RabbitMQ

# 1 安装

## 1.1 Linux下Erlang语言的安装

1、下载otp\_src\_18.3.tar.gz

       http://erlang.org/download/otp\_src\_18.3.tar.gz

2、解压otp\_src\_18.3.tar.gz

 tar -xzvf otp\_src\_18.3.tar.gz

3、otp\_src\_18.3.tar.gz编译需要依赖下面的包

* gcc
* gcc-c++
* unixODBC-devel
* openssl-devel
* ncurses-devel

安装依赖包： yum install -y gcc gcc-c++ unixODBC-devel  openssl-devel ncurses-devel

4、安装otp\_src\_18.3.tar.gz

      ./configure --prefix=/opt/erlang --without-javac

      make

      make install

5、设置环境变量

      vim  ~/.base\_profile  
      PATH=$PATH:/opt/erlang/bin  
      export PATH  
      :wq  
      source  ~/.base\_profile

注：config这一步会显示："wx: wxWidgets not found, wx will NOT be usable"。

       不安装wxWidgets不能使用debugger。

       安装wxWidgets 需使用下面configure

       ./configure --with-opengl --enable-debug --enable-unicode

## 1.2 安装rabbitMQ

下载网址：<http://www.rabbitmq.com/download.html>

### 1.2.1\*安装RabbitMQ

上传rabbitmq-server-3.4.1-1.noarch.rpm文件到/usr/local/src/rabbitmq/

安装：

rpm –ivh rabbitmq-server.3.4.1-1.noarch.rpm

### 1.2.2设置配置文件

cd /etc/rabbitmq

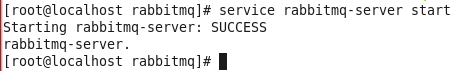
cp /usr/share/doc/rabbitmq-server-3.4.1/rabbitmq.config.example /etc/rabbitmq/

(安装目录：/usr/share/doc/rabbitmq-server-3.4.1)

mv rabbitmq.config.example rabbitmq.config

### 1.2.3启动、停止服务

service rabbitmq-server start



service rabbitmq-server stop

service rabbitmq-server restart

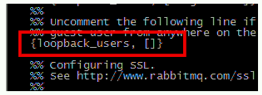
设置开机启动

chkconfig rabbitmq-server on

### 1.2.4开启用户远程访问

vi /etc/rabbitmq/rabbitmq.config P64行

注意：有两处，修改如图处

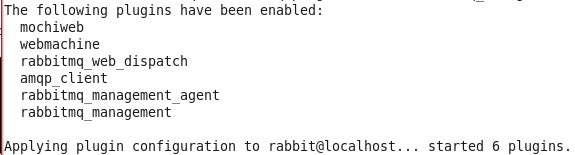


去掉前面的两个%%，和最后面的逗号，保存。

### 1.2.5开启后台管理插件

执行下面命令，开启web界面管理工具

rabbitmq-plugins enable rabbitmq\_management



service rabbitmq-server restart

### 1.2.6访问管理平台

firefox #打开虚拟机上的火狐

<http://localhost:15672/>

默认用户名密码都为guest

注意：出于安全的考虑，guest这个默认的用户只能通过http://localhost:15672 本地来登录，不能外部服务器登录，也就是不能远程访问，这对于服务器上没有安装桌面的情况是无法管理维护的。必须新创建管理员账号。

### 1.2.7防火墙打开15672端口和5672端口

/sbin/iptables –I INPUT –p tcp --dport 15672 –j ACCEPT

/sbin/iptables –I INPUT –p tcp --dport 5672 –j ACCEPT

/etc/rc.d/init.d/iptables save

/etc/init.d/iptables status

# 2 tabbit的使用

1. 管理地址：<http://localhost:15672>

出现下面界面，说明安装成功。

系统提供默认用户名guest密码guest

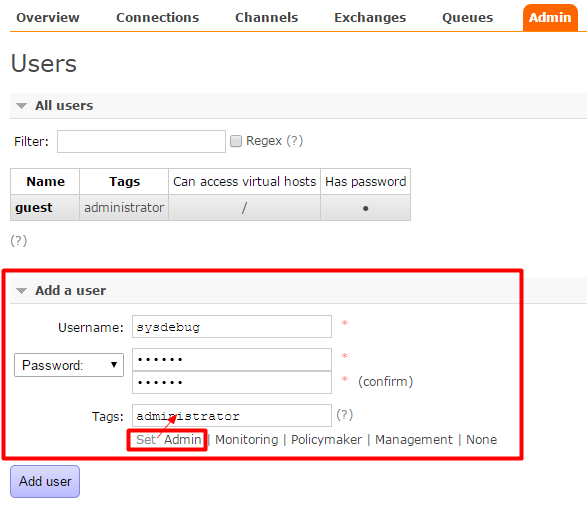


1. 功能介绍



### 2.1添加用户

用户名sysdebug，密码123456



添加后



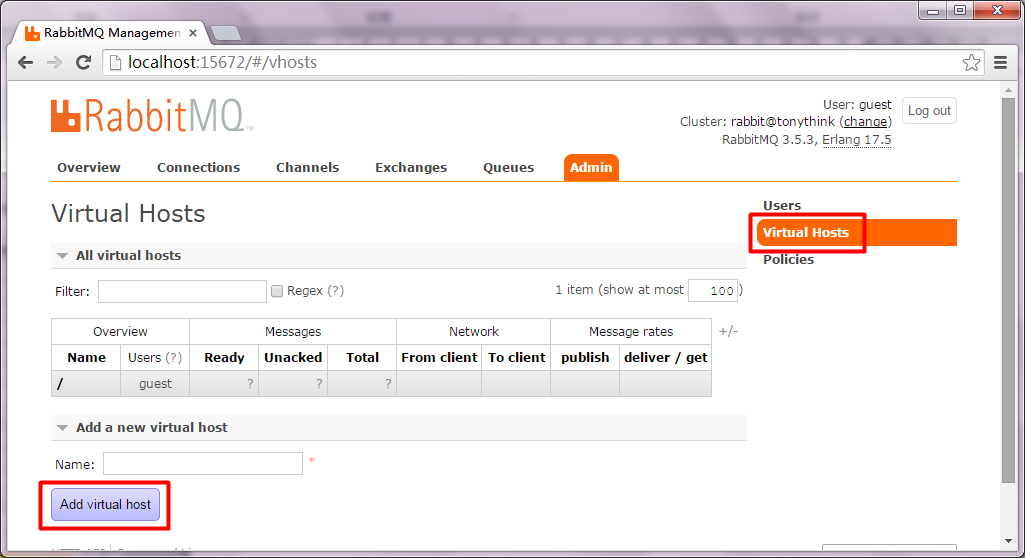
用户角色：



|  |  |  |
| --- | --- | --- |
| **序号** | **角色名称** | **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rabbingmq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 |
| **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 | 无法登录管理控制台，通常就是普通的生产者和消费者。 |

