# 京淘前台搭建

## 京淘前台搭建

### 分布式项目构建

根据分布式的思想,将项目进行拆分,将京淘项目拆分为前台和后台系统.

如图-1所示

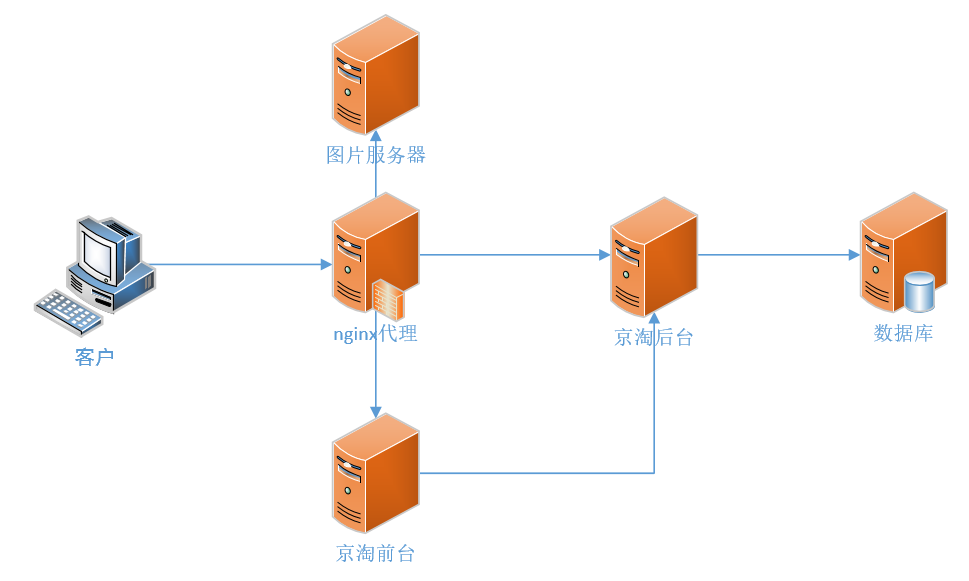


图- 1

### 创建项目

选择webapp骨架

如图-2所示

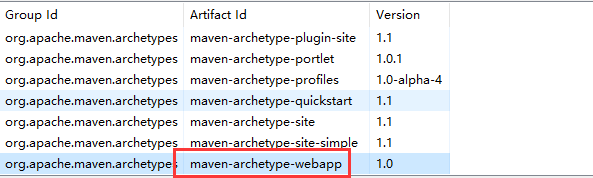


图- 2

2.定义项目名称

如图-3所示

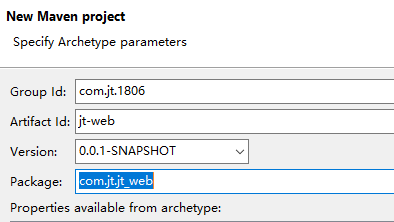


图- 3

### 添加继承和依赖

如图-4所示:添加继承

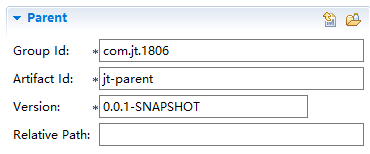


图- 4

如图-5所示:添加依赖

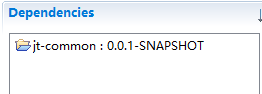


图- 5

### 添加tomcat插件

<build>

<plugins>

<plugin>

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

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

<version>2.2</version>

<configuration>

<port>8092</port>

<path>/</path>

</configuration>

</plugin>

</plugins>

</build>

通过eclipse配置tomcat插件,配置方式如下

如图-6所示

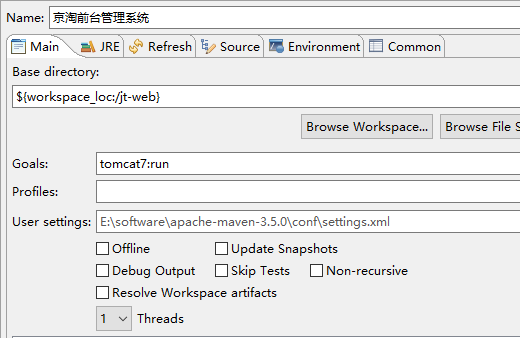


图- 6

配置源码

如图-7所示

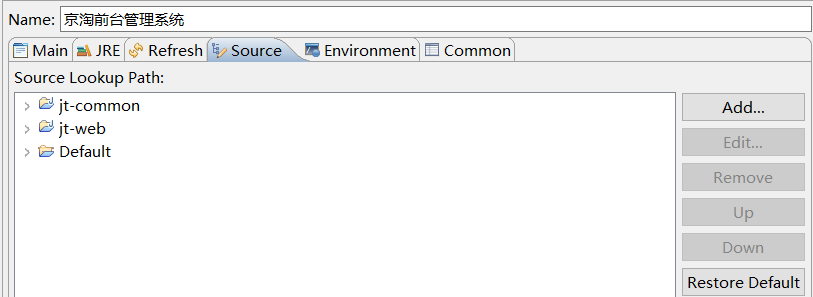


图- 7

## 配置文件编辑

### 配置文件说明

说明:jt-web主要负责与用户进行交互.项目中用到的全部的数据都通过特殊的方式从对应的服务器动态获取.前台只需要配置SpringMVC/Spring/静态资源文件即可.

### 导入静态配置文件

说明:将课前资料中的静态资源文件导入项目中

如图-8所示

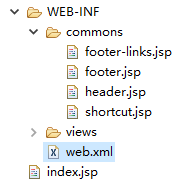


图- 8

### Web.xml配置文件说明

1.以监听器的方式启动Spring容器

<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>

对于监听器的说明:

有些就的企业现在一直沿用监听器的方式启spring容器.该方式其实是一种懒加载的方式.当SpringMVC容器需要实例化Controller对象时,并且内部需要注入业务层Service时,该操作会被监听器所拦截.这时启动Spring容器实例化对象.之后将对象注入注入.整个容器启动完成.

2.前端控制器拦截的策略

<servlet-mapping>

