taotao商城项目总结

1、taotao-manage-web中图片上传问题

1)利用nginx做图片静态资源放置,文件名的生成，富文本编辑器的使用。
String fileName = new DateTime(nowDate).toString(“yyyyMMddhhmmssss”)+
RandomUtils.nextInt(100,9999)+”.”+StringUtils.subStringAfterLast(sourceFileName, “.”) ;
2) 检查图片格式,把图片状态都存放在fileuploadResult对象中,使用ObjtectMapper把fileuploadResult对象转换成换成json发送给前端页面.fileuploadResult中保存有
//上传是否成功的判断标识，0-成功，1-失败
private Integer error;
private String url;
private String width;
private String height;
// 校验图片是否合法

```
isLegal = false;
try {
    BufferedImage image = ImageIO.read(newFile);
    if (image != null) {
        fileUploadResult.setWidth(image.getWidth() + "");
        fileUploadResult.setHeight(image.getHeight() + "");
        isLegal = true;
    }
} catch (IOException e) {
}
```

//将java对象转化成（序列化）json字符串
return mapper.writeValueAsString(fileUploadResult);

2、EasyUI datagride中数据的显示

/**
* 查询商品列表
*/

```
@RequestMapping(method = RequestMethod.GET)
public ResponseEntity<EasyUIResult> queryItemList(
        @RequestParam(value = "page", defaultValue = "1") Integer p
age,
        @RequestParam(value = "rows", defaultValue = "30") Integer
rows) {
    try {
        PageInfo<Item> pageInfo = this.itemService.queryPageList(pa
ge, rows);
        EasyUIResult easyUIResult = new EasyUIResult(pageInfo.getTo
tal(), pageInfo.getList());
        return ResponseEntity.ok(easyUIResult);
    } catch (Exception e) {
        e.printStackTrace();
    }
    // 出错 500
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERRO
R).body(null);
}
        //service层,pageInfo是一个分页助手com.github.pagehelper
public PageInfo<Item> queryPageList(Integer page, Integer rows) {
    Example example = new Example(Item.class);
    example.setOrderByClause("updated DESC");

    // 设置分页参数
    PageHelper.startPage(page, rows);

    List<Item> list = this.itemMapper.selectByExample(example);
    return new PageInfo<Item>(list);
}

public class EasyUIResult {
// 定义jackson对象
private static final ObjectMapper MAPPER = new ObjectMapper();
private Integer total;
private List<?> rows;
/**
 * Object是集合转化
 * @param jsonData json数据
 * @param clazz 集合中的类型
 * json数据如下
     {
    "total": 3091,
    "rows": [
    {
        "created": 1474723409000,
        "updated": 1474723409000,
        "id": 1474391930,
```

```
        "title": "6-12个月",
        "sellPoint": "6-12个月",
        "price": 1100,
        "num": 11,
        "barcode": "",
        "image": "http://localhost:8888/images/2016/09/24/201609240
9232138103245.jpg",
        "cid": 498,
        "status": 1
    }
]
```

}

```
 */
public static EasyUIResult formatToList(String jsonData, Class<?> c
lazz) {
    try {
        JsonNode jsonNode = MAPPER.readTree(jsonData);
        JsonNode data = jsonNode.get("rows");
        List<?> list = null;
        if (data.isArray() && data.size() > 0) {
            list = MAPPER.readValue(data.traverse(),
                    MAPPER.getTypeFactory().constructCollectionTyp
e(List.class, clazz));
        }
        return new EasyUIResult(jsonNode.get("total").intValue(), l
ist);
    } catch (Exception e) {
        return null;
    }
}
}
```

2、数据,图片的回显

1) 商品类目,图片的显示,前台利用js实现
2)描述数据的回显,用ajax发送请求

```
// 加载商品描述
$.getJSON('/rest/item/desc/'+data.id,function(_data){
    itemEditEditor.html(_data.itemDesc);//调用富文本编辑器的API实现回显
});
```

3)提交事件时事务的处理,

/*

* 新增商品

* */

```
@RequestMapping(method=RequestMethod.POST)
public ResponseEntity<Void> saveItem(Item item,
          String desc,String itemParams){
    try{
        if(StringUtils.isEmpty(item.getTitle())){
            //400
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).bu
ild();
        }
        //保存商品的基本数据
        this.itemService.saveItem(item,desc,itemParams);
        //保存成功 201
        return ResponseEntity.status(HttpStatus.CREATED).build();
    }catch(Exception e){
        e.printStackTrace();
    }
    // 出错 500
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERRO
R).build();
}
//事务的处理xml中配置
<!-- 定义事务管理器 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactio
nManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!-- 定义事务策略 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!--所有以query开头的方法都是只读的 -->
        <tx:method name="query*" read-only="true" />
        <!--其他方法使用默认事务策略 -->
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>
```

```

## 3、商品的规格参数功能实现

1)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。使用模板的思想实现,数据结构如下
```

```
[
  {
    "group": "主体",
    "params": [
      {
        "k": "品牌",
        "v": "苹果（Apple）"
      },
      {
        "k": "型号",
        "v": "iPhone 6 A1589"
      }
    ]
  }
]
```


表结构如下

CREATE TABLE `tb_item_param_item` (
`id` bigint(20) NOT NULL AUTO_INCREMENT,
`item_id` bigint(20) DEFAULT NULL COMMENT '商品ID',
`param_data` text COMMENT '参数数据，格式为json格式',
`created` datetime DEFAULT NULL,
`updated` datetime DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `item_id` (`item_id`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8 COMMENT='商品规格和商品的关系表';

实现后台添加规格参数在增加商品时回限规格参数。当选择类目时触发加载模板，

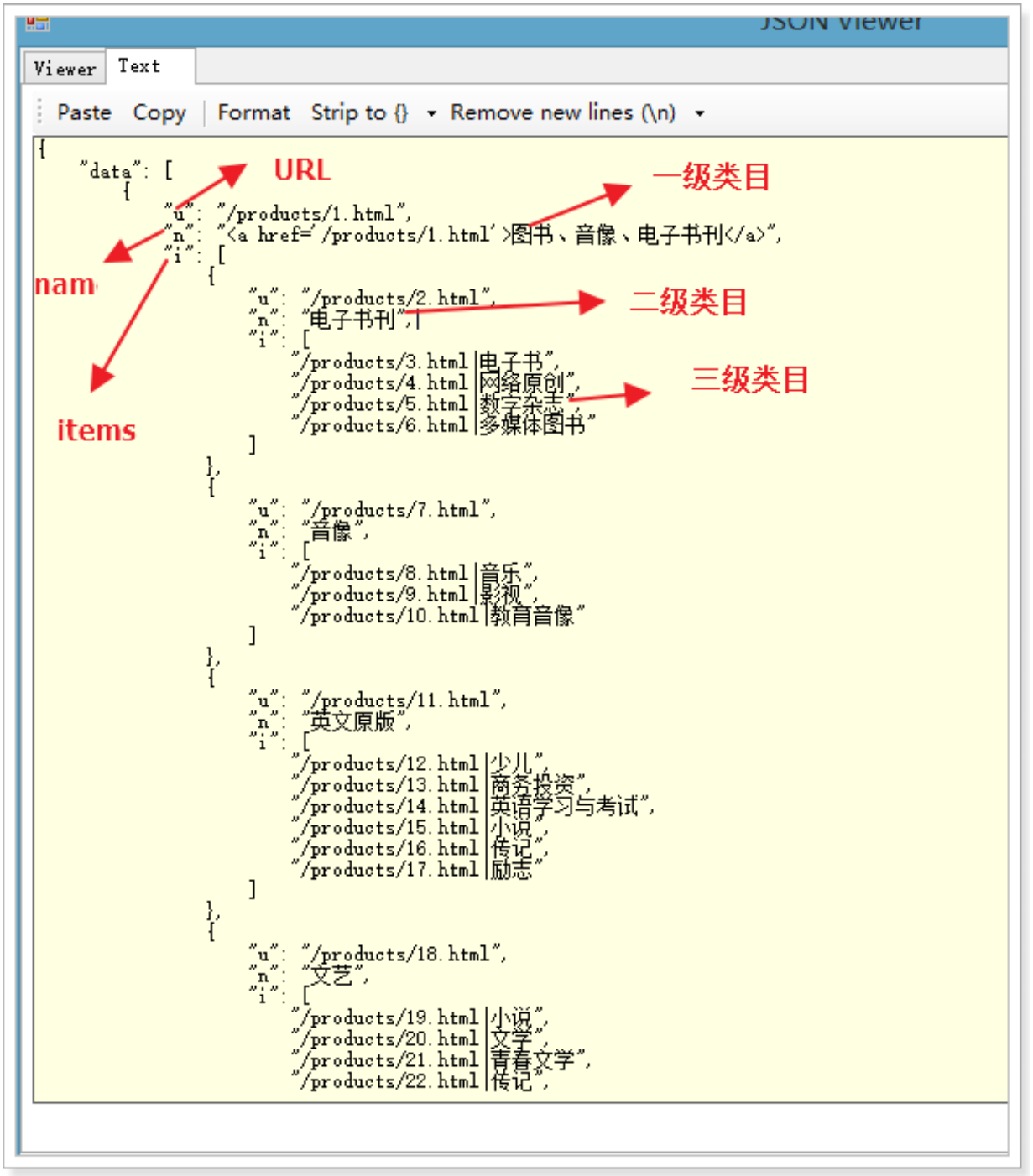
```
<script type="text/javascript">
    var itemAddEditor ;
    $(function(){
        itemAddEditor = TAOTAO.createEditor("#itemAddForm [name=desc]");
        TAOTAO.init({fun:function(node){
            TAOTAO.changeItemParam(node, "itemAddForm");
        }});
    });

