# 概念介绍

## 三层结构介绍

\*\*\*我们的开发架构一般都是基于两种形式，一种是 C/S 架构，也就是客户端/服务器；另一种是 **B/S 架构**，也就是浏览器服务器。在 JavaEE 开发中，几乎全都是基于 B/S 架构的开发。那么在 B/S 架构中，系统标准的三层架构包括：**表现层、业务层、持久层**。三层架构在我们的实际开发中使用的非常多，所以我们课程中的案例也都是基于三层架构设计的。

\*\*\*三层架构中，每一层各司其职，接下来我们就说说每层都负责哪些方面：

\*\*\***表现层**：

\* 也就是我们常说的web 层。它负责接收客户端请求，向客户端响应结果，通常客户端使用http 协议请求web 层，web 需要接收 http 请求，完成 http 响应。

\* 表现层包括展示层和控制层：控制层负责接收请求，展示层负责结果的展示。

\* 表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。

\* 表现层的设计一般都使用 MVC 模型。（MVC 是表现层的设计模型，和其他层没有关系）

\*\*\***业务层**：

\* 也就是我们常说的 service 层。它负责业务逻辑处理，和我们开发项目的需求息息相关。web 层依赖业务层，但是业务层不依赖 web 层。

\* 业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性。（也就是我们说的， 事务应该放到业务层来控制）

\*\*\***持久层**：

\* 也就是我们是常说的 dao 层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。通俗的讲，持久层就是和数据库交互，对数据库表进行曾删改查的。

## MVC设计模式介绍

\*\*\***MVC** 全名是 Model View Controller，是**模型(model)－视图(view)－控制器(controller)**的缩写， 是一种用于设计创建 Web 应用程序表现层的模式。MVC 中每个部分各司其职：

\*\*\*Model（模型）：

模型包含业务模型和数据模型，数据模型用于封装数据，业务模型用于处理业务。

\*\*\*View（视图）：

通常指的就是我们的 jsp 或者 html。作用一般就是展示数据的。

通常视图是依据模型数据创建的。

\*\*\*Controller（控制器）：

是应用程序中处理用户交互的部分。作用一般就是处理程序逻辑的。

## SpringMVC介绍

* **Spring MVC是什么？**

\*\*\*SpringMVC 是一种基于 Java 的实现 MVC 设计模型的请求驱动类型的轻量级 Web 框架，属于 SpringFrameWork 的后续产品，已经融合在 Spring Web Flow 里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用 Spring 进行 WEB 开发时，可以选择使用 Spring 的 Spring MVC 框架或集成其他 MVC 开发框架，如 Struts1(现在一般不用)，Struts2 等。

\*\*\*SpringMVC 已经成为*目前最主流的 MVC 框架*之一，并且**随着 Spring3.0 的发布，全面超越 Struts2，成为最优秀的 MVC 框架。**

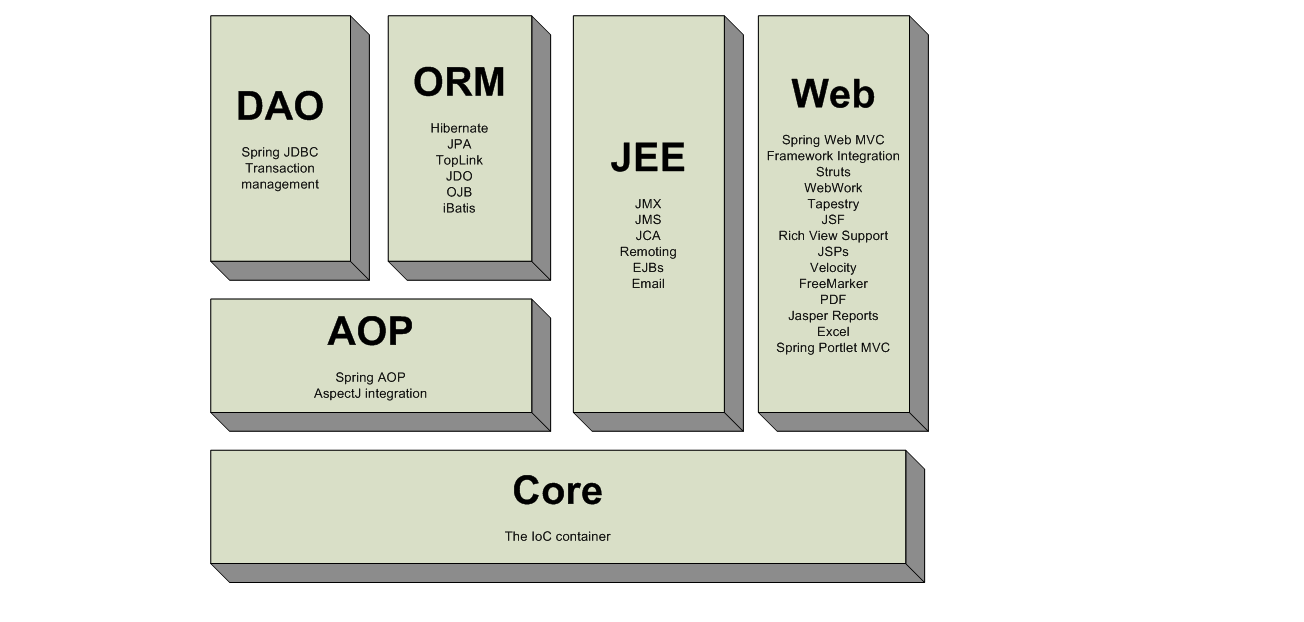
\*\*\*它通过一套注解，让一个简单的 Java 类成为处理请求的控制器，而无须实现任何接口。同时它还支持RESTful 编程风格的请求。

**总之：**

Spring MVC和Struts2一样，都是**为了解决表现层问题**的web框架，它们都是基于MCC设计模式的。而这些表现层框架的主要职责就是**处理前端HTTP请求**。

* **Spring MVC由来？**

Spring MVC全名叫Spring Web MVC，它是Spring家族Web模块的一个重要成员。这一点,我们可以从Spring的整体结构中看得出来：

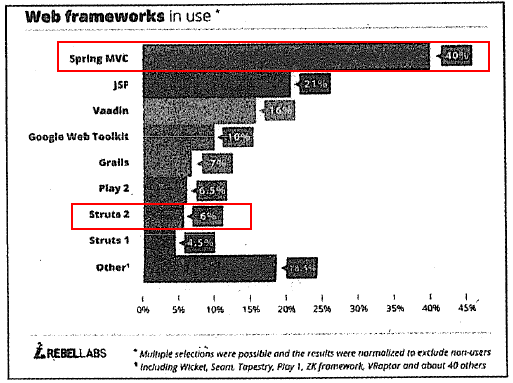


* **为什么学习SpringMVC？**

也许你要问，为什么要学习Spring MVC呢？struts2不才是主流吗？看SSH的概念有多火？

其实很多初学者混淆了一个概念，SSH实际上指的是Struts1.x+Spring+Hibernate。这个概念已经有十几年的历史了。在Struts1.x时代，它是当之无愧的霸主，但是在新的MVC框架涌现的时代，形式已经不是这样了，Struts2.x借助了Struts1.x的好名声，让国内开发人员认为Struts2.x是霸主继任者（其实两者在技术上无任何关系），导致国内程序员大多数学习基于Struts2.x的框架，又一个貌似很火的概念出来了S2SH（Struts2+Spring+Hibernate）整合开发。

不要再被蒙蔽了，看看下面的调查统计吧：



**SpringMVC的市场占有率是40%**，而Struts2只有可怜的6%。这已然说明了学习SpringMVC的必要性了，再说了，SpringMVC本身就是spring家族的一员，与整合spring时，SpringMVC根本无需中间整合包，而struts2得需要。

既然已经知道了SpringMVC的重要性了，那么下面就跟着我一起看看它的神奇之处吧！

*接下来，让我们一起看看SpringMVC是如何处理HTTP请求的？*

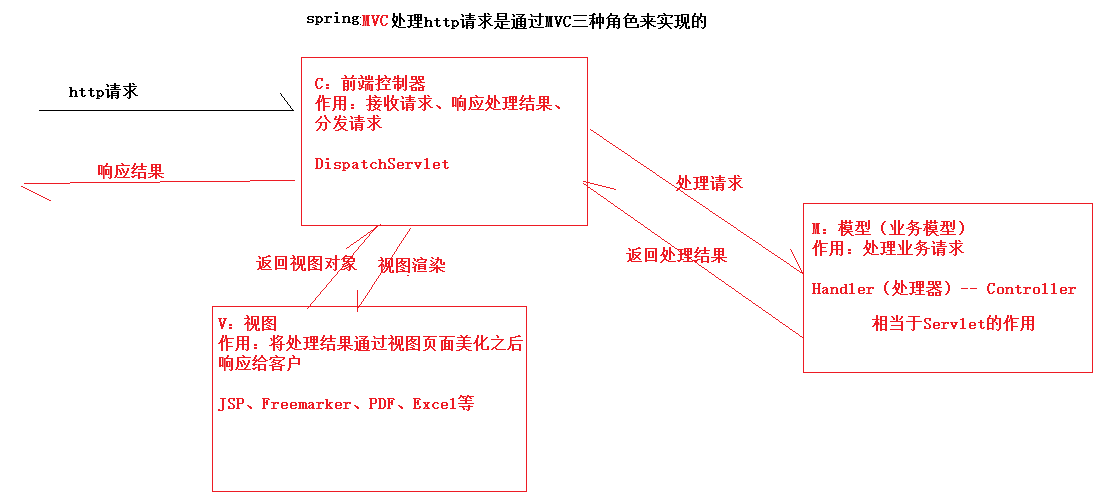
## SpringMVC如何处理请求？

*SpringMVC是基于MVC设计模式的，MVC模式指的就是Model（业务模型）、View（视图）、Controller（控制器）*。SpringMVC处理请求的过程就是通过MVC这三个角色的功能来实现的。

不过千万不要把MVC设计模式和工程的三层结构混淆，三层结构指的是表现层、业务层、数据持久层。而MVC只针对表现层进行设计。

下面，让我们来看看具体的处理流程吧！

流程来学校已经解除



# 入门程序

## 学习目标

* 如何配置前端控制器
* 如何开发处理器（方式有多种）

## 入门需求

访问/queryItem,返回商品列表页面，商品数据暂时使用静态数据（不从数据库查询）。

## 开发环境

本案例使用开发环境：

Jdk：jdk1.8

IDE：Oxygen

Tomcat：apache-tomcat-8.5.x

SpringMVC：5.0.7

Maven：3.3.9

## 工程搭建

### 创建一个maven web工程

工程名称为：springmvc

### POM文件添加依赖

就是spring ioc依赖、aop依赖、mvc依赖、jstl依赖还有servlet-api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>  <groupId>com.kkb</groupId>  <artifactId>springmvc-demo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>war</packaging>  <dependencies>  <!-- spring ioc组件需要的依赖包 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-expression</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <!-- 基于AspectJ的aop依赖 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <dependency>  <groupId>aopalliance</groupId>  <artifactId>aopalliance</artifactId>  <version>1.0</version>  </dependency>  <!-- spring MVC依赖包 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <!-- jstl -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  <version>1.2</version>  </dependency>  <!-- servlet -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <version>2.5</version>  <scope>provided</scope>  </dependency>  </dependencies>  <build>  <plugins>  <!-- 配置Maven的JDK编译级别 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.2</version>  <configuration>  <source>1.8</source>  <target>1.8</target>  <encoding>UTF-8</encoding>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <version>2.2</version>  <configuration>  <port>8080</port>  </configuration>  </plugin>  </plugins>  </build>  </project> |

## 开发步骤

### MVC中的C：配置前端控制器

在web.xml中添加DispatcherServlet的配置。

|  |
| --- |
| <?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"*  version=*"2.5"*>  <!-- 学习前置条件 -->  <!-- 问题1：web.xml中servelet、filter、listener、context-param加载顺序 -->  <!-- 问题2：load-on-startup标签的作用，影响了servlet对象创建的时机 -->  <!-- 问题3：url-pattern标签的配置方式有四种：/dispatcherServlet、 /servlet/\* 、\* 、/ ,以上四种配置，加载顺序是如何的-->  <!-- 问题4：url-pattern标签的配置为/\*报错，原因是它拦截了JSP请求，但是又不能处理JSP请求。为什么配置/就不拦截JSP请求，而配置/\*，就会拦截JSP请求-->  <!-- 问题5：配置了springmvc去读取spring配置文件之后，就产生了spring父子容器的问题 -->    <!-- 配置前端控制器 -->  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- 设置spring配置文件路径 -->  <!-- 如果不设置初始化参数，那么DispatcherServlet会读取默认路径下的配置文件 -->  <!-- 默认配置文件路径：/WEB-INF/springmvc-servlet.xml -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:springmvc.xml</param-value>  </init-param>  <!-- 指定初始化时机，设置为2，表示Tomcat启动时，DispatcherServlet会跟随着初始化 -->  <!-- 如果不指定初始化时机，DispatcherServlet就会在第一次被请求的时候，才会初始化，而且只会被初始化一次 -->  <load-on-startup>2</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!-- URL-PATTERN的设置 -->  <!-- 不要配置为/\*,否则报错 -->  <!-- 通俗解释：/\*，会拦截整个项目中的资源访问，包含JSP和静态资源的访问，对于静态资源的访问springMVC提供了默认的Handler处理器 -->  <!-- 但是对于JSP来讲，springmvc没有提供默认的处理器，我们也没有手动编写对于的处理器，此时按照springmvc的处理流程分析得知，它短路了 -->  <url-pattern>/</url-pattern>  </servlet-mapping>  <!-- <servlet> -->  <!-- <servlet-name>sss</servlet-name> -->  <!-- <servlet-class>sss</servlet-class> -->  <!-- </servlet> -->  <!-- <servlet-mapping> -->  <!-- <servlet-name>sss</servlet-name> -->  <!-- <url-pattern>/sss</url-pattern> -->  <!-- </servlet-mapping> -->  </web-app> |

### 创建springmvc.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc.xsd"*>  </beans> |

### MVC中的M：编写处理器

|  |
| --- |
| 处理器开发方式有多种：实现**HttpRequestHandler**接口、实现**Controller**接口、**注解方式**等。  不过企业开发中，推荐使用注解方式开发处理器，本课程讲解的就是注解方式。 |

*注解方式必要的注解主要有以下两个：*

* **@Controller**注解：在类上添加该注解，指定该类为一个请求处理器，不需要实现任何接口或者继承任何类。
* **@RequestMapping**注解：在方法上或者类上添加该注解，指定请求的url由该方法处理。如果url-pattern配置的是\*的话，url中的“”可以加也可以不加。

*处理器的返回值是ModelAndView对象，该对象的具体理解如下：*

**ModelAndView**：方法返回值对象，该对象包含两个功能：

一个是将数据存储到Request域中

一个是设置响应视图，比如将视图设置为“/WEB-INF/jsp/item-list.jsp”

*处理器代码：*

|  |
| --- |
| @Controller  **public** **class** ItemController {  @RequestMapping("/queryItem")  **public** ModelAndView queryItem() **throws** Exception {    List<Item> itemList = **new** ArrayList<>();    //商品列表  Item item\_1 = **new** Item();  item\_1.setName("联想笔记本");  item\_1.setPrice(6000f);  item\_1.setDetail("ThinkPad T430 联想笔记本电脑！");    Item item\_2 = **new** Item();  item\_2.setName("苹果手机");  item\_2.setPrice(5000f);  item\_2.setDetail("iphone6苹果手机！");    itemList.add(item\_1);  itemList.add(item\_2);  //创建modelandView对象  ModelAndView modelAndView = **new** ModelAndView();  //添加model  modelAndView.addObject("itemList", itemList);  //添加视图  modelAndView.setViewName("/WEB-INF/jsp/itemList.jsp");  **return** modelAndView;  }  } |

**注意：商品数据使用Item类描述，可以使用下面提供的pojo类。**

|  |
| --- |
| **public** **class** Item {  **private** Integer id;  **private** String name;  **private** Float price;  **private** String pic;  **private** Date createtime;  **private** String detail;  //get\set方法  //toString方法  } |

### MVC中的M：配置处理器

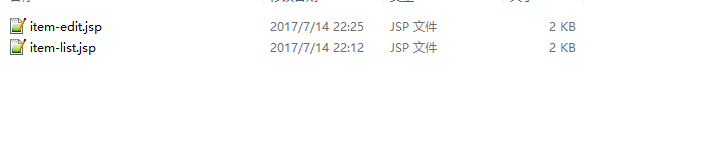
使用组件扫描器省去在spring容器配置每个controller类的繁琐。

使用<context:component-scan>自动扫描标记@controller的控制器类，在springmvc.xml配置如下：

<!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->

<context:component-scan base-package=*"com.kkb.springmvc.controller"*/>

### MVC中的V：创建视图



把参考资料中的item-list.jsp复制到工程的/WEB-INF/jsp目录下。

## 访问测试

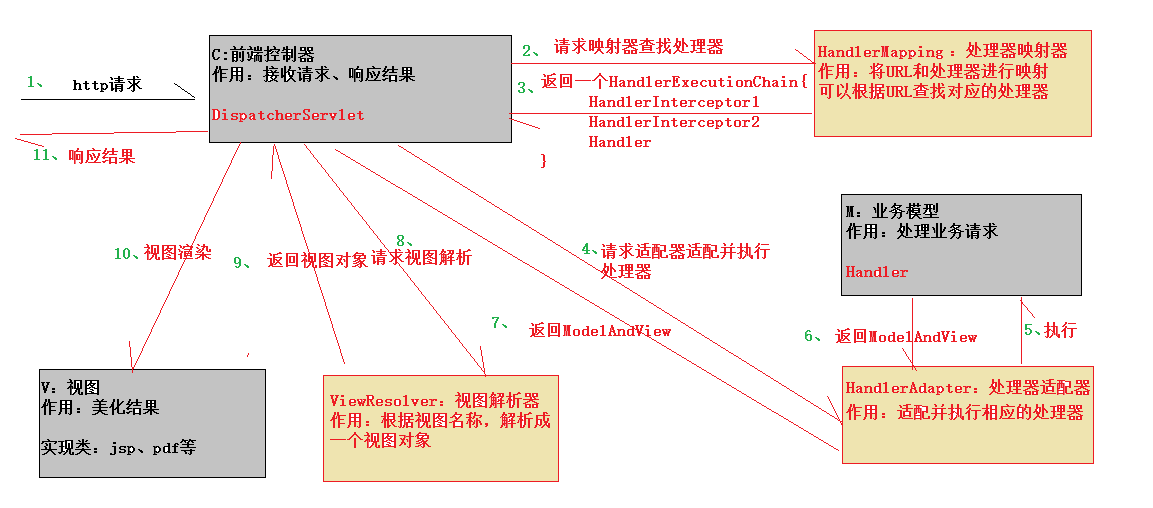
访问地址：<http://localhost:8080/springmvc/queryItem>

