**淘淘商城**

**第十天**

# 课程计划

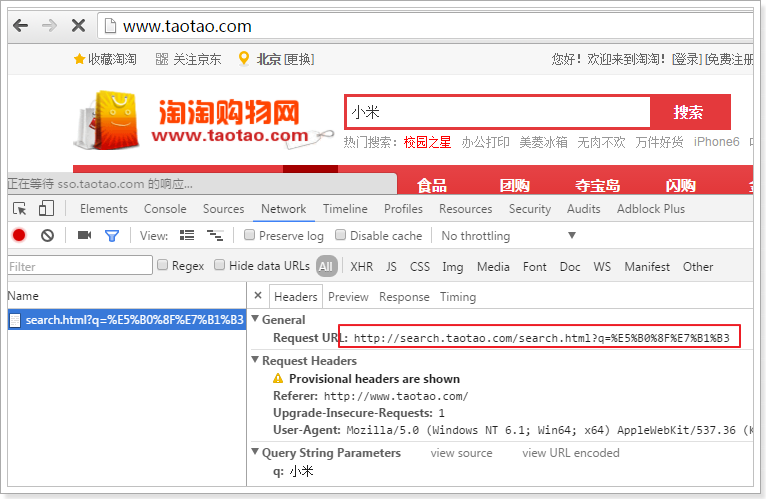
1. 实现搜索功能
2. 数据同步问题
3. ActiveMQ介绍
4. ActiveMQ使用
5. 整合到项目中

# 实现搜索功能

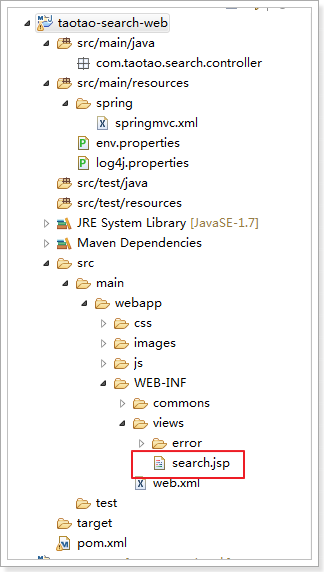
## 实现页面跳转

在首页可以搜索商品

点击商品搜索，发起请求到search.taotao.com，携带关键字



在search.taotao.com有search.jsp页面



### controller实现

在taotao-search-web实现

@Controller

@RequestMapping("search")

**public** **class** SearchController {

// http://search.taotao.com/search.html?q=%E5%B0%8F%E7%B1%B3

/\*\*

\* 搜索

\*

\* **@param** q

\* **@return**

\*/

@RequestMapping(method = RequestMethod.***GET***)

**public** String search(String q) {

**return** "search";

}

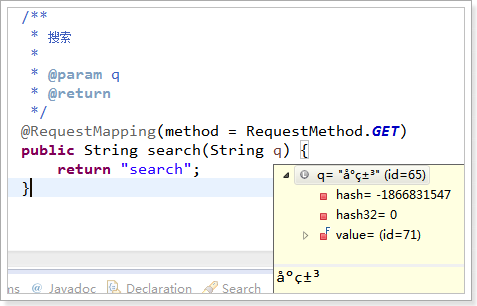
}

### 效果



## 问题

出现请求参数q的乱码问题



## 解决乱码问题

### 分析

用户搜索时候，关键字传递到后台出现乱码问题。

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

q = **new** String(q.getBytes("ISO-8859-1"));

1. 修改Tomcat的编码为UTF-8

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

### 改造Controller

对taotao-search-web的Controller的方法进行改造

@RequestMapping(method = RequestMethod.***GET***)

**public** String search(String q) {

// 解决乱码问题

**try** {

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

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** "search";

}

## 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

${totalPages}

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

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

(total+rows-1)/rows(了解)

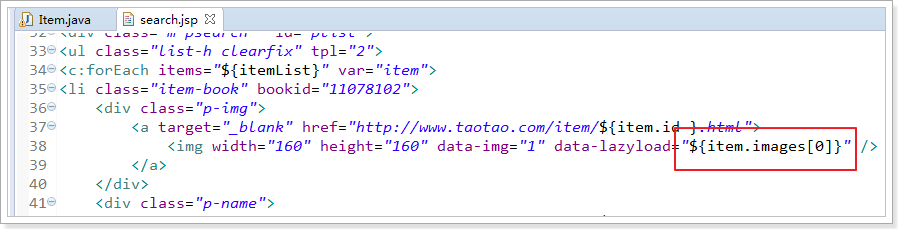
这样查询一次solr索引库，需要返回两个数据，一个是商品结果集list，另一个是数据总条数

有当前页码数page，但是没有页面显示数据条数rows

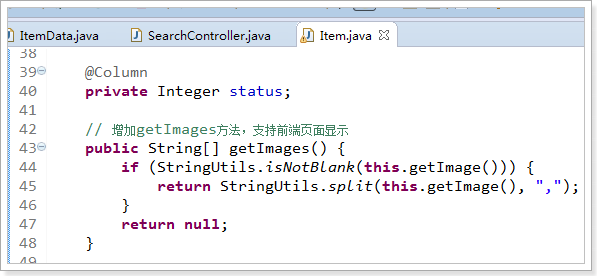
基于安全考虑，这里不能让用户指定rows。否则用户可以指定一次查询很多商品数据。这样返回的数据量会变得很大，如果恶意访问会造成很大的访问压力，很不安全。所以由我们自己指定。