```

动态生成form表单内容:

taotao-web

4、首页左侧商品类目栏的显示，达到鼠标放上去后能显示其叶子节点

1)处理方法在后台开放一个接口，让前台发送请求获取数据，优点:
i. 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
ii. 数据安全
iii. 前端团队无需了解学习后端团队的底层数据库结构
iv. 后端团队可以在接口处添加缓存逻辑
若直接从数据库中获取
i. 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）
ii. 前端系统团队需要有学习的成本，才能使用数据库
iii. 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
前台需要的json数据格式如下:



后台系统开发接口返回数据

定义ItemCatReult

```java
public class ItemCatResult {

    @JsonProperty("data")//指定对象序列化json时的名称
    private List<ItemCatData> itemCats = new ArrayList<ItemCatData>();

    public List<ItemCatData> getItemCats() {
        return itemCats;
    }

    public void setItemCats(List<ItemCatData> itemCats) {
        this.itemCats = itemCats;
    }

}
```

```java
public class ItemCatData {

    @JsonProperty("u")
    // 序列化成json数据时为 u
    private String url;

    @JsonProperty("n")
    private String name;

    @JsonProperty("i")
    private List<?> items;

```

Controller

@RequestMapping(“api/item/cat”)

//前台ajax发送请求url如下,跨域

//URL_Serv: “http://localhost:8989/rest/api/item/cat?callback=category.getDataService“,

```
public class ApiItemCatController {
@Autowired
private ItemCatService itemCatService;
/*
 * 对外提供的查询商品类目的数据的接口
 * */
@RequestMapping(method=RequestMethod.GET)
public ResponseEntity<ItemCatResult> queryItemCat(){
    try{
        ItemCatResult itemCatResult = this.itemCatService.queryAllT
oTree();
        return ResponseEntity.ok(itemCatResult);
    }catch(Exception e){
        e.printStackTrace();
    }
    //500
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(null);
}
```

面临跨域问题

前台:www.taotao.com------>manage.taotao.com

浏览器对ajax请求的限制，不允许跨域请求资源。

不同的域名或不同的端口都是跨域请求。

解决方式：借助script标签的src进行加载数据，可解决。

json.jsp页面**后台**

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
    <%
String callback = request.getParameter("callback");
if(callback != null){
    out.print(callback+"({\"abc\":123})");
}else{
    out.print("{\"abc\":123}");
}
/*  out.print(fun"({\"abc\":123})"); */
%>
```

test-json.html **前台**

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=IS
O-8859-1">
    <title>test-json</title>
    </head>
    <body>
        <script type="text/javascript">
            function fun(data){
                alert(data.abc);
            }
        </script>
```

```
        <script type="text/javascript" src="http://manage.taotao.co
m/json.jsp?callback=fun"></script>
    </body>
    </html>

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
```

通过jQuery使用jsonp请求

```
<script type="text/javascript">
                alert($);
                   $(function(){
                       $.ajax({
                           type:"GET",
                url:"http://localhost/json.jsp",
                           dataType:"jsonp",
                           success:function(data){
                           alert(data.abc);
                       }
                       });
                   });
//http://localhost/json.jsp?callback=jQuery111107145450099902366_14
73946456281&_=147394645628    ajax在解析的时候会自动携带一个callback
    </script>
```

解决项目中跨域问题

后台系统Controller

```java
/**
 * 对外提供接口查询商品类目数据
 *
 * @return
 */
@RequestMapping(method = RequestMethod.GET)
public ResponseEntity<String> queryItemCat(@RequestParam("callback") String callback) {
    try {
        ItemCatResult itemCatResult = this.itemCatService.queryAllToTree();
        String json = MAPPER.writeValueAsString(itemCatResult);
        if (StringUtils.isEmpty(callback)) {
            return ResponseEntity.ok(json);
        }
        return ResponseEntity.ok(callback + "(" + json + ");");
    } catch (Exception e) {
        e.printStackTrace();
    }
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
}
```

这样可以解决但是出现了乱码,解决乱码问题

在SpringMVC中产生的响应有2类：

1、 ModelAndView

2、 返回数据响应

a) 使用消息转化器完成

解决一：看spring的源码分析知道，默认的消息转换器StringHttpMessageConvert编码是iso8859-1,在注解驱动中配置其构造方法，设置编码格式为utf-8

```xml
    <!-- 定义注解驱动 -->
<mvc:annotation-driven>
    <mvc:message-converters register-defaults="true">
    <bean class="org.springframework.http.converter.StringHttpMessa
geConverter">
            <constructor-arg index="0" value="UTF-8"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

解决方案二：

统一支持jsonp ，扩展CallbackMappingJackson2HttpMessageConverter

```
 public class CallbackMappingJackson2HttpMessageConverter extends M
appingJackson2HttpMessageConverter {

// 做jsonp的支持的标识，在请求参数中加该参数
private String callbackName;

@Override
protected void writeInternal(Object object, HttpOutputMessage outpu
tMessage) throws IOException,
        HttpMessageNotWritableException {
    // 从threadLocal中获取当前的Request对象
    HttpServletRequest request = ((ServletRequestAttributes) Reques
tContextHolder
            .currentRequestAttributes()).getRequest();
    String callbackParam = request.getParameter(callbackName);
    if (StringUtils.isEmpty(callbackParam)) {
        // 没有找到callback参数，直接返回json数据
        super.writeInternal(object, outputMessage);
    } else {
        JsonEncoding encoding = getJsonEncoding(outputMessage.getHe
aders().getContentType());
        try {
            String result = callbackParam + "(" + super.getObjectMa
pper().writeValueAsString(object)
                    + ");";
            IOUtils.write(result, outputMessage.getBody(), encodin
g.getJavaName());
        } catch (JsonProcessingException ex) {
            throw new HttpMessageNotWritableException("Could not wr
ite JSON: " + ex.getMessage(), ex);
        }
    }