<servlet-name>springmvc-web</servlet-name>

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

</servlet-mapping>

<!-- 防止springMVC框架返回json时和html冲突报 406 错误 -->

<servlet-mapping>

<servlet-name>springmvc-web</servlet-name>

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

</servlet-mapping>

**\*.html 该操作拦截所有以.html结尾的请求**

**/service/\* 该配置拦截所有以/Service开头的请求**

### 静态页面动态页面优劣

1.静态页面优点:

1. 加载速度快
2. **容易被搜索引擎收录**,增加页面的友好性

2.静态页面缺点:

1. 用户体验感差/交互性差
2. 静态页面数据不安全.

3.动态页面优点:

1. 用户交互性强
2. 安全性更好

4.动态页面缺点:

1. 搜索引擎不会收录动态页面.

### 伪静态介绍:

伪静态是相对真实静态来讲的，通常我们为了增强[搜索引擎](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E/104812" \t "_blank)的友好面，都将文章内容生成静态页面，但是有的朋友为了实时的显示一些信息。或者还想运用动态脚本解决一些问题。不能用静态的方式来展示网站内容。但是这就损失了对搜索引擎的友好面。怎么样在两者之间找个中间方法呢，这就产生了伪静态技术。就是展示出来的是以html一类的静态页面形式，但其实是用ASP一类的动态脚本来处理的。

**总结:以html静态页面展现形式的动态页面技术.**

### Spring配置文件

1. 修改Spring配置文件

如图-9所示

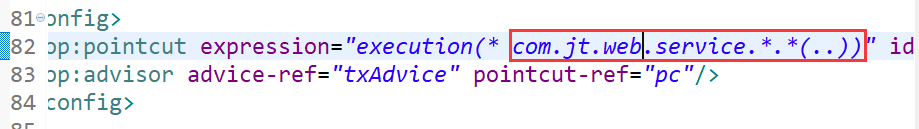


图- 9

1. 修改配置文件后效果

如图-10所示

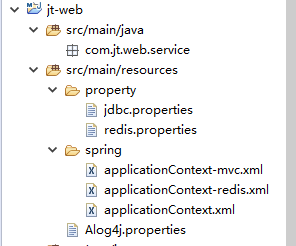


图- 10

### 展现京淘前台首页

1. 编辑Controller

@Controller

**public** **class** IndexController {

@RequestMapping("/index")

**public** String index(){

**return** "index";

}

}

1. 页面效果展现

如图-11所示



图- 11

# 实现商品信息展现

## 页面分析

### 需求说明

业务系统划分:

1. jt-manage 管理后台商品信息
2. jt-web 展现前台数据

当前台展现商品信息时,应该通过后台查询数据后,返回给前台页面.最终进行展现.

问题:

如何实现系统之间的数据访问???

思考:

Http协议.浏览器中使用最多的通信协议.