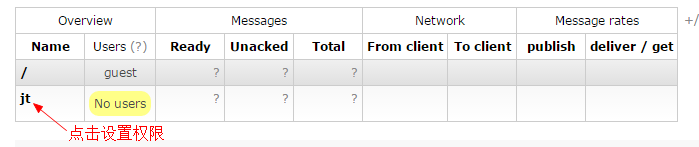
### 2.2添加VirtualHosts

相当于mysql中的数据库

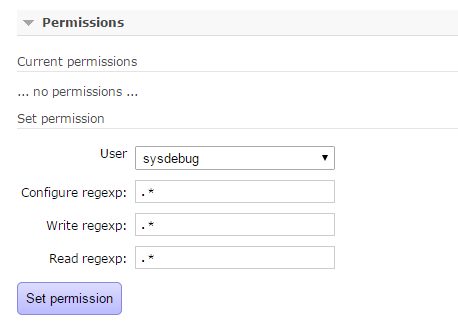


添加/jt，注意前面的/，不然数据可能造成丢失。

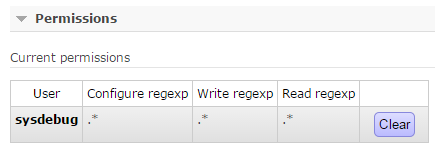
### 2.3权限设置



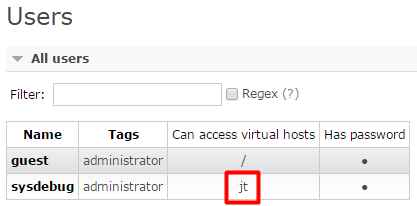
选择可以访问的用户



设置好权限



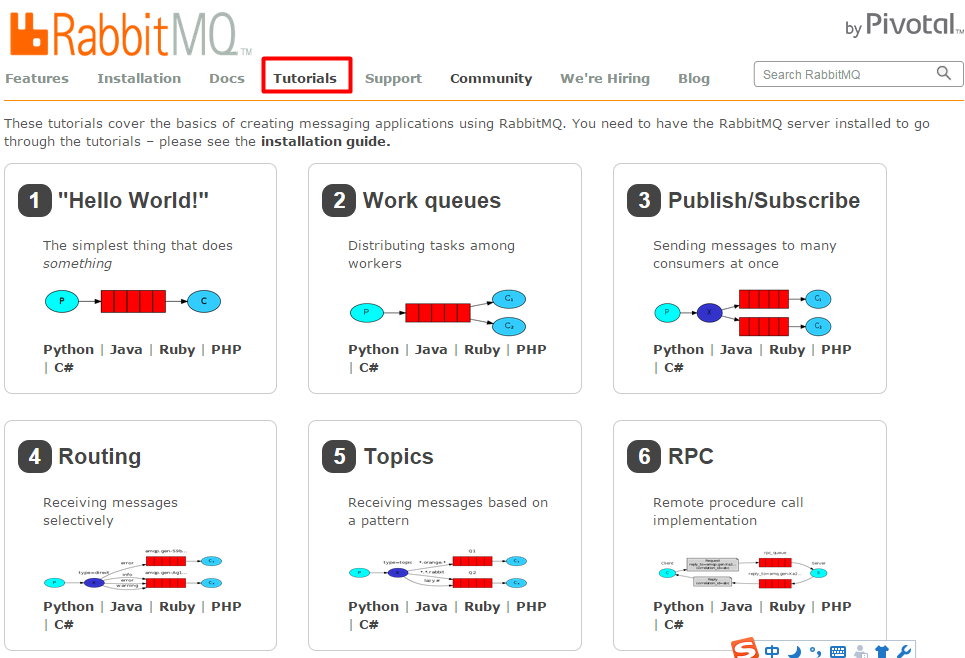
切换到User界面



2.4 代码使用

提供6中消息队列

http://www.rabbitmq.com/getstarted.html



### 引入依赖包

<!-- 消息队列 -->

<dependency>

<groupId>com.rabbitmq</groupId>

<artifactId>amqp-client</artifactId>

<version>3.5.1</version>

</dependency>

<dependency>

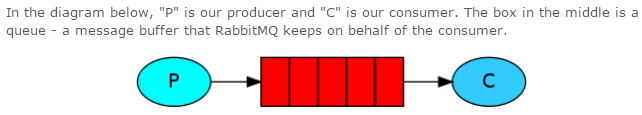
<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

<version>1.4.0.RELEASE</version>

</dependency>

### Hello World



P代表生产者，C代表消费者，红色代码消息队列。P将消息发送到消息队列，C对消息进行处理。

工具类，获取链接

package com.rabbitmq.util;

import com.rabbitmq.client.ConnectionFactory;

import com.rabbitmq.client.Connection;

public class ConnectionUtil {

public static Connection getConnection() throws Exception {

//定义连接工厂

ConnectionFactory factory = new ConnectionFactory();

//设置服务地址

factory.setHost("localhost");

//端口，HTTP管理端口默认15672，访问端口默认5672

factory.setPort(5672);

//设置账号信息，用户名、密码、vhost

factory.setVirtualHost("jtmq"); //注意这个值，vh配置如果加了斜杠，这里也必须加

factory.setUsername("jtmqadmin");

factory.setPassword("123456");

// 通过工程获取连接

Connection connection = factory.newConnection();

return connection;

}

}

### 发送消息

package com.rabbitmq.simple;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.util.ConnectionUtil;

public class Send {

private final static String QUEUE\_NAME = "test\_queue";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

// 从连接中创建通道

Channel channel = connection.createChannel();

// 声明（创建）队列

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

// 消息内容

String message = "Hello World!";

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

System.out.println(" [x] Sent '" + message + "'");