我们指定16，一页显示16条

前端页面展示需要Item的images属性



修改Item，增加images属性

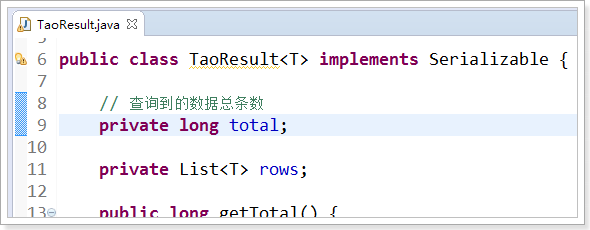


## 后端开发

### 使用TaoResult封装数据

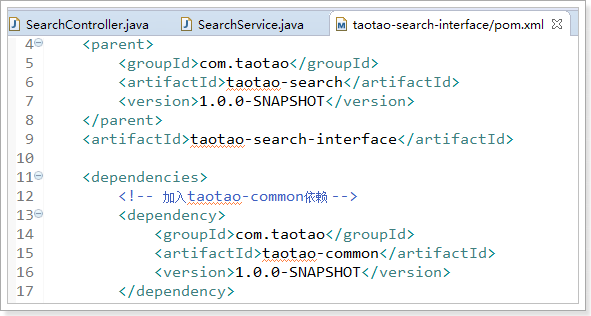
search的服务层需要向表现层传递商品结果集list和查询数据总条数

以前在taotao-common创建了TaoResult，里面包含结果集和数据总条数两个属性，可以直接使用。



#### 添加依赖

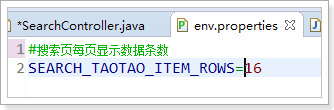
taotao-search-interface添加taotao-common依赖



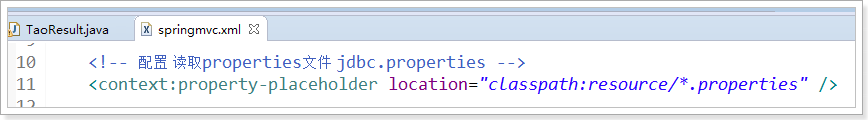
### 实现Controller

在taotao-search-web实现

添加配置文件



加载配置文件



Controller编写

@Controller

@RequestMapping("search")

**public** **class** SearchController {

@Autowired

**private** SearchService searchService;

@Value("${TAOTAO\_SEARCH\_ITEM\_ROWS}")

**private** Integer rows;

// http://search.taotao.com/search.html?q=apple

/\*\*

\* 搜索商品

\*

\* **@return**

\*/

@RequestMapping(method = RequestMethod.***GET***)

**public** String search(Model model, @RequestParam("q") String query,

@RequestParam(value = "page", defaultValue = "1") Integer page) {

**try** {

query = **new** String(query.getBytes("ISO-8859-1"));

} **catch** (Exception e) {

e.printStackTrace();

}

TaoResult<Item> taoResult = **this**.searchService.search(query, page, **this**.rows);

// 把数据放到模型中，页面展示

// 搜索关键词

model.addAttribute("query", query);

// 搜索结果集list

model.addAttribute("itemList", taoResult.getRows());

// 当前页码数

model.addAttribute("page", page);

// 总页数,查询到的数据总数和每页显示条数进行计算

**long** total = taoResult.getTotal();

// long pages = total % 16 == 0 ? total / 16 : (total / 16) + 1;

**long** pages = (total + **this**.rows - 1) / **this**.rows;

model.addAttribute("totalPages", pages);

**return** "search";

}

}

### 实现Service接口

在taotao-search-interface实现

**public** **interface** SearchService {

/\*\*

\* 根据关键词搜索

\*

\* **@param** query

\* **@param** page

\* **@param** rows

\* **@return**

\*/

TaoResult<Item> search(String query, Integer page, Integer rows);

}

### 实现Service接口方法

在taotao-search-service实现

@Service

**public** **class** SearchServiceImpl **implements** SearchService {

@Autowired

**private** CloudSolrServer cloudSolrServer;

@Override

**public** TaoResult<Item> search(String query, Integer page, Integer rows) {

// 封装查询对象

SolrQuery solrQuery = **new** SolrQuery();

// 设置查询语句

**if** (StringUtils.*isNotBlank*(query)) {

solrQuery.setQuery("item\_title:" + query + " AND item\_status:1");

} **else** {

solrQuery.setQuery("item\_status:1");

}

// 设置分页

solrQuery.setStart((page - 1) \* rows);

solrQuery.setRows(rows);

// 高亮

solrQuery.setHighlight(**true**);

solrQuery.addHighlightField("item\_title");

solrQuery.setHighlightSimplePre("<font color='red'>");

solrQuery.setHighlightSimplePost("</font>");

// 声明返回结果对象taoResult

TaoResult<Item> taoResult = **new** TaoResult<>();

**try** {

// 执行查询

QueryResponse response = **this**.cloudSolrServer.query(solrQuery);

SolrDocumentList results = response.getResults();

// 获取高亮数据

Map<String, Map<String, List<String>>> map = response.getHighlighting();

// 解析结果集

// 声明存放商品的集合

List<Item> list = **new** ArrayList<>();

**for** (SolrDocument solrDocument : results) {

Item item = **new** Item();

// 解析Document

// 商品id

item.setId(Long.*parseLong*(solrDocument.get("id").toString()));

// 获取高亮的数据

List<String> hlist = map.get(solrDocument.get("id").toString()).get("item\_title");

// 商品title

**if** (hlist != **null** && hlist.size() > 0) {

item.setTitle(hlist.get(0));

} **else** {

item.setTitle(solrDocument.get("item\_title").toString());

}

// 商品图片image

item.setImage(solrDocument.get("item\_image").toString());

// 商品价格price

item.setPrice(Long.*parseLong*(solrDocument.get("item\_price").toString()));

