# 课程计划

* 商品数据导入
* 实现商品搜索
* 数据同步问题
* ActiveMQ介绍
* ActiveMQ使用

# 商品数据导入

## 导入商品数据到数据库

将“资料\sql”的数据库文件导入到淘淘商城数据库中。

## 设置Nginx访问图片

### 解压图片

将“资料\京东商品图片\jd.zip”文件进行解压，放置到某个路径下；如：



### 配置nginx

在nginx的配置文件nginx.conf中添加如下一个服务信息：

|  |
| --- |
| server {  listen 80;  server\_name image.taotao.com;  proxy\_set\_header X-Forwarded-Host $host;  proxy\_set\_header X-Forwarded-Server $host;  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  location / {  root F://taotao;  }  } |

### 设置hosts

在系统hosts文件中添加：

127.0.0.1 image.taotao.com

测试：可以编辑商品，查看图片是否能够显示。

## 导入商品数据到solr

### 分析

**需求**：在淘淘商城数据库中，商品表已经有了比较多的商品；现在的需求是：将商品表的商品数据导入到solr索引库中。

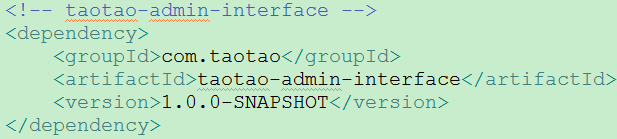
**解决**：使用java代码将后台管理系统中的商品读取到然后再使用solrj把数据插入到solr索引库中。

### 实现

#### 声明ItemService的调用

因为需要查询商品数据列表；所以在taotao-search-web项目需要声明对itemService的调用。

加入依赖：



声明服务调用：

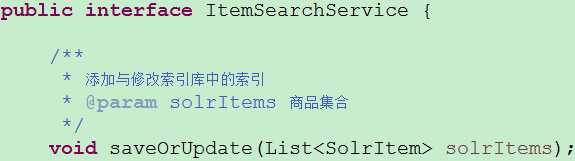
IMG_256

#### 引入SolrItem类

将“资料\vo”中的文件引入到taotao-search-interface项目中

的com.taotao.search.vo包中；引入后修改该文件的package为正确的包路径。

#### ItemSearchService方法



#### ItemSearchServiceImpl方法

|  |
| --- |
| @Service  **public** **class** ItemSearchServiceImpl **implements** ItemSearchService {  /\*\* 注入SolrClient \*/  @Autowired  **private** SolrClient solrClient;  @Value("${defaultCollection}")  **private** String defaultCollection;  /\*\*  \* 批量添加或修改索引  \* **@param** solrItems 集合  \*/  **public** **void** saveOrUpdate(List<SolrItem> solrItems){  **try**{  /\*\* 批量添加，返回修改响应对象 \*/  UpdateResponse response = solrClient  .addBeans(defaultCollection,solrItems);  /\*\* 判断状态码: 0 代表成功 \*/  **if** (response.getStatus() == 0){  /\*\* 提交事务 \*/  solrClient.commit(defaultCollection);  }**else**{  /\*\* 回滚事务 \*/  solrClient.rollback(defaultCollection);  }  }**catch**(Exception ex){  **throw** **new** RuntimeException(ex);  }  }  } |

#### 测试类

实现

taotao-search-web/src/test/java/com/taotao/search/ItemIndexToSolrTest.java内容如下：

|  |
| --- |
| **public** **class** ItemIndexToSolrTest {    /\*\* 定义ItemSearchService \*/  **private** ItemSearchService itemSearchService;  **private** ItemService itemService;  /\*\* 获取Spring容器 \*/  **private** ApplicationContext ac;    /\*\* 在测试方法之前 \*/  @Before  **public** **void** before() **throws** Exception {  /\*\* 获取Spring容器 \*/  **this**.ac = **new** ClassPathXmlApplicationContext("classpath:taotao-search-web-servlet.xml");  /\*\* 获取ItemSearchService服务 \*/  **this**.itemSearchService = ac.getBean(ItemSearchService.**class**);  /\*\* 获取ItemService服务 \*/  **this**.itemService = ac.getBean(ItemService.**class**);  }  /\*\* 测试查询数据库中的数据全部创建索引 \*/  @Test  **public** **void** index() {  /\*\* 定义当前页码 \*/  **int** page = 1;  /\*\* 定义每页显示的数据 \*/  **int** rows = 500;  /\*\* 循环读取数据库中的数据，写入索引库 \*/  **do** {  System.***out***.println("开始写第【" + page + "】页的索引");  /\*\* 查询数据 \*/  List<Item> items = itemService.selectByPage(page, rows);  /\*\* 判断集合 \*/  **if** (items != **null** && items.size() > 0){  /\*\* 把List<Item>转化成List<SolrItem> \*/  List<SolrItem> solrItems = **new** ArrayList<SolrItem>();  /\*\* 迭代items \*/  **for** (Item item : items){  SolrItem solrItem = **new** SolrItem();  solrItem.setId(item.getId());  solrItem.setImage(item.getImage());  solrItem.setPrice(item.getPrice());  solrItem.setSellPoint(item.getSellPoint());  solrItem.setStatus(item.getStatus());  solrItem.setTitle(item.getTitle());  solrItems.add(solrItem);  }  /\*\* 再把List<SolrItem>添加到索引库 \*/  itemSearchService.saveOrUpdate(solrItems);  System.***out***.println("结束写第【" + page + "】页的索引");  System.***out***.println("=========华丽分隔线==========");  page++;  rows = items.size();  }else{  rows = 0;  }  } **while** (rows == 500);  }  } |