//关闭通道和连接

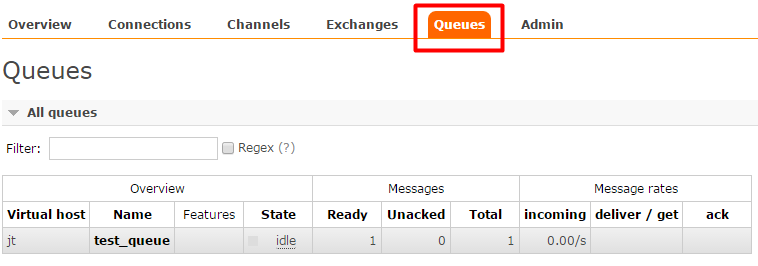
channel.close();

connection.close();

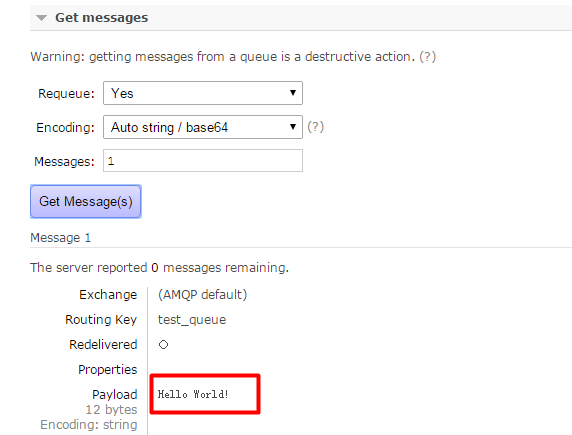
}

}

### 控制台查看消息



注意，需要用sysdebug登录才能看到。



### 接收消息

package com.rabbitmq.simple;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv {

private final static String QUEUE\_NAME = "test\_queue";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

}

}

}

接收消息后，将消息删除。

## 工作模式work（竞争）



一个消息产生者，多个消息的消费者。竞争抢消息。

### 产生多个消息

package com.rabbitmq.work;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.util.ConnectionUtil;

public class Send {

private final static String QUEUE\_NAME = "test\_queue\_work";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

for (int i = 0; i < 100; i++) {

// 消息内容

String message = "" + i;

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

System.out.println(" [x] Sent '" + message + "'");

Thread.sleep(i \* 10); //越往后，停歇的时间越长

}

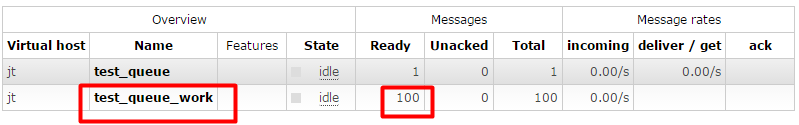
channel.close();

connection.close();

}

}

控制台查看消息



产生100条消息。

### 消费者1

package com.rabbitmq.work;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv {

private final static String QUEUE\_NAME = "test\_queue\_work";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 同一时刻服务器只会发一条消息给消费者，每一次服务器只会向客户端发送一条

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

//休眠

Thread.sleep(10);

// 返回确认状态

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

}

}

}

### 消费者2

package com.rabbitmq.work;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv2 {

private final static String QUEUE\_NAME = "test\_queue\_work";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 同一时刻服务器只会发一条消息给消费者，每一次服务器只会向客户端发送一条

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成状态

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

// 休眠1秒

Thread.sleep(1000);

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

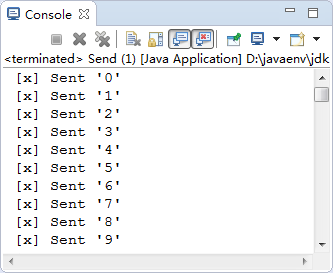
}

}

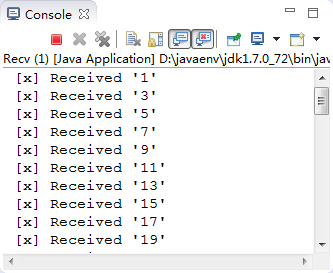
}

先注释掉两个消费者中的channel.basicQos(1);运行产生消息，同时运行消费者1和消费者2。消费者1中延时Thread.sleep(10)~10毫秒，消费者2中延时Thread.sleep(1000)~1秒。模拟两个消费者处理速度是不同的。

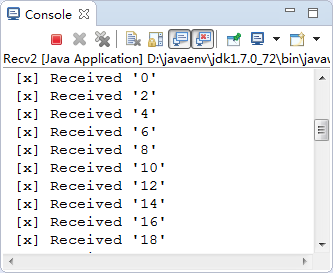
发送消息：



消费者1：



消费者2：



观察控制台，会发现在产生消息的同时，两个消费者开始同时消费消息。但产生一个现象。消费者1消费偶数消息，消费者2消费奇数消息。平均分配合理吗？

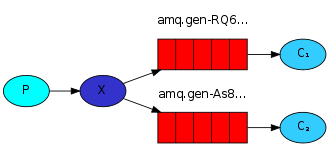
这明显不合理，干的快的应该处理的多，干的慢的应该处理的少。应该是能者多劳，多劳多得。

### 消息负载均衡

解决办法：放开之前的注释，也就是加上channel.basicQos(1)。同一时刻服务器只会发一条消息给消费者，每一次服务器只会向客户端发送一条。测试可以看到实现能者多劳，多劳多得。

## 发布订阅publish/subscribe

上面是单消息队列，下面有两个队列。

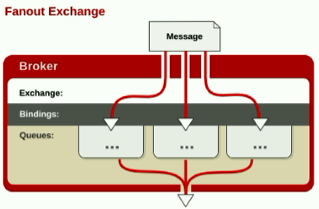


生产者将消息不是直接发送到队列，而是发送到X交换机，然后由交换机发送给两个队列，两个消费者各自监听一个队列，来消费消息。

这个X交换机就类似QQ群一样。老师发送一个作业题，大家都可以从这里获取。

这种方式实现同一个消息被多个消费者消费。工作模式是同一个消息只能有一个消费者。

### Fanout Exchange



一个发送到交换机的消息都会被转发到与该交换机绑定的所有队列上。很像子网广播，每台子网内的主机都获得了一份复制的消息。Fanout交换机转发消息是最快的。

### 发送消息

package com.rabbitmq.ps;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.util.ConnectionUtil;

public class Send {

private final static String EXCHANGE\_NAME = "jt\_test\_exchange\_fanout";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明exchange

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

// 消息内容

String message = "Hello World!";

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

System.out.println(" [x] Sent '" + message + "'");

channel.close();

connection.close();