// 商品cid

item.setCid(Long.*parseLong*(solrDocument.get("item\_cid").toString()));

// 把封装好的商品数据放到集合中

list.add(item);

}

// 封装返回数据taoResult

// 设置结果集

taoResult.setRows(list);

// 设置查询的数据总条数

taoResult.setTotal(results.getNumFound());

**return** taoResult;

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

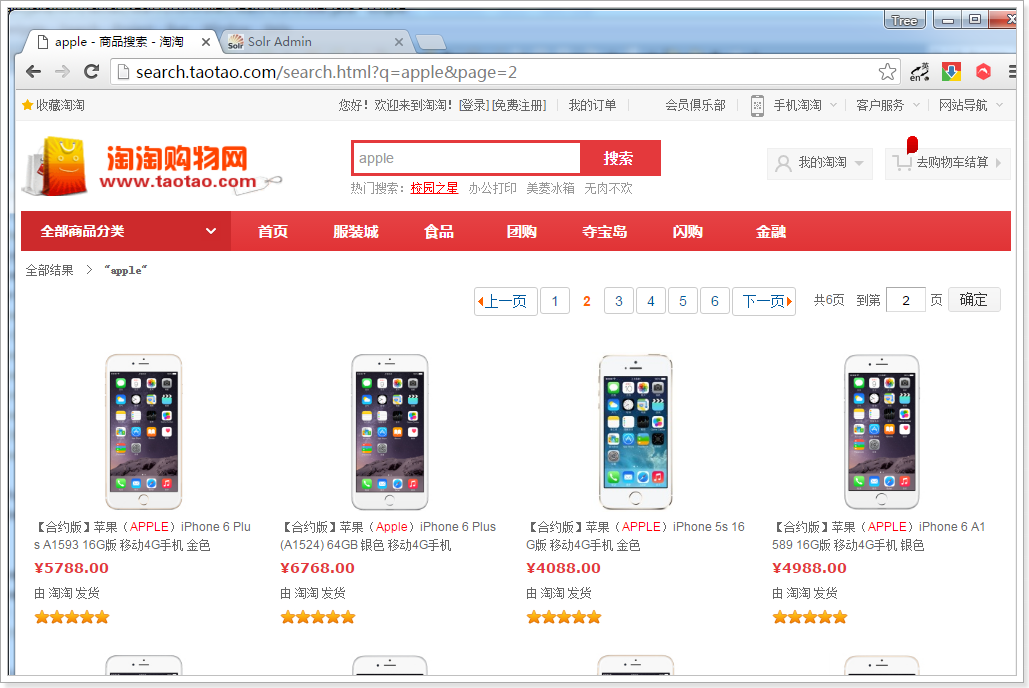
}

// 如果查询有异常，就返回一个空的结果

**return** taoResult;

}

## 效果



## 搜索功能数据同步的问题

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

同步的问题：

两边都进行商品数据的保存，会存在数据同步的问题

后台增加商品数据的时候，索引库没有同步更新

解决方案：

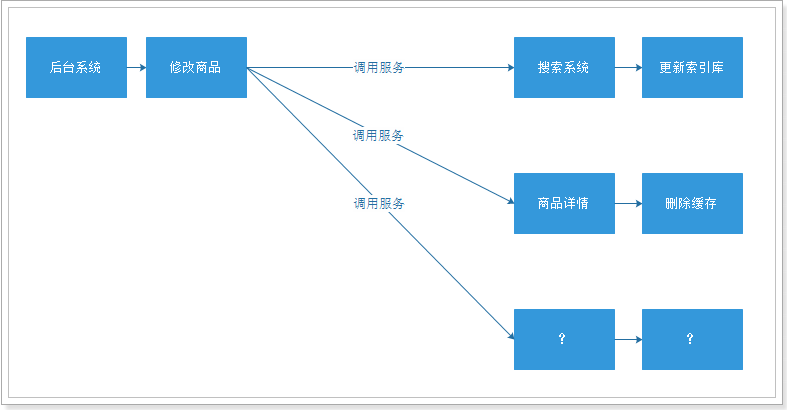
可以在taotao-search编写保存数据到solr索引库的方法，taotao-manager进行商品保存的时候，调用taotao-search的服务，更新索引库

# ActiveMQ介绍

## 分析现在解决数据同步存在的问题

现在的解决方案：

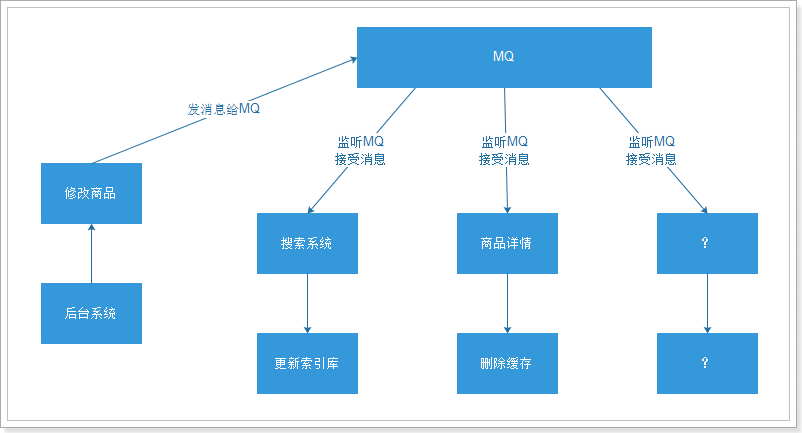
1. 在搜索系统中，提供一个服务，这个服务能新增和更新索引库数据
2. 后台在新增或更新，调用搜索系统的这个服务。



