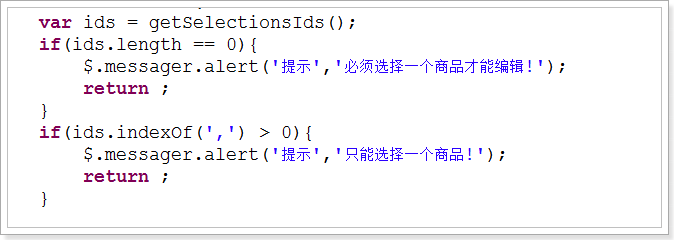
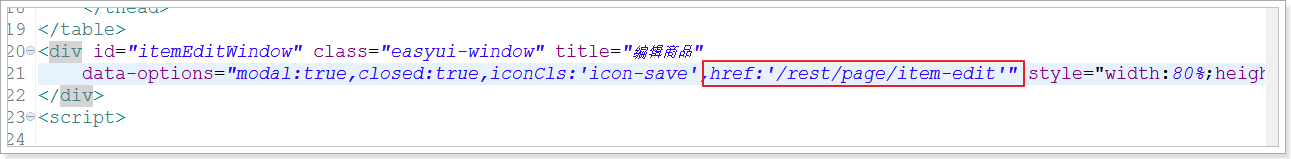
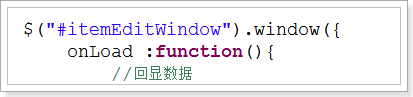
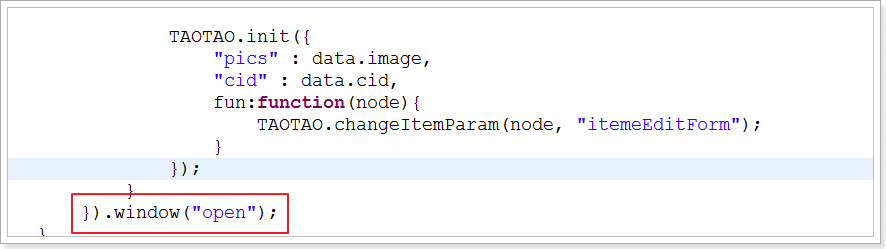
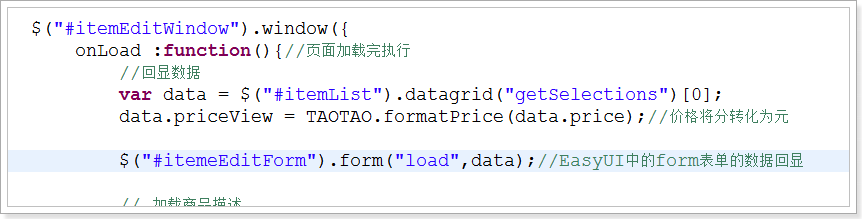
# 今日大纲

1. 实现商品的编辑
2. 实现商品的规格参数功能
3. 搭建前台系统
4. 实现首页商品类目的显示

# 商品的编辑

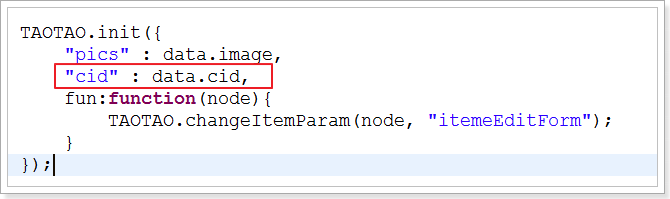
数据的编辑核心是：数据回显。

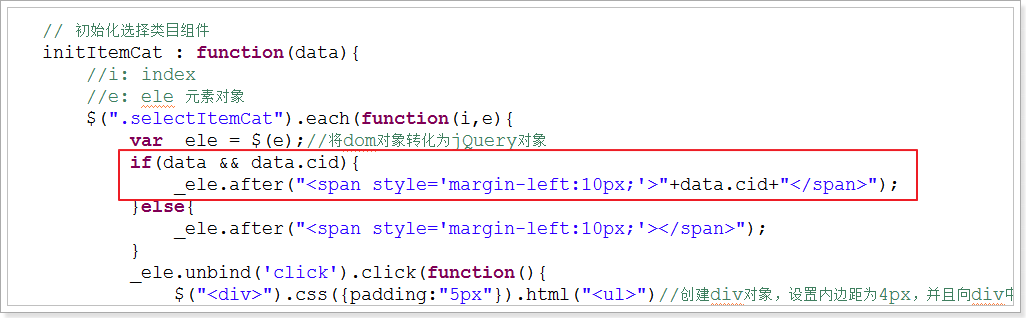
## 编辑按钮事件

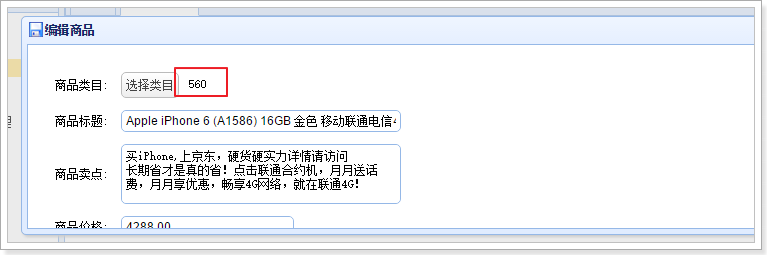
1. 判断选中的行数  
   
2. 弹出window  
     
     
   
