# 关于Dubbo使用

## 关于名称的定义

说明:定义Dubbo的名称,应该根据接口的名称进行定义.

不同的接口使用不同的应用名称.一个接口的多个实现类,使用相同的应用名称.否则会报警告错误



# 消息队列

## 消息队列作用

### 问题分析

1. 用户的请求的数量要远高于数据库处理能力
2. 用户要求现在马上执行.如果超时客户端报错(同步)

### 消息队列

MQ全称为Message Queue, [消息队列](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97)（MQ）是一种应用程序对应用程序的通信方法。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无需专用连接来链接它们。消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于诸如[远程过程调用](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8)的技术。排队指的是应用程序通过 队列来通信。队列的使用除去了接收和发送应用程序同时执行的要求。其中较为成熟的MQ产品有IBM WEBSPHERE MQ等等。

总结:消息队列,是一端写入数据到队列中,一端从队列中获取消息,并且通信通过RPC调用,可以传递java对象,方便操作.

### 消息队列作用

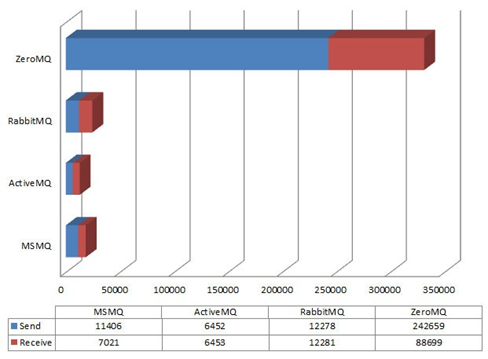
作用:消息队列主要的作用是**缓解服务器压力**.

说明:用户将请求先写入消息队列中.之后后台服务器器从消息队列中获取消息之后处理.实现请求的异步的操作.

特点:平峰削谷

缺点:消耗的是用户等待的时间

### 消息队列的产品



1. 阿里巴巴 rocketmq 业务消息队列
2. Apache Kafka 适用于海量的数据收集 大数据中 50万/s

### RabbitMQ介绍

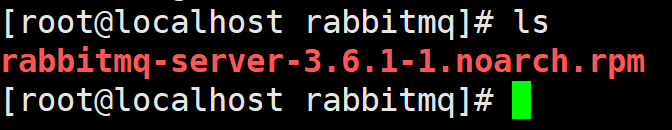
是电信具体公司爱立信公司,为银行业务负责研发的消息队列.

Erlang语言:

Erlang(['ə:læŋ])是一种通用的面向[并发](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91" \t "_blank)的编程语言，它由[瑞典](https://baike.baidu.com/item/%E7%91%9E%E5%85%B8)电信设备制造商爱立信所辖的CS-Lab开发，目的是创造一种可以应对大规模并发活动的编程语言和运行环境。Erlang问世于1987年，经过十年的发展，于1998年发布开源版本。Erlang是运行于[虚拟机](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA/104440" \t "_blank)的[解释性语言](https://baike.baidu.com/item/%E8%A7%A3%E9%87%8A%E6%80%A7%E8%AF%AD%E8%A8%80/4665504)，但是现在也包含有[乌普萨拉大学](https://baike.baidu.com/item/%E4%B9%8C%E6%99%AE%E8%90%A8%E6%8B%89%E5%A4%A7%E5%AD%A6/8472305)高性能Erlang计划（HiPE）开发的[本地代码](https://baike.baidu.com/item/%E6%9C%AC%E5%9C%B0%E4%BB%A3%E7%A0%81/860403" \t "_blank)[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8/8853067)，自R11B-4版本开始，Erlang也开始支持脚本式解释器。在[编程范型](https://baike.baidu.com/item/%E7%BC%96%E7%A8%8B%E8%8C%83%E5%9E%8B/1475451" \t "_blank)上，Erlang属于多重范型编程语言，涵盖函数式、并发式及[分布式](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F/19276232)。顺序执行的Erlang是一个及早求值, 单次赋值和动态类型的函数式编程语言。

