# Springmvc

1、springMVC简介

2、hello world

3、整体架构介绍

4、注解

5、如何配置springmvc的访问路径

6、如何接受用户传递过来的参数

7、json的处

8、文件上传

9、spring的拦截器

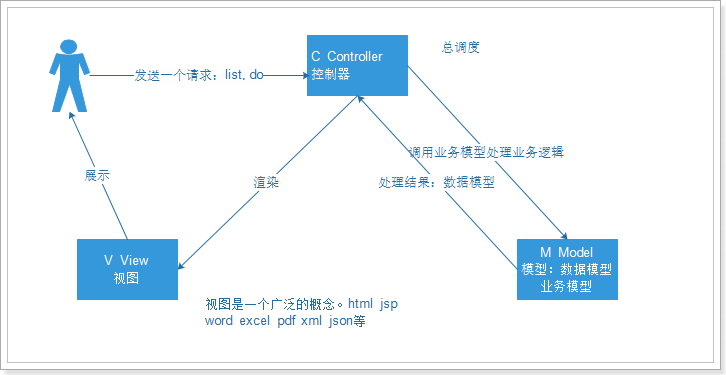
# 认识Springmvc

## MVC回顾

1、 模型（Model）：负责封装应用的状态，并实现应用的功能。通常分为数据模型和业务逻辑模型，数据模型用来存放业务数据，比如订单信息、用户信息等；而业务逻辑模型包含应用的业务操作，比如订单的添加或者修改等。通常由java开发人员编写程序完成，代码量最多

　　2、 视图（View）：视图通过控制器从模型获得要展示的数据，然后用自己的方式展现给用户，相当于提供界面来与用户进行人机交互。通常有前端和java开发人员完成，代码量较多。

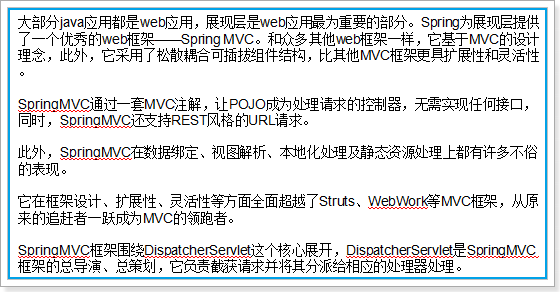
　　3、 控制器（Controller）：用来控制应用程序的流程和处理用户所发出的请求。当控制器接收到用户的请求后，会将用户的数据和模型的更新相映射，也就是调用模型来实现用户请求的功能；然后控制器会选择用于响应的视图，把模型更新后的数据展示给用户。起到总调度的作用，Controller通常由框架实现，使用时基本不需要编写代码



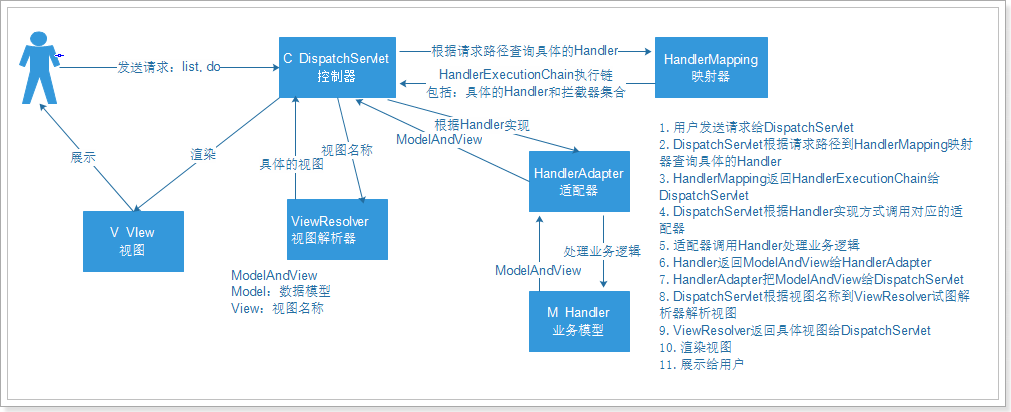
Mvp

Mvvm

## SpringMVC介绍

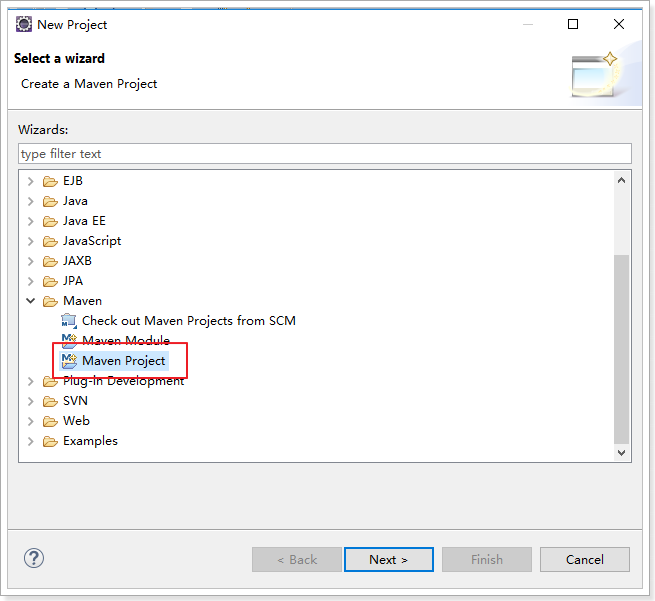


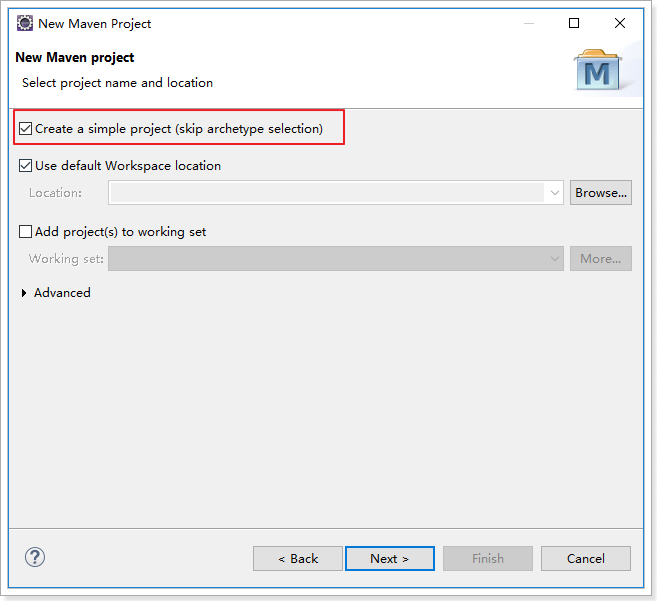
## Springmvc架构

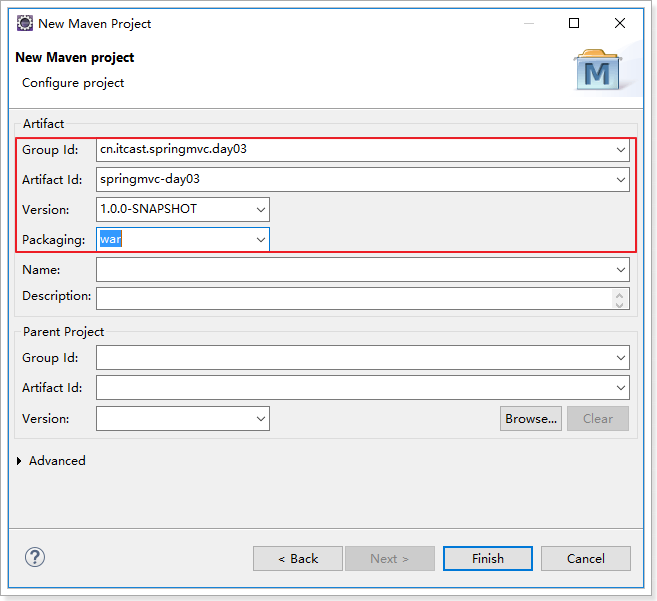


# 第一个springmvc程序(Hello World)

## 创建工程





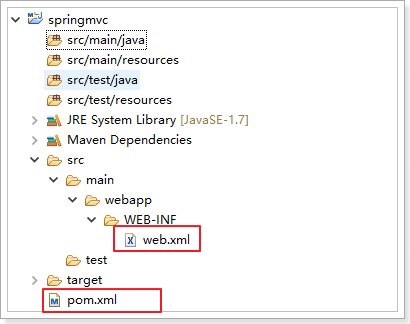


## 引入依赖

Pom.xml

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  </dependency>  <!-- JSP相关 -->  <dependency>  <groupId>jstl</groupId>  <artifactId>jstl</artifactId>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jsp-api</artifactId>  <scope>provided</scope>  </dependency>  </dependencies>  <build>  <plugins>  <!-- 配置Tomcat插件 -->  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <configuration>  <port>8080</port>  <path>/</path>  </configuration>  </plugin>  </plugins>  </build> |

## 配置web.xml



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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* version=*"2.5"*>

<display-name>springmvc</display-name>

<servlet>

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

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

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

</servlet>

<servlet-mapping>

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

<!--

/\*：拦截所有请求，包括jsp

/ ：拦截所有请求，不包含jsp

\*.do,\*.action

-->

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

</servlet-mapping>

<welcome-file-list>

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

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

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

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

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

</welcome-file-list>

</web-app>

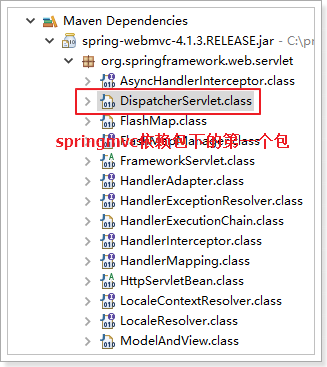
## springmvc的配置文件

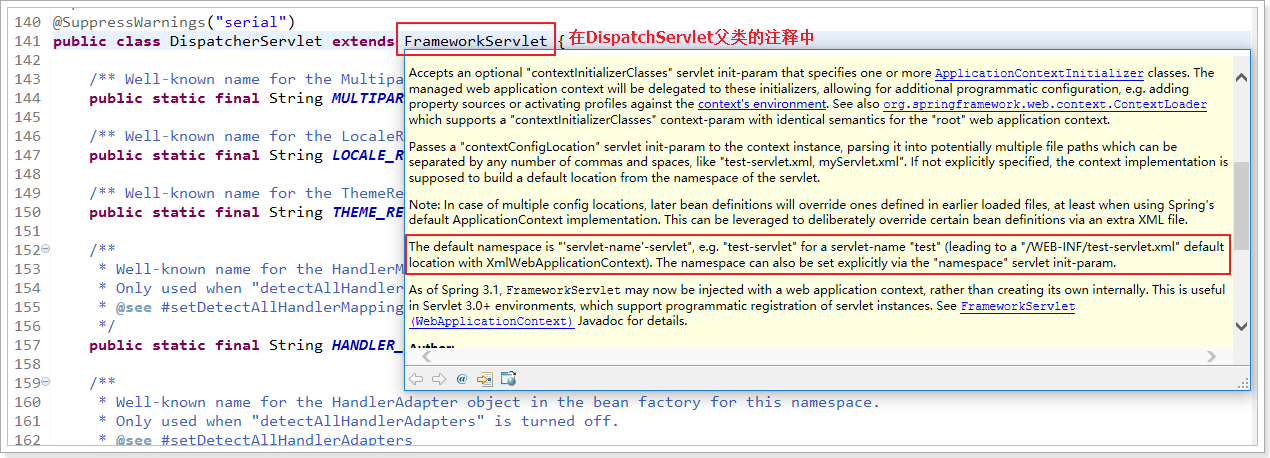
### {servlet-name}-servlet.xml

用户发送请求到web容器，并被DispatchServlet拦截之后进入springmvc容器，springmvc该怎么处理那，这就需要springmvc的配置文件。

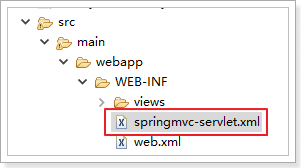
那么springmvc的配置文件该放在什么位置，又该怎么命名呢？

找到DispatchServlet这个类：





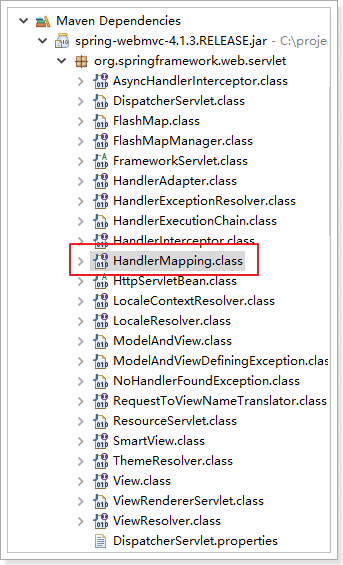
由此知道，springmvc默认读取/WEB-INF/{servlet-name}-servlet.xml这个配置文件，因为我们在web.xml中的servlet-name配置的是springmvc，所以在WEB-INF目录下创建springmvc-servlet.xml文件：

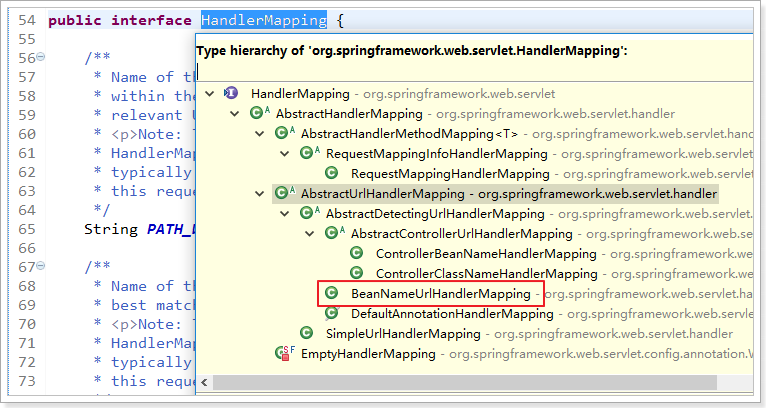


springmvc配置文件的头信息和spring一样。

|  |
| --- |
| <?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-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*>  </beans> |