存在的问题：

1. 耦合性很高，每次都需要两边都进行修改
2. 随着业务的发展，业务逻辑越来越复杂，后台系统需要调用的服务会越来越多，之间的调用越来越复杂，难以管理。

使用MQ的解决方案



1. 后台系统进行商品修改，通知MQ
2. 其他系统监听MQ是否有消息。如果没有消息，持续监听，如果有消息，处理相关业务逻辑处理

存在的问题：

1. （MQ）要是挂了，所有相关服务都挂了
2. （MQ）性能瓶颈的问题，在使用时，尽量减少消息内容大小

## ActiveMQ

MQ ： Message Queue 就是消息队列

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JMS1.1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消息的传递有两种类型：

一种是点对点的，即一个生产者和一个消费者一一对应；

另一种是发布/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



## JMS介绍

JMS即Java消息服务（Java Message Service）应用程序接口，是一个Java平台中关于面向消息中间件（MOM）的API，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 -- Java原始值的数据流

　　• MapMessage--一套名称-值对

**• TextMessage--一个字符串对象**

　　• ObjectMessage--一个序列化的 Java对象

　　• BytesMessage--一个字节的数据流

# ActiveMQ安装

## ActiveMQ下载

下载地址

http://activemq.apache.org/



## ActiveMQ安装

### 环境准备

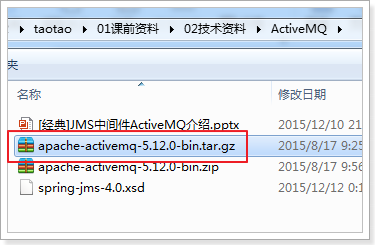
ActiveMQ是一个java语言开发的MQ服务器。在安装之前必须安装jdk。使用jdk1.7以上版本。

安装环境：

CentOS6.5

jdk1.7

安装文件



### 上传

上传ActiveMQ安装文件

alt+p进入SFTP界面，进行上传

移动到/usr/local中

[root@itcast-01 ~]# mv apache-activemq-5.12.0-bin.tar.gz /usr/local/

### 解压

进入/usr/local/目录

[root@itcast-01 ~]# cd /usr/local/

解压文件

[root@itcast-01 local]# tar -zxvf apache-activemq-5.12.0-bin.tar.gz

### 启动ActiveMQ

进入解压后的文件夹的bin/linux-x86-32/目录

[root@itcast-01 local]# cd apache-activemq-5.12.0/bin/linux-x86-32/

启动ActiveMQ

[root@itcast-01 linux-x86-32]# ./activemq start

### 访问

在宿主机打开浏览器访问

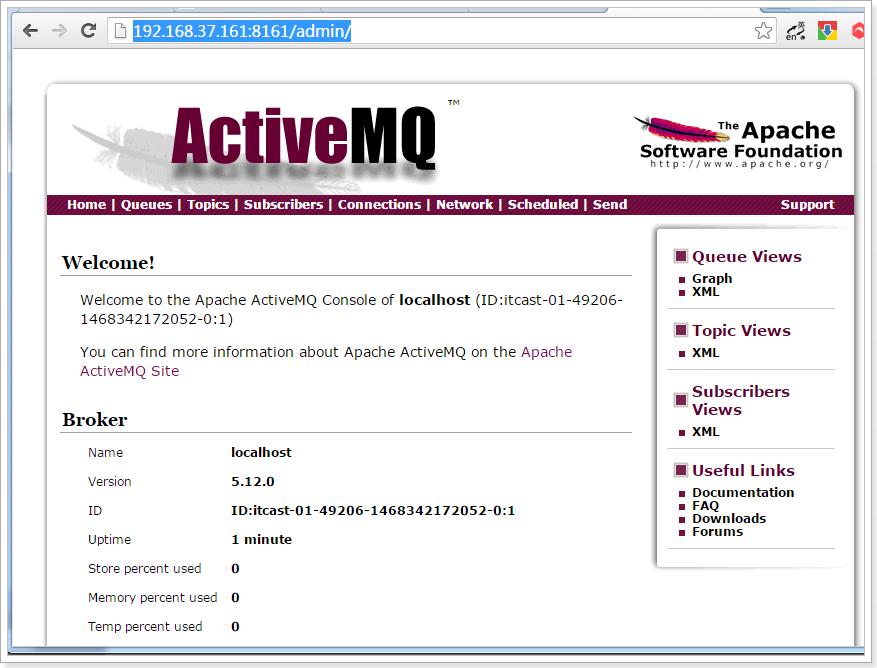
访问地址：

http://192.168.37.161:8161/admin/

账号：admin

密码：admin

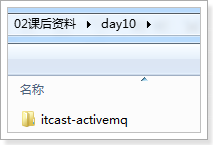
效果



# ActiveMQ使用

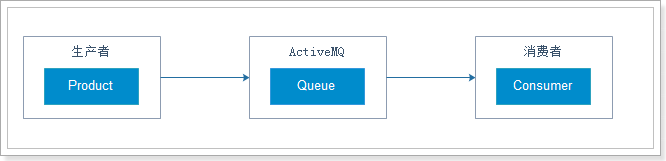
## 导入工程

导入itcast-activemq工程



## queue的使用

点对点模式



Product：消息的生产者 ，就是发消息的

Queue：消息队列，存放消息，和提供消息

Consumer：消息的消费者，就是收消息和消费消息

### Product

开发步骤：

1. 创建ActiveMQConnectionFactory，创建工厂需要ip和端口号，指定的是tcp协议

2. 从工厂中获取连接

3. 使用连接的start方法，开启连接

4. 使用连接获取session对象

5. 使用session创建Destination，就是创建queue，（可以设置queue（点对点）和topic（订阅）

6. 使用session创建Product，消息的是生产者

7. 创建消息对象TextMessage，设置消息

8. 发送消息

9. 释放资源，关闭Product、session、connection

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

// 1. 创建连接工厂ActiveMQConnectionFactory，需要ip和端口61616

ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory("tcp://192.168.37.161:61616");