}

}

注意：如果没有队列绑定到交换机，那么发送到交换机的消息将丢失。

### 消费者1

package com.rabbitmq.ps;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv {

private final static String QUEUE\_NAME = "test\_queue\_exchange";

private final static String EXCHANGE\_NAME = "test\_exchange\_fanout";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 绑定队列到交换机

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "");

// 同一时刻服务器只会发一条消息给消费者

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

Thread.sleep(10);

// 返回消息消费状态

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

}

}

}

### 消费者2

package com.rabbitmq.ps;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv2 {

private final static String QUEUE\_NAME = "test\_queue\_exchange2";

private final static String EXCHANGE\_NAME = "test\_exchange\_fanout";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 绑定队列到交换机

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "");

// 同一时刻服务器只会发一条消息给消费者

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

Thread.sleep(10);

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

}

}

}

注意：

1）消费者1、消费者2都必须绑定到交换机上，channel.queueBind。否则无法获取消息。

2）消息消费标示

在工作模式中消费完消息会返回消息消费状态为true。

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

在发布订阅时，不是立即返回消息消费状态，而是手工设置返回消息消费状态

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

……

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

例如：当获取到消息时，开始处理业务，这时有台计算机突然宕机了，此时这个消息并没有消费完。实际应当等业务逻辑处理完，在个服务端发出一个消息消费完表示。

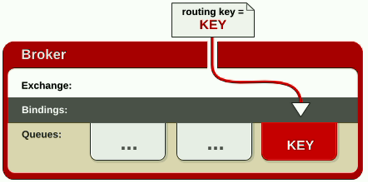
这里是在开发时要根据具体的应用场景选择不同的学习方法。

问题：能否选择性的接收消息。

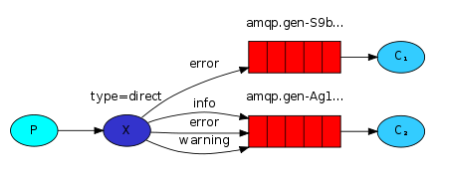
例如新增消息，对于前台系统就不需要，但对于搜寻系统就需要。那这样的情况下，是否能需要的信息才接收，不需要的不管。答案当然可以有。这就是路由模式。

## \*路由模式routing

### Direct Exchange



需要将一个队列绑定到交换机上，要求该消息与一个特定的路由键完全匹配，这是一个完整的匹配。如果一个队列绑定到该交换机上要求路由键“cow”，则只有被标记为“cow”的消息才能被转发，不会转发cat，也不会转发mouse，只会转发cow。



对于路由的这种结构，我们就可以将它应用在我们的项目中，前台系统只接收更新的消息，而检索可以接收新增和更新的消息。

### 发送消息

package com.rabbitmq.routing;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.util.ConnectionUtil;

public class Send {

private final static String EXCHANGE\_NAME = "test\_exchange\_direct";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明exchange

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

// 消息内容

String message = "Hello World!";

channel.basicPublish(EXCHANGE\_NAME, "key2", null, message.getBytes());

System.out.println(" [x] Sent '" + message + "'");

channel.close();

connection.close();

}

}

### 消费者1

package com.rabbitmq.routing;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv {

private final static String QUEUE\_NAME = "test\_queue\_direct\_work";

private final static String EXCHANGE\_NAME = "test\_exchange\_direct";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 绑定队列到交换机

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "key");

// 同一时刻服务器只会发一条消息给消费者

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

Thread.sleep(10);

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

}

}

}

### 消费者2

package com.rabbitmq.routing;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv2 {

private final static String QUEUE\_NAME = "test\_queue\_direct\_work2";

private final static String EXCHANGE\_NAME = "test\_exchange\_direct";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 绑定队列到交换机

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "key2");

// 同一时刻服务器只会发一条消息给消费者

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

Thread.sleep(10);

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

}

}

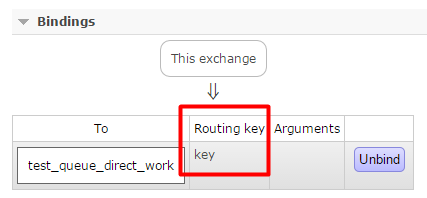
}

注意：在绑定时，设置路由KEY。

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "key2");

### 绑定结果

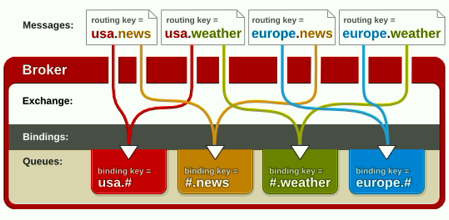
可以看到定义了一个路由KEY。



## 主题模式Topic

### Topic Exchange

将路由键和某模式进行匹配。此时队列需要绑定在一个模式上。符合#匹配一个或多个词，符号\*匹配不多不少一个词。



### 发送消息

package com.rabbitmq.topic;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.util.ConnectionUtil;

public class Send {

private final static String EXCHANGE\_NAME = "test\_exchange\_topic";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明exchange

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

// 消息内容

String message = "Hello World! topic";

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

System.out.println(" [x] Sent '" + message + "'");

channel.close();

connection.close();

}

}

### 接收消息

package com.rabbitmq.topic;

import com.rabbitmq.client.Channel;

import com.rabbitmq.client.Connection;

import com.rabbitmq.client.QueueingConsumer;

import com.rabbitmq.util.ConnectionUtil;

public class Recv {

private final static String QUEUE\_NAME = "test\_queue\_topic\_work";

private final static String EXCHANGE\_NAME = "test\_exchange\_topic";

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

// 获取到连接以及mq通道

Connection connection = ConnectionUtil.getConnection();

Channel channel = connection.createChannel();

// 声明队列

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

// 绑定队列到交换机

channel.queueBind(QUEUE\_NAME, EXCHANGE\_NAME, "item.#");

// 同一时刻服务器只会发一条消息给消费者

channel.basicQos(1);

// 定义队列的消费者

QueueingConsumer consumer = new QueueingConsumer(channel);

// 监听队列，手动返回完成

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

// 获取消息

while (true) {

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String message = new String(delivery.getBody());

System.out.println(" [x] Received '" + message + "'");

Thread.sleep(10);

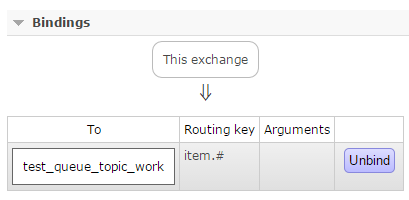
channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