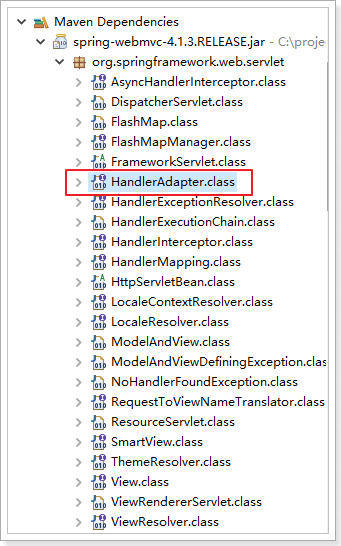
### HandlerMapping映射器

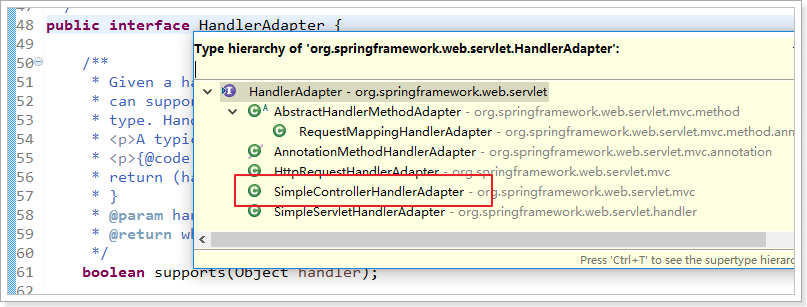




|  |
| --- |
| <!-- 配置HandlerMapping -->  <!-- 把bean的name属性作为Url -->  <bean class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"* /> |

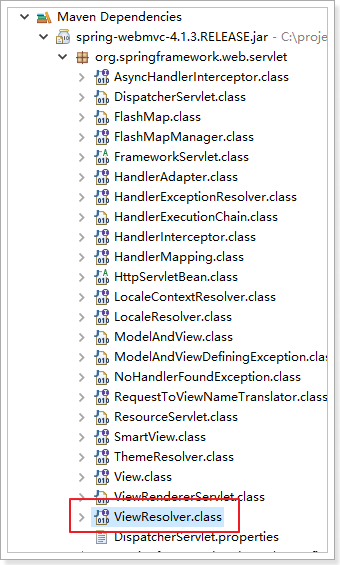
### HandlerAdapter适配器

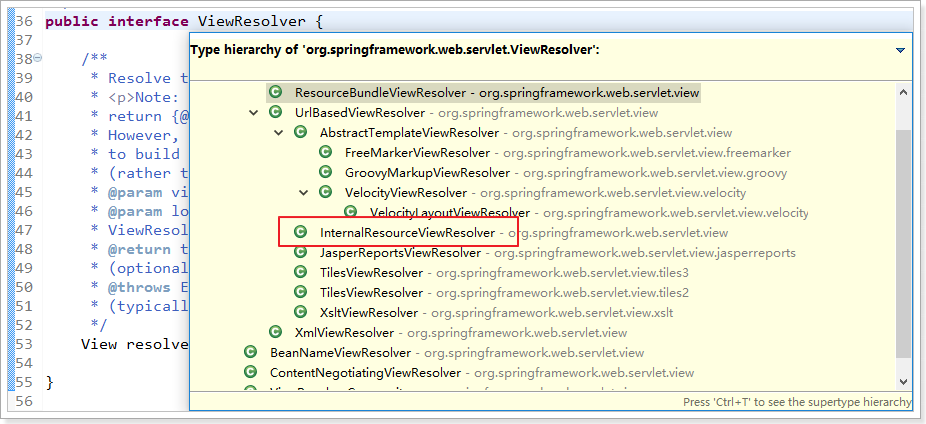


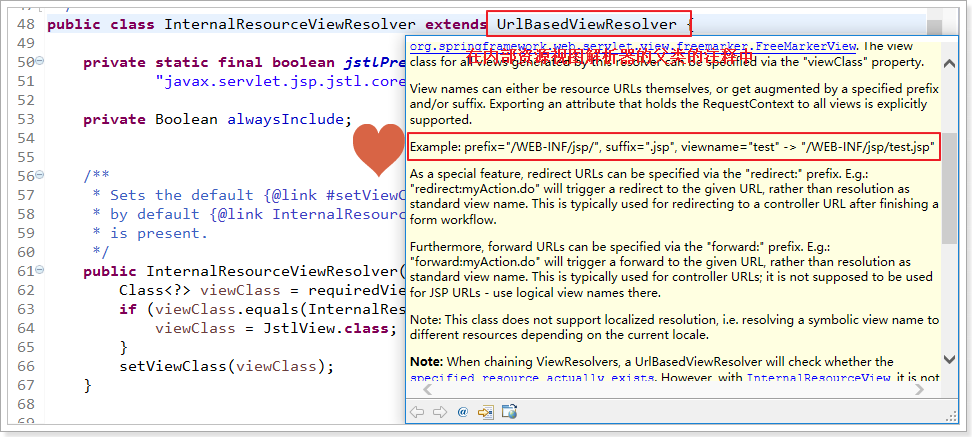


|  |
| --- |
| <!-- 配置HandlerAdapter -->  <bean class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"* /> |

### ViewResolver试图解析器





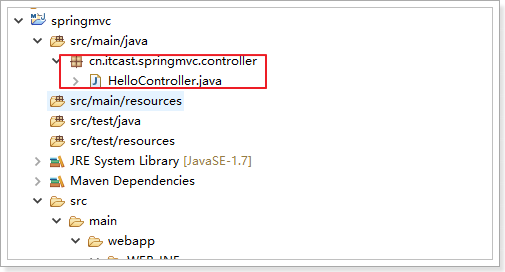


由此可见，视图解析器的规则是：prefix+viewName+suffix

### 完整的配置

|  |
| --- |
| <?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-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*>    <!-- 配置映射器,把bean的name属性作为一个url -->  <bean class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"* />    <!-- 配置适配器 -->  <bean class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"* />    <bean name=*"/hello.do"* class=*"cn.itcast.springmvc.controller.HelloController"* />    <!-- 配置视图解析器 -->  <!-- Example: prefix="/WEB-INF/jsp/", suffix=".jsp", viewname="test" -> "/WEB-INF/jsp/test.jsp" -->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/views/"*></property>  <property name=*"suffix"* value=*".jsp"*></property>  </bean>    </beans> |

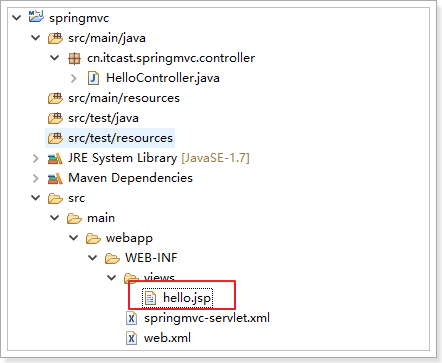
## HelloController



HelloController内容：

|  |
| --- |
| **public** **class** HelloController **implements** Controller {  @Override  **public** ModelAndView handleRequest(HttpServletRequest request,  HttpServletResponse response) **throws** Exception {  ModelAndView mv = **new** ModelAndView();  mv.setViewName("hello");  **return** mv;  }  } |

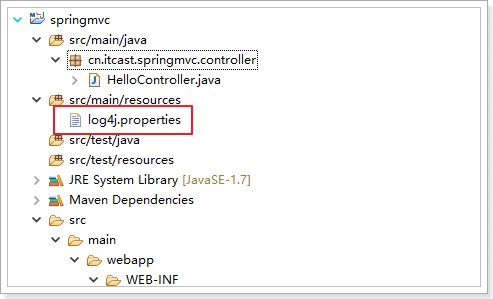
## 添加jsp页面（hello.jsp）



Jsp内容：

|  |
| --- |
| <%@ 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>Insert title here</title>  </head>  <body>  <!-- <span style="font-size:30px; color:red;">第一个springmvc程序</span> -->  <span style="font-size:*30px*; color:*red*;">springmvc的第一个程序：${msg }</span>  </body>  </html> |

## 添加log日志



Log4j.properties内容：

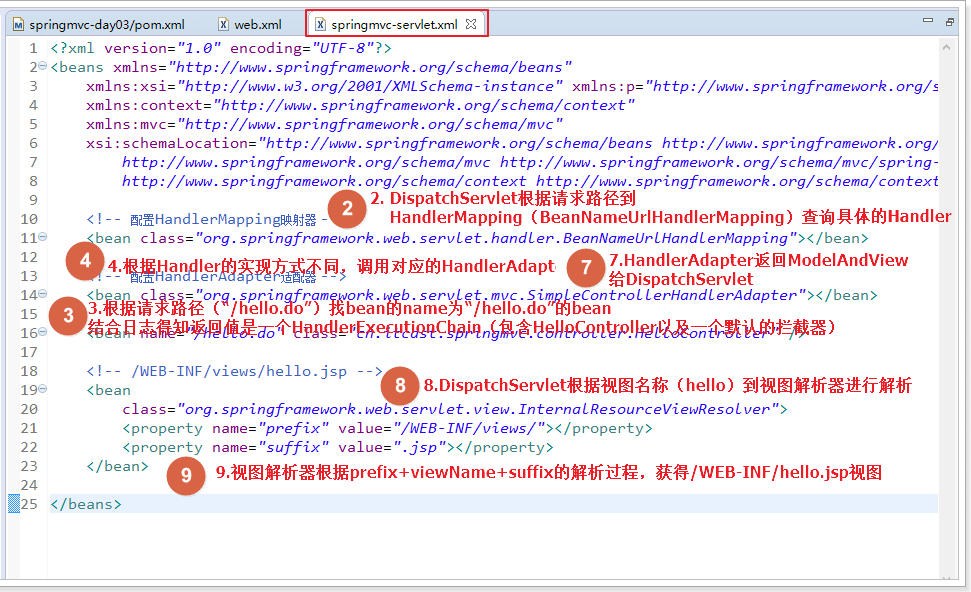
|  |
| --- |
| log4j.rootLogger=DEBUG,A1  log4j.logger.org.apache=DEBUG  log4j.appender.A1=org.apache.log4j.ConsoleAppender  log4j.appender.A1.layout=org.apache.log4j.PatternLayout  log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n |

## 日志打印信息

|  |
| --- |
| 2017-04-13 21:20:48,070 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] DispatcherServlet with name 'springmvc' processing GET request for [/hello.do]  2017-04-13 21:20:48,082 [http-bio-8080-exec-1] [org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping]-[DEBUG] Mapping [/hello.do] to HandlerExecutionChain with handler [cn.itcast.springmvc.controller.HelloController@3c61e75a] and 1 interceptor  2017-04-13 21:20:48,085 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Last-Modified value for [/hello.do] is: -1  2017-04-13 21:20:48,089 [http-bio-8080-exec-1] [org.springframework.beans.factory.support.DefaultListableBeanFactory]-[DEBUG] Invoking afterPropertiesSet() on bean with name 'hello'  2017-04-13 21:20:48,090 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Rendering view [org.springframework.web.servlet.view.JstlView: name 'hello'; URL [/WEB-INF/views/hello.jsp]] in DispatcherServlet with name 'springmvc'  2017-04-13 21:20:48,090 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG] Added model object 'msg' of type [java.lang.String] to request in view with name 'hello'  2017-04-13 21:20:48,107 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG] Forwarding to resource [/WEB-INF/views/hello.jsp] in InternalResourceView 'hello'  m2e SMAP merge is not supported!  SMAP  hello\_jsp.java  JSP  \*S JSP  \*F  + 0 hello.jsp  WEB-INF/views/hello.jsp  \*L  2,8:62  10:70,3  11,2:73  \*E  2017-04-13 21:20:48,233 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Successfully completed request |

## 流程分析





第三步,第四步,第五步:

第三步 有了请求路径 /hello.do 和 对应的HelloController, 而HelloController implements Controller ,

而第四步,SimpleControllerHandlerAdapter适配器中 的方法

@Override

public boolean supports(Object handler) {

return (handler instanceof Controller); --> 判断是否是Controller类或其子类

}

因为HelloController implements Controller ,那么就满足上面方法,那么就会执行这个HelloController 的

@Override

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {

ModelAndView modelAndView = new ModelAndView();

modelAndView.setViewName("hello");

modelAndView.addObject("msg", "iiii");

return modelAndView;