// 2. 从连接工厂中创建连接对象

Connection connection = factory.createConnection();

// 3. 执行start方法开启连接

connection.start();

// 4. 从连接中创建session对象

// 第一个参数，是否开启事务，JTA分布式事务

// 第二个参数，是否自动应答，如果第一个参数为true，第二个参数失效

Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

// 5. 从session中创建Destination对象，设置queue名称（有两种类型queue和topic）

Queue queue = session.createQueue("test-queue");

// 6. 从session中创建Product对象

MessageProducer producer = session.createProducer(queue);

// 7. 创建消息对象

TextMessage textMessage = **new** ActiveMQTextMessage();

// 设置消息内容

textMessage.setText("开始发消息！");

// 8. 发送消息

producer.send(textMessage);

// 9. 关闭session、连接

producer.close();

session.close();

connection.close();

}

### Consumer

开发步骤

1. 创建连接工厂ActiveMQConnectionFactory，需要ip和端口
2. 从连接工厂中创建连接对象
3. 执行start方法开始连接
4. 从连接中创建session对象
5. 从session中创建Destination对象，设置queue名称（有两种类型queue和topic）
6. 从session中创建Consumer对象
7. 接收消息
8. 打印结果
9. 关闭session、连接

#### 直接获取消息

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

// 1. 创建连接工厂ActiveMQConnectionFactory，需要ip和端口61616

ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory("tcp://192.168.37.161:61616");

// 2. 使用工厂创建连接

Connection connection = factory.createConnection();

// 3. 使用start开启连接

connection.start();

// 4. 从连接中创建session对象

Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

// 5. 从session中创建Destination对象，设置queue名字

Queue queue = session.createQueue("test-queue");

// 6. 从session中创建Consumer

MessageConsumer consumer = session.createConsumer(queue);

// 7， 接收消息,直接获取

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

// 消息超时时间是20秒

Message message = consumer.receive(20000);

// 如果消息为空，则跳出死循环

**if** (message == **null**) {

**break**;

}

// 8. 打印消息

**if** (message **instanceof** TextMessage) {

// 获取消息

TextMessage textMessage = (TextMessage) message;

// 打印

System.***out***.println(textMessage.getText()); }

}

// 9. 关闭session、连接等

consumer.close();

session.close();

connection.close();

}

#### 使用监听器

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

// 1. 创建连接工厂ActiveMQConnectionFactory，需要ip和端口61616

ActiveMQConnectionFactory factory = **new** ActiveMQConnectionFactory("tcp://192.168.37.161:61616");

// 2. 使用工厂创建连接

Connection connection = factory.createConnection();

// 3. 使用start开启连接

connection.start();

// 4. 从连接中创建session对象

Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

// 5. 从session中创建Destination对象，设置queue名字

Queue queue = session.createQueue("test-queue");

// 6. 从session中创建Consumer

MessageConsumer consumer = session.createConsumer(queue);

// // 7， 接收消息,直接获取

// while (true) {

// // 消息超时时间是20秒

// Message message = consumer.receive(20000);

// // 如果消息为空，则跳出死循环

// if (message == null) {

// break;

// }

//

// // 8. 打印消息

// if (true) {

// if (message instanceof TextMessage) {

// // 获取消息

// TextMessage textMessage = (TextMessage) message;

// // 打印

// System.out.println(textMessage.getText());

// }

// }

// }

// 7.接收消息

// 监听器的方式实际上是开启了一个新的线程，专门处理消息的接受

// 现在的情况是，主线程执行完就结束了，新的线程也跟着没了

consumer.setMessageListener(**new** MessageListener() {

@Override

**public** **void** onMessage(Message message) {

**if** (message **instanceof** TextMessage) {

// 获取消息

TextMessage textMessage = (TextMessage) message;

**try** {

// 打印

System.***out***.println(textMessage.getText());

} **catch** (JMSException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

});

// 让主线程等待一会，监听器能够有时间执行

Thread.*sleep*(10000);

// 9. 关闭session、连接等

consumer.close();

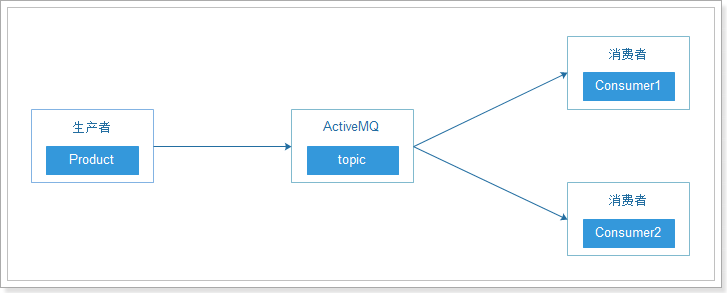
session.close();

connection.close();

}

## topic的使用

发布/订阅模式



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。

### Product

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

// 1. 创建连接工厂ActiveMQConnectionFactory

ActiveMQConnectionFactory activeMQConnectionFactory = **new** ActiveMQConnectionFactory(

"tcp://192.168.37.161:61616");

// 2. 使用工厂创建连接

Connection connection = activeMQConnectionFactory.createConnection();

// 3. 使用start方法开启连接

connection.start();

// 4. 从连接创建session

Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

// 5. 从session创建Destination对象，设置topic名称

Topic topic = session.createTopic("test-topic");

// 6. 从session创建Product

MessageProducer producer = session.createProducer(topic);