## 问题思考

* **前端控制器**是如何找到对应的处理器的？(Object)
* 前端控制器是如何执行对应的处理器的？Object
* 前端控制器是如何根据视图名称解析视图对象的？

# SpringMVC原理分析（重点）

## 框架结构



## 架构流程

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. HandlerAdapter执行处理器(handler，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将handler执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View对象
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户

## 组件说明

以下组件通常使用框架提供的实现：

* **DispatcherServlet**：前端控制器

用户请求到达前端控制器，它就相当于mvc模式中的C，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

* **HandlerMapping**：处理器映射器

HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

* **Handler**：处理器

Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

由于Handler涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发Handler。

* **HandlAdapter**：处理器适配器

通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

* **View Resolver**：视图解析器

View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。

* **View**：视图

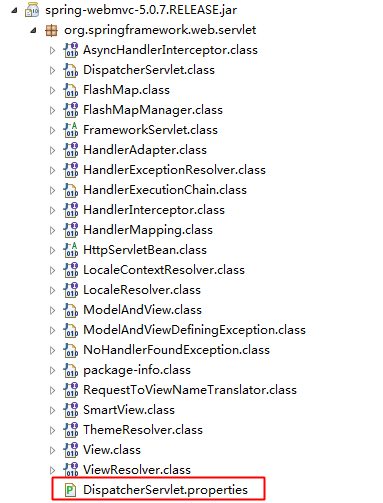
springmvc框架提供了很多的View视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是jsp。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

|  |
| --- |
| 说明：在springmvc的各个组件中，**处理器映射器、处理器适配器、视图解析器**称为springmvc的**三大组件**。  需要用户开发的组件有：**处理器、视图** |

## 默认配置文件

spring-webmvc-xxx.jar包中有一个DispatcherServlet.properties文件，该配置中默认加载了一些springmvc默认的其他组件，其中就包括三大组件。



|  |
| --- |
| # Default implementation classes for DispatcherServlet's strategy interfaces.  # Used as fallback when no matching beans are found in the DispatcherServlet context.  # Not meant to be customized by application developers.  org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver  org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver  org.springframework.web.servlet.**HandlerMapping**=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\  org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping  org.springframework.web.servlet.**HandlerAdapter**=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\  org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\  org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter  org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver,\  org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\  org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver  org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator  org.springframework.web.servlet.**ViewResolver**=org.springframework.web.servlet.view.InternalResourceViewResolver  org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager |

# 三大组件配置（注解方式）

## 注解映射器和适配器、

**显式、隐式**

### 通过bean标签配置

\*\*\***RequestMappingHandlerMapping**：注解式处理器映射器

\* 对类中标记@ResquestMapping的方法进行映射，根据ResquestMapping定义的url匹配ResquestMapping标记的方法，匹配成功返回HandlerMethod对象给前端控制器，HandlerMethod对象中封装url对应的方法Method。

\*\*\***RequestMappingHandlerAdapter**：注解式处理器适配器

\* 对标记@ResquestMapping的方法进行适配。

配置如下：

<!--注解映射器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"*/>

<!--注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*/>

### 通过mvc标签配置（推荐）

|  |
| --- |
| <mvc:annotation-drivern /> |

mvc:annotation-drivern标签的作用，就是往spring容器中注册了以下的一些BeanDefinition：（具体流程，参考《源码分析》章节）

* ContentNegotiationManagerFactoryBean
* **RequestMappingHandlerMapping**
* ConfigurableWebBindingInitializer
* **RequestMappingHandlerAdapter**
* CompositeUriComponentsContributorFactoryBean
* ConversionServiceExposingInterceptor
* MappedInterceptor
* ExceptionHandlerExceptionResolver
* ResponseStatusExceptionResolver
* DefaultHandlerExceptionResolver
* **BeanNameUrlHandlerMapping**
* **HttpRequestHandlerAdapter**
* **SimpleControllerHandlerAdapter**
* HandlerMappingIntrospector

## 视图解析器

在springmvc.xml文件配置如下：

|  |
| --- |
| <bean class=*"org.springframework.web.servlet.view.****InternalResourceViewResolver****"*>  <!-- 该视图解析器，默认的视图类就是JstlView，可以不写 -->  <property name=*"viewClass"*  value=*"org.springframework.web.servlet.view.JstlView"* />  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean> |

* **InternalResourceViewResolver**：默认支持**JSP视图**解析
* **viewClass**：JstlView表示JSP模板页面需要使用JSTL标签库，所以classpath中必须包含jstl的相关jar 包。*此属性可以不设置，默认为JstlView*。
* **prefix** 和**suffix**：查找视图页面的前缀和后缀，最终视图的址为：前缀+**逻辑视图名**+后缀，逻辑视图名需要在controller中返回的ModelAndView指定，比如逻辑视图名为hello，则最终返回的jsp视图地址 “WEB-INF/jsp/hello.jsp”

# SSM框架整合

## 整合思路

将工程的三层结构中的JavaBean分别使用Spring容器（**通过XML方式**）进行管理。

1. 整合持久层mapper，包括数据源、会话工程及mapper代理对象的整合；
2. 整合业务层Service，包括事务及service的bean的配置；
3. 整合表现层Controller，直接使用springmvc的配置。
4. Web.xml加载spring容器（包含多个XML文件）

**Spring核心配置文件：**

* applicationContext-dao.xml
* applicationContext-service.xml
* springmvc.xml

## 工程搭建

**依赖包：**

* spring（包括springmvc）、
* mybatis、
* mybatis-spring整合包、
* 数据库驱动、
* 第三方连接池
* JSTL
* servlet-api

## 工程整合（配置文件）

### 整合Mapper

#### applicationContext-dao.xml（核心）

在classpath下，创建spring目录，然后创建SqlMapConfig.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:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"*>  <!-- 加载db.properties -->  <context:property-placeholder location=*"classpath:****db.properties****"* />  <!-- 配置数据源 -->  <bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*  destroy-method=*"close"*>  <property name=*"driverClassName"* value=*"${jdbc.driver}"* />  <property name=*"url"* value=*"${jdbc.url}"* />  <property name=*"username"* value=*"${jdbc.username}"* />  <property name=*"password"* value=*"${jdbc.password}"* />  <property name=*"maxActive"* value=*"30"* />  <property name=*"maxIdle"* value=*"5"* />  </bean>  <!-- 配置SqlSessionFacotory -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 加载mybatis的配置文件（如果配置文件中没有配置项，可以忽略该文件） -->  <property name=*"configLocation"* value=*"classpath:****mybatis/SqlMapConfig.xml****"* />  <!-- 配置数据源 -->  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- 配置mapper扫描器，SqlSessionConfig.xml中的mapper配置去掉 -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <!-- 指定扫描的包 -->  <property name=*"basePackage"* value=*"****com.kkb.ssm.mapper****"* />  </bean>  </beans> |

#### SqlMapConfig.xml

在classpath下，创建mybatis目录，然后创建SqlMapConfig.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* ?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  </configuration> |

#### db.properties

在classpath下，创建db.properties：

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://localhost:3306/ssm?characterEncoding=utf8  jdbc.username=root  jdbc.password=root |

#### log4j.properties

在classpath下，创建log4j.properties：

|  |
| --- |
| #dev env [debug] product env [info]  log4j.rootLogger=DEBUG, stdout  # Console output...  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

### 整合Service

#### applicationContext-service.xml

在spring文件夹下创建applicationContext-service.xml，文件中配置service。在这个配置文件中，需要配置service的bean和事务管理。

|  |
| --- |
| <?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:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"*>  <!-- 扫描Service -->  <context:component-scan base-package=*"com.kkb.ssm.service"* />  <!-- 配置事务 -->  <!-- 事务管理器，对mybatis操作数据库进行事务控制，此处使用jdbc的事务控制 -->  <bean id=*"transactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <!-- 指定要进行事务管理的数据源 -->  <property name=*"dataSource"* ref=*"dataSource"*></property>  </bean>  <!-- 通知 -->  <tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>  <tx:attributes>  <!-- 传播行为 -->  <tx:method name=*"save\*"* propagation=*"REQUIRED"* />  <tx:method name=*"add\*"* propagation=*"REQUIRED"* />  <tx:method name=*"insert\*"* propagation=*"REQUIRED"* />  <tx:method name=*"delete\*"* propagation=*"REQUIRED"* />  <tx:method name=*"del\*"* propagation=*"REQUIRED"* />  <tx:method name=*"remove\*"* propagation=*"REQUIRED"* />  <tx:method name=*"update\*"* propagation=*"REQUIRED"* />  <tx:method name=*"modify\*"* propagation=*"REQUIRED"* />  <tx:method name=*"find\*"* read-only=*"true"* />  <tx:method name=*"query\*"* read-only=*"true"* />  <tx:method name=*"select\*"* read-only=*"true"* />  <tx:method name=*"get\*"* read-only=*"true"* />  </tx:attributes>  </tx:advice>  <!-- aop -->  <aop:config>  <aop:advisor advice-ref=*"txAdvice"*  pointcut=*"execution(\* com.kkb.ssm.service.impl.\*.\*(..))"* />  </aop:config>  </beans> |

### 整合Controller

Spring和springmvc之间无需整合，直接用springmvc的配置

#### web.xml

在web.xml中添加springmvc的配置：

|  |
| --- |
| <?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"* xmlns:web=*"http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*  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>ssm</display-name>  <!-- 配置springmvc的前端控制器 -->  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/springmvc.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  </web-app> |

#### springmvc.xml

在spring包下创建springmvc.xml文件，内容如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*>  <!-- 配置处理器映射器和处理器适配器 -->  <mvc:annotation-driven />  <!-- 配置视图解析器 -->  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean>  <!-- 使用注解的handler可以使用组件扫描器，加载handler -->  <context:component-scan base-package=*"com.kkb.ssm.controller"* />  </beans> |

### web.xml加载spring父容器

在web.xml中，使用监听器来对spring的配置文件进行加载：

|  |
| --- |
| <!-- 加载spring容器 -->  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>  classpath:spring/applicationContext-\*.xml  </param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener> |

## 整合测试（编写代码）

### 需求

实现商品查询列表，从mysql数据库查询商品信息。

### 需求分析

* **表现层**

请求URL：/queryItem

请求参数：无

请求返回值：ModelAndView 指定Model和View（item-list.jsp）

Request域（Model）：key为**itemList**

* **业务层**

业务处理逻辑（需求分析）：实现商品列表的查询

* **持久层**

只针对表进行增删改查操作

### 持久层代码

根据需求分析，持久层需要查询item表中的记录，使用逆向工程的代码即可。

通过逆向工程，把po类、mapper.xml和mapper.java类生成出来并拷贝到项目中。

### 业务层代码

根据需求开发service的接口以及实现类，**注意：使用注解 @Service开发service**。

|  |
| --- |
| @Service  **public** **class** ItemServiceImpl **implements** ItemService {  @Autowired  **private** ItemMapper mapper;  **public** List<Item> queryItemList() {  ItemExample example = **new** ItemExample();  **return** mapper.selectByExample(example);  }  } |

### 表现层代码

#### Controller类

在Controller类上添加 **@Controller** 注解

在Controller方法上添加 **@RequestMapping**注解进行url请求映射

|  |
| --- |
| @Controller  **public** **class** ItemController {  @Autowired  **private** ItemService service;  @RequestMapping("/queryItem")  **public** ModelAndView queryItem() {  // 根据查询条件去数据库中查询商品列表  List<Item> itemList = service.queryItemList();  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("itemList", itemList);  // 设置视图（配置逻辑视图）  modelAndView.setViewName("item/itemList");  **return** modelAndView;  }  } |

#### 开发JSP

入门程序中的JSP

#### 部署测试

<http://localhost:8080/ssm/queryItem>

# Controller方法返回值

## 不使用注解修饰

### 返回ModelAndView

controller方法中定义ModelAndView对象并返回，对象中可添加model数据、指定view。

### 返回void

在controller**方法形参上可以定义request和response**，使用request或response指定响应结果：

**void service(HttpServletRequest request,HttpServletResponse response){}**

1、使用request转发向页面，如下：

request.getRequestDispatcher("页面路径").forward(request, response);

2、也可以通过response页面重定向：

response.sendRedirect("url")

3、也可以通过response指定响应结果，例如响应json数据如下：

response.setCharacterEncoding("utf-8");

response.setContentType("application/json;charset=utf-8");

response.getWriter().write("json串");

### 返回字符串（推荐）

#### 逻辑视图名

**return** "item/item-list";

#### redirect重定向

**return** "redirect:**testRedirect**";

**redirect:**

\*\*\*相当于“response.sendRedirect()”

\*\*\*浏览器URL发生改变

\*\*\*Request域不能共享

#### forward转发

**return** "forward:**testForward**";

**forward：**

\*\*\*相当于“request.getRequestDispatcher().forward(request,response)”

\*\*\*浏览器URL不发送改变

\*\*\*Request域可以共享

## 使用注解修饰

### 返回带@ResponseBody注解的值

#### @ResponseBody注解和@RequestBody注解介绍

* **@ResponseBody的作用：**

\*\*\***ResponseBody**注解可以通过内置的9种**HttpMessageConverter**，匹配不同的**Controller返回值类型**，然后进行不同的**消息转换处理**。

\*\*\*将转换之后的数据放到HttpServletResponse对象的**响应体**返回到页面，

\*\*\*不同的HttpMessageConverter处理的数据，指定的**ContentType值**也不同。

* **@RequestBody**注解的作用和@ResponseBody注解正好相反，它是处理**请求参数**的Http消息转换的。

#### 常用的HttpMessageConverter

* **MappingJacksonHttpMessageConverter处理POJO类型返回值**

\*\*\*MappingJacksonHttpMessageConverter是专门处理**POJO类型**的。

\*\*\*默认使用**MappingJackson**的JSON处理能力，将后台返回的Java对象（POJO类型），转为JSON格式输出到页面

\*\*\*将响应体的**Content-Type**设置为**application/json**；charset=utf-8

* **StringHttpMessageConverter处理String类型返回值**

\*\*\*StringHttpMessageConverter是专门处理**String类型**的。

\*\*\*调用response.getWriter()方法将String类型的字符串写回给调用者。

\*\*\*将响应体的**Content-Type**设置为**text/plain**；charset=utf-8

# @RequestMapping

通过RequestMapping注解可以定义不同的处理器映射规则。

## URL路径映射

@RequestMapping(**value**="/item")或@RequestMapping("/item）

value的值是数组，可以将多个url映射到同一个方法

@RequestMapping(**value={"/item",”/queryItem”}**)

## 窄化请求映射

在**class上添加@RequestMapping(url)**指定通用**请求前缀**， 限制此类下的所有方法的访问请求url必须以**请求前缀**开头，通过此方法对url进行**模块化分类**管理。

比如：

商品模块：

/**item**/add

/**item**/update

/**item**/delete

用户模块：

/**user**/add

/**user**/update

/**user**/delete

## 请求方法限定

* **限定GET方法**

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

如果通过Post访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

@RequestMapping(value="/editItem",method=RequestMethod.GET)

* **限定POST方法**

@RequestMapping(method = RequestMethod.*POST*)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'GET' not supported

* **GET和POST都可以**

@RequestMapping(method={RequestMethod.GET,RequestMethod.POST})

# 请求参数绑定（重点）

## 什么是请求参数绑定

* **请求参数格式**

默认是**key/value**格式，比如： http://XXXXX?id=1&type=301

* **请求参数值的数据类型**

都是**字符串类型**的各种值

* **请求参数值要绑定的目标类型**

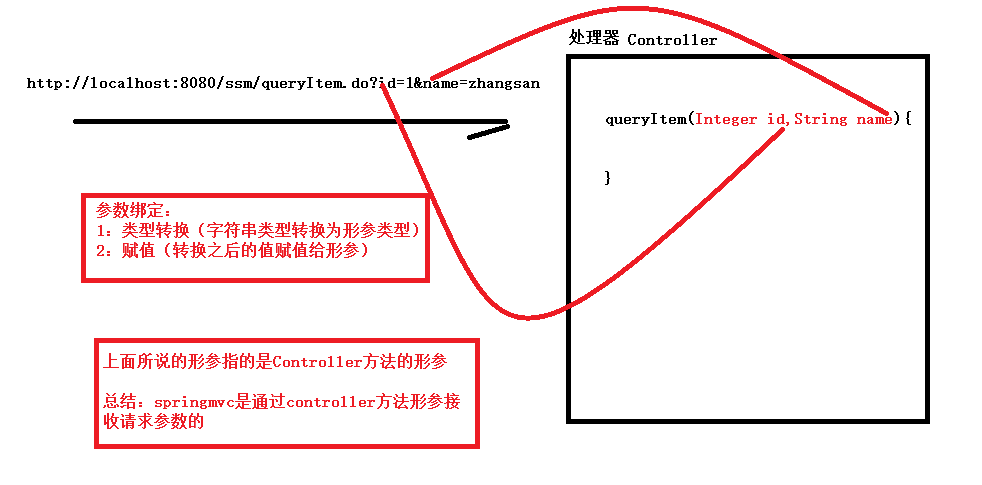
**Controller类中的方法参数**，比如简单类型、POJO类型、集合类型等。

* SpringMVC内置的**参数解析组件**

**默认内置了24种参数解析组件（ArgumentResolver）**

* **什么是参数绑定？**

就是将请求参数串中的**value值获取到之后，再**进行**类型转换**，然后将转换后的值赋值给Controller类中**方法的形参**，这个过程就是***参数绑定***。



## 默认支持的参数类型（Servlet API支持）

**Controller方法形参中可以随时添加如下类型的参数，处理适配器会自动识别并进行赋值。**

* HttpServletRequest

通过request对象获取请求信息

* HttpServletResponse

通过response处理响应信息

* HttpSession

通过session对象得到session中存放的对象

* InputStream、OutputStream
* Reader、Writer
* Model/ModelMap

ModelMap继承自LinkedHashMap，Model是一个接口，它们的底层实现都是同一个类（**BindingAwareModelMap**），作用就是向页面传递数据，相当于Request的作用，如下：

//调用service查询商品信息

Item item = service.queryItemById(id);

model.addAttribute("item", item);

## 绑定简单数据类型

### 简单类型参数绑定方式

* **简单类型**指的就是8种基本类型数据以及它们的包装类，还有String类型。
* 在springmvc中，对于java简单类型的参数，推荐的参数绑定方式有两种：