}



## 优化helloworld程序

### web.xml

Load-on-strartup >=0 的时候,就会在tomcat启动的时候就初始化

<0, 的时候,就会在访问的时候

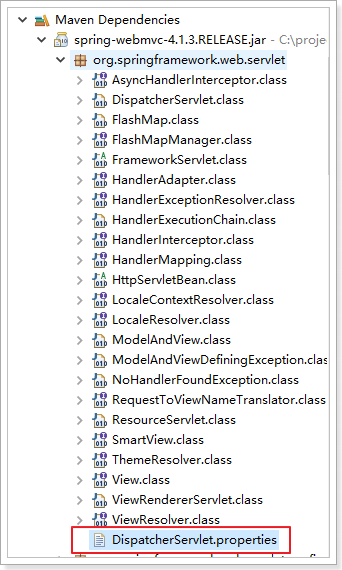


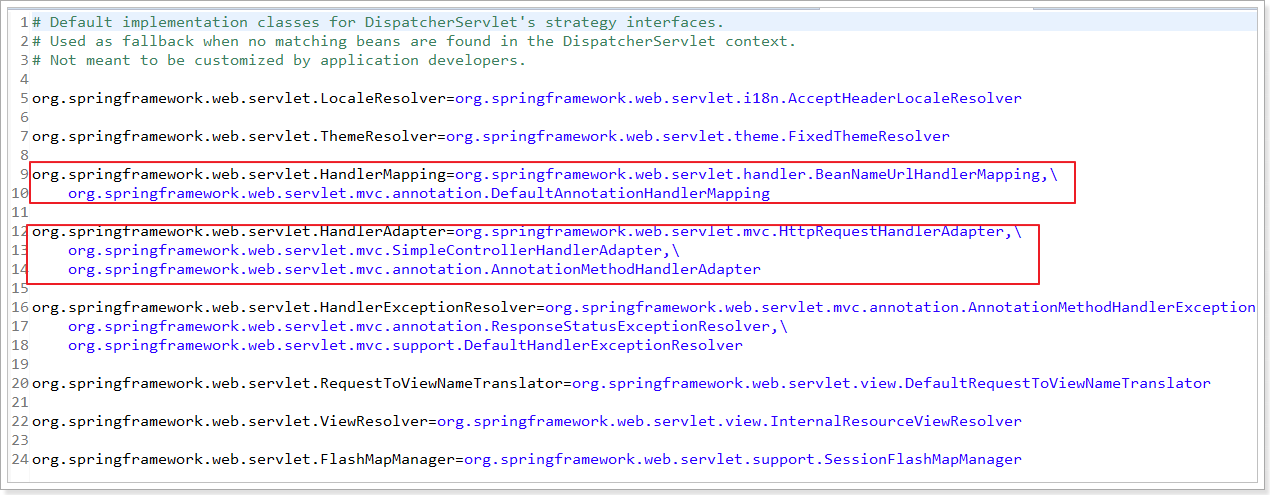
### springmvc-servlet.xml

DispatchServlet.class源码中：



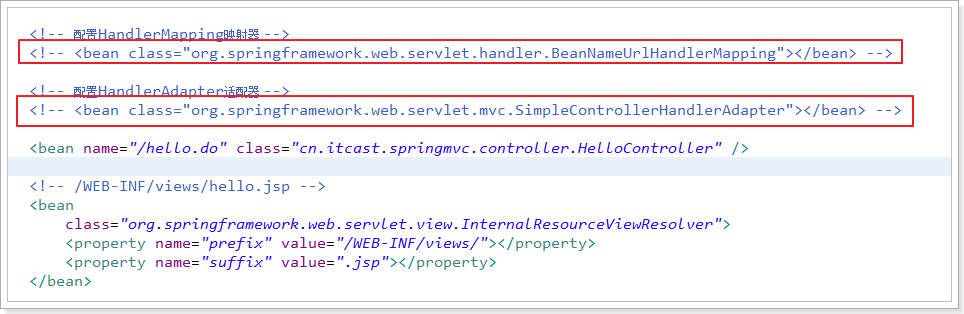
找到DispatchServlet.properties文件：





在这个默认的配置文件中，已经配置了映射器和适配器。

所以在springmvc-servlet.xml文件中可以省略之前配置的映射器和适配器



再次测试：



## helloword的缺点

1）每个类需要都实现Controller接口

2）每个类（Controller）只能完成一个用户请求（或者只能处理一个业务逻辑）

3）每个类（Controller）都要在配置文件里，进行配置

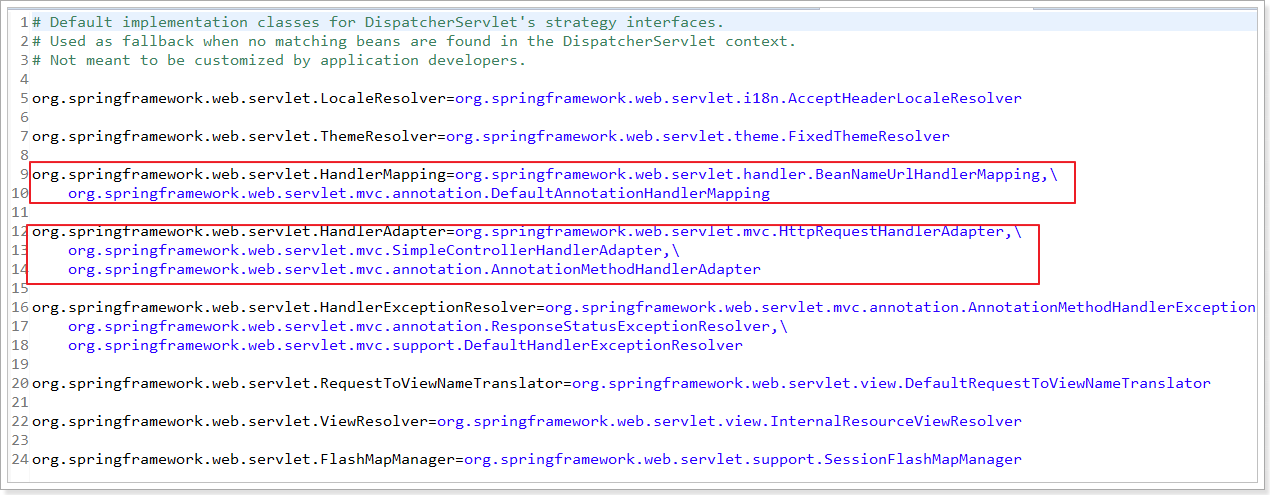
解决方案：

注解程序

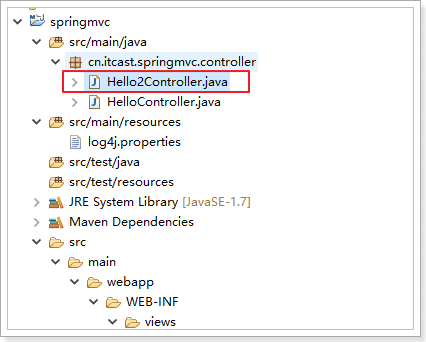
# 注解

## 默认注解配置

在DispatchServlet.properties文件中，已经提供了默认的注解映射器和适配器，所以咱们可以直接书写注解的代码



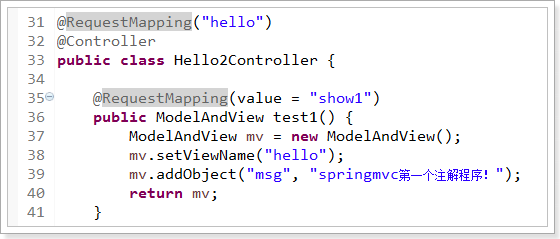
### 创建hello2Controller



内容：

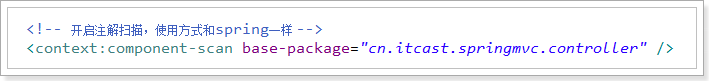
// 什么样的请求能够进入到 这个方法 --> 请求路径 show1.do springmvc优化 直接 show1不需要后缀

@RequestMapping(value="show1")

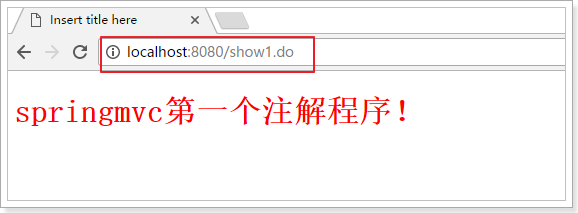


### 配置扫描器

在springmvc-servlet.xml中，开启注解扫描 , **注意配置文件修改后,需要重新重启,否则无效**



### 测试

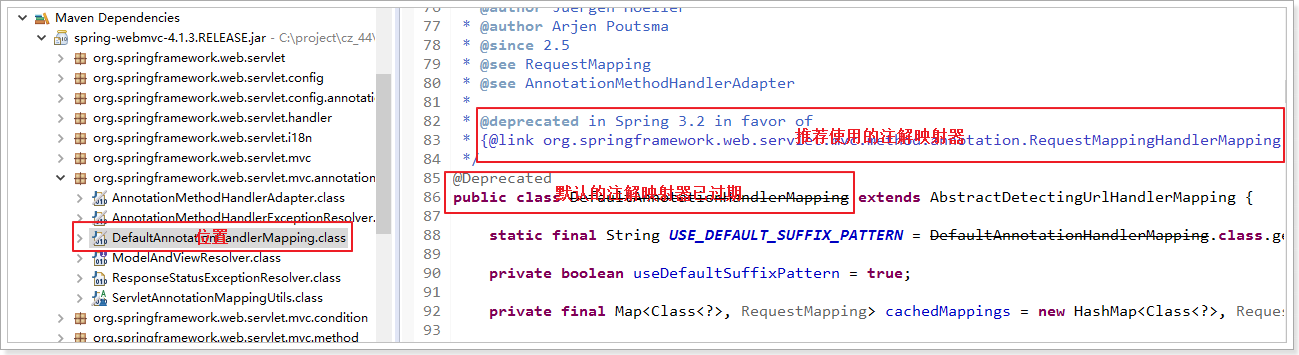


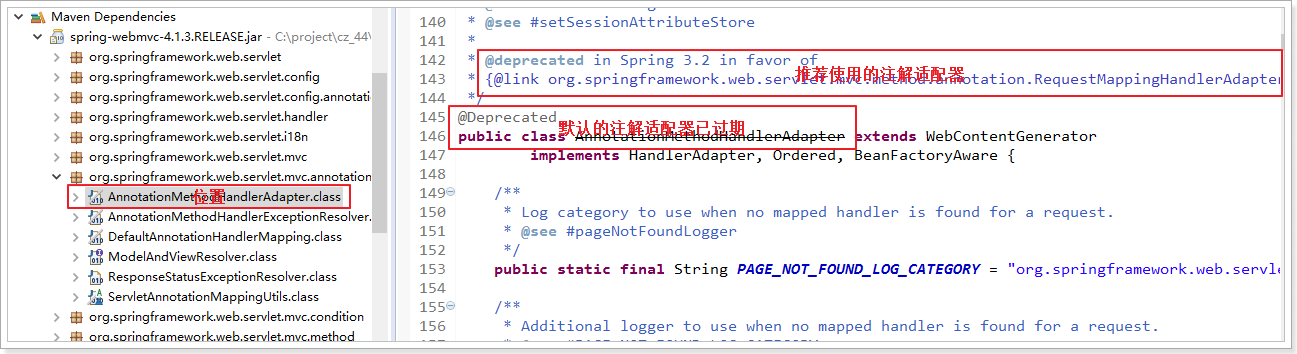
### 日志

输出的是默认配置的映射器，说明这种配置OK

|  |
| --- |
| 2017-04-14 01:10:27,133 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] DispatcherServlet with name 'springmvc' processing GET request for [/hello/show1.do]  2017-04-14 01:10:27,140 [http-bio-8080-exec-1] [org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping]-[DEBUG] Matching patterns for request [/hello/show1.do] are [/hello/show1.\*]  2017-04-14 01:10:27,142 [http-bio-8080-exec-1] [org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping]-[DEBUG] URI Template variables for request [/hello/show1.do] are {}  2017-04-14 01:10:27,144 [http-bio-8080-exec-1] [org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping]-[DEBUG] Mapping [/hello/show1.do] to HandlerExecutionChain with handler [cn.itcast.springmvc.controller.Hello2Controller@68372c9f] and 1 interceptor  2017-04-14 01:10:27,148 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Last-Modified value for [/hello/show1.do] is: -1  2017-04-14 01:10:27,169 [http-bio-8080-exec-1] [org.springframework.web.bind.annotation.support.HandlerMethodInvoker]-[DEBUG] Invoking request handler method: public org.springframework.web.servlet.ModelAndView cn.itcast.springmvc.controller.Hello2Controller.test1()  2017-04-14 01:10:27,174 [http-bio-8080-exec-1] [org.springframework.beans.factory.support.DefaultListableBeanFactory]-[DEBUG] Invoking afterPropertiesSet() on bean with name 'hello'  2017-04-14 01:10:27,174 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Rendering view [org.springframework.web.servlet.view.JstlView: name 'hello'; URL [/WEB-INF/views/hello.jsp]] in DispatcherServlet with name 'springmvc'  2017-04-14 01:10:27,174 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG] Added model object 'msg' of type [java.lang.String] to request in view with name 'hello'  2017-04-14 01:10:27,179 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG] Forwarding to resource [/WEB-INF/views/hello.jsp] in InternalResourceView 'hello'  m2e SMAP merge is not supported!  SMAP  hello\_jsp.java  JSP  \*S JSP  \*F  + 0 hello.jsp  WEB-INF/views/hello.jsp  \*L  2,8:62  10:70,3  11,2:73  \*E  2017-04-14 01:10:27,291 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Successfully completed request |

### 缺点

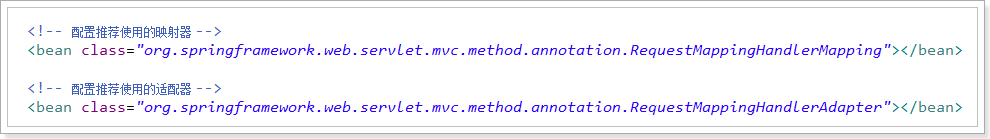




既然默认配置的映射器和适配器都已经过期，并且springmvc也推荐了相应的支持注解的映射器和适配器

## 推荐使用的注解配置

### springmvc-servlet.xml



### 测试



### 日志

输出的是推荐使用的映射器，说明这种配置OK

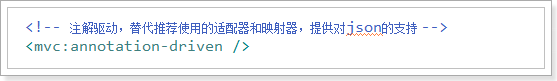
|  |
| --- |
| 2017-04-14 01:06:38,797 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] DispatcherServlet with name 'springmvc' processing GET request for [/hello/show1.do]  2017-04-14 01:06:38,799 [http-bio-8080-exec-1] [org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping]-[DEBUG] Looking up handler method for path /hello/show1.do  2017-04-14 01:06:38,808 [http-bio-8080-exec-1] [org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping]-[DEBUG] Returning handler method [public org.springframework.web.servlet.ModelAndView cn.itcast.springmvc.controller.Hello2Controller.test1()]  2017-04-14 01:06:38,808 [http-bio-8080-exec-1] [org.springframework.beans.factory.support.DefaultListableBeanFactory]-[DEBUG] Returning cached instance of singleton bean 'hello2Controller'  2017-04-14 01:06:38,809 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Last-Modified value for [/hello/show1.do] is: -1  2017-04-14 01:06:38,826 [http-bio-8080-exec-1] [org.springframework.beans.factory.support.DefaultListableBeanFactory]-[DEBUG] Invoking afterPropertiesSet() on bean with name 'hello'  2017-04-14 01:06:38,827 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Rendering view [org.springframework.web.servlet.view.JstlView: name 'hello'; URL [/WEB-INF/views/hello.jsp]] in DispatcherServlet with name 'springmvc'  2017-04-14 01:06:38,827 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG] Added model object 'msg' of type [java.lang.String] to request in view with name 'hello'  2017-04-14 01:06:38,835 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG] Forwarding to resource [/WEB-INF/views/hello.jsp] in InternalResourceView 'hello'  m2e SMAP merge is not supported!  SMAP  hello\_jsp.java  JSP  \*S JSP  \*F  + 0 hello.jsp  WEB-INF/views/hello.jsp  \*L  2,8:62  10:70,3  11,2:73  \*E  2017-04-14 01:06:38,947 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Successfully completed request |