// 7. 创建消息对象

TextMessage textMessage = **new** ActiveMQTextMessage();

textMessage.setText("topic消息");

// 8. 发送消息

producer.send(textMessage);

// 9. 关闭session、连接等

producer.close();

session.close();

connection.close();

}

}

### Consumer1

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

// 1. 创建连接工厂ActiveMQConnectionFactory

ActiveMQConnectionFactory activeMQConnectionFactory = **new** ActiveMQConnectionFactory(

"tcp://192.168.37.161:61616");

// 2. 从连接工厂创建连接

Connection connection = activeMQConnectionFactory.createConnection();

// 3. 使用start方法开启连接

connection.start();

// 4. 从连接创建session对象

Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

// 5. 从session创建安Destination，设置topic名称

Topic topic = session.createTopic("test-topic");

// 6. 从session创建Consumer对象

MessageConsumer consumer = session.createConsumer(topic);

// 7. 接收消息，直接接受

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

Message message = consumer.receive(20000);

**if** (message == **null**) {

**break**;

}

**if** (message **instanceof** TextMessage) {

TextMessage textMessage = (TextMessage) message;

// 8. 打印消息

System.***out***.println(textMessage.getText());

}

}

// 9. 关闭session、消息等

consumer.close();

session.close();

connection.close();

}

### Consumer2

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

// 1. 创建连接工厂ActiveMQConnectionFactory

ActiveMQConnectionFactory activeMQConnectionFactory = **new** ActiveMQConnectionFactory(

"tcp://192.168.37.161:61616");

// 2. 从连接工厂创建连接

Connection connection = activeMQConnectionFactory.createConnection();

// 3. 使用start方法开启连接

connection.start();

// 4. 从连接创建session对象

Session session = connection.createSession(**false**, Session.***AUTO\_ACKNOWLEDGE***);

// 5. 从session创建安Destination，设置topic名称

Topic topic = session.createTopic("test-topic");

// 6. 从session创建Consumer对象

MessageConsumer consumer = session.createConsumer(topic);

// 7. 接收消息，直接接受

// while (true) {

// Message message = consumer.receive(20000);

//

// if (message == null) {

// break;

// }

//

// if (message instanceof TextMessage) {

// TextMessage textMessage = (TextMessage) message;

// // 8. 打印消息

// System.out.println(textMessage.getText());

// }

// }

// 7.接受消息，使用监听器

consumer.setMessageListener(**new** MessageListener() {

@Override

**public** **void** onMessage(Message message) {

**if** (message **instanceof** TextMessage) {

TextMessage textMessage = (TextMessage) message;

**try** {

// 打印消息

System.***out***.println(textMessage.getText());

} **catch** (JMSException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

});

// 等待监听器执行

Thread.*sleep*(10000);

// 9. 关闭session、消息等

consumer.close();

session.close();

connection.close();

}

## activeMQ整合spring

### 加入依赖

itcast-activemq加入依赖

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.taotao</groupId>

<artifactId>taotao-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast.activemq</groupId>

<artifactId>itcast-activemq</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- 加入ActiveMQ依赖 -->

<dependency>

<groupId>org.apache.activemq</groupId>

<artifactId>activemq-all</artifactId>

</dependency>

<!-- 加入spring-jms依赖 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jms</artifactId>

</dependency>

<!-- Spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

</dependency>

</dependencies>

</project>

### 消息发送

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

**public** **class** Product {

**public** **static** **void** main(String[] args) {

// 1. 创建spring容器

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(

"applicationContext-activemq.xml");

// 2. 从容器中获取JMSTemplate对象

JmsTemplate jmsTemplate = context.getBean(JmsTemplate.**class**);

// 3. 从容器中获取Destination对象

Destination destination = context.getBean(Destination.**class**);

// 4. 使用JMSTemplate发送消息

jmsTemplate.send(destination, **new** MessageCreator() {

@Override

**public** Message createMessage(Session session) **throws** JMSException {

// 创建消息对象

TextMessage textMessage = **new** ActiveMQTextMessage();

// 设置消息内容

textMessage.setText("spring整合ActiveMQ");

// 打印消息

System.***out***.println(textMessage.getText());

**return** textMessage;

}

});

}

}

### 消息接受

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DefaultMessageListenerContainer对象，引用MessageListener 实现类对象接收消息。

**public** **class** MyMessageListener **implements** MessageListener {

@Override

**public** **void** onMessage(Message message) {

**if** (message **instanceof** TextMessage) {

TextMessage textMessage = (TextMessage) message;

**try** {

// 获取消息内容

String msg = textMessage.getText();

// 打印消息

System.***out***.println("接受消息:" + msg);

} **catch** (JMSException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

### queue方式配置spring

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

xmlns:p=*"http://www.springframework.org/schema/p"* xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"* xmlns:jms=*"http://www.springframework.org/schema/jms"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-4.0.xsd*

*http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>

<!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供 -->

<bean id=*"targetConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"* value=*"tcp://192.168.37.161:61616"* />

</bean>

<!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->

<bean id=*"connectionFactory"* class=*"org.springframework.jms.connection.SingleConnectionFactory"*>

<!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->

<property name=*"targetConnectionFactory"* ref=*"targetConnectionFactory"* />

</bean>

<!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->

<bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

</bean>

<!--这个是队列目的地，点对点的 -->

<bean id=*"queueDestination"* class=*"org.apache.activemq.command.ActiveMQQueue"*>

<constructor-arg>

<value>queue</value>

</constructor-arg>

</bean>

<!--这个是主题目的地，一对多的 -->

<!-- <bean id="topicDestination" class="org.apache.activemq.command.ActiveMQTopic"> -->