}

}

}

### 绑定结果



## 消息确认应答Ack

### 自动模式

String com.rabbitmq.client.Channel.basicConsume(String queueName, boolean autoAck, Consumer arg2) throws IOException

第二个参数autoAck，是否自动应答。

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

### 手动模式

手工模式的优点是业务真正执行完，消息队列才会把消息删除。防止RabbitMQ已经把消息删除。

//设置手工应答

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

……

//人工进行确认

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

### 案例

接收者2故意抛出异常

int j = 1/0; 先接收消息，然后业务抛出异常，只要没有进行basicAck，就不会删除消息。业务正常后，可以再次获取到这个消息。

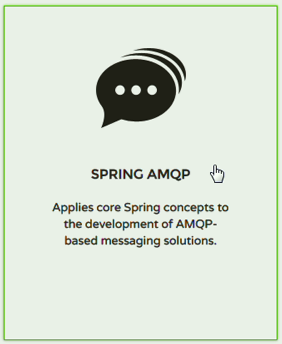
## \*几种模式的使用场景

|  |  |  |
| --- | --- | --- |
| **序号** | **工作模式** | **说明** |
| 1 | 简单模式：单发送单接收  http://images.cnitblog.com/i/434101/201408/171507104838529.png | 简单的发送与接收，没有特别的处理。 |
| 2 | 工作模式：单发送多接收（竞争）  http://images.cnitblog.com/i/434101/201408/171513043112193.png | 一个发送端，多个接收端，如分布式的任务派发。为了保证消息发送的可靠性，不丢失消息，使消息持久化了。同时为了防止接收端在处理消息时down掉，只有在消息处理完成后才发送ack消息。 |
| 3 | 发布订阅模式：Publish/Subscribe  http://images.cnitblog.com/i/434101/201408/171657207955618.png | 发布、订阅模式，发送端发送广播消息，多个接收端接收。 |
| \*4 | 路由模式：Routing (按路线发送接收)  http://images.cnitblog.com/i/434101/201408/171709256231158.png | 发送端按routing key发送消息，不同的接收端按不同的routing key接收消息。  在实际项目中普遍采用Direct路由方式 |
| 5 | 主题模式：Topics (按topic发送接收)  http://images.cnitblog.com/i/434101/201408/171739210926049.png | 发送端不只按固定的routing key发送消息，而是按字符串“匹配”发送，接收端同样如此。 |

## Spring-AMQP

Spring对AMQP做了支持，目前只是做了RabbitMQ的实现

官网：http://projects.spring.io/spring-amqp



“The project consists of two parts; spring-amqp is the base abstraction, and spring-rabbit is the [RabbitMQ](http://www.rabbitmq.com/) implementation.”

可以看到SpringAMQP官网上声明它只是做了[RabbitMQ](http://www.rabbitmq.com/) 的实现。

### 启动服务

com.rabbitmq.spring.SpringMain

package com.rabbitmq.spring;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.context.support.AbstractApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringMain {

public static void main(final String... args) throws Exception {

AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(

"classpath:spring/rabbitmq-context.xml");

//RabbitMQ模板

RabbitTemplate template = ctx.getBean(RabbitTemplate.class);

//发送消息

template.convertAndSend("Hello, world!");

Thread.sleep(10000);// 休眠1秒

ctx.destroy(); //容器销毁

}

}

### 业务类

com.rabbitmq.spring.Foo

package com.rabbitmq.spring;

/\*\*

\* 消费者

\*/

public class Foo {

//具体执行业务的方法

public void listen(String foo) {

System.out.println("消费者： " + foo);

}

}

### 配置文件

/demo-rabbitmq/src/main/resources/spring/rabbitmq-context.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:rabbit="http://www.springframework.org/schema/rabbit"

xsi:schemaLocation="http://www.springframework.org/schema/rabbit

http://www.springframework.org/schema/rabbit/spring-rabbit-1.4.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<!-- 定义RabbitMQ的连接工厂 -->

<rabbit:connection-factory id="connectionFactory"

host="127.0.0.1" port="5672" username="sysdebug" password="123456"

virtual-host="/jt" />

<!-- 定义Rabbit模板，指定连接工厂以及定义exchange -->

<rabbit:template id="amqpTemplate" connection-factory="connectionFactory" exchange="fanoutExchange"/>

<!-- <rabbit:template id="amqpTemplate" connection-factory="connectionFactory"

exchange="fanoutExchange" routing-key="foo.bar" /> -->

<!-- MQ的管理，包括队列、交换器等 -->

<rabbit:admin connection-factory="connectionFactory" />

<!-- 定义队列，自动声明 -->

<rabbit:queue name="myQueue" auto-declare="true" durable="true"/>

<!-- 定义交换器，自动声明 -->

<rabbit:fanout-exchange name="fanoutExchange" auto-declare="true">

<!-- <rabbit:bindings>

<rabbit:binding queue="myQueue"/>

</rabbit:bindings> -->

</rabbit:fanout-exchange>

<!-- <rabbit:topic-exchange name="myExchange">

<rabbit:bindings>

<rabbit:binding queue="myQueue" pattern="foo.\*" />

</rabbit:bindings>

</rabbit:topic-exchange> -->

<!-- 队列监听 -->

<rabbit:listener-container connection-factory="connectionFactory">

<rabbit:listener ref="foo" method="listen" queue-names="myQueue" />

</rabbit:listener-container>

<bean id="foo" class="com.rabbitmq.spring.Foo" />

</beans>

### 队列持久化

非持久化时，当rabbitmq宕机时，消息是丢失的。持久化时，会将数据写入磁盘，宕机重启后，会恢复这些消息。非持久化信息保存到内存，持久化信息保存到磁盘。

durable默认为true。默认持久化。

<!-- 定义队列，自动声明，durable是否持久化 -->

<rabbit:queue name="myQueue" auto-declare="true" durable="true"/>



## Spring-RabbitMQ集成

### 后台系统-增加MQ依赖

/jt-manage/pom.xml

<!-- RabbitMQ -->

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

<version>1.4.5.RELEASE</version>

</dependency>

### 后台系统-和spring框架整合

/jt-manage-web/src/main/resources/spring/applicationContext-rabbitmq.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:rabbit="http://www.springframework.org/schema/rabbit"

xsi:schemaLocation="http://www.springframework.org/schema/rabbit

http://www.springframework.org/schema/rabbit/spring-rabbit-1.4.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<!-- 定义RabbitMQ的连接工厂 -->