## 最佳方案（注解驱动）

### 注解驱动的配置

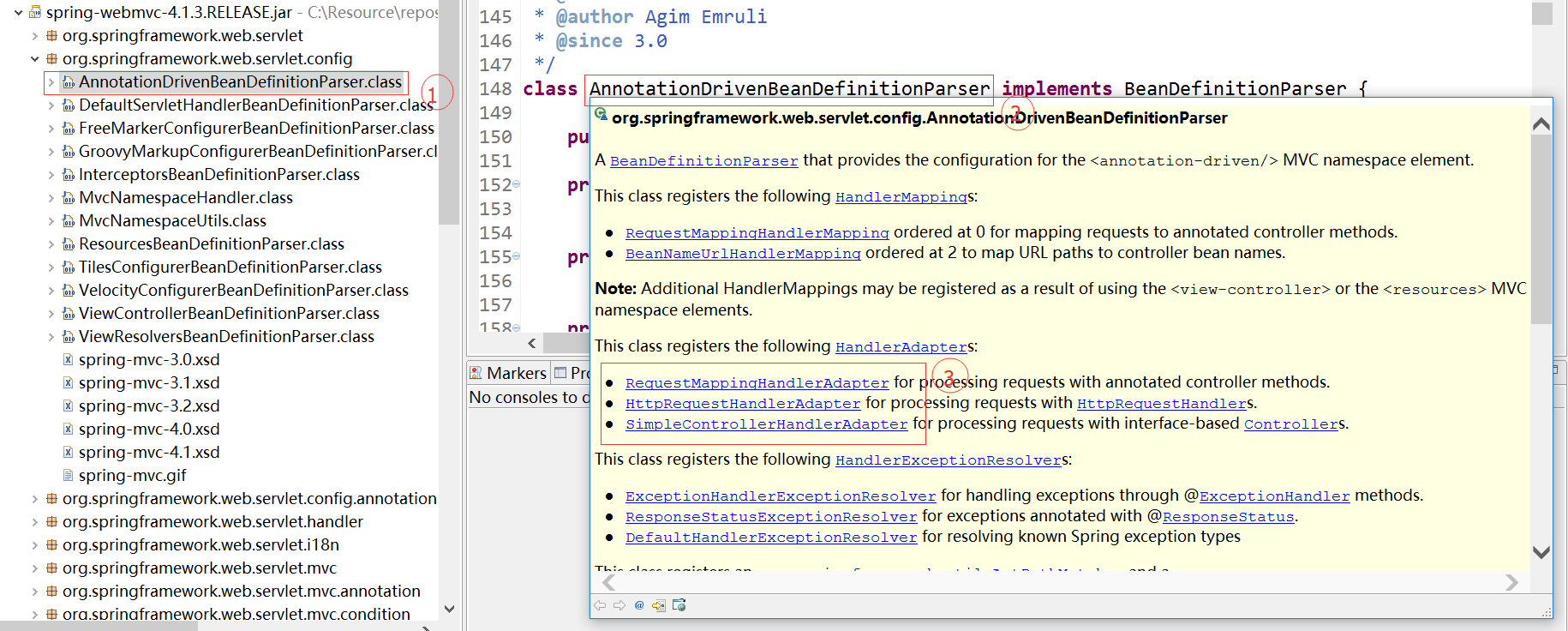
在springmvc-servlet.xml中配置注解驱动

<mvc:annotation-driven />



### 注解驱动的原理

AnnotationDrivenBeanDefinitionParser的注释



## 注解配置最终方案

使用注解驱动后springmvc-servlet.xml这个配置文件：

|  |
| --- |
| <?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-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"*>    <!-- 配置映射器,把bean的name属性作为一个url -->  <!-- <bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" /> -->  <!-- 配置推荐使用的映射器 -->  <!-- <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping" /> -->    <!-- 配置适配器 -->  <!-- <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" /> -->  <!-- 推荐使用的适配器 -->  <!-- <bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter" /> -->    <!-- 配置注解驱动，替代推荐使用的映射器以及适配器，json转换器  annotation-driven 位置org.springframework.web.servlet.config.AnnotationDrivenBeanDefinitionParser  第二个包下第一个类,有一个   * A range of [HttpMessageConverter](eclipse-javadoc:%E2%98%82=springmvc/D:\\/IT\\/Tool\\/Maven\\/repository_20170912\\/repository\\/org\\/springframework\\/spring-webmvc\\/4.1.3.RELEASE\\/spring-webmvc-4.1.3.RELEASE.jar<org.springframework.web.servlet.config(AnnotationDrivenBeanDefinitionParser.class%E2%98%83AnnotationDrivenBeanDefinitionParser%E2%98%82HttpMessageConverter)s depending on what 3rd party libraries are available on the classpath. * 处理json的   -->  <mvc:annotation-driven />    <!-- 开启注解扫描 -->  <context:component-scan base-package=*"cn.itcast.springmvc.controller"*></context:component-scan>    <!-- <bean name="/hello.do" class="cn.itcast.springmvc.controller.HelloController" /> -->    <!-- 配置视图解析器 -->  <!-- Example: prefix="/WEB-INF/jsp/", suffix=".jsp", viewname="test" -> "/WEB-INF/jsp/test.jsp" -->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/views/"*></property>  <property name=*"suffix"* value=*".jsp"*></property>  </bean>    </beans> |
|  |
|  |

目前这些配置已经能够完成springmvc的基本使用，后续还会添加一些高级用法的配置，例如：拦截器、自定义试图、文件上传等

# RequestMapping（映射请求）

标准URL映射

Ant风格的映射 通配符：?:通配一个字符 \*：通配任意多个字符 \*\*：通配任意多个路径

Resful 风格的映射 占位符：{xxx}

限定请求方法的映射：get post put delete

限定参数的映射

## 标准URL映射

@RequestMapping(value=”xxx”)

在springmvc众多Controller以及每个Controller的众多方法中，请求时如何映射到具体的处理方法上

它可以定义在方法上，也可以定义在类上

请求映射的规则：

类上的@RequestMapping的value+方法上的@RequestMapping的value，如果value不以“/”开头，springmvc会自动加上

并且,注解只有一个参数,且是value的时候 , value可以省略 @RequestMapping(”xxx”) --> 请求路径上面需要 /hello

类上的@RequestMapping可省略，这时请求路径就是方法上的@RequestMapping的value

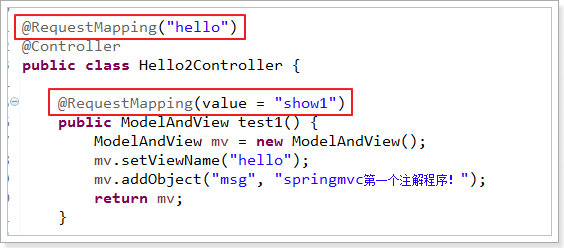
作用在方法上的时候 value做好不要省略 , 应为方法一般经常伴有其他属性的使用,如果value不以“/”开头，springmvc会自动加上 --> 请求路径上面 不 需要 /hello

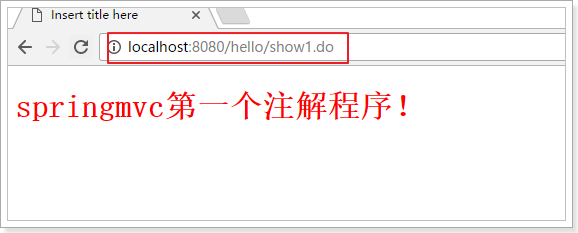
注意: 无论是在类上 还是在方法上面

modelAndView.setViewName("hello"); // 必须要有

可以 new ModelAndView("hello"); // 必须要有

路径不可重复:注意此时的访问路径





## Ant风格的映射（通配符）

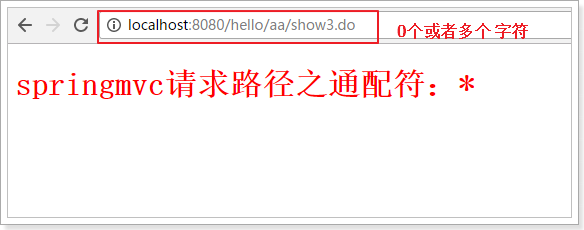
?：通配一个字符

\*：通配0个或者多个字符

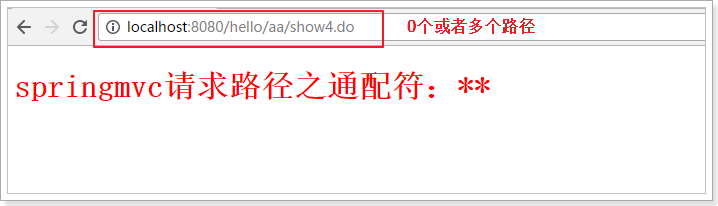
\*\*：通配0个或者多个路径







Localhost:8080/hello//show4.do --> springmvc 请求路径之通配符 : \*\*



思考：如果把test4方法的请求路径，改为”/\*\*/show3”，访问路径：localhost:8080/hello/aa/show3.do会进入test3方法还是test4方法呢？

会进入到 /\*\*/show3的方法 ??? 不应该是精确匹配吗 ??

## 占位符的映射

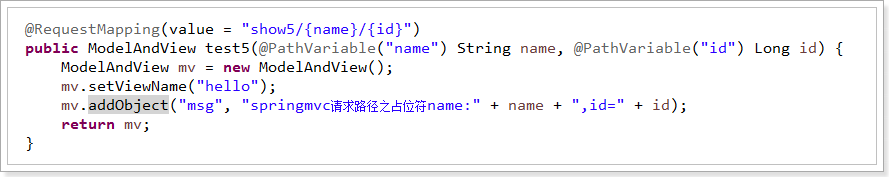
*@RequestMapping(value=“/user/{userId}/{name} ")*

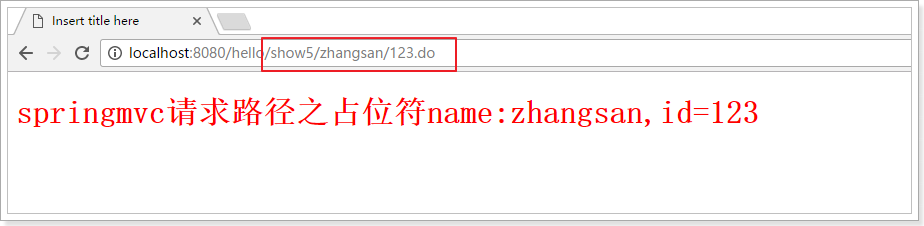
请求URL：<http://localhost:8080/user/1001/zhangsan.do>

这种方式虽然和通配符“\*”类似，却比通配符更加强大，占位符除了可以起到通配的作用，最精要的地方是在于它还可以传递参数。

比如：@PathVariable(“userId”) Long id, @PathVariable(“name”)String name获取对应的参数。

注意：@PathVariable(“key”)中的key必须和对应的占位符中的参数名一致，而方法形参的参数名可任意取



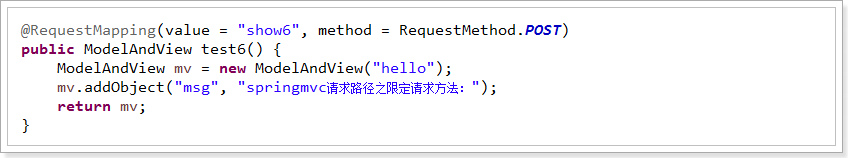


如果传递的参数类型和接受参数的形参类型不一致，则会自动转换，如果转换出错（例如：id传了abc字符串，方法形参使用Long来接受参数），则会报400错误（参数列表错误）。

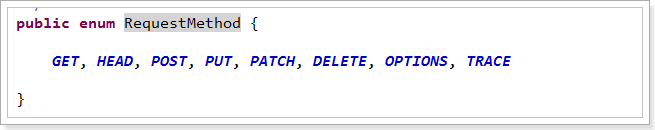


## 限定请求方法的映射

@RequestMapping(value=””, method=RequestMethod.POST)



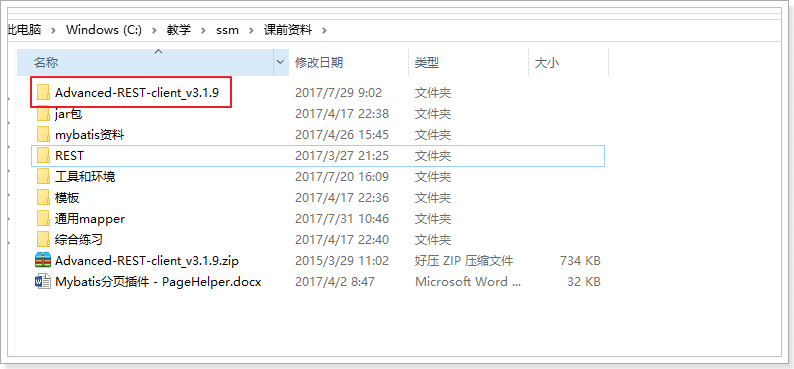
用到了框架提供的RequestMethod枚举类，源代码截图：



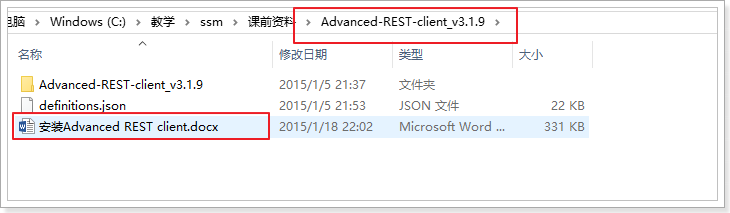
此时show6限定请求方法为POST请求，如果通过浏览器地址栏输入请求路径（也就是GET请求），结果：



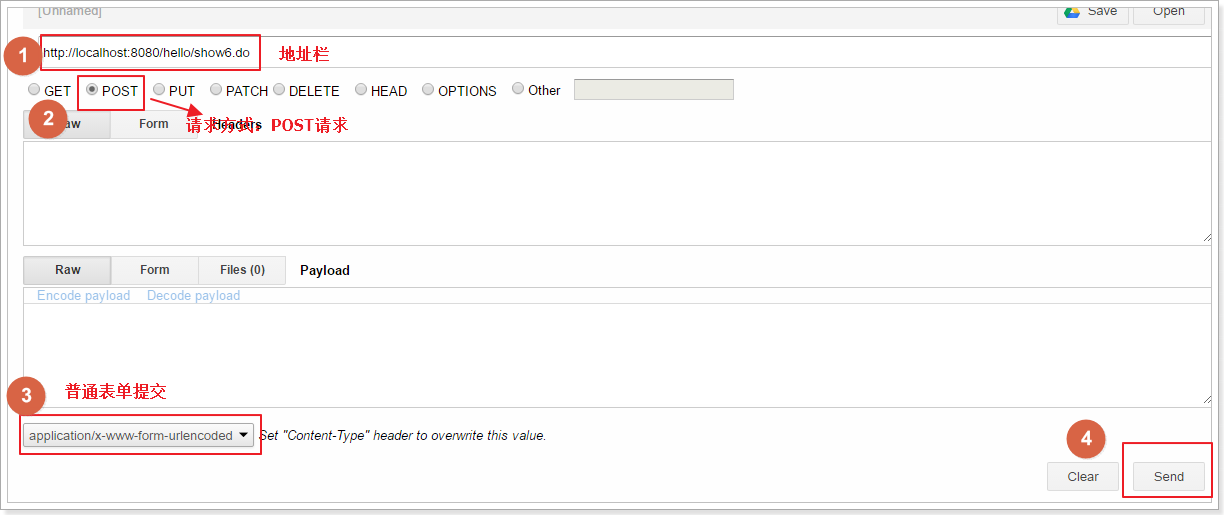
地址栏无法模拟POST请求，需要使用浏览器插件，模拟POST请求，下面为chrome浏览器的模拟插件，参见课前资料



