# SpringMVC

1、springMVC简介

2、整体架构介绍

3、hello world

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的处理

8、文件上传

9、springmvc的拦截器

# 认识Springmvc

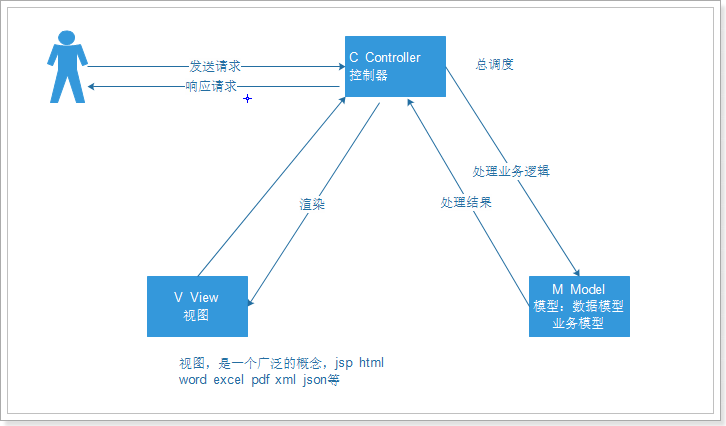
## MVC回顾

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。通常由前端和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

//



## ※SpringMVC介绍

大部分java应用都是web应用，表现层是web应用最为重要的部分。Spring为表现层提供了一个优秀的web框架------SpringMVC。和众多其他web框架一样，它是基于mvc的设计理念，此外，它采用了松散耦合可插拔组件结构，比其他mvc框架更具有扩展性和灵活性。

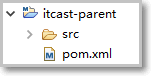
SpringMVC通过一套MVC注解，让POJO（Plain Ordinary Java Object，简单的Java对象）成为处理请求的处理器，无需实现任何接口，同时，SpringMVC还支持REST风格的URL请求。此外，SpringMVC在数据绑定、视图解析、本地化处理以及静态资源处理上都有许多不俗的表现。

它在框架设计、扩展性、灵活性等方法全面超越的Struts、WebWork等MVC框架，从原来的追赶者一跃成为了MVC的领跑者。

SpringMVC框架围绕DispatcherServlet这个核心展开，DispatcherServlet是SpringMVC框架的总导演、总策划，它负责拦截请求并将其分派给相应的处理器。

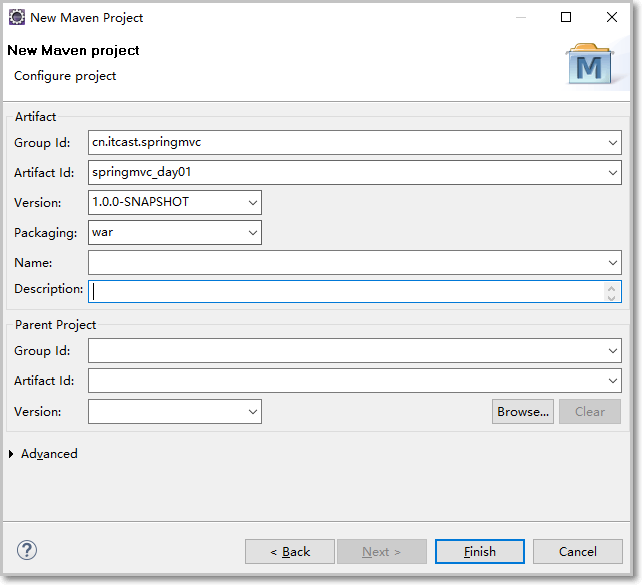
# Springmvc入门(Hello World)

## 步骤一：导入父工程

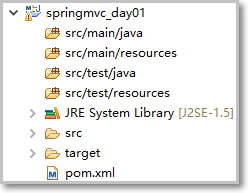


## 步骤二：创建子工程

选择打包方式为war:

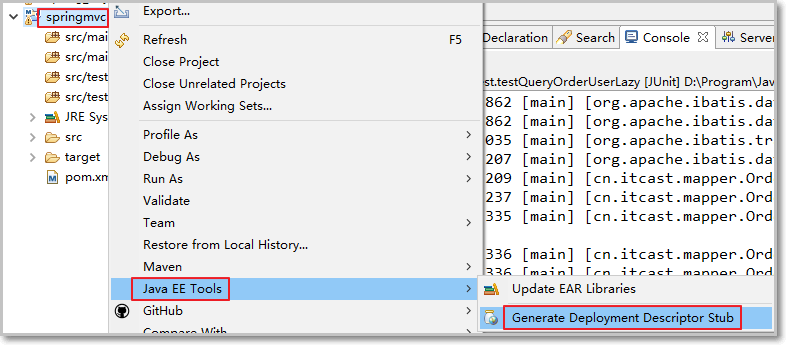


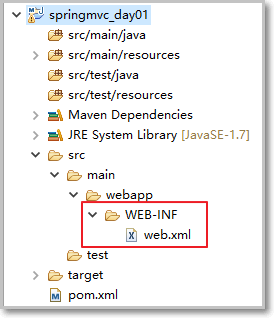
创建如下：



## 步骤三：生成web.xml

右键工程：利用工具生成web.xml





## 步骤四：引入依赖

配置Pom.xml：

<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>cn.itcast.springmvc</groupId>

<artifactId>springmvc\_day01</artifactId>

<version>1.0.0-SNAPSHOT</version>

<packaging>war</packaging>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<!-- springmvc -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

</dependency>

<!-- 日志 -->

<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>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

</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><!-- 相当于工程名称，一旦配置了/，那么在访问时可以省略工程名称 -->

</configuration>

</plugin>

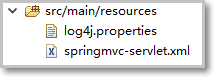
</plugins>

</build>

</project>

## 步骤五：添加日志和springmvc配置文件

在resources目录下添加日志的配置文件和springmvc的配置文件：



Log4j.properties内容：

### direct log messages to stdout ###

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

### direct messages to file mylog.log ###

log4j.appender.file=org.apache.log4j.FileAppender

log4j.appender.file.File=f:/mylog.log

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=debug, stdout,file

springmvc配置文件内容：

<?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/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*

*http://www.springframework.org/schema/context*

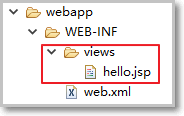
*http://www.springframework.org/schema/context/spring-context.xsd"*>

</beans>

## 步骤六：添加视图

在WEB-INF下创建一个views目录，在views目录下添加一个hello.jsp。

注意：将页面反正WEB-INF下的好处是不能直接通过url请求来访问，必须通过内部程序才能访问，相对安全。



内容如下:

<%@ 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>

<h1>hello world</h1>

</body>

</html>

## 步骤六：配置前端控制器

在web.xml中配置DispatcherServlet控制器：

<?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\_day01</display-name>

<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>

<servlet>

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

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

<!--加载指定位置的springmvc的配置文件-->

<init-param>

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

<param-value>classpath:springmvc-servlet.xml</param-value>

</init-param>

<!--配置tomcat启动是初始化前端控制器-->

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

</servlet>

<servlet-mapping>

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

<!--控制器会拦截以.do为后缀的请求-->

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

</servlet-mapping>

</web-app>

## 步骤七：创建自定义处理器

@Controller

**public** **class** HelloController {

/\*\*

\***@RequestMapping**("/show1.do"):使得请求的url可以映射到指定的目标方法上

\* **@return**

\*/

@RequestMapping("/show1.do")

**public** ModelAndView test1(){

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

//设置要显示的页面：视图

mv.setViewName("/WEB-INF/views/hello.jsp");

//添加要显示的数据

mv.addObject("msg","这是springmvc的第一个程序！");

**return** mv;

}

}

## 步骤八：添加包扫描

<?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/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"*>

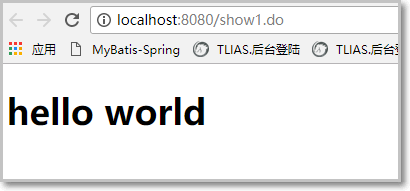
<!-- 开启注解扫描 -->

<context:component-scan base-package=*"cn.itcast.controller"*></context:component-scan>

</beans>

## ※步骤九：执行测试

测试：



从ModelAndView中获取数据并显示到页面,将页面修改如下:

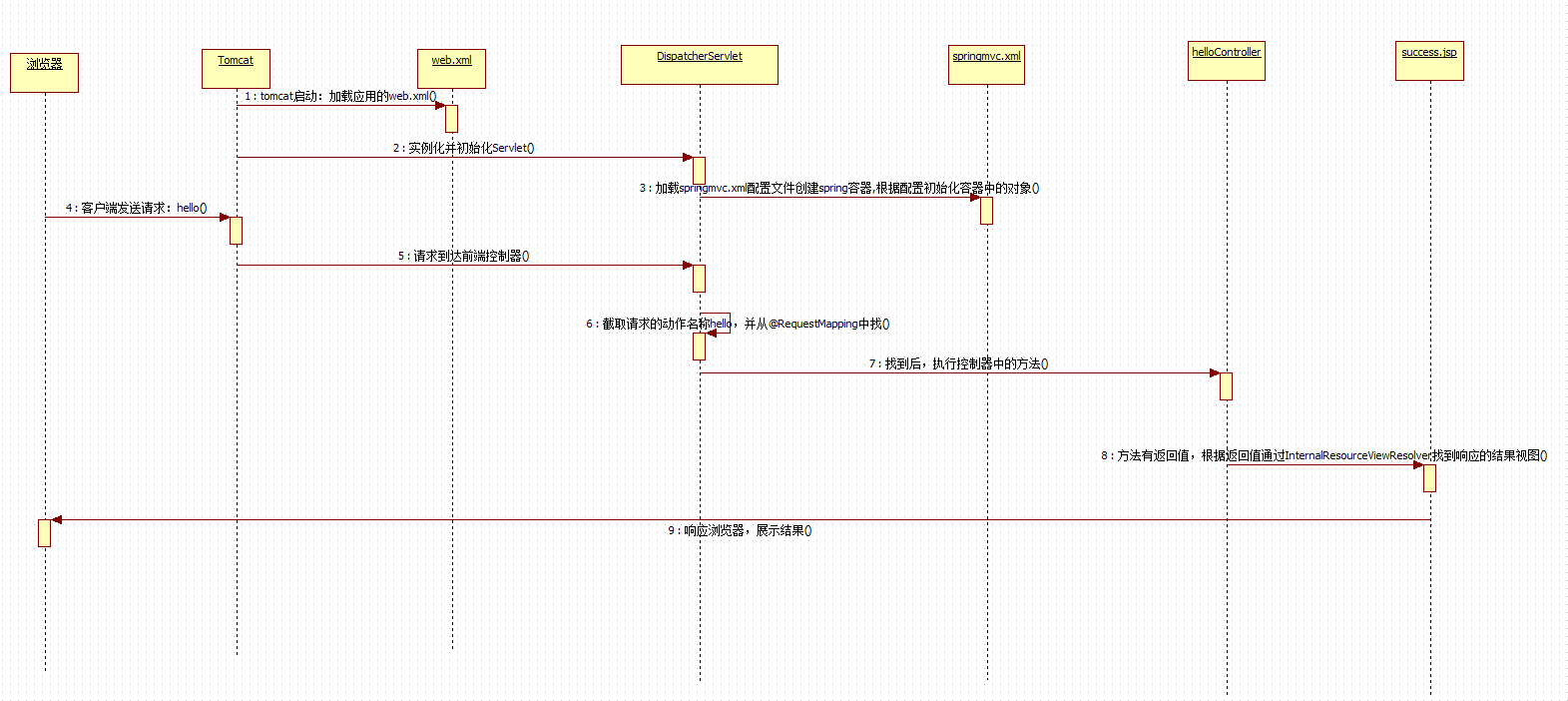
<body>

<h1>${msg}</h1>

</body>

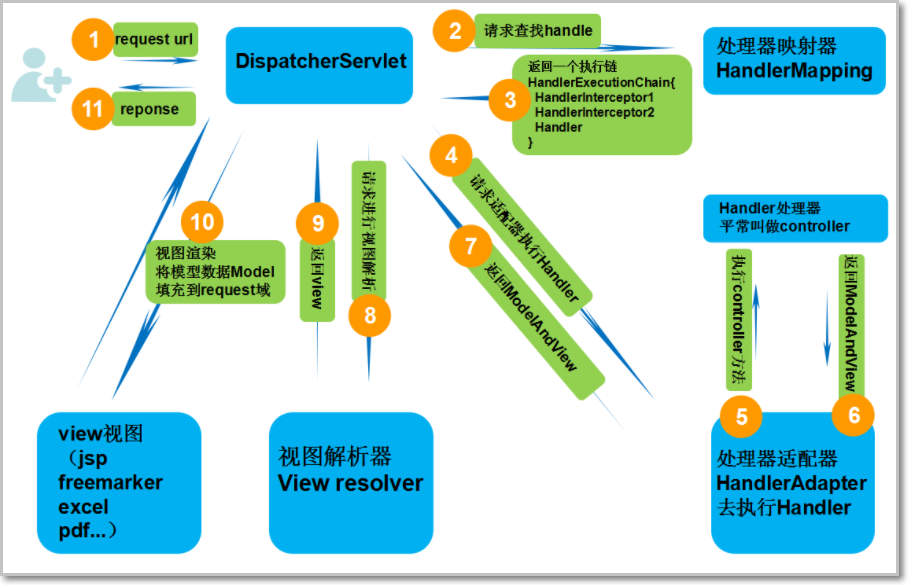


## 入门案例的执行过程

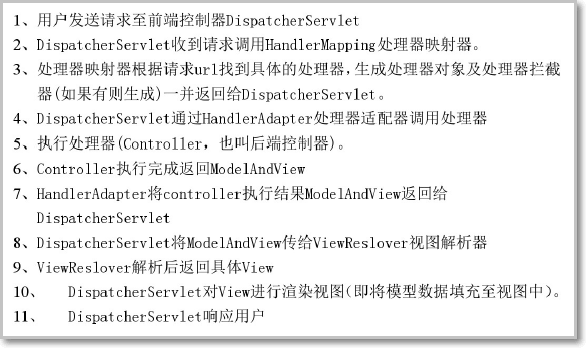


# Springmvc架构

## 框架结构



## 架构流程



## 组件说明

### 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。

### 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。

### Handler：处理器

它就是我们开发中要编写的具体业务处理器。由DispatcherServlet把用户请求转发到Handler。由Handler对具体的用户请求进行处理。

### 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。

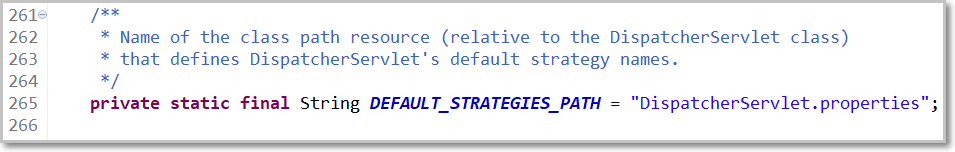
一般情况下需要通过页面标签将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

## 框架默认加载的组件

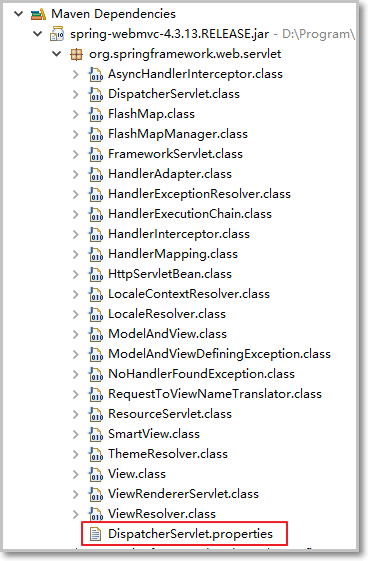
我们没有做任何配置，就可以使用这些组件，因为框架已经默认加载了这些组件。

在springmvc的执行流程中需要配置映射器和适配器，而这两个有springmvc框架帮我们默认配置完成了，因此无需用户手动配置。

在控制器DispatcherServlet中：有一行默认配置，控制器会读取该默认的资源文件DispatcherServlet.properties.



DispatcherServlet所在包的最后一个文件就是DispatcherServlet.properties：



而在该资源文件中就配置了默认的映射器和适配器：

# 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.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\

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

## 注解映射器和适配器

### 配置处理器映射器

注解式处理器映射器，对类中标记了@RequestMapping的方法进行url和方法的映射。根据url配置@RequestMapping标记的方法，匹配成功返回HandlerMethod给前端控制器。

HandlerMethod中封装了url对应的Method.

从spring3.1开始，废除了DefaultAnnotationHandlerMapping的使用，推荐使用RequestMappingHandlerMapping来完成注解式的处理器映射。

在springmvc的配置文件中需要配置如下:

<!--配置注解式的处理器映射器 -->

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

注解描述:

**@RequestMapping**：定义请求的url到处理器功能方法的映射

### 配置处理器适配器

注解式处理器适配器对标记了@RequestMapping的方法进行适配

从spring3.1开始，废除了DefaultHandlerExceptionResolver的使用，推荐使用RequestMappingHandlerAdaptor来完成注解式的处理器映射。

在springmvc的配置文件中需要配置如下:

<!--配置注解式的处理器适配器 -->

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

### 注解驱动

如果手动配置注解式的处理器映射器和处理器适配器比较麻烦，可以使用注解驱动来加载。

使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping（处理映射器）和RequestMappingHandlerAdapter（处理适配器），可以在springmvc的配置文件中使用<mvc:annotation-driven>替代注解处理器和适配器的配置。

## 视图解析器

视图解析器使用框架默认的InternalResourceViewResolver，该视图解析器支持jsp视图的解析。

<!--配置视图解析器解析规则 -->

<bean

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<!--配置视图解析器的前缀：去哪个位置找视图 -->

<property name=*"prefix"* value=*"/WEB-INF/views/"*></property>

<!--配置视图解析器的后缀：找以.jsp为后缀的视图 -->

<property name=*"suffix"* value=*".jsp"*></property>

</bean>

视图名称需要在处理器的方法中使用ModelAndView进行封装后返回，如果视图名称名为hello,那么最终返回的jsp视图地址为：

“/WEB-INF/views/hello.jsp”

最终jsp的物理地址为：前缀+视图名称+后缀

### 修改controller

@Controller

**public** **class** HelloController {

/\*\*

\* **@RequestMapping**("/show1.do"):使得请求的url可以映射到指定的目标方法上

\* **@return**

\*/

@RequestMapping("/show1.do")

**public** ModelAndView test1() {

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

// 设置要显示的页面：视图

// mv.setViewName("/WEB-INF/views/hello.jsp");

// 由于在springmvc的配置文件中配置好了前缀和后缀，这里只需要配置视图名称即可

mv.setViewName("hello");

// 添加要显示的数据

mv.addObject("msg", "这是springmvc的第一个程序！");

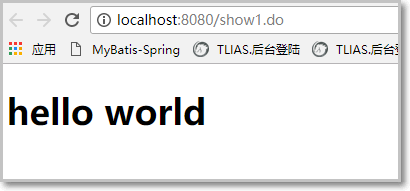
**return** mv;

}

}

### 效果

和之前一样



## ※总结

Springmvc的使用步骤：

1. 配置web.xml：
   1. 配置控制器：tomcat启动时初始化、指定springmvc的配置文件的位置
2. 编写springmvc的配置文件：

开包扫描

配置mvc注解驱动：注册各种功能-使用spring推荐的映射器和适配器

配置视图解析器：指定如果去查找视图

1. 编写自定义处理器

将处理器装配到容器中

编写请求的url映射关系：@RequestMapping(“xxx”):来映射要执行的方法

在@RequestMapping(“xxx”)下编写方法处理具体的业务逻辑，返回一个ModelAndView，其中可以封装视图名称和数据。

# RequestMapping（映射请求）

* 标准URL映射
* Ant风格的映射：\*(0个或多个字符)、？(单个字符)、\*\*(0个或多个路径)
* Rest风格的映射：占位符
* 限定请求方法的映射：get、post、put、delete
* 限定参数的映射：限定哪些请求参数可以访问

## ※标准URL映射

@RequestMapping(value=”xxx”)

在springmvc众多Controller以及每个Controller的众多方法中，请求是如何映射到具体的处理方法上的呢？

它可以定义在方法上，也可以定义在类上。

请求映射的规则：

* 定义在方法上

直接定义在方法上，这时请求路径就是方法上的@RequestMapping的value

* 定义在类上

相当于给url多加了一个一级路径。

/show1.do可以简写为show1

@Controller

@RequestMapping("hello")//相当于添加了一个一级路径

**public** **class** HelloController {

/\*\*

\* **@RequestMapping**("/show1.do"):使得请求的url可以映射到指定的目标方法上

\* **@return**

\*/

@RequestMapping("show")//简写方式

**public** ModelAndView test1() {

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

// 设置要显示的页面：视图

// mv.setViewName("/WEB-INF/views/hello.jsp");

// 由于在springmvc的配置文件中配置好了前缀和后缀，这里只需要配置视图名称即可

mv.setViewName("hello");

// 添加要显示的数据:最终会将数据存入request域中

mv.addObject("msg", "这是springmvc的第一个程序！");

**return** mv;

}

}

页面可以使用${msg}从request域中取值：

<body>

<h1>${msg }</h1>

</body>

## Ant风格的映射（通配符）

?：通配一个字符

\*：通配任意多个字符，可以是0个，可以是1个，可以是多个.但是最好和其他字符一起使用

\*\*：通配任意多个路径

### ?的使用

设置三级目录：把\*当成多个

/\*\*

\* show2?:?标识可以匹配任意单个字符

\* **@return**

\*/

@RequestMapping(value="show2?")