}

public String getCallbackName() {
    return callbackName;
}

public void setCallbackName(String callbackName) {
    this.callbackName = callbackName;
}
    }
```

配置:

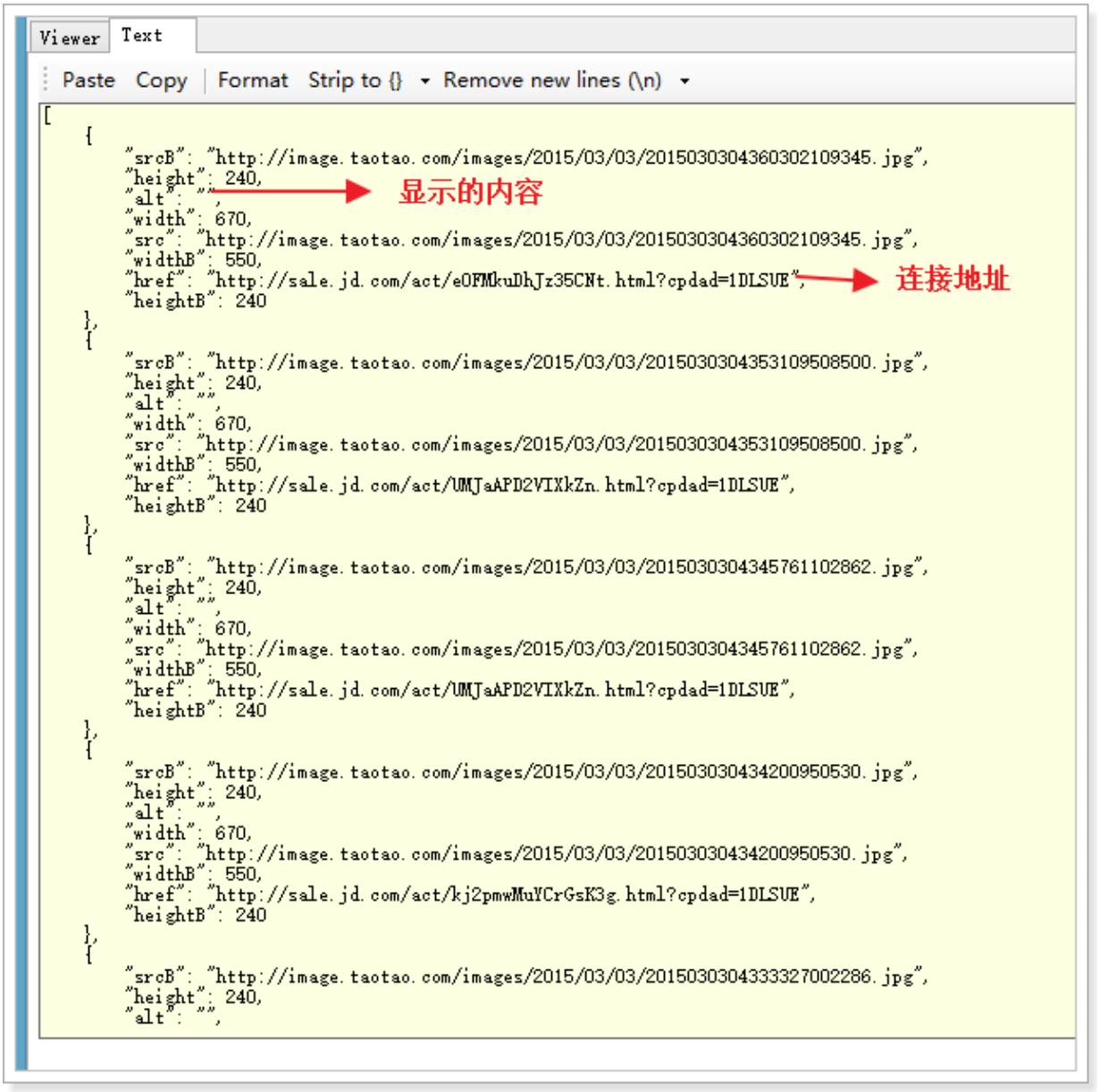
```
 <mvc:annotation-driven>
    <mvc:message-converters register-defaults="true">
    <bean class="com.taotao.common.spring.exetend.converter.json.Ca
llbackMappingJackson2HttpMessageConverter">
        <property name="callbackName" value="callback"/>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>
```

Controller实现

```
        /*
     * 对外提供的查询商品类目的数据的接口
     * */
    @RequestMapping(method=RequestMethod.GET)
    public ResponseEntity<ItemCatResult> queryItemCat(){
        try{
            ItemCatResult itemCatResult = this.itemCatService.query
AllToTree();
            return ResponseEntity.ok(itemCatResult);
        }catch(Exception e){
            e.printStackTrace();
        }
        //500
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(nul
l);
    }
```

5.首页的大广告

数据结构




```
[
    {
        "srcB": "http://image.taotao.com/images/2015030304360302109345.jpg",
        "height": 240,
        "alt": "",
        "width": 670,
        "src": "http://image.taotao.com/images/2015030304360302109345.jpg",
        "widthB": 550,
        "href": "http://sale.jd.com/act/eOFMkuDhJz35CNt.html?cpdad=1DLSUE",
        "heightB": 240
    },
    {
        "srcB": "http://image.taotao.com/images/2015030304353109508500.jpg",
        "height": 240,
        "alt": "",
        "width": 670,
        "src": "http://image.taotao.com/images/2015030304353109508500.jpg",
        "widthB": 550,
        "href": "http://sale.jd.com/act/UMJaAPD2VIXkZn.html?cpdad=1DLSUE",
        "heightB": 240
    },
    {
        "srcB": "http://image.taotao.com/images/2015030304345761102862.jpg",
        "height": 240,
        "alt": "",
        "width": 670,
        "src": "http://image.taotao.com/images/2015030304345761102862.jpg",
        "widthB": 550,
        "href": "http://sale.jd.com/act/UMJaAPD2VIXkZn.html?cpdad=1DLSUE",
        "heightB": 240
    },
    {
        "srcB": "http://image.taotao.com/images/2015030304342009505 30.jpg",
        "height": 240,
        "alt": "",
        "width": 670,
        "src": "http://image.taotao.com/images/2015030304342009505 30.jpg",
        "widthB": 550,
        "href": "http://sale.jd.com/act/kj2pmwMuYCrGsK3g.html?cpdad=1DLSUE",
        "heightB": 240
    },
    {
        "srcB": "http://image.taotao.com/images/2015030304333327002286.jpg",
        "height": 240,
        "alt": "",
```


方案：

1、 将首页显示的广告都抽象的看作是内容

2、 在后台系统中创建一张内容表

3、 创建一个内容分类表用于区分内容的分类

4、 后台系统对内容表以及分类表进行CRUD

5、 对外提供接口服务

6、 前端系统通过接口获取数据，进行封装，即可实现

```
CREATE TABLE `tb_content_category` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '类目ID',
  `parent_id` bigint(20) DEFAULT NULL COMMENT '父类目ID=0时，代表的是一级的类目',
  `name` varchar(50) DEFAULT NULL COMMENT '分类名称',
  `status` int(1) DEFAULT '1' COMMENT '状态。可选值:1(正常),2(删除)',
  `sort_order` int(4) DEFAULT NULL COMMENT '排列序号，表示同级类目的展现次序，如数值相等则按名称次序排列。取值范围:大于零的整数',
  `is_parent` tinyint(1) DEFAULT '1' COMMENT '该类目是否为父类目，1为true，0为false',
  `created` datetime DEFAULT NULL COMMENT '创建时间',
  `updated` datetime DEFAULT NULL COMMENT '创建时间',
  PRIMARY KEY (`id`),
  KEY `parent_id` (`parent_id`,`status`) USING BTREE,
  KEY `sort_order` (`sort_order`)
) ENGINE=InnoDB AUTO_INCREMENT=32 DEFAULT CHARSET=utf8 COMMENT='内容分类';

内容表
CREATE TABLE `tb_content` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `category_id` bigint(20) NOT NULL COMMENT '内容类目ID',
  `title` varchar(200) DEFAULT NULL COMMENT '内容标题',
  `sub_title` varchar(100) DEFAULT NULL COMMENT '子标题',
  `title_desc` varchar(500) DEFAULT NULL COMMENT '标题描述',
  `url` varchar(500) DEFAULT NULL COMMENT '链接',
  `pic` varchar(300) DEFAULT NULL COMMENT '图片绝对路径',
  `pic2` varchar(300) DEFAULT NULL COMMENT '图片2',
  `content` text COMMENT '内容',
  `created` datetime DEFAULT NULL,
  `updated` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `category_id` (`category_id`),
  KEY `updated` (`updated`)
) ENGINE=InnoDB AUTO_INCREMENT=30 DEFAULT CHARSET=utf8;
```

表结构中字段是否添加索引判断依据是什么？ – 字段是否是查询条件或者是排序条件。
是否将所有的字段都添加索引，来加快查询？ – 不行的
1、 索引会占用存储空间，索引越多，使用的存储空间越多
2、 插入数据，存储索引也会消耗时间，索引越多，插入数据的速度越慢
后台实现,达到动态添加、删除、编辑节点.
首页广告页面实现细节首页中前端页面广告显示是以json格式的数据相应，而后台总是从easyUi的datagride类型数据结构中获取，节点分别为total,rows.因此要把此类型的数据转化为需要的类型.做法:
service层中用httpclient的doGet(url), 获得json数据，然后：

```
formatToList(String jsonData, Class<?> calzz){
JsonNode jsonNode = MAPPER.readTree(jsonData);
        JsonNode data = jsonNode.get("rows");
        List<?> list = null;
if (data.isArray() && data.size() > 0) {
                list = MAPPER.readValue(data.traverse(),
                        MAPPER.getTypeFactory().constructCollection
Type(List.class, clazz));
            }
            return new EasyUIResult(jsonNode.get("total").intValu
e(), list);
}
 //解析json,生成前端所需要的json数据
            EasyUIResult easyUIResult = EasyUIResult.formatToList(j
sonData, Content.class);
            List<Map<String, Object>> result = new ArrayList<Map<St
ring, Object>>();
        for (Content content : (List<Content>) easyUIResult.getRow
s()) {
            Map<String, Object> map = new LinkedHashMap<String, Obj
ect>();
            map.put("srcB", content.getPic());
            map.put("height", 240);
            map.put("alt", content.getTitle());
            map.put("width", 670);
            map.put("src", content.getPic());
            map.put("widthB", 550);
            map.put("href", content.getUrl());
            map.put("heightB", 240);
            result.add(map);
        }
        return MAPPER.writeValueAsString(result);
        }
```

6. Httpclient和Spring的整合

ApplicationContexthttpclient.xml的配置
Service层的封装