3. 加载编辑页面，在页面加载完时完成回显  
   

## 自定义格式回显

### 商品类目回显

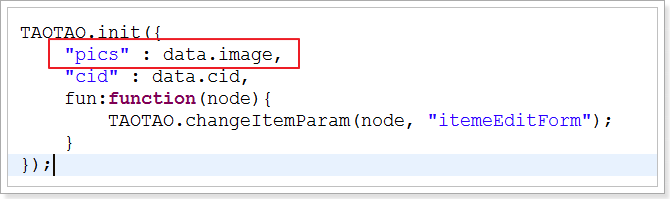


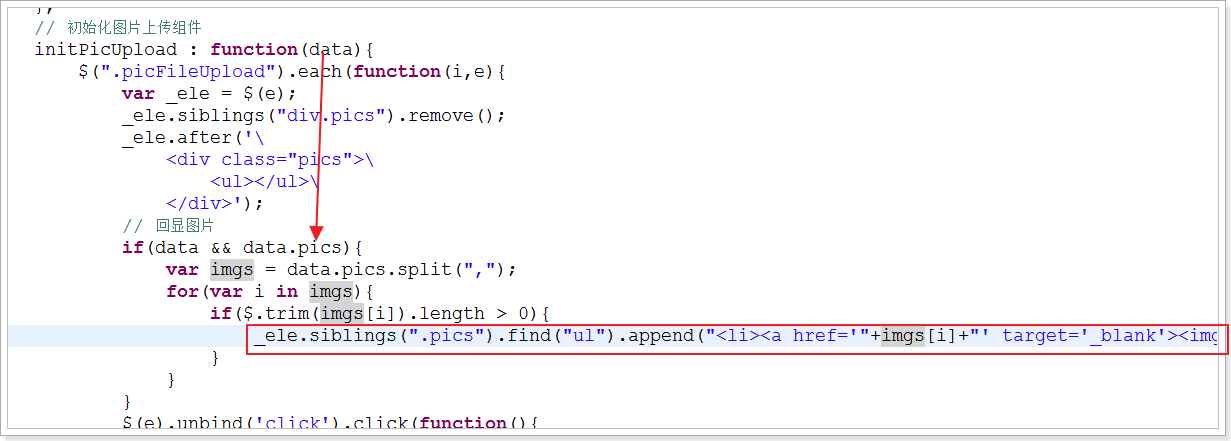




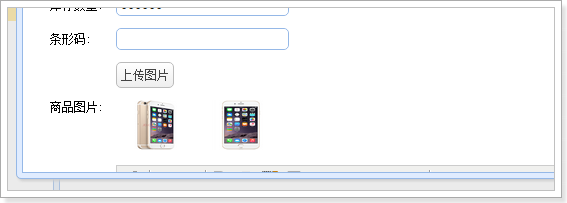
TODO：显示选中的类目的中文名称。

### 图片回显



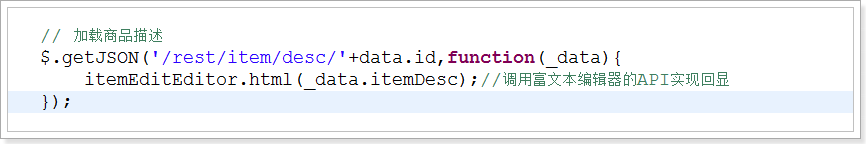


效果：



### 商品描述的回显

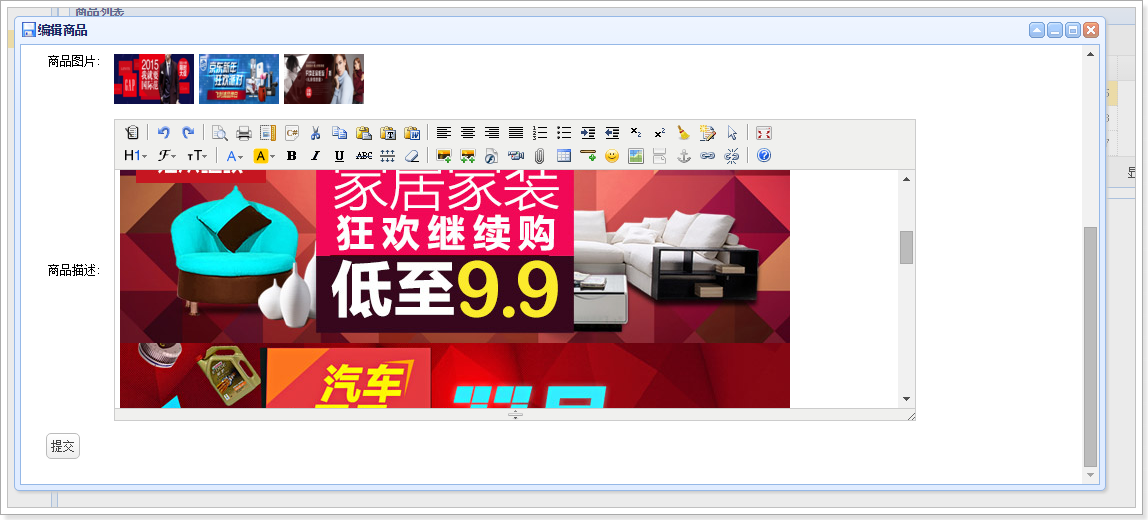
JS实现：



后台实现：

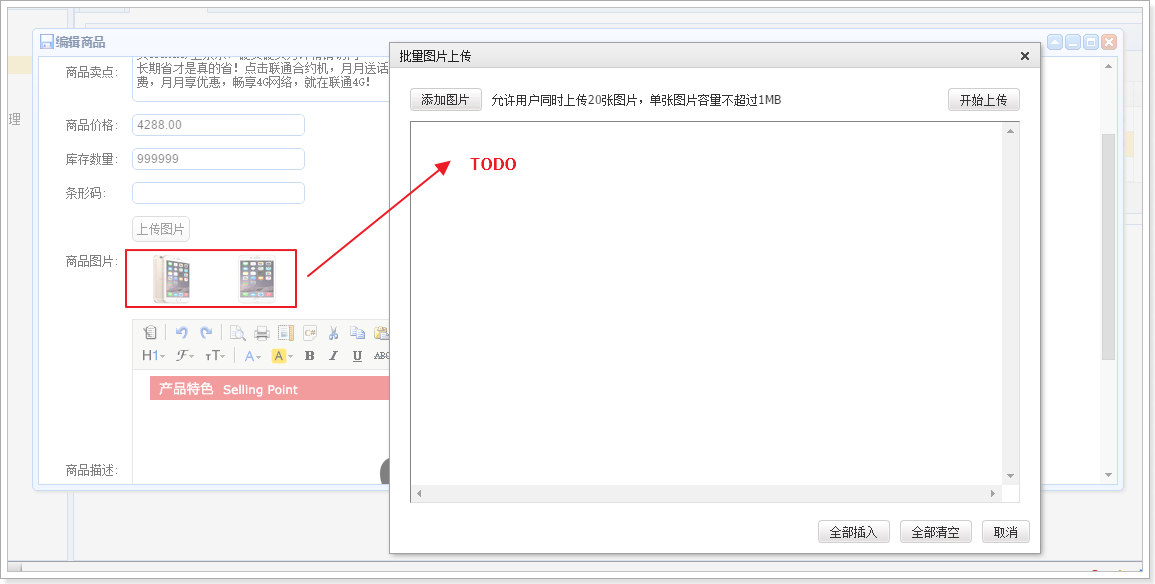


效果：



### 未实现TODO

编辑时图片回显：



思路：

1. 查找KindEditor是否有回显图片的API
2. 没有提供Api
   1. 扩展原有的Api支持回显（技术难度比较大）
   2. 功能实现的变通（在预览图片上添加删除图片，实现删除即可）

## 提交事件

**function** submitForm(){

**if**(!$('#itemeEditForm').form('validate')){

$.messager.alert('提示','表单还未填写完成!');

**return** ;

}

$("#itemeEditForm [name=price]").val(eval($("#itemeEditForm [name=priceView]").val()) \* 100);

itemEditEditor.sync();

**var** paramJson = [];

$("#itemeEditForm .params li").each(**function**(i,e){

**var** trs = $(e).find("tr");

**var** group = trs.eq(0).text();

**var** ps = [];

**for**(**var** i = 1;i<trs.length;i++){

**var** tr = trs.eq(i);

ps.push({

"k" : $.trim(tr.find("td").eq(0).find("span").text()),

"v" : $.trim(tr.find("input").val())

});

}

paramJson.push({

"group" : group,

"params": ps

});

});

paramJson = JSON.stringify(paramJson);

$("#itemeEditForm [name=itemParams]").val(paramJson);

//提交到后台的RESTful

$.ajax({

type: "PUT",

url: "/rest/item",

data: $("#itemeEditForm").serialize(),

statusCode : {

204 : **function**(){

$.messager.alert('提示','修改商品成功!','info',**func**tion(){

$("#itemEditWindow").window('close');

$("#itemList").datagrid("reload");

});

},

400 : function(){

$.messager.alert('提示','请求参数不合法!');

},

500 : **function**(){

$.messager.alert('提示','修改商品失败!');

}

}

});

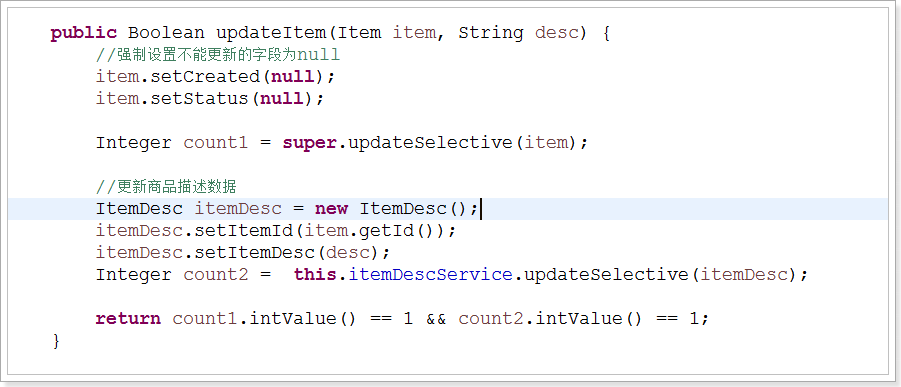
}

## 修改的后台实现

Controller：



Service：



# 商品规格参数

## 什么是商品规格参数



## 分析

同一个商品类目下的商品的规格参数的格式（内容）一样，只是具体的数据不同。

不同的类目的商品规格参数的格式是不同的。

## 如何实现？

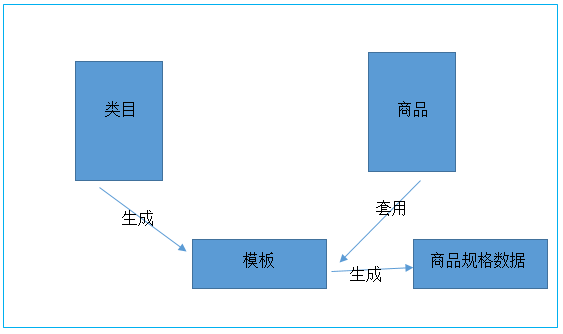
方案一：

针对每一个商品类目都创建一张表，来存储规格参数数据。

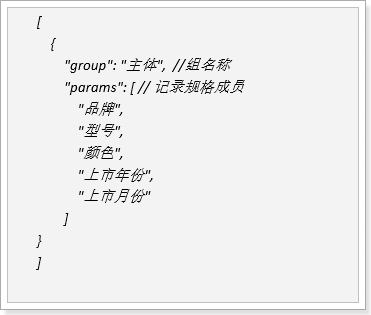
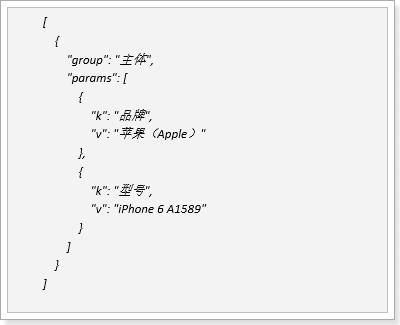
可行性: 不推荐。 维护的表太多了。

方案二：

使用模板的思想实现。



方案二具体实现：

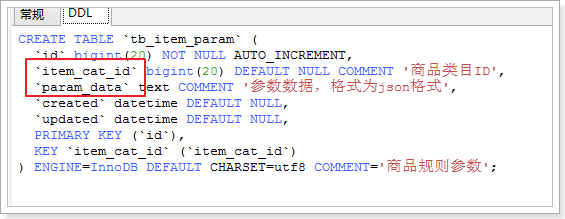
1. 模板如何存储？
   1. 存储到数据库
   2. 字段不能固定
      1. Map
      2. Json
2. 存储的json结构
   1. 模板结构  
      
