# 京淘的架构优化

## 消息中间件

### 现在京淘架构中存在的问题

说明:如果服务器遇到了高并发,但是又没有足够的机器时,可能会导致服务器出现宕机现象,直接影响整个服务.

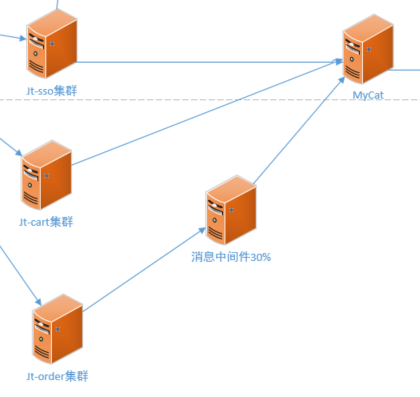
直接原因:遇到了高并发,请求要求数据库**立即**执行操作,并且未执行的请求处于等待的状态.

### 消息中间件介绍

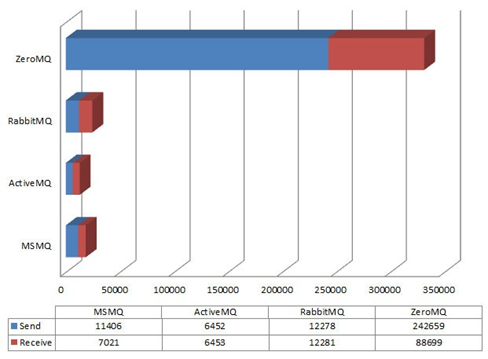
说明:使用了消息中间件可以将数据先存入消息队列中,之后消息队列中的消息被别人进行消费(信息处理).

主要作用:

1. 平衡数据库压力
2. 实现入库的异步操作



### 消息中间件的种类



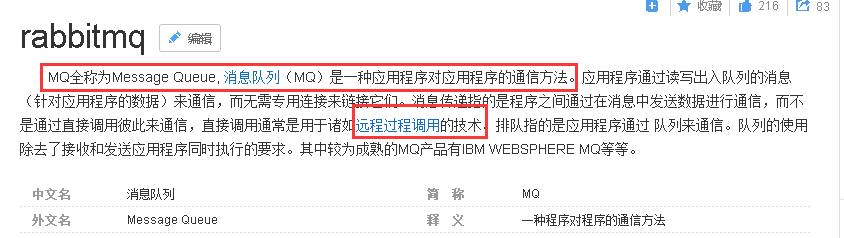
zeroMQ:该中间件的性能超然,但是丢数据也是时常发生.(大数据)

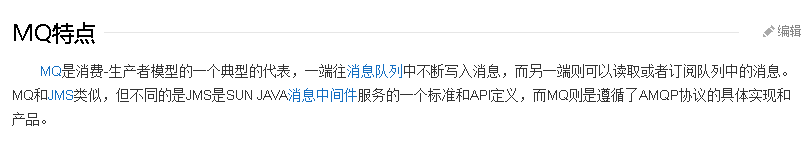
Kafka: 大数据中高吞吐量的消息中间件(大数据)

RabbitMQ:早期专门为银行设计的消息中间件性能和安全性更好(爱立信)