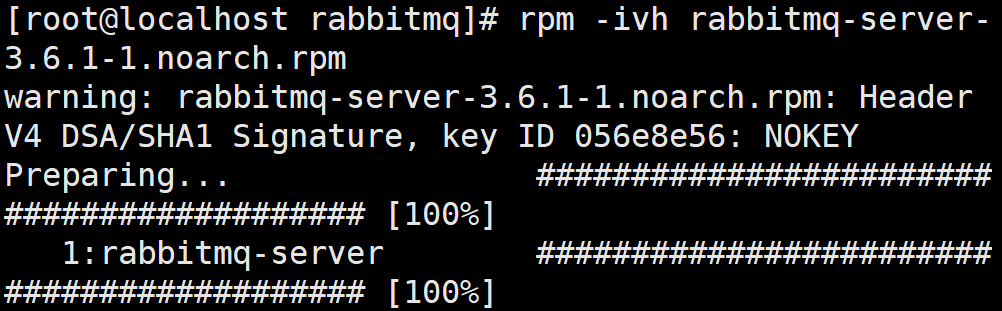
## RabbitMQ安装

### 上传文件



### 安装rabbitMQ

rpm -ivh rabbitmq-server-3.6.1-1.noarch.rpm



### 修改配置文件

将文件复制到指定目录下:

cp /usr/share/doc/rabbitmq-server-3.6.1/rabbitmq.config.example /etc/rabbitmq/rabbitmq.config

**修改新复制的文件64行**

计算机生成了可选文字:
Vla
。loopbackInterface(e.g.localhost).
{100pbaCk--USerS,[<<"guest''>>]},
UncommentthefollowinglineIfyouwonttoallo
g材习石4升心石习石司肺习用es口林丫树杯习FeonthenetWOFk.
%％小％%%%
%%{loopbac悦users,[l},
默认条件下是关系的

1.将%%去掉

2.将,号去掉

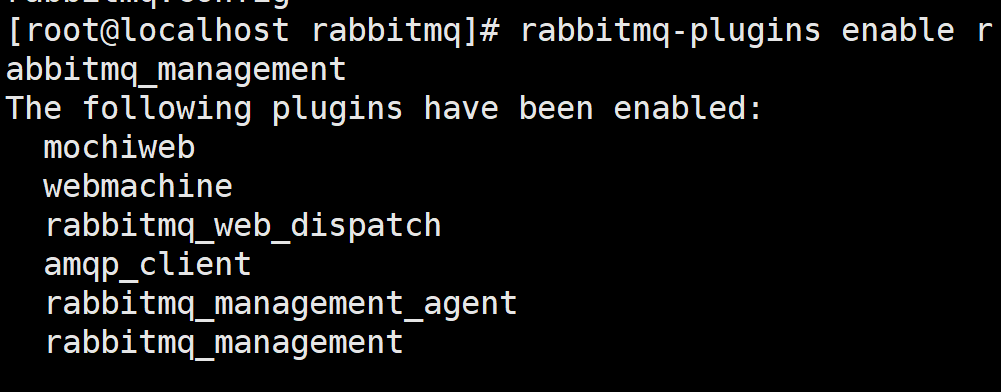
修改为:

计算机生成了可选文字:
VlddLUUpUdL民
{10opback--users
111LelldLe气e·g·
LU仁dLllUSL）。
【＜<',guest"'>>]}
UnC0mmentthe
{loopback--users
followlng11ne
黔”he「e"n
Ifyouwanttoallowacce悠
thenetWOFk.
万％、％生
万％%％怪
.