**public** ModelAndView test2(){

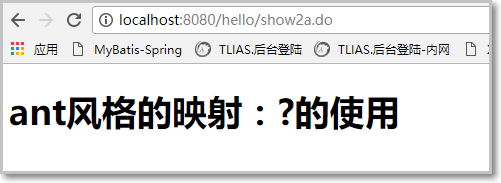
ModelAndView mv = **new** ModelAndView();

mv.setViewName("hello");

mv.addObject("msg", "ant风格的映射：?的使用");

**return** mv;

}



### \*的使用

/\*\*

\* show3\*:\*标识可以匹配任意多个字符

\* **@return**

\*/

@RequestMapping(value="show3\*")

**public** ModelAndView test3(){

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

mv.setViewName("hello");

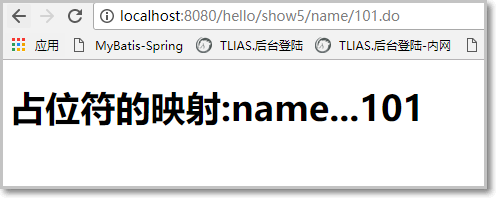
mv.addObject("msg", "ant风格的映射：\*的使用");

**return** mv;

}

把\*当成多个的意思是指hello和show2中间的目录，可以有多个字符。

可以在show3后面跟任意多个字符。



### \*\*的使用

发现\*\*可以是多个路径:

/\*\*

\* show4\*:\*\*表示可以匹配任意多级路径

\* **@return**

\*/

@RequestMapping("show4/\*\*")

**public** ModelAndView test4(){

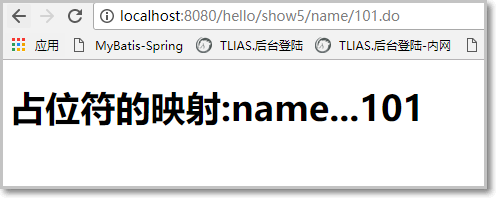
ModelAndView mv = **new** ModelAndView();

mv.setViewName("hello");

mv.addObject("msg", "ant风格的映射：\*的使用");

**return** mv;

}



注意:如果将请求修改成@RequestMapping(value="\*\*/show1")

那么就会访问到test1，相似度越高匹配度越高。

## Rest风格（占位符）的映射

@RequestMapping(value=“show5/{name}/{id} ")

请求URL：<http://localhost:8080/hello/show5/zhangsan/1001.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。

注意:占位符的主要应用是用来接收url中参数

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

@RequestMapping(value="show5/{name}/{id}")

**public** ModelAndView test5(@PathVariable("name")String name,@PathVariable("id")Long id){

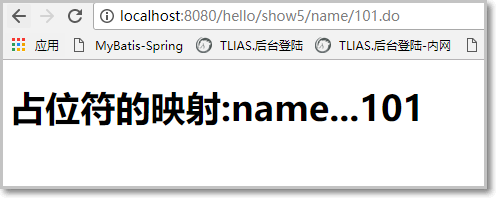
ModelAndView mv = **new** ModelAndView();

mv.setViewName("hello");

mv.addObject("msg", "占位符的映射:"+name+"..."+id);

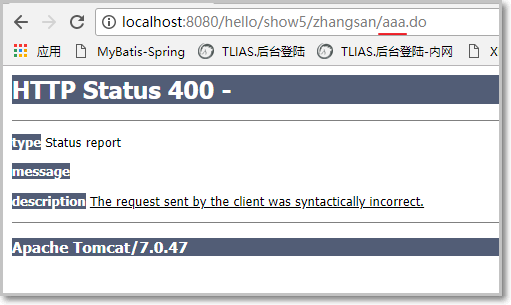
**return** mv;

}



这种传递参数的方式比之：<http://www.itcast.com?name=zhangsan&id=123>的传递方式的优势在于，不容易被猜测出来。

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）。



### 注意点

@PathVariable(value=”id”)获取占位符中的参数

注意：注解中的(value=”id”)可以省略，但是千万不要省。

代码：在@PathVariable()没有设置参数名称，也能接收参数。

@RequestMapping(value="show5/{name}/{id}")

**public** ModelAndView test5(@PathVariable()String name,@PathVariable()Long id){

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

mv.setViewName("hello");

mv.addObject("msg", "占位符的映射:"+name+"..."+id);

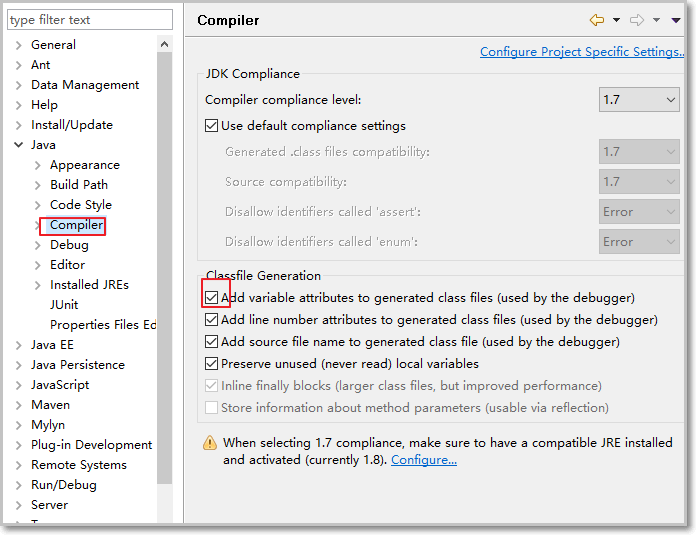
**return** mv;

}

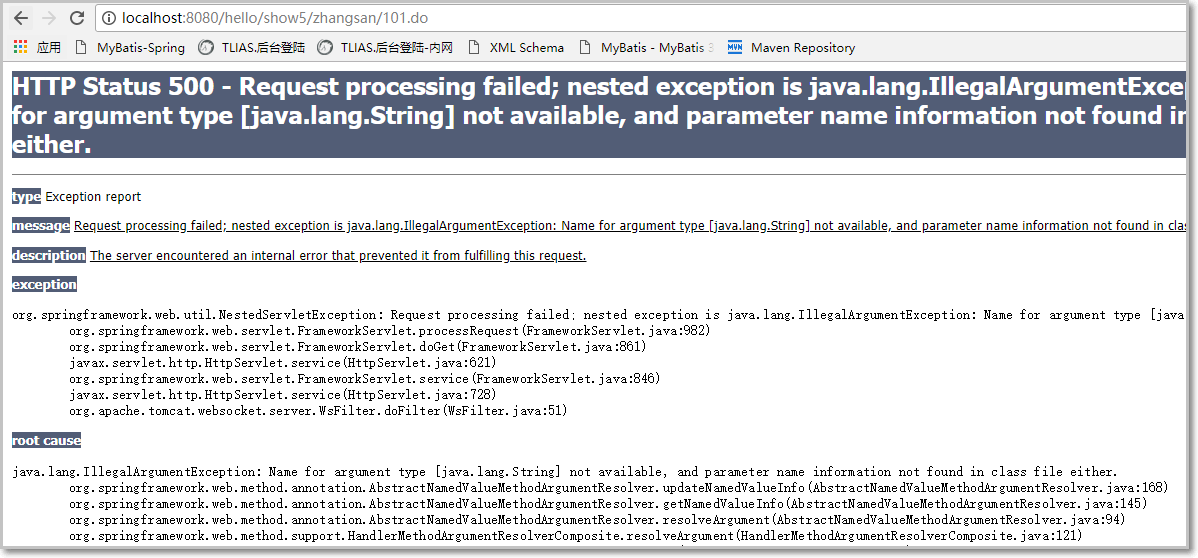
原因:这是由于我们使用的eclipse工具自动帮我们设置的。

右键项目-Properties:

Eclipse自动会将参数名称设置到class文件中，以帮助我们正常运行并获取参数。如果我们以后使用的不是eclipse，那么就可能报错。



如果将Add的勾选去除，再次访问会报错如下：



但是如果此时在注解中添加了参数名称就能正常访问

@RequestMapping(value="show5/{name}/{id}")

**public** ModelAndView test5(@PathVariable("name")String name,@PathVariable("id")Long id){

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

mv.setViewName("hello");

mv.addObject("msg", "占位符的映射:"+name+"..."+id);

**return** mv;

}



因此，注意：一定要做注解中设置参数名称，这样的话，哪怕使用的是其他的工具，也不会有任何问题。

## 限定请求方法的映射

### 限定单个请求方法

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

@RequestMapping(value="show6",method=RequestMethod.***POST***)

**public** ModelAndView test6(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

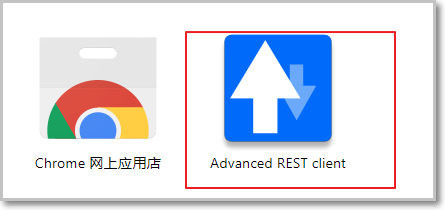
mv.addObject("msg", "限定请求方法的映射：post");

**return** mv;

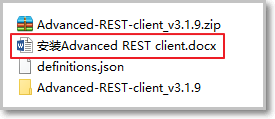
}

原来的post请求一般都是通过编写一个表单，然后通过表单发送请求。

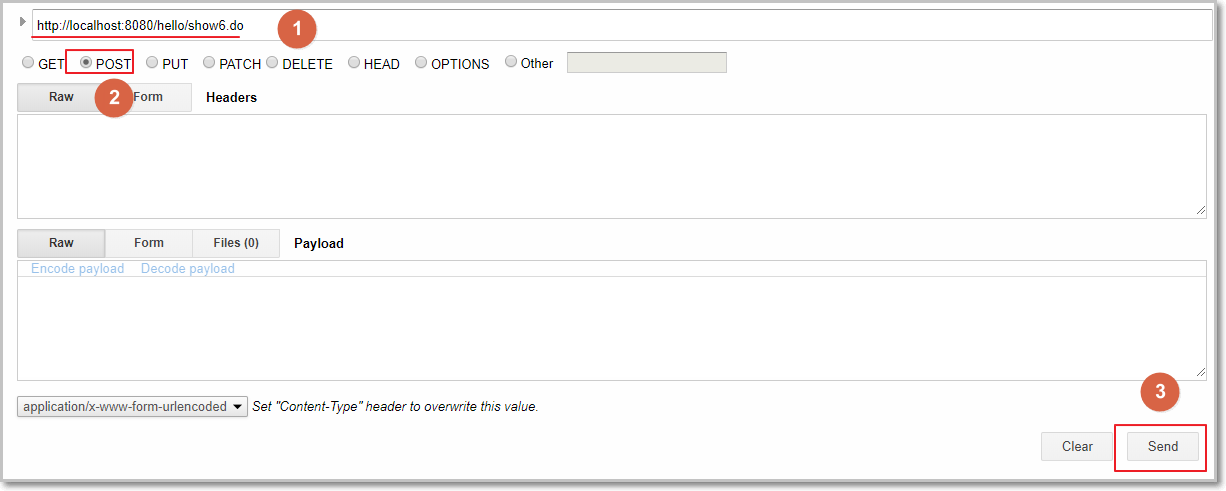
因此为了以后测试方便，我们可以安装一个chrome的插件：在发送请求时可以模拟各种请求方法、get、post、put、delete等