```
@Service
public class ApiService implements BeanFactoryAware {

private BeanFactory beanFactory;

@Autowired(required=false)//若spring容器中有对象就注入没有就忽略
private RequestConfig requestConfig;

/**
 * 执行GET请求，响应200返回内容，404返回null
 *
 * @param url
 * @return
 * @throws ClientProtocolException
 * @throws IOException
 */
public String doGet(String url) throws ClientProtocolException, IOE
xception {
    System.out.println("----+++++++++++++++++++-"+ur
l+"++++++++++++++++++++++++++++++");
    // 创建http GET请求
    HttpGet httpGet = new HttpGet(url);
    httpGet.setConfig(requestConfig);
    CloseableHttpResponse response = null;
    System.out.println("--------------------"+url+"----------------
--------------------");
    try {
        // 执行请求
        response = getHttpClient().execute(httpGet);
        // 判断返回状态是否为200
        if (response.getStatusLine().getStatusCode() == 200) {
            String content = EntityUtils.toString(response.getEntit
y(), "UTF-8");
            return content;
        }
    } finally {
        if (response != null) {
            response.close();
        }
    }
    return null;
}
   /* public static void main(String[] args) throws ClientProtocolE
xception, IOException {
    String url = "http://localhost:8989/rest/content?categoryId=8&p
age=1&rows=20";
    HttpGet httpGet = new HttpGet(url);
```

```
    CloseableHttpResponse response = null;
    CloseableHttpClient closeableHttpClient = HttpClients.createDef
ault();;
    try {
        // 执行请求
        response = closeableHttpClient.execute(httpGet);
        // 判断返回状态是否为200
        if (response.getStatusLine().getStatusCode() == 200) {
            String content = EntityUtils.toString(response.getEntit
y(), "UTF-8");
            System.out.println(content);
//              return EntityUtils.toString(response.getEntity(), "U
TF-8");
        }
    } finally {
        if (response != null) {
            response.close();
        }
    }
//        return null;
}*/

/**
 * 带有参数的GET请求，响应200返回内容，404返回null
 *
 * @param url
 * @param params
 * @return
 * @throws ClientProtocolException
 * @throws IOException
 * @throws URISyntaxException
 */
public String doGet(String url, Map<String, String> params)
        throws ClientProtocolException, IOException, URISyntaxExcep
tion {
    URIBuilder builder = new URIBuilder(url);
    for (Map.Entry<String, String> entry : params.entrySet()) {
        builder.setParameter(entry.getKey(), entry.getValue());
    }
    return doGet(builder.build().toString());
}

/**
 * 执行post请求
 *
 * @param url
 * @param params
 * @return
 * @throws IOException
```

```
     */
    public HttpResult doPost(String url, Map<String, String> params) th
    rows IOException {
        // 创建http POST请求
        HttpPost httpPost = new HttpPost(url);
        httpPost.setConfig(requestConfig);
        if (null != params) {
            List<NameValuePair> parameters = new ArrayList<NameValuePai
    r>(0);
            for (Map.Entry<String, String> entry : params.entrySet()) {
                parameters.add(new BasicNameValuePair(entry.getKey(), e
    ntry.getValue()));
            }
            // 构造一个form表单式的实体
            UrlEncodedFormEntity formEntity = new UrlEncodedFormEntit
    y(parameters, "UTF-8");
            // 将请求实体设置到httpPost对象中
            httpPost.setEntity(formEntity);
        }

        CloseableHttpResponse response = null;
        try {
            // 执行请求
            response = getHttpClient().execute(httpPost);
            return new HttpResult(response.getStatusLine().getStatusCod
    e(),
                    EntityUtils.toString(response.getEntity(), "UTF-
    8"));
        } finally {
            if (response != null) {
                response.close();
            }
        }
    }

    /**
     * 执行post请求，发送json数据,在提交订单时使用
     * @param url
     * @param json
     * @throws IOException
     */
    public HttpResult doPostJson(String url, String json) throws IOExce
    ption {
        // 创建http POST请求
        HttpPost httpPost = new HttpPost(url);
        httpPost.setConfig(requestConfig);
        if (null != json) {
            // 构造一个字符串的实体
            StringEntity stringEntity = new StringEntity(json, ContentT
```

```
ype.APPLICATION_JSON);
            // 将请求实体设置到httpPost对象中
            httpPost.setEntity(stringEntity);
        }

        CloseableHttpResponse response = null;
        try {
            // 执行请求
            response = getHttpClient().execute(httpPost);
            return new HttpResult(response.getStatusLine().getStatusCod
e(), EntityUtils.toString(
                    response.getEntity(), "UTF-8"));
        } finally {
            if (response != null) {
                response.close();
            }
        }
    }
    private CloseableHttpClient getHttpClient() {
        return this.beanFactory.getBean(CloseableHttpClient.class);
    }
       @Override
        public void setBeanFactory(BeanFactory beanFactory) throws Bean
sException {
            this.beanFactory = beanFactory;
        }
    }
```

7、缓存的需求

大广告位数据,商品的类目数据无需每次查询后台系统的接口，可以在前台系统添加缓存，提高访问首页的速度.

实现:使用 redis缓存技术.

1、 单纯从缓存命中的角度来说，是Memcached要高，Redis和Memcache的差距不大

2、 但是，Redis提供的功能更加的强大

二者的区别是什么？

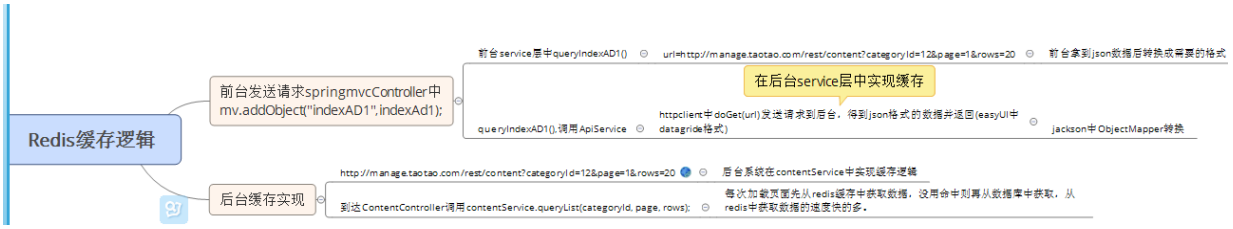
1、 Memcache是多线程

2、 Redis是单线程

xml文件中的配置

```
<bean class="redis.clients.jedis.ShardedJedisPool">
        <constructor-arg index="0">
            <bean class="redis.clients.jedis.JedisPoolConfig">
                <property name="maxTotal"     value="${redis.maxTot
al}"/>
            </bean>
        </constructor-arg>
        <constructor-arg index="1">
            <list>
                <bean class="redis.clients.jedis.JedisShardInfo">
                    <constructor-arg index="0" value="${redis.node
1.host}"/>
                    <constructor-arg index="1" value="${redis.node
1.port}"/>
                </bean>
            </list>
            <!-- 定义第二个集群节点
            <list>
                <bean class="redis.clients.jedis.JedisShardInfo">
                    <constructor-arg index="0" value="{redis.node
l.host}"/>
                    <constructor-arg index="1" value="{redis.node
l.port}"/>
                </bean>
            </list> -->
        </constructor-arg>
    </bean>
```

Service层中RedisService的优化,获取ShardedJedisPool,建立连接，设置set(),get(),del(),expire().
缓存逻辑



8. 商品基本数据显示

```
@RequestMapping(value="{itemId}",method=RequestMethod.GET)
public ModelAndView showDetail(@PathVariable("itemId") Long itemId)
{
    ModelAndView mv = new ModelAndView("item");
    //基本数据
    Item item = this.itemService.queryItemById(itemId);
    mv.addObject("item", item);
    //描述数据
    ItemDesc itemDesc = this.itemService.queryItemDescByItemId(item
Id);
    mv.addObject("itemDesc", itemDesc);