### 安装rabbitMQ

执行命令:

rabbitmq-plugins enable rabbitmq\_management



### 启动rabbitMQ

service rabbitmq-server start 启动

service rabbitmq-server stop 停止

service rabbitmq-server restart 重启

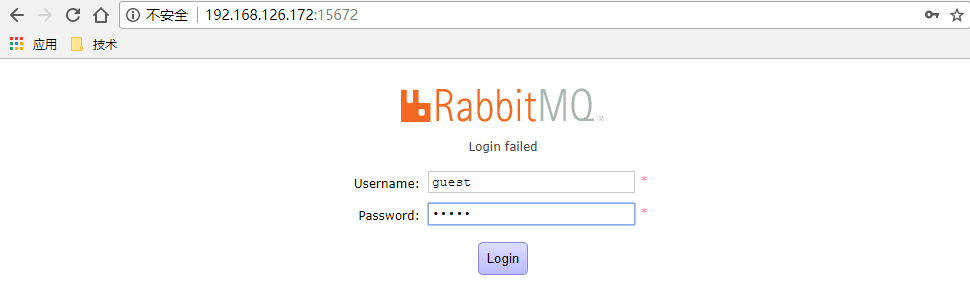
计算机生成了可选文字:
[rootOlocalhost
rabbitmql逆
SeFVICe
rabbltmq一serverstart
Startlngrabbltmq一server:
rabbltmq一server.
[rootOlo。alhostrabbitmql#.

### 端口介绍

1. rabbitMQ监控系统的端口:15672
2. rabbitMQ服务调用端口:5672

### 登录

用户名和密码:guest



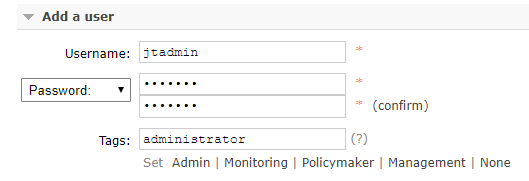
## 消息队列监控介绍



Exchanges: 交换机 作用:将消息发往不同的队列中

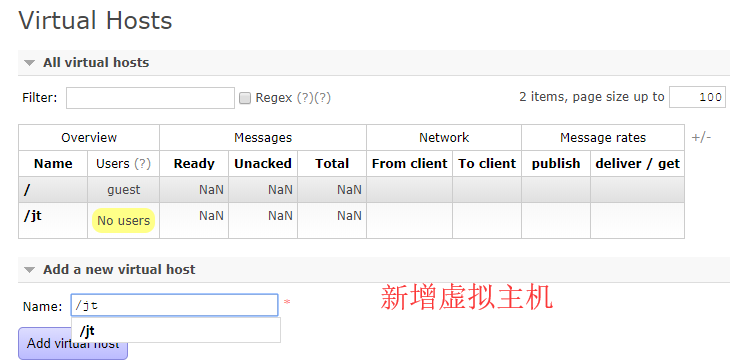
Queues:表示消息队列 存储数据的位置是内存.

### 新建用户

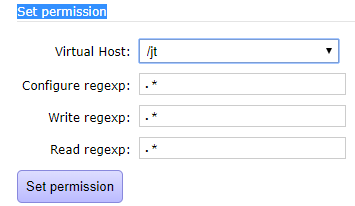


|  |  |  |
| --- | --- | --- |
| **序号** | **角色名称** | **说明** |
| 1. | 超级管理员（administrator） | 可登陆管理控制台，可查看所有的信息，并且可以对用户，策略（policy）进行操作。 |
| 2. | 监控者（monitoring） | 可登陆管理控制台，同时可以查看rabbingmq节点的相关信息（进程数，内存使用情况，磁盘使用情况等） |
| 3. | 策略制定者（policymaker） | 可登陆管理控制台，同时可以对policy进行管理。但无法查看节点的相关信息 |
| 4. | 普通管理者（management） | 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。 |

### 新增虚拟主机



### 用户关联虚拟主机



# 消息队列模式

## 简单模式

### 定义



P:消息的生产者

红色部分:代表队列

C:代表消费者 从队列中获取消息后执行.

工作原理:

当客户端(生产者)将消息写入消息队列中时,消息队列中信息的数量加1.