安装过程参见教程：

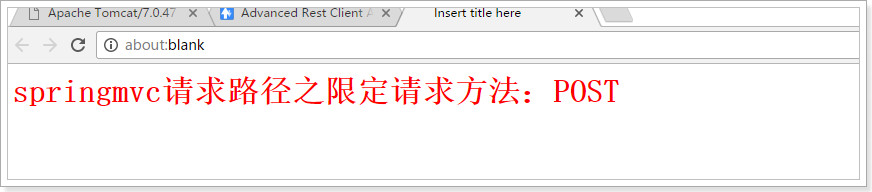


安装完成后的使用：





点击步骤7，可查看效果如下：

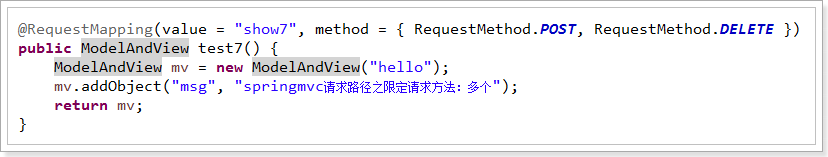


限定多种请求方法

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

可以单独使用: method=RequestMethod.POST

多个使用 :method={RequestMethod.POST,RequestMethod.GET}



## 限定请求参数的映射

@RequestMapping(value=””,params=””)

params=”userId”：请求参数中必须带有userId

params=”!userId”：请求参数中不能包含userId , 没有参数也正确

params=”userId=1”：请求参数中userId必须为1

params=”userId!=1”：请求参数中userId必须不为1，参数中可以不包含userId

params={“userId”, ”name”}：请求参数中必须有userId，name参数

@RequestMapping(value="show8", params="id")

**public** ModelAndView test8(){

ModelAndView mv = **new** ModelAndView("hello");

mv.addObject("msg", "springmvc的映射之限定请求参数，id");

**return** mv;

}

@RequestMapping(value="show9", params="!id")

**public** ModelAndView test9(){

ModelAndView mv = **new** ModelAndView("hello");

mv.addObject("msg", "springmvc的映射之限定请求参数，!id");

**return** mv;

}

@RequestMapping(value="show10", params="id=1")

**public** ModelAndView test10(){

ModelAndView mv = **new** ModelAndView("hello");

mv.addObject("msg", "springmvc的映射之限定请求参数，id=1");

**return** mv;

}

@RequestMapping(value="show11", params="id!=1")

**public** ModelAndView test11(){

ModelAndView mv = **new** ModelAndView("hello");

mv.addObject("msg", "springmvc的映射之限定请求参数，id!=1");

**return** mv;

}

@RequestMapping(value="show12", params={"id","name"})

**public** ModelAndView test12(){

ModelAndView mv = **new** ModelAndView("hello");

mv.addObject("msg", "springmvc的映射之限定请求参数，id,name");

**return** mv;

}

总结：

Value：标准 ant风格（通配符：？ \* \*\*） Rest风格（占位符）

Method：限定请求方法 put post delete get RequestMethod

Params：限定请求参数 params={，}

# 接收数据及数据绑定

1. 接收servlet的内置对象 HttpServletRequest HttpServletResponse HttpSession
2. 接收占位符请求路径中的参数 @PathVariable(参数)
3. 接收普通的请求参数 @RequestParam
4. 获取cookie参数 request.getCookies @CookieValue
5. 基本数据类型的绑定
6. Pojo对象的绑定
7. 集合的绑定

请求参数—>方法形参，方法形参没有顺序。

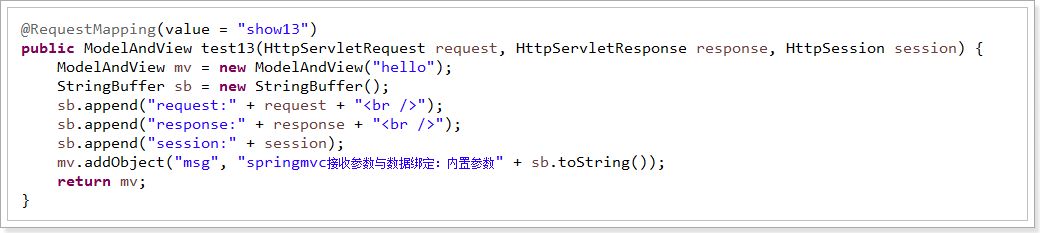
可以和 jsp 进行数据交流的:

1. Model model.addAttribute("msg", "获取Cookie的值：JESSIONID="+cookie.getValue());
2. ModelMap modelMap.addAttribute("msg", "获取Cookie的值：JESSIONID="+cookie.getValue());
3. ModelAndView modelAndView.addObject("msg", "映射之通配符：\*");

## 接收servlet的内置对象

1. 并且这些对象的接收非常简单，只需要在方法形参中有该对象就能接收，不需要任何配置

2. 方法参数和顺序无关



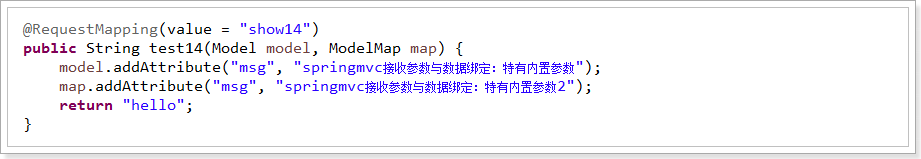
**主流写法:**

特有内置对象:

Model

ModelMap

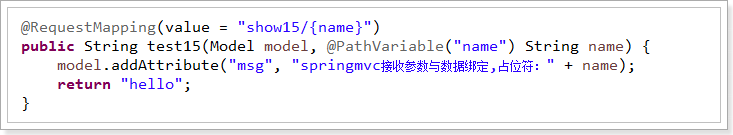
当方法返回是String的时候,默认是 ModelAndView的 视图名称 , 所以可以直接 return “hello”;



## 接收请求路径中的占位符

@PathVariable(value=”id”)获取占位符中的参数

注意：(value=”id”)不能省



小猫腻:

@RequestMapping(value="{name}/{id}/show15")

public ModelAndView test15(@PathVariable("name")String name, @PathVariable("id")Long id){

ModelAndView mv = new ModelAndView();

mv.setViewName("hello");

mv.addObject("msg", "REST风格的映射,id="+id+",name="+name);

return mv;

}

只要参数名字和 value的名字相同 那么可以省略掉 --> public ModelAndView test15(@PathVariable( )String name, @PathVariable( )Long id)

原因: window -> preferences -> java -> compiler -> Classfile Generation -> Add variable attributes to xxxxxxxxx(编译的时候会将参数名称编译到class里面) 勾掉

勾掉后,就会用默认的参数名称替代 比如: args0 , args1...

那么在build --> url访问 --> error 500错误 --> 所以 : 最好不省略

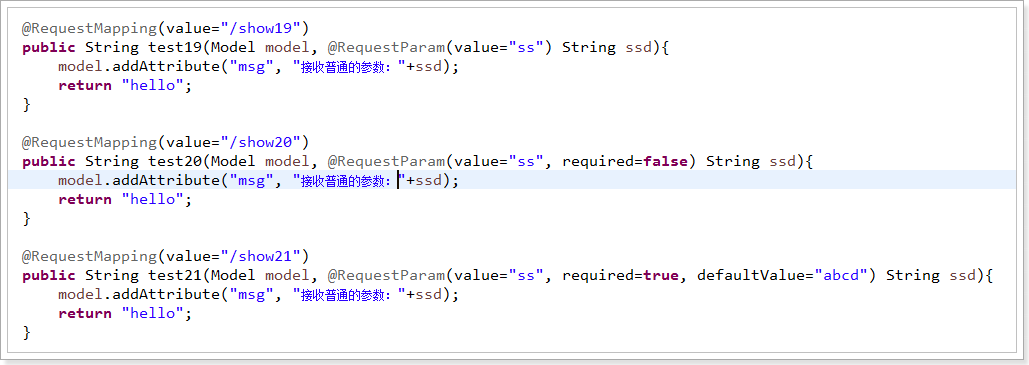
## 接收普通的请求参数

1. form表单
2. Url的 ?后面的key=value格式参数的
3. 基本数据类型

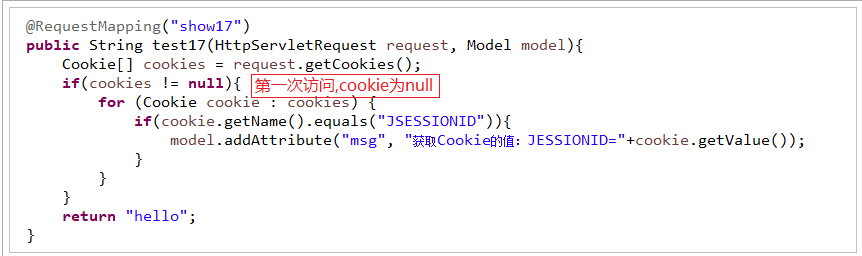
@RequestParam(value=””, required=true/false, defaultValue=””)

1. value：参数名
2. required：是否必须，默认为true，标示请求参数中必须包含该参数，如果不包含则抛出异常
3. defaultValue：默认参数值，如果设置了该值，required=true将失效(即使手动这只也没用)，自动为false，如果请求中不包含该参数则使用默认值。

注意: 不要和 @RequestMapping(value=””,params=””) 中的 param (限定请求参数)起冲突

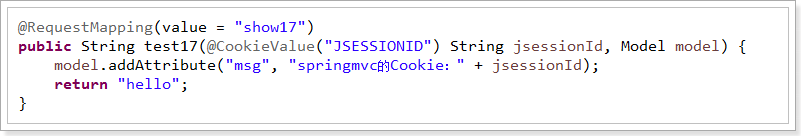


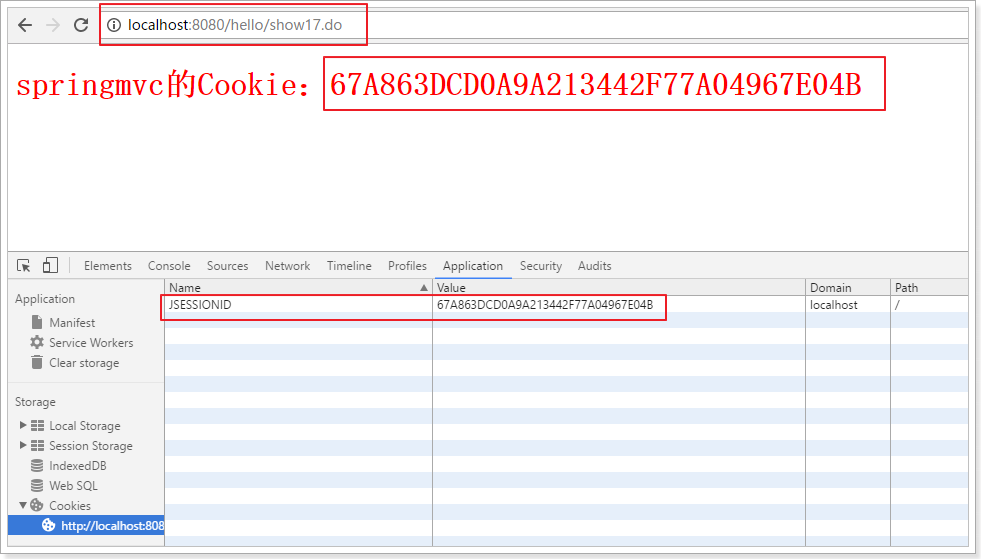
## 获取cookie



@CookieValue使用方法同@RequestParam

@CookieValue("JSESSIONID")String jsessionId --> JSESSIONID 指定要获取的是哪一个cookie值

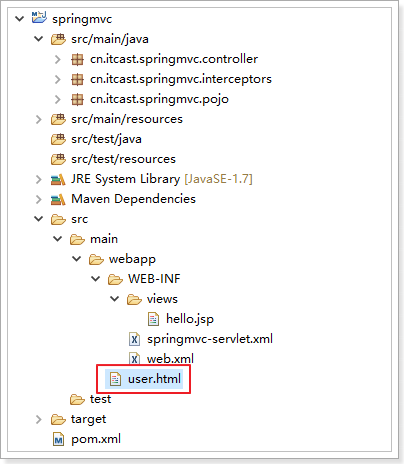




## 基本数据类型的绑定

字符串、整型、浮点型、布尔型、数组。

在webapp目录下，创建user.html表单：

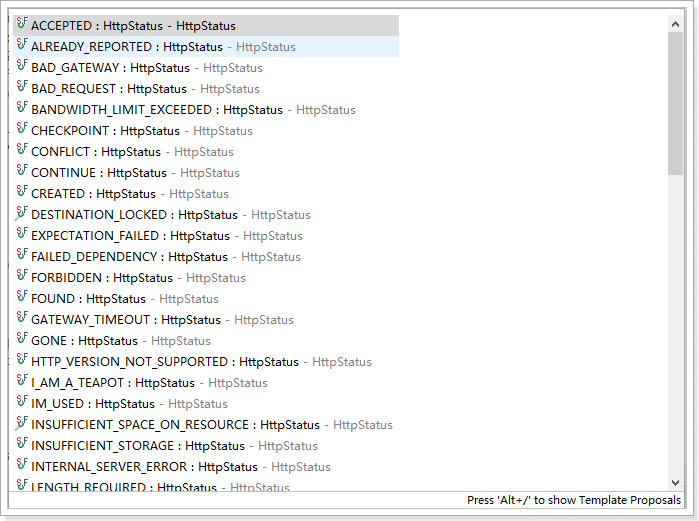


|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"UTF-8"*>  <title>Insert title here</title>  </head>  <body>  <form action=*"/hello/show19.do"* >  name:<input type=*"text"* name=*"name"* /><br />  age:<input type=*"text"* name=*"age"* /><br />  isMarry:<input type=*"checkbox"* name=*"isMarry"*/><br />  income:<input type=*"text"* name=*"income"* /><br />  interests:<input type=*"checkbox"* name=*"interests"* value=*"bb"* />basketball  <input type=*"checkbox"* name=*"interests"* value=*"fb"* />football  <input type=*"checkbox"* name=*"interests"* value=*"vb"* />vollyball<br />  <input type=*"submit"* value=*"提交"* />  </form>  </body>  </html> |

