建立连接

Connection c = f.newConnection();

//建立信道

Channel ch = c.createChannel();

//声明队列

ch.queueDeclare("helloworld",false,false,false,null);

System.out.println("等待接收数据");

//收到消息后用来处理消息的回调对象

DeliverCallback callback = new DeliverCallback() {

@Override

public void handle(String consumerTag, Delivery message) throws IOException {

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

System.out.println("收到: "+msg);

for (int i = 0; i < msg.length(); i++) {

if (msg.charAt(i)=='.') {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

}

}

System.out.println("处理结束");

//发送回执

ch.basicAck(message.getEnvelope().getDeliveryTag(), false);

}

};

//消费者取消时的回调对象

CancelCallback cancel = new CancelCallback() {

@Override

public void handle(String consumerTag) throws IOException {

}

};

//autoAck设置为false,则需要手动确认发送回执

ch.basicConsume("helloworld", false, callback, cancel);

}

}

使用以上代码，就算杀掉一个正在处理消息的工作进程也不会丢失任何消息，工作进程挂掉之后，没有确认的消息就会被自动重新传递。

忘记确认(ack)是一个常见的错误, 这样后果是很严重的, 由于未确认的消息不会被释放, rabbitmq会吃掉越来越多的内存

可以使用下面命令打印工作队列中未确认消息的数量

rabbitmqctl list\_queues name messages\_ready messages\_unacknowledged

### **消息持久化**

当rabbitmq关闭时, 我们队列中的消息仍然会丢失, 除非明确要求它不要丢失数据

要求rabbitmq不丢失数据要做如下两点: 把队列和消息都设置为可持久化(durable)

队列设置为可持久化, 可以在定义队列时指定参数durable为true

//第二个参数是持久化参数durable

ch.queueDeclare("helloworld", true, false, false, null);

由于之前我们已经定义过队列"hello"是不可持久化的, 对已存在的队列, rabbitmq不允许对其定义不同的参数, 否则会出错, 所以这里我们定义一个不同名字的队列"task\_queue"

*//定义一个新的队列,名为 task\_queue*

*//第二个参数是持久化参数 durable*

*ch.queueDeclare("task\_queue", true, false, false, null);*

生产者和消费者代码都要修改

这样即使rabbitmq重新启动, 队列也不会丢失. 现在我们再设置队列中消息的持久化, 使用

MessageProperties.PERSISTENT\_TEXT\_PLAIN参数

//第三个参数设置消息持久化

ch.basicPublish("", "task\_queue",

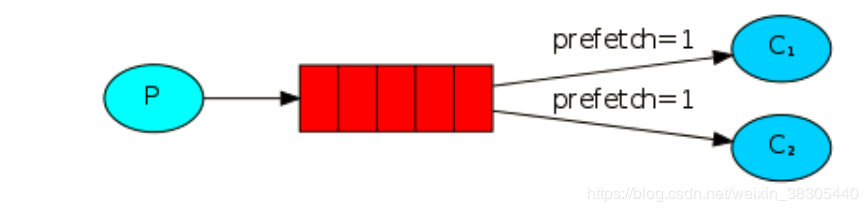
MessageProperties.PERSISTENT\_TEXT\_PLAIN,

msg.getBytes());

### **合理地分发**

rabbitmq会一次把多个消息分发给消费者, 这样可能造成有的消费者非常繁忙, 而其它消费者空闲. 而rabbitmq对此一无所知, 仍然会均匀的分发消息

我们可以使用 basicQos(1) 方法, 这告诉rabbitmq一次只向消费者发送一条消息, 在返回确认回执前, 不要向消费者发送新消息. 而是把消息发给下一个空闲的消费者



下面是"工作模式"最终完成的生产者和消费者代码

### **生产者代码**

package rabbitmq.workqueue;

import java.util.Scanner;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.MessageProperties;

public class Test3 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setPort(5672);

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//第二个参数设置队列持久化

ch.queueDeclare("task\_queue", true,false,false,null);

while (true) {

System.out.print("输入消息: ");

String msg = new Scanner(System.in).nextLine();

if ("exit".equals(msg)) {

break;

}

//第三个参数设置消息持久化

ch.basicPublish("", "task\_queue", MessageProperties.PERSISTENT\_TEXT\_PLAIN, msg.getBytes("UTF-8"));

System.out.println("消息已发送: "+msg);

}

c.close();