消费者**实时监听**消息队列,当消息队列中有消息时,则获取消息,之后执行业务逻辑.同时消息队列的数量减一

### 导入jar包

<dependencies>

<!-- 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>

</dependencies>

### 初始化链接

/\*\*

\* 1.初始化链接

\* 链接rabbitMQ步骤:

\* 定义主机地址IP/5672/用户名和密码/虚拟机主机名称

\* **@throws** IOException

\*/

@Before //表示在执行@Test注解之前执行该操作

**public** **void** init() **throws** IOException{

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("192.168.126.172");

factory.setPort(5672);

factory.setUsername("jtadmin");

factory.setPassword("jtadmin");

factory.setVirtualHost("/jt");

connection = factory.newConnection();

}

### 定义生产者

/\*\*

\* 1.定义通道

\* 主要作用:创建队列/消费者/发布消息

\* **@throws** IOException

\*/

@Test

**public** **void** provader() **throws** IOException{

//1.定义通道

Channel channel = connection.createChannel();

//2.定义消息队列

String queueName = "simple";

/\*\*

\* queue 队列的名称

\* durable 是否持久化 true/false

\* exclusive 是否是生成者独有 false

\* autoDelete 是否自动删除 当队列中的消息处理完成后,队列是否删除

\* arguments 是否需要传递其他的参数 null

\*/

channel.queueDeclare(queueName, **true**, **false**, **false**, **null**);

//3.定义队列的消息 rpc 默认使用utf-8编码

String msg = "我是简单模式";

//4.发送消息

/\*\*

\* exchange : 交换机名称 如果没有交换机 则为""串

\* routingKey : 路由key 如果没有理由key则使用队列的名称代替

\* props : 表示消息队列的配置文件

\* body : 消息内容的二进制数组

\*/

channel.basicPublish("", queueName, **null**, msg.getBytes());

channel.close();

connection.close();

System.***out***.println("消息队列执行成功!!!!");

}

### 定义消费者

@Test

**public** **void** consumer() **throws** IOException, ShutdownSignalException, ConsumerCancelledException, InterruptedException{

//定义队列

Channel channel = connection.createChannel();

String queueName = "simple";

channel.queueDeclare(queueName, **true**, **false**, **false**, **null**);

//定义消费者队列

QueueingConsumer consumer = **new** QueueingConsumer(channel);

//将消费者与队列进行绑定

/\*\*

\* queue : 队列

\* callback 回调给消费者

\* autoACK 发送确认消息 true 自动回复 false

\*/

channel.basicConsume(queueName, **true**, consumer);

//让消费者主动去队列中获取消息 配置监听器

**while**(**true**){

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String msg = **new** String(delivery.getBody());

System.***out***.println("成功获取消息:"+msg);

}

}

## 工作模式

### 模式说明



说明:

由一个生产者负责消息写入队列,但是如果有一个消费者负责消费,可能会造成消息的积压.所以准备多个消费者共同消费一个队列中的消息.

### 定义生产者

@Before //表示在执行@Test注解之前执行该操作

**public** **void** init() **throws** IOException{

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("192.168.126.172");

factory.setPort(5672);

factory.setUsername("jtadmin");

factory.setPassword("jtadmin");

factory.setVirtualHost("/jt");

connection = factory.newConnection();

}

@Test

**public** **void** provider() **throws** IOException{

Channel channel = connection.createChannel();

channel.queueDeclare(queueName, **false**, **false**, **false**, **null**);

String msg = "我是工作模式";

channel.basicPublish("", queueName, **null**, msg.getBytes());

System.***out***.println("消息队列执行成功!!!!");

}

### 定义消费者

//定义消费者

@Test

**public** **void** consumer1() **throws** IOException, ShutdownSignalException, ConsumerCancelledException, InterruptedException{

//定义通道

Channel channel = connection.createChannel();

//定义队列

channel.queueDeclare(queueName, **false**, **false**, **false**, **null**);

//定义消费数 每次只能消费一条记录.当消息执行后需要返回ack确认消息 才能执行下一条

channel.basicQos(1);

//定义消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

//将队列和消费者绑定 false表示手动返回ack

channel.basicConsume(queueName, **false**, consumer);

**while**(**true**){

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String msg = **new** String(delivery.getBody());