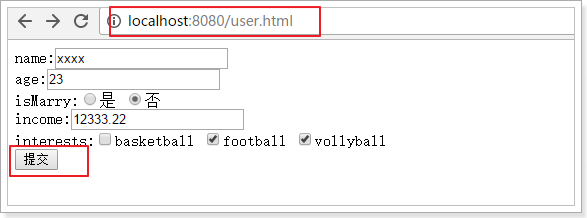
可以通过localhost:8080/user.html访问



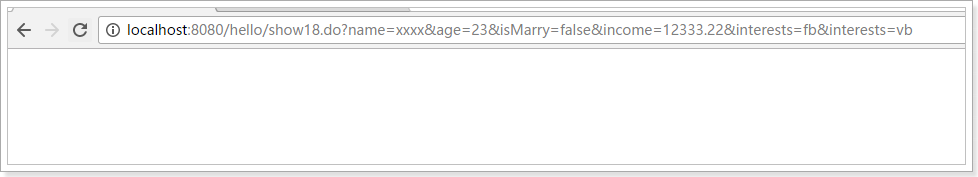
相应状态HttpStatus是一个枚举类：



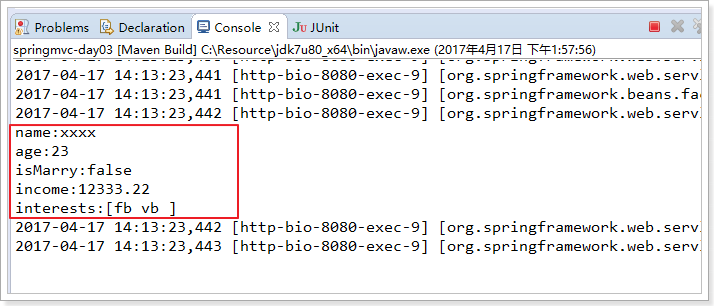
测试：



由于Controller方法没有任何返回值，所以浏览器没有任何信息：



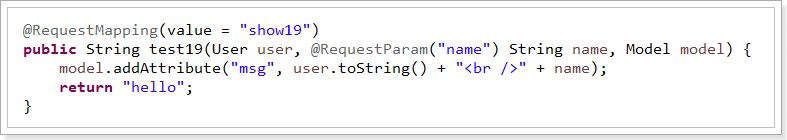
Log日志



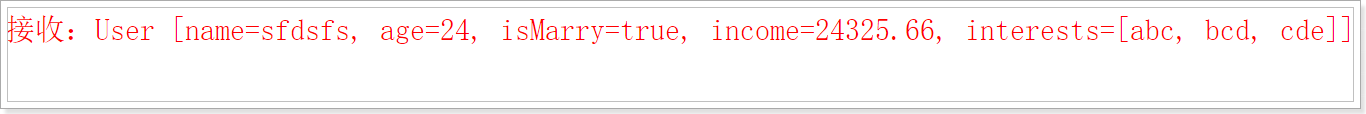
## Pojo对象的绑定

SpringMVC会将请求参数名和POJO实体中的属性名(set方法)进行自动匹配，如果名称一致，将把值填充到对象属性中，并且支持级联（例如：user.dept.id）。

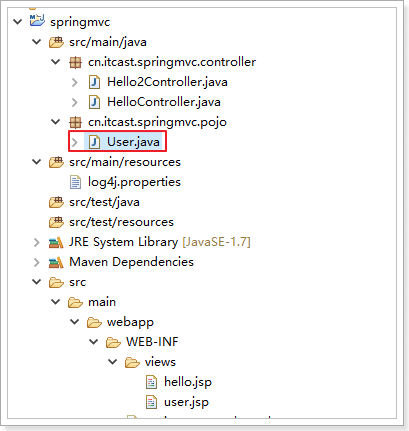
Controller方法：



测试：



User类：



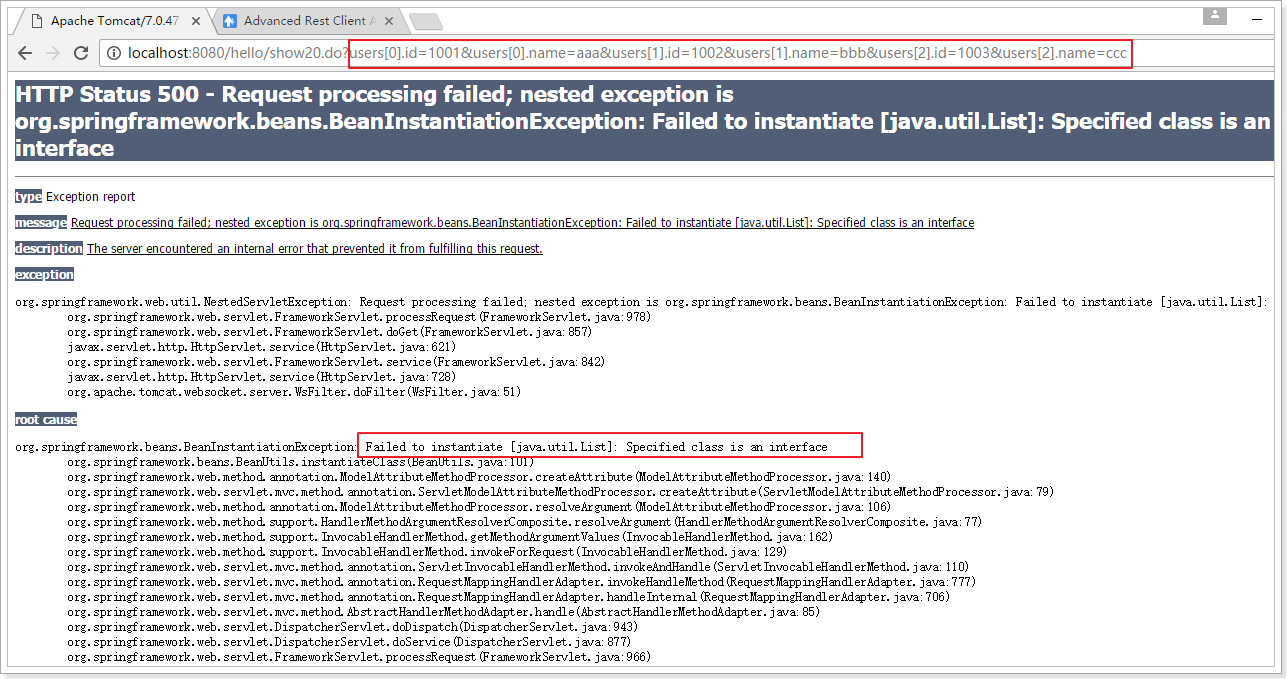
User内容：

|  |
| --- |
| **package** cn.itcast.springmvc.pojo;  **import** java.util.Arrays;  **public** **class** User {  **private** String name;  **private** String userName;  **private** Integer id;  **private** Integer age;  **private** Double income;  **private** Boolean isMarry;  **private** String[] interests;    **public** String getUserName() {  **return** userName;  }  **public** **void** setUserName(String userName) {  **this**.userName = userName;  }  **public** Integer getId() {  **return** id;  }  **public** **void** setId(Integer id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  **public** Double getIncome() {  **return** income;  }  **public** **void** setIncome(Double income) {  **this**.income = income;  }  **public** Boolean getIsMarry() {  **return** isMarry;  }  **public** **void** setIsMarry(Boolean isMarry) {  **this**.isMarry = isMarry;  }  **public** String[] getInterests() {  **return** interests;  }  **public** **void** setInterests(String[] interests) {  **this**.interests = interests;  }  @Override  **public** String toString() {  **return** "User [name=" + name + ", age=" + age + ", income=" + income + ", isMarry=" + isMarry + ", interests="  + Arrays.*toString*(interests) + "]";  }    } |

## 集合的绑定

反例：



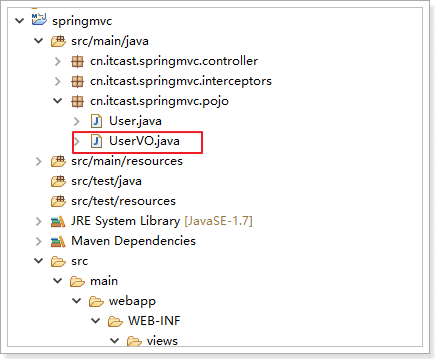


如果方法需要接受的list集合，不能够直接在方法中形参中使用List<User>

List的绑定，需要将List对象包装到一个类中才能绑定

要求：表单中input标签的name的值和集合中元素的属性名一致。

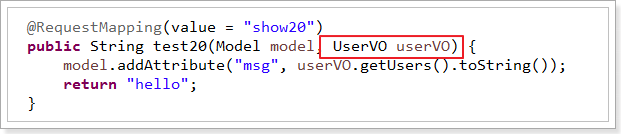
UserVO：



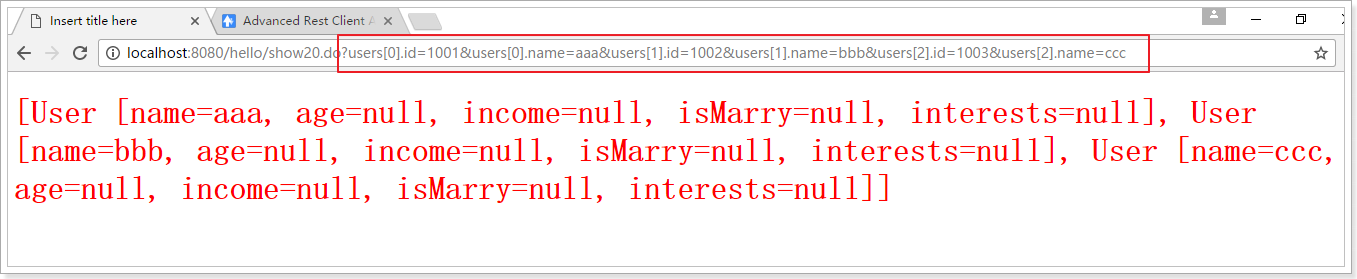
内容：

|  |
| --- |
| **package** cn.itcast.springmvc.pojo;  **import** java.util.List;  **public** **class** UserVO {  **private** List<User> users;  **public** List<User> getUsers() {  **return** users;  }  **public** **void** setUsers(List<User> users) {  **this**.users = users;  }  } |

Controller方法：



效果：



内置对象：HttpServletRequest HttpServletResponse HttpSession Model

占位参数：@PathVariable（“参数名”）

普通参数：@RequestParam（value=“参数名”，required=true， defaultValue=“默认值”）

Cookie：@CookieValue request.getCookies

基本数据类型：int string float Boolean string[]

Pojo接收：

集合：泛型是包装对象，需要另外一个包装对象；

List<User> --> 需要另外一个包装对象 包装List

# jstl标签的使用

JSTL：标准标签库

JSP标准标签库（JSTL）是一个JSP标签集合，它封装了JSP应用的通用核心功能。

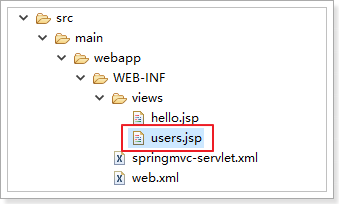
JSTL支持通用的、结构化的任务，比如迭代，条件判断，XML文档操作，国际化标签，SQL标签。

## 导入jstl依赖（pom.xml）

参考父工程：



## 静态页面



内容

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>JSTL Demo</title>  </head>  <body>  <table cellpadding=0 cellspacing=0 border="1">  <thead>  <tr>  <th>ID</th>  <th>UserName</th>  <th>Name</th>  <th>Age</th>  </tr>  </thead>  <tbody>  <tr>  <td>1001</td>  <td>zhangsan</td>  <td>张三</td>  <td>18</td>  </tr>  <tr>  <td>1002</td>  <td>lisi</td>  <td>李四</td>  <td>19</td>  </tr>  <tr>  <td>1003</td>  <td>wangwu</td>  <td>王五</td>  <td>20</td>  </tr>  <tr>  <td>1004</td>  <td>gary</td>  <td>张三</td>  <td>18</td>  </tr>  <tr>  <td>1005</td>  <td>gary</td>  <td>张三</td>  <td>18</td>  </tr>  <tr>  <td>1006</td>  <td>gary</td>  <td>张三</td>  <td>18</td>  </tr>  <tr>  <td>1007</td>  <td>gary</td>  <td>张三</td>  <td>18</td>  </tr>  <tr>  <td>1008</td>  <td>gary</td>  <td>张三</td>  <td>18</td>  </tr>  </tbody>  </table>  </body>  </html> |