## RabbitMQ介绍

### RabbitMQ介绍



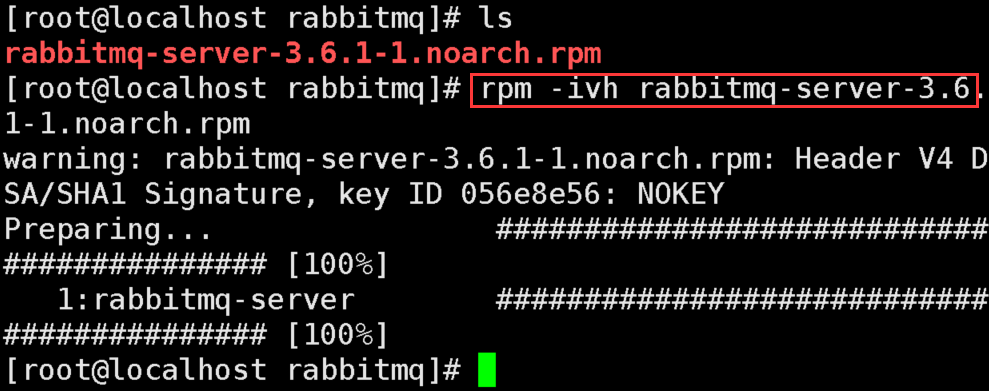


### RabbitMQ下载



## 安装 RabbitMQ

### 上传安装文件



### 添加配置文件

说明:将课前资料中的配置文件加入到/etc/rabbitmq下

下边的配置表示配置文件的来源地址

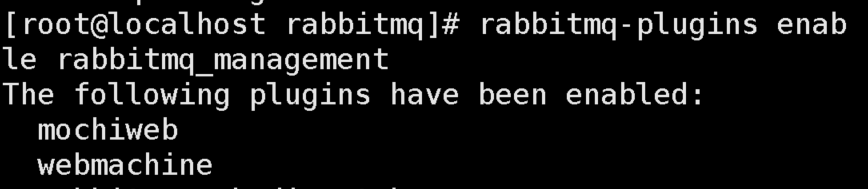
Cp /usr/share/doc/rabbitmq-server-3.6.1/rabbitmq.config.example /etc/rabbitmq/rabbitmq.config

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

### RabbitMQ服务注册

命令: rabbitmq-plugins enable rabbitmq\_management



端口说明:

1.15672:表示rabbitmq的客户端平台可以通过发可视化的图形界面操作rabbitmq

2.5672: 链接rabbitMQ客户端

启动命令:

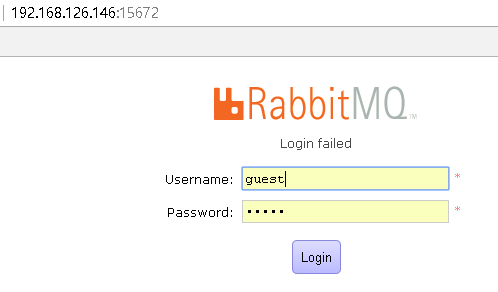
service rabbitmq-server start

service rabbitmq-server stop

service rabbitmq-server restart

### rabbitMQ登录

说明:根据ip:端口登录项目



### rabbitMQ视图介绍



Connections:连接rabbitMQ的链接数量

Channels:用来管理队列/交换机/路由器

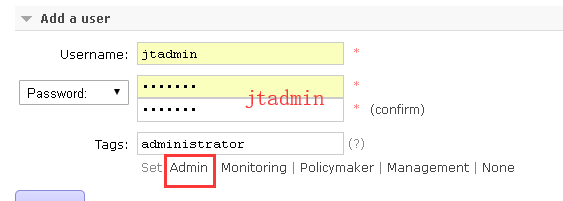
Exchanges:作用将消息发往多个队列中

Queues:可以用来存放消息

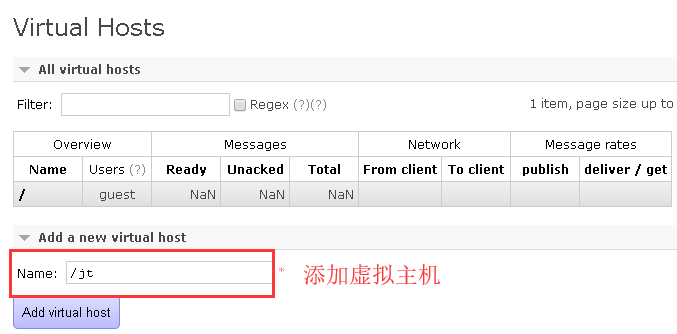
Admin:rabbitMQ的控台台

### 创建新用户

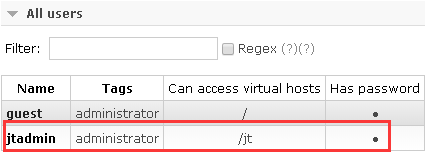
1. 创建新的用户



1. 创建新的虚拟主机



1. 将用户与虚拟主机关联



# RabbitMQ的工作模式

## 准备工作

### 引入jar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>

## 简单模式

### 模式设计



P:provider 消息的提供者

C:Consumer 消息的消费者

队列:先进先出

原理说明:  
 当消息的提供者将信息写入队列中,如果这时队列中没有其他的消息,则会在第一时间被消费者处理.