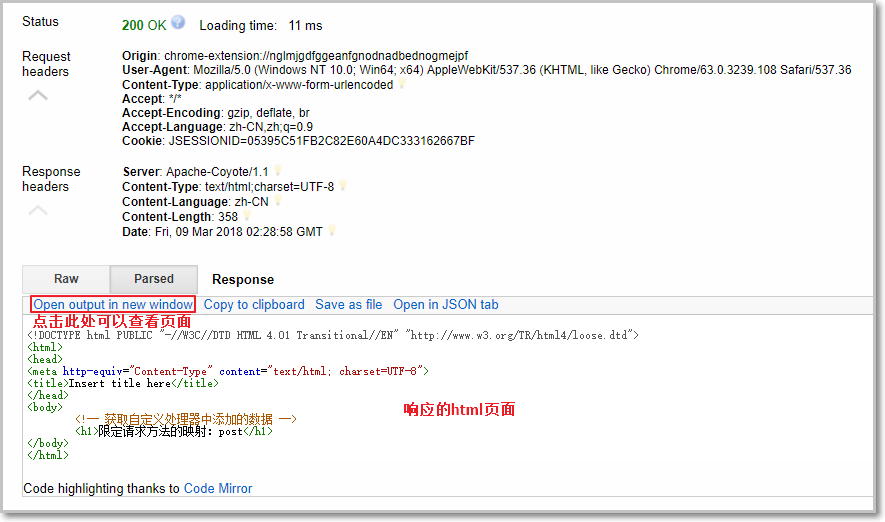
安装过程，参见资料：将压缩包解压出来后有三个文件，参见其中的安装文档进行安装



然后使用该插件进行测试：



点击send,执行请求后:可以正常访问



注意：在发送请求时一定要选择对应的请求方法，否则会报405请求方法错误。

### 限定多个请求方法

@RequestMapping(value="show6",method={RequestMethod.***POST***,RequestMethod.***GET***})

编写测试方法：

@RequestMapping(value="show7",method={RequestMethod.***POST***,RequestMethod.***GET***})

**public** ModelAndView test7(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "限定请求方法的映射：post/get");

**return** mv;

}

此时不论使用post请求还是get请求都可以访问。

## 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

params=”userId=1”：请求参数中必须包含参数userId并且值必须为1

params=”userId!=1”：请求的所有参数中如果有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

注意：参数名称和参数值直接不要有空格，即userId=1,而不是userId = 1，因为一般在url中是不会有空格的

### 必须携带参数

params=”userId”：请求参数中必须带有userId

代码：

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

**public** ModelAndView test8(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "限定请求参数的映射：必须有参数userId");

**return** mv;

}

没有参数的访问:报400，参数列表不合法。



### 必须排除参数

params=”!userId”：请求参数必须不能是userId

代码:

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

**public** ModelAndView test9(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "限定请求参数的映射：必须没有参数userId");

**return** mv;

}

携带参数userId访问:



不携带userId:



### 必须固定参数值

params=”userId=101”：请求参数userId必须为101

代码：

@RequestMapping(value="show10",params="userId=101")

**public** ModelAndView test10(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "限定请求参数的映射：参数userId的值必须为101");

**return** mv;

}

### 必须排除参数值

params=”userId!=1”：请求参数中userId必须不为1，或者参数也没有userId

代码：

@RequestMapping(value="show11",params="userId!=101")

**public** ModelAndView test11(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "限定请求参数的映射：参数userId的值必须不能为101，也可以没有参数userId");

**return** mv;

}

只要请求携带的userId不为101即可，即使不携带userId参数也行

### 必须携带部分参数

params={“name”, ”userId”}：请求参数中必须有userId，name参数，之前的几种携带规则都可以应用到此处。

params={“name=zhangsan”, ”userId!=101”}：请求参数中name必须式zhangsan,并且userId必须不为101.

代码：

@RequestMapping(value="show12",params={"name=zhangsan","userId!=101"})

**public** ModelAndView test12(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "限定请求参数的映射：参数必须username为zhangsan并且userId不为101");

**return** mv;

}

## 组合注解

@GetMapping：相当于@RequestMapping（method = RequestMethod.GET）

@PostMapping：相当于@RequestMapping（method = RequestMethod.POST）

@PutMapping：相当于@RequestMapping（method = RequestMethod.PUT）

@DeleteMapping：相当于@RequestMapping（method = RequestMethod.DELETE）

也可以在注解中限定请求参数，用法和之前的一样。

### @GetMapping

代码：

@GetMapping(value="show13")

**public** ModelAndView test13(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "注解限定请求参数的映射：GetMapping");

**return** mv;

}

测试：



### @PostMapping

代码：

@PostMapping(value="show14")

**public** ModelAndView test14(){

ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "注解限定请求参数的映射：@PostMapping");

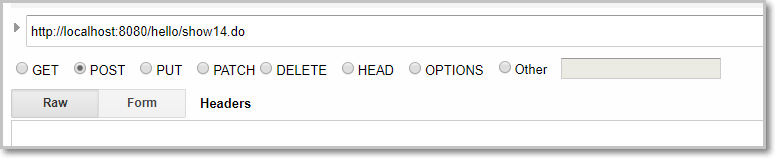
**return** mv;

}

页面测试：默认是get请求，因此报405请求方法错误



必须使用插件测试：



### @DeleteMapping

代码：

@DeleteMapping(value="show15")

**public** ModelAndView test15(){

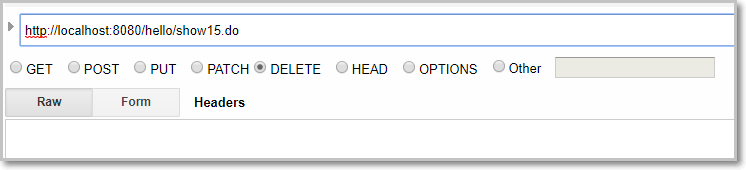
ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "注解限定请求参数的映射：@DeleteMapping");

**return** mv;

}

必须使用插件测试：



### ※/@PutMapping

代码：

@PutMapping(value="show16")

**public** ModelAndView test16(){

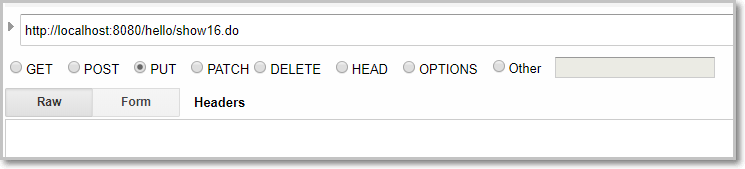
ModelAndView mv = **new** ModelAndView("hello");//在构造函数中设置视图名称

mv.addObject("msg", "注解限定请求参数的映射：@PutMapping");

**return** mv;

}

使用插件测试：



# 其他返回值类型的方法

## 返回值是String类型

在讲接收数据和数据绑定之前，为了编写代码方便，我们可以先对之前的代码进行优化。在原来的代码中，我们必须手动初始化ModelAndView才能设置视图名称和数据，这样比较麻烦，可以使用更加简单的编写方式。

代码简化如下：

@RequestMapping(value="show17")

**public** String test17(Model model){

model.addAttribute("msg", "优化后的代码");//添加数据

**return** "hello";//springmvc默认将返回值最为视图名称

}

## 返回值是void类型

@ResponseStatus(value=HttpStatus.***OK***)：如果不响应页面，就需要响应状态。

@RequestMapping("show18")

@ResponseStatus(value=HttpStatus.***OK***)

**public** **void** test18(Model model) {

System.***out***.println("返回值是void");

}

# 接收数据及数据绑定

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

## 接收servlet常用的内置对象

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

@RequestMapping(value="show19")

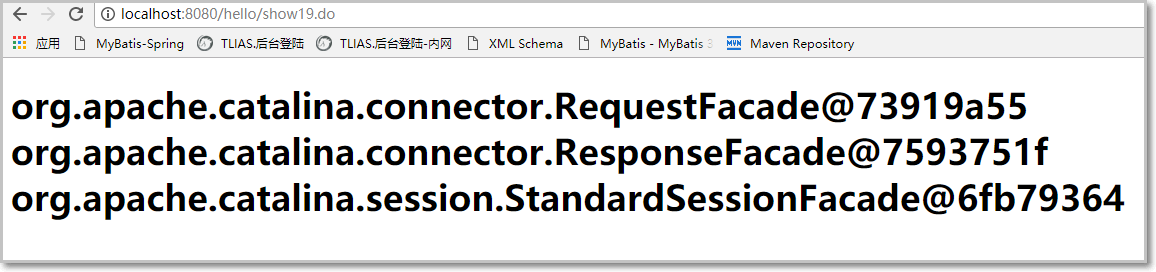
**public** String test19(Model model,HttpServletRequest request,HttpServletResponse response,HttpSession session){

model.addAttribute("msg", request+"<br/>"+response+"<br/>"+session);

**return** "hello";

}

测试：



## ※接收普通的请求参数

@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（即使手动设置了也会失效），自动为false，如果请求中不包含该参数则使用默认值。

@RequestParam和@PathVariable的区别：

@RequestParam比之@PathVariable的区别是不需要在注解中使用占位符{xxx}

不设置required属性，那么默认就必须要提交参数name，因为required默认为true

代码：

@RequestMapping(value="show20")

**public** String test20(Model model,@RequestParam(value="name")String name){

model.addAttribute("msg", "使用@RequestParam接收到的参数为："+name);

**return** "hello";

}

只有当手动设置required为false时，才可以不用传递参数：

修改如下：

@RequestMapping(value="show20")

**public** String test20(Model model,@RequestParam(value="name",required=**false**)String name){

model.addAttribute("msg", "使用@RequestParam接收到的参数为："+name);

**return** "hello";

}

测试不传递参数:

defaultValue属性

如果设置了defaultValue属性，那么表示可以不用传递参数，一旦不传递参数，就使用默认值。

代码如下:

@RequestMapping(value="show20")

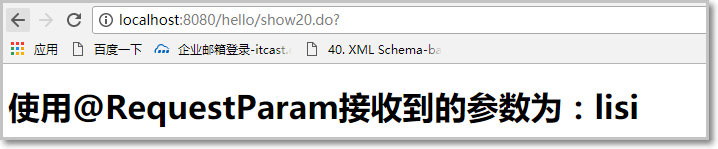
**public** String test20(Model model,@RequestParam(value="name",defaultValue="lisi")String name){

model.addAttribute("msg", "使用@RequestParam接收到的参数为："+name);

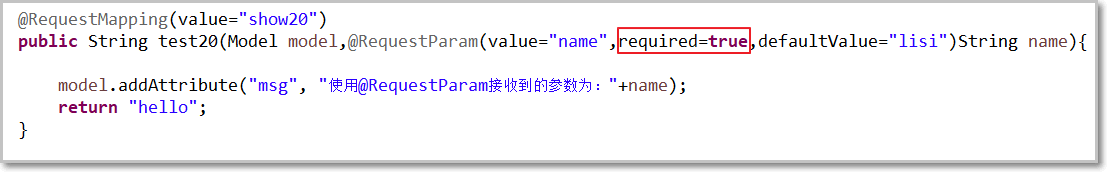
**return** "hello";

}

测试不传递参数：使用默认值

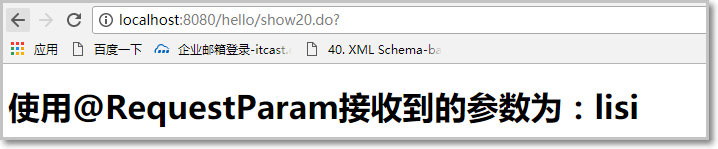


哪怕测试手动设置required为true也没有，照样可以不传递参数。



再次测试：效果一样。

结论：只要配置了defaultValue属性，required属性就失效了。



## 获取cookie

### 传统方式获取cookie

需求：将cookie中的jsessionid打印到页面

代码：

@RequestMapping(value = "show21")

**public** String test21(Model model, HttpServletRequest request) {

Cookie[] cookies = request.getCookies();