### HttpClient技术

说明:

HTTP 协议可能是现在 Internet 上使用得最多、最重要的协议了，越来越多的 Java 应用程序需要直接通过 HTTP 协议来访问网络资源。虽然在 JDK 的 java net包中已经提供了访问 HTTP 协议的基本功能，但是对于大部分应用程序来说，JDK 库本身提供的功能还不够丰富和灵活。HttpClient 是 Apache Jakarta Common 下的子项目，**用来提供高效的、最新的、功能丰富的支持 HTTP 协议的客户端编程工具包**，并且它支持 HTTP 协议最新的版本和建议。HttpClient 已经应用在很多的项目中，比如 Apache Jakarta 上很著名的另外两个开源项目 Cactus 和 [HTMLUnit](https://baike.baidu.com/item/HTMLUnit" \t "_blank) 都使用了 HttpClient。现在HttpClient最新版本为 HttpClient 4.5 .6（2015-09-11）

总结:HttpClient是java代码中发起Http请求协议的工具包

### 导入jar包

<!-- httpclient -->

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>${httpclient.version}</version>

</dependency>

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpmime</artifactId>

<version>4.3.1</version>

</dependency>

### HttpClient入门

**public** **class** TestHttpClient {

/\*\*

\* 模拟发起Get请求

\* 1.创建httpClient对象

\* 2.定义url路径

\* 3.定义请求方式 GET/POST

\* 4.发起request请求,获取response响应

\* 5.判断状态码是否正确

\* /200 表示请求正确

\* /400 请求参数异常

\* /406 返回结果与页面要求不匹配

\* /404 请求路径不匹配

\* /500 服务器异常

\* 6.获取响应结果

\* **@throws** IOException

\* **@throws** ClientProtocolException

\*/

@Test

**public** **void** get() **throws** ClientProtocolException, IOException {

CloseableHttpClient httpClient =

HttpClients.*createDefault*();

String url = "http://tv.cctv.com/2019/02/15/VIDEJIMBtGjqdO21lWKgSOnY190215.shtml?spm=C96370.PFyMLkh9dXI4.S24968.10";

HttpGet get = **new** HttpGet(url);

CloseableHttpResponse response =

httpClient.execute(get);

**if**(response.getStatusLine().getStatusCode()

== 200) {

System.***out***.println("恭喜你,学会了跨域系统访问");

String result =

EntityUtils.*toString*(response.getEntity());

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

}

}

}

## Spring容器整合HttpClient

### 编辑pro文件

#从连接池中获取到连接的最长时间

http.request.connectionRequestTimeout=500

#5000

http.request.connectTimeout=5000

#数据传输的最长时间

http.request.socketTimeout=30000

#提交请求前测试连接是否可用

http.request.staleConnectionCheckEnabled=true

#设置连接总数

http.pool.maxTotal=200

#设置每个地址的并发数

http.pool.defaultMaxPerRoute=100

### 编辑Spring配置文件

<!-- 定义httpclient连接池 -->

<bean id=*"httpClientConnectionManager"* class=*"org.apache.http.impl.conn.PoolingHttpClientConnectionManager"* destroy-method=*"close"*>

<!-- 设置连接总数 -->

<property name=*"maxTotal"* value=*"${http.pool.maxTotal}"*></property>

<!-- 设置每个地址的并发数 -->

<property name=*"defaultMaxPerRoute"* value=*"${http.pool.defaultMaxPerRoute}"*></property>

</bean>

<!-- 定义 HttpClient工厂，这里使用HttpClientBuilder构建-->

<bean id=*"httpClientBuilder"* class=*"org.apache.http.impl.client.HttpClientBuilder"* factory-method=*"create"*>

<property name=*"connectionManager"* ref=*"httpClientConnectionManager"*></property>

</bean>

<!-- 得到httpClient的实例 -->

<bean id=*"httpClient"* factory-bean=*"httpClientBuilder"* factory-method=*"build"*/>

<!-- 定期清理无效的连接 -->

<bean class=*"com.jt.common.util.IdleConnectionEvictor"* destroy-method=*"shutdown"*>

<constructor-arg index=*"0"* ref=*"httpClientConnectionManager"* />

<!-- 间隔一分钟清理一次 -->

<constructor-arg index=*"1"* value=*"60000"* />

</bean>

<!-- 定义requestConfig的工厂 -->

<bean id=*"requestConfigBuilder"* class=*"org.apache.http.client.config.RequestConfig.Builder"*>

<!-- 从连接池中获取到连接的最长时间 -->

<property name=*"connectionRequestTimeout"* value=*"${http.request.connectionRequestTimeout}"*/>