消费者会实时的监听队列中是否有消息,如果有则进行处理.

### 实现代码

简单模式实现-提供者

**public** **class** TestRabbitMQ {

/\*\*

\* 1.通过对象获取Connection连接

\* 1.1注入ip/端口/用户名/密码/虚拟主机

\*/

**private** Connection connection = **null**;

@Before

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

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

factory.setHost("192.168.126.146");

factory.setPort(5672);

factory.setUsername("jtadmin");

factory.setPassword("jtadmin");

factory.setVirtualHost("/jt");

connection = factory.newConnection();

}

//定义提供者

@Test

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

//1.创建channel对象 管理队列/生产者/消费者

Channel channel = connection.createChannel();

//2.定义队列的名称

String queueName = "simple";

//3.创建对象队列

/\*\*

\* 参数介绍

\* 1.queue 队列的名称

\* 2.durable 是否持久化 如果队列中有消息那么当队列重启时,队列

\* 中的消息是否恢复 true/false

\* 3.exclusive 表示是否为提供者独有,如果设置为true则消费者不能消费

\* 4.autoDelete 如果消息队列中没有消息是否自动删除 false

\* 5.arguments 是否传递参数 如果没有参数添加null

\*/

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

//4.准备需要发送的消息

String msg = "我是简单模式";

//5.将消息发送到消息队列中

/\*\*

\* exchange:表示交换机 将消息发往不同的队列中 如果没有添加""空串

\* routingKey:路由key 根据指定的路由key将消息发往特定的队列中.

\* 如果没有路由key则使用队列名称代替

\*

\* props:第三方的配置文件 如果没有使用null

\* body: 消息转化后的字节码数组

\*/

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

//6.关闭通道和连接

channel.close();

System.***out***.println("程序执行成功!!!");

}

}

消息队列-消费者

/\*\*

\* 1.创建channel对象

\* 2.创建队列

\* 3.创建消费者对象

\* 4.从对象中获取信息

\* 5.信息输出

\* **@throws** IOException

\* **@throws** InterruptedException

\* **@throws** ConsumerCancelledException

\* **@throws** ShutdownSignalException

\*/

@Test

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

Channel channel = connection.createChannel();

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

//创建消费者对象

QueueingConsumer consumer =

**new** QueueingConsumer(channel);

/\*\*

\* 将消费者和队列进行绑定

\* 1.queue:队列的名称

\* 2.autoAck rabbitMQ中的返回信息 true表示自动回复 false需要手动回复

\* 由于网络原因可能会造成回复不及时. 最大允许3次没有回复.

\* 3.callback 表示回调 消息最终发送给谁.

\*/

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

//通过循环方式 实现队列的监听

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

//delivery中包含了消息

QueueingConsumer.Delivery delivery =

consumer.nextDelivery();

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

System.***out***.println("获取队列消息成功!!!!"+msg);

}

}

## 工作模式

### 工作模式原理



说明:

由一个生产者和多个消费者共同构成,当生产者将消息写入队列中时,由多个消费者争抢执行.谁先抢到谁执行.

特点:一般公司中采用工作模式,也有人称之为”双工模式”.目的是为了快速的消费队列中的消息.

### 测试代码

工作模式测试代码

**public** **class** TestRabbitMQ\_\_2\_work {

**private** Connection connection = **null**;

String queueName = "work";

@Before

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

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

factory.setHost("192.168.126.146");

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, **true**, **false**, **false**, **null**);

String msg = "我是工作模式";

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

channel.close();

connection.close();

}

@Test

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

Channel channel = connection.createChannel();

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

//设定每次消费的消息数 每次最大拿一个消息进行处理

channel.basicQos(1);

//定义消费者对象

QueueingConsumer consumer1 = **new** QueueingConsumer(channel);

//消息者和队列进行绑定 如果为false需要手动返回

channel.basicConsume(queueName, **false**, consumer1);

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

QueueingConsumer.Delivery delivery =

consumer1.nextDelivery();

System.***out***.println("获取消息1:"+**new** String(delivery.getBody()));