1. **直接绑定**
2. **注解绑定**

### 直接绑定

#### 使用要求

http请求参数的**key**和controller方法的**形参名称**一致

#### 请求URL

[http://localhost:8080/xxx/findItem?**id=1**](http://localhost:8080/xxx/findItem?id=1)

请求参数的key为**id**

#### Controller方法

Controller的形参为Integer **id**,它和请求参数的key一致，所以直接绑定成功

|  |
| --- |
| @RequestMapping(value = "/findItem")  **public** String findItem(**Integer id**) {  System.out.println("接收到的请求参数是："+ **id**);  **return** "success";  } |

### 通过注解绑定

#### 使用要求

请求参数的key和controller方法的形参名称不一致时，需要使用**@RequestParam**注解才能将请求参数绑定成功。

#### 请求URL

<http://localhost:8080/xxx/findItem?itemid=1>

请求参数的key为**itemid**

#### Controller方法

Controller的形参为Integer **id**,它和请求的参数不一致，要使用**@RequestParam**注解才能绑定成功

|  |
| --- |
| @RequestMapping(value = "/findItem")  // 通过@RequestParam注解绑定简单类型  **public** String findItem(**@RequestParam("itemid")** Integer id) {  System.out.println("接收到的请求参数是："+ **id**);  **return** "success";  } |

#### RequestParam注解介绍

* **value：**参数名字，即入参的请求参数名字，如value=“itemid”表示请求的参数中的名字为itemid的参数的值将传入；
* **required：**是否必须，默认是true，表示请求中一定要有相应的参数，否则将报；

TTP Status 400 - Required Integer parameter 'XXXX' is not present

* **defaultValue：**默认值，表示如果请求中没有同名参数时的默认值

|  |
| --- |
| @RequestMapping(value = "/findItem")  // 通过@RequestParam注解绑定简单类型  // 学习@RequestParam注解中的value、required、defaultValue属性  **public** String findItem (  @RequestParam(**value = "itemid", required = true, defaultValue = "2"**) Integer id) {  System.out.println("接收到的请求参数是："+ **id**);  **return** "success";  } |

## 绑定POJO类型

### 使用要求

\* 控制器方法的参数类型是 POJO 类型。

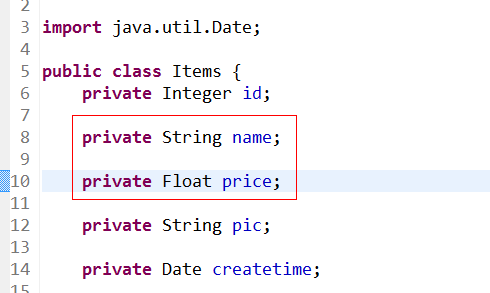
\* 要求表单中**参数名称**和 POJO 类的**属性名称**保持一致。

### 请求URL

[http://localhost:8080/xxx/updateItem?**id=1&name=iphone&price=1000**](http://localhost:8080/xxx/updateItem?id=1&name=iphone&price=1000)

### Controller方法

* **POJO定义：**



* **Controller方法**

|  |
| --- |
| @RequestMapping("/updateItem")  **public** String updateItem(Integer id,**Item item**) {    System.out.println("接收到的请求参数是："+ id);  System.out.println("接收到的请求参数是："+ item);  **return** "success";  } |

## 绑定包装POJO

**包装POJO类**，依然是一个POJO，只是说为了方便沟通，将POJO中包含另一个POJO的这种类，称之为包装POJO。

### 包装对象

|  |
| --- |
| **public** **class** ItemQueryVO {  //商品信息  **private** Item item;  } |

### 页面定义（item-list.jsp）

|  |
| --- |
| 查询条件：  <table width=*"100%"* border=*1*>  <tr>  <td>商品名称：<input type=*"text"* name=*"item.name"* /></td>  <td><input type=*"submit"* value=*"查询"* /></td>  </tr>  </table> |

### Controller方法



**测试方法：断点跟踪，查看vo中的item对象是否有值。**

## 使用简单类型数组接收参数

### 使用要求

**通过HTTP请求批量传递简单类型数据的情况，Controller方法中可以用String[]或者pojo的String[]属性接收（两种方式任选其一），但是不能使用集合接收。**

### 请求URL

[http://localhost:8080/xxx/deleteItem?**id=1&id=2&id=3**](http://localhost:8080/xxx/deleteItem?id=1&id=2&id=3)

### Controller方法

|  |
| --- |
| @RequestMapping("/deleteItem")  **public** String deleteitem(Integer[] itemId){    **return** "success";  } |

## 使用POJO类型集合或者数组接收参数

### 使用要求

批量传递的请求参数，最终要使用List<POJO>来接收，那么这个List<POJO>必须放在另一个POJO类中。

### 接收商品列表的pojo

|  |
| --- |
| **public** **class** ItemQueryVO {  // 商品信息  **private** Item item;  // 其他信息  // 商品信息集合  **private** List<Item> itemList;  } |

### 请求URL

<http://localhost:8080/xxx/batchUpdateItem>?

[**itemList[0].id=1& itemList[0].name=iphone& itemList[0].price=1000**](http://localhost:8080/xxx/batchUpdateItem?%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20itemList%5b0%5d.id=1&%20itemList%5b0%5d.name=iphone&%20itemList%5b0%5d.price=1000)**&itemList[1].id=2& itemList[1].name=iphone x& itemList[1].price=2000**

### Contrller

|  |
| --- |
| @RequestMapping("/batchUpdateItem")  **public** String batchUpdateItem**(ItemQueryVO vo)** {  **return** "success";  } |

## 自定义参数绑定

### 请求URL

[http://localhost:8080/xxx/saveItem?**date=2018-08-12**](http://localhost:8080/xxx/saveItem?date=2018-08-12)

### Controller方法

|  |
| --- |
| @RequestMapping("/saveItem")  **public** String saveItem(String date){  System.out.println("接收到的请求参数是："+ date);  **return** "success";  } |

但是如果将date参数的类型由String改为Date，则报错

### 自定义Converter

|  |
| --- |
| **public** **class** DateConverter **implements** Converter<String, Date> {  @Override  **public** Date convert(String source) {  SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat("**yyyy-MM-dd**");  **try** {  **return** simpleDateFormat.parse(source);  } **catch** (ParseException e) {  e.printStackTrace();  }  **return** **null**;  }  } |

### 配置Converter

在springmvc.xml中，进行以下配置：

|  |
| --- |
| <!-- 加载注解驱动 -->  <mvc:annotation-driven conversion-service=*"conversionService"*/>  <!-- 转换器配置 -->  <bean id=*"conversionService"*  class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>  <property name=*"converters"*>  <set>  <bean class=*"com.kkb.ssm.controller.converter.DateConverter"*/>  </set>  </property>  </bean> |

# @ControllerAdvice注解

## @ControllerAdvice

### 介绍

\*\*\*该注解顾名思义是一个增强器，是对注解了@Controller注解的类进行增强。

\*\*\*该注解使用@Component注解，这样的话当我们使用<context:component-scan>扫描时也能扫描到。

\*\*\*该注解内部使用@ExceptionHandler、@InitBinder、@ModelAttribute注解的方法会应用到所有的Controller类中 @RequestMapping注解的方法。

### 使用

|  |
| --- |
| **@ControllerAdvice**  **public** **class** MyControllerAdvice {  //应用到所有@RequestMapping注解方法，在其执行之前把返回值放入ModelMap中  **@ModelAttribute**  **public** Map<String, Object> ma(){  Map<String, Object> map = **new** HashMap<>();  map.put("name", "tom");  **return** map;  }    //应用到所有【带参数】的@RequestMapping注解方法，在其执行之前初始化数据绑定器  **@InitBinder**  **public** **void** initBinder(WebDataBinder dataBinder) {  }    //应用到所有@RequestMapping注解的方法，在其抛出指定异常时执行  **@ExceptionHandler**(Exception.**class**)  @ResponseBody  **public** String handleException(Exception e) {  **return** e.getMessage();  }  } |

## @ModelAttribute

### 介绍

**\* 该注解特点：**主要作用于ModelMap这个模型对象的，用于在视图中显示数据。

**\* 该注解注意事项：**和@ResponseBody注解的使用是互斥的。

**\* 该注解有两个用途：**

* 作用于方法（**有没有@RequestMapping注解的方法都可以**）

\* **非@RequestMapping注解方法**

\* **执行时机**：在本类内所有@RequestMapping注解方法之前执行。

\* **作用**：

\* 如果方法有返回值，则直接将该返回值放入ModelMap中，key可以指定。

\* 如果方法没有返回值，则可以利用它的执行时机这一特点，做一些预处理。

\* **有@RequestMapping注解方法**

\* **作用：**

\* **返回值会放入ModelMap中**

\* 逻辑视图名称根据请求URL生成，比如URL：item/itemList，那么这个URL就是逻辑视图名称

* 作用于方法参数（**有@RequestMapping注解的方法**）

\*\*\* STEP1：获取ModelMap中指定的数据（由@ModelAttribute注解的方法产生）

\*\*\* STEP2：将使用该注解的参数对象放到ModelMap中。

\*\*\* 如果STEP1和STEP2中的对象在ModelMap中key相同，则合并这两部分对象的数据。

### 使用

课堂补充

## @InitBinder

### 介绍

\*\*\* 由 @InitBinder 标识的方法，可以通过**PropertyEditor解决数据类型转换问题，比如String-->Date类型**。

\*\*\* 不过PropertyEditor只能完成**String**-->**任意类型**的转换，这一点不如Converter灵活，Converter可以完成**任意类型**-->**任意类型**的转换。

\*\*\* 它可以对 **WebDataBinder** 对象进行初始化，WebDataBinder 是 DataBinder 的子类,用于完成由表单字段到 JavaBean 属性的绑定。

\*\*\* 注意事项：

\* @InitBinder方法**不能有返回值**,它必须声明为void。

\* @InitBinder方法的**参数通常是 WebDataBinder**。

### 使用

课堂补充

## @ExceptionHandler

### 介绍

\*\*\* 这个注解表示Controller中任何一个@RequestMapping方法发生异常，则会被注解了@ExceptionHandler的方法拦截到。

\*\*\* 拦截到对应的异常类则执行对应的方法，如果都没有匹配到异常类，则采用近亲匹配的方式。

### 使用

课堂补充

# 异常处理器

springmvc在处理请求过程中出现异常信息交由异常处理器进行处理，自定义异常处理器可以实现一个系统的异常处理逻辑。

## 异常理解

异常包含**编译时异常**和**运行时异常**，其中编译时异常也叫**预期异常**。

运行时异常只有在项目运行的情况下才会发现，编译的时候不需要关心。

运行时异常，比如：**空指针异常、数组越界异常**，对于这样的异常，只能通过程序员丰富的经验来解决和测试人员不断的严格测试来解决。

**编译时异常**，比如：数据库异常、文件读取异常、自定义异常等。对于这样的异常，必须使用try catch代码块或者throws关键字来处理异常。

## 异常处理思路

系统中异常包括两类：预期异常（编译时异常）和运行时异常RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。

系统的dao、service、controller出现都通过throws Exception向上抛出，最后由springmvc前端控制器交由异常处理器进行异常处理，如下图：

Controller

客户端

Service

Dao

Springmvc

**DispatcherServlet**

请求

异常

**HandlerExceptionResolver**

异常处理器

异常

异常

**全局范围只有一个异常处理器。**

## 自定义异常类

为了区别不同的异常通常根据异常类型自定义异常类，这里我们创建一个自定义系统异常，如果controller、service、dao抛出此类异常说明是系统预期处理的异常信息。

**public** **class** BusinessException **extends** Exception {

**private** **static** **final** **long** *serialVersionUID* = 1L;

//异常信息

**private** String message;

**public** BusinessException(String message) {

**super**(message);

**this**.message = message;

}

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

## 自定义异常处理器

**public** **class** BusinessExceptionResolver **implements** **HandlerExceptionResolver** {

@Override

**public** ModelAndView resolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex) {

//自定义预期异常

BusinessException businessException = **null**;

//如果抛出的是系统自定义的异常

**if**(ex **instanceof** BusinessException){

businessException = (BusinessException) ex;

}**else**{

businessException = **new** BusinessException("未知错误");

}

ModelAndView modelAndView = **new** ModelAndView();

//把错误信息传递到页面

modelAndView.addObject("message", businessException.getMessage());

//指向错误页面

modelAndView.setViewName("error");

**return** modelAndView;

}

}

## 错误页面

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>错误信息</title>

</head>

<body>

${message }

</body>

</html>

## 异常处理器配置

在springmvc.xml中加入以下代码：

<!-- 自定义异常处理器（全局） -->

<bean class=*"com.kkb.ssm.resolver.BusinessExceptionResolver"*/>

## 异常测试

@RequestMapping(value = "/showItemEdit")

**public** String showItemEdit(Integer id,Model model) **throws** Exception{

// 查询要显示的商品内容

Item item = itemService.queryItemById(id);

**if**(item == **null**) **throw** **new** BusinessException("查询不到商品无法修改");

model.addAttribute("item", item);

// 由于配置了ViewResolver，所以此处只写逻辑视图名称即可

**return** "item/item-edit";

}

# 上传图片

SpringMVC文件上传的实现，是由commons-fileupload这个jar包实现的。

## 需求

在修改商品页面，添加上传商品图片功能。

## 加入依赖包

|  |
| --- |
| <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>1.3.1</version>  </dependency> |

## 上传页面

**文件上传需要指定enctype=”multipart/form-data”**

## 配置Multipart解析器

在springmvc.xml中配置multipart类型解析器：

<!-- multipart类型解析器，文件上传 -->

<bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- 上传文件的最大尺寸 5M-->

<property name=*"maxUploadSize"* value=*"5242880"*/>

</bean>

## 上传图片代码

### Controller类

@RequestMapping(value = "/updateItem")

**public** String updateItem(Model model,Item item,

**MultipartFile pictureFile**) **throws** Exception {

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

System.*out*.println(pictureFile.getOriginalFilename());

//原始图片名称

String originalFilename = pictureFile.getOriginalFilename();

//如果没有图片名称，则上传不成功

**if**(originalFilename != **null** && originalFilename.length()>0)

{

//存放图片的物理路径

String picPath = "E:\\03-teach\\07-upload\\temp\\";

//新文件的名称

String newFileName = UUID.*randomUUID*()+originalFilename.substring(originalFilename.lastIndexOf("."));

//新的文件

File newFile = **new** File(picPath+newFileName);

//把上传的文件保存成一个新的文件

pictureFile.transferTo(newFile);

//同时需要把新的文件名更新到数据库中

item.setPic(newFileName);

}**else**{

**throw** **new** BusinessException("图片名称不存在，上传不成功");

}

}

// 根据页面传入的商品信息，调用修改方法，进行修改（此时还没有讲参数绑定，暂时无法进行）

itemService.updateItem(item);

**return** "success";

}

### Jsp页面

item-edit.jsp：

<tr>

<td>商品图片</td>

<td>

<c:if test=*"*${item.pic !=null}*"*>

<img src=*"/pic/*${item.pic}*"* width=*100* height=*100*/>

<br/>

</c:if>

<input type=*"file"* name=*"****pictureFile****"*/>

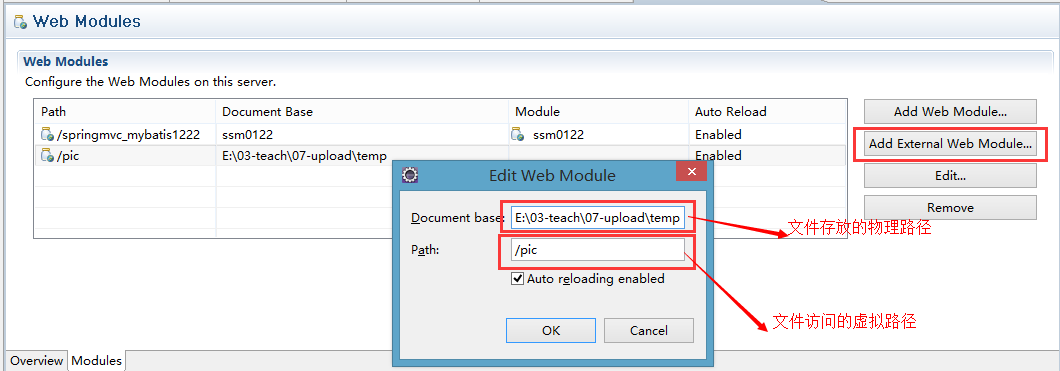
</td>

</tr>

## 配置图片文件的虚拟目录

**注意：在图片虚拟目录 中，一定将图片目录分级创建（提高i/o性能），一般我们采用按日期(年、月、日)进行分级创建。**

### 通过eclipse配置tomcat的虚拟目录



### 直接修改tomcat的配置文件

在tomcat安装目录中的conf/server.xml文件中，添加虚拟目录：

<Context docBase="E:\03-teach\07-upload\temp" path="/pic" reloadable="false"/>

# JSON数据交互

## 为什么使用JSON进行数据交互

JSON数据格式比较简单、解析比较方便，在接口调用及HTML页面Ajax调用时较常用。

## JSON交互方式

* **请求是KV，响应是JSON（推荐使用）**
* **请求是JSON，响应是JSON**

## 加入依赖包

|  |
| --- |
| <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.9.6</version>  </dependency> |

## JSON交互测试

### 输入key/value、输出JSON（推荐）

#### JSP页面

**function** responseKV(){

$.ajax({

type:"post",

url:'${pageContext.request.contextPath }/responseKV',

//输入是key/value时，默认就指定好了contentType了，不需要再指定了

//contentType:'application/json;charset=utf-8',

//data为key/value形式

data:'name=json测试&price=999',

success:**function**(data){

alert(data);

}

});

}

#### Controller类

// 输入是key/value，输出是json

// @ResponseBody 将返回值转成json串响应给前台

@RequestMapping("/responseKV")

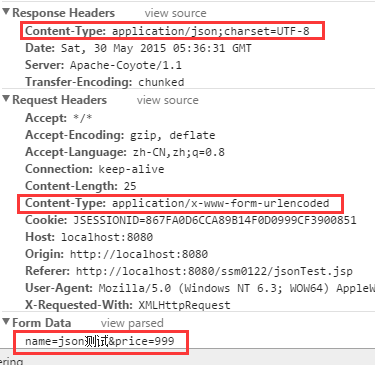
@ResponseBody

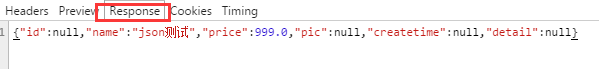
**public** Item responseKV(Item item) {

**return** item;

}

#### 页面控制台输出





### 输入JSON、输出JSON

#### JSP页面

在webroot下创建一个jsonTest.jsp，然后添加以下JavaScript脚本：

**function** requestJson(){

$.ajax({

type:"post",

url:'${pageContext.request.contextPath }/requestJson',

//输入是json是 ，需要指定contentType为application/json

**contentType:'application/json;charset=utf-8',**

data:'{"name":"json测试","price":999}',

success:**function**(data){

alert(data.name);