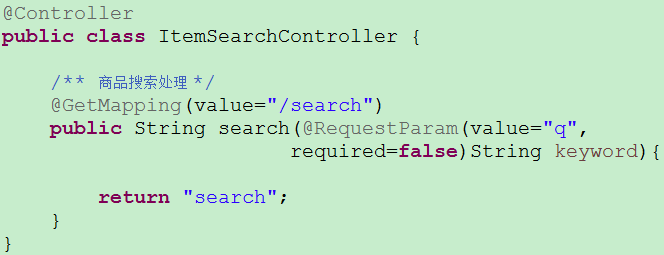
# 实现商品搜索

## 跳转到搜索页面

在淘淘商城首页可以搜索商品；点击商品搜索，发起请求到search.taotao.com，携带关键字q

请求地址：http://search.taotao.com/search?q=xxx&page=1

在taotao-search-web项目实现ItemSearchController处理器处理搜索请求。



## 解决乱码问题

在上述跳转到搜索结果后，发现在后台接收q这个搜索关键字如果搜索中文接收的时候出现了乱码。

解决方案：

1. 重新编码

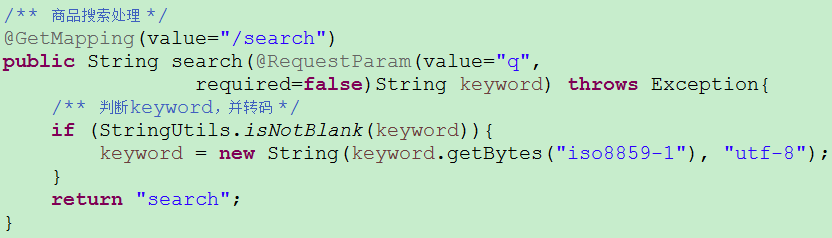
可以把请求的关键字用ISO-8859-1解码，再用UTF-8编码即可

keyWords = new String(keyWords.getBytes("ISO-8859-1"), "UTF-8");

1. 修改Tomcat的server.xml文件，改里面的一个配置编码为UTF-8

<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

在项目中；选择方案1，具体修改后如下：



## 搜索页面分析

taotao-search-web的search.jsp页面

分析页面需要展示(需要放到Model中的)数据有：

${query}

搜索关键字，直接回显即可

${itemList}

商品列表

${page}

当前页码数，由页面直接提交到后台，直接回显

${totalPage}

总页数，查询结果的总页数，需要返回查询数据总条数进行计算

方法一：total%rows==0?total/rows:(total/rows)+1

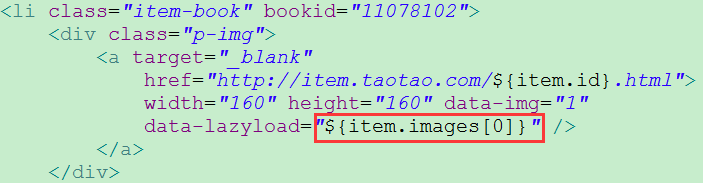
方法二：(total -1)/rows + 1

这样查询一次solr索引库，需要返回两个数据，一个是商品结果集list，另一个是数据总条数。

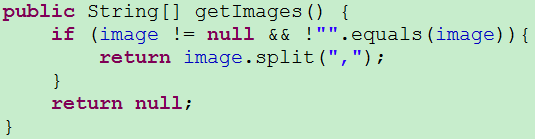
在页面发送搜索请求时会携带页号page，但是没有携带页大小rows

基于安全考虑，这里不能让用户指定rows。否则用户可以指定一次查询很多商品数据。这样返回的数据量会变得很大，如果恶意访问会造成很大的访问压力，很不安全。所以在开发时自行指定。指定20，一页显示20条