    //商品规格参数数据
    String itemParam = this.itemService.queryItemParamByItemId(item
Id);
        mv.addObject("itemParam", itemParam);

    return mv;
}
```

在Service层中设置缓存

9. 商品数据同步问题

后台系统中将商品修改，仅仅是在数据库中做修改，redis缓存中并没有修改，前台系统仍然从数据库中得到数据，没有进行数据的同步，导致前端系统不能够实时显示最新的数据。
暂时做法
1、 在前台系统中开发一个接口

```
@Controller
@RequestMapping("item/cache")
public class ItemCacheController {
@Autowired
private RedisService redisService;
/*
 * 删除redis中商品的缓存数据
 * */
@RequestMapping(value="{itemId}",method=RequestMethod.POST)
public ResponseEntity<Void> deleteCache(@PathVariable("itemId") Lon
g itemId){
    try{
        String key = ItemService.REDIS_KEY + itemId;
        this.redisService.del(key);
    }catch(Exception e){
        e.printStackTrace();
    }
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERRO
R).build();
}
```

}

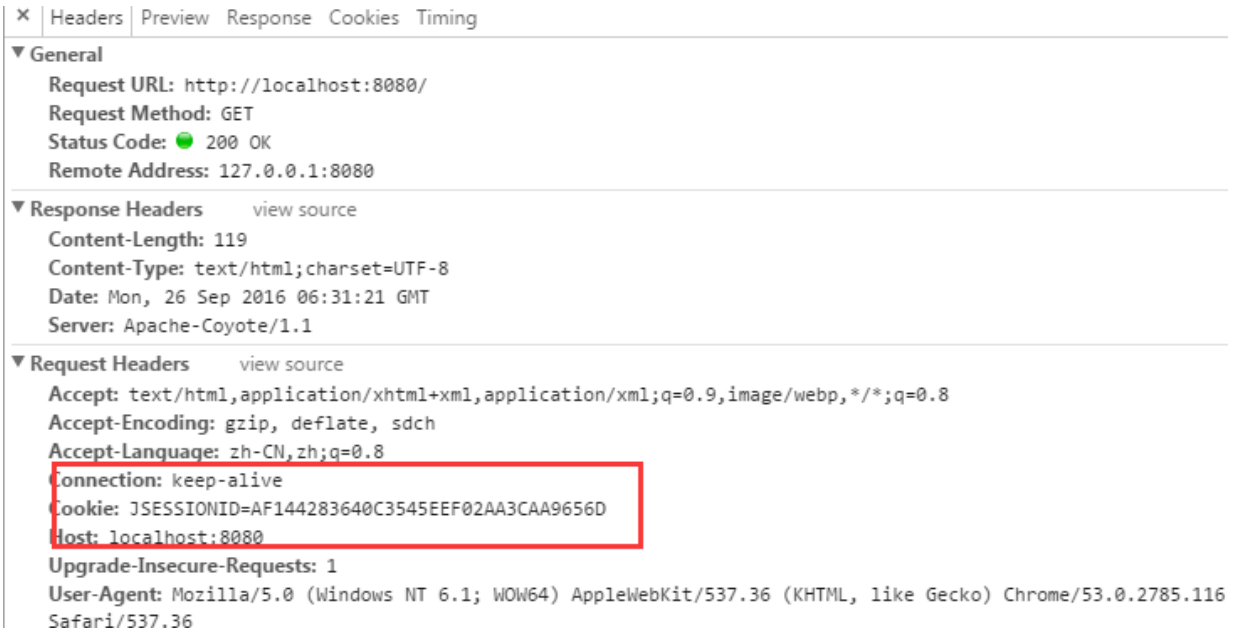
后台系统中调用该接口,一旦数据发生改变就删除reids中的对应数据。

```
//编辑商品的service层
    public void editItem(Item item, String desc,String itemParams)
{
        // 强制设置不能修改的字段为空
        item.setStatus(null);
        item.setCreated(null);
        super.updateSelective(item);

    // 修改商品描述数据
    ItemDesc itemDesc = new ItemDesc();
    itemDesc.setItemId(item.getId());
    itemDesc.setItemDesc(desc);
    this.itemDescService.updateSelective(itemDesc);

    //修改商品规格参数数据
    this.itemParamItemService.updateItemParamItem(item.getId(),item
Params);

    try{
        //通知其他系统该商品已经更新
        String url = TAOTAO_WEB_URL + "/item/cache"+item.getId() +
".html";
//            System.out.println(ur
l+"==============================================");

        this.apiService.doPost(url, null);

    }catch(Exception e){
        e.printStackTrace();
    }}
```

这样做用通知的形式告诉前台，代码的耦合度太高了，采用消息队列的形式解决。

10.单点登录系统

之前实现的登录和注册是在同一个tomcat内部完成，我们现在的系统架构是每一个系统都是由一个团队进行维护，每个系统都是单独部署运行一个单独的tomcat，所以，不能将用户的登录信息保存到session中（多个tomcat的session是不能共享的），所以我们需要一个单独的系统来维护用户的登录信息。
请求—->request—->tomcat会生成一个session域，后台把用户名密码保存在session中，user对象保存在session中。同时session域会生成一个id=JSESSIONID,保存在cookie中。Cookie:JSESSIONID=AF144283640C3545EEF02AA3CAA9656D,以后用户再次访问，服务器都会去cookie查找jsessionid来判断是否是对应的用户。再次访问可以看到在request Headers中有Cookie:JSESSIONID=AF144283640C3545EEF02AA3CAA9656D



SSO

是在多个应用系统中，用户只需要登录一次就可以访问整个系统，他可以把这次主要的登录映射到其他应用中用于同一个用户的登录机制。

实现过程

tb_user.sql

CREATE TABLE `tb_user` (

`id` bigint(20) NOT NULL AUTO_INCREMENT,

`username` varchar(50) NOT NULL COMMENT '用户名',

`password` varchar(32) NOT NULL COMMENT '密码，加密存储',

`phone` varchar(20) DEFAULT NULL COMMENT '注册手机号',

`email` varchar(50) DEFAULT NULL COMMENT '注册邮箱',

`created` datetime NOT NULL,

`updated` datetime NOT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `username` (`username`) USING BTREE,

UNIQUE KEY `phone` (`phone`) USING BTREE,

UNIQUE KEY `email` (`email`) USING BTREE

) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8 COMMENT='用户表';

后台实现

```
/*
* 注册时检验用户名
* */ @RequestMapping(value="check/{param}/{type}",method= RequestMethod.GET)
public ResponseEntity check(@PathVariable("param") String param,
@PathVariable("type") Integer type){
try{
Boolean bool = this.userService.check(param,type);
if(null==bool){
return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
}
// 为了兼容前端的业务逻辑，做出妥协处理
```

```
return ResponseEntity.ok(!bool);
}catch(Exception e){
e.printStackTrace();
}
return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null);
}
```

```
public Boolean check(String param, Integer type) {
    if (type < 1 || type > 3) {
        return null;
    }
    User record = new User();
    switch (type) {
    case 1:
        record.setUsername(param);
        break;
    case 2:
        record.setPhone(param);
        break;
    case 3:
        record.setEmail(param);
        break;
    default:
        break;
    }
    return this.userMapper.selectOne(record) == null;
}
 //加入校验注解@Valid
   @RequestMapping(value = "doRegister", method = RequestMethod.POS
T)
   @ResponseBody
public Map<String, Object> doRegister(@Valid User user, BindingResu
lt bindingResult) {
    Map<String, Object> result = new HashMap<String, Object>();
    // 校验有错误时执行
    if (bindingResult.hasErrors()) {
        List<String> msgs = new ArrayList<String>();
        List<ObjectError> allErrors = bindingResult.getAllErrors();
        for (ObjectError objectError : allErrors) {
            String msg = objectError.getDefaultMessage();
            msgs.add(msg);
        }
        result.put("status", "400");
        result.put("data", StringUtils.join(msgs, '|'));
        return result;
    }

    Boolean bool = this.userService.saveUser(user);
    if (bool) {
        // 注册成功
        result.put("status", "200");
 public Boolean saveUser(User user) {
    user.setId(null);
    user.setCreated(new Date());
    user.setUpdated(user.getCreated());
```