}

});

}

#### Controller类

@Controller

**public** **class** JsonController {

// 输入是json，输出是json

// @RequestBody 将请求的json串转成java对象

// @ResponseBody 将返回值转成json串响应给前台

@RequestMapping("/requestJson")

**@ResponseBody**

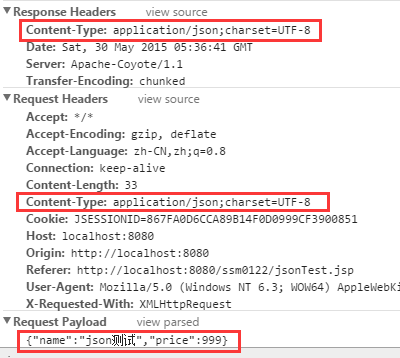
**public** Item requestJson(**@RequestBody** Item item) {

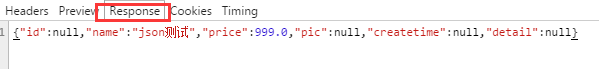
**return** item;

}

}

#### 页面控制台输出





# Mock测试（模拟测试）

## 什么是mock测试:

\*\*\* 在测试过程中，对于某些不容易构造或者不容易获取的对象，用一个**虚拟的对象**来创建以便测试的测试方法，就是**mock测试**。

Servlet、Request、Response等Servlet API相关对象本来是由Servlet容器（tomcat）创建的。

\*\*\* 这个**虚拟的对象**就是**mock对象**。

\*\*\* **mock对象**就是真实对象在调试期间的**代替品**。

## 为什么使用mock测试

* 避免开发模块之间的耦合
* 轻量、简单、灵活

## MockMVC介绍

基于RESTful风格的SpringMVC的测试，我们可以测试完整的Spring MVC流程，即**从URL请求到控制器处理，再到视图渲染都可以测试**。

* MockMVCBuilder

\* MockMvcBuilder是用来构造MockMvc的构造器

\*其主要有两个实现：StandaloneMockMvcBuilder和DefaultMockMvcBuilder，分别对应之前的两种测试方式。

\* 对于我们来说直接使用**静态工厂MockMvcBuilders创建**即可

* MockMVCBuilders

\* 负责创建MockMVCBuilder对象

\* 有两种创建方式

\* standaloneSetup(Object... controllers):

\* 通过参数指定一组控制器，这样就不需要从上下文获取了。

\* **webAppContextSetup**(WebApplicationContext wac)

\* 指定WebApplicationContext，将会从该上下文获取相应的控制 器并得到相应的MockMvc

* **MockMvc**

\* 对于服务器端的Spring MVC测试支持**主入口点**。

\* 通过MockMVCBuilder构造

\* MockMVCBuilder由MockMVCBuilders建造者的静态方法去建造。

\* 核心方法：**perform(RequestBuilder rb)---** **执行一个RequestBuilder请求，会自动执行SpringMVC的流程并映射到相应的控制器执行处理，该方法的返回值是一个ResultActions；**

* ResultActions

\* **andExpect**：添加ResultMatcher验证规则，验证控制器执行完成后结果是否正确；

\* **andDo**：添加ResultHandler结果处理器，比如调试时打印结果到控制台；

\* **andReturn**：最后返回相应的**MvcResult**；然后进行自定义验证/进行下一步的异步处理；

* MockMvcRequestBuilders

\* 用来构建请求的

\*其主要有两个子类**MockHttpServletRequestBuilder**和**MockMultipartHttpServletRequestBuilder**（如文件上传使用），即用来Mock客户端请求需要的所有数据。

* MockMvcResultMatchers

\* 用来匹配执行完请求后的**结果验证**

\* 如果匹配失败将抛出相应的异常

\* 包含了很多验证API方法

* MockMvcResultHandlers

\* 结果处理器，表示要对结果做点什么事情

\* 比如此处使用MockMvcResultHandlers.print()输出整个响应结果信息。

* MvcResult

\* 单元测试执行结果，可以针对执行结果进行**自定义验证逻辑**。

## MockMVC使用

### 添加依赖

|  |
| --- |
| <!-- spring 单元测试组件包 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>5.0.7.RELEASE</version>  </dependency>  <!-- 单元测试Junit -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  </dependency>  <!-- Mock测试使用的json-path依赖 -->  <dependency>  <groupId>com.jayway.jsonpath</groupId>  <artifactId>json-path</artifactId>  <version>2.2.0</version>  </dependency> |

### 测试类

|  |
| --- |
| **import** **static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*get*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultHandlers.*print*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*status*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*jsonPath*;  **import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.*content*;  **import** org.junit.Before;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.http.MediaType;  **import** org.springframework.test.context.ContextConfiguration;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  **import** org.springframework.test.context.web.WebAppConfiguration;  **import** org.springframework.test.web.servlet.MockMvc;  **import** org.springframework.test.web.servlet.MvcResult;  **import** org.springframework.test.web.servlet.request.MockMvcRequestBuilders;  **import** org.springframework.test.web.servlet.result.MockMvcResultHandlers;  **import** org.springframework.test.web.servlet.result.MockMvcResultMatchers;  **import** org.springframework.test.web.servlet.setup.MockMvcBuilders;  **import** org.springframework.web.context.WebApplicationContext;  //@WebAppConfiguration：可以在单元测试的时候，不用启动Servlet容器，就可以获取一个Web应用上下文  @RunWith(SpringJUnit4ClassRunner.**class**)  @ContextConfiguration(locations = "classpath:spring/\*.xml")  @WebAppConfiguration  **public** **class** TestMockMVC {  @Autowired  **private** WebApplicationContext wac;  **private** MockMvc mockMvc;  @Before  **public** **void** setup() {  // 初始化一个MockMVC对象的方式有两种：单独设置、web应用上下文设置  // 建议使用Web应用上下文设置  mockMvc = MockMvcBuilders.*webAppContextSetup*(wac).build();  }  @Test  **public** **void** test() **throws** Exception {  // 通过perform去发送一个HTTP请求  // andExpect：通过该方法，判断请求执行是否成功  // andDo :对请求之后的结果进行输出  MvcResult result = mockMvc.perform(**MockMvcRequestBuilders**.*get*("/item/showEdit").param("id", "1"))  .andExpect(**MockMvcResultMatchers**.*view*().name("item/item-edit"))  .andExpect(**MockMvcResultMatchers**.*status*().isOk())  .andDo(**MockMvcResultHandlers**.*print*())  .andReturn();    System.***out***.println("================================");  System.***out***.println(result.getHandler());  }    @Test  **public** **void** test2() **throws** Exception {  // 通过perform去发送一个HTTP请求  // andExpect：通过该方法，判断请求执行是否成功  // andDo :对请求之后的结果进行输出  MvcResult result = mockMvc.perform(*get*("/item/findItem").param("id", "1").accept(MediaType.***APPLICATION\_JSON***))  .andExpect(*status*().isOk())  .andExpect(*content*().contentTypeCompatibleWith(MediaType.***APPLICATION\_JSON***))  .andExpect(*jsonPath*("$.id").value(1))  .andExpect(*jsonPath*("$.name").value("台式机123"))  .andDo(*print*())  .andReturn();    System.***out***.println("================================");  System.***out***.println(result.getHandler());  }  } |

* **@WebAppConfiguration**

用于声明一个ApplicationContext集成测试加载WebApplicationContext。

# RESTful支持

理解什么是REST之前，先去理解一下什么是HTTP，参考【HTTP协议章节】

## 什么是RESTful？

* **什么是REST？**

**\*\*\* REST**（英文：Representational State Transfer，简称 **REST**，意思是：（资源）**表述性状态转化**）描述了一个**架构**样式的网络系统， 比如 web 应用程序。

\*\*\* 它是一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于**客户端和服务器**交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

\*\*\* 它本身并没有什么实用性，其核心价值在于如何设计出符合 REST 风格的网络接口。

* **什么是RESTful？**

\*\*\* **REST** 指的是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 **RESTful**。

* **RESTful 的特性：**

**\* 资源（Resources）**：网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。可以用一个 URI（统一资源定位符）指向它，每种资源对应一个特定的 **URI** 。要获取这个资源，访问它的 URI 就可以，因此 URI 即为每一个资源的独一无二的识别符。

**\* 表现层（Representation）：**把资源具体呈现出来的形式，叫做它的表现层 （Representation）。比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式。

**\* 状态转化（State Transfer）：**每发出一个请求，就代表了客户端和服务器的一次交互过程。HTTP协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器， 必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。而这种转化是建立在表现层之上的，所以就是 “ 表现层状态转化” 。具体说， 就是 HTTP 协议里面，四个表示操作方式的动词：**GET 、POST 、PUT 、DELETE** 。它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源。

* **如何设计RESTful应用程序的API**

\* **路径设计**：数据库设计完毕之后，基本上就可以确定有哪些资源要进行操作，相对于的路径也可以设计出来

**\* 动词设计：**也就是针对资源的具体操作类型，由HTTP动词表示，常用的HTTP动词如下：POST、DELETE、PUT、GET

* **RESTful 的示例：**

/account/1 HTTP GET ： 得到 id = 1 的 account

/account/1 HTTP DELETE： 删除 id = 1 的 account

/account/1 HTTP PUT： 更新 id = 1 的 account

## SpringMVC对RESTful的支持

### RESTful的URL路径变量

* **URL-PATTERN** ：设置为**/**，方便拦截RESTful 请求。
* @**PathVariable**：可以解析出来URL中的**模板变量**（**{id}**）

|  |
| --- |
| URL: <http://localhost:8080/ssm/item/1/zhangsan>  Controller:  @RequestMapping(“**{id}/{name}**”)  @ResponseBody  public Item queryItemById(**@PathVariable** Integer id, **@PathVariable** String name) |

### RESTful的CRUD

* **@RequestMapping**：通过设置**method**属性的CRUD，可以将**同一个URL**映射到不同的HandlerMethod方法上
* **@GetMapping、@PostMapping、@PutMapping、@DeleteMapping**注解同@RequestMapping注解的method属性设置。

### RESTful的资源表述

* **RESTful**服务中一个重要的特性就是**一种资源**可以有**多种表现形式**，在SpringMVC中可以使用**ContentNegotiatingManager**这个**内容协商管理器**来实现这种方式。
* 内容协商的方式有三种：

\* **扩展名**,比如.json表示我要JSON格式数据、.xml表示我要XML格式数据

\* **请求参数**：默认是”format”

\* **请求头设置Accept参数**，比如设置Accept为application/json表示要JSON格式数据

* 不过现在RESTful响应的数据一般都是**JSON格式**，所以一般也不使用内容协商管理器，直接使用**@ResponseBody**注解将数据按照JSON格式返回

## 静态资源访问<mvc:resources>

**如果在DispatcherServlet中设置url-pattern为 /则必须对静态资源进行访问处理。**

在springmvc.xml文件中，使用mvc:resources标签，具体如下：

<!-- 当DispatcherServlet配置为/来拦截请求的时候，需要配置静态资源的访问映射 -->

<mvc:resources location=*"/js/"* mapping=*"/js/\*\*"*/>

<mvc:resources location=*"/css/"* mapping=*"/css/\*\*"*/>

Springmvc会把mapping映射到**ResourceHttpRequestHandler**，这样静态资源在经过DispatcherServlet转发时就可以找到对应的Handler了。

# 拦截器

SpringMVC的拦截器主要是针对特定处理器进行拦截的。

## SpringMVC拦截器介绍

* SpringMVC拦截器（**Interceptor**）实现对每一个请求处理前后进行相关的业务处理，类似与servlet中的**Filter**。
* SpringMVC 中的Interceptor 拦截请求是通过**HandlerInterceptor**来实现的。
* 在SpringMVC中定义一个Interceptor非常简单，主要有4种方式：

1）实现Spring的**HandlerInterceptor**接口；

2）继承实现了HandlerInterceptor接口的类，比如Spring 已经提供的实现了HandlerInterceptor 接口的抽象类HandlerInterceptorAdapter；

3）实现Spring的WebRequestInterceptor接口；

4）继承实现了WebRequestInterceptor的类；

## 定义拦截器

实现HandlerIntercepter接口：

**public** **class** MyHandlerIntercepter1 **implements** **HandlerInterceptor**{

//Handler执行前调用

//应用场景：**登录认证、身份授权**

//返回值为true则是放行，为false是不放行

@Override

**public** **boolean** **preHandle**(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

**return** **false**;

}

//进入Handler开始执行，并且在返回ModelAndView之前调用

//应用场景：对ModelAndView对象操作，可以把公共模型数据传到前台，可以统一指定视图

@Override

**public** **void** **postHandle**(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView **modelAndView**) **throws** Exception {

}

//执行完Handler之后调用

//应用场景：**统一异常处理、统一日志处理**

@Override

**public** **void** **afterCompletion**(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

}

}

## 配置拦截器

**SpringMVC拦截器是绑定在HandlerMapping中的**，即；如果某个HandlerMapping中配置拦截，则该HandlerMapping映射成功的Handler会使用该拦截器。

### 针对单个HandlerMapping配置

只有通过该处理器映射器查找到的处理器，才能使用该拦截器。

如果现在有两个处理器映射器：其中一个设置了处理器拦截器，另外一个没有设置，如果通过第二个映射器查找到的处理器，是无法使用拦截器的。

<bean class=*"org.springframework.web.servlet.handler.****BeanNameUrlHandlerMapping****"*>

<property name=*"interceptors"*>

<list>

<ref bean=*"interceptor"* />

</list>

</property>

</bean>

<bean id=*"interceptor"* class=*"com.kkb.ssm.interceptor.MyHandlerInterceptor"* />

### 全局拦截器配置（推荐）

* SpringMVC的全局拦截器配置，其实是把配置的拦截器注入到每个已初始化的**HandlerMapping**中了。

<!-- 配置全局mapping的拦截器 -->

<mvc:interceptors>

<!-- 公共拦截器可以拦截所有请求，而且可以有多个 -->

**<bean class=*"com.kkb.ssm.interceptor.MyHandlerInterceptor1"* />**

**<bean class=*"com.kkb.ssm.interceptor.MyHandlerInterceptor2"* />**

<!-- 如果有多个拦截器，则按照顺序进行配置 -->

<mvc:interceptor>

<!-- /\*\*表示所有URL和子URL路径 -->

<mvc:mapping path=*"/test/\*\*"* />

<!-- 特定请求的拦截器只能有一个 -->

<bean class=*"com.kkb.ssm.interceptor.MyHandlerInterceptor3"* />

</mvc:interceptor>

</mvc:interceptors>

## 多拦截器拦截规则

**如果有多个拦截器，那么配置到springmvc.xml中最上面的拦截器，拦截优先级最高**。

## 拦截器应用（实现登录认证）

### 需求

拦截器对访问的请求URL进行拦截校验

1. 如果请求的URL是公开地址（无需登录就可以访问的URL,具体指的就是保护login字段的请求URL），采取放行。
2. 如果用户session存在，则放行。
3. 如果用户session中不存在，则跳转到登录页面。

### 登录jsp页面

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>登录页面</title>

</head>

<body>

<form action=*"*${pageContext.request.contextPath }*/login"* method=*"post"*>

<table align=*"center"* border=*"1"* cellspacing=*"0"* >

<tr>

<td>用户名：<input type=*"text"* name=*"username"*/></td>

</tr>

<tr>

<td>密 码：<input type=*"text"* name=*"password"*/></td>

</tr>

<tr>

<td><input type=*"submit"* value=*"登录"*/></td>

</tr>

</table>

</form>

</body>

</html>

### Controller类

@Controller

**public** **class** LoginController {

//显示登录页面

@RequestMapping("/loginPage")

**public** String loginPage(){

**return** "login";

}

// 登录

@RequestMapping("/login")

**public** String login(HttpSession session, String username, String password) {

// Service进行用户身份验证

// 把用户信息保存到session中

session.setAttribute("username", username);

// 重定向到商品列表页面

**return** "redirect:/item/queryItem";

}

// 退出

@RequestMapping("/logout")

**public** String logout(HttpSession session) {

//清空session

session.invalidate();

// 重定向到登录页面

**return** "redirect:/loginPage";

}

}

### HandlerInterceptor类

**public** **class** LoginInterceptor **implements** HandlerInterceptor {

@Override

**public** **boolean** preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

//获取请求的URI

String requestURI = request.getRequestURI();

System.*out*.println(requestURI);

// 1、 如果请求的URL是公开地址（无需登录就可以访问的URL），采取放行。

**if**(requestURI.indexOf("login")>-1) **return** **true**;

// 2、 如果用户session存在，则放行。

String username = (String) request.getSession().getAttribute("username");

**if**(username !=**null** && !username.equals("")) **return** **true**;

// 3、 如果用户session中不存在，则跳转到登录页面。

response.sendRedirect("/ssm/loginPage");

**return** **false**;

}

@Override

**public** **void** postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

// **TODO** Auto-generated method stub

}

}

### HandlerInterceptor配置

<!-- 配置全局mapping的拦截器 -->

<mvc:interceptors>

<!-- 如果有多个拦截器，则按照顺序进行配置 -->

<mvc:interceptor>

<!-- /\*\*表示所有URL和子URL路径 -->

<mvc:mapping path=*"/\*\*"* />

<bean class=*"com.kkb.ssm.interceptor.LoginInterceptor"* />

</mvc:interceptor>

<!-- 如果有多个拦截器，则按照顺序进行配置 -->

<mvc:interceptor>

<!-- /\*\*表示所有URL和子URL路径 -->

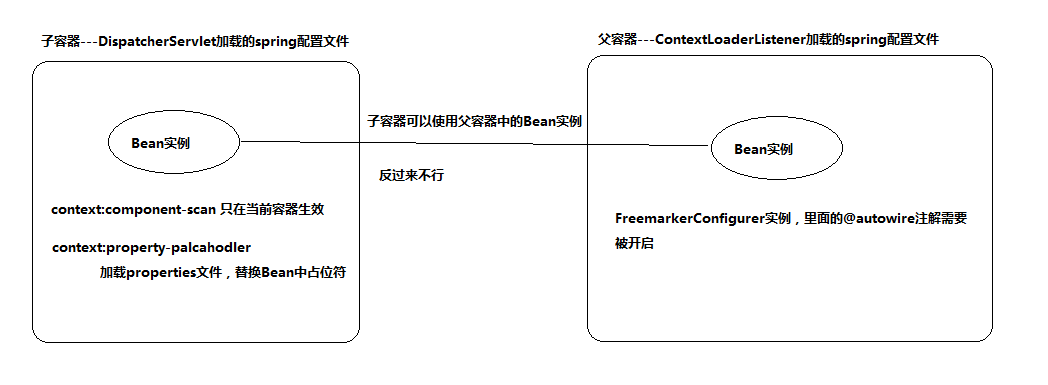
<mvc:mapping path=*"/\*\*"* />

<bean class=*" com.kkb.ssm.interceptor.MyHandlerInterceptor"* />

</mvc:interceptor>

</mvc:interceptors>

# SpringMVC父子容器



问题1：在子容器声明一个Bean，然后父容器中使用@Autowired注解注入，验证注入是否成功。

问题2：@Autowired注解的开启方式有哪些？如果父容器中不开启@Autowired注解，是如何操作？

# 跨域处理

## 什么是跨域？

\*\*\* 由于浏览器对于**Javascript**的**同源策略**的限制，导致A网站不能通过JS（主要就是**Ajax请求**）去访问B网站的数据，于是跨域问题就出现了。