前端页面展示需要Item的images属性



修改taotao-search-interface的SolrItem，增加getImages()方法：

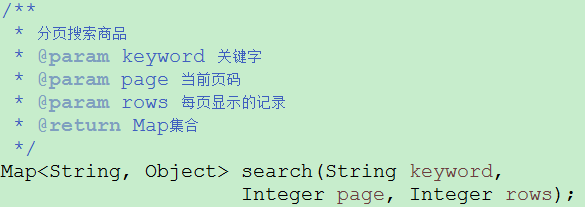


## 后端实现

### ItemSearchController方法

|  |
| --- |
| @Controller  **public** **class** ItemSearchController {  /\*\* 注入商品检索服务 \*/  @Autowired  **private** ItemSearchService itemSearchService;    /\*\* 商品搜索处理 \*/  @GetMapping(value="/search")  **public** String search(@RequestParam(value="q",required=**false**)String keyword,  @RequestParam(value="page",defaultValue="1")Integer page,  Model model) **throws** Exception{  /\*\* 判断keyword，并转码 \*/  **if** (StringUtils.*isNotBlank*(keyword)){  keyword = **new** String(keyword.getBytes("iso8859-1"), "utf-8");  }  Map<String,Object> maps = itemSearchService.search(keyword, page, 20);  model.addAttribute("query", keyword);  model.addAttribute("page", page);  **if** (maps != **null** && maps.size() > 0){  model.addAllAttributes(maps);  }  **return** "search";  }  } |

### ItemSearchService方法



### ItemSearchServiceImpl方法

|  |
| --- |
| /\*\*  \* 分页搜索商品  \* **@param** keyword 搜索关键字  \* **@param** page 当前页码  \* **@param** rows 每页显示的记录数  \* **@return** Map集合  \*/  **public** Map<String, Object> search(String keyword, Integer page, Integer rows){  **try**{  /\*\* 判断搜索关键字 \*/  **if** (StringUtils.*isBlank*(keyword)){  keyword = "\*";  }  /\*\* 创建SolrQuery封装查询参数 \*/  SolrQuery solrQuery = **new** SolrQuery();  /\*\* 查询字符串 \*/  solrQuery.setQuery("title:" + keyword + " AND status:1");  /\*\* 设置分页开始记录数 limit的第一个号\*/  solrQuery.setStart((page - 1) \* rows);  /\*\* 设置每页显示的记录数 \*/  solrQuery.setRows(rows);    /\*\* 判断是否需要高亮显示 \*/  **if** (!"\*".equals(keyword)){  /\*\* 设置高亮显示 \*/  solrQuery.setHighlight(**true**);  /\*\* 设置高亮显示字段 \*/  solrQuery.addHighlightField("title");  /\*\* 设置高亮文本截断 \*/  solrQuery.setHighlightFragsize(60);  /\*\* 设置高亮格式器前缀 \*/  solrQuery.setHighlightSimplePre("<font color='red'>");  /\*\* 设置高亮格式器后缀 \*/  solrQuery.setHighlightSimplePost("</font>");  }    /\*\* 检索，得到查询响应对象 \*/  QueryResponse response = solrClient.query(defaultCollection, solrQuery);  /\*\* 判断响应状态码 \*/  **if** (response.getStatus() == 0){  /\*\* 定义Map集合封装数据 \*/  Map<String, Object> data = **new** HashMap<>();  /\*\* 获取总命中的记录数 \*/  **long** count = response.getResults().getNumFound();  // ${totalPage}: 总页数  // count % rows == 0 ? count / rows : (count / rows) + 1  // ((count - 1) / rows) + 1  data.put("totalPage", ((count - 1) / rows) + 1);    // ${itemList}: 商品集合List<SolrItem>  List<SolrItem> itemList = response.getBeans(SolrItem.**class**);  /\*\* 判断是否需要高亮: title要高亮 \*/  **if** (solrQuery.getHighlight()){  /\*\* 获取高亮内容 \*/  Map<String, Map<String, List<String>>> hMaps = response.getHighlighting();  /\*\* 迭代itemList集合 \*/  **for** (SolrItem solrItem : itemList){  /\*\* 获取标题高亮内容 \*/  String title = hMaps.get(String.*valueOf*(solrItem.getId())).get("title").get(0);  /\*\* 设置标题内容为高亮后的内容 \*/  solrItem.setTitle(title);  }  }  data.put("itemList", itemList);  **return** data;  }  }**catch**(Exception ex){  **throw** **new** RuntimeException(ex);  }  **return** **null**;  } |