**for** (Cookie cookie : cookies) {

**if** (cookie.getName().equalsIgnoreCase("jsessionid")) {

model.addAttribute("msg", "jsessionid：" + cookie.getValue());

}

}

**return** "hello";

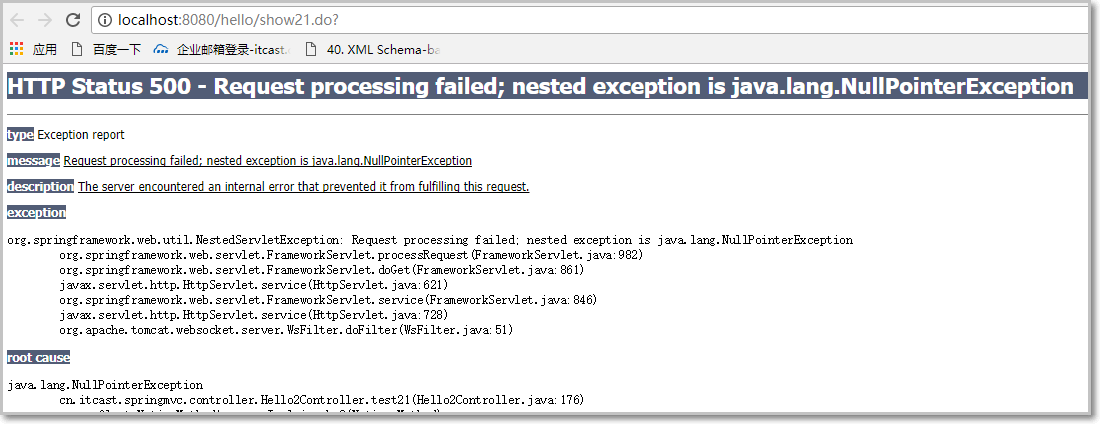
}

测试访问：

注意:由于chrome自带缓存，因此当第一次访问时就会有cookie数据，因此在测试前请记得将chrome的缓存先清空一下或者可以使用IE浏览器。

Chrome清空缓存快捷键：shift+ctrl+delete

然后访问时，发现由于第一次访问，cookie为null.因此会报错：



Tomcat启动后的第一次请求，Cookie是没有数据的，当服务器进行响应的时候才会利用session生成一个sessionid响应给客户端，这个sessionid就是cookie中的jsessionid,当发送第二次以后的请求时，cookie会自动携带jsessionid到服务器端。

为了防止空指针，需要手动在代码中进行判断一下，正确代码如下:

@RequestMapping(value = "show21")

**public** String test21(Model model, HttpServletRequest request) {

Cookie[] cookies = request.getCookies();

**if** (cookies!=**null**) {//由于第一次访问，cookie是null,因此需要手动判断一下

**for** (Cookie cookie : cookies) {

**if** (cookie.getName().equalsIgnoreCase("jsessionid")) {

model.addAttribute("msg", "jsessionid：" + cookie.getValue());

}

}

}

**return** "hello";

}

以上代码太麻烦。

### 注解方式获取cookie

使用注解方式获取cookie就比传统方式就简单的多。

@CookieValue使用方法同@RequestParam

代码:注意@CookieValue("JSESSIONID")必须大写

@RequestMapping(value = "show22")

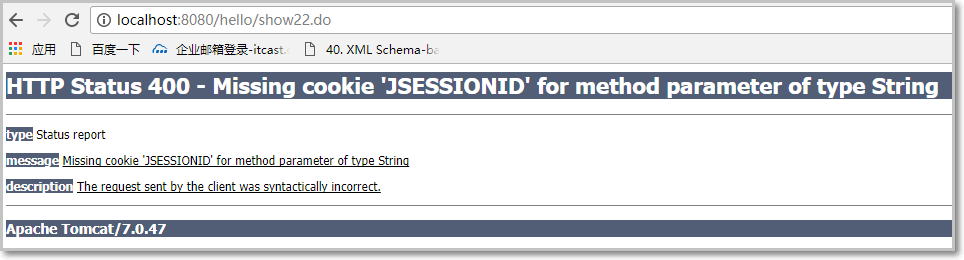
**public** String test22(Model model, @CookieValue("JSESSIONID")String jsessionid) {

model.addAttribute("msg", "jsessionid：" + jsessionid);

**return** "hello";

}

注意：如果是清除chrome缓存后发送请求，此时cookie中没有数据，因此也无法在参数中使用注解获取jsessionid,那么就会报400参数列表错误。



此时当cookie中没有该属性时，可以设置一个默认值：

@RequestMapping(value = "show22")

**public** String test22(Model model, @CookieValue(value="JSESSIONID",defaultValue="101")String jsessionid) {

model.addAttribute("msg", "jsessionid：" + jsessionid);

**return** "hello";

}

发现使用注解获取cookie数据比传统方式简单的多。

## 基本数据类型的绑定

字符串、整型、浮点型、布尔型、数组。

需求：通过一个页面提交某些数据，由自定义处理器获取这些数据并打印到控制台，但是不需要响应任何页面。

### 步骤一：创建页面

在webapp目录下创建一个user.html页面，内如如下：

<!DOCTYPE html>

<html>

<head>

<meta charset=*"UTF-8"*>

<title>Insert title here</title>

</head>

<body>

<form action=*"/hello/show23.do"* method=*"post"*>

name:<input type=*"text"* name=*"name"* /><br />

age:<input type=*"text"* name=*"age"* /><br />

isMarry:<input type=*"radio"* name=*"isMarry"* value=*"1"*/>Y

<input type=*"radio"* name=*"isMarry"* value=*"0"*/>N<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>

注意：在页面中，如果提交的是1或者0，也可以使用Boolean值接收，转换成true或者false.

### 步骤二：添加请求方法

在自定义处理器中添加一个请求方法：

@RequestMapping("show23")

**public** String test23(Model model, @RequestParam("name") String name,

@RequestParam("age") Integer age,

@RequestParam("isMarry") Boolean isMarry,

@RequestParam("income") Float income,

@RequestParam("interests") String[] interests) {

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

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

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

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

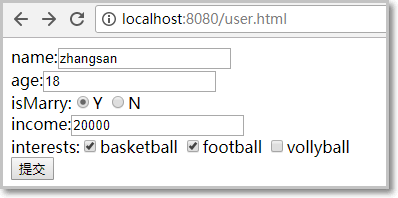
System.***out***.println(Arrays.*toString*(interests));

**return** "hello";

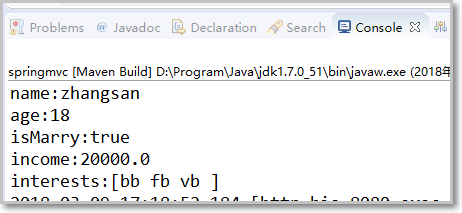
}

### 步骤三：访问user.html

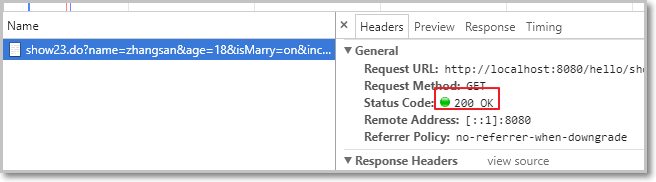
通过该页面提交数据



提交后控制台如下:测试成功！



并且页面F12可以看到:状态码为200



注意:如果不设置状态码：即不配置注解@ResponseStatus(value=HttpStatus.***OK***)就会报404。

相应状态HttpStatus是一个枚举类

## 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）。

### 步骤一：创建user对象

内容如下：

**public** **class** User {

**private** String name;

**private** Integer age;

**private** Boolean isMarry;

**private** Float income;

**private** String [] interests;

**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** Boolean getIsMarry() {

**return** isMarry;

}

**public** **void** setIsMarry(Boolean isMarry) {

**this**.isMarry = isMarry;

}

**public** Float getIncome() {

**return** income;

}

**public** **void** setIncome(Float income) {

**this**.income = income;

}

**public** String[] getInterests() {

**return** interests;

}

**public** **void** setInterests(String[] interests) {

**this**.interests = interests;

}

@Override

**public** String toString() {

**return** "User [name=" + name + ", age=" + age + ", isMarry=" + isMarry + ", income=" + income + ", interests="

+ Arrays.*toString*(interests) + "]";

}

}

### 步骤二：编写Controller方法

@RequestMapping(value="show24")

**public** String test24(Model model,User user) {

model.addAttribute("msg",user);

**return** "hello";

}

### 步骤三：修改页面

再添加一个show24.do的表单

<form action=*"/hello/show24.do"* method=*"post"*>

name:<input type=*"text"* name=*"name"* /><br />

age:<input type=*"text"* name=*"age"* /><br />

isMarry:<input type=*"radio"* name=*"isMarry"* value=*"1"*/>Y

<input type=*"radio"* name=*"isMarry"* value=*"0"*/>N<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

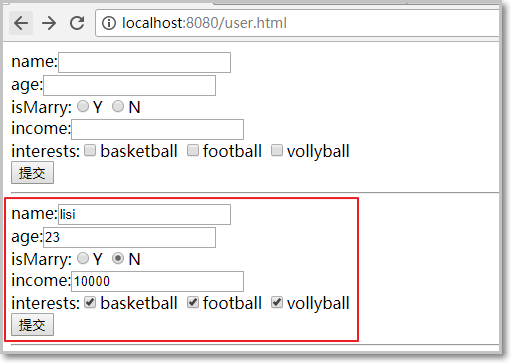
<input type=*"checkbox"* name=*"interests"* value=*"vb"* />vollyball<br />

<input type=*"submit"* value=*"提交"* />

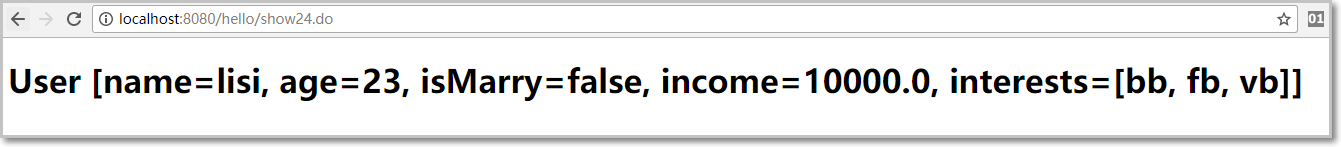
</form>

### 步骤四：测试

提交表单执行测试:



测试结果:



### 步骤五：再次获取对象中的参数

当自动讲数据封装到User对象中之后，如果还想单独获取某个或几个参数，可以单独设置形参获取。

将方法修改如下:

@RequestMapping(value="show24")

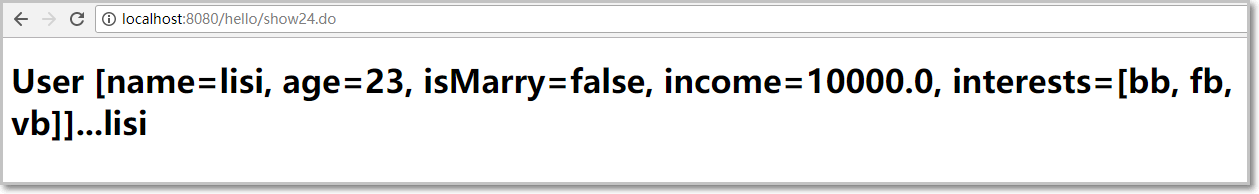
**public** String test24(Model model,User user,@RequestParam(value="name")String name) {

model.addAttribute("msg","打印参数："+user+"..."+name);