\*\*\* 跨域指的是**域名、端口、协议**的组合不同就是跨域。

<http://www.kkb.com/>

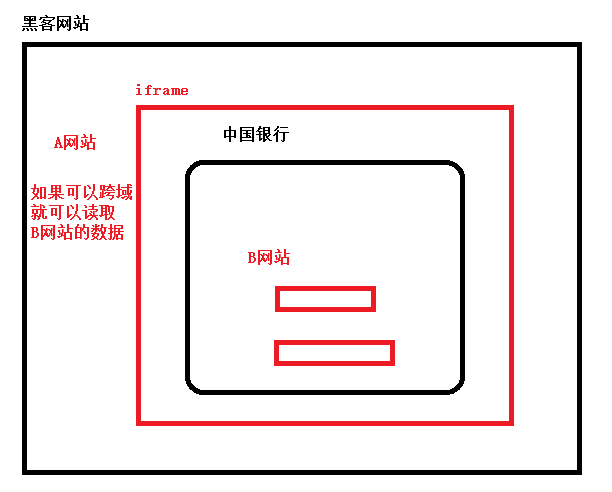
<https://www.kkb.com>

<http://www.kkb.cn>

<http://www.kkb.com:8080/>

## 为什么要有同源策略？

我们举例说明：比如一个黑客程序，他利用IFrame把真正的银行登录页面嵌到他的页面上，当你使用真实的用户名，密码登录时，他的页面就可以通过Javascript读取到你的表单中input中的内容，这样用户名，密码就轻松到手了。



## 如何解决跨域？

\*\*\* 解决跨域的方式有多种，比如**基于JavaScript的解决方式**、**基于Jquery的JSONP方式**、以及**基于CORS的方式**。

\*\*\*JSONP和CORS的区别之一：**JSONP只能解决get方式提交**、CORS不仅支持GET方式，同时也支持POST提交方式。

\*\*\* 我们重点就来讲解**CORS跨域**方式。

## 什么是CORS？

\*\*\* CORS是一个**W3C**标准，全称是"**跨域资源共享**"（Cross-origin resource sharing）。

\*\*\* 它允许浏览器向跨源服务器，发出**XMLHttpRequest**请求，从而克服了AJAX只能同源使用的限制。

\*\*\* CORS需要**浏览器和服务器同时支持**。目前，所有浏览器都支持该功能，**IE浏览器不能低于IE10。**

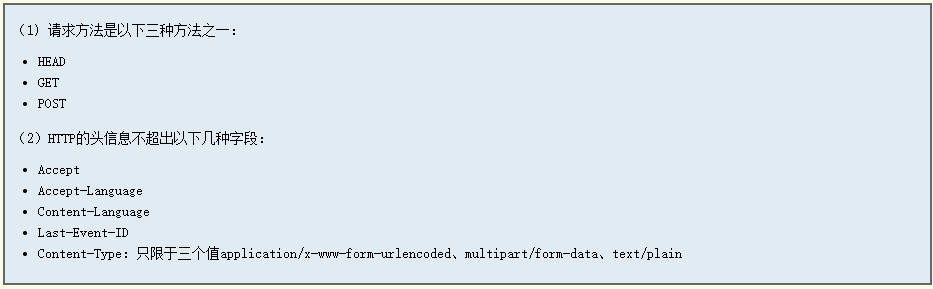
\*\*\* CORS原理：只需要向**响应头**header中注入**Access-Control-Allow-Origin**，这样浏览器检测到header中的Access-Control-Allow-Origin，则就可以跨域操作了。

## CORS请求分类

### CORS请求分类标准

浏览器将CORS请求分成两类：**简单请求**（simple request）和**非简单请求**（not-so-simple request）。

只要同时满足以下两大条件，就属于**简单请求**。



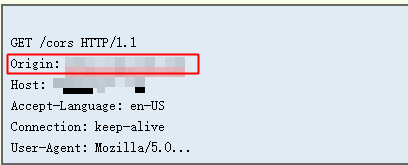
凡是不同时满足上面两个条件，就属于**非简单请求**。

浏览器对这两种请求的处理，是不一样的。

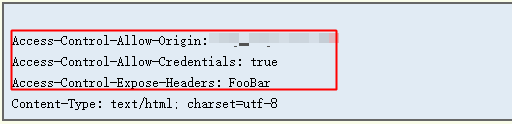
### 简单请求

对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个**Origin**字段。

* 请求信息：



* 响应信息：



* 字段说明

**（1）Access-Control-Allow-Origin**

该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个\*，表示接受任意域名的请求。

**（2）Access-Control-Allow-Credentials**

该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。

### 非简单请求

\*\*\* 非简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。

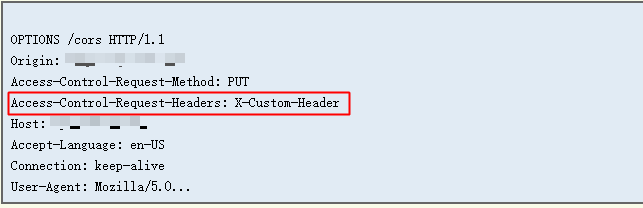
\*\*\* 非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称**为"预检"请求（preflight）**。

\*\*\* 浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

* 请求信息：

HTTP请求的方法是PUT，并且发送一个自定义头信息X-Custom-Header。

浏览器发现，这是一个非简单请求，就自动发出一个"预检"请求，要求服务器确认可以这样请求。下面是这个"预检"请求的HTTP头信息。



"预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。

除了Origin字段，"预检"请求的头信息包括两个特殊字段。

**（1）Access-Control-Request-Method**

该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法，上例是PUT。

**（2）Access-Control-Request-Headers**

该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段，上例是X-Custom-Header。

**一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样**，会有一个Origin头信息字段。服务器的回应，也都会有一个Access-Control-Allow-Origin头信息字段。

## CORS实现

使用**springmvc的拦截器**实现

### 跨域不提交Cookie

|  |
| --- |
| **public** **class** AllowOriginInterceptor **implements** HandlerInterceptor {        @Override  **public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object arg2) **throws** Exception {         // 有跨域行为时参考网址 http://namezhou.iteye.com/blog/2384434  **if** (request.getHeader("Origin") != **null**) {             response.setContentType("text/html;charset=UTF-8");             // 允许哪一个URL  **response.setHeader("Access-Control-Allow-Origin", "\*");**             // 允许那种请求方法            response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE");  **response.setHeader("XDomainRequestAllowed", "1");**             System.***out***.println("正在跨域");         }  **return** **true**;      }   } |

配置拦截器的代码，请自行补充！！

### 跨域提交Cookie

#### 注意事项

**\*\*\* Access-Control-Allow-Credentials** 为 **true**的时候，**Access-Control-Allow-Origin一定不能设置为”\*”，否则报错**

**\*\*\* 如果有多个拦截器，一定要把处理跨域请求的拦截器放到首位。**

#### JS代码

* Jquery Ajax

|  |
| --- |
| $.ajax({  url: '自己要请求的url',  method:'请求方式', //GET POST PUT DELETE  **xhrFields:{withCredentials:true},**  success:function(data){  //自定义请求成功做什么  },  error:function(){  //自定义请求失败做什么  }  }) |

* angularJS

**1. 全局 在模块配置中添加**

app.config(['$httpProvider',function($httpProvider) {

  **$httpProvider.defaults.withCredentials = true;**

}

]);

2. 单个请求

$http.get(url, {withCredentials: true});

$http.post(url,data, {withCredentials: true});

**$httpProvider.defaults.withCredentials = true;**

#### Java代码

|  |
| --- |
| **public** **class** AllowOriginInterceptor **implements** HandlerInterceptor {        @Override  **public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object arg2) **throws** Exception {         // 有跨域行为时参考网址 http://namezhou.iteye.com/blog/2384434  **if** (request.getHeader("Origin") != **null**) {             response.setContentType("text/html;charset=UTF-8");             // 允许哪一个URL 访问 request.getHeader("Origin") 根据请求来的url动态允许  **response.setHeader("Access-Control-Allow-Origin", request.getHeader("Origin"));**             // 允许那种请求方法            response.setHeader("Access-Control-Allow-Methods", "POST, GET, OPTIONS, DELETE,HEAD");             response.setHeader("Access-Control-Max-Age", "0");             // 允许请求头里的参数列表             response.setHeader("Access-Control-Allow-Headers",                    "Origin, No-Cache, X-Requested-With, If-Modified-Since, Pragma, Last-Modified, Cache-Control, Expires, Content-Type, X-E4M-With,userId,token");             // 允许对方带cookie访问  **response.setHeader("Access-Control-Allow-Credentials", "true");**  **response.setHeader("XDomainRequestAllowed", "1");**             System.***out***.println("正在跨域");         }  **return** **true**;      }  } |

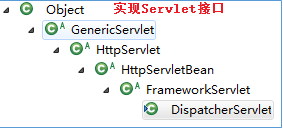
# 源码分析专题

## 知识储备

### Servlet的生命周期方法

* **init：Servlet对象创建之后调用**
* **service：Servlet对象被HTTP请求访问时调用**
* **destroy：Servlet对象销毁之前调用**

### DispatcherServlet继承体系



### InitializingBean接口介绍

* Spring有两种Bean的初始化（**不是实例化**）方式：

\*\*\* 一种是实现**InitializingBean**接口

\*\*\* 一种是通过反射调用bean标签中的**init-method**属性指定的方法。

\*\*\* **不同点**：接口比配置效率高，但是配置消除了对spring的依赖。

* **InitializingBean**接口为bean提供了初始化方法的方式。

\*\*\*它只包括**afterPropertiesSet**方法。

\*\*\*凡是实现该接口的类，在**初始化bean**的时候会执行该方法。

* 实现InitializingBean接口与在配置文件中指定init-method有什么不同？

\*\*\*系统**先调用**afterPropertiesSet方法

\*\*\*然后**再调用**init-method中指定的方法。

* 这种Bean的初始化方式在Spring中是怎么实现的？

\*\*\* **执行时机**：在创建Bean实例及设置完属性之后执行。

\*\*\*通过查看spring加载bean的源码类(**AbstractAutowireCapableBeanFactory**)可看出其中奥妙。

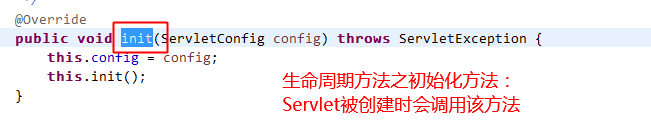
\*\*\*AbstractAutowireCapableBeanFactory类中的invokeInitMethods讲解的非常清楚，源码如下：



## 主流程（DispatcherServlet）

### 初始化流程

* 初始化入口：GenericServlet（实现了Servlet接口）的init(config)



* 接下来准备调用GenericServlet的init()方法了，不过该方法它没有实现，而是被子类HttpServletBean给覆盖了，我们直接看看HttpServletBean的init()方法吧



* 接下来调用initServletBean()，不过该方法需要去HttpServletBean的子类FrameworkServlet中去看看



* 方法调用到这里，我们终于知道DispatcherServlet初始化的主要工作是干嘛的了，就是为了创建Spring容器（其实容器内还要初始化一些组件）。接下来我们去看看FrameworkServlet类的initWebApplicationContext方法

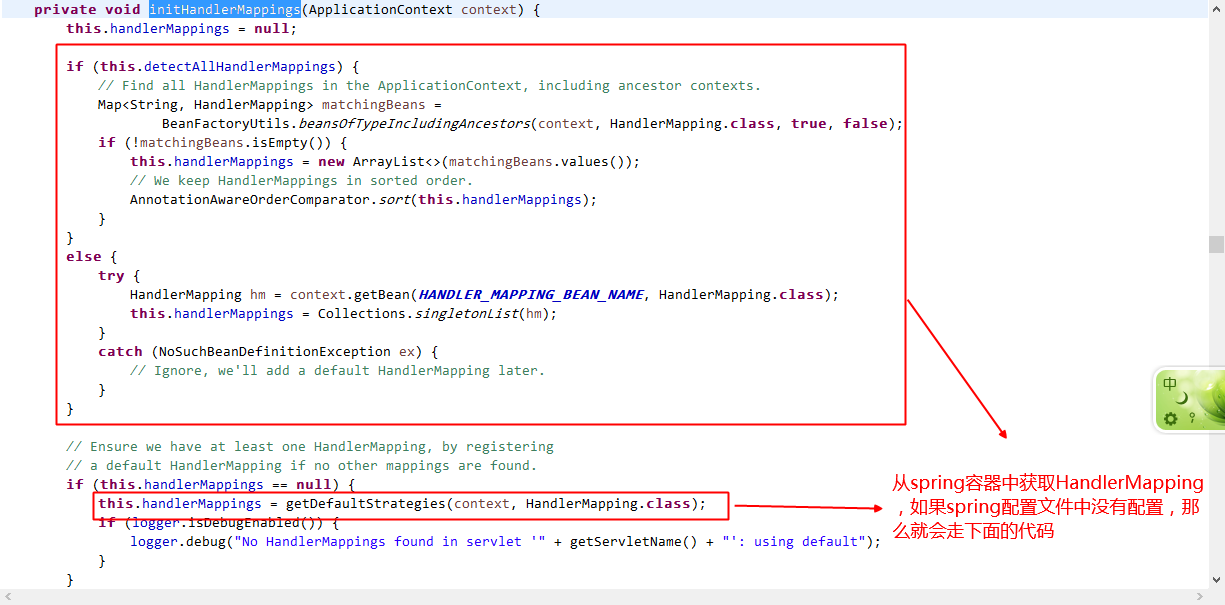




* 557行以上是根据springmvc.xml配置文件创建spring容器，onRefresh()方法是初始化一些默认组件，比如HandlerMapping组件中的（BeanNameURLHandlerMapping），我们进入这个方法看看



* 看到了这个地方，我想大家已经明白入门程序为什么没有配置三大组件，spring容器中却依然有这些组件了吧。这个地方初始化了很多默认配置，我们随便找个initHandlerMapping来了解一下它们是怎么实现的吧



* 看看是如何加载默认策略的，进入getDefaultStrategies方法看看



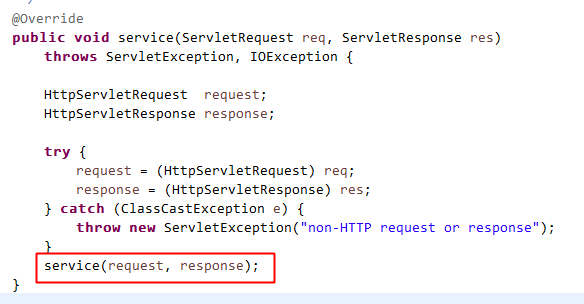
* 那么defaultStrategies集合是如何初始化的呢？



* 至此，DispatcherServlet初始化工作就完成了。

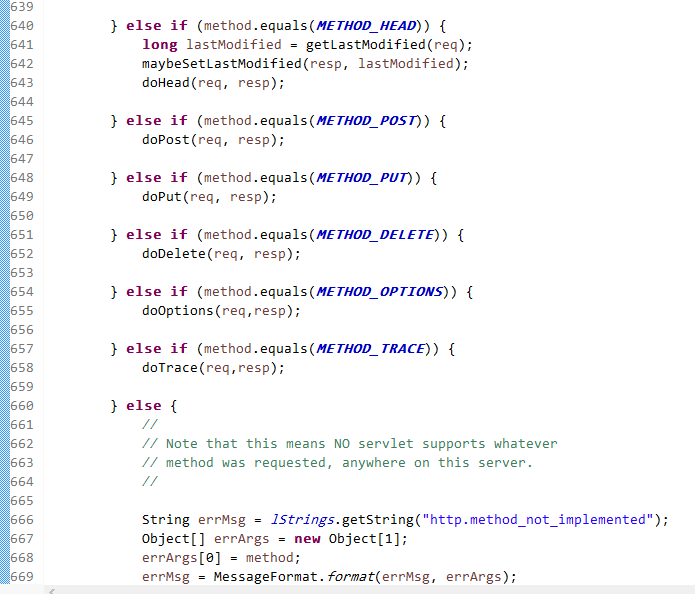
### 访问处理流程

* 访问入口：HttpServlet类的service方法（该方法的request和response参数不是Http开头的）

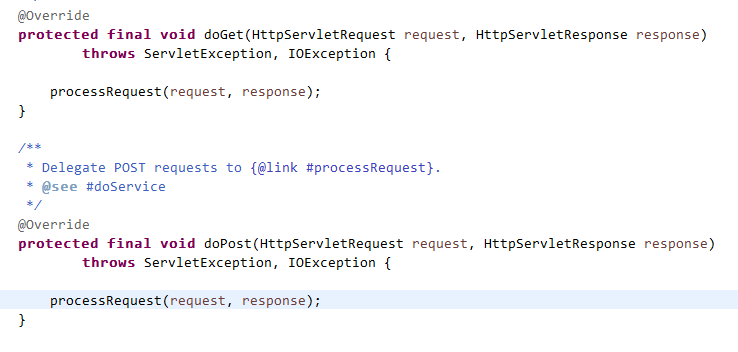


* 继续跟踪到另一个service方法（请求和响应都是HTTP开头），该方法根据请求的method分别调用相应的处理。





* 那接下来我们应该去看看doGet和doPost方法是如何处理请求的？但是问题是我们应该看哪个类中的doGet和doPost方法呢？通过上面的继承体系分析得知，我们应该去FrameworkServlet类看看