## 商品数据同步问题

后台管理实现商品的MySQL数据管理；搜索系统实现商品的solr索引库管理

问题：

后台管理系统增加、更新、删除商品数据的时候，索引库没有同步执行

解决方案：

**方案一**：在后台管理系统CRUD操作时候，调用搜索系统服务进行更新；但是如果以后其它系统也需要同步商品数据的话又得调用其它系统的服务方法，这么一来后台管理系统和其它系统的耦合度太大，增加开发、维护成本。

**方案二**：在后台管理系统CRUD操作成功后通知哪些需要知道商品发生了变更的系统，通知这些系统哪个商品做了什么操作，让这些系统自身去更新它该更新的数据。这样系统之间的耦合度就大大降低了。要做得服务器之间这种消息的通知，可以使用消息队列。

# ActiveMQ

## MQ解决同步问题



1. 后台管理系统进行商品修改，通知MQ；
2. 其它系统监听MQ是否有消息。如果没有消息，持续监听，如果有消息，处理相关业务逻辑

## MQ

MQ全称为Message Queue，消息队列是应用程序和应用程序之间的通信方法。

常见MQ框架：Kafka、RabbitMQ、ActiveMQ、RocketMQ。

* 为什么使用MQ

在项目中，将一些无需即时返回且耗时的操作提取出来，进行了**异步处理**，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而**提高**了**系统**的**吞吐量**。

* ActiveMQ解决了如下的问题

1. 信息的发送者和接收者如何维持这个连接，如果一方的连接中断，这期间的数据如何方式丢失？
2. 如何降低发送者和接收者的耦合度？
3. 如何让优先级高的接收者先接到数据？
4. 如何做到有效均衡接收者的负载？
5. 如何有效的将数据发送到相关的接收者？也就是说将接收者订阅不同的数据，如何做有效的过滤。
6. 如何做到可扩展，甚至将这个通信模块发到集群上？
7. 如何保证接收者接收到了完整，正确的数据？

## JMS

JMS即Java消息服务（Java Message Service），是一个Java平台中关于面向消息中间件（MOM）的规范，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

JMS是一个与具体平台无关的API，具有跨平台性。

它主要用于在生产者和消费者之间进行消息传递，生产者负责产生消息，而消费者负责接收消息。把它应用到实际的业务需求中的话我们可以在特定的时候利用生产者生成一消息，并进行发送，对应的消费者在接收到对应的消息后去完成对应的业务逻辑。

JMS定义了五种不同的消息正文格式：

　　• StreamMessage : 数据流消息

　　• **MapMessage:** Map键值对消息

　　• **TextMessage:** 文本消息

　　• ObjectMessage: 序列化Java对象消息

　　• BytesMessage: 字节消息

## ActiveMQ

ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS（Java Message Service）1.1和J2EE 1.4规范的 JMS Provider实现,尽管JMS规范出台已经是很久的事情了,但是JMS在当今的J2EE应用中间仍然扮演着特殊的地位。

ActiveMQ消息的传递有两种模式：

一种是**点对点**的，即一个生产者和一个消费者一一对应；

一种是**发布/订阅**模式，即一个生产者产生消息并进行发送后，可以由多个消费者进行接收。

主要特点：

1. 多种语言和协议编写客户端。语言: Java, C, C++, C#, Ruby, Perl, Python, PHP。应用协议: OpenWire,Stomp REST,WS Notification,XMPP,AMQP

2. 完全支持JMS1.1和J2EE 1.4规范 (持久化,XA消息,事务)

3. 对Spring的支持,ActiveMQ可以很容易内嵌到使用Spring的系统里面去

4. 通过了常见J2EE服务器(如 Geronimo,JBoss 4, GlassFish,WebLogic)的测试,其中通过JCA 1.5 resource adaptors的配置,可以让ActiveMQ可以自动的部署到任何兼容J2EE 1.4 商业服务器上

5. 支持多种传送协议:in-VM,TCP,SSL,NIO,UDP,JGroups,JXTA

6. 支持通过JDBC和journal提供高速的消息持久化

7. 从设计上保证了高性能的集群,客户端-服务器,点对点

8. 支持Ajax

9. 支持与Axis的整合

10. 可以很容易得调用内嵌JMS provider,进行测试

## ActiveMQ安装

详见“资料\activeMQ\Linux安装ActiveMQ.docx”

# ActiveMQ使用

## 导入示例工程

复制“资料\activeMQ\activemq-test”到工作空间；然后导入工程。

## queue的使用

点对点模式(Point To Point)



Producer：消息的生产者 ，就是发消息的

Queue：消息队列方式，存储消息到ActiveMQ消息中间件服务器

Consumer：消息的消费者，就是收消息和消费消息

### Producer消息生产者

开发步骤：

1. 创建ConnectionFactory连接工厂

2. 通过连接工厂创建连接Connection

3. 通过连接的start方法开启连接

4. 通过连接对象创建Session会话对象

5. 通过session创建队列Queue，（可以设置queue（点对点）和topic（订阅）)