```
        // 密码通过MD5进行加密处理
```

user.setPassword(DigestUtils.md5Hex(user.getPassword());
return this.userMapper.insert(user) == 1;
}

详细见sso.taotao.com工程

11.使用拦截器实现用户是否登录的校验

具体的业务逻辑

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletRes
ponse response, Object handler)
                        throws Exception {
    UserThreadLocal.set(null);//清空当前线程中的User对象
    //登录页面sso.taotao.com/user/login.html
    String loginUrl = propertieService.TAOTAO_SSO_URL + "/user/logi
n.html";
    String token = CookieUtils.getCookieValue(request, COOKIE_NAM
E);
    if (StringUtils.isEmpty(token)) {
        // 未登录状态,重定向到登录页面
        response.sendRedirect(loginUrl);
        return false;
    }
    //从redis中查询token对应的值
    User user = this.userService.queryUserByToken(token);
    if (null == user) {
        // 未登录状态
        response.sendRedirect(loginUrl);
        return false;
    }
    //处于登录状态

    UserThreadLocal.set(user); //将User对象放置到ThreadLocal中
    return true;
}
```

通过token进行了2次查询
一次是在拦截器中查询，一次是在Controller查询，存在性能和资源浪费问题。

如何将拦截器中的数据传递到Controller？

方案：
1、 将User对象放置到request对象中
2、 使用ThreadLocal实现
a) 进入tomcat和产生响应前，都处于同一个线程中

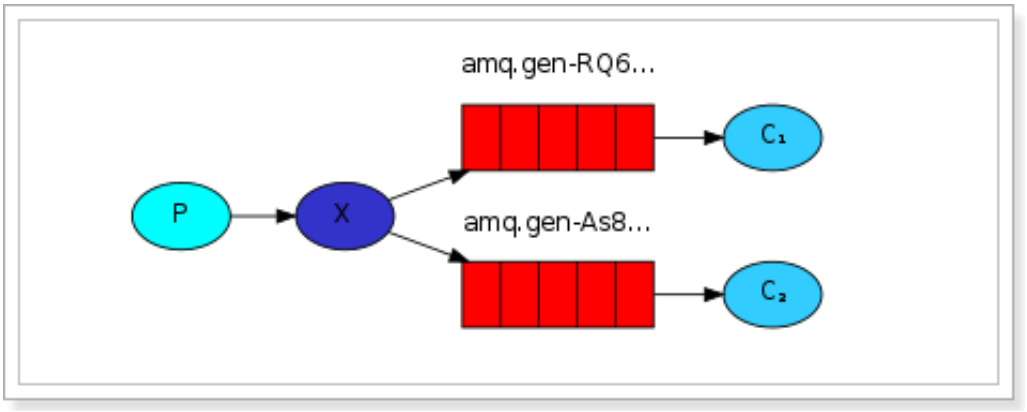
实现：
1、 定义ThreadLocal

12.search.taotao.com搜索功能的实现

slor的使用，具体见文档

13.rabbitMQ

MQ:消息队列，应用程序和应用程序之间通信的方法。
rabbitMQ:一个开源的在AMQP基础上完整的,可复用是企业消息系统.
问题: 如何实现商品数据同步.
以前解决：在前台系统开放接口，当2、 后台系统在商品编辑、删除时调用该接口,给前台系统发送通知，前台系统调用接口删除redis中的数据。该方案的问题系统之间的耦合性太高。商品的数据已经实现了和前台系统的同步，但是，搜索系统中索引数据没有和后台系统的数据同步，导致，后台系统的商品数据进行了更新，在搜索系统中搜索到的数据是旧的数据，也可以采用之前的方案解决，但是，因为系统间的耦合度太高了，所以不推荐使用该方案，所以需要使用MQ来解决该问题。
订阅者模式
1、1个生产者，多个消费者
2、每一个消费者都有自己的一个队列
3、生产者没有将消息直接发送到队列，而是发送到了交换机
4、每个队列都要绑定到交换机
5、生产者发送的消息，经过交换机，到达队列，实现，一个消息被多个消费者获取的目的



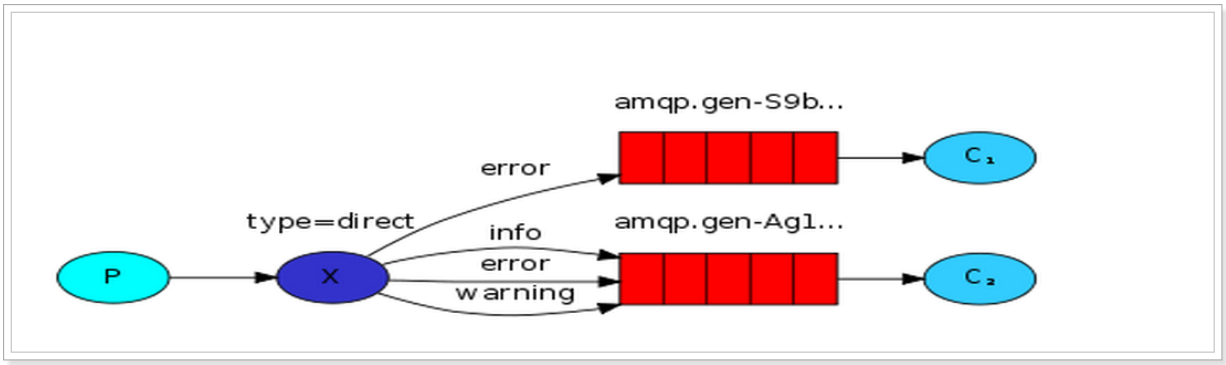
注意：消息发送到没有队列绑定的交换机时，消息将丢失，因为，交换机没有存储消息的能力，消息只能存在在队列中。
使用订阅模式可以实现商品数据的同步
后台系统就是消息的生产者。
前台系统和搜索系统是消息的消费者。
后台系统将消息发送到交换机中，前台系统和搜索系统都创建自己的队列，然后将队列绑定到交换机，即可实现。

消息，新增商品、修改商品、删除商品。

前台系统：修改商品、删除商品。
搜索系统：新增商品、修改商品、删除商品。

所以使用订阅模式实现商品数据的同步并不合理。

路由模式



生产者(后台系统)

```
    // 声明exchange,交换机类型是direct
    channel.exchangeDeclare(EXCHANGE_NAME, "direct");
    // 消息内容,delete消息的key
    String message = "Hello World!";
    channel.basicPublish(EXCHANGE_NAME, "delete", null, message.get
Bytes());
    System.out.println(" [x] Sent '" + message + "'");
```

消费者一(前台系统)

```
 // 声明队列      channel.queueDeclare(QUEUE_NAME, false, false, fals
e, null);
    // 前台系统绑定队列到交换机
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "update");
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "delete");
```

消费者二(搜索系统)

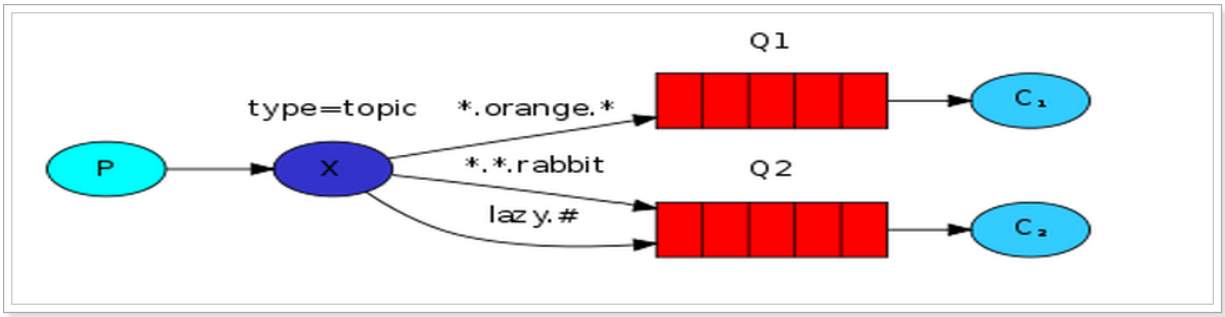
```
    // 声明队列
 channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    // 绑定队列到交换机
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "update");
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "delete");
    channel.queueBind(QUEUE_NAME, EXCHANGE_NAME, "add");
```

通配符模式

Topic Exchange – 将路由键和某模式进行匹配。此时队列需要绑定要一个模式上。符号“#”匹配一个或多个词，符号“*”匹配不多不少一个词。因此“audit.#”能够匹配到“audit.irs.corporate”，但是“audit.*” 只会匹配到“audit.irs”。我在 RedHat 的朋友做了一张不错的图，来表明 topic 交换机是如何工作的：

“#”匹配一个或多个词

“*”仅匹配一个词



生产者(后台系统)

```
    // 声明exchange,指定交换机类型是topic
   channel.exchangeDeclare(EXCHANGE_NAME, "topic");
    // 消息内容,消息的key为item.delete
    String message = "id=1000";
    channel.basicPublish(EXCHANGE_NAME, "item.delete", null, messag
e.getBytes());
    System.out.println(" [x] Sent '" + message + "'");
```