}

}

### **消费者代码**

package rabbitmq.workqueue;

import java.io.IOException;

import java.util.concurrent.TimeoutException;

import com.rabbitmq.client.CancelCallback;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.DeliverCallback;

import com.rabbitmq.client.Delivery;

public class Test4 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//第二个参数设置队列持久化

ch.queueDeclare("task\_queue",true,false,false,null);

System.out.println("等待接收数据");

ch.basicQos(1); //一次只接收一条消息

//收到消息后用来处理消息的回调对象

DeliverCallback callback = new DeliverCallback() {

@Override

public void handle(String consumerTag, Delivery message) throws IOException {

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

System.out.println("收到: "+msg);

for (int i = 0; i < msg.length(); i++) {

if (msg.charAt(i)=='.') {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

}

}

}

System.out.println("处理结束");

//发送回执

ch.basicAck(message.getEnvelope().getDeliveryTag(), false);

}

};

//消费者取消时的回调对象

CancelCallback cancel = new CancelCallback() {

@Override

public void handle(String consumerTag) throws IOException {

}

};

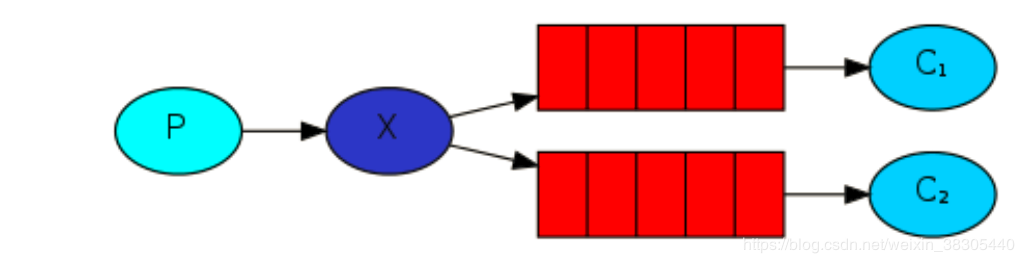
//autoAck设置为false,则需要手动确认发送回执

ch.basicConsume("task\_queue", false, callback, cancel);

}

}

## **发布订阅模式**



在前面的例子中，我们任务消息只交付给一个工作进程。在这部分，我们将做一些完全不同的事情——我们将向多个消费者传递同一条消息。这种模式称为“发布/订阅”。

为了说明该模式，我们将构建一个简单的日志系统。它将由两个程序组成——第一个程序将发出日志消息，第二个程序接收它们。

在我们的日志系统中，接收程序的每个运行副本都将获得消息。这样，我们就可以运行一个消费者并将日志保存到磁盘; 同时我们可以运行另一个消费者在屏幕上打印日志。

最终, 消息会被广播到所有消息接受者

### **Exchanges 交换机**

RabbitMQ消息传递模型的核心思想是，生产者永远不会将任何消息直接发送到队列。实际上，通常生产者甚至不知道消息是否会被传递到任何队列。

相反，生产者只能向交换机(Exchange)发送消息。交换机是一个非常简单的东西。一边接收来自生产者的消息，另一边将消息推送到队列。交换器必须确切地知道如何处理它接收到的消息。它应该被添加到一个特定的队列中吗?它应该添加到多个队列中吗?或者它应该被丢弃。这些规则由exchange的类型定义。

有几种可用的交换类型:direct、topic、header和fanout。我们将关注最后一个——fanout。让我们创建一个这种类型的交换机，并称之为 logs: ch.exchangeDeclare("logs", "fanout");

fanout交换机非常简单。它只是将接收到的所有消息广播给它所知道的所有队列。这正是我们的日志系统所需要的。

我们前面使用的队列具有特定的名称(还记得hello和task\_queue吗?)能够为队列命名对我们来说至关重要——我们需要将工作进程指向同一个队列,在生产者和消费者之间共享队列。