6. 通过session创建MessageProducer消息的生产者

7. 创建消息对象TextMessage并设置消息文本

8. 通过MessageProducer消息生产者发送消息

9. 关闭Producer、session、connection

|  |
| --- |
| **public** **class** ProducerTest {  **public** **static** **void** main(String[] args) **throws** Exception{  /\*\* 定义消息中间件ActiveMQ服务地址(tcp协议)\*/  String brokerURL = "tcp://192.168.12.128:61616";  /\*\* 创建消息中间件ConnectionFactory连接工厂 \*/  ConnectionFactory connectionFactory  = **new** ActiveMQConnectionFactory(brokerURL);  /\*\* 通过连接工厂创建连接对象 \*/  Connection connection = connectionFactory.createConnection();  /\*\* 通过连接对象开启连接 \*/  connection.start();  /\*\*  \* 通过连接对象创建session  \* 第一个参数：是否开启消息事务 true开启 false不开启  \* 第二个参数：设置应答模式:  \* Session.AUTO\_ACKNOWLEDGE 自动应答  \* Session.CLIENT\_ACKNOWLEDGE: 客户端应答  \* Session.DUPS\_OK\_ACKNOWLEDGE: 重复确定应答  \* Session.SESSION\_TRANSACTED: 会话事务应答(第一个参数为true)  \*/  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  /\*\* 通过Session创建队列(点对点) \*/  Queue queue = session.createQueue("test-queue");  /\*\* 通过Session创建该消息生产者 \*/  MessageProducer messageProducer = session.createProducer(queue);    /\*\* 创建消息体(文本消息体) \*/  TextMessage textMessage = **new** ActiveMQTextMessage();  /\*\* 设置消息体的内容 \*/  textMessage.setText("您好，jms 我来了!");  /\*\* 通过MessageProducer消息生产者发送消息 \*/  messageProducer.send(textMessage);  System.***out***.println("【生产者】已发送了消息：您好，jms 我来了!");    /\*\* 关闭消息生产者、关闭Session、关闭Connection \*/  messageProducer.close();  session.close();  connection.close();  }  } |

### Consumer消息消费者

开发步骤

1. 创建ConnectionFactory连接工厂

2. 通过连接工厂创建连接Connection

3. 通过连接的start方法开启连接

4. 通过连接对象创建Session会话对象

5. 通过session创建队列Queue，（可以设置queue（点对点）和topic（订阅）)

6. 通过session创建MessageConsumer消息消费者对象

7. 通过MessageConsumer消息消费者对象循环接收消息

8. 输出结果

9. 关闭session、连接

#### 循环接收消息

|  |
| --- |
| **public** **class** Consumer1Test {  **public** **static** **void** main(String[] args) **throws** Exception {  /\*\* 定义消息中件ActiveMQ服务地址(tcp协议)\*/  String brokerURL = "tcp://192.168.12.128:61616";  /\*\* 创建消息中间件ConnectionFactory连接工厂 \*/  ConnectionFactory connectionFactory =  **new** ActiveMQConnectionFactory(brokerURL);  /\*\* 通过连接工厂创建Connection连接对象 \*/  Connection connection = connectionFactory.createConnection();  /\*\* 通过Connection开启连接 \*/  connection.start();  /\*\* 通过Connection连接创建Session \*/  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);    /\*\* 通过Session创建queue队列(点对点) \*/  Queue queue = session.createQueue("test-queue");  /\*\* 通过Session创建消费者 \*/  MessageConsumer messageConsumer = session.createConsumer(queue);    /\*\* 消费消息: 第一种方式(循环接收) \*/  **while**(**true**){  /\*\* 接收消息并设置超时时长，单位是毫秒 \*/  Message message = messageConsumer.receive(2000);  /\*\* 判断消息的类型 \*/  **if** (message **instanceof** TextMessage){  /\*\* 类型转换 \*/  TextMessage textMessage = (TextMessage)message;  /\*\* 获取接收到得消息 \*/  System.***out***.println(textMessage.getText());  }  }  }  } |