   2. 最终数据结构  
      

## 数据库表结构

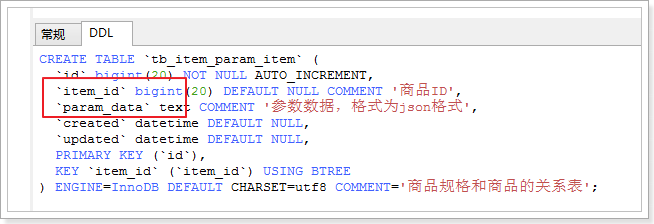
需要有2张表：

1. 模板表，需要和商品类目关联
2. 规格参数数据表，需要和商品关联

### 模板表

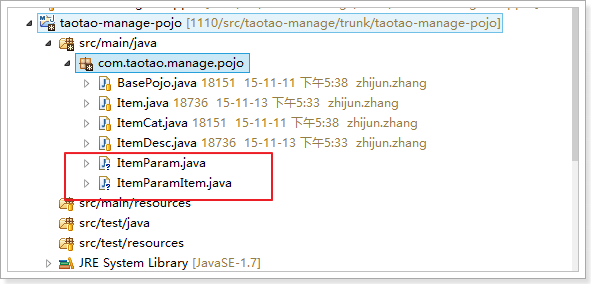


### 最终数据表

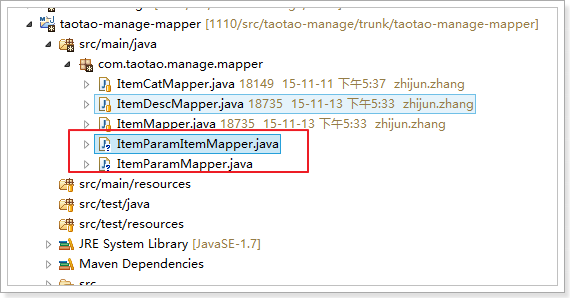


## 实现

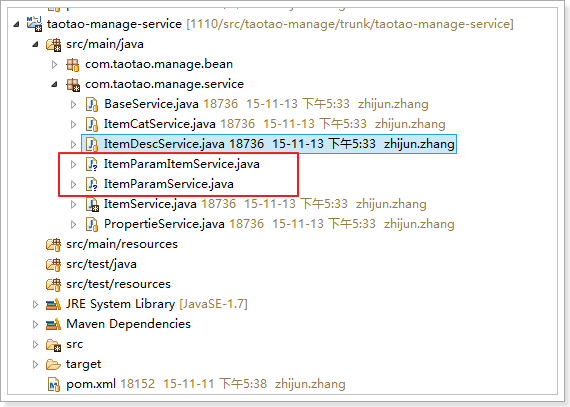
### 导入pojo



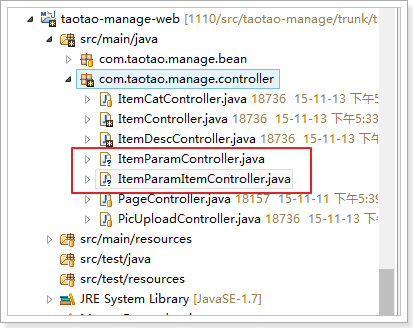
### 创建Mapper



### 创建Service



### 创建Controller



## 页面功能

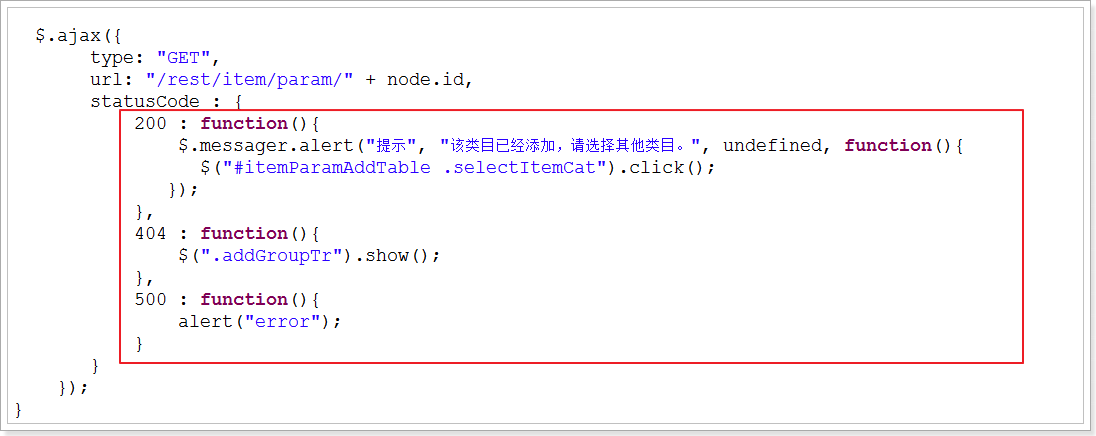
### 选择类目

根据选择的类目进行判断，如果该类目所对应的模板存在，提醒用户已经存在，如果模板不存在，可以创建模板。

### 后台开发根据类目id查找模板的接口



### JS实现



### 自定义回调函数



执行传入的函数：



## 点击提交事件

$("#itemParamAddTable .submit").click(**function**(){

**var** params = [];

**var** groups = $("#itemParamAddTable [name=group]");

groups.each(**function**(i,e){

**var** p = $(e).parentsUntil("ul").parent().find("[name=param]");

**var** \_ps = [];

p.each(**function**(\_i,\_e){

**var** \_val = $(\_e).siblings("input").val();

**if**($.trim(\_val).length>0){

\_ps.push(\_val);

}

});

**var** \_val = $(e).siblings("input").val();

**if**($.trim(\_val).length>0 && \_ps.length > 0){

params.push({

"group":\_val,

"params":\_ps

});

}

});

**var** url = "/rest/item/param/"+$("#itemParamAddTable [name=cid]").val();

//JSON.stringify将js的对象序列化为json字符串

$.post(url,{"paramData":JSON.stringify(params)},**function**(data){

$.messager.alert('提示','新增商品规格成功!',undefined,**function**(){

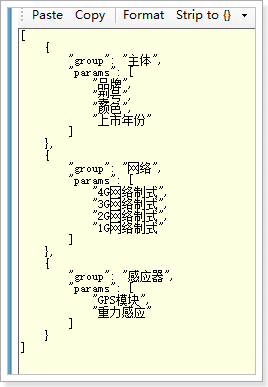
$(".panel-tool-close").click();

$("#itemParamList").datagrid("reload");

});

});

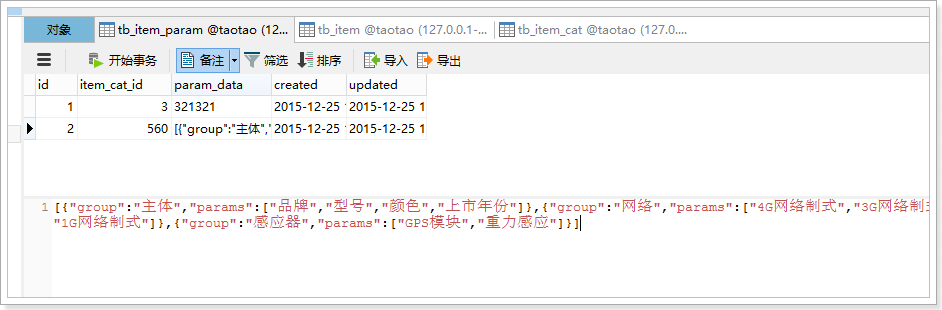
});

提交的数据结构：  


## 后端实现



效果：

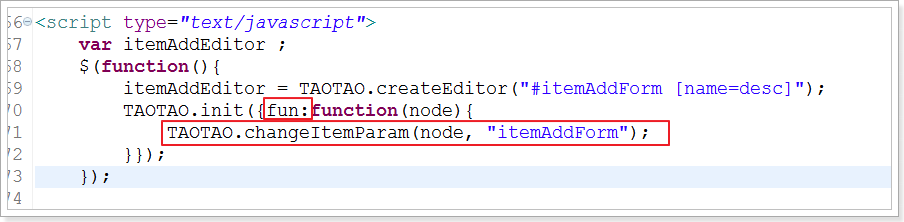


## 查询规格参数模板列表

TODO.

## 新增商品时套用模板输入数据

### 选择类目时触发加载模板



动态生成form表单内容:

changeItemParam : **function**(node,formId){

$.ajax({

type: "GET",

url: "/rest/item/param/" + node.id,

statusCode : {

200 : **function**(data){

$("#"+formId+" .params").show();

**var** paramData = JSON.parse(data.paramData);

**var** html = "<ul>";

**for**(**var** i **in** paramData){

**var** pd = paramData[i];

html+="<li><table>";

html+="<tr><td colspan=\"2\" class=\"group\">"+pd.group+"</td></tr>";

**for**(**var** j **in** pd.params){

**var** ps = pd.params[j];

html+="<tr><td class=\"param\"><span>"+ps+"</span>: </td><td><input autocomplete=\"off\" type=\"text\"/></td></tr>";

}

html+="</li></table>";

}

html+= "</ul>";

$("#"+formId+" .params td").eq(1).html(html);

},

404 : **function**(){

$("#"+formId+" .params").hide();

$("#"+formId+" .params td").eq(1).empty();

},

500 : **function**(){

alert("error");

}

}

});

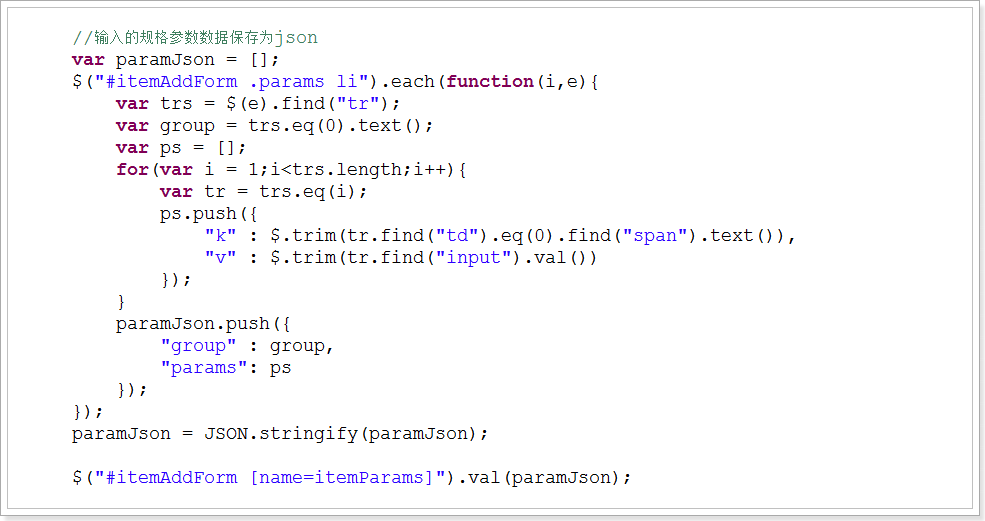
},

效果：



### 点击提交按钮，将用户的输入，生成json数据

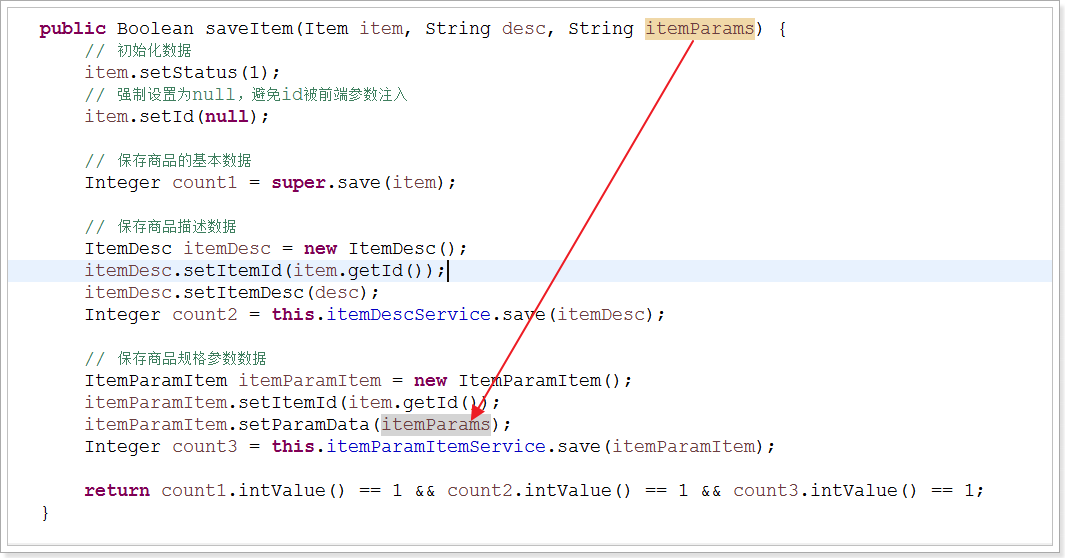
强烈建议大家自己写一遍。



### 后台实现



Service：



效果：



## 编辑商品 – 规格参数回显

编辑商品时通过商品id查询规格参数数据：

后台实现：



前端JS：

//加载商品规格

$.getJSON('/rest/item/param/item/'+data.id,**function**(\_data){

**if**(\_data.paramData){

$("#itemeEditForm .params").show();

$("#itemeEditForm [name=itemParams]").val(\_data.paramData);

$("#itemeEditForm [name=itemParamId]").val(\_data.id);

//回显商品规格

**var** paramData = JSON.parse(\_data.paramData);

**var** html = "<ul>";

**for**(**var** i **in** paramData){

**var** pd = paramData[i];

html+="<li><table>";

html+="<tr><td colspan=\"2\" class=\"group\">"+pd.group+"</td></tr>";

**for**(**var** j **in** pd.params){

**var** ps = pd.params[j];

html+="<tr><td class=\"param\"><span>"+ps.k+"</span>: </td><td><input autocomplete=\"off\" type=\"text\" value='"+ps.v+"'/></td></tr>";

}

html+="</li></table>";