但日志记录案例不是这种情况。我们想要接收所有的日志消息，而不仅仅是其中的一部分。我们还只对当前的最新消息感兴趣，而不是旧消息。

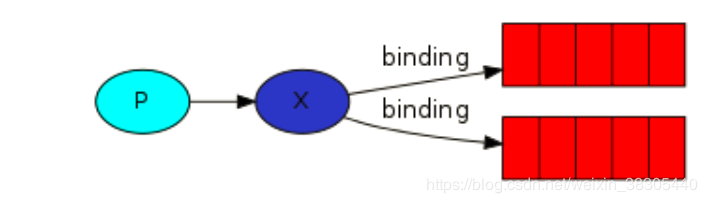
要解决这个问题，我们需要两件事。首先，每当我们连接到Rabbitmq时，我们需要一个新的空队列。为此，我们可以创建一个具有随机名称的队列，或者，更好的方法是让服务器为我们选择一个随机队列名称。其次，一旦断开与使用者的连接，队列就会自动删除。在Java客户端中，当我们不向queueDeclare()提供任何参数时，会创建一个具有生成名称的、非持久的、独占的、自动删除队列

//自动生成队列名

//非持久,独占,自动删除

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

### **绑定 Bindings**



我们已经创建了一个fanout交换机和一个队列。现在我们需要告诉exchange向指定队列发送消息。exchange和队列之间的关系称为绑定。

//指定的队列,与指定的交换机关联起来

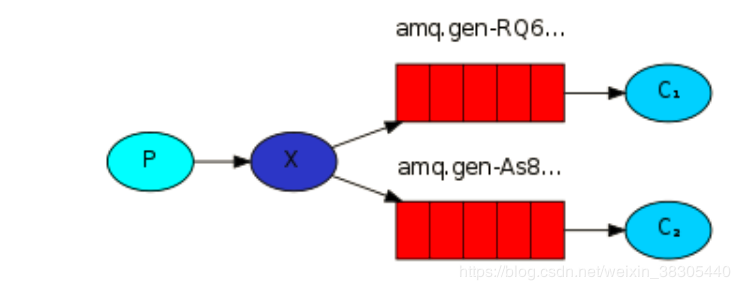
//成为绑定 -- binding

//第三个参数时 routingKey, 由于是fanout交换机, 这里忽略 routingKey ch.queueBind(queueName, "logs", "");

现在, logs交换机将会向我们指定的队列添加消息

列出绑定关系:  
  
rabbitmqctl list\_bindings

### **完成的代码**



#### **生产者**

生产者发出日志消息，看起来与前一教程没有太大不同。最重要的更改是，我们现在希望将消息发布到logs交换机，而不是无名的日志交换机。我们需要在发送时提供一个routingKey，但是对于fanout交换机类型，该值会被忽略。

package rabbitmq.publishsubscribe;

import java.util.Scanner;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

public class Test1 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setPort(5672);

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//定义名字为logs的交换机,交换机类型为fanout

//这一步是必须的，因为禁止发布到不存在的交换。

ch.exchangeDeclare("logs", "fanout");

while (true) {

System.out.print("输入消息: ");

String msg = new Scanner(System.in).nextLine();

if ("exit".equals(msg)) {

break;

}

//第一个参数,向指定的交换机发送消息

//第二个参数,不指定队列,由消费者向交换机绑定队列

//如果还没有队列绑定到交换器，消息就会丢失，

//但这对我们来说没有问题;即使没有消费者接收，我们也可以安全地丢弃这些信息。

ch.basicPublish("logs", "", null, msg.getBytes("UTF-8"));

System.out.println("消息已发送: "+msg);

}

c.close();

}

}

#### **消费者**

如果还没有队列绑定到交换器，消息就会丢失，但这对我们来说没有问题;如果还没有消费者在听，我们可以安全地丢弃这些信息。  
ReceiveLogs.java代码:

package rabbitmq.publishsubscribe;

import java.io.IOException;

import com.rabbitmq.client.CancelCallback;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.DeliverCallback;

import com.rabbitmq.client.Delivery;

public class Test2 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//定义名字为 logs 的交换机, 它的类型是 fanout

ch.exchangeDeclare("logs", "fanout");

//自动生成对列名,

//非持久,独占,自动删除

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

//把该队列,绑定到 logs 交换机

//对于 fanout 类型的交换机, routingKey会被忽略，不允许null值

ch.queueBind(queueName, "logs", "");

System.out.println("等待接收数据");

//收到消息后用来处理消息的回调对象

DeliverCallback callback = new DeliverCallback() {

@Override

public void handle(String consumerTag, Delivery message) throws IOException {

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

System.out.println("收到: "+msg);

}

};