#### 消息监听器

|  |
| --- |
| **public** **class** Consumer2Test {  **public** **static** **void** main(String[] args) **throws** Exception {    /\*\* 定义消息中间件ActiveMQ服务地址(tcp协议)\*/  String brokerURL = "tcp://192.168.12.128:61616";  /\*\* 创建消息中间件ConnectionFactory连接工厂 \*/  ConnectionFactory connectionFactory =  **new** ActiveMQConnectionFactory(brokerURL);  /\*\* 通过连接工厂创建Connection连接对象 \*/  Connection connection = connectionFactory.createConnection();  /\*\* 通过Connection开启连接 \*/  connection.start();  /\*\* 通过Connection连接创建Session \*/  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);    /\*\* 通过Session创建queue队列(点对点) \*/  Queue queue = session.createQueue("test-queue");  /\*\* 通过Session创建消费者 \*/  MessageConsumer messageConsumer = session.createConsumer(queue);    /\*\* 消费消息: 第二种方式(设置消息监听器) \*/  messageConsumer.setMessageListener(**new** MessageListener() {  @Override  **public** **void** onMessage(Message message){  /\*\* 判断消息的类型 \*/  **if** (message **instanceof** TextMessage){  /\*\* 类型转换 \*/  TextMessage textMessage = (TextMessage)message;  **try**{  /\*\* 获取接收到得消息 \*/  System.***out***.println(textMessage.getText());  }**catch**(Exception ex){  ex.printStackTrace();  }  }  }  });  }  } |

## topic的使用

发布/订阅模式(Publish/Subscribe)



使用方法和queue是一样的，只是在使用Session对象创建Destination时选择topic就可以。

### ProducerTest\_1

|  |
| --- |
| **public** **class** ProducerTest\_1 {  **public** **static** **void** main(String[] args) **throws** Exception{    /\*\* 定义消息中间件ActiveMQ服务地址(tcp协议)\*/  String brokerURL = "tcp://192.168.12.128:61616";  /\*\* 创建ConnectionFactory连接工厂 \*/  ConnectionFactory connectionFactory  = **new** ActiveMQConnectionFactory(brokerURL);  /\*\* 通过连接工厂创建连接对象 \*/  Connection connection = connectionFactory.createConnection();  /\*\* 通过连接对象开启连接 \*/  connection.start();  /\*\*  \* 通过连接对象创建session  \* 第一个参数：是否开启消息事务 true开启 false不开启  \* 第二个参数：设置应答模式:  \* Session.AUTO\_ACKNOWLEDGE 自动应答  \* Session.CLIENT\_ACKNOWLEDGE: 客户端应答  \* Session.DUPS\_OK\_ACKNOWLEDGE: 重复确定应答  \* Session.SESSION\_TRANSACTED: 会话事务应答  \*/  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);  /\*\* 通过Session创建Topic(发布与订阅) \*/  Topic topic = session.createTopic("test-topic");  /\*\* 通过Session创建消息生产者 \*/  MessageProducer messageProducer = session.createProducer(topic);    /\*\* 创建消息体(文本消息体) \*/  TextMessage textMessage = **new** ActiveMQTextMessage();  /\*\* 设置消息体的内容 \*/  textMessage.setText("您好，jms 我来了 Topic!");  /\*\* 通过MessageProducer消息生产者发送消息 \*/  messageProducer.send(textMessage);  System.***out***.println("【生产者】已发送了消息：您好，jms 我来了Topic!");  /\*\* 关闭消息生产者、关闭Session、关闭Connection \*/  messageProducer.close();  session.close();  connection.close();  }  } |

### ConsumerTest\_1

|  |
| --- |
| **public** **class** ConsumerTest\_1 {  **public** **static** **void** main(String[] args) **throws** Exception {  /\*\* 定义消息中间件ActiveMQ服务地址(tcp协议)\*/  String brokerURL = "tcp://192.168.12.128:61616";  /\*\* 创建ConnectionFactory连接工厂 \*/  ConnectionFactory connectionFactory =  **new** ActiveMQConnectionFactory(brokerURL);  /\*\* 通过连接工厂创建Connection连接对象 \*/  Connection connection = connectionFactory.createConnection();  /\*\* 通过Connection开启连接 \*/  connection.start();  /\*\* 通过Connection连接创建Session \*/  Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);    /\*\* 通过Session创建Topic订阅 \*/  Topic topic = session.createTopic("test-topic");  /\*\* 通过Session创建消息消费者 \*/  MessageConsumer messageConsumer = session.createConsumer(topic);    /\*\* 消费消息: (设置消息监听器) \*/  messageConsumer.setMessageListener(**new** MessageListener() {  @Override  **public** **void** onMessage(Message message){  /\*\* 判断消息的类型 \*/  **if** (message **instanceof** TextMessage){  /\*\* 类型转换 \*/  TextMessage textMessage = (TextMessage)message;  **try**{  /\*\* 获取接收到得消息 \*/  System.***out***.println(textMessage.getText());  }**catch**(Exception ex){  ex.printStackTrace();  }  }  }  });  }  } |

## spring整合activeMQ

### 加入依赖

|  |
| --- |
| <!-- 配置依赖关系 -->  <dependencies>  <!-- junit -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>    <!-- 配置activemq -->  <dependency>  <groupId>org.apache.activemq</groupId>  <artifactId>activemq-all</artifactId>  <version>5.14.5</version>  </dependency>  <!-- spring-jms -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jms</artifactId>  <version>4.3.3.RELEASE</version>  </dependency>  </dependencies> |