<rabbit:connection-factory id="connectionFactory"

host="${rabbit.ip}" port="${rabbit.port}" username="${rabbit.username}" password="${rabbit.password}"

virtual-host="${rabbit.vhost}" />

<!-- 定义Rabbit模板，指定连接工厂以及定义exchange -->

<rabbit:template id="amqpTemplate" connection-factory="connectionFactory"

exchange="itemDirectExchange" />

<!-- MQ的管理，包括队列、交换器等 -->

<rabbit:admin connection-factory="connectionFactory" />

<!-- 定义交换器，自动声明,持久化 -->

<rabbit:direct-exchange name="itemDirectExchange" auto-declare="true" durable="true">

</rabbit:direct-exchange>

</beans>

/jt-manage-web/src/main/resources/rabbitmq.properties

rabbit.ip=127.0.0.1

rabbit.port=5672

rabbit.username=jtmqadmin

rabbit.password=123456

rabbit.vhost=jtmq

/jt-manage-web/src/main/resources/spring/applicationContext.xml

<!-- 配置资源文件 -->

<property name="locations">

<list>

<value>classpath:jdbc.properties</value>

<value>classpath:env.properties</value>

<value>classpath:redis.properties</value>

<value>classpath:httpclient.properties</value>

<value>classpath:rabbitmq.properties</value>

</list>

</property>

### 后台系统-商品更新发送消息

com.jt.manage.service.ItemService.updateItem(Item, String, String)

@Autowired

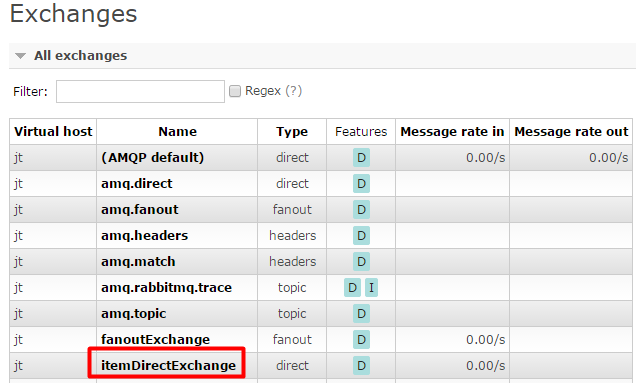
private RabbitTemplate rabbitTemplate;

……

//发送更新消息到RabbitMQ中

rabbitTemplate.convertAndSend("jt\_item\_update", item.getId());

这里发送消息时，可以发送id信息，更新了哪个货物的信息，也可以将整个货物信息发送，也可以发送json，那应该选择什么样的内容呢？首先，我们应该有一个意识，发的内容越少越好。第二，根据具体需求。那如果要货物的全部信息呢？也可以只发送id，然后通过后台提供的接口查询。



控制台日志：

2015-06-11 09:42:37,647 [http-bio-8081-exec-2] [org.springframework.amqp.rabbit.core.RabbitAdmin]-[DEBUG] declaring Exchange 'itemDirectExchange'

2015-06-11 09:42:37,703 [http-bio-8081-exec-2] [org.springframework.amqp.rabbit.core.RabbitAdmin]-[DEBUG] Declarations finished

### 后台系统-商品新增发送消息

com.jt.manage.service.ItemService.saveItem(Item, String, String)

this.rabbitTemplate.convertAndSend("jt\_item\_insert", item.getId());

### 前台系统-增加MQ依赖

/jt-manage-web/pom.xml

<!-- RabbitMQ -->

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

<version>1.4.5.RELEASE</version>

</dependency>

### 前台系统-和spring框架整合

/jt-web/src/main/resources/spring/applicationContext-rabbitmq.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:rabbit="http://www.springframework.org/schema/rabbit"

xsi:schemaLocation="http://www.springframework.org/schema/rabbit

http://www.springframework.org/schema/rabbit/spring-rabbit-1.4.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<!-- 定义RabbitMQ的连接工厂 -->

<rabbit:connection-factory id="connectionFactory"

host="${rabbit.ip}" port="${rabbit.port}" username="${rabbit.username}" password="${rabbit.password}"

virtual-host="${rabbit.vhost}" />

<!-- MQ的管理，包括队列、交换器等 -->

<rabbit:admin connection-factory="connectionFactory" />

<!-- 定义消息队列 -->

<rabbit:queue name="jt-web.itemQueue" auto-declare="true"/>

<!-- 定义交换机，并且完成队列和交换机的绑定 -->

<rabbit:direct-exchange name="itemDirectExchange" auto-declare="true">

<rabbit:bindings>

<!-- 前台系统只接收商品更新的消息 -->

<rabbit:binding queue="jt-web.itemQueue" key="update"/>

</rabbit:bindings>

</rabbit:direct-exchange>

<!-- 定义监听 -->

<rabbit:listener-container connection-factory="connectionFactory">

<rabbit:listener ref="rabbitItemService" method="updateItem" queue-names="jt-web.itemQueue"/>

</rabbit:listener-container>

</beans>

/jt-web/src/main/resources/rabbitmq.properties

rabbit.ip=127.0.0.1

rabbit.port=5672

rabbit.username=jtmqadmin

rabbit.password=123456

rabbit.vhost=jtmq

和后台系统一致。

/jt-web/src/main/resources/spring/applicationContext.xml

<!-- 配置资源文件 -->

<property name="locations">

<list>

<value>classpath:env.properties</value>

<value>classpath:httpclient.properties</value>

<value>classpath:redis.properties</value>

<value>classpath:rabbitmq.properties</value>

</list>

</property>

### 前台系统-接收商品更新消息

com.jt.web.service.RabbitItemService

package com.jt.web.service;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

@Component

public class RabbitItemService {

private static final Logger LOGGER = LoggerFactory.getLogger(RabbitItemService.class);

@Autowired

private ItemService itemService;

public void updateItem(Long itemId) {

LOGGER.info("接受到MQ的消息，内容为：{}", itemId);

this.itemService.updateRedis(itemId);

}

}

在applicationContext-rabbitmq.xml中就定义了监听的消息队列，当有消息时，会根据配置调用RabbitItemService的updataItem方法。并将参数传入。