}

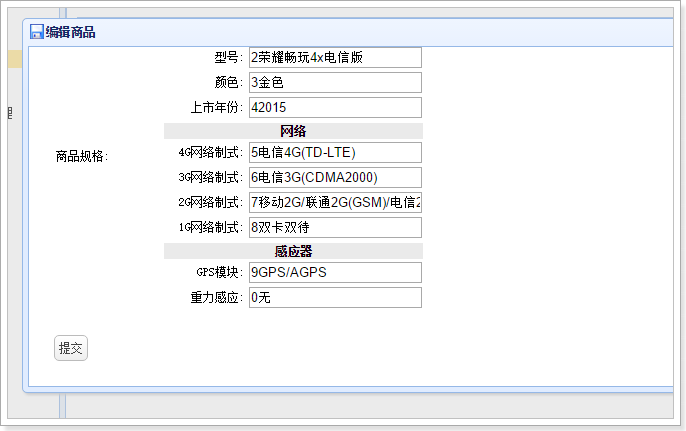
html+= "</ul>";

$("#itemeEditForm .params td").eq(1).html(html);

}

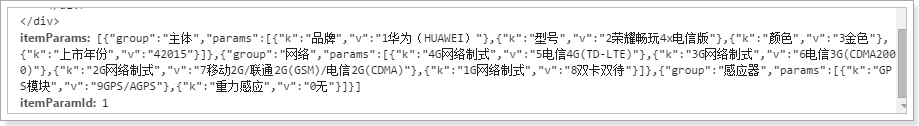
});

效果：

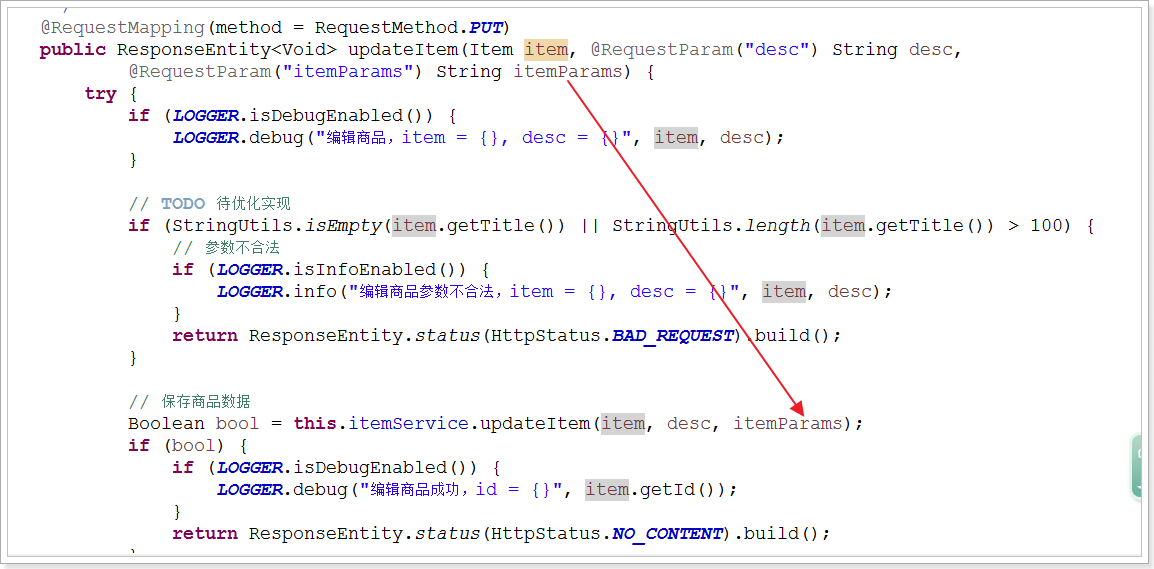


## 更新规格参数数据

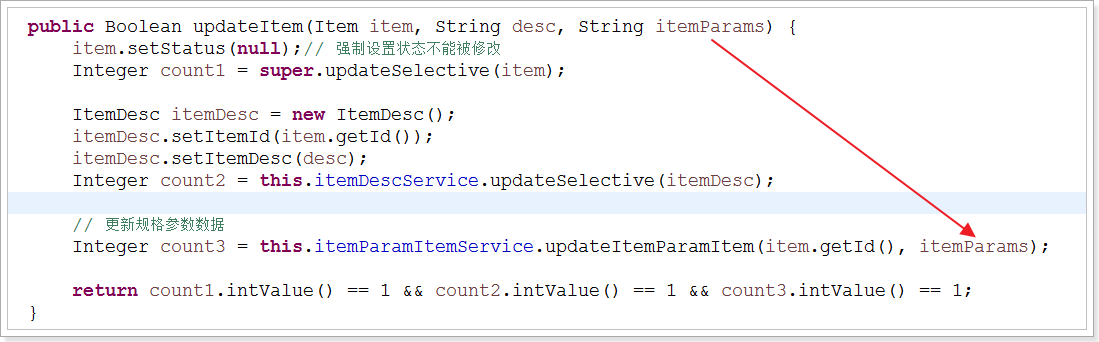
提交的参数：



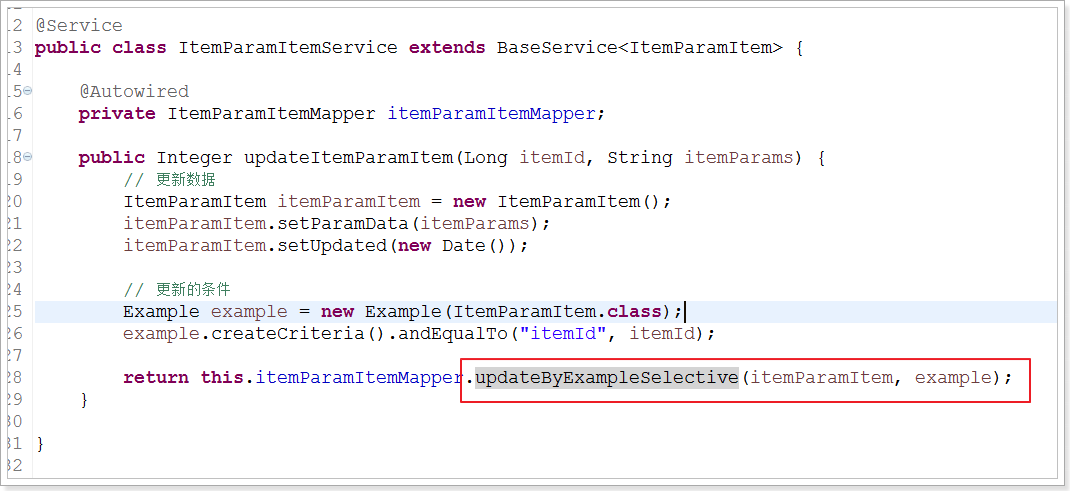
后台处理：



Service：



ItemParamItemService：



效果：



# 搭建前台系统

## 所使用的技术

后台技术： Spring SpringMVC ~~Mybatis~~？

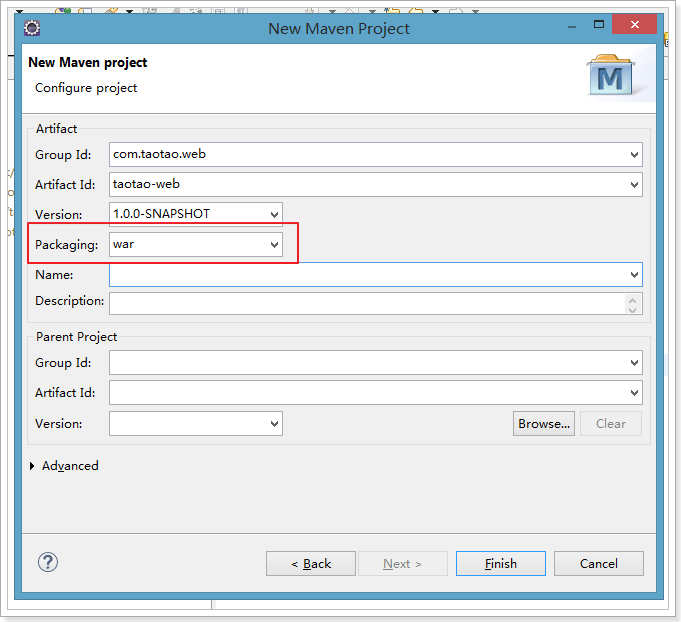
前台技术：html、CSS、JS

如果不使用Mybatis，商品的数据从何而来？ -- 来源于Mysql数据库

获取数据的途径：

1. 从JDBC获取
   1. 优点
      1. 直接，获取的途径较短，简单
   2. 缺点
      1. 对后台系统团队而言，数据不安全（只要开放的账户是只读的账户即可）
      2. 前端系统团队需要有学习的成本，才能使用数据库
      3. 依赖、耦合度太高，后端团队将数据库结构修改，那么其他团队必须跟着修改逻辑，才能使用
      4. 直接走数据库查询，无法添加缓存逻辑
2. 通过后台系统接口获取
   1. 优点
      1. 耦合度降低，后端团队只要保证接口的返回数据格式不变化，其他团队就无需升级
      2. 数据安全
      3. 前端团队无需了解学习后端团队的底层数据库结构
      4. 后端团队可以在接口处添加缓存逻辑
   2. 缺点
      1. 获取的路径较长（不是真正的缺点）

## 创建taotao-web



## 导入依赖

<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.parent</groupId>

<artifactId>taotao-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>com.taotao.web</groupId>

<artifactId>taotao-web</artifactId>

<version>1.0.0-SNAPSHOT</version>

<packaging>war</packaging>

<dependencies>

<dependency>

<groupId>com.taotao.common</groupId>

<artifactId>taotao-common</artifactId>

<version>1.0.0-SNAPSHOT</version>

</dependency>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<!-- Jackson Json处理工具包 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

</dependency>

<!-- JSP相关 -->

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

<scope>provided</scope>

</dependency>

<!-- Apache工具组件 -->

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-io</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<configuration>

<port>8082</port>

<path>/</path>

</configuration>

</plugin>

</plugins>

</build>

</project>

## Web.xml

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

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

xmlns=*"http://java.sun.com/xml/ns/javaee"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

id=*"WebApp\_ID"* version=*"2.5"*>

<display-name>taotao-web</display-name>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring/applicationContext\*.xml</param-value>

</context-param>

<!--Spring的ApplicationContext 载入 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- 编码过滤器，以UTF8编码 -->

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>UTF8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<!-- 配置SpringMVC框架入口 -->

<servlet>

<servlet-name>taotao-web</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring/taotao-web-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>taotao-web</servlet-name>

<!--

伪静态

伪静态有利于SEO（搜索引擎优化）

-->

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

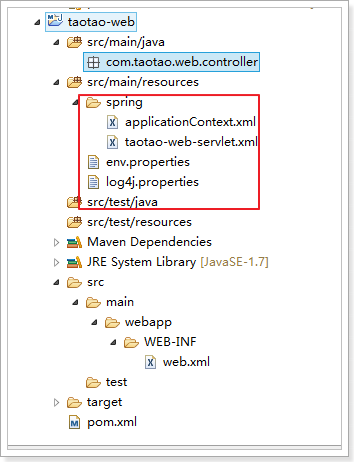
<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