### 发送消息

1. 创建Spring容器
2. 从容器中获取JMSTemplate对象
3. 定义目标Destination
4. 使用JMSTemplate对象发送消息

|  |
| --- |
| **public** **class** Producer {    **public** **static** **void** main(String[] args) {  /\*\* 创建spring容器 \*/  @SuppressWarnings("resource")  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext-activemq.xml");  /\*\* 从spring容器中获取JMSTemplate，这个对象是用于发送消息的 \*/  JmsTemplate jmsTemplate = context.getBean(JmsTemplate.**class**);    /\*\* 使用jmsTemplate发送消息 \*/  jmsTemplate.send(**new** MessageCreator() {  @Override  **public** Message createMessage(Session session) **throws** JMSException {  TextMessage textMessage = **new** ActiveMQTextMessage();  textMessage.setText("你好！黑马。");  System.***out***.println("发送消息...");  **return** textMessage;  }  });  }  } |

### 接收消息

第一种方式(无消息事务)：

1. 创建一个类实现MessageListener 接口。业务处理在此类中实现。
2. 在spring容器中配置JMS消息监听容器，应用MessageListener 实现类对象接收消息。

|  |
| --- |
| **public** **class** MyMessageListener **implements** MessageListener{  /\*\* 处理消息方法 \*/  @Override  **public** **void** onMessage(Message message) {  /\*\* 判断消息类型 \*/  **if** (message **instanceof** TextMessage){  TextMessage textMessage = (TextMessage)message;  **try** {  System.***out***.println("MyMessageListener: " + textMessage.getText());  } **catch** (Exception e) {  e.printStackTrace();  }  }  }  } |

第二种方式(有消息事务)重点：

1、创建一个类实现SessionAwareMessageListener 接口。业务处理在此类中实现。

2、在spring容器中配置JMS消息监听容器，应用SessionAwareMessageListener 实现类对象接收消息。

|  |
| --- |
| **public** **class** MyMessageListener **implements**  SessionAwareMessageListener<TextMessage> {  /\*\* 处理消息方法 \*/  @Override  **public** **void** onMessage(TextMessage textMessage,  Session session) **throws** JMSException {  /\*\* 获取消息内容 \*/  System.***out***.println("MessageListener: " + textMessage.getText());  /\*\* 提交消息事务 \*/  session.commit();  }  } |

### queue方式配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"utf-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:c=*"http://www.springframework.org/schema/c"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:jms=*"http://www.springframework.org/schema/jms"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/jms*  *http://www.springframework.org/schema/jms/spring-jms.xsd"*>    <!-- ############ 配置消息中间件连接工厂 ################ -->  <bean id=*"activeMQConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>  <!-- 设置消息中间件服务地址brokerURL -->  <property name=*"brokerURL"* value=*"tcp://192.168.12.128:61616"* />  </bean>  <!-- 配置整合Spring的JMS产生单例的连接工厂 -->  <bean id=*"connectionFactory"* class=*"org.springframework.jms.connection.SingleConnectionFactory"*>  <!-- 设置ActiveMQ连接工厂给单例的连接工厂 -->  <property name=*"targetConnectionFactory"* ref=*"activeMQConnectionFactory"*/>  </bean>      <!-- ############## 配置消息生产者 ################# -->  <!-- 配置消息模版JmsTemplate -->  <bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>  <!-- 设置连接工厂 -->  <property name=*"connectionFactory"* ref=*"connectionFactory"*/>  <!-- 设置默认的目标(采用点对点) -->  <property name=*"defaultDestination"* ref=*"queue"*/>  <!-- 设置开启消息事务 -->  <property name=*"sessionTransacted"* value=*"true"*/>  </bean>  <!-- 配置目标(点对点) -->  <bean id=*"queue"* class=*"org.apache.activemq.command.ActiveMQQueue"*>  <!-- 设置队列名称 -->  <constructor-arg value=*"spring-queue"*/>  </bean>    <!-- ############## 配置消息消费者 ################# -->  <!--  配置JMS消息监听容器  connection-factory: 连接工厂  destination-type="queue|topic" 目标类型  acknowledge="transacted" 应答方式(事务)  -->  <jms:listener-container connection-factory=*"connectionFactory"*  destination-type=*"queue"*  acknowledge=*"transacted"*>  <jms:listener destination=*"spring-queue"* ref=*"myMessageListener"*/>  </jms:listener-container>  <!-- 配置消息监听器 -->  <bean id=*"myMessageListener"* class=*"cn.itcast.activemq.spring.MyMessageListener"*/>  </beans> |