<rabbit:listener-container connection-factory="connectionFactory">

<rabbit:listener ref="rabbitItemService" method="updateItem" queue-names="jt-web.itemQueue"/>

</rabbit:listener-container>

注意：和后台系统配置时的区别，后台中并未绑定消息队列到交换机上，这里根据key进行了绑定。并通过指定的service的方法来侦听消息。

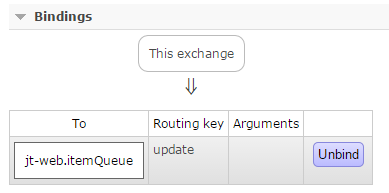
启动后台系统，启动前台系统，控制台日志：

2015-06-11 10:21:38,902 [SimpleAsyncTaskExecutor-1] [org.springframework.amqp.rabbit.listener.BlockingQueueConsumer]-[DEBUG] Retrieving delivery for Consumer: tags=[[amq.ctag-RCY-gP7bwTM1xeXb2CNAIA]], channel=Cached Rabbit Channel: AMQChannel(amqp://jtmqadmin@127.0.0.1:5672jtmq,1), acknowledgeMode=AUTO local queue size=0

2015-06-11 10:21:39,904 [SimpleAsyncTaskExecutor-1] [org.springframework.amqp.rabbit.listener.BlockingQueueConsumer]-[DEBUG] Retrieving delivery for Consumer: tags=[[amq.ctag-RCY-gP7bwTM1xeXb2CNAIA]], channel=Cached Rabbit Channel: AMQChannel(amqp://jtmqadmin@127.0.0.1:5672jtmq,1), acknowledgeMode=AUTO local queue size=0

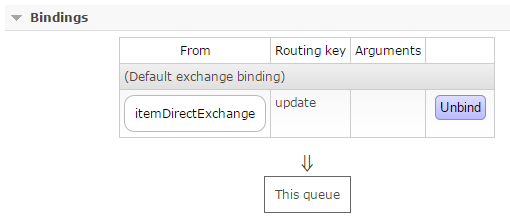
可以看到前台系统已经开始监听消息队列。

查看Exchanges：



可以看到itemDirectExchange交换机绑定了路由为update的jt-web.itemQueue消息。

查看Queues：



可以看到jt-web.itemQueue消息绑定了路由为update的itemDirectExchange交换机。

修改信息，查看控制台日志：

2015-06-11 10:34:07,489 [pool-1-thread-4] [org.springframework.amqp.rabbit.listener.BlockingQueueConsumer]-[DEBUG] Storing delivery for Consumer: tags=[[amq.ctag-RCY-gP7bwTM1xeXb2CNAIA]], channel=Cached Rabbit Channel: AMQChannel(amqp://jtmqadmin@127.0.0.1:5672jt,1), acknowledgeMode=AUTO local queue size=0

2015-06-11 10:34:07,591 [SimpleAsyncTaskExecutor-1] [org.springframework.amqp.rabbit.listener.BlockingQueueConsumer]-[DEBUG] Received message: (Body:'1474391935'MessageProperties [headers={}, timestamp=null, messageId=null, userId=null, appId=null, clusterId=null, type=null, correlationId=null, replyTo=null, contentType=application/x-java-serialized-object, contentEncoding=null, contentLength=0, deliveryMode=PERSISTENT, expiration=null, priority=0, redelivered=false, receivedExchange=itemDirectExchange, receivedRoutingKey=update, deliveryTag=1, messageCount=0])

2015-06-11 10:34:07,602 [SimpleAsyncTaskExecutor-1] [com.jt.web.service.RabbitItemService]-[INFO] 接受到MQ的消息，内容为：1474391935

监听已经收到消息，并得到更新货物的id=1474391935。

### 搜索系统-增加依赖

/jt-search/pom.xml

<!-- RabbitMQ -->

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

<version>1.4.5.RELEASE</version>

</dependency>

### 搜索系统-和spring框架整合

/jt-search/src/main/resources/rabbitmq.properties

rabbit.ip=127.0.0.1

rabbit.port=5672

rabbit.username=sysdebug

rabbit.password=123456

rabbit.vhost=jt

/jt-search/src/main/resources/spring/applicationContext-rabbitmq.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:rabbit="http://www.springframework.org/schema/rabbit"

xsi:schemaLocation="http://www.springframework.org/schema/rabbit

http://www.springframework.org/schema/rabbit/spring-rabbit-1.4.xsd

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.1.xsd">

<!-- 定义RabbitMQ的连接工厂 -->

<rabbit:connection-factory id="connectionFactory"

host="${rabbit.ip}" port="${rabbit.port}" username="${rabbit.username}" password="${rabbit.password}"

virtual-host="${rabbit.vhost}" />

<!-- MQ的管理，包括队列、交换器等 -->

<rabbit:admin connection-factory="connectionFactory" />

<!-- 定义消息队列 -->

<rabbit:queue name="jt-search.itemQueue" auto-declare="true"/>

<!-- 定义交换机，并且完成队列和交换机的绑定 -->

<rabbit:direct-exchange name="itemDirectExchange" auto-declare="true">

<rabbit:bindings>

<rabbit:binding queue="jt-search.itemQueue" key="insert"/>

<rabbit:binding queue="jt-search.itemQueue" key="update"/>

</rabbit:bindings>

</rabbit:direct-exchange>

<!-- 定义监听 -->

<rabbit:listener-container connection-factory="connectionFactory">

<rabbit:listener ref="rabbitItemService" method="saveOrUpdateItem" queue-names="jt-search.itemQueue"/>

</rabbit:listener-container>

</beans>

/jt-search/src/main/resources/spring/applicationContext.xml

<!-- 配置资源文件 -->

<property name="locations">

<list>

<value>classpath:env.properties</value>

<value>classpath:httpclient.properties</value>

<value>classpath:solr.properties</value>

<value>classpath:rabbitmq.properties</value>

</list>

</property>

### 搜索系统-接收商品新增和修改消息

com.jt.search.service.RabbitItemService

package com.jt.search.service;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import com.jt.common.service.HttpClientService;

import com.jt.common.spring.exetend.PropertyConfig;

import com.jt.common.vo.SysResult;

import com.jt.search.pojo.Item;

@Component

public class RabbitItemService {

private static final Logger LOGGER = LoggerFactory.getLogger(RabbitItemService.class);

@Autowired

private SearchService searchService;

@Autowired

private HttpClientService apiService;