消费者(前台系统)

```
  channel.basicPublish(EXCHANGE_NAME, "item.update", null, messag
e.getBytes());
    channel.basicPublish(EXCHANGE_NAME, "item.delete", null, messag
e.getBytes());
```

消费者(搜索系统)

```
  channel.basicPublish(EXCHANGE_NAME, "item.#", null, message.getBy
tes());
```

持久化：将交换机或队列的数据保存到磁盘，服务器宕机或重启之后依然存在。
非持久化：将交换机或队列的数据保存到内存，服务器宕机或重启之后将不存在。
非持久化的性能高于持久化。

队列和交换机的绑定关系

实现：
1、 在配置文件中将队列和交换机完成绑定
2、 可以在管理界面中完成绑定
a) 绑定关系如果发生变化，需要修改配置文件，并且服务需要重启
b) 管理更加灵活
c) 更容易对绑定关系的权限管理，流程管理
具体实现
1)后台系统applicationContext-rabbitmq.xml

```
<!-- 定义RabbitMQ的连接工厂 -->
    <rabbit:connection-factory id="connectionFactory"
host="${rabbitmq.host}"
 port="${rabbitmq.port}" username="${rabbitmq.username}" passwor
d="${rabbitmq.password}"
virtual-host="${rabbitmq.vhost}" />
<!-- 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 -->
<rabbit:topic-exchange name="taotao-item-exchange" durable="true" a
uto-declare="true"/>
<!-- 定义模板 -->
<rabbit:template id="template" connection-factory="connectionFactor
y" exchange="taotao-item-exchange"/>
```

消息内容
方案：
1、 将Item对象做json序列化发送
a) 数据大
b) 有些数据其他人是可能用不到的
2、 发送商品的id、操作类型
后台商品更新时在后台系统service层发送消息

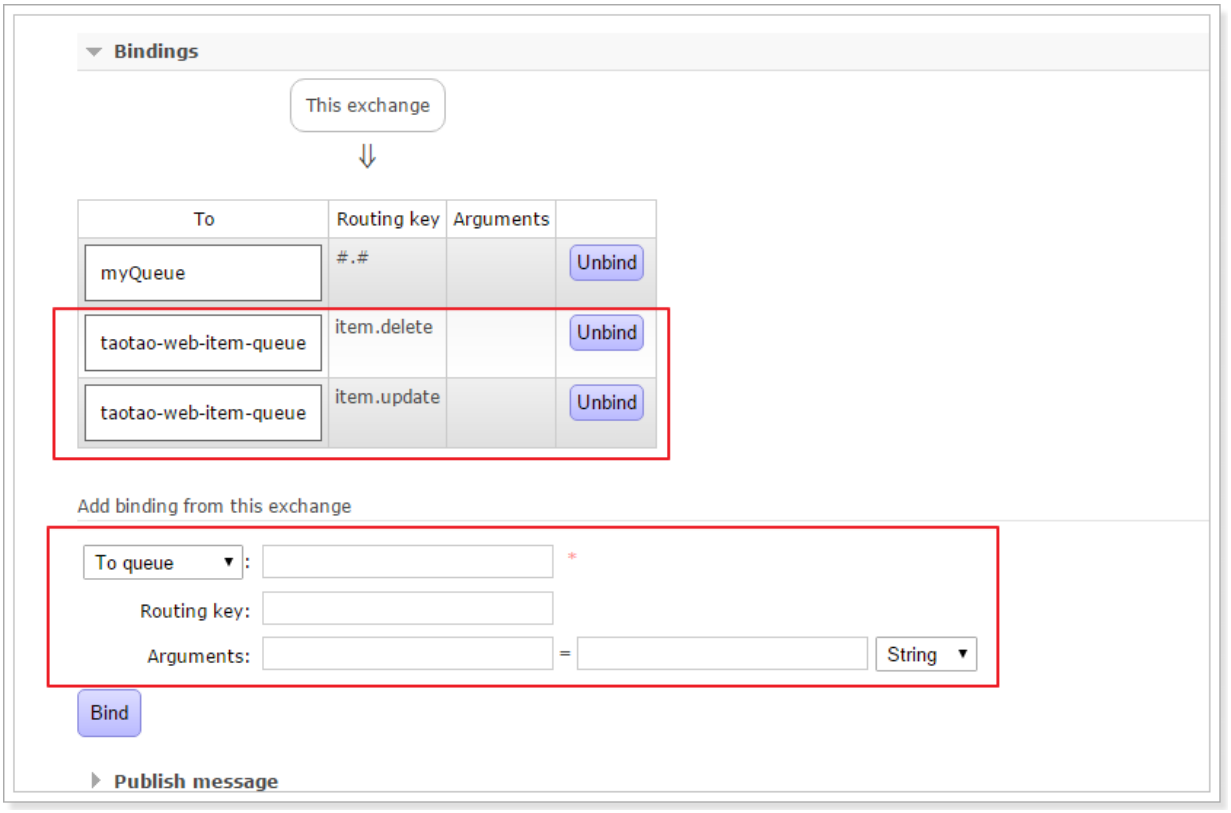
```
    // 发送消息
 sendMsg(item.getId(), "update");
 private void sendMsg(Long itemId,String type){
   try {
    //调用rabbitMQ发送消息,发送商品的id,操作类型
    Map<String,Object> msg = new HashMap<String,Object>();
    msg.put("itemId", itemId);
    msg.put("type", type);
    msg.put("date", System.currentTimeMillis());
    this.rabbitTemplate.convertAndSend("item." + type,MAPPER.writeV
alueAsString(msg));
}  catch (Exception e) {
    e.printStackTrace();
}
}
```

前台系统接收消息

配置

```
<!-- 定义队列，自动声明，durable="false"设置队列为非持久化-->
<rabbit:queue name="taotao-web-item-queue" auto-declare="true"/>
<!-- 定义消费者 -->
<bean id="itemMQHandler" class="com.taotao.web.mq.handler.ItemMQHan
dler"/>
<!-- 消费者监听 -->
<rabbit:listener-container connection-factory="connectionFactory">
    <rabbit:listener ref="itemMQHandler" method="execute" queue-nam
es="taotao-web-item-queue"/>
</rabbit:listener-container>

//前台系统接收到消息后根据id删除缓存中的数据当前数据
public void execute(String msg) {
    try {
        JsonNode jsonNode = MAPPER.readTree(msg);
        Long itemId = jsonNode.get("itemId").asLong();
        String key = ItemService.REDIS_KEY + itemId;
  this.redisService.del(key);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

在界面管理工具中完成交换机和队列绑定关系



搜索系统中接收消息

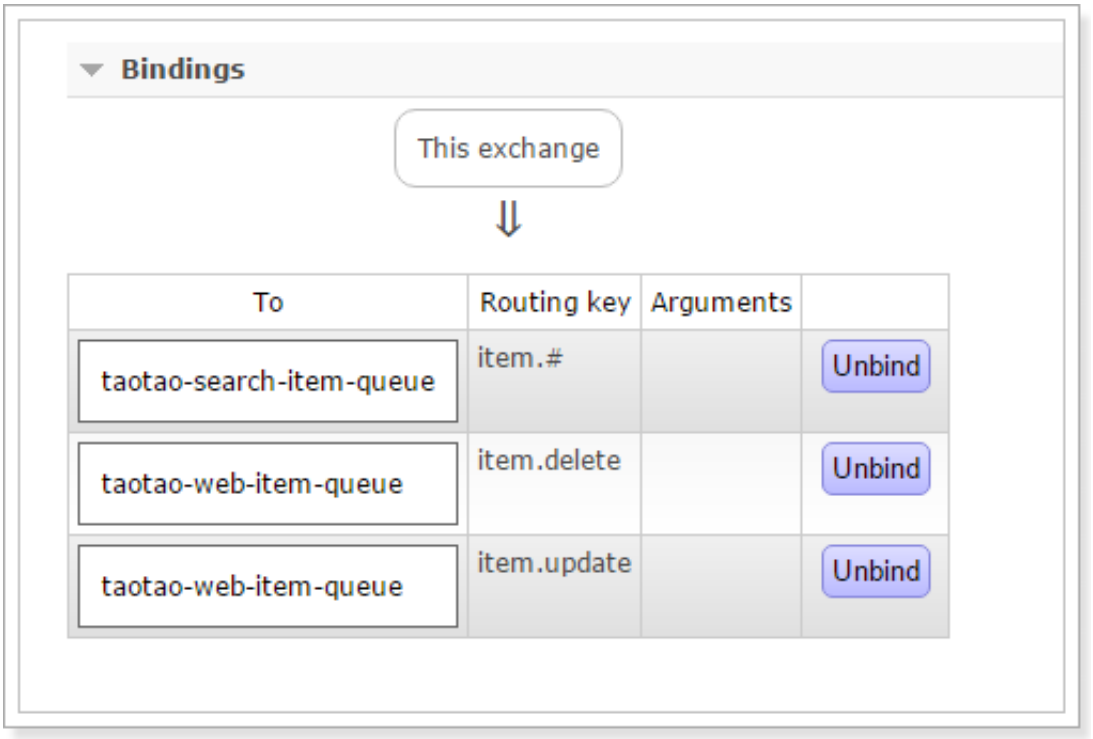
配置