## jsp头信息

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

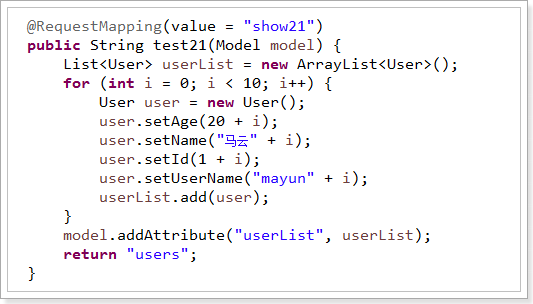
## 引入核心标签库

|  |
| --- |
| <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"*%> |

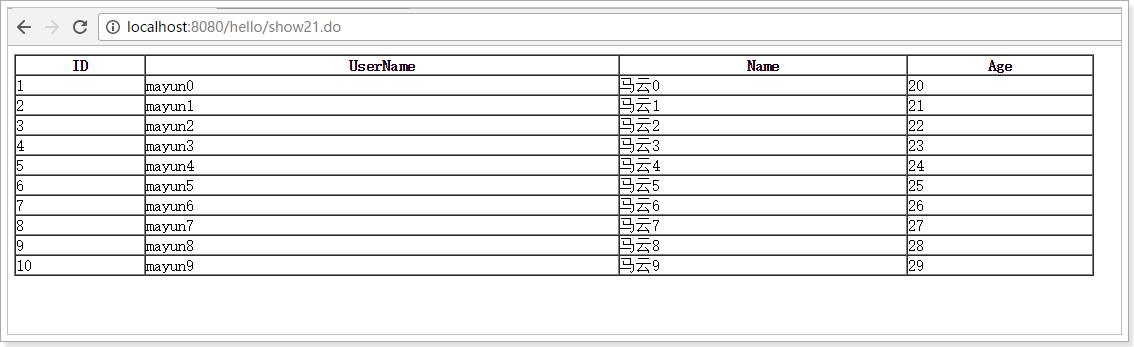
## 使用<c:foreach>标签



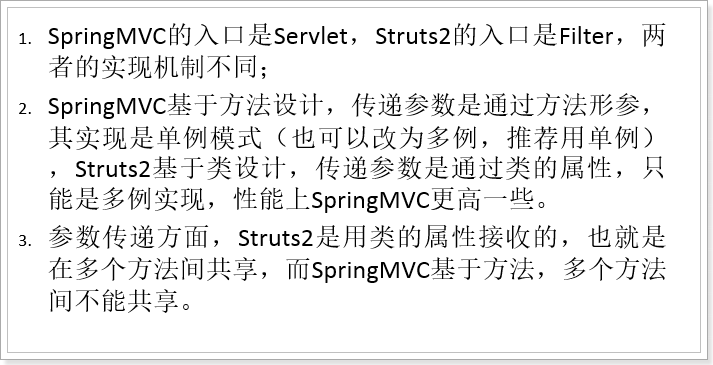
## Controller方法



## 效果



# Springmvc和struts2比较 : 视频 day02 - 10



单例：在使用的过程中，一直是一个对象。只被创建一次

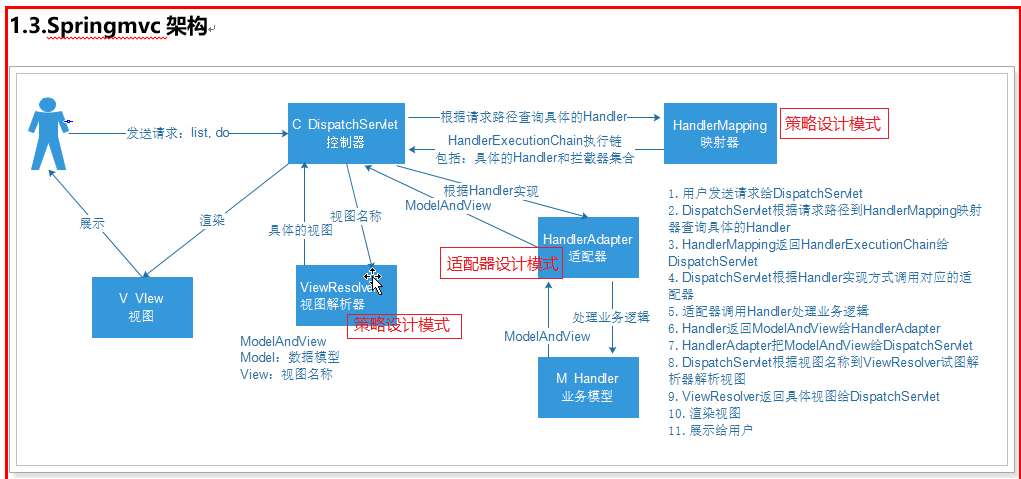
多例：会被创建多次。

单例模式是spring推荐的配置，它在高并发下能极大的节省资源，提高服务抗压能力。spring IOC的bean管理器是“绝对的线程安全”。

设计模式简介:







# JSON

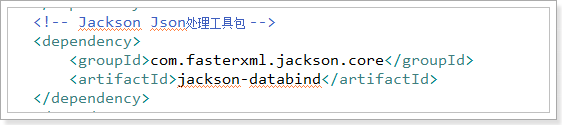
在实际开发过程中，json是最为常见的一种方式，所以springmvc提供了一种更为简便的方式传递数据。

@ResponseBody 是把Controller方法返回值转化为JSON，称为序列化

@RequestBody 是把接收到的JSON数据转化为Pojo对象，称为反序列化

## 引入依赖

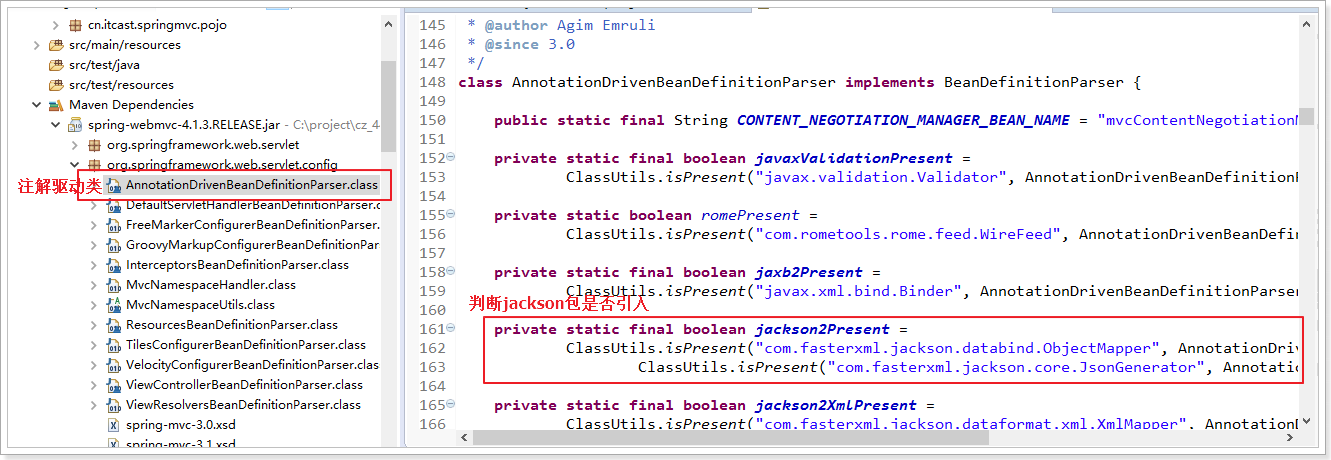
在Pom.xml中引入jackson依赖，参考父工程：

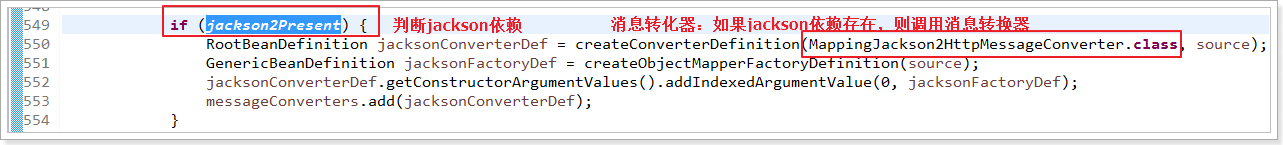


## 原理

Json的转化是由注解驱动完成的。

注解驱动会判断是否引入了jackson依赖，并决定是否加载json转化的消息转化器



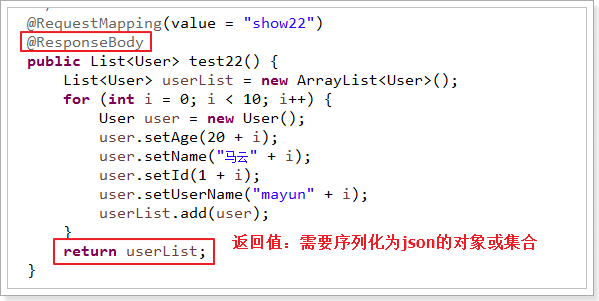


## @ResponseBody : 序列化

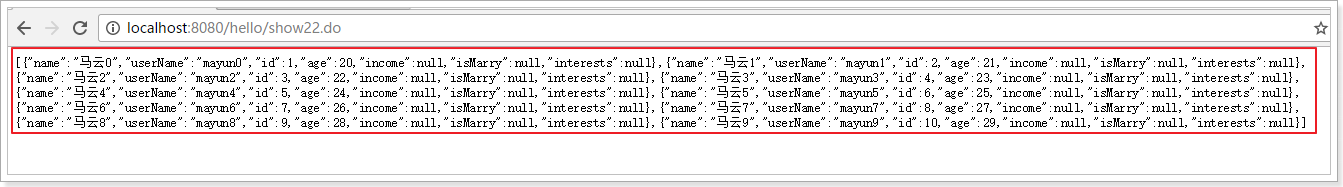
当一个处理请求的方法标记为@ResponseBody时，表示该方法需要输出其他视图（json、xml），springmvc通过默认的json转化器转化输出

Json本身就是一个视图,所以可以直接 return ;

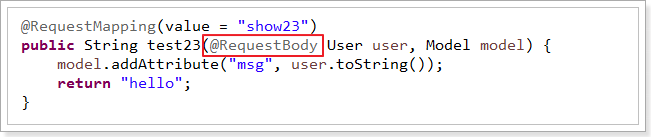
Controller方法：

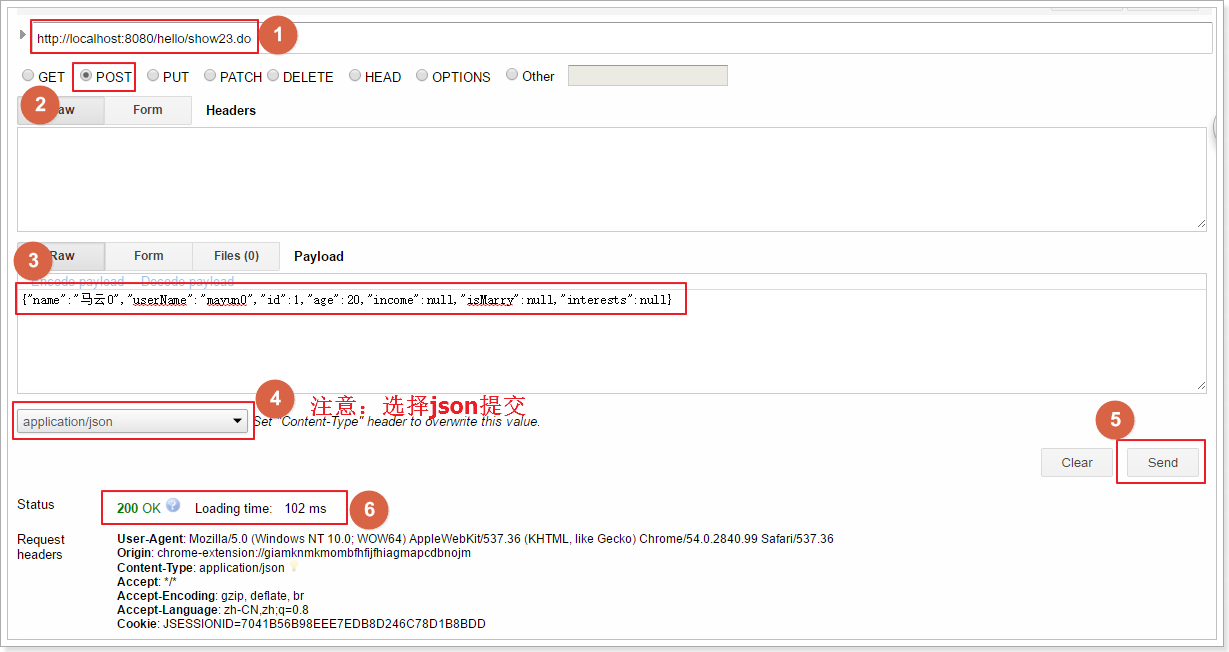


测试：



## @RequestBody :反序列化





注意:

如果是 用String 接收 , 那么也成功

@RequestMapping(value="show24")

public String test26(@RequestBody() String user, Model model){

model.addAttribute("msg", user.toString());

return "hello";

}

说明: jackson 做了两件事情

1. 接收Json数据
2. 将json数据反序列化给一个对象

如果是User对象接收 那么 两件事情都做了

如果是 String 接收,那么只做一件事

# 文件上传 :

SpringMVC的文件上传，底层也是使用的Apache的Commons-fileupload

## 添加依赖

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

## 文件上传解析器

在springmvc-servlet.xml中配置

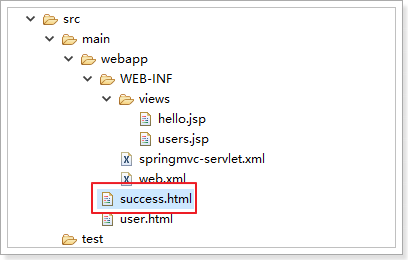
这个文件上传的解析器是属于Spring的不是属于springmvc Spring-web包下的

|  |
| --- |
| <!-- 定义文件上传解析器 -->  <!-- id 一定要配置 -->  <bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>  <!-- 设定默认编码 -->  <property name=*"defaultEncoding"* value=*"UTF-8"*></property>  <!-- 设定文件上传的最大值5MB，5\*1024\*1024 单位是b -->  <!-- 如果是批量上传,那么批量大小不能超过5M  如果是单个,那么单个的大小不能超过5M  -->  <property name=*"maxUploadSize"* value=*"5242880"*></property>  </bean> |

## Controller方法

|  |
| --- |
| @RequestMapping(value="show25")  **public** String test25(Model model, @RequestParam("file")MultipartFile file)  **throws** IllegalStateException, IOException{  **if** (file!=**null**) {  file.transferTo(**new** File("c:\\tmp\\" + file.getOriginalFilename()));  }  **return** "redirect:/success.html";  } |

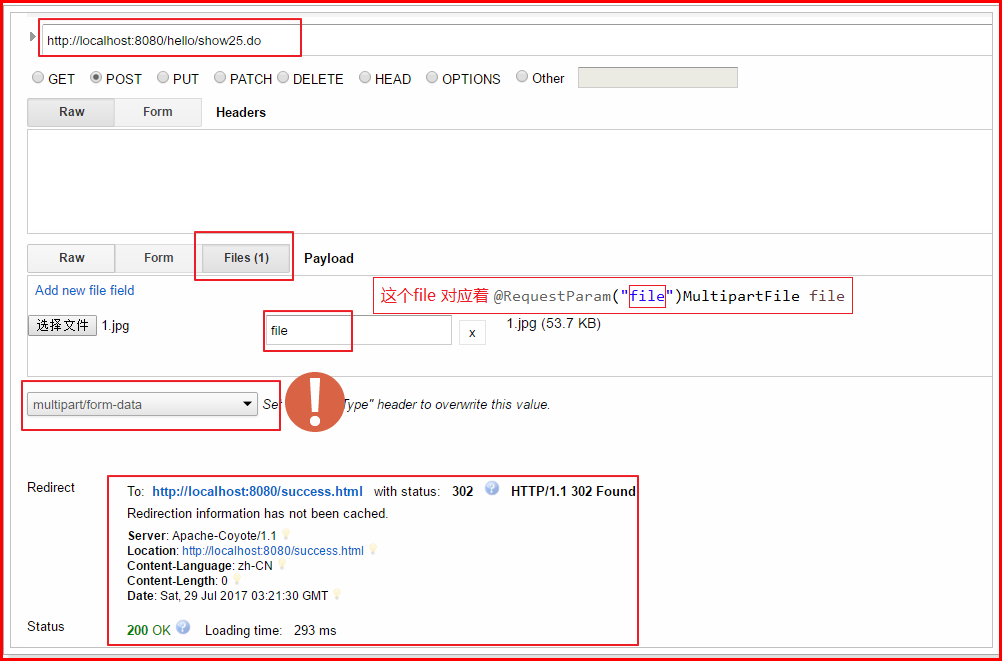
## 成功页面



在webapp目录下创建success.html文件

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset=*"UTF-8"*>  <title>Insert title here</title>  </head>  <body>  <span style="font-size:*30px*;color:*red*">上传成功</span>  </body>  </html> |