//消费者取消时的回调对象

CancelCallback cancel = new CancelCallback() {

@Override

public void handle(String consumerTag) throws IOException {

}

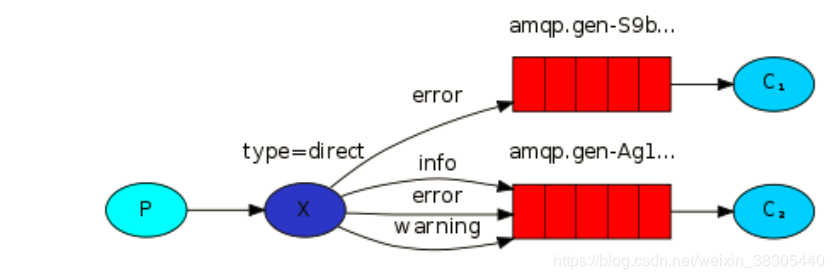
};

ch.basicConsume(queueName, true, callback, cancel);

}

}

## **路由模式**



在上一小节，我们构建了一个简单的日志系统。我们能够向多个接收者广播日志消息。

在这一节，我们将向其添加一个特性—我们将只订阅所有消息中的一部分。例如，我们只接收关键错误消息并保存到日志文件(以节省磁盘空间)，同时仍然能够在控制台上打印所有日志消息。

### **绑定 Bindings**

在上一节，我们已经创建了队列与交换机的绑定。使用下面这样的代码:

ch.queueBind(queueName, "logs", "");

绑定是交换机和队列之间的关系。这可以简单地理解为:队列对来自此交换的消息感兴趣。

绑定可以使用额外的routingKey参数。为了避免与basic\_publish参数混淆，我们将其称为bindingKey。这是我们如何创建一个键绑定:

ch.queueBind(queueName, EXCHANGE\_NAME, "black");

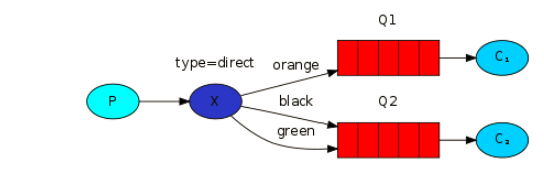
bindingKey的含义取决于交换机类型。我们前面使用的fanout交换机完全忽略它。

### **直连交换机 Direct exchange**

上一节中的日志系统向所有消费者广播所有消息。我们希望扩展它，允许根据消息的严重性过滤消息。例如，我们希望将日志消息写入磁盘的程序只接收关键error，而不是在warning或info日志消息上浪费磁盘空间。

前面我们使用的是fanout交换机，这并没有给我们太多的灵活性——它只能进行简单的广播。

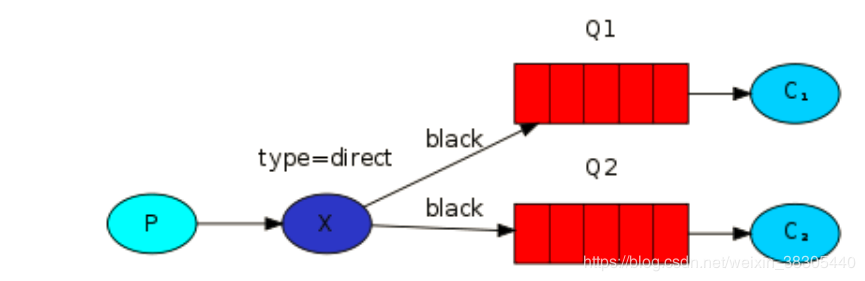
我们将用直连交换机(Direct exchange)代替。它背后的路由算法很简单——消息传递到bindingKey与routingKey完全匹配的队列。为了说明这一点，请考虑以下设置



其中我们可以看到直连交换机X，它绑定了两个队列。第一个队列用绑定键orange绑定，第二个队列有两个绑定，一个绑定black，另一个绑定键green。

这样设置，使用路由键orange发布到交换器的消息将被路由到队列Q1。带有black或green路由键的消息将转到Q2。而所有其他消息都将被丢弃。

### **多重绑定 Multiple bindings**



使用相同的bindingKey绑定多个队列是完全允许的。如图所示，可以使用binding key black将X与Q1和Q2绑定。在这种情况下，直连交换机的行为类似于fanout，并将消息广播给所有匹配的队列。一条路由键为black的消息将同时发送到Q1和Q2。

### **发送日志**

我们将在日志系统中使用这个模型。我们把消息发送到一个Direct交换机，而不是fanout。我们将提供日志级别作为routingKey。这样，接收程序将能够选择它希望接收的级别。让我们首先来看发出日志。