<!-- <constructor-arg value="topic" /> -->

<!-- </bean> -->

<!-- messageListener实现类 -->

<bean id=*"myMessageListener"* class=*"cn.itcast.activemq.spring.MyMessageListener"*></bean>

<!-- 配置一个jsm监听容器 -->

<bean id=*"jmsContainer"* class=*"org.springframework.jms.listener.DefaultMessageListenerContainer"*>

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

<property name=*"destination"* ref=*"queueDestination"* />

<property name=*"messageListener"* ref=*"myMessageListener"* />

</bean>

</beans>

### topic方式配置spring

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-4.0.xsd*

*http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>

<!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供 -->

<bean id=*"targetConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"* value=*"tcp://192.168.37.161:61616"* />

</bean>

<!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->

<bean id=*"connectionFactory"*

class=*"org.springframework.jms.connection.SingleConnectionFactory"*>

<!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->

<property name=*"targetConnectionFactory"* ref=*"targetConnectionFactory"* />

</bean>

<!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->

<bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

</bean>

<!--这个是队列目的地，点对点的 -->

<!-- <bean id="queueDestination" class="org.apache.activemq.command.ActiveMQQueue"> -->

<!-- <constructor-arg> -->

<!-- <value>queue</value> -->

<!-- </constructor-arg> -->

<!-- </bean> -->

<!--这个是主题目的地，一对多的 -->

<bean id=*"topicDestination"* class=*"org.apache.activemq.command.ActiveMQTopic"*>

<constructor-arg value=*"topic"* />

</bean>

<!-- messageListener实现类 -->

<bean id=*"myMessageListener"* class=*"cn.itcast.activemq.spring.MyMessageListener"*></bean>

<!-- messageListener实现类 -->

<bean id=*"myMessageListener2"* class=*"cn.itcast.activemq.spring.MyMessageListener2"*></bean>

<!-- 配置一个jsm监听容器 -->

<bean id=*"jmsContainer"*

class=*"org.springframework.jms.listener.DefaultMessageListenerContainer"*>

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

<property name=*"destination"* ref=*"topicDestination"* />

<property name=*"messageListener"* ref=*"myMessageListener"* />

</bean>

<!-- 配置一个jsm监听容器 -->

<bean id=*"jmsContainer2"*

class=*"org.springframework.jms.listener.DefaultMessageListenerContainer"*>

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

<property name=*"destination"* ref=*"topicDestination"* />

<property name=*"messageListener"* ref=*"myMessageListener2"* />

</bean>

</beans>

# 项目整合ActiveMQ

使用ActiveMQ的方式解决数据同步的问题

需要在taotao-manager发送消息

在taotao-search接收并消费消息

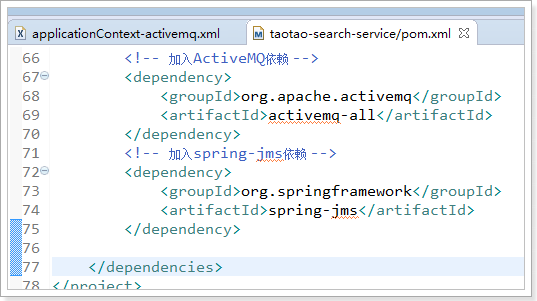
## 加入依赖

消息的发送方和接收方都需要添加依赖

### taotao-manager-service加入依赖



### taotao-search-service



## 消息发布改造

### 消息生产者整合ActiveMQ

消息生产者只需要发送消息

需要把JMSTemplate和Destination交给spring进行管理

在taotao-manager-service添加配置文件

applicationContext-activemq.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-4.0.xsd*

*http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>

<!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供 -->

<bean id=*"targetConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"* value=*"tcp://192.168.37.161:61616"* />

</bean>

<!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->

<bean id=*"connectionFactory"*

class=*"org.springframework.jms.connection.SingleConnectionFactory"*>

<!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->

<property name=*"targetConnectionFactory"* ref=*"targetConnectionFactory"* />

</bean>

<!-- Spring提供的JMS工具类，它可以进行消息发送、接收等 -->

<bean id=*"jmsTemplate"* class=*"org.springframework.jms.core.JmsTemplate"*>

<!-- 这个connectionFactory对应的是我们定义的Spring提供的那个ConnectionFactory对象 -->

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

</bean>

<!--这个是主题目的地，一对多的 -->

<bean id=*"topicDestination"* class=*"org.apache.activemq.command.ActiveMQTopic"*>

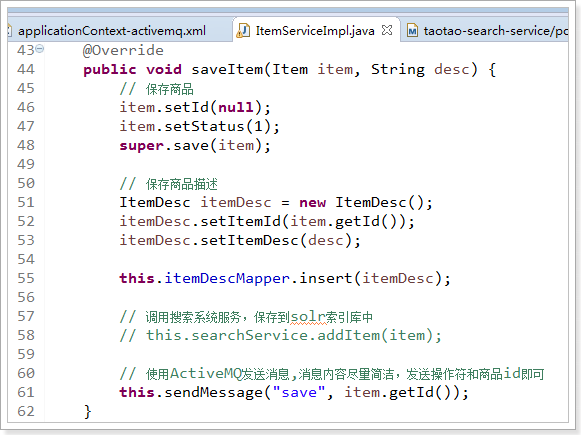
<constructor-arg value=*"topic"* />

</bean>

</beans>

### 改造新增商品

改造taotao-manager-service新增方法加入发送消息逻辑



发送消息方法实现：

**private** **static** **final** ObjectMapper ***MAPPER*** = **new** ObjectMapper();

// 注入JMSTemplate

@Autowired

**private** JmsTemplate jmsTemplate;

// 注入Destination

@Autowired

**private** Destination destination;

// 通用发送消息方法，第一个参数是操作符，第二个参数是商品id

**private** **void** sendMQ(**final** String type, **final** Long itemId) {

**this**.jmsTemplate.send(destination, **new** MessageCreator() {

@Override

**public** Message createMessage(Session session) **throws** JMSException {

// 声明消息

TextMessage textMessage = **new** ActiveMQTextMessage();