**return** "hello";

}

测试结果：依旧可以获取数据



## 集合的绑定

### 集合中元素为基本类型

Controller方法：

@RequestMapping(value="show25")

**public** String test25(Model model,@RequestParam("ids")List<Long> ids) {

model.addAttribute("msg","打印参数："+ids.toString());

**return** "hello";

}

测试：



或者也可以：



### 集合中元素为pojo类型

#### 步骤一：添加表单

在user.html中添加一个表单，提交两个人的数据。

<form action=*"/hello/show26.do"* method=*"post"*>

name0:<input type=*"text"* name=*"name"* /><br />

age0:<input type=*"text"* name=*"age"* /><br />

isMarry0:<input type=*"radio"* name=*"isMarry"* value=*"1"*/>Y

<input type=*"radio"* name=*"isMarry"* value=*"0"*/>N<br>

income0:<input type=*"text"* name=*"income"* /><br />

interests0:<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 />

name1:<input type=*"text"* name=*"name"* /><br />

age1:<input type=*"text"* name=*"age"* /><br />

isMarry1:<input type=*"radio"* name=*"isMarry"* value=*"1"*/>Y

<input type=*"radio"* name=*"isMarry"* value=*"0"*/>N<br>

income1:<input type=*"text"* name=*"income"* /><br />

interests1:<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 />

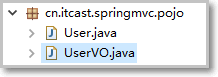
<input type=*"submit"* value=*"提交"* />

</form>

报错如下:



#### 步骤二：创建一个UserVO来包装List集合：



内容：

**public** **class** UserVO {

**private** List<User> users;

**public** List<User> getUsers() {

**return** users;

}

**public** **void** setUsers(List<User> users) {

**this**.users = users;

}

@Override

**public** String toString() {

**return** "UserVO [users=" + users + "]";

}

}

#### 步骤三：修改Controller方法：

@RequestMapping(value="show26")

**public** String test26(Model model,UserVO userVO) {

model.addAttribute("msg","打印参数："+userVO.toString());

**return** "hello";

}

#### 步骤四：修改user.html

<form action=*"/hello/show26.do"* method=*"post"*>

name0:<input type=*"text"* name=*"users[0].name"* /><br />

age0:<input type=*"text"* name=*"users[0].age"* /><br />

isMarry0:<input type=*"radio"* name=*"users[0].isMarry"* value=*"1"*/>Y

<input type=*"radio"* name=*"users[0].isMarry"* value=*"0"*/>N<br>

income0:<input type=*"text"* name=*"users[0].income"* /><br />

interests0:<input type=*"checkbox"* name=*"users[0].interests"* value=*"bb"* />basketball

<input type=*"checkbox"* name=*"users[0].interests"* value=*"fb"* />football

<input type=*"checkbox"* name=*"users[0].interests"* value=*"vb"* />vollyball<br />

name1:<input type=*"text"* name=*"users[1].name"* /><br />

age1:<input type=*"text"* name=*"users[1].age"* /><br />

isMarry1:<input type=*"radio"* name=*"users[1].isMarry"* value=*"1"*/>Y

<input type=*"radio"* name=*"users[1].isMarry"* value=*"0"*/>N<br>

income1:<input type=*"text"* name=*"users[1].income"* /><br />

interests1:<input type=*"checkbox"* name=*"users[1].interests"* value=*"bb"* />basketball

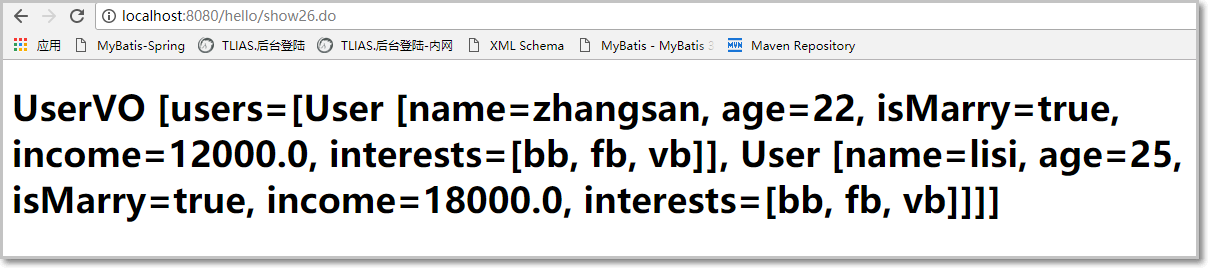
<input type=*"checkbox"* name=*"users[1].interests"* value=*"fb"* />football

<input type=*"checkbox"* name=*"users[1].interests"* value=*"vb"* />vollyball<br />

<input type=*"submit"* value=*"提交"* />

</form>

效果：



# jstl标签的使用

springmvc虽然有自己的标签库，但是其使用并不方便，因此几乎没有人使用，甚至没有人知道其存在。因而一般还是使用的jstl标签。

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， SQL标签。

需求：将一个静态页面userList.html转换成jsp页面并在页面上使用jstl标签显示数据

## 步骤一：导入jstl依赖（pom.xml）

导入jstl依赖包：

<dependency>

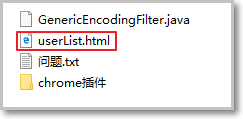
<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

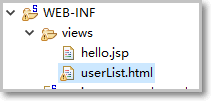
</dependency>

## 步骤二：创建静态页面

将资料中的userList.html复制到view目录下即可

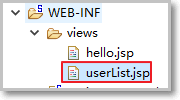


复制：



## 步骤三：改成jsp动态页面

将userList.html重命名成jsp页面



找一个jsp页面，将其头信息复制该userList.jsp中：

<%@ 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">

引入核心标签库

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

## 步骤四：检查页面数据

由于一般页面都是由前台提供的，因此需要核对一下页面所需数据是否在pojo对象中有对应的属性。

发现如果没有，需要手动更改pojo对象：

添加id和username属性并提供set和get方法，并且重新toString一下：

**public** **class** User {

**private** Long id;

**private** String username;

**private** String name;

**private** Integer age;

**private** Boolean isMarry;

**private** Float income;

**private** String [] interests;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**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** Boolean getIsMarry() {

**return** isMarry;

}

**public** **void** setIsMarry(Boolean isMarry) {

**this**.isMarry = isMarry;

}

**public** Float getIncome() {

**return** income;

}

**public** **void** setIncome(Float income) {

**this**.income = income;

}

**public** String[] getInterests() {

**return** interests;

}

**public** **void** setInterests(String[] interests) {

**this**.interests = interests;

}

@Override

**public** String toString() {

**return** "User [id=" + id + ", username=" + username + ", name=" + name + ", age=" + age + ", isMarry=" + isMarry

+ ", income=" + income + ", interests=" + Arrays.*toString*(interests) + "]";

}

}

## 步骤五：编写Controller方法

创建一个list集合，集合中循环添加20个user对象

@RequestMapping(value="show27")

**public** String test27(Model model) {

List<User> list = **new** ArrayList<User>();

**for**(**int** i = 0;i< 20;i++) {

User user = **new** User();

user.setId(i+1L);

user.setUsername("zhangsan"+i);

user.setName("张三"+i);

user.setAge(18);

list.add(user);

}

model.addAttribute("users", list);

**return** "userlist";

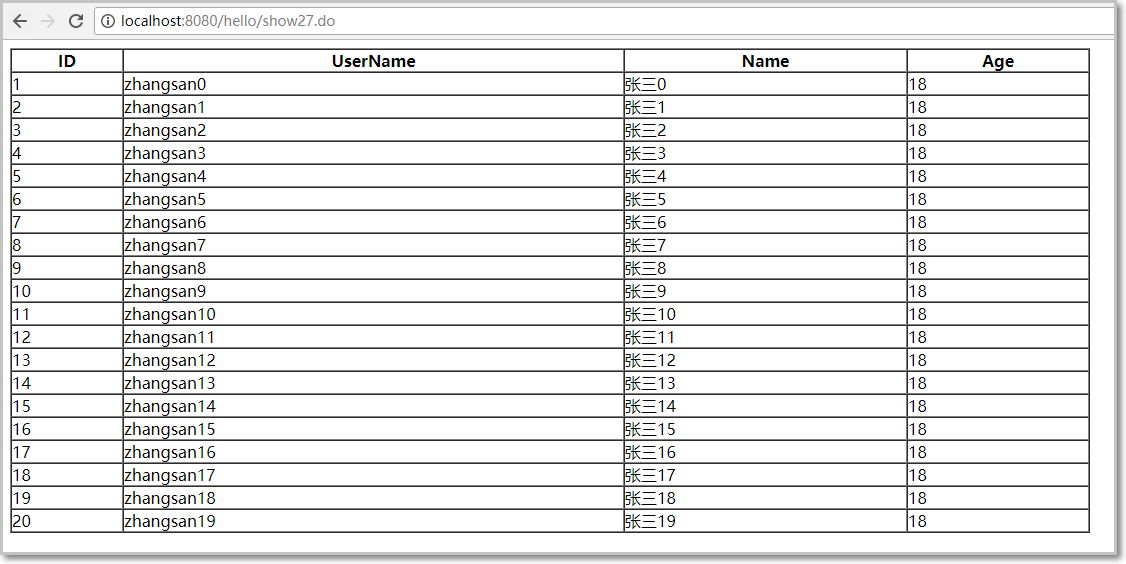
}

## 步骤六：使用<c:foreach>标签

使用jstl标签在页面上显示数据



执行测试



# JSON

在实际开发过程中，json是最为常见的一种方式（本质上就是字符串），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

转换json的常用方式：

1. Gson

Google的工具，功能强大，但是效率稍逊。

1. Fastjson

阿里的工具，效率高，但是功能稍逊。

1. jackson

springmvc内置的转换工具，功能和效率都有不俗的表现，在世界范围内使用较广。

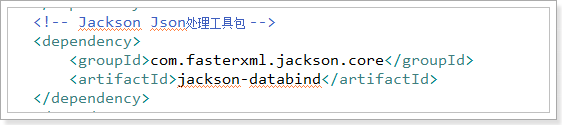
## 响应json数据

@ResponseBody：

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会通过默认的json转化器转化输出。

### 步骤一：引入依赖

在pom.xml中引入jackson依赖，参考父工程：



### 步骤二：编写controller方法

/\*\*

\* 将list集合响应成json数据

\* **@return**

\*/

@RequestMapping(value="show28")

@ResponseBody//将数据响应成json格式的数据

**public** List<User> test28() {

List<User> list = **new** ArrayList<User>();

**for**(**int** i = 0;i< 20;i++) {

User user = **new** User();

user.setId(i+1L);

user.setUsername("zhangsan"+i);

user.setName("张三"+i);

user.setAge(18);