//手动回复 getDeliveryTag()获取消息的下标位置

//multiple 是否批量回复 ack

channel.basicAck

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

}

}

@Test

**public** **void** consumer2() **throws** IOException, ShutdownSignalException, ConsumerCancelledException, InterruptedException{

Channel channel = connection.createChannel();

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

//设定每次消费的消息数 每次最大拿一个消息进行处理

channel.basicQos(1);

//定义消费者对象

QueueingConsumer consumer1 = **new** QueueingConsumer(channel);

//消息者和队列进行绑定 如果为false需要手动返回

channel.basicConsume(queueName, **false**, consumer1);

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

QueueingConsumer.Delivery delivery =

consumer1.nextDelivery();

System.***out***.println("获取消息2:"+**new** String(delivery.getBody()));

//手动回复 getDeliveryTag()获取消息的下标位置

//multiple 是否批量回复 ack

channel.basicAck

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

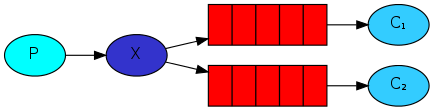
}

}

}

## 发布订阅模式

### 发布订阅的原理



**说明:**

X:代表交换机 主要的作用就是将消息发往不同的队列中.

发布订阅模式中队列可以有多个,每个队列中有不同的消费者进行消费.

并且每个消费者只关注自己的队列.

测试时,先启动消费者 之后启动生产者

**特点:**

**一个消息被消费多次,适用于群发或者广播**

### 发布订阅模式代码

发布定于模式代码 1个生产者多个消费者

//发布订阅模式

**private** Connection connection = **null**;

//定义rabbit连接池

@Before

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

//定义工厂对象

ConnectionFactory connectionFactory = **new** ConnectionFactory();

//设定参数

connectionFactory.setHost("192.168.126.146");

connectionFactory.setPort(5672);

connectionFactory.setVirtualHost("/jt");

connectionFactory.setUsername("jtadmin");

connectionFactory.setPassword("jtadmin");

//创建连接

connection = connectionFactory.newConnection();

}

//定义生产者 生成者不会直接链接队列

@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

/\*\*

\* exchange\_name:交换机的名称

\* type : fanout表示发布订阅模式 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();

}

/\*\*

\* 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"; //订阅了生成者

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 String 路由key

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**);

}

}

/\*\*

\* 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** consumer2() **throws** IOException, ShutdownSignalException, ConsumerCancelledException, InterruptedException{

Channel channel = connection.createChannel();

String exchange\_name = "E1";

String queue\_name = "c\_2";

//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

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

channel.basicQos(1);

//定义消费者

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

//定义回复方式

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

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

QueueingConsumer.Delivery delivery = consumer.nextDelivery();

String msg = "发布订阅模式-消费者2"+**new** String(delivery.getBody());

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

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

}

}

## 路由模式

### 原理说明



Fanout发布订阅 direct 路由模式 topic 主题模式

**说明:**

路由模式是发布订阅模式的升级版,可以根据不同的路由key将消息发往不同的队列中,并且执行.

路由key的指定消息的发送方指定,在消息的消费方再次指定,之后通过路由key比较,如果路由key相同则向该队列中发送消息.

### 代码测试

路由模式,需要指定路由key

**private** Connection connection = **null**;

//定义rabbit连接池

@Before

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

//定义工厂对象

ConnectionFactory connectionFactory = **new** ConnectionFactory();

//设定参数

connectionFactory.setHost("192.168.126.146");

connectionFactory.setPort(5672);

connectionFactory.setVirtualHost("/jt");

connectionFactory.setUsername("jtadmin");

connectionFactory.setPassword("jtadmin");

//创建连接

connection = connectionFactory.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();

}

/\*\*

\* 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**);

}

}

}

## 主题模式

### 原理说明



说明:利用主题模式可以将消息发往一类路由key相同的队列中

使用 \*号#号进行通配

\*号只能匹配一个或者几个字符 不能匹配多个.

#号表示任意类型的通配 可以匹配多个**.** Item.update.1712(一般使用很少)

### 测试代码

//主题模式

**public** **class** TestRabbitMQ\_5\_topic {

**private** Connection connection = **null**;

@Before

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

//1.定义ConnectionFactory对象

ConnectionFactory connectionFactory = **new** ConnectionFactory();

connectionFactory.setHost("192.168.126.146");

connectionFactory.setPort(5672);

connectionFactory.setVirtualHost("/jt");

connectionFactory.setUsername("jtadmin");

connectionFactory.setPassword("jtadmin");

//获取连接

connection = connectionFactory.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**", **null**, msg.getBytes());

}

channel.close();

connection.close();

}

@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.