* 到现在为止，我们还没有进入到我们的DispatcherServlet类，所以我们继续去processRequest方法看一下



* 通过这个doService方法才正式进入到DispatcherServlet类中去执行

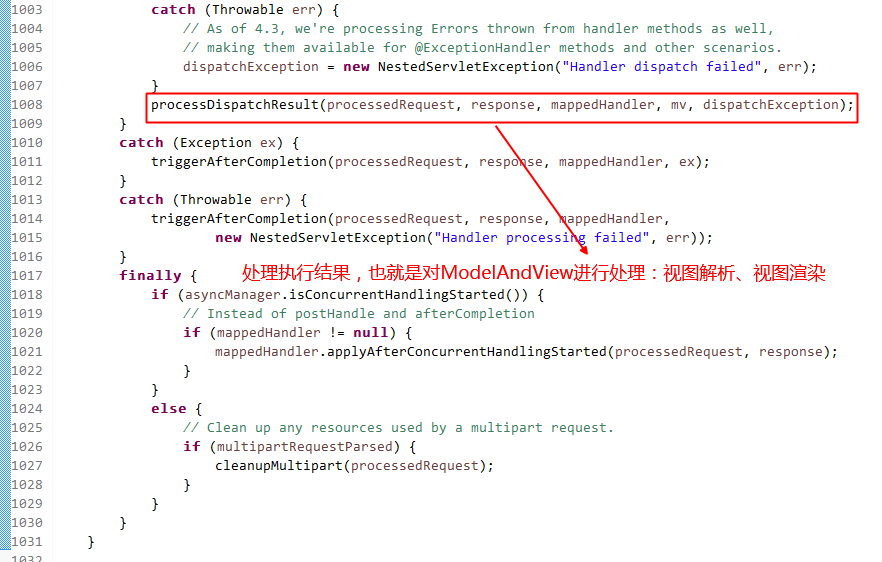




* 至此，我们终于找到了DispatcherServlet中最核心的一个方法：doDispatch，我们一起去看看







* 源码阅读到这里，最起码从主流程中我们可以得知DispatcherServlet是如何处理一个请求的了。但是我们看到现在，还没有看出来**DispatcherServlet是如何与三大组件进行联系**的。所以我们要分别找到三大组件与DispatcherServlet交互的地方。
* 处理器映射器与DispatcherServlet交互的代码：getHandler方法



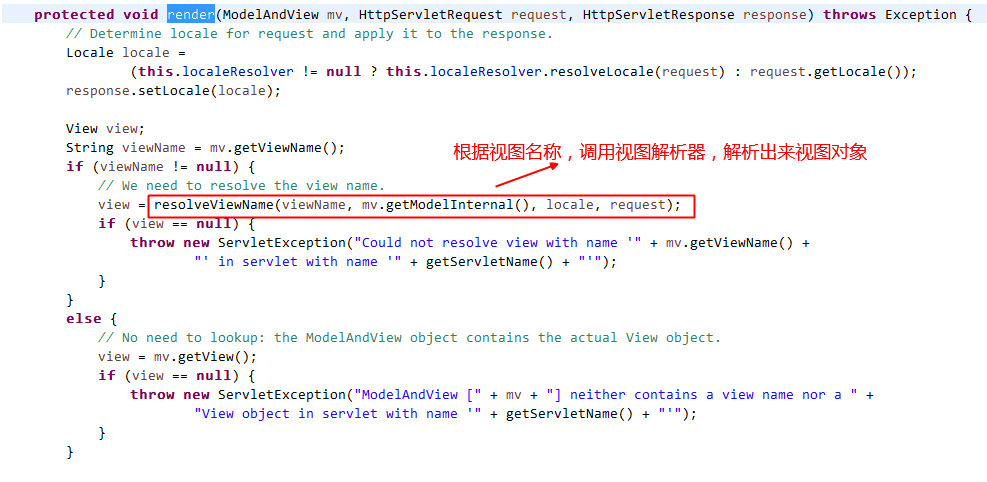
* 处理器适配器与DispatcherServlet交互的代码：getHandlerAdapter方法



* 视图解析器与DispatcherServlet交互的代码需要多深入几层去了解，先进入processDispatchResult方法



* 再进入render方法



* 这个resolveViewName方法就是视图解析器和DispatcherServlet交互的方法



* 到此，DispatcherServlet的主流程以及三大组件和它的联系，我们都已经搞清楚了，如果想继续深入了解三大组件的具体实现，可以自己尝试着去阅读以下。

## 处理器映射器

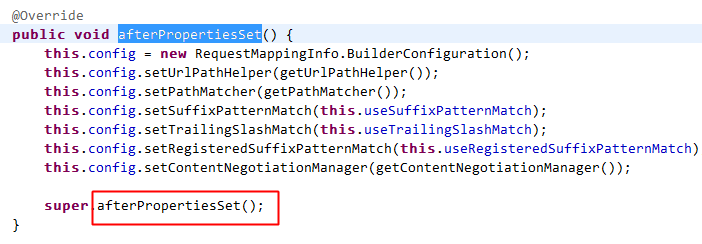
### 注册流程

本章节主要是分析注解方式的处理器映射器：

**RequestMappingHandlerMapping**

**根据@RequestMapping注解查找处理器（HandlerMethod）**

* **分析入口**：RequestMappingHandlerMapping的afterPropertiesSet方法



* 继续进入父类AbstractHandlerMethodMapping的afterPropertiesSet方法看看，其中又调用了intitHandlerMethods方法（**定义了映射器处理的主流程**）



* 我们继续去看看detectHandlerMethods方法（核心处理方法）



* 上面的方法处理逻辑主要分为两大步骤：

\*\*\* 映射Method和RequestMappingInfo的关系

\*\*\* 映射关系：RequestMappingInfo与URL和HandlerMethod的关系。

* 我们先看**第一大步骤**，也就是调用MethodIntrospector.selectMethods方法

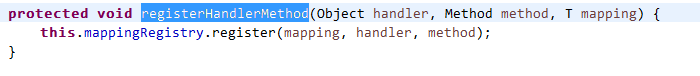


* 我们去看一下MetadataLookup的匿名内部类实现中调用的getMappingForMethod方法是如何实现的？需要去RequestMappingHandlerMapping类中去查看该方法

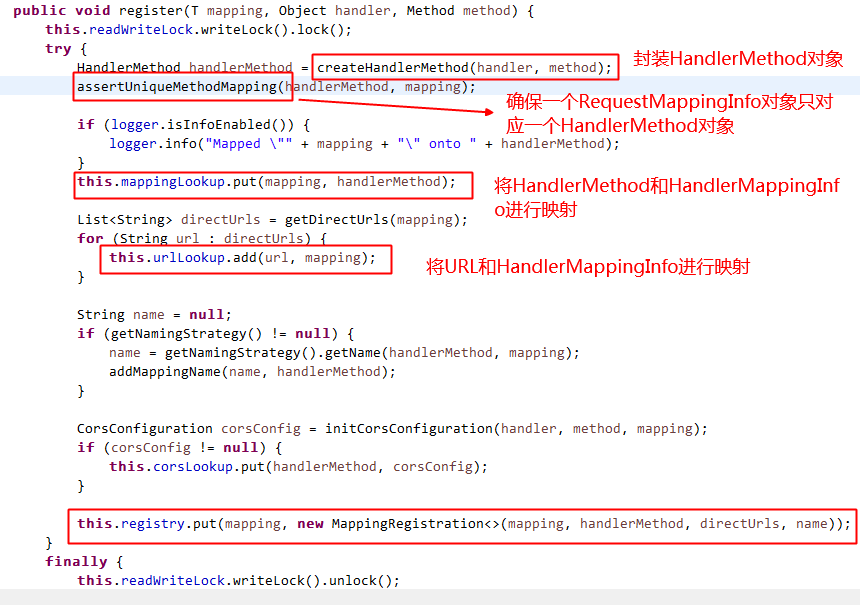




* 至此第一大步骤我们阅读完了，接下来去看看**第二大步骤**，我们看看AbstractHandlerMethodMapping类的registerHandlerMethod方法



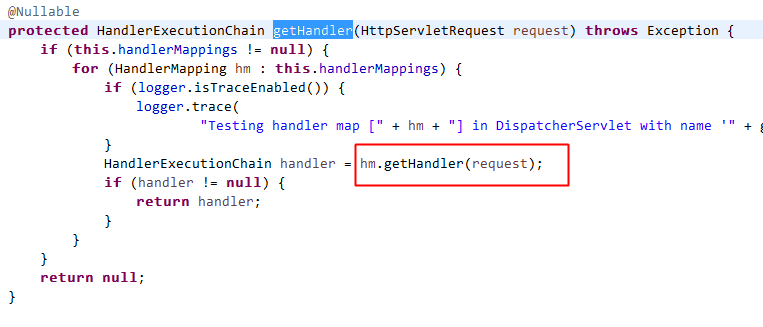
* 接下来，我们去看看register方法



* 至此处理器映射器的初始化流程就完成了。

### 处理流程

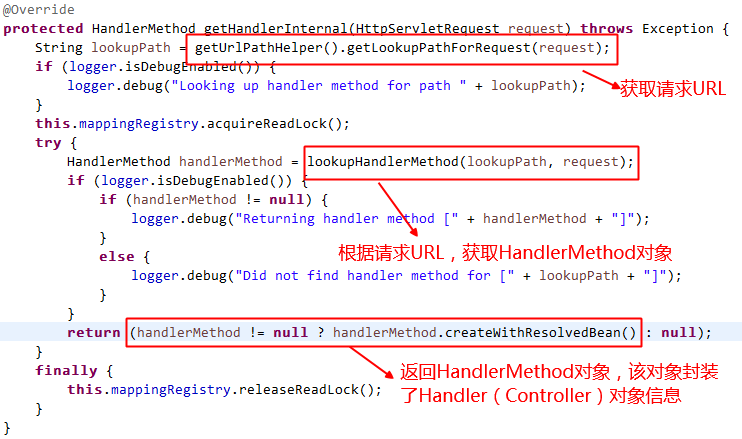
* 分析入口：DispatcherServlet类中的getHandler方法



* 接下来我们需要进入到具体的AbstractHandlerMapping去执行getHandler方法



* 接下来我们重点看看getHandlerInternal方法，这个方法是由AbstractHandlerMethodMapping类去实现的。



* 接下来我们重点去看看lookupHandlerMethod方法（**核心方法**）

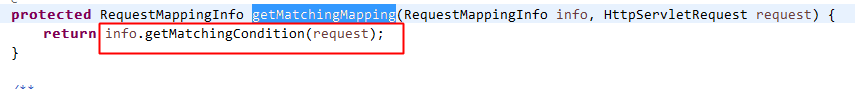




* 我们继续去看看addMatchingMappings方法



* 接着去RequestMappingInfoHandlerMapping类看看getMatchingMapping方法



* 最后去RequestMappingInfo类去看看getMatchingCondition方法



### 拦截器处理流程

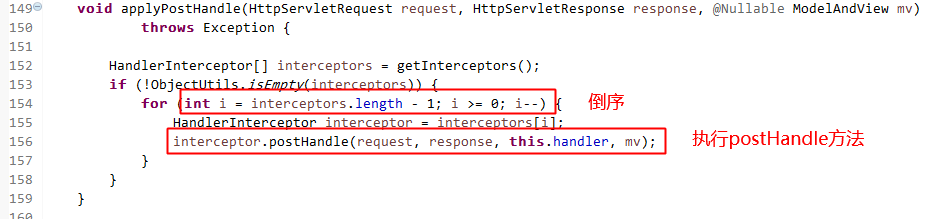
* 分析源码入口：DispatcherServlet的doDispatcher方法



* preHandler流程分析（HandlerExecutionChain类）



* postHandle流程分析（HandlerExecutionChain类）



* afterCompletion流程分析（HandlerExecutionChain类）



## 处理器适配器

### 注册流程

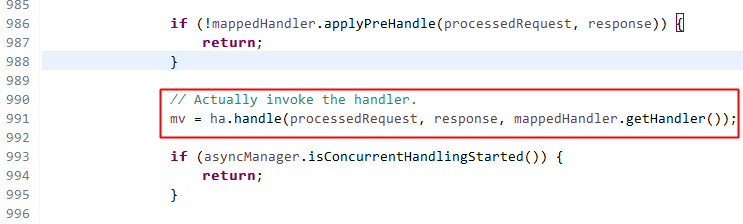
* 分析源码入口：**RequestMappingHandlerAdapter**



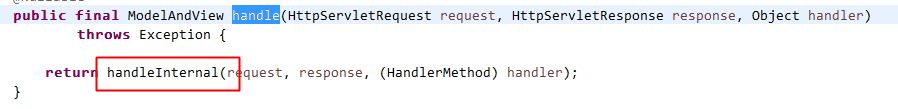
* 以上方法，分别注册了**处理器增强类**的相关操作（**@ModelAttribute、@InitBinder、@ExceptionHandler）**、参数解析器、initBinder参数解析器、返回值处理器
* 这些注册的参数解析器和返回值处理器会在执行Handler方法时进行调用。

### 处理流程

* 分析源码入口：**DispatcherServlet的doDispatcher方法**



* 接着进入AbstractHandlerMethodAdapter类的Handle方法



* 进入**RequestMappingHandlerAdapter类执行handleInternal方法**

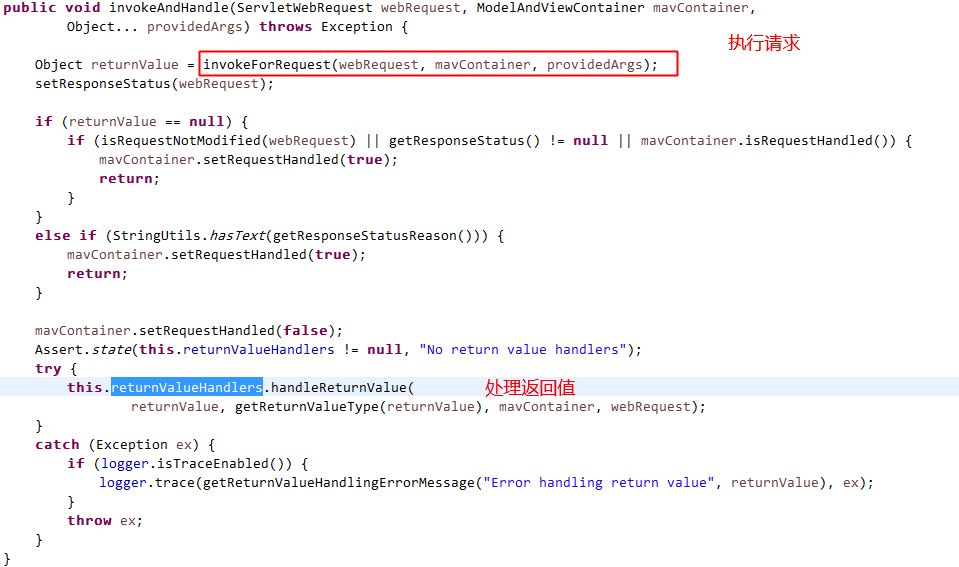


* 进入invokeHandlerMethod方法





* 进入ServletInvocableHandlerMethod类的invokeAndHandle方法



* 到此为止，我们已经看完处理器适配器的处理流程了。

### 参数绑定流程

* 分析入口：InvocableHandlerMethod类的invokeForRequest方法



* 进入getMethodArgumentValues方法



* 重点看看resolveArgument方法，该方法在**HandlerMethodArgumentResolverComposite**类中



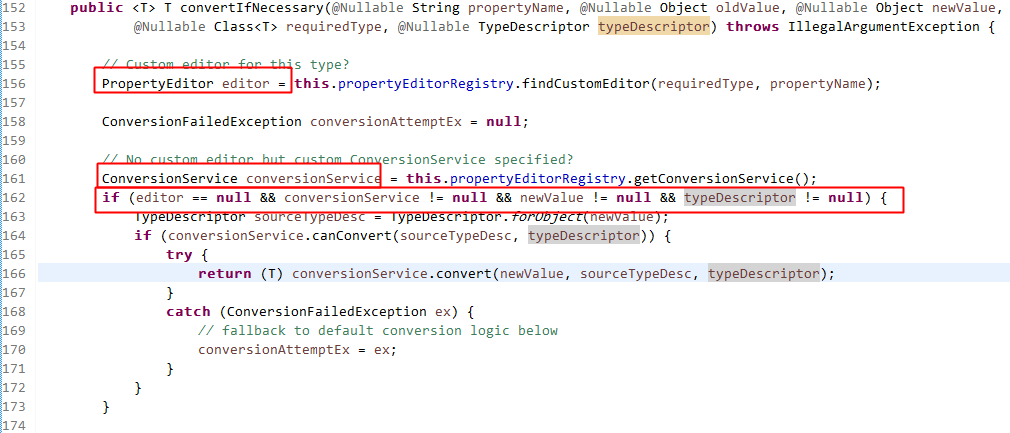
* 此处需要根据具体参数类型调用相应的解析器，其中**简单类型**调用的都是**AbstractNamedValueMethodArgumentResolver**类中的方法；**POJO类型**调用的都是**ModelAttributeMethodProcessor**类中的方法。我们就以简单类型（**对应AbstractNamedValueMethodArgumentResolver**）为例给大家讲解一下具体的参数绑定过程



* 我们进入RequestParamMethodArgumentResolver类中的resolveName方法看看



* 至于参数绑定器的源码比较复杂，我们就不深入追踪了，我们只需要知道**参数转换**方式有两种处理方式：**PropertyEditor和ConversionService**，以及它们处理参数转换的区别就行了。下面的代码是在**TypeConverterDelegate**类中的（至于怎么过来的，同学们自己跟踪一下试试吧）

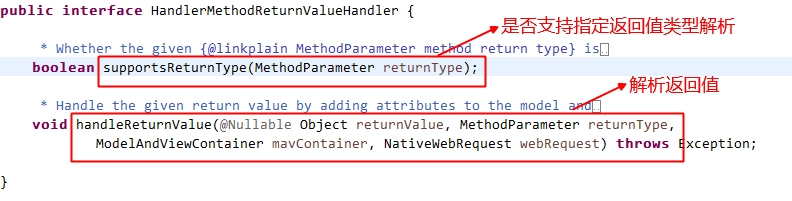


### 返回值处理流程

* 分析入口：ServletInvocableHandlerMethod类的invokeAndHandle方法



* 返回值由this.returnValueHandlers进行处理，这个成员变量的类型是**HandlerMethodReturnValueHandlerComposite**，它继承自HandlerMethodReturnValueHandler类，该类有两个方法：



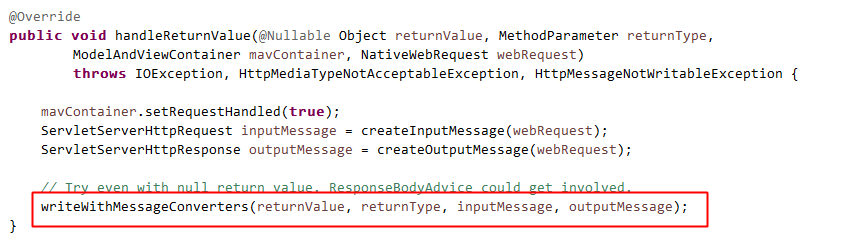
* 我们进入HandlerMethodReturnValueHandlerComposite的handleReturnValue方法