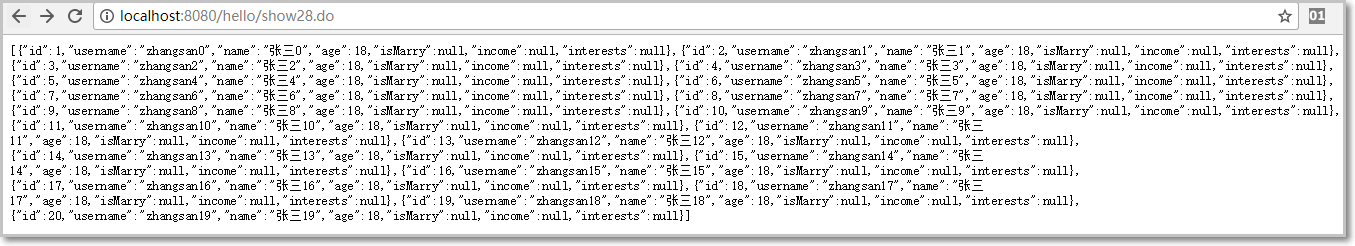
list.add(user);

}

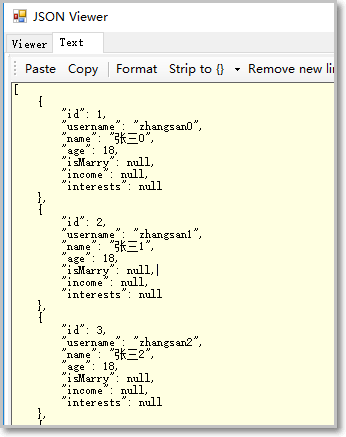
**return** list;

}

### 步骤三：执行测试



也可以使用jsonview格式化数据：



## 使用pojo接收json数据

@RequestBody：接收一个json并且转换成一个对象。

接收一个json数据并反序列化为一个user对象。

### 步骤一：编写Controller方法

/\*\*

\* 将提交的json格式的数据封装到user对象中

\*

\* **@RequestBody**()：自动将json数据序列化成一个user对象

\* **@param** model

\* **@param** user

\* **@return**

\*/

@RequestMapping(value="show29")

**public** String test29(Model model,@RequestBody()User user) {

model.addAttribute("msg", user);

**return** "hello";

}

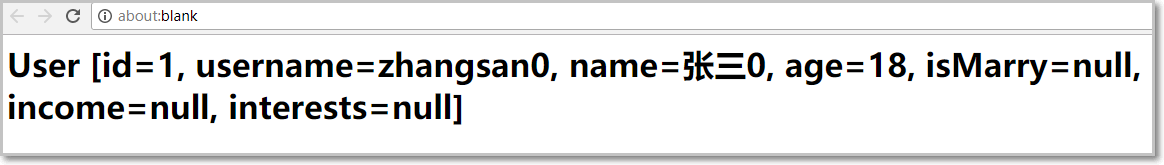
### 步骤二：测试



执行结果：



也可以打开页面：



问题：前台发送json格式的数据，后台能否执行使用String类型来接收？

可以：因为json本质上就是一个字符串。

## 使用String接收json数据

### 步骤一：controller方法

@RequestMapping(value="show30")

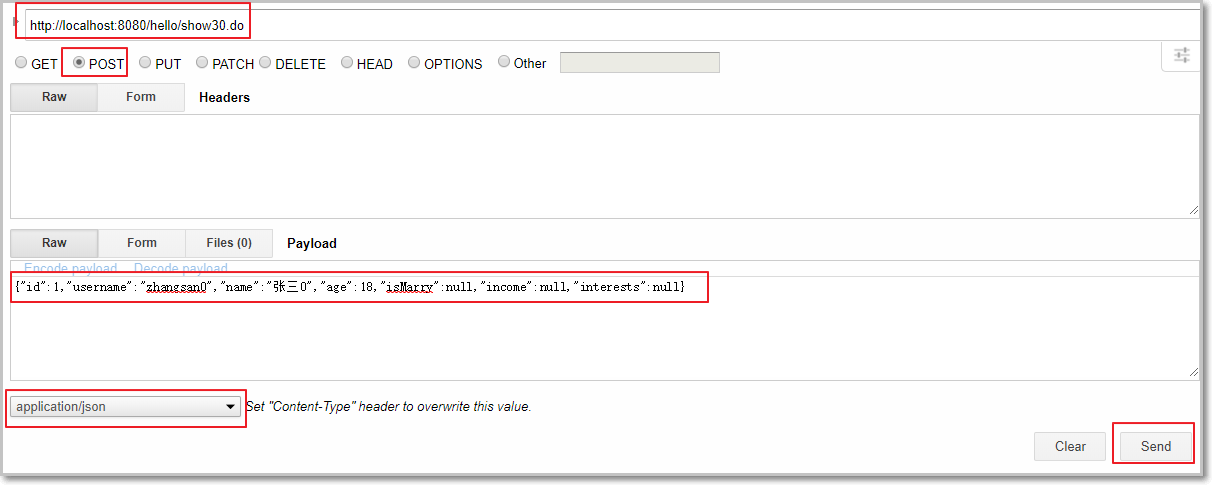
**public** String test30(Model model,@RequestBody()String user) {

model.addAttribute("msg", user);

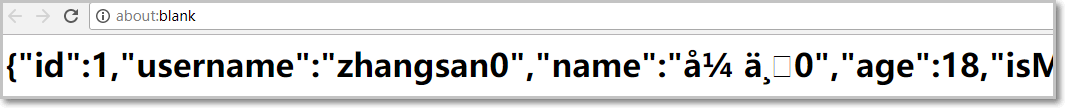
**return** "hello";

}

### 步骤二：测试

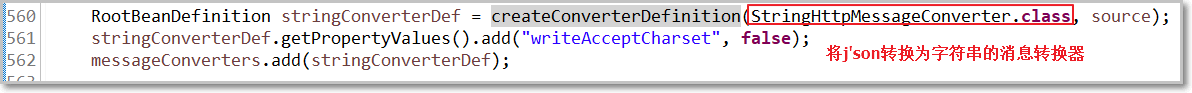


测试结果:中文乱码了

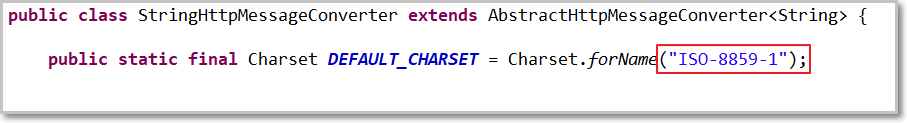


### 解决中文乱码问题

原因:使用的消息转换器换成了StringHttpMessageConverter



该转换器中使用的编码为：



解决方案:

1、可以配置一个全站乱码过滤器。

2、手动修改消息转换器中的编码集

这里我们使用第二种方式：手动修改消息转换器中的编码集

将注解驱动修改如下：

1. 设置String的消息转换器
2. 该消息转换器中有一个构造函数可以设置编码集，因此只要直接赋值即可。

<mvc:annotation-driven>

<mvc:message-converters>

<bean class=*"org.springframework.http.converter.StringHttpMessageConverter"*>

<constructor-arg index=*"0"* value=*"UTF-8"*></constructor-arg>

</bean>

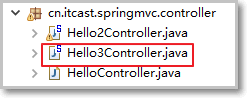
</mvc:message-converters>

</mvc:annotation-driven>

## @RestController

有时如果在一个Contoller中所有的方法都是用来响应json格式数据的，那么如果有多个方法，就需要在多个方法上使用@ResponseBody，这样太麻烦，springmvc提供了一个@RestController，将该注解使用在Controller类上，那么该controller中的所有方法都默认是响应json格式的数据了。

### 步骤一：编写一个Hello3Controller



@Controller

@RequestMapping("hello3")

@RestController//声明所有的方法都使用json格式进行响应

**public** **class** Hello3Controller {

/\*\*

\* 此处不使用@ResponseBody

\* **@return**

\*/

@RequestMapping(value="demo1")

**public** List<User> demo1(){

List<User> list = **new** ArrayList<User>();

**for**(**int** i = 0;i< 20;i++) {

User user = **new** User();

user.setId(i+1L);

user.setUsername("zhangsan"+i);

user.setName("张三"+i);

user.setAge(18);

list.add(user);

}

**return** list;

}

@RequestMapping(value="demo2")

**public** String demo2(Model model) {

model.addAttribute("msg", "使用@RestController注解");

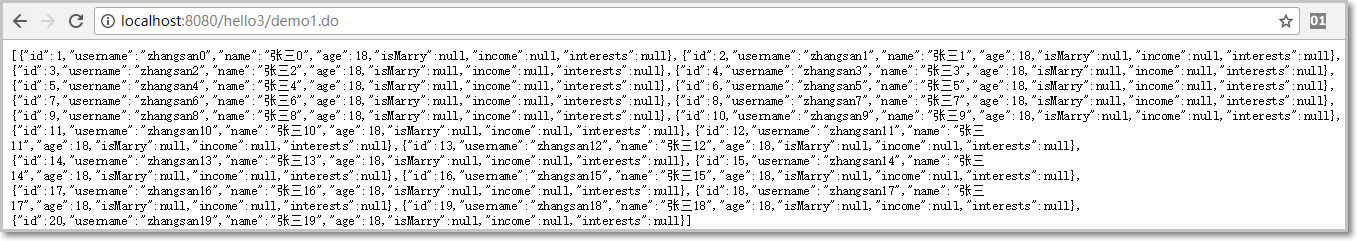
**return** "hello";

}

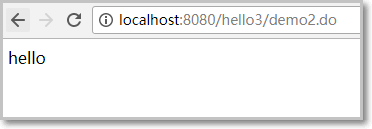
}

### ※步骤二：测试

测试demo1:



测试demo2:发现哪怕本来应该响应一个页面的，最终将返回值响应成了一个json格式的数据。



# 文件上传

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

/\*

Spring有两个web相关的包：



该包中存放的springmvc的核心功能



该包中存放的是web的一些通用功能：比如监听器ContextLoaderListener以及文件上传解析器CommonsMultipartResolver等, 而文件上传解析器就需要fileupload的支持。

\*/

## 步骤一：添加依赖

添加文件上传的依赖：

<!-- 文件上传 -->

<dependency>

<groupId>commons-fileupload</groupId>

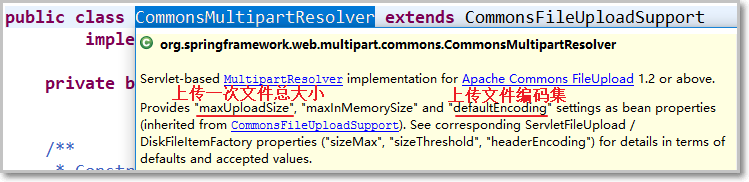
<artifactId>commons-fileupload</artifactId>

<version>1.3.1</version>

</dependency>

## 步骤二：配置文件上传解析器

需要在springmvc的配置文件中配置一个文件上传解析器（在spring-web包下），并且设置其解析器的某些参数，如上传文件大小和编码集等。



在springmvc-servlet.xml中配置：

注意：必须配置id属性，并且名称固定为*multipartResolver，*否则无法使用.