## 效果



## 上传多个



# 转发及重定向（forward、redirect）

返回值为字符串时，默认为视图名称。当返回值字符串是以”forward:”或者”redirect:”开头，则会被认为是转发或者重定向。

使用方式如下：

转发：forward:/hello/show.do --> 绝对路径 / 或者forward:show.do --> 相对路径 没有 /

重定向：redirect:/hello/show.do或者redirect:show.do

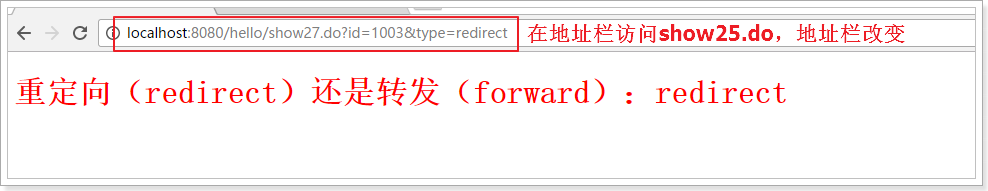
注意：后面必须跟上URL路径而非视图名



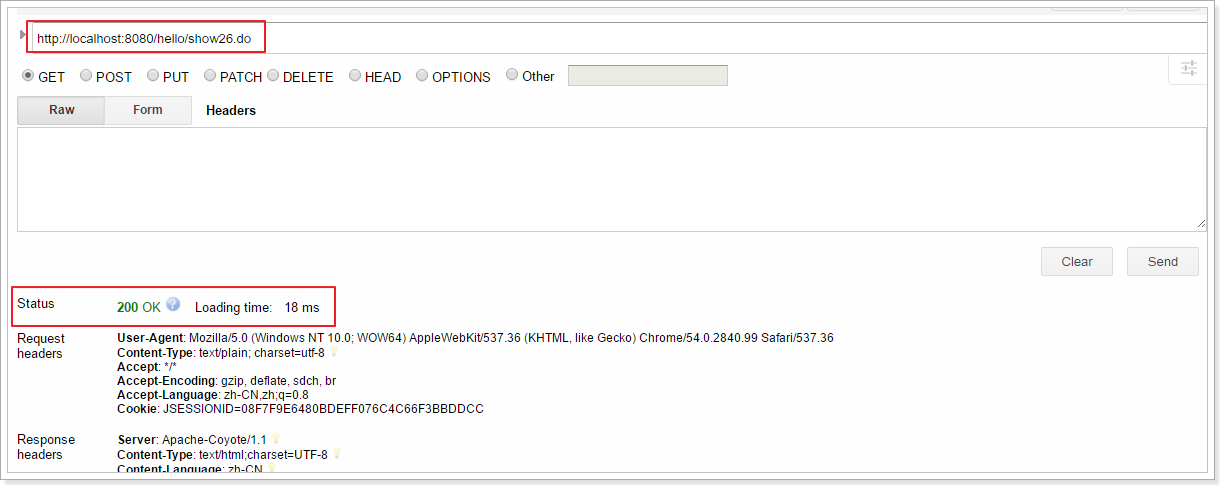
在RestClient测试重定向（show25.do）：



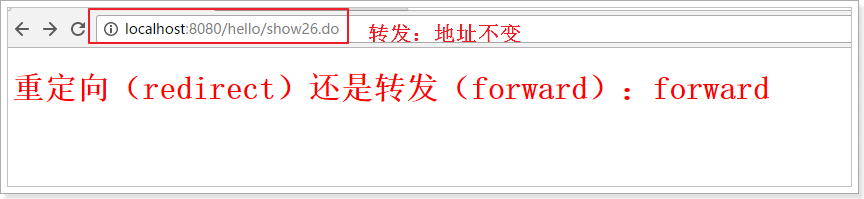
在浏览器地址栏测试重定向（show25.do）：



在RestClient测试转发（show26.do）：



在浏览器中测试转发（show26.do）：



# 拦截器

HandlerExecutionChain是一个执行链，当请求到达DispatchServlet时，DispatchServlet根据请求路径到HandlerMapping查询具体的Handler，

从HandlerMapping返回给DispatcherServlet，其中包含了一个具体的Handler对象和Interceptors（拦截器集合）。

如何自定义拦截器：

springmvc的拦截器接口（HandlerInterceptor）定义了三个方法：

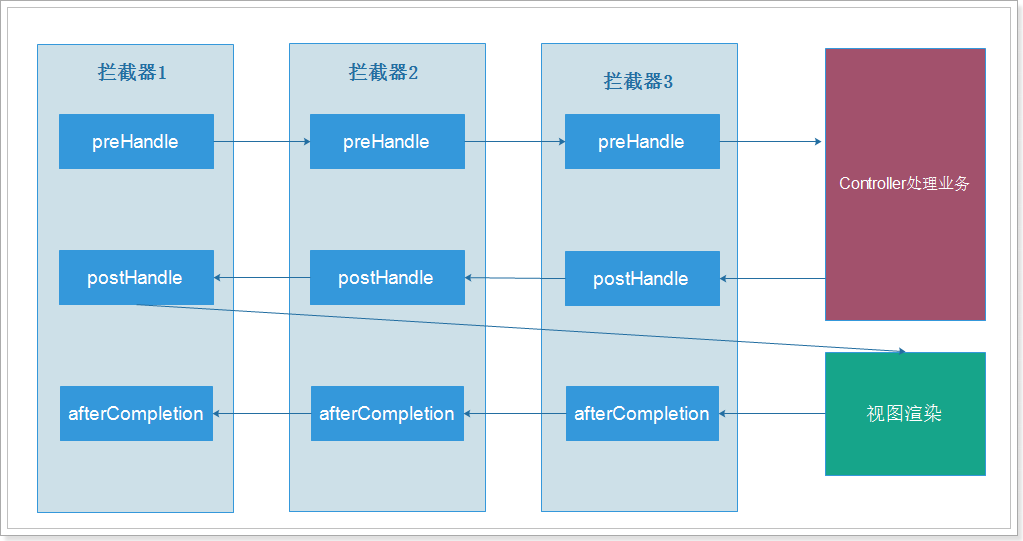
1. preHandle调用Handler之前执行，称为前置方法

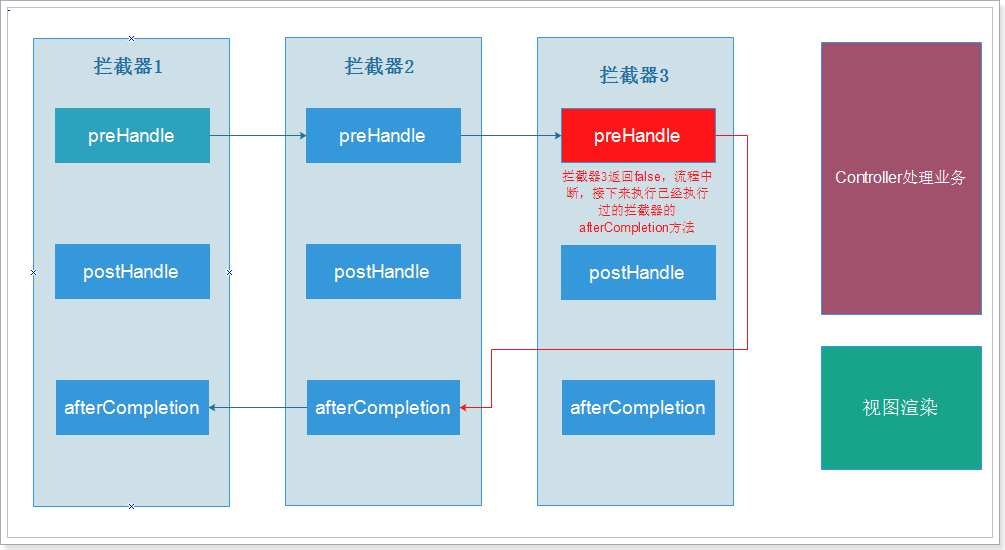
返回值：true表示放行，后续业务逻辑继续执行

false表示被拦截，后续业务逻辑不再执行，但之前返回true的拦截器的完成方法会倒叙执行

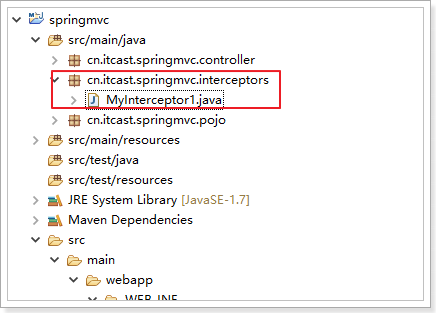
1. postHandle调用Handler之后执行，称为后置方法
2. afterCompletion视图渲染完成之后执行

## 拦截器的执行过程





## 编写自定义拦截器



MyInterceptor1内容：

**public** **class** MyInterceptor1 **implements** HandlerInterceptor {

/\*\*

\* 前置方法，在Handler方法执行之前执行，顺序执行

\* 返回值，返回true拦截器放行 false拦截器不通过，后续业务逻辑不再执行

\*/

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

System.***out***.println("MyInterceptor1，前置方法正在执行");

**return** **true**;

}

/\*\*

\* 后置方法，在执行完Handler方法之后执行，倒序执行

\* ModelAndView modelAndView 可以获取信息,打印日志等

\*/

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

System.***out***.println("MyInterceptor1，后置方法正在执行");

}

/\*\*

\* 完成方法，在视图渲染完成之后执行，倒序执行

\*/

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

System.***out***.println("MyInterceptor1，完成方法正在执行");

}

}

## 配置拦截器

在springmvc-servlet.xml中配置自定义的拦截器，/\*\*：拦截所有请求

<!-- 注册自定义的拦截器 -->

<mvc:interceptors>

<mvc:interceptor>

<!-- 拦截所有请求 -->

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

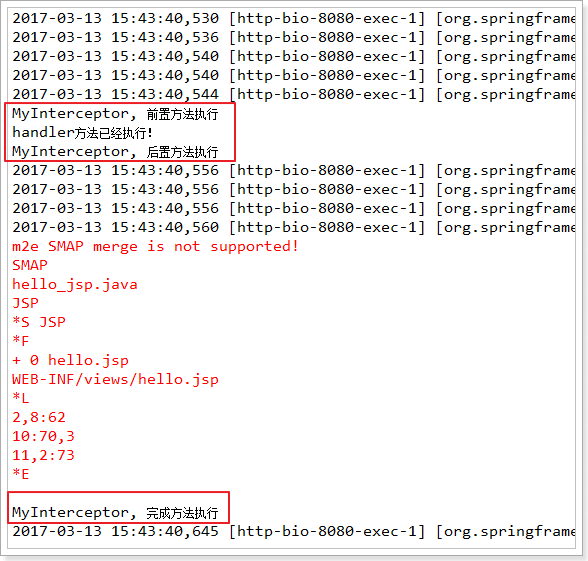
<!-- 自定义拦截器的全路径 -->

<bean class=*"cn.itcast.springmvc.interceptors.MyInterceptor1"*/>

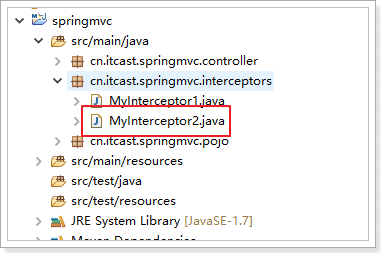
</mvc:interceptor>

</mvc:interceptors>

## 测试



## 配置多个拦截器



编写拦截器2：

**public** **class** MyInterceptor2 **implements** HandlerInterceptor {

/\*\*

\* 前置方法，在Handler方法执行之前执行

\* 返回值，返回true拦截器放行 false拦截器不通过，后续业务逻辑不再执行

\*/

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

System.***out***.println("MyInterceptor2，前置方法正在执行");

**return** **false**;

}

/\*\*

\* 后置方法，在执行完Handler方法之后执行

\*/

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

System.out.println("MyInterceptor2，后置方法正在执行 : "+ modelAndView );

// MyInterceptor2，后置方法正在执行 : ModelAndView: reference to view with name 'hello'; model is {msg=注解1}

}

/\*\*

\* 完成方法，在视图渲染完成之后执行

\*/

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

System.***out***.println("MyInterceptor2，完成方法正在执行");

}

}

配置拦截器2：

<!-- 注册自定义的拦截器 -->

<mvc:interceptors>

<mvc:interceptor>

<!-- 拦截所有请求 -->

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

<!-- 自定义拦截器的全路径 -->

<bean class=*"cn.itcast.springmvc.interceptors.MyInterceptor1"*/>

</mvc:interceptor>

<mvc:interceptor>

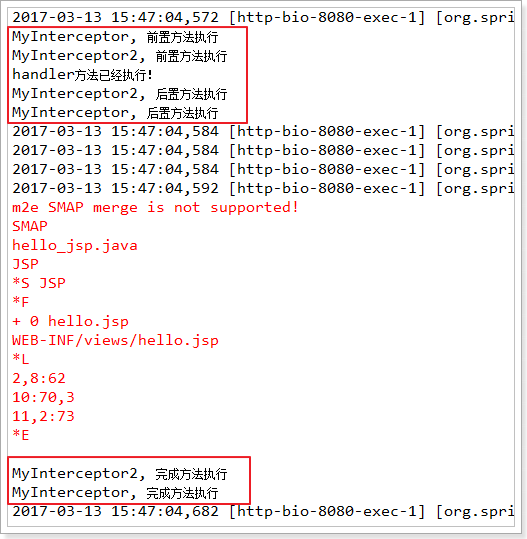
<mvc:mapping path=*"/\*\*"*/>

<bean class=*"cn.itcast.springmvc.interceptors.MyInterceptor2"*/>

</mvc:interceptor>

</mvc:interceptors>

测试：



结论：拦截器的前置方法依次执行，

后置方法和完成方法倒续执行

当前置方法返回false时，后续的拦截器以及Handler方法不再执行，但它前序的前置方法返回true的拦截器的完成方法会倒续执行。

完成方法会在视图渲染完成之后才去执行。