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**);

}

}

@Test

**public** **void** consumer2() **throws** Exception{

Channel channel = connection.createChannel();

String exchange\_name = "TOP";

String queue\_name = "TOP2";

//生命交换机模式

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定义路由key

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);

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2:"+msg);

//定义回执

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

}

}

}

# Spring整合RabbitMQ

## 消息队列的准备工作

### 说明

通过架构设计,规定了jt-order中它是生产者负责将消息写入到消息队列中.

应该创建一个jt-rabbitmq项目实现消息队列的消费.将order信息进行入库处理.

为了实现该项目简单实现的步骤

1.依然使用jt-order充当生产者.同时jt-order充当消费者.通过不同的配置文件来标识.

## 定义生产者

### 添加properties配置文件

说明:将properties文件添加到项目中的具体文件夹下

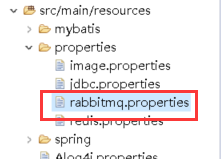
rabbit.ip=192.168.126.146

rabbit.port=5672

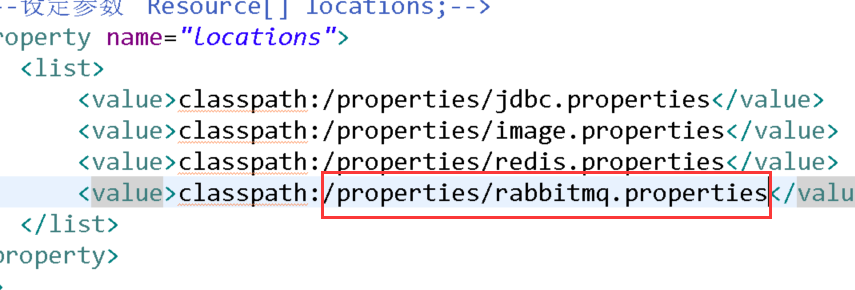
rabbit.username=jtadmin

rabbit.password=jtadmin

rabbit.vhost=/jt



2.spring容器加载次文件



### 添加生产者配置文件

生产者的配置文件

<!-- 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=*"****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,durable持久化 -->

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

</rabbit:direct-exchange>

### 通过rabbitMQ实现消息的发送

将order对象发送到消息队列中

@Autowired

**private** RabbitTemplate rabbitTemplate;

//先将数据写入消息队列中

@Override

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

//生成orderId

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

order.setOrderId(orderId);

//routingKey 定义路由key 将对象进行发送

rabbitTemplate.convertAndSend("save.order", order);

**return** orderId;

}

## 定义消费者

### 添加消费者配置文件

<!-- 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=*"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=*"rabbitOrderService"* method=*"****saveOrder****"* queue-names=*"orderQueue"*/>

</rabbit:listener-container>

<!--自定义bean处理消息队列中的内容 -->

<bean id=*"rabbitOrderService"* class=*"com.jt.order.service.RabbitMQOrderService"*></bean>

### 编辑入库操作

消息实现入库操作

**public** **class** RabbitMQOrderService {

@Autowired

**private** OrderMapper orderMapper;

@Autowired

**private** OrderItemMapper orderItemMapper;

@Autowired

**private** OrderShippingMapper orderShippingMapper;

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

String orderId = order.getOrderId();

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

//1.实现订单表入库

order.setOrderId(orderId);

order.setStatus(1);//表示未支付

order.setCreated(date);

order.setUpdated(date);

orderMapper.insert(order);

System.***out***.println("订单表入库成功");

//2.实现订单物流入库

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");

//3.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("订单商品入库成功:消息队列执行完成");

}

}

### 页面展现



### RabbitMQ中存在的问题

1. 如何解决异步后数据回显问题

由于使用了rabbitMQ使得数据入库和数据的回显是异步的.为了达到一致性的要求.

一般弹出对话框或者定时跳转页面.目的是等待消息入库.之后再次查询数据库.