* 至此返回值处理流程我们就分析完了

### @ResponseBody注解解析

* 如果返回值使用@ResponseBody注解，那么由它注解的返回值会使用**RequestResponseBodyMethodProcessor**类进行返回值处理。



* 进入writeWithMessageConverters方法，这个方法在**AbstractMessageConverterMethodProcessor**类中，而且这个方法的处理逻辑比较长，我们挑重点的代码进行阅读

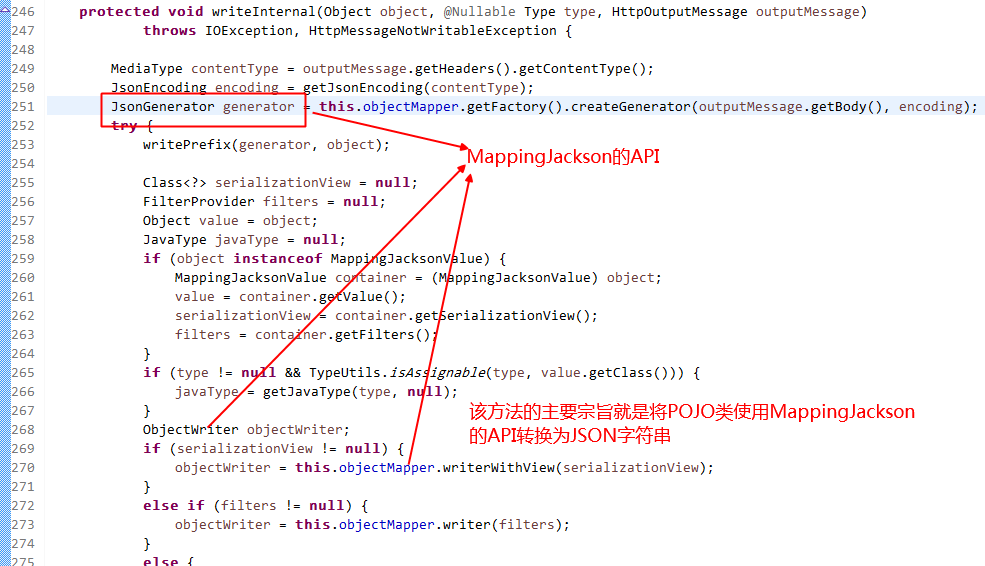


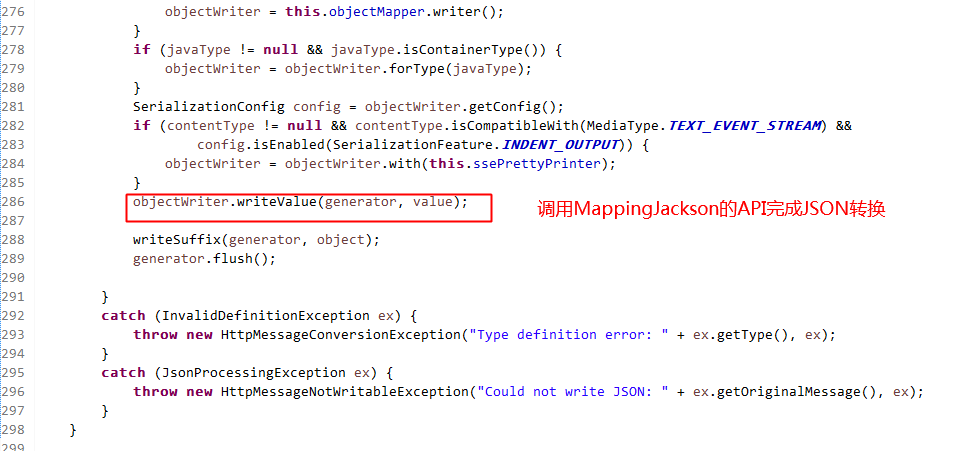


* 继续深入AbstractGenericHttpMessageConverter类的write方法



* 进入





## 视图解析器

### 注册流程

### 处理流程

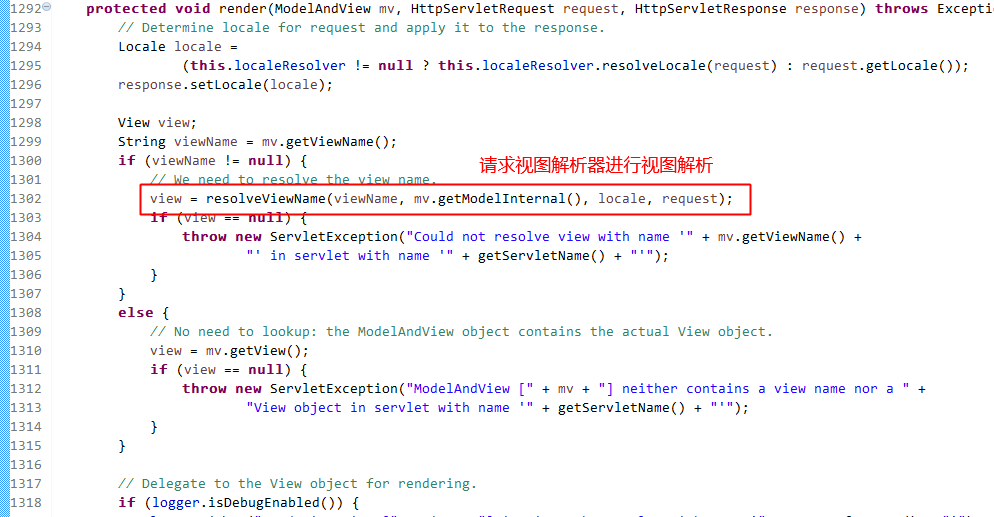
* 分析入口：DispatcherServlet类的1008行代码



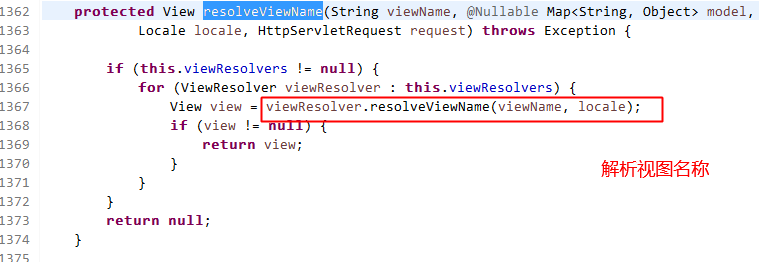
* 进入processDispatchResult方法



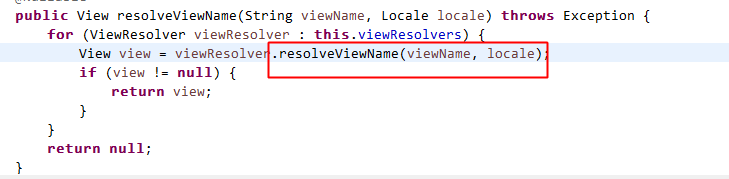
* 我们此时注意目的是为了解决视图解析，所以进入render方法看看



* 进入resolveViewName方法



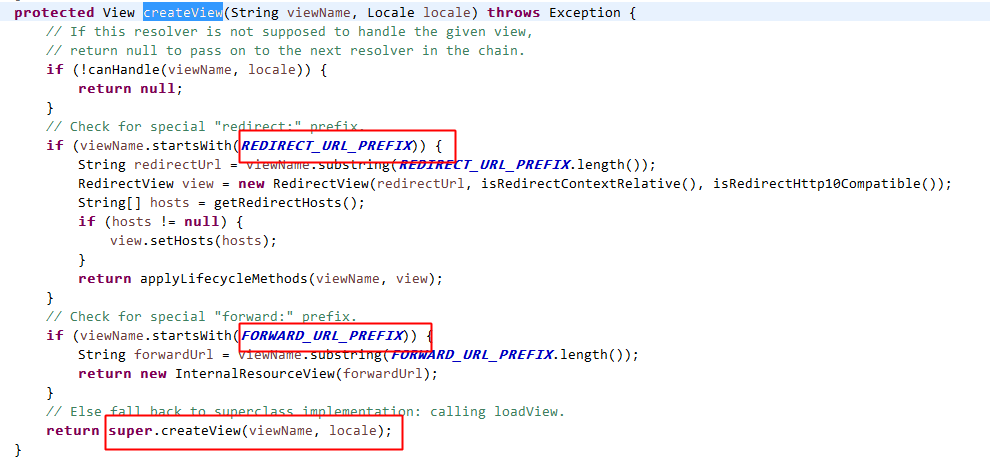
* 接下来进入ViewResolverComposite类的resolveViewName方法



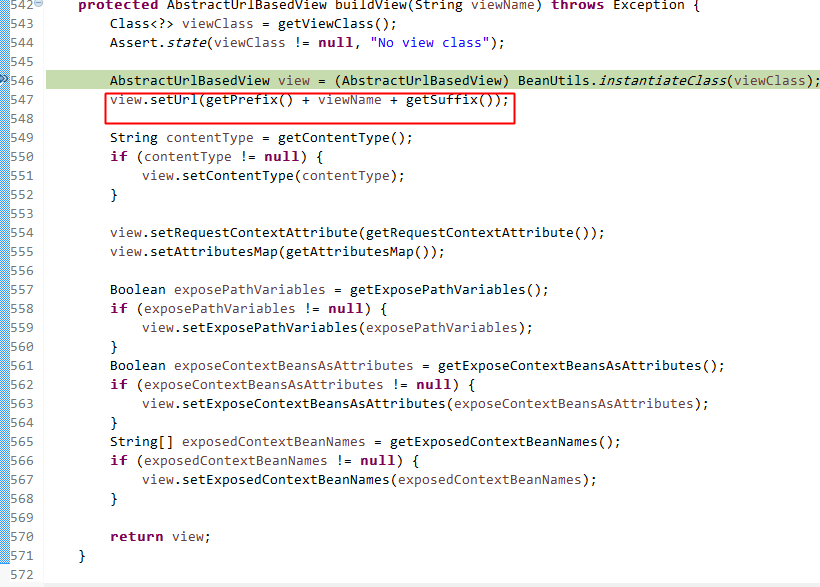
* 接下来进入AbstractCachingViewResolver方法



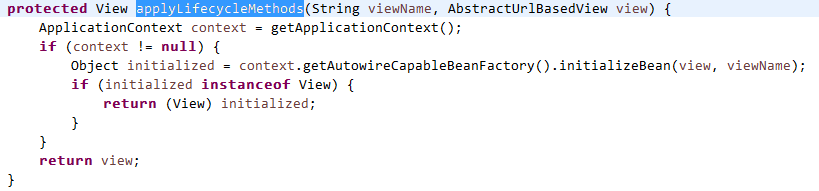
* 进入UrlBasedViewResolver类的createView方法



* 经过几波周转，最终请求进入到UrlBasedViewResolver类的buildView方法，获得基础视图对象



* 最终由该类的以下方法获取真正的视图对象



## mvc:annotation-drivern标签

## 异常处理流程

## 文件上传流程

# 辅助资料

## 非注解方式开发Handler

### 实现HttpRequestHandler接口

|  |
| --- |
| //第二种Handler处理器的编写方式：实现HttpRequestHandler接口  public class ItemController2 implements **HttpRequestHandler** {  @Override  **public** **void** handleRequest(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  List<Item> itemList = **new** ArrayList<Item>();  Item item\_1 = **new** Item();  item\_1.setName("联想笔记本");  item\_1.setPrice(6000f);  item\_1.setDetail("ThinkPad T430 联想笔记本电脑！");  Item item\_2 = **new** Item();  item\_2.setName("苹果手机");  item\_2.setPrice(5000f);  item\_2.setDetail("iphone6苹果手机！");  itemList.add(item\_1);  itemList.add(item\_2);    //数据填充  request.setAttribute("itemList", itemList);    request.getRequestDispatcher("/WEB-INF/jsp/item/item-list.jsp").forward(request, response);  }  } |

### 实现Controller接口

开发时需要实现Controller接口，该接口类只有一个接口方法。

|  |
| --- |
| //第一种方式：实现Controller接口  **public** **class** ItemController **implements** Controller {  @Override  **public** ModelAndView handleRequest(HttpServletRequest req,  HttpServletResponse resp) **throws** Exception {  // **TODO** Auto-generated method stub  List<Item> itemList = **new** ArrayList<Item>();  Item item\_1 = **new** Item();  item\_1.setName("联想笔记本");  item\_1.setPrice(6000f);  item\_1.setDetail("ThinkPad T430 联想笔记本电脑！");  Item item\_2 = **new** Item();  item\_2.setName("苹果手机");  item\_2.setPrice(5000f);  item\_2.setDetail("iphone6苹果手机！");  itemList.add(item\_1);  itemList.add(item\_2);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("itemList", itemList);  // 指定视图的地址  modelAndView.setViewName("WEB-INF/jsp/item/itemList.jsp");  **return** modelAndView;  }  } |

### 非注解Handler配置

|  |
| --- |
| <!-- 配置处理器 -->  <bean class=*"com.kkb.springmvc.controller.ItemController"*/> |

## 非注解方式的处理器映射器和适配器

\*\*\*

如果在DispatcherServlet.properties文件中出现的处理器适配器和处理器映射器，如果没有特殊要求（比如在适配器中配置转换器、在映射器中配置拦截器），那么就不需要再单独进行配置了。

### 映射器之BeanNameUrlHandlerMapping

\*\*\*BeanNameUrlHandlerMapping：

\* 将bean标签的name属性值设置为对应的**url**

\* 这样就可以通过getBean(**url**)获取到指定的处理器对象了。

#### 配置Handler和url映射关系

根据映射器的映射规则，在springmvc.xml中**配置URL和处理器bean的name之间的映射关系**

|  |
| --- |
| <!-- 配置处理器 -->  <bean **name=*"/queryItem"*** class=*"com.kkb.springmvc.controller.ItemController"*/> |

### 适配器之HttpRequestHandlerAdapter

\*\*\*HttpRequestHandlerAdapter（HTTP请求处理器适配器）

\* 通过supports方法适配所有实现了**HttpRequestHandler**接口的处理器

\* 通过handle方法执行适配成功的处理器。

#### Handler实现

该适配器只能适配执行实现了HttpRequestHandler接口的handler类

|  |
| --- |
| **public** **class** ItemController1 **implements** **HttpRequestHandler** {  @Override  **public** **void** handleRequest(HttpServletRequest request,  HttpServletResponse response) **throws** ServletException, IOException {  // **此处省略**  }  } |

#### 适配器配置

* 默认配置:DispatcherServlet.properties
* **显式配置**:在springmvc.xml中进行以下配置：

|  |
| --- |
| <!-- 配置http请求处理器适配器 -->  <bean class=*"org.springframework.web.servlet.mvc.****HttpRequestHandlerAdapter****"*></bean> |

#### 如何适配和执行

|  |
| --- |
| **public** **class** **HttpRequestHandlerAdapter** **implements** HandlerAdapter {  **public** **boolean** **supports**(Object handler) {  **return** (handler **instanceof** HttpRequestHandler);  }  **public** ModelAndView **handle**(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  ((HttpRequestHandler) handler).handleRequest(request, response);  **return** **null**;  }  } |

### 适配器之SimpleControllerHandlerAdapter

\*\*\*SimpleControllerHandlerAdapter（简单控制器处理器适配器）

\* 通过supports方法适配所有实现了**Controller**接口的处理器

\* 通过handle方法执行适配成功的处理器。

#### Handler实现

开发Handler时需要**实现Controller接口**

|  |
| --- |
| //springmvc的handler开发要实现Controller接口（除了实现Controller接口，还可以实现其他接口）  **public** **class** ItemController **implements** **Controller** {  @Override  **public** ModelAndView handleRequest(HttpServletRequest req,  HttpServletResponse resp) **throws** Exception {  // **此处省略**  }  } |

#### 适配器配置

* 默认配置:DispatcherServlet.properties
* **显式配置**:在springmvc.xml中进行以下配置：

|  |
| --- |
| <!-- 配置处理器适配器 ，所有适配器都实现HandlerAdapter接口 -->  <bean class=*"org.springframework.web.servlet.mvc.****SimpleControllerHandlerAdapter****"* /> |

#### 如何适配和执行

|  |
| --- |
| **public** **class** SimpleControllerHandlerAdapter **implements** **HandlerAdapter** {  **public** **boolean** **supports**(Object handler) {  **return** (handler **instanceof** Controller);  }  **public** ModelAndView **handle**(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  **return** ((Controller) handler).handleRequest(request, response);  }  } |

### 注意事项

* 处理器映射器和处理器适配器可以同时存在多个。
* 非注解方式开发的Handler只能使用非注解的映射器和适配器进行处理。

# 乱码解决

## 请求乱码解决之get乱码问题

### GET请求乱码原因分析

\* GET**请求参数**是通过**请求行中的URI**发送给Web服务器（Tomcat）的。

\* Tomcat服务器会对URI进行编码操作（**此时使用的是Tomcat设置的字符集，默认是iso8859-1**）

\* 到了我们的应用程序中的请求参数，已经是被Tomcat使用ISO8859-1字符集进行编码之后的了。

### 解决方式1之修改tomcat配置文件

**指定UTF-8编码，如下：**

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

### 解决方式2之对请求参数进行重新编码：

String username = new

String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")