<!-- 创建连接的最长时间 -->

<property name=*"connectTimeout"* value=*"${http.request.connectTimeout}"*/>

<!-- 数据传输的最长时间 -->

<property name=*"socketTimeout"* value=*"${http.request.socketTimeout}"*/>

<!-- 提交请求前测试连接是否可用 -->

<property name=*"staleConnectionCheckEnabled"* value=*"${http.request.staleConnectionCheckEnabled}"*/>

</bean>

<!-- 得到requestConfig实例 -->

<bean id=*"requestConfig"* factory-bean=*"requestConfigBuilder"* factory-method=*"build"* />

### 编辑工具API

@Service

**public** **class** HttpClientService {

**private** **static** **final** Logger ***LOGGER*** = LoggerFactory.*getLogger*(HttpClientService.**class**);

@Autowired(required=**false**)

**private** CloseableHttpClient httpClient;

@Autowired(required=**false**)

**private** RequestConfig requestConfig;

/\*\*

\* 目的:简化业务层操作

\* 参数问题:

\* url:http://www.jd.com/addUser?id=1&name=tom

\* 1.url 2.Map<String,String> 3.字符集编码

\* 用户参数问题:

\* url:www.jd.com/addUser

\* map<k,v> id=1,name=tom

\*

\* 拼串传统写法:

\* /\*if(params != null) {

//url://www.jt.com/addUser?id=1&name=tom&

String tempUrl = url + "?";

for (Map.Entry<String,String> entry

: params.entrySet()) {

String key = entry.getKey();

String value = entry.getValue();

//www.jt.com/addUser?id=1&name=tom&

tempUrl = tempUrl+key+"="+value+"&";

}

//www.jt.com/addUser?id=1&name=tom

url = tempUrl.substring(0, tempUrl.length()-1);

}\*/

**public** String doGet(String url,Map<String,String> params,String encode) {

String result = **null**; //代表返回结果

//1.判断用户字符集编码格式是否为null

**if**(StringUtils.*isEmpty*(encode)) {

encode = "UTF-8";

}

**try** {

//2.判断参数是否为null

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

//2.1定义工具类

URIBuilder builder = **new** URIBuilder(url);

**for** (Map.Entry<String,String> entry:params.entrySet()) {

String key = entry.getKey();

String value = entry.getValue();

builder.addParameter(key,value);

}

url = builder.build().toString();

}

//3.定义请求类型

HttpGet httpGet = **new** HttpGet(url);

//3.1定义请求的超时时间

httpGet.setConfig(requestConfig);

//4.发起请求,获取响应

CloseableHttpResponse response =

httpClient.execute(httpGet);

//5.判断操作是否正确

**if**(response.getStatusLine().getStatusCode() == 200) {

result = EntityUtils.*toString*(response.getEntity(),encode);

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** result;

}

//为了用户体验重载方法

**public** String doGet(String url,

Map<String,String> params){

**return** doGet(url, params, **null**);

}

**public** String doGet(String url){

**return** doGet(url, **null**, **null**);

}

/\*\*

\* 1.定义请求方式httpPost

\* 2.将参数进行表单实体封装.

\* 3.发起url请求,获取返回值

\* **@param** url

\* **@param** params

\* **@param** charset

\* **@return**

\*/

**public** String doPost(String url,Map<String,String> params,String charset){

String result = **null**;

**if**(StringUtils.*isEmpty*(charset)){

charset = "UTF-8";

}

//1.定义请求类型

HttpPost post = **new** HttpPost(url);

post.setConfig(requestConfig); //定义链接时长

**try** {

//2.参数封装

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

List<BasicNameValuePair> parameters = **new** ArrayList<>();

//动态获取用户数据

**for** (Map.Entry<String,String> entry : params.entrySet()) {

parameters.add(

**new** BasicNameValuePair(entry.getKey(),entry.getValue()));

}

//封装FORM表单实体对象,作用传递参数

UrlEncodedFormEntity entity =

**new** UrlEncodedFormEntity(parameters,charset);

post.setEntity(entity);

}

//3.发起url请求.

CloseableHttpResponse httpResponse =

httpClient.execute(post);

//504 访问超时 500 服务器异常 406 浏览器解析参数异常 404 请求没有对应的处理方式

//400 参数提交到后台参数类型错误. 200 请求正常 304 浏览器已缓存

**if**(httpResponse.getStatusLine().getStatusCode() == 200){

result =

EntityUtils.*toString*(httpResponse.getEntity(),charset);

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** result;

}

**public** String doPost(String url,Map<String,String> params){

**return** doPost(url, params, **null**);

}

**public** String doPost(String url){

**return** doPost(url, **null**, **null**);

}

}