### topic方式配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"utf-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:c=*"http://www.springframework.org/schema/c"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:jms=*"http://www.springframework.org/schema/jms"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/jms*  *http://www.springframework.org/schema/jms/spring-jms.xsd"*>    <!-- ############ 配置消息连接工厂 ################ -->  <bean id=*"activeMQConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>  <!-- 设置消息服务器服务地址brokerURL -->  <property name=*"brokerURL"* value=*"tcp://192.168.12.128:61616"* />  </bean>  <!-- 配置整合Spring的JMS产生单例的连接工厂 -->  <bean id=*"connectionFactory"* class=*"org.springframework.jms.connection.SingleConnectionFactory"*>  <!-- 设置ActiveMQ连接工厂给单例的连接工厂 -->  <property name=*"targetConnectionFactory"* ref=*"activeMQConnectionFactory"*/>  </bean>    <!-- ############## 配置消息生产者 ################# -->  <!-- 配置消息模版JmsTemplate -->  <bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>  <!-- 设置连接工厂 -->  <property name=*"connectionFactory"* ref=*"connectionFactory"*/>  <!-- 设置默认的目标(采用点对点) -->  <property name=*"defaultDestination"* ref=*"topic"*/>  <!-- 设置开启消息事务 -->  <property name=*"sessionTransacted"* value=*"true"*/>  </bean>  <!-- 配置目标(发布/订阅) -->  <bean id=*"topic"* class=*"org.apache.activemq.command.ActiveMQTopic"*>  <!-- 设置队列名称 -->  <constructor-arg value=*"spring-topic"*/>  </bean>  <!-- ############## 配置消息消费者 ################# -->  <!-- 配置JMS消息监听容器  connection-factory: 连接工厂  destination-type: 目标类型  acknowledge="transacted" 应答方式(事务)  -->  <jms:listener-container connection-factory=*"connectionFactory"*  destination-type=*"topic"*  acknowledge=*"transacted"*>  <jms:listener destination=*"spring-topic"* ref=*"myMessageListener"*/>  </jms:listener-container>  <!-- 配置消息监听器 -->  <bean id=*"myMessageListener"* class=*"cn.itcast.activemq.spring.MyMessageListener"*/>  </beans> |

## Queue与pub/sub区别

1、点对点PTP(queue队列):

a. 一个消息的生产者只能对应一个消息的消费者(一对一)

b. 消费者一定可以消费该消息(在线或不在线都可以)，

只要有一个消费者消费了该消息就会从消息队列中删除.

2、发布/订阅pub/sub:

a. 一个消息的生产者能对应多个消息的消费者(一对多)

b. 只针对目前在线的所有消费者.(不在线的消费者不能消费该消息)

# ActiveMQ消息事务

怎样保证消息能正确的消费？

假如消费消息的业务处理出现异常？

假如消费消息的服务器宕机？

解决思路：

1. 接收消息方，采用消息事务。处理业务成功时提交消息事务，处理业务失败时回滚消息事务。
2. 采用点对点消息模式，保证消息一定可以被消费者都消费，不然永远存储在消息队列中。

### 发送消息（添加消息事务）

|  |
| --- |
| <!-- ############## 配置消息生产者 ################# -->  <!-- 配置消息模版JmsTemplate -->  <bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>  <!-- 设置连接工厂 -->  <property name=*"connectionFactory"* ref=*"connectionFactory"*/>  <!-- 设置默认的目标(采用点对点) -->  <property name=*"defaultDestination"* ref=*"queue"*/>  <!-- 设置开启消息事务 -->  <property name=*"sessionTransacted"* value=*"true"*/>  </bean>  <!-- 配置目标(点对点) -->  <bean id=*"queue"* class=*"org.apache.activemq.command.ActiveMQQueue"*>  <!-- 设置队列名称 -->  <constructor-arg value=*"spring-queue"*/>  </bean> |

### 接收消息（添加消息事务）

|  |
| --- |
| <!-- ############## 配置消息消费者 ################# -->  <!-- 配置JMS消息监听容器  connection-factory: 连接工厂  destination-type="queue|topic" 目标类型  acknowledge="transacted" 应答方式(事务)  -->  <jms:listener-container connection-factory=*"connectionFactory"*  destination-type=*"queue"*  acknowledge=*"transacted"*>  <jms:listener destination=*"spring-queue"* ref=*"myMessageListener"*/>  </jms:listener-container>  <!-- 配置消息监听器 -->  <bean id=*"myMessageListener"* class=*"cn.itcast.activemq.spring.MyMessageListener"*/>  </beans> |

注意：

1、自定消息监听器需要实现SessionAwareMessageListener接口(有消息事务)。

1. 用了消息事务时，当处理消息失败，默认重发6次，并把消息存放到

ActiveMQ.DLQ死信队列中。