@PropertyConfig

private String MANAGE\_URL;

public void saveOrUpdateItem(Long itemId) {

LOGGER.info("接受到消息，内容为: {}", itemId);

Item item = this.getItemFromApi(itemId);

if (null != item) {

this.searchService.update(item);

}

}

private Item getItemFromApi(Long itemId) {

try {

String url = MANAGE\_URL + "/item/query/" + itemId;

String jsonData = this.apiService.doGet(url);

SysResult sysResult = SysResult.formatToPojo(jsonData, Item.class);

return (Item) sysResult.getData();

} catch (Exception e) {

LOGGER.error("更新Solr数据出错! itemId = " + itemId, e);

}

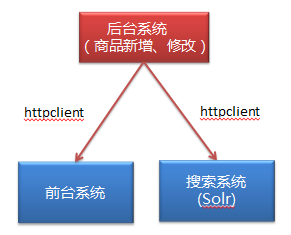
return null;

}

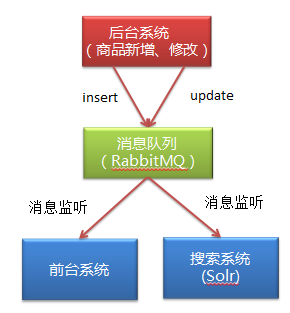
}

### 方案对比

传统方案：



改进方案：



传统方案紧密耦合，维护难度大，改进方案做到两个系统松耦合。松耦合系统的优点在于更新一个模块不会引起其它模块的改变。例如：现在又要增加一个系统，原有系统必须加代码逻辑实现，而新方案只需新的系统创建队列，绑定到交换机中，进行监听即可。显然新的方案更加方便系统的升级和维护。

第二个方案的缺点：

1）RabbitMQ的高可用性保证优先级最高（运维）。一旦RabbitMQ崩溃，那各系统之间就无法进行数据更新。

解决办法：集群，主备。

1. 如果消息发送失败，会导致数据不同步。

### 问题：Failed to check/redeclare auto-delete queue(s).

启动RabbitMQ后会在RabbitServer创建一个消息队列。而如果已经存在则上面错误，到RabbitServer将消息队列删除即可。

### 问题：如果消息发送失败，应该怎么解决？

首先进行重试，重试三次，第一次停顿1秒，第二次停顿5秒，第三次停顿20秒。停顿间隔根据具体情况进行调整。第二，通知相关负责人进行处理，发邮件，发短信。第三，记录日志。第四，将发送失败的数据保存下来，并且有错误回复机制。如将失败的数据写入数据库，写专门的程序从数据库中读取数据重新发送。

## RabbitMQ与Redis队列对比

RabbitMQ是实现AMQP（高级消息队列协议）的消息中间件的一种，最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

Redis是一个Key-Value的NoSQL数据库，开发维护很活跃，虽然它是一个Key-Value数据库存储系统，但它本身支持MQ功能，所以完全可以当做一个轻量级的队列服务来使用。

### 具体对比

|  |  |  |  |
| --- | --- | --- | --- |
|  | 对比项 | Redis | RabbitMQ |
| 1 | 可靠消费 | 没有相应的机制保证消息的消费，当消费者消费失败的时候，消息体丢失，需要手动处理 | 具有消息消费确认，即使消费者消费失败，也会自动使消息体返回原队列，同时可全程持久化，保证消息体被正确消费 |
| 2 | 可靠发布 | 不提供，需自行实现 | 具有发布确认功能，保证消息被发布到服务器 |
| 3 | 高可用 | 采用主从模式，读写分离，但是故障转移还没有非常完善的官方解决方案 | 集群采用磁盘、内存节点，任意单点故障都不会影响整个队列的操作 |
| 4 | 持久化 | 将整个Redis实例持久化到磁盘 | 队列，消息，都可以选择是否持久化 |
| 5 | 消费者负载均衡 | 不提供，需自行实现 | 根据消费者情况，进行消息的均衡分发 |
| 6 | 队列监控 | 不提供，需自行实现 | 后台可以监控某个队列的所有信息，（内存，磁盘，消费者，生产者，速率等） |
| 7 | 流量控制 | 不提供，需自行实现 | 服务器过载的情况，对生产者速率会进行限制，保证服务可靠性 |
| 8 | 应用场景分析 | 轻量级，高并发，延迟敏感  即时数据分析、秒杀计数器、缓存等 | 重量级，高并发，异步  批量数据异步处理、并行任务串行化，高负载任务的负载均衡等 |
|  | 出入队性能 | 对于RabbitMQ和Redis的入队和出队操作，各执行100万次，每10万次记录一次执行时间。  测试数据分为128Bytes、512Bytes、1K和10K四个不同大小的数据。 | |
|  | http://static.oschina.net/uploads/img/201505/20100528_VENS.png | | |

### 应用场景分析

Redis：轻量级，高并发，延迟敏感

即时数据分析、秒杀计数器、缓存等

RabbitMQ：重量级，高并发，异步

批量数据异步处理、并行任务串行化，高负载任务的负载均衡等

### 面试：rabbitmq的优先级做法

目前成熟的消息队列产品有很多，著名的例如rabbitmq。它使用起来相对还是比较简单的，功能也相对比较丰富，一般场合下是完全够用的。但是有个很烦人的就是它不支持优先级。

例如一个发邮件的任务，某些特权用户希望它的邮件能够更加及时的发送出去，至少比普通用户要优先对待。默认情况下rabbitmq是无法处理掉的，扔给rabbitmq的任务都是FIFO先进先出。但是我们可以使用一些变通的技巧来支持这些优先级。创建多个队列，并为rabbitmq的消费者设置相应的路由规则。

例如默认情况下有这样一个队列，我们拿list来模拟 [task1, task2, task3]，消费者轮流按照FIFO的原则一个个拿出task来处理掉。如果有高优先级的任务进来，它也只能跟在最后被处理[task1, task2, task3, higitask1]. 但是如果使用两个队列，一个高优先级队列，一个普通优先级队列。 普通优先级[task1, task2, task3], 高优先级[hightask1 ] 然后我们设置消费者的路由让消费者随机从任意队列中取数据即可。

并且我们可以定义一个专门处理高优先级队列的消费者，它空闲的时候也不处理低优先级队列的数据。这类似银行的VIP柜台，普通客户在银行取号排队，一个VIP来了他虽然没有从取号机里拿出一个排在普通会员前面的票，但是他还是可以更快地直接走VIP通道。