System.***out***.println("队列A获取消息:"+msg);

//deliveryTag 队列下标位置

//multiple是否批量返回

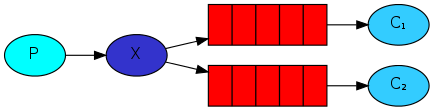
channel.basicAck(delivery.getEnvelope().getDeliveryTag(),**true**);

}

}

## 发布订阅模式

### 发布订阅



X: 代表交换机 如果有消息则在发布订阅模式中,将消息发送到连接交换机的队列中

特点:如果生产者发送消息,那么订阅的全部消费者都会执行消息.

### 定义生产者

//定义生产者

@Test

**public** **void** proverder() **throws** IOException{

//定义通道

Channel channel = connection.createChannel();

//定义交换机名称

String exchange\_name = "E1";

//定义发布订阅模式 fanout redirect 路由模式 topic 主题模式

channel.exchangeDeclare(exchange\_name, "fanout");

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

String msg = "发布订阅模式"+i;

channel.basicPublish(exchange\_name, "", **null**, msg.getBytes());

}

channel.close();

connection.close();

}

### 定义消费者

/\*\*

\* 消费者需要定义队列名称 并且与交换机绑定

\* **@throws** IOException

\* **@throws** InterruptedException

\* **@throws** ConsumerCancelledException

\* **@throws** ShutdownSignalException

\*/

@Test

**public** **void** consumer1() **throws** IOException, ShutdownSignalException, ConsumerCancelledException, InterruptedException{

Channel channel = connection.createChannel();

String exchange\_name = "E1";

String queue\_name = "c\_1";

//定义交换机模式

channel.exchangeDeclare(exchange\_name, "fanout");

//定义队列

channel.queueDeclare(queue\_name, **false**, **false**, **false**, **null**);

//将队列和交换机绑定 key:表示接收数据标识

channel.queueBind(queue\_name, exchange\_name, "");

//定义消费数量

channel.basicQos(1);

//定义消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

//将消费者和队列绑定,并且需要手动返回

channel.basicConsume(queue\_name, **false**, consumer);

**while**(**true**){

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String msg = "发布订阅模式-消费者1"+**new** String(delivery.getBody());

System.***out***.println(msg);

//false表示一个一个返回

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

}

}

说明:先启动消费者,之后再启动生产者

## 路由模式

### 调用过程



说明:路由模式是发布订阅模式的升级,通过定义不同的路由key使得程序将消息发送到不同的队列中.

### 定义生产者

**private** Connection connection;

@Before //表示在执行@Test注解之前执行该操作

**public** **void** init() **throws** IOException{

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("192.168.126.172");

factory.setPort(5672);

factory.setUsername("jtadmin");

factory.setPassword("jtadmin");

factory.setVirtualHost("/jt");

connection = factory.newConnection();

}

//定义生产者

@Test

**public** **void** proverder() **throws** IOException{

Channel channel = connection.createChannel();

//定义交换机名称

String exchange\_name = "redirect";

//定义发布订阅模式 fanout direct 路由模式 topic 主题模式

channel.exchangeDeclare(exchange\_name, "direct");

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

String msg = "生产者发送消息"+i;

String rontKey = "1707B";

/\*\*

\* exchange:交换机名称

\* routingKey:路由key

\* props:参数

\* body:发送消息

\*/

channel.basicPublish(exchange\_name,rontKey, **null**, msg.getBytes());

}

channel.close();

connection.close();

}

### 定义消费者

/\*\*

\* 消费者需要定义队列名称 并且与交换机绑定

\* **@throws** IOException

\* **@throws** InterruptedException

\* **@throws** ConsumerCancelledException

\* **@throws** ShutdownSignalException

\*/

@Test

**public** **void** consumer1() **throws** IOException, ShutdownSignalException, ConsumerCancelledException, InterruptedException{

//定义通道

Channel channel = connection.createChannel();

//定义交换机名称

String exchange\_name = "redirect";

//定义队列名称

String queue\_name = "c\_r\_1";