// 构建消息内容

// 使用json格式的数据封装需要传递的消息

Map<String, Object> map = **new** HashMap<>();

// 操作符

map.put("type", type);

// 商品id

map.put("itemId", itemId);

**try** {

// 把map转为接送格式的数据

String json = ***MAPPER***.writeValueAsString(map);

// 设置消息内容

textMessage.setText(json);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** textMessage;

}

});

}

## 消息消费改造

### 添加消息监听

在taotao-search-service添加

**public** **class** MyMessageListener **implements** MessageListener {

**private** **final** **static** ObjectMapper ***MAPPER*** = **new** ObjectMapper();

@Autowired

**private** SearchService searchService;

@Override

**public** **void** onMessage(Message message) {

**if** (message **instanceof** TextMessage) {

// 获取消息对象

TextMessage textMessage = (TextMessage) message;

**try** {

// 获取消息内容

String msg = textMessage.getText();

**if** (StringUtils.*isNotBlank*(msg)) {

// 解析消息

// 把json转为jsonNode

JsonNode jsonNode = ***MAPPER***.readTree(msg);

// 获取操作符

String type = jsonNode.get("type").asText();

// 获取商品id

**long** itemId = jsonNode.get("itemId").asLong();

// 更新索引库

**this**.searchService.saveItem(item);

}

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

### Service增加保存索引库方法

在taotao-search-service增加

@Autowired

**private** ItemMapper itemMapper;

@Override

**public** **void** saveItem(**long** itemId) {

// 1. 调用商品服务，根据商品id获取商品信息

Item item = **this**.itemMapper.selectByPrimaryKey(itemId);

// 2. 根据获取到的数据，保存索引库

SolrInputDocument document = **new** SolrInputDocument();

// 商品id

document.setField("id", item.getId().toString());

// 商品标题

document.setField("item\_title", item.getTitle());

// 商品价格

document.setField("item\_price", item.getPrice());

// 商品图片

document.setField("item\_image", item.getImage());

// 商品类目id

document.setField("item\_cid", item.getCid());

// 商品状态

document.setField("item\_status", item.getStatus());

**try** {

// 保存到索引库中

**this**.cloudSolrServer.add(document);

**this**.cloudSolrServer.commit();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

### 消息消费者整合ActiveMQ

消费者只需要消费消息

需要有Destination和监听、监听容器

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

xmlns:p=*"http://www.springframework.org/schema/p"* xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"* xmlns:jms=*"http://www.springframework.org/schema/jms"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*

*http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-4.0.xsd*

*http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>

<!-- 真正可以产生Connection的ConnectionFactory，由对应的 JMS服务厂商提供 -->

<bean id=*"targetConnectionFactory"* class=*"org.apache.activemq.ActiveMQConnectionFactory"*>

<property name=*"brokerURL"* value=*"tcp://192.168.37.161:61616"* />

</bean>

<!-- Spring用于管理真正的ConnectionFactory的ConnectionFactory -->

<bean id=*"connectionFactory"* class=*"org.springframework.jms.connection.SingleConnectionFactory"*>

<!-- 目标ConnectionFactory对应真实的可以产生JMS Connection的ConnectionFactory -->

<property name=*"targetConnectionFactory"* ref=*"targetConnectionFactory"* />

</bean>

<!--这个是主题目的地，一对多的 -->

<bean id=*"topicDestination"* class=*"org.apache.activemq.command.ActiveMQTopic"*>

<constructor-arg value=*"topic"* />

</bean>

<!-- messageListener实现类 -->

<bean id=*"MyMessageListener"* class=*"com.taotao.search.activemq.MyMessageListener"*></bean>

<!-- 配置一个jsm监听容器 -->

<bean id=*"jmsContainer"* class=*"org.springframework.jms.listener.DefaultMessageListenerContainer"*>

<property name=*"connectionFactory"* ref=*"connectionFactory"* />

<property name=*"destination"* ref=*"topicDestination"* />

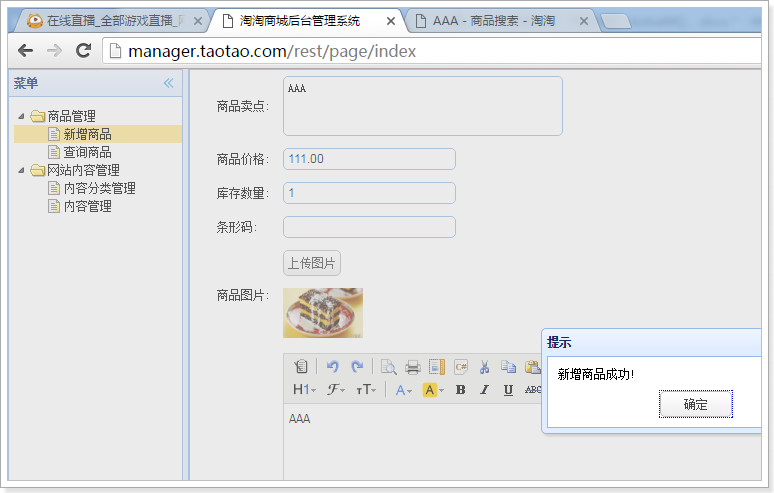
<property name=*"messageListener"* ref=*"MyMessageListener"* />

</bean>

</beans>

## 效果

后台修改商品



搜索系统就可以查到