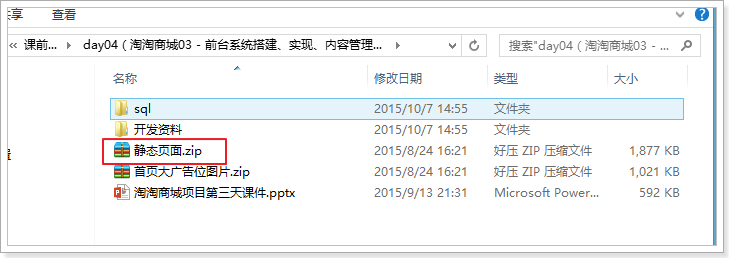
</welcome-file-list>

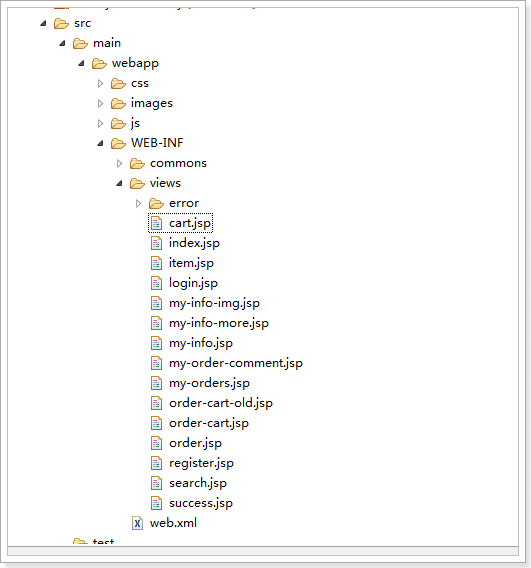
</web-app>

## Spring和SpringMVC配置文件

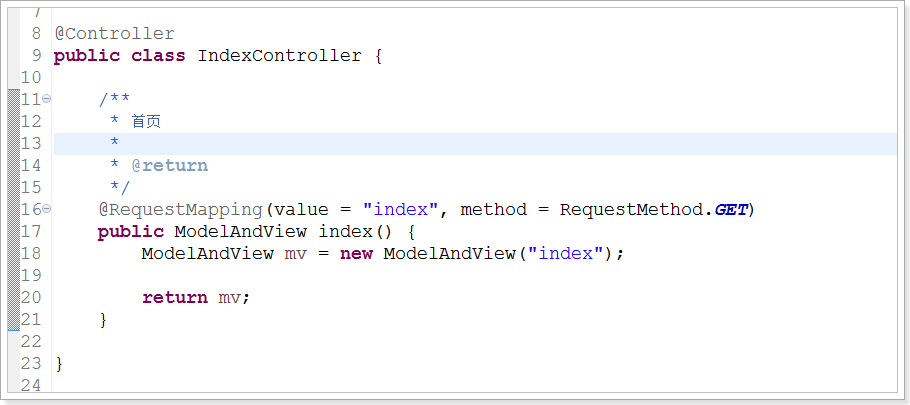


## 导入静态文件

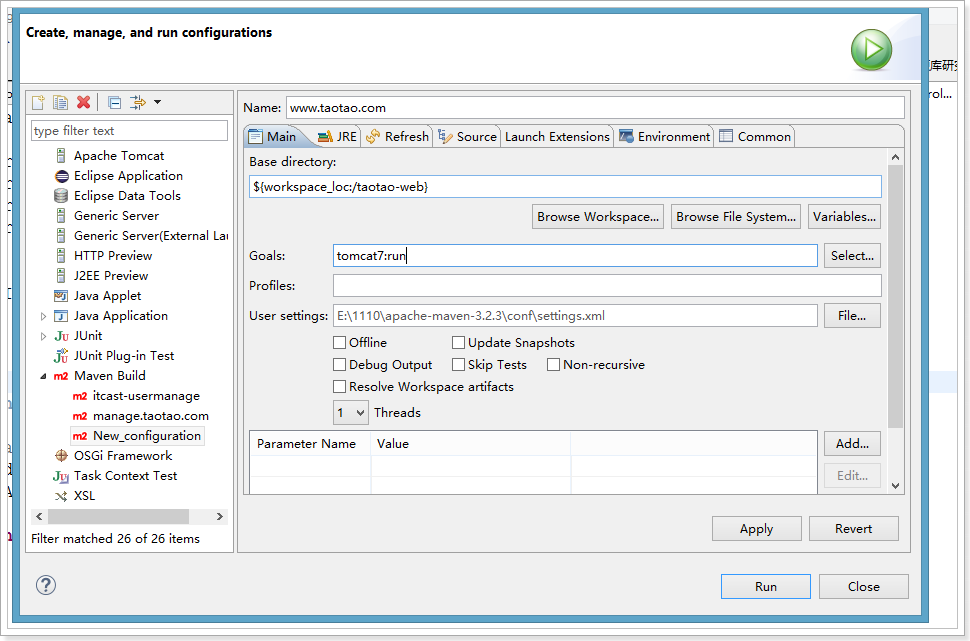




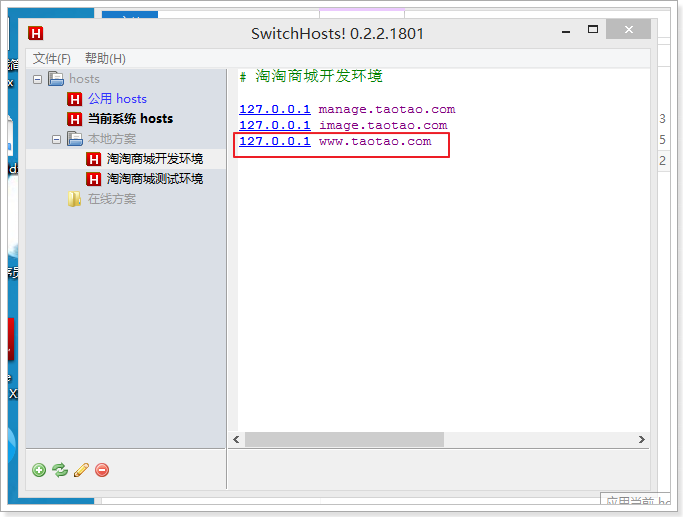
## 编写IndexController



## 配置tomcat插件



## 配置hosts和nginx



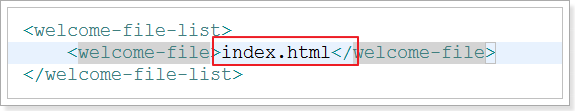


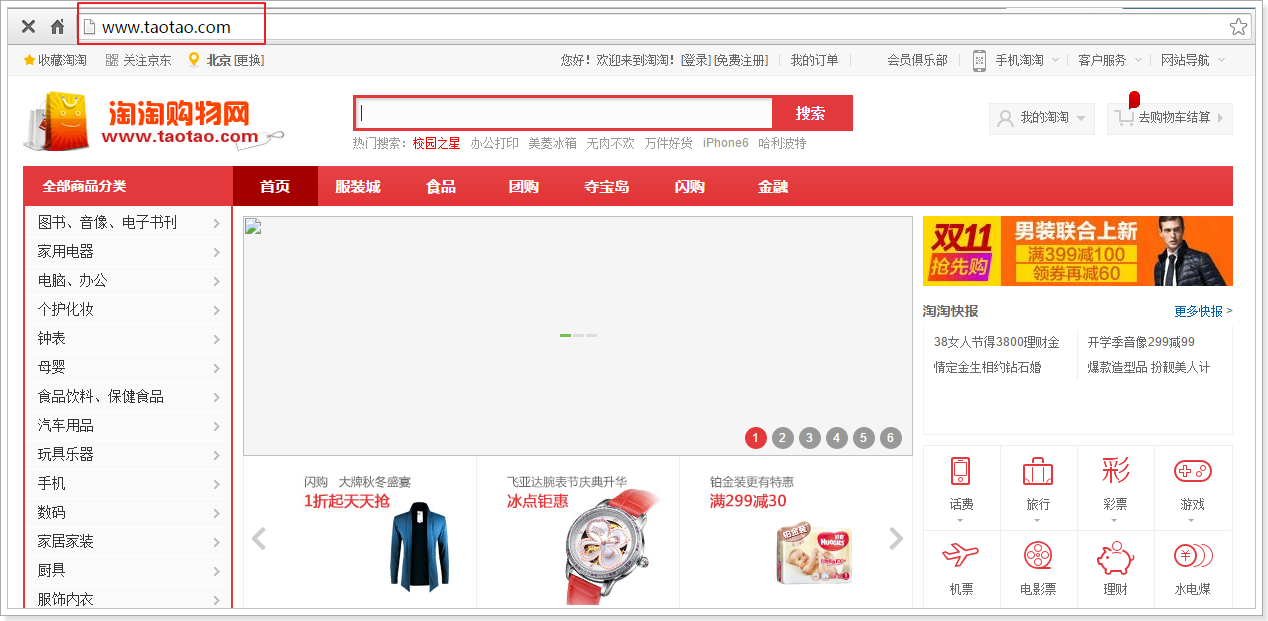
## 测试



## 如何通过域名直接访问？

Web.xml:



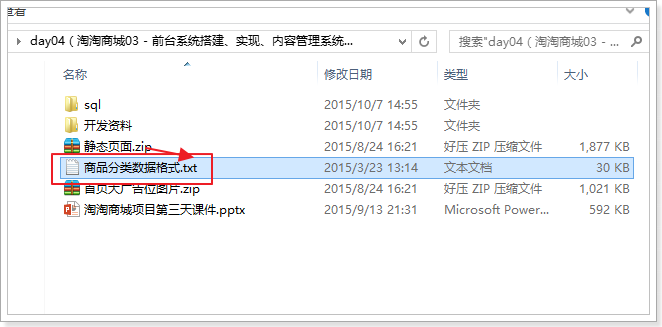


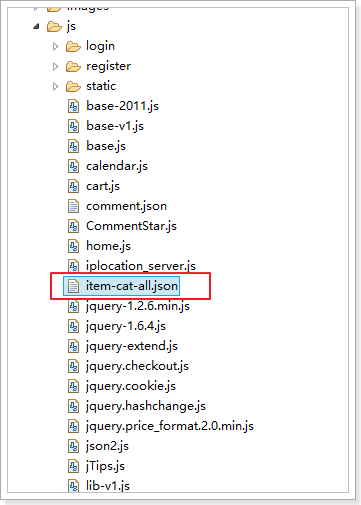
# 首页左侧商品类目显示

## 功能

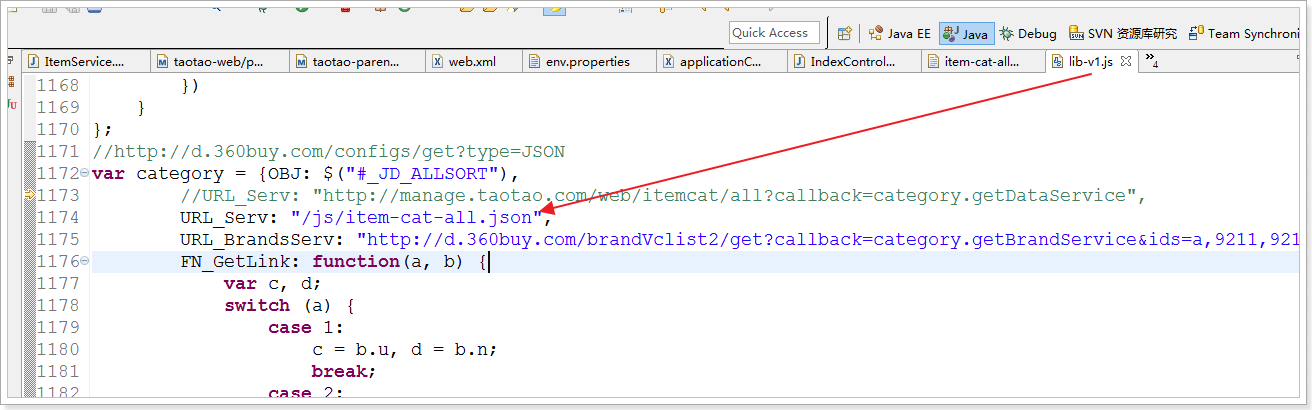


## 构造数据





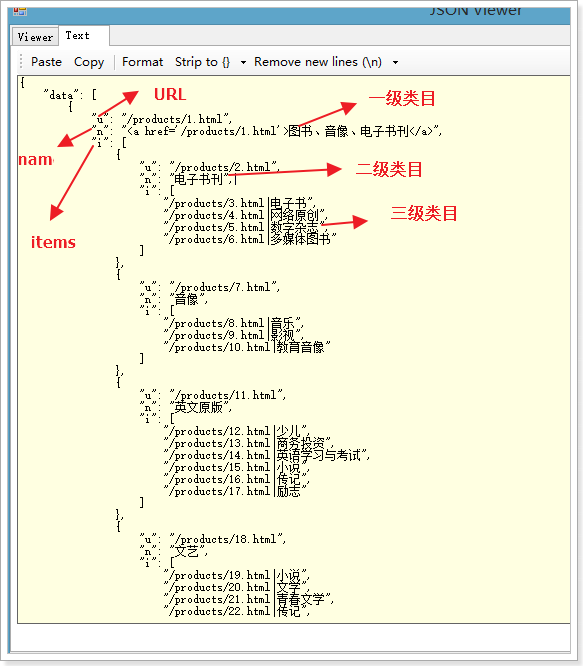
## 修改JS



## 测试



## 数据结构

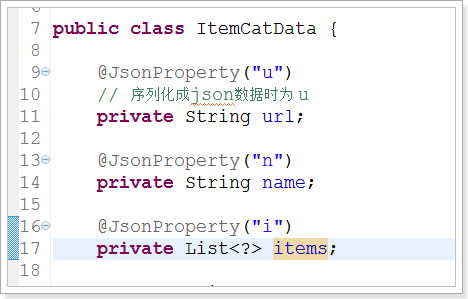


## 后台系统开发接口返回数据

### 定义ItemCatReult



### 定义ItemCatData



### Controller



### Service（强烈建议自己写一遍，不是抄一遍）

/\*\*

\* 全部查询，并且生成树状结构

\*

\* **@return**

\*/

**public** ItemCatResult queryAllToTree() {

ItemCatResult result = **new** ItemCatResult();

// 全部查出，并且在内存中生成树形结构

List<ItemCat> cats = **super**.queryAll();

// 转为map存储，key为父节点ID，value为数据集合

Map<Long, List<ItemCat>> itemCatMap = **new** HashMap<Long, List<ItemCat>>();

**for** (ItemCat itemCat : cats) {

**if** (!itemCatMap.containsKey(itemCat.getParentId())) {

itemCatMap.put(itemCat.getParentId(), **new** ArrayList<ItemCat>());

}

itemCatMap.get(itemCat.getParentId()).add(itemCat);

}

// 封装一级对象

List<ItemCat> itemCatList1 = itemCatMap.get(0L);

**for** (ItemCat itemCat : itemCatList1) {

ItemCatData itemCatData = **new** ItemCatData();

itemCatData.setUrl("/products/" + itemCat.getId() + ".html");

itemCatData.setName("<a href='" + itemCatData.getUrl() + "'>" + itemCat.getName() + "</a>");

result.getItemCats().add(itemCatData);

**if** (!itemCat.getIsParent()) {

**continue**;

}

// 封装二级对象

List<ItemCat> itemCatList2 = itemCatMap.get(itemCat.getId());

List<ItemCatData> itemCatData2 = **new** ArrayList<ItemCatData>();

itemCatData.setItems(itemCatData2);

**for** (ItemCat itemCat2 : itemCatList2) {

ItemCatData id2 = **new** ItemCatData();

id2.setName(itemCat2.getName());

id2.setUrl("/products/" + itemCat2.getId() + ".html");

itemCatData2.add(id2);

**if** (itemCat2.getIsParent()) {

// 封装三级对象

List<ItemCat> itemCatList3 = itemCatMap.get(itemCat2.getId());

List<String> itemCatData3 = **new** ArrayList<String>();

id2.setItems(itemCatData3);

**for** (ItemCat itemCat3 : itemCatList3) {

itemCatData3.add("/products/" + itemCat3.getId() + ".html|" + itemCat3.getName());

}

}

}

**if** (result.getItemCats().size() >= 14) {

**break**;

}

}

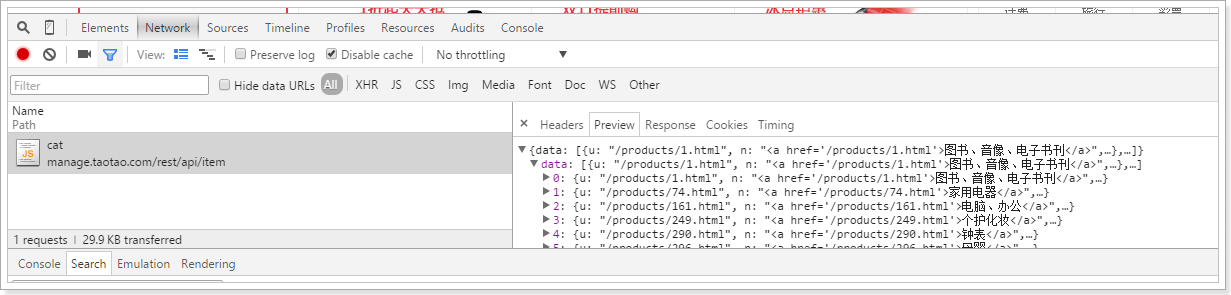
**return** result;

}

### 测试



### 集成到前台系统



可以看到数据是以及获取到。

但是，出错（JS解析出错）：



## 跨域问题

浏览器对ajax请求的限制，不允许跨域请求资源。

<http://www.a.com> 🡺 <http://www.b.com> 是跨域

<http://www.a.com> 🡺 <http://www.a.com>:8080 是跨域

<http://a.a.com> 🡺 <http://b.a.com> 是跨域

<http://www.a.com> 🡺 <http://www.a.com/api> 不是

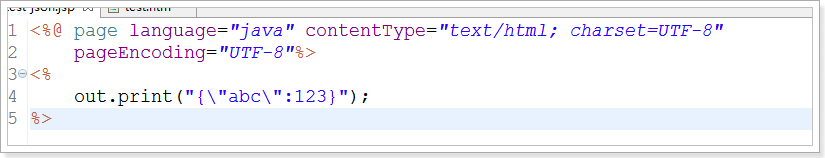
总结：

不同的域名或不同的端口都是跨域请求。

如何解决跨域问题？ -- jsonp

## Jsonp

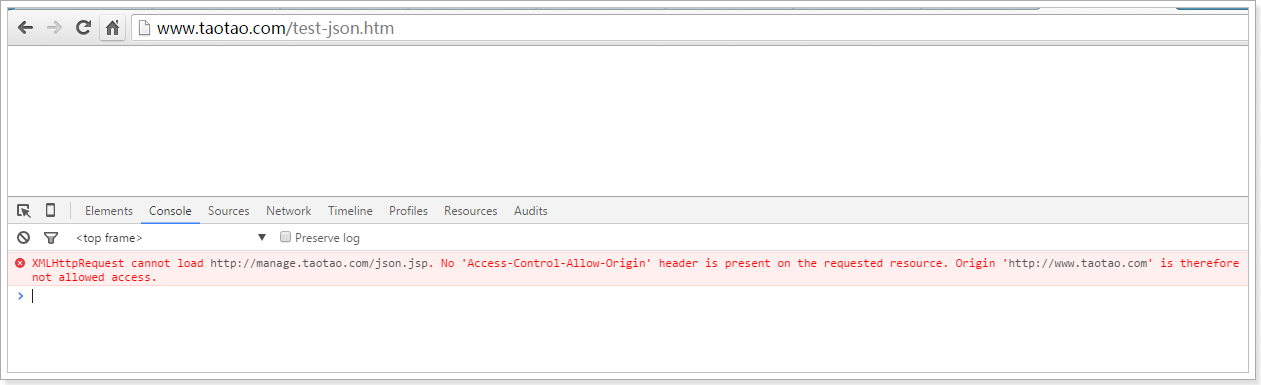
### 编写json.jsp（后台系统）



### 在后台系统中编写test-json.htm



### 将test-json.htm拷贝到前台系统进行测试



发现:

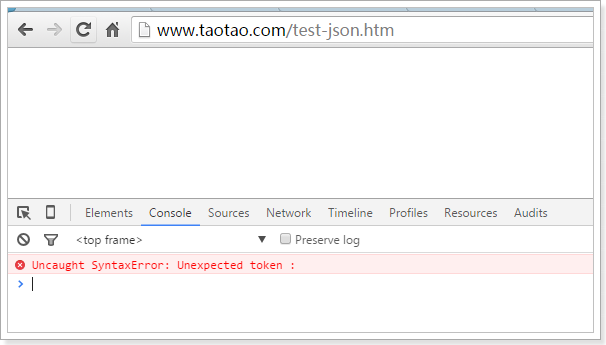
1. alert($) 可以正常弹出
2. alert(data.abc) 不能够正常的弹出，出现跨域问题