```
<!-- MQ的管理，包括队列、交换器等 -->
<rabbit:admin connection-factory="connectionFactory" />
<!-- 定义队列，自动声明，durable="false"设置队列为非持久化-->
<rabbit:queue name="taotao-search-item-queue" auto-declare="true"/>
<!-- 定义消费者 -->
<bean id="itemMQHandler" class="com.taotao.search.mq.ItemMQHandle
r"/>
<!-- 消费者监听 -->
<rabbit:listener-container connection-factory="connectionFactory">
    <rabbit:listener ref="itemMQHandler" method="execute" queue-nam
es="taotao-search-item-queue"/>
</rabbit:listener-container>
```

处理业务逻辑,只处理insert(),update()两种方式

```
public void execute(String msg) {
        try{
            JsonNode jsonNode = MAPPER.readTree(msg);
            Long itemId = jsonNode.get("itemId").asLong();
            String type = jsonNode.get("type").asText();
            if(StringUtils.equals(type, "insert")||StringUtils.equa
ls(type, "update")){
                // 从后台系统中查询商品数据
                Item item = this.itemService.queryItemById(itemId);
                if (item != null) {
                    //id存在更新，不存在新增
                    this.httpSolrServer.addBean(item);
                    this.httpSolrServer.commit();
                }
            }else if(StringUtils.equals(type, "delete")){
                //删除索引数据
                this.httpSolrServer.deleteById(String.valueOf(itemI
d));
                this.httpSolrServer.commit();
            }
        }catch(Exception e){
            e.printStackTrace();
        }
```

在管理工具中绑定



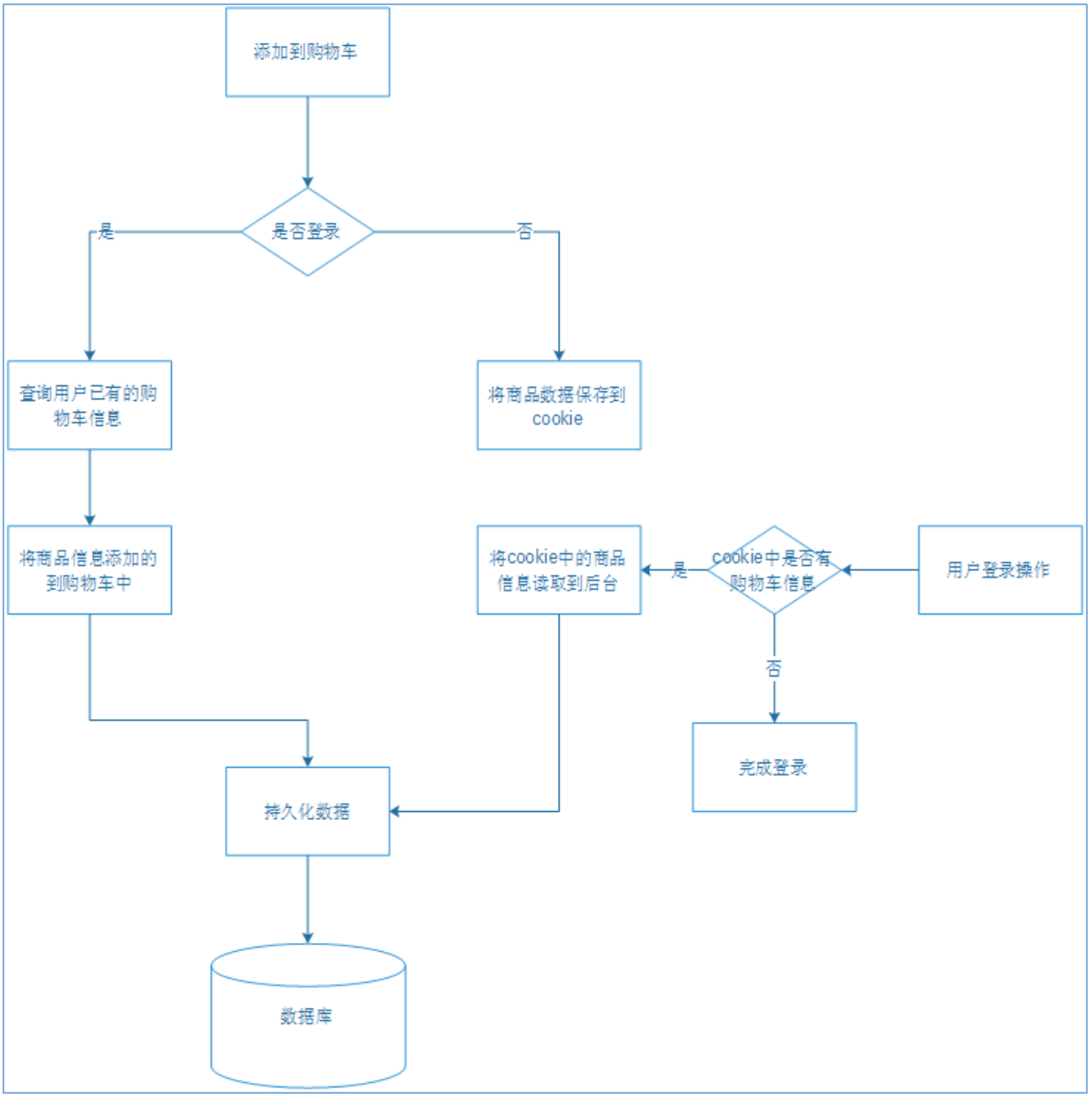
使用MQ实现商品数据的同步优势：

1、 降低系统间耦合度

2、 便于管理数据的同步

购物车功能的实现

业务流程



表结构

```
CREATE TABLE `tb_cart` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '自增ID',
  `user_id` bigint(20) DEFAULT NULL COMMENT '用户ID',
  `item_id` bigint(20) DEFAULT NULL COMMENT '商品ID',
  `item_title` varchar(100) DEFAULT NULL COMMENT '商品标题',
  `item_image` varchar(200) DEFAULT NULL COMMENT '商品主图',
  `item_price` bigint(20) DEFAULT NULL COMMENT '商品价格，单位为：分',
  `num` int(10) DEFAULT NULL COMMENT '购买数量',
  `created` datetime DEFAULT NULL,
  `updated` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `userId_itemId` (`user_id`,`item_id`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='购物车模块';
```

联合索引，注意索引的位置

使用联合索引，一定要注意索引字段的顺序。

拦截器的使用taotao-cart-web.xml中配置

```
<!-- springmvc拦截器  -->
    <mvc:interceptors>
        <mvc:interceptor>
            <mvc:mapping path="/cart/**"/>
            <bean class="com.taotao.cart.handlerInterceptor.UserLog
inHandlerInterceptor"/>
        </mvc:interceptor>
    </mvc:interceptors>
```

允许用户在未登录状态就把商品添加到购物车

```
 @Override
public boolean preHandle(HttpServletRequest request, HttpServletRes
ponse response, Object handler)
        throws Exception {
    UserThreadLocal.set(null);//清空当前线程中的User对象
    //登录页面sso.taotao.com/user/login.html
    String token = CookieUtils.getCookieValue(request, COOKIE_NAM
E);
    if (StringUtils.isEmpty(token)) {
        // 未登录状态,重定向到登录页面
//    response.sendRedirect(loginUrl);
        return true;//放行
    }
    //从redis中查询token对应的值
    User user = this.userService.queryUserByToken(token);
    if (null == user) {
        // 未登录状态
        return true;//放行
    }
    //处于登录状态
    UserThreadLocal.set(user); //将User对象放置到ThreadLocal中
    return true;
}
```

未登录状态下的购物车

以什么样数据格式保存购物车数据到cookie

可以使用json数据格式。

[

{

itemId:1001

itemTitle:”小米手机手机中的战斗机，欧耶”

……

}

]