//定义交换机模式

channel.exchangeDeclare(exchange\_name, "direct");

//定义队列

channel.queueDeclare(queue\_name, **false**, **false**, **false**, **null**);

//将队列和交换机绑定

/\*\*

\* 参数介绍:

\* queue:队列名称

\* exchange:交换机名称

\* routingKey:路由key

\*/

//channel.queueBind(queue, exchange, routingKey)

channel.queueBind(queue\_name, exchange\_name, "1707A");

//定义消费个数

channel.basicQos(1);

//定义消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

//绑定消息与消费者

channel.basicConsume(queue\_name, **false**, consumer);

**while**(**true**){

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String msg = "路由模式-消费者1"+**new** String(delivery.getBody());

System.***out***.println(msg);

//手动回复 一个一个回复

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

}

}

总结:通过路由key 可以将消息发送到指定的队列中.

## 主题模式

### 调用逻辑



说明:

可以通过路由key将消息发送到一类相同的key中 使用通配符实现

符号说明:

#号:表示任意字符(任意个.)

\*号:任意单个字符或者词组(单个.)

例子:

路由key save.user.aa.bb

save.user.\* A 不可以

save.user.# B 可以

### 定义生产者

**private** Connection connection;

@Before //表示在执行@Test注解之前执行该操作

**public** **void** init() **throws** IOException{

ConnectionFactory factory = **new** ConnectionFactory();

factory.setHost("192.168.126.172");

factory.setPort(5672);

factory.setUsername("jtadmin");

factory.setPassword("jtadmin");

factory.setVirtualHost("/jt");

connection = factory.newConnection();

}

//定义生产者

@Test

**public** **void** proverder() **throws** Exception{

//获取通道

Channel channel = connection.createChannel();

//定义交换机的名称

String exchange\_name = "TOP";

//创建交换机队列

//exchange 交换机名称

//type 定义类型 fanout 发布订阅模式 direct-路由模式 topic-主题模式

channel.exchangeDeclare(exchange\_name, "topic"); //主题模式

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

String msg = "主题模式"+i;

/\*\*

\* 参数说明:

\* exchange:交换机名称

\* routingKey:路由key

\* props:参数

\* body:数据字节码

\*/

//channel.basicPublish(exchange, routingKey, props, body);

channel.basicPublish(exchange\_name,"item.update.aa.bb", **null**, msg.getBytes());

}

channel.close();

connection.close();

}

### 定义消费者

@Test

**public** **void** consumer1() **throws** Exception{

//定义通道

Channel channel = connection.createChannel();

//定义交换机名称

String exchange\_name = "TOP";

//定义队列名称

String queue\_name = "TOP1";

//声明交换机名称以及主题模式

channel.exchangeDeclare(exchange\_name, "topic");

//定义队列

channel.queueDeclare(queue\_name, **false**, **false**, **false**, **null**);

//将交换机和队列进行绑定

//参数1.队列名称 参数2交换机名称 参数3 路由key #号匹配多个字符

channel.queueBind(queue\_name, exchange\_name, "item.#");

channel.basicQos(1); //定义消费数量 1

//定义消费者

QueueingConsumer consumer = **new** QueueingConsumer(channel);

//将队列和消费者绑定

channel.basicConsume(queue\_name, **false**, consumer); //定义手动回复

**while**(**true**){

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

//获取消息队列中的数据

String msg = **new** String(delivery.getBody());

System.***out***.println("item.#消费者1:"+msg);

//手动回复

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

}

}

# Spring整合rabbitMQ

## 整合说明

### 业务设计

说明:由于订单业务逻辑相对复杂.同时入库3张表.所以使用消息队列实现订单入库.