### 解决方式3之过滤器+请求装饰器统一解决请求乱码

* MyRequestWrapper
* MyCharacterEncodingFilter

## 请求乱码解决之post乱码问题

在web.xml中加入：

|  |
| --- |
| <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

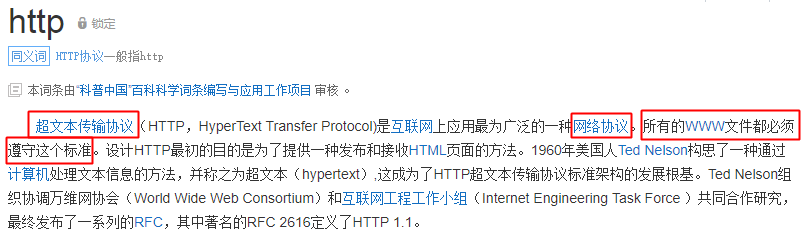
## 响应乱码解决

使用@RequestMapping注解中的produces属性，指定响应体的编码格式

# HTTP协议

## HTTP协议概述

* 什么是HTTP协议？
* HTTP协议的作用是什么？



* **什么是HTTP协议？**

**\* HTTP协议**是建立在**客户端和服务器**之间的一个**协议**，在客户端和服务器之间需要数据的传输，而**传输数据**的时候，我们**要按照指定的规则或者叫协议**去传输数据。

\* HTTP是建立在**TCP/IP协议**基础之上的一个网络协议。

\* HTTP协议属于网络七层结构中最上层（应用层）的协议。

\* HTTP协议是一个无状态协议（不会记录每次访问时的信息）

\* HTTP是一个客户端和服务器端请求和应答的标准（TCP）。客户端是终端用户，服务器端是网站。

* **什么是WEB资源？**

通过浏览器可以访问到的所有资源都是**web资源**

web资源分为**静态资源**和**动态资源**

\* **动态资源**是通过后台程序展示页面数据的，比如Servlet请求。

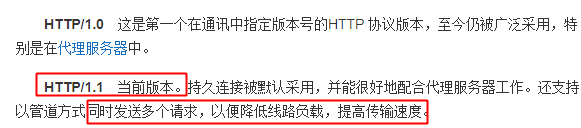
\* **静态资源**的数据是不变的，比如HTML、JPG、AVI。

* **HTTP的作用是什么？**

就是为了约束**客户端和服务器之间传输web资源时的格式**。

## HTTP协议版本

* HTTP协议现在有哪些版本？
* 1.0版本和1.1版本的区别是什么？



HTTP1.1和HTTP1.0版本之间最大的区别就是：可以一个连接传输多个web资源。

**推荐使用HTTP1.1版本！！！**

## HTTP协议组成

HTTP协议由两部分组成：请求协议信息和响应协议信息。

### 请求协议信息

* 请求协议由哪几部分组成？
* 请求协议的请求行包含哪些信息？
* 请求协议的请求头如何理解？请求头中常用的一些配置的作用各自是什么？
* MIME是什么？常见的MIME类型有哪些？
* 请求协议的请求体有几种表现形式？

**HTTP请求协议信息由三部分组成**：请求行、请求头、请求体，简称行头体。



* **请求行**

也叫请求首行，它包含四部分（请求方法、URI、协议/版本、回车换行）：

GET /user.html HTTP/1.1

**请求方法**：GET、POST等8种



互联网中WEB资源操作也有增删改查方法，它们分别是POST、DELETE、PUT和GET。

**URI**：Uniform Resource Identifier，**统一资源标识符**。它相当于一个网络资源的名称，只是名称的表现形式是/开头的路径形式。

URL：Uniform Resource Location，统一资源定位符

URL和URI的区别：URL是URI的子集。

**协议/版本**：表示这次请求是通过哪个协议发送的，比如HTTP协议、HTTPS协议等，**使用的HTTP协议一般都是1.1版本的**。

* **请求头**

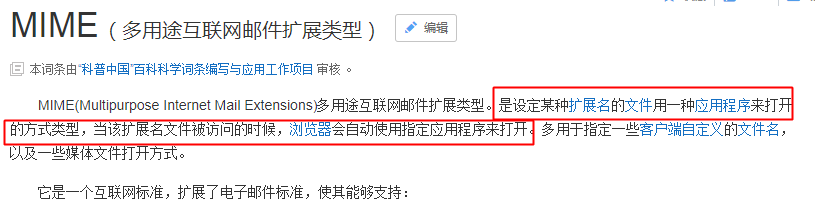
请求头的信息是以**key:value形式**展现的。

一般来说，大多数请求头的信息都不是必须的，我们只需要了解一些常见的请求头信息即可！

**请求头说明：**

|  |  |
| --- | --- |
| **Content-Type** | 是请求消息中非常重要的内容，表示请求正文中的文档属于什么MIME类型。**Content-Type: [type]/[subtype]; parameter**。例如最常见的就是text/html，它的意思是说返回的内容是文本类型，这个文本又是HTML格式的。 |
| **Host** | 指定请求资源的Intenet主机和端口号，必须表示请求url的原始服务器或网关的位置。HTTP/1.1请求必须包含主机头域，否则系统会以400状态码返回 |
| **Accept** | 浏览器可接受的MIME类型 |
| Accept-Charset | 浏览器可接受的字符集 |
| Accept-Encoding | 浏览器能够进行解码的数据编码方式，比如gzip。Servlet能够向支持gzip的浏览器返回经gzip编码的HTML页面。许多情形下这可以减少5到10倍的下载时间 |
| Accept-Language | 浏览器所希望的语言种类，当服务器能够提供一种以上的语言版本时要用到 |
| Authorization | 授权信息，通常出现在对服务器发送的WWW-Authenticate头的应答中 |
| Connection | 表示是否需要持久连接。如果Servlet看到这里的值为“Keep- Alive”，或者看到请求使用的是HTTP1.1（HTTP 1.1默认进行持久连接），它就可以利用持久连接的优点，当页面包含多个元素时（例如Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet需要在应答中发送一个Content-Length头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小 |
| Content-Length | 表示请求消息正文的长度 |
| **Cookie** | 这是最重要的请求头信息之一，可以在客户端记录访问状态。 |
| From | 请求发送者的email地址，由一些特殊的Web客户程序使用，浏览器不会用到它 |
| If-Modified-Since | 只有当所请求的内容在指定的日期之后又经过修改才返回它，否则返回304“Not Modified”应答 |
| Pragma | 指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝 |
| **Referer** | 包含一个URL，用户从该URL代表的页面出发访问当前请求的页面，使用场景：防盗链、统计网站访问信息。 |
| **User-Agent** | 浏览器类型（**客户端类型**），如果Servlet返回的内容与浏览器类型有关则该值非常有用 |
| UA-Pixels，UA-Color，UA-OS，UA-CPU | 由某些版本的IE浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和CPU类型 |

**\* MIME概述：**多用途互联网邮件扩展类型，也叫媒体类型。



**\* MIME格式：**大类型/小类型，阅读是反过来，比如text/html,读成html文本。

**\* 常见MIME类型如下**：



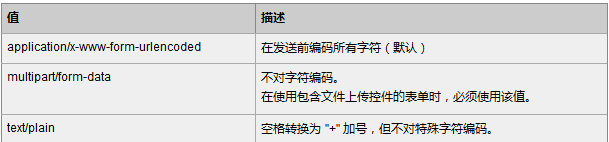
* **请求体**

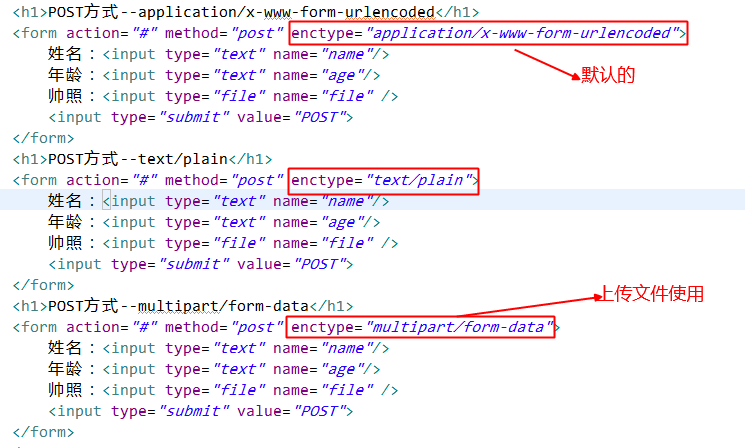
也叫请求正文。

\* GET请求的请求体是空的，请求参数都是通过请求行传给服务器端

\* POST请求的请求体可以承载数据，请求头和请求体之间有一个空行作为分割线。

**\* 通过表单POST提交的请求体的表现形式主要有三种：**





* **总结**

请求协议由三部分组成：行头体

\* 请求首行：请求方法 web资源URI http/1.1

\* 请求头：key value方式，不同的请求头配置会告诉服务器端不同的辅助信息。

\* 请求体：承载传输的具体数据，不过请求体中的数据表现形式有三种，这三种都是通过form表单的enctype属性来决定的。

* application/x-www-form-urlencoded：会对中文进行URL编码，并且多个参数以&连接，上传文件只能上传文件名称。
* text/plain：纯文本方式，不会对中文进行URL编码，不会使用&连接多个key-value参数，上传文件只能上传文件名称。
* multipart/form-data：多部件表现形式，这种方式主要可以完成文件上传，可以将上传的文件名称和文件内容都传递给服务器端。

MIME：媒体类型，可以表示互联网web资源的扩展名。

text/html .html

text/plain .txt

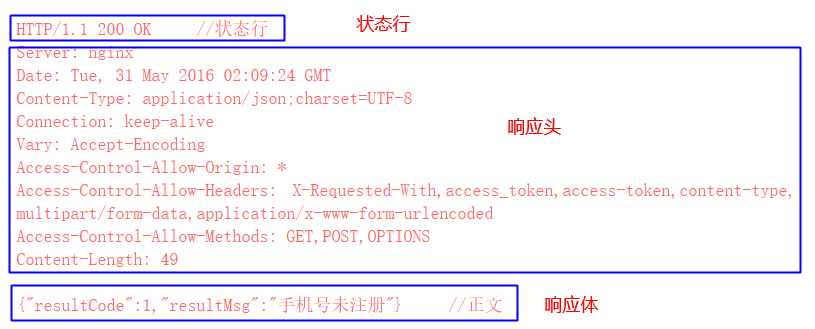
image/jpeg .jpeg

image/gif .gif

### 响应协议信息

* 响应协议由哪几部分组成？
* 响应协议的状态行包含哪些信息？
* 状态行中的状态码的含义分别是如何表示的？列出常见的几个状态码及说明？
* 响应协议的响应头包含哪些头信息？
* 响应协议的响应体如何理解？
* **响应协议由哪几部分组成？**

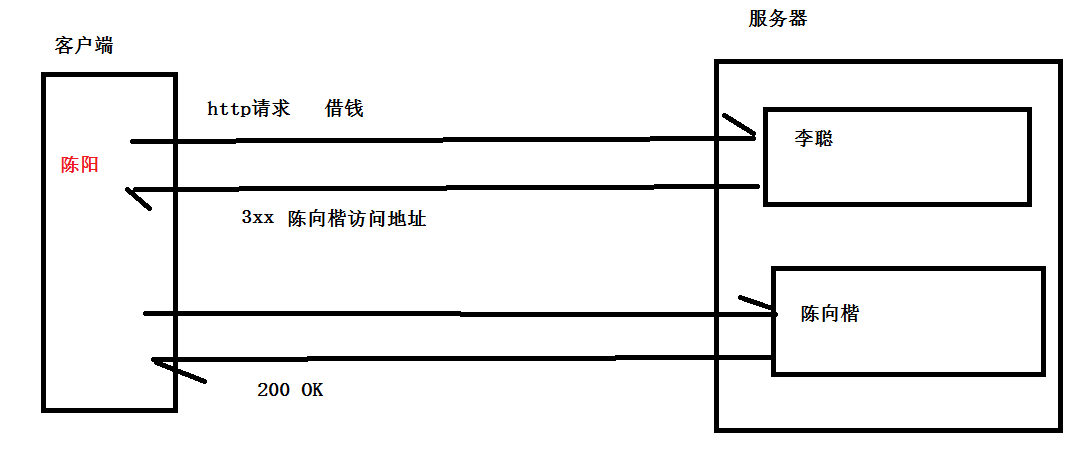
**响应协议信息，也由三部分组成：状态行、响应头、响应体（响应正文）。**



* **状态行**

HTTP/1.1 200 OK

状态行由**协议/版本**、**数字形式的状态码**、**状态描述**三部分组成。



**状态码说明：**



* **响应头**

响应头中的信息也是key value方式展现的。

|  |  |
| --- | --- |
| **Content-Type** | 是返回消息中非常重要的内容，表示后面的文档属于什么MIME类型。Content-Type: [type]/[subtype]; parameter。例如最常见的就是text/html，它的意思是说返回的内容是文本类型，这个文本又是HTML格式的。原则上浏览器会根据Content-Type来决定如何显示返回的消息体内容 |
| **Location** | Location响应报头域用于重定向接受者到一个新的位置。例如：客户端所请求的页面已不存在原先的位置，为了让客户端重定向到这个页面新的位置，服务 器端可以发回Location响应报头后使用重定向语句，让客户端去访问新的域名所对应的服务器上的资源。当我们在JSP中使用重定向语句的时候，服务器 端向客户端发回的响应报头中，就会有Location响应报头域。 |
| **Server** | Server响应报头域包含了服务器用来处理请求的软件信息。它和User-Agent请求报头域是相对应的，前者发送服务器端软件的信息，后者发送客户 端软件(浏览器)和操作系统的信息。下面是Server响应报头域的一个例子：Server: Apache-Coyote/1.1 |
| WWW-Authenticate | WWW-Authenticate响应报头域必须被包含在401(未授权的)响应消息中，这个报头域和前面讲到的Authorization请求报头域是 相关的，当客户端收到401响应消息，就要决定是否请求服务器对其进行验证。如果要求服务器对其进行验证，就可以发送一个包含了 Authorization报头域的请求，下面是WWW-Authenticate响应报头域的一个例子：WWW-Authenticate: Basic realm="Basic Auth Test!"  从这个响应报头域，可以知道服务器端对我们所请求的资源采用的是基本验证机制。 |
| **Content-Length** | 表示响应消息正文的长度 |
| Expires | Expires实体报头域给出响应过期的日期和时间。通常，代理服务器或浏览器会缓存一些页面。当用户再次访问这些页面时，直接从缓存中加载并显示给用 户，这样缩短了响应的时间，减少服务器的负载。为了让代理服务器或浏览器在一段时间后更新页面，我们可以使用Expires实体报头域指定页面过期的时 间。当用户又一次访问页面时，如果Expires报头域给出的日期和时间比Date普通报头域给出的日期和时间要早(或相同)，那么代理服务器或浏览器就 不会再使用缓存的页面而是从服务器上请求更新的页面。不过要注意，即使页面过期了，也并不意味着服务器上的原始资源在此时间之前或之后发生了改变。 |
| Last-Modified | Last-Modified实体报头域用于指示资源最后的修改日期及时间。 |
| **Set-Cookie** | 设置和页面关联的Cookie。Servlet不应使用response.setHeader("Set-Cookie", ...)，而是应使用HttpServletResponse提供的专用方法addCookie。参见下文有关Cookie设置的讨论。 |
| **Allow** | 服务器支持哪些请求方法（如GET、POST等）。 |
| Content-Encoding | 文档的编码（Encode）方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩文档能够显著地减少HTML文档的下载时间。Java的GZIPOutputStream可以很方便地进行gzip压缩，但只有Unix上的Netscape和Windows上的IE 4、IE 5才支持它。因此，Servlet应该通过查看Accept-Encoding头（即request.getHeader("Accept-Encoding")）检查浏览器是否支持gzip，为支持gzip的浏览器返回经gzip压缩的HTML页面，为其他浏览器返回普通页面。 |

* 响应体

响应体，也叫响应正文，里面包含服务器发给客户端的web资源信息。

响应正文信息返回到浏览器时，浏览器需要根据响应头中**Content-type**设置的MIME类型来打开响应正文信息。

## GET请求和POST请求的区别

* 提交数据的方式不同

GET是通过请求行提交请求参数的。

POST是通过请求体提交请求参数的。

* 使用场景不同

GET请求的目的是获取到数据，简单点说，就是客户端向服务器端要东西

POST请求的目的是给服务器提交数据。就是客户端向服务器端给东西。

* 传递参数的大小不同

GET请求是通过请求行中的请求URL传递给客户端的。HTTP协议对请求URL的长度没有限制，但是不同的浏览器对请求URL长度是由限制的。

POST请求是通过请求体传递请求参数的。

总之POST传递的请求参数大小比GET方式要大，要多。

# 设计模式讲解

## 装饰模式

\*\*\* 通过**装饰类**可以将一个**目标类**进行**增强。**

\*\*\* **装饰类**和**目标类**依然是同一个类型（**接口**）

* 接口类

|  |
| --- |
| **public** **interface** Iphone {  **void** call();  } |

* 目标类

|  |
| --- |
| **public** **class** Iphone6 **implements** Iphone {  @Override  **public** **void** call() {  System.***out***.println("使用iphone6打电话，性能刚刚滴");  }  } |

* 装饰类

|  |
| --- |
| **public** **class** IphoneDecorate **implements** Iphone{  //被装饰的目标类  **private** Iphone iphone;  //通过构造参数将被装饰的类，传入过来  **public** IphoneDecorate(Iphone iphone) {  **super**();  **this**.iphone = iphone;  }  @Override  **public** **void** call() {  System.***out***.println("人猿泰山music。。。。。。");  iphone.call();  }  } |

* 测试类

|  |
| --- |
| **public** **class** TestDecorate {  @Test  **public** **void** test() {  Iphone iphone = **new** Iphone6();  iphone.call();  System.***out***.println("=======================");  //装饰之后，依然是一个手机  Iphone iphone2 = **new** IphoneDecorate(iphone);  iphone2.call();  }  } |

## 适配模式

\*\*\* 将不同的类型适配成同一类型

* A接口

|  |
| --- |
| **public** **interface** DBSocket {  **void** charge();  } |

* A实现类

|  |
| --- |
| **public** **class** DBSocketImpl **implements** DBSocket {  @Override  **public** **void** charge() {  System.***out***.println("使用两眼插孔充电");  }  } |

* B接口

|  |
| --- |
| **public** **interface** GBSocket {  **void** charge();  } |

* B实现类

|  |
| --- |
| **public** **class** GBSocketImpl **implements** GBSocket {  @Override  **public** **void** charge() {  System.***out***.println("使用三眼插孔充电");  }  } |

* 适配器类

|  |
| --- |
| **public** **class** SockerAdapter {  //不同类型的插座  **private** Object socket;  **public** SockerAdapter(Object socket) {  **super**();  **this**.socket = socket;  }  //统一的充电方法  **public** **void** charge() {  **if** (socket **instanceof** GBSocket) {  ((GBSocket)socket).charge();  }**else** **if** (socket **instanceof** DBSocket) {  ((DBSocket)socket).charge();  }  }  } |

* 测试类

|  |
| --- |
| **public** **class** TestAdapter {  @Test  **public** **void** test() {  //德国标准的插座充电测试  DBSocket socket = **new** DBSocketImpl();  socket.charge();  System.***out***.println("==========");  //中国标准的插座充电测试  GBSocket gbSocket = **new** GBSocketImpl();  gbSocket.charge();  System.***out***.println("==========");    //通过适配器将国标的插排适配成统一的标准  SockerAdapter adapter1 = **new** SockerAdapter(gbSocket);  adapter1.charge();  System.***out***.println("==========");  //通过适配器将德标的插排适配成统一的标准  SockerAdapter adapter2 = **new** SockerAdapter(socket);  adapter2.charge();  }  } |

## 策略模式