结论：script标签的src可以跨域请求资源，但是ajax请求不可以跨域请求。

疑问：能否借助script标签的src进行加载数据？ -- 可以的。

### 借助script的src跨域加载资源





发现：

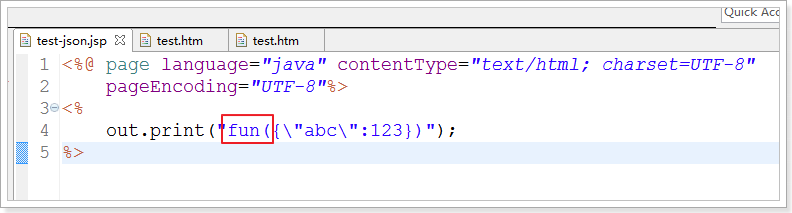
请求资源可以正常请求，但是，报js解析出错。

原因：

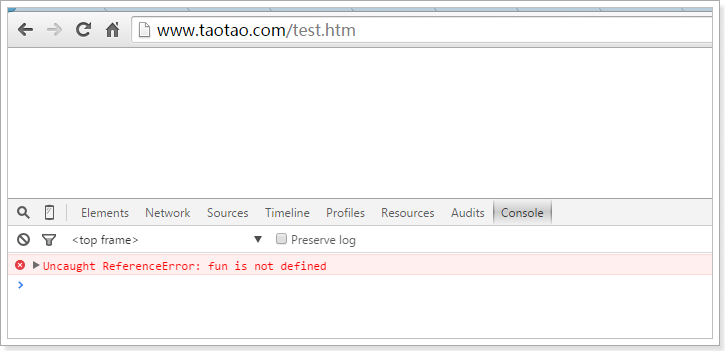
Script标签加载到资源后，会将资源当做是js脚本解析，但是我们返回的是json数据，所以导致解析失败。

解决：只需要返回js脚本即可。

### 后端系统返回js脚本



测试：

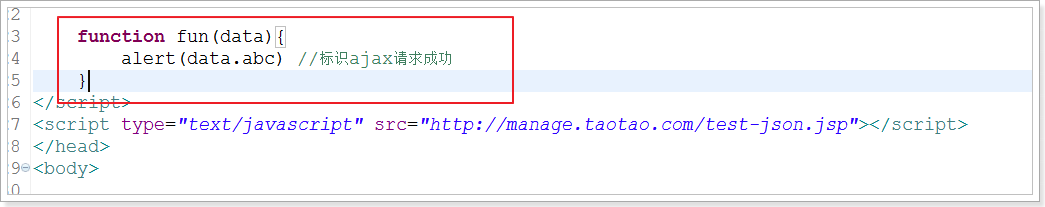


发现:

返回的js脚本成功解析，但是，fun没有定义。

解决：定义个一个fun方法即可。

### 定义fun方法



测试：



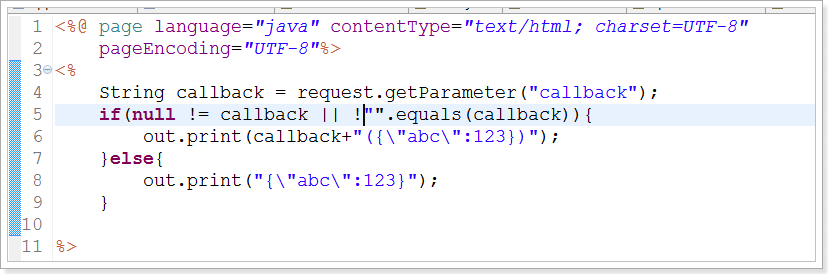
### 总结

Jsonp的原理：

1. jsonp通过script标签的src可以跨域请求的特性，加载资源
2. 将加载的资源（通过一个方法名将数据进行包裹）当做是js脚本解析
3. 定义一个回调函数，获取传入的数据

### 优化

将回调函数名传递到服务端，返回：

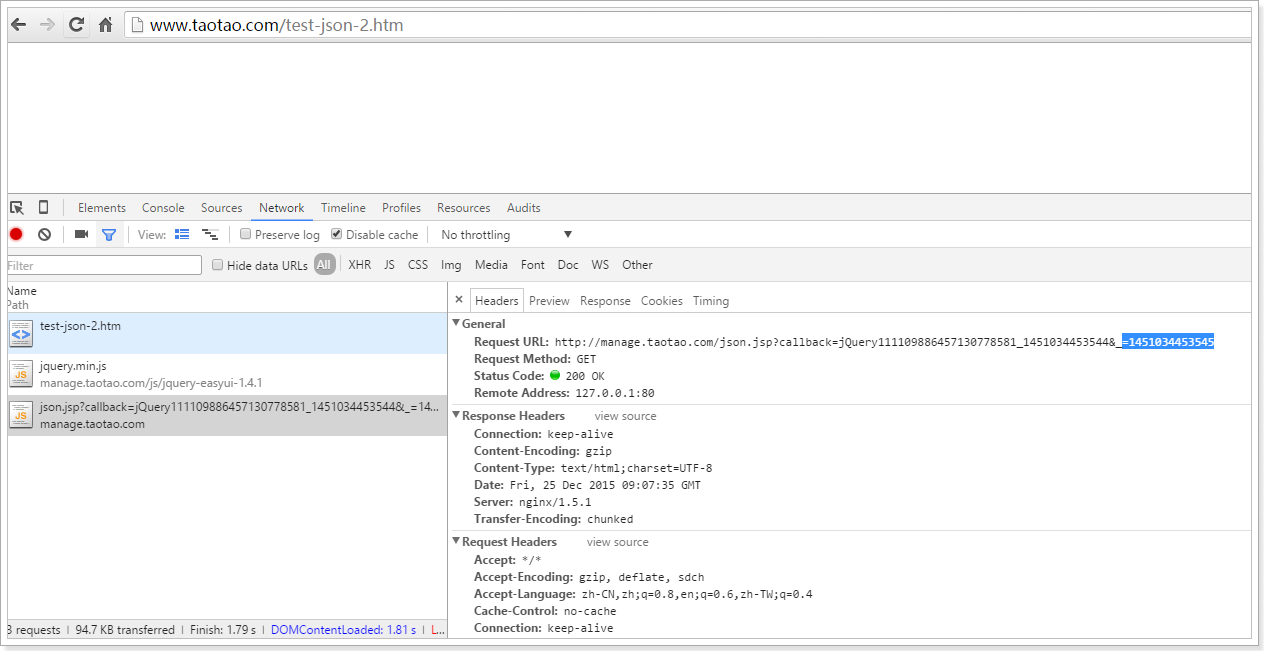


调用方：



## 通过jQuery使用jsonp请求



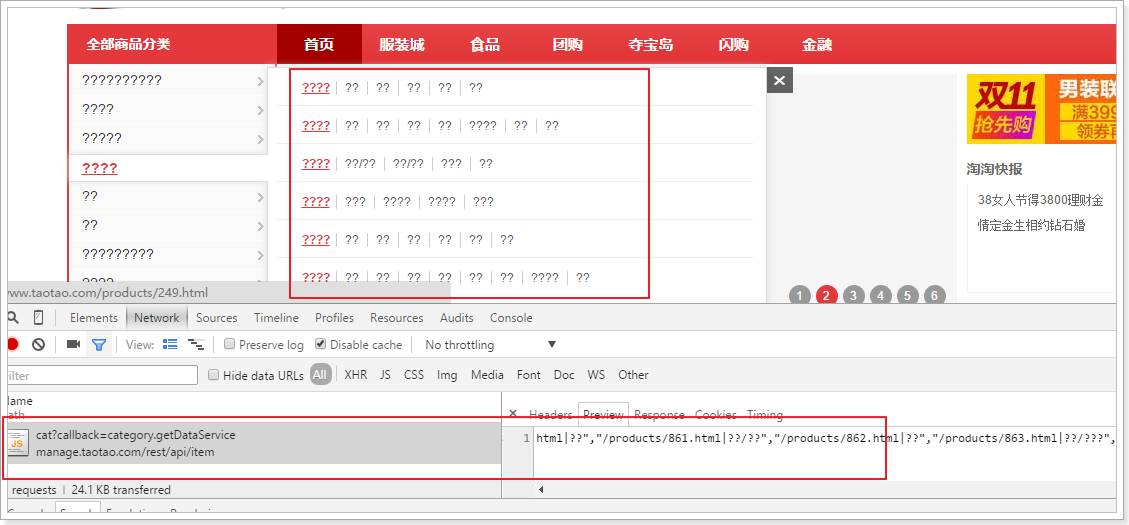


## 解决项目中跨域问题

### 后台系统Controller



### 测试



问题解决，但是带来了新问题，乱码问题。

## 解决乱码问题

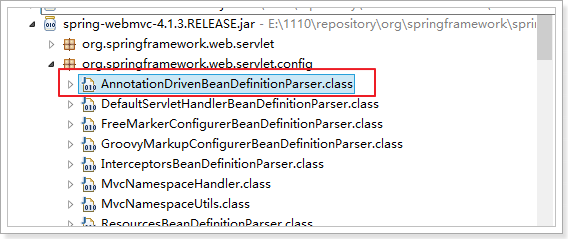
### 分析乱码产生的原因

在SpringMVC中产生的响应有2类：

1. ModelAndView
2. 返回数据响应
   1. 使用消息转化器完成

配置：





配置默认的消息转化器：

**private** ManagedList<?> getMessageConverters(Element element, Object source, ParserContext parserContext) {

Element convertersElement = DomUtils.*getChildElementByTagName*(element, "message-converters");

ManagedList<? **super** Object> messageConverters = **new** ManagedList<Object>();

**if** (convertersElement != **null**) {

messageConverters.setSource(source);

**for** (Element beanElement : DomUtils.*getChildElementsByTagName*(convertersElement, "bean", "ref")) {

Object object = parserContext.getDelegate().parsePropertySubElement(beanElement, **null**);

messageConverters.add(object);

}

}

**if** (convertersElement == **null** || Boolean.*valueOf*(convertersElement.getAttribute("register-defaults"))) {

messageConverters.setSource(source);

messageConverters.add(createConverterDefinition(ByteArrayHttpMessageConverter.**class**, source));

RootBeanDefinition stringConverterDef = createConverterDefinition(StringHttpMessageConverter.**class**, source);

stringConverterDef.getPropertyValues().add("writeAcceptCharset", **false**);

messageConverters.add(stringConverterDef);

messageConverters.add(createConverterDefinition(ResourceHttpMessageConverter.**class**, source));

messageConverters.add(createConverterDefinition(SourceHttpMessageConverter.**class**, source));

messageConverters.add(createConverterDefinition(AllEncompassingFormHttpMessageConverter.**class**, source));

**if** (*romePresent*) {

messageConverters.add(createConverterDefinition(AtomFeedHttpMessageConverter.**class**, source));

messageConverters.add(createConverterDefinition(RssChannelHttpMessageConverter.**class**, source));