角色划分:

生产者:jt-dubbo-order

消费者:jt-dubbo-rabbitmq

### 添加配置文件

rabbit.ip=192.168.126.172

rabbit.port=5672

rabbit.username=jtadmin

rabbit.password=jtadmin

rabbit.vhost=/jt

交给spring容器管理

<list>

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

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

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

</list>

### 编辑生产者配置文件

<!-- 异步的线程池，线程池的最在数不能设定太小，不然<rabbit:listener/>/@RabbitListener太多的话，会出现发无法正常消费问题 -->

<task:executor id=*"taskExecutor"* pool-size=*"4-256"* queue-capacity=*"128"* />

<!-- 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}"*

publisher-confirms=*"true"*

publisher-returns=*"true"*

channel-cache-size=*"5"*

executor=*"taskExecutor"*/>

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

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

exchange=*"orderExchange"* />

<!-- MQ的管理，包括队列、交换器等 -->

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

<!-- 定义交换器，自动声明交换机 ,durable持久化 -->

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

</rabbit:direct-exchange>

### 发送消息

//通过消息队列实现订单入库

@Override

**public** String saveOrder(Order order) {

String orderId = order.getUserId() + "" +System.*currentTimeMillis*();

order.setOrderId(orderId);

String routingKey = "save.order"; //定义路由key

rabbitTemplate.convertAndSend(routingKey, order);

**return** orderId;

}

### 定义消费者配置文件

<!-- 异步的线程池，线程池的最在数不能设定太小，不然<rabbit:listener/>/@RabbitListener太多的话，会出现发无法正常消费问题 -->

<task:executor id=*"taskExecutor"* pool-size=*"4-256"* queue-capacity=*"128"* />

<!-- 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}"*

publisher-confirms=*"true"*

publisher-returns=*"true"*

channel-cache-size=*"5"*

executor=*"taskExecutor"*/>

<!-- 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=*"orderQueue"* 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=*"orderExchange"* auto-declare=*"true"*>

<rabbit:bindings>

<!-- 前台系统只接收商品更新的消息，key路由key -->

<rabbit:binding queue=*"orderQueue"* key=*"save.order"*/>

</rabbit:bindings>

</rabbit:direct-exchange>

<!-- 定义监听 -->

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

<!-- 监听一个队列，当队列中有消息，就会自动触发类.方法，传递消息就作为方法的参数，根据方法声明的参数强转 -->

<rabbit:listener ref=*"rabbitMQService"* method=*"saveOrder"* queue-names=*"orderQueue"*/>

</rabbit:listener-container>

<bean id=*"rabbitMQService"* class=*"com.jt.order.service.RabbitMQService"*></bean>

### 定义消费者实现类

**public** **class** RabbitMQService {

@Autowired

**private** OrderMapper orderMapper;

@Autowired

**private** OrderItemMapper orderItemMapper;

@Autowired

**private** OrderShippingMapper orderShippingMapper;

//将消息队列中的内容进行入库操作

**public** **void** saveOrder(Order order){

String orderId = order.getOrderId();

Date date = **new** Date();

//入库订单

order.setOrderId(orderId);

order.setStatus(1);

order.setCreated(date);

order.setUpdated(date);

orderMapper.insert(order);

System.***out***.println("订单入库成功!!!!!");

//获取订单物流信息

OrderShipping orderShipping = order.getOrderShipping();

orderShipping.setOrderId(orderId);

orderShipping.setCreated(date);

orderShipping.setUpdated(date);

orderShippingMapper.insert(orderShipping);

System.***out***.println("订单物流信息入库成功!!");

//实现订单商品入库

List<OrderItem> orderItemList = order.getOrderItems();

**for** (OrderItem orderItem : orderItemList) {

orderItem.setOrderId(orderId);

orderItem.setCreated(date);

orderItem.setUpdated(date);

orderItemMapper.insert(orderItem);

}

System.***out***.println("消息队列执行成功!!!!");

}

}

### 关于RabbitMQ返回值问题

说明:由于使用消息队列目的是为了减轻后台服务器压力.采用**异步**的操作.导致数据返回过早查询数据失败.

解决:

1. 订单入库时定时跳转(5秒)
2. 订单入库成功后,给用户弹框提示
3. 如果遇到高并发的问题.提示友好页面让用户稍后查询(15-30分钟)