和前面一样，我们首先需要创建一个exchange:

//参数1: 交换机名

//参数2: 交换机类型

ch.exchangeDeclare("direct\_logs", "direct");

接着来看发送消息的代码

//参数1: 交换机名

//参数2: routingKey, 路由键,这里我们用日志级别,如"error","info","warning"

//参数3: 其他配置属性

//参数4: 发布的消息数据

ch.basicPublish("direct\_logs", "error", null, message.getBytes());

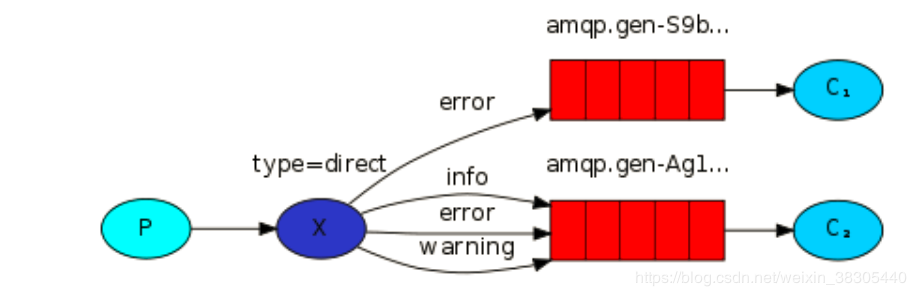
### **订阅**

接收消息的工作原理与前面章节一样，但有一个例外——我们将为感兴趣的每个日志级别创建一个新的绑定, 示例代码如下:

ch.queueBind(queueName, "logs", "info");

ch.queueBind(queueName, "logs", "warning");

### **完整的代码**



#### **生产者**

package rabbitmq.routing;

import java.util.Random;

import java.util.Scanner;

import com.rabbitmq.client.BuiltinExchangeType;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

public class Test1 {

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

String[] a = {"warning", "info", "error"};

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setPort(5672);

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//参数1: 交换机名

//参数2: 交换机类型

ch.exchangeDeclare("direct\_logs", BuiltinExchangeType.DIRECT);

while (true) {

System.out.print("输入消息: ");

String msg = new Scanner(System.in).nextLine();

if ("exit".equals(msg)) {

break;

}

//随机产生日志级别

String level = a[new Random().nextInt(a.length)];

//参数1: 交换机名

//参数2: routingKey, 路由键,这里我们用日志级别,如"error","info","warning"

//参数3: 其他配置属性

//参数4: 发布的消息数据

ch.basicPublish("direct\_logs", level, null, msg.getBytes());

System.out.println("消息已发送: "+level+" - "+msg);

}

c.close();

}

}

#### **消费者**

package rabbitmq.routing;

import java.io.IOException;

import java.util.Scanner;

import com.rabbitmq.client.BuiltinExchangeType;

import com.rabbitmq.client.CancelCallback;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.DeliverCallback;

import com.rabbitmq.client.Delivery;

public class Test2 {

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

ConnectionFactory f = new ConnectionFactory();

f.setHost("192.168.64.140");

f.setUsername("admin");

f.setPassword("admin");

Connection c = f.newConnection();

Channel ch = c.createChannel();

//定义名字为 direct\_logs 的交换机, 它的类型是 "direct"

ch.exchangeDeclare("direct\_logs", BuiltinExchangeType.DIRECT);

//自动生成对列名,

//非持久,独占,自动删除

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

System.out.println("输入接收的日志级别,用空格隔开:");

String[] a = new Scanner(System.in).nextLine().split("\\s");

//把该队列,绑定到 direct\_logs 交换机

//允许使用多个 bindingKey

for (String level : a) {

ch.queueBind(queueName, "direct\_logs", level);

}

System.out.println("等待接收数据");

//收到消息后用来处理消息的回调对象

DeliverCallback callback = new DeliverCallback() {

@Override

public void handle(String consumerTag, Delivery message) throws IOException {

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

String routingKey = message.getEnvelope().getRoutingKey();

System.out.println("收到: "+routingKey+" - "+msg);

}

};

//消费者取消时的回调对象

CancelCallback cancel = new CancelCallback() {

@Override

public void handle(String consumerTag) throws IOException {

}

};

ch.basicConsume(queueName, true, callback, cancel);

}

}