}

**if** (***jackson2XmlPresent***) {

RootBeanDefinition jacksonConverterDef = createConverterDefinition(MappingJackson2XmlHttpMessageConverter.**class**, source);

GenericBeanDefinition jacksonFactoryDef = createObjectMapperFactoryDefinition(source);

jacksonFactoryDef.getPropertyValues().add("createXmlMapper", **true**);

jacksonConverterDef.getConstructorArgumentValues().addIndexedArgumentValue(0, jacksonFactoryDef);

messageConverters.add(jacksonConverterDef);

}

**else** **if** (***jaxb2Present***) {

messageConverters.add(createConverterDefinition(Jaxb2RootElementHttpMessageConverter.**class**, source));

}

**if** (***jackson2Present***) {

RootBeanDefinition jacksonConverterDef = createConverterDefinition(MappingJackson2HttpMessageConverter.**class**, source);

GenericBeanDefinition jacksonFactoryDef = createObjectMapperFactoryDefinition(source);

jacksonConverterDef.getConstructorArgumentValues().addIndexedArgumentValue(0, jacksonFactoryDef);

messageConverters.add(jacksonConverterDef);

}

**else** **if** (***gsonPresent***) {

messageConverters.add(createConverterDefinition(GsonHttpMessageConverter.**class**, source));

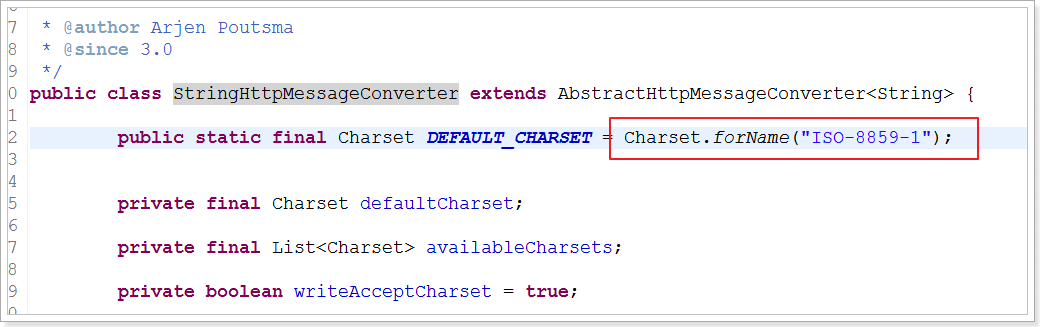
}

}

**return** messageConverters;

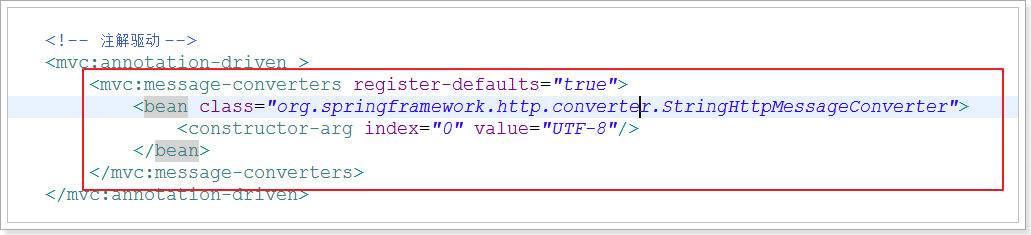
}

### 默认使用的字符串的消息转化器

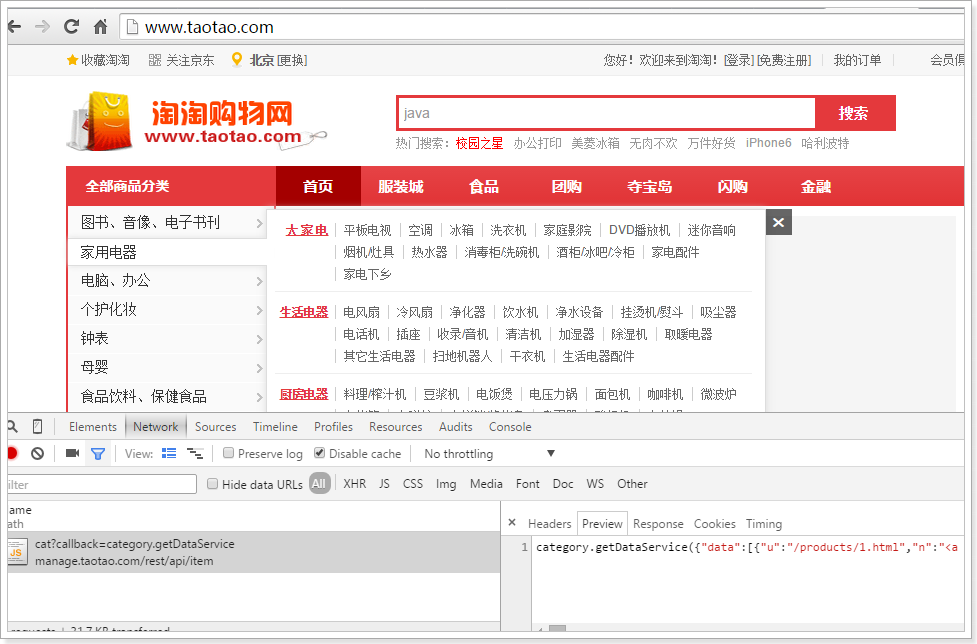


发现，默认使用ISO-8859-1，所以会产生乱码。

### 解决乱码



问题解决：

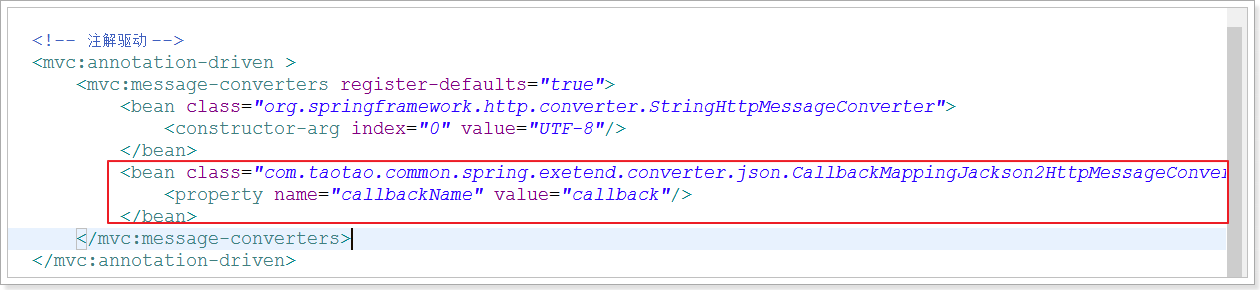


## 统一支持jsonp

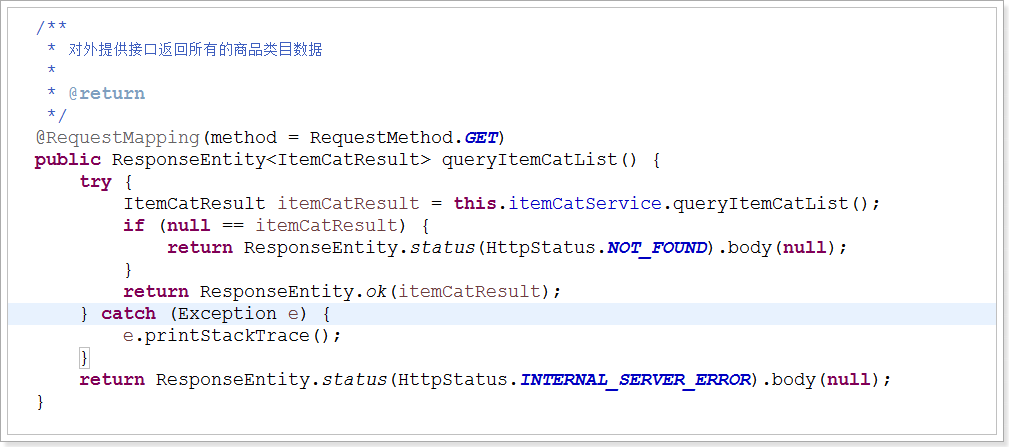
### 扩展CallbackMappingJackson2HttpMessageConverter



### 配置



### Controller实现

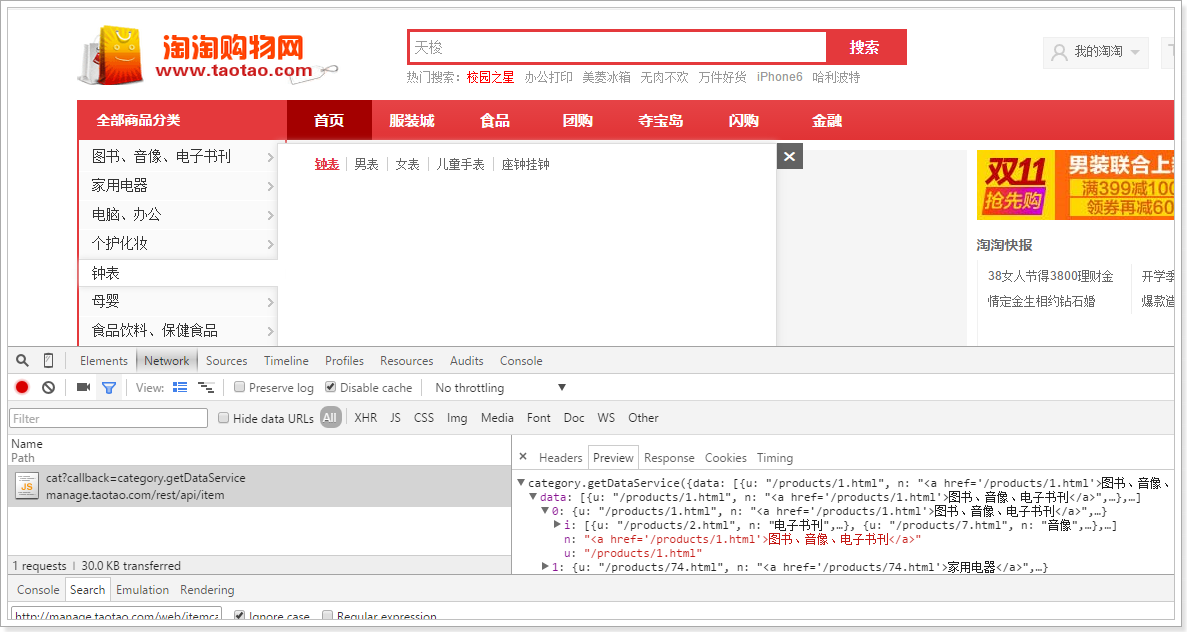


### 测试





效果：



至此，首页左侧商品类目的显示，完美实现。