<!-- 配置文件上传解析器 -->

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

<!--

maxUploadSize:默认单位是字节

设置为5m:5\*1024\*1024=5242880 -->

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

<property name=*"defaultEncoding"* value=*"utf-8"*></property>

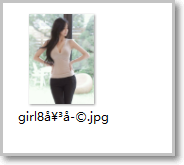
</bean>

defaultEncoding:用来设置上传文件名称的参数，如果不设置的话，问题如下:

1. 上传的参数名称不能为中文



2、图片名称如果有中文，保存时会乱码：



## 步骤三：Controller方法

注意:存放文件的目录upload需要先行创建好

/\*\*

\* 文件上传

\* 需要通过MultipartFile类型来接收数据

\*

\* **@throws** Exception

\* **@throws** IllegalStateException

\*/

@RequestMapping("show31")

**public** String test31(Model model,@RequestParam("file")MultipartFile file) **throws** Exception{

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

file.transferTo(**new** File("d://upload//"+file.getOriginalFilename()));

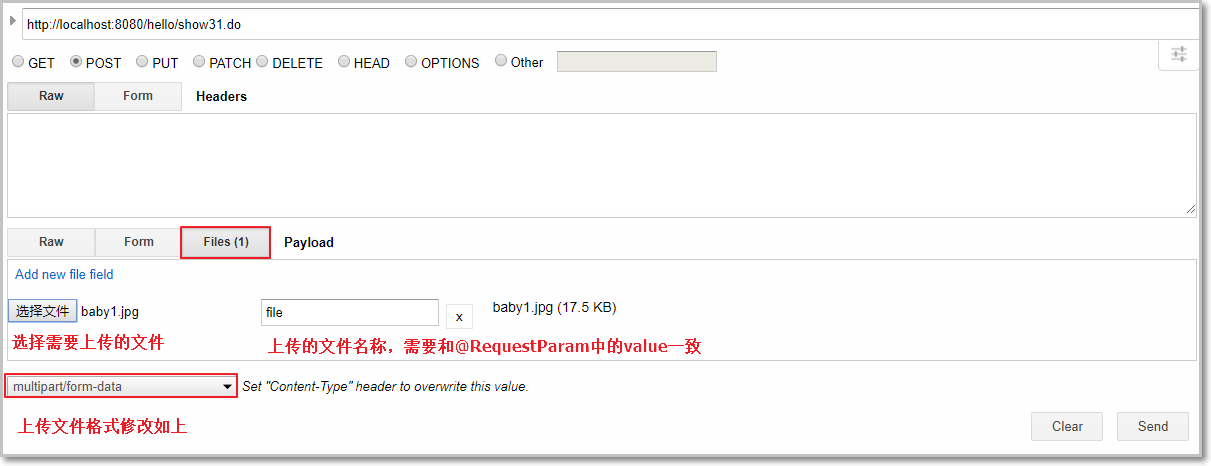
}

model.addAttribute("msg", "上传成功!");

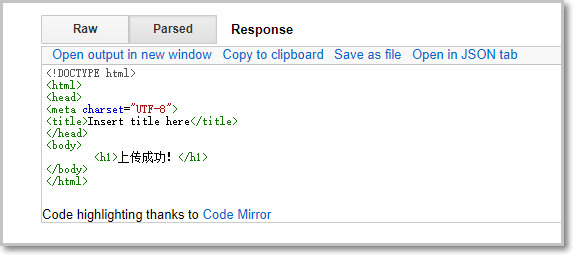
**return** "hello";

}

## 步骤四：执行测试



效果如下：上传成功，并且可以在d:\upload目录下找到上传的文件。



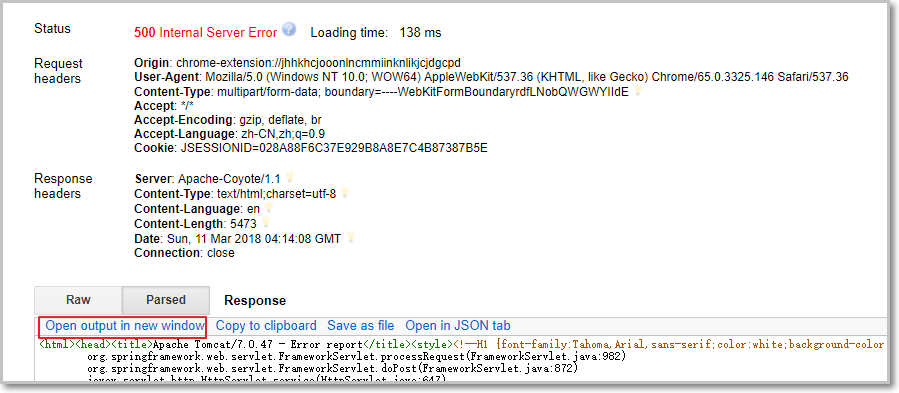
## 问题：上传文件超出大小怎么办？

注意：在上传文件解析器中，maxUploadSize的属性是用来设置一次性上传文件的总大小的。

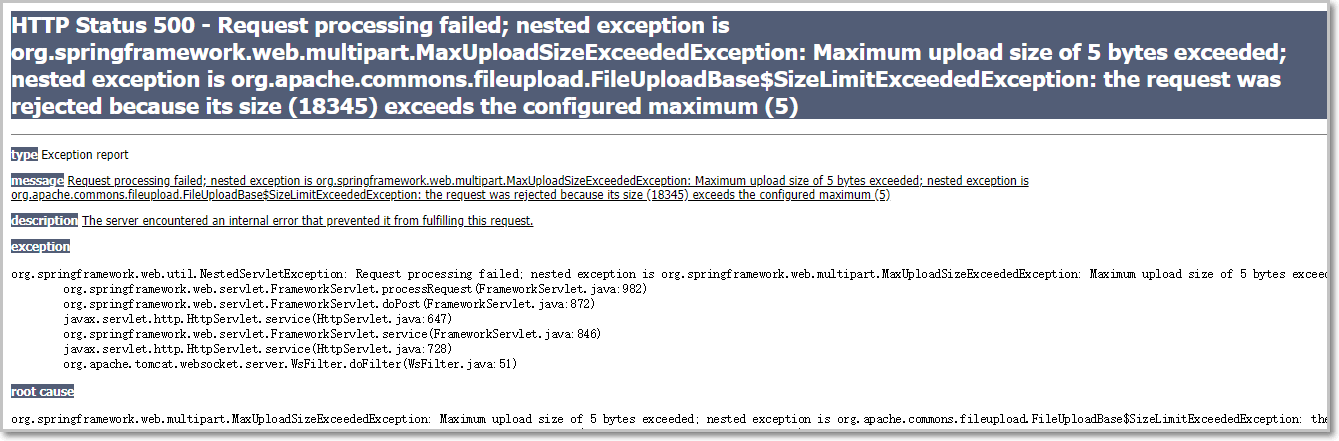
比如:如果上传了一个文件，那么该文件的不得超过5m,如果一次性上传了10个文件，那么10个文件的总大小不得超出5m.



一旦这个是设置的比较小，上传的文件比较大的话，就会报错：

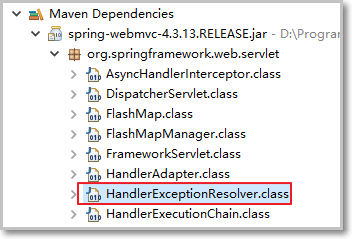


然后打开页面后，错误如下：



如此的话，用户体验不好，我们应该自定义一个错误页面来告诉用户，哪里出问题了。

这是有一个处理器异常解析器抛出的错误，在请求到达我们自定义的Controller之前就抛出了异常。

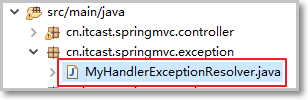


如何解决？

自定义一个处理器异常解析器:

1. 自定义处理器异常解析器实现HandlerExceptionResolver
2. 在springmvc-servlet.xml中注册自定义的处理器异常解析器

### 步骤一：自定义处理器异常解析器



内容如下:

**public** **class** MyHandlerExceptionResolver **implements** HandlerExceptionResolver{

@Override

**public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler,

Exception ex) {

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

//对文件大小异常进行判断

**if** (ex **instanceof** MaxUploadSizeExceededException) {

mv.setViewName("hello");

mv.addObject("msg", "上传文件太大了！");

}

**return** mv;

}

}

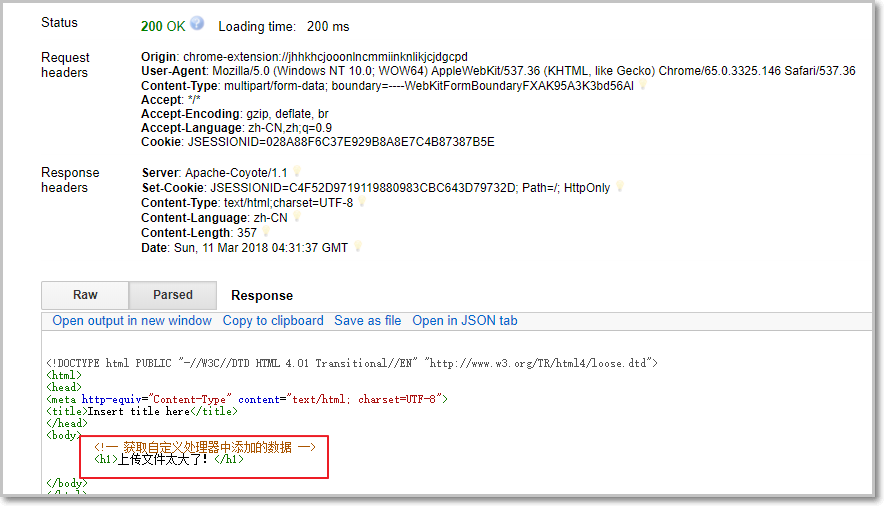
### 步骤二：注册自定义处理器异常解析器

在springmvc-servlet.xml中配置如下:

<!-- 注解处理器异常解析器 -->

<bean class=*"cn.itcast.springmvc.exception.MyHandlerExceptionResolver"*></bean>

### 步骤三：测试



打开页面：



## 问题：能否上传多个？



可以：直接使用数组接收后遍历循环即可。

@RequestMapping("show32")

**public** String test32(Model model,@RequestParam("files")MultipartFile[] files) **throws** Exception{

**for** (MultipartFile file : files) {

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

file.transferTo(**new** File("d://upload//"+file.getOriginalFilename()));

}

}

model.addAttribute("msg", "上传成功");

**return** "hello";

}

# 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)

重定向：redirect:/hello/show.do(绝对路径)或者redirect:show.do(相对路径)

/：表示绝对路径，指的是localhost:8080/springmvc（项目名称可以省略）

不带/：表示相对路径，相对于当前请求的路径

如果当前请求是：localhost:8080/springmvc（项目名称可以省略）/hello/show32

那么不带/：表示localhost:8080/springmvc（项目名称可以省略）/hello/

【示例】

Controller方法：

@RequestMapping(value="show32")

**public** String test32() {

**return** "forward:show34.do?id=101&type=forword";

}

@RequestMapping(value="show33")

**public** String test33() {

**return** "redirect:show34.do?id=101&type=redirect";

}

@RequestMapping(value="show34")

**public** String test34(Model model,@RequestParam("id")Long id,@RequestParam("type")String type) {

model.addAttribute("msg", "是转发还是重定向？"+id+"..."+type);

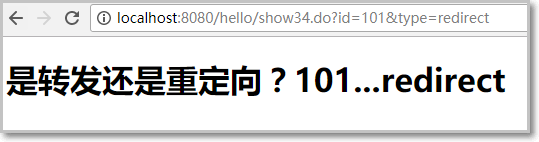
**return** "hello";

}

测试：show32



测试：show33



# 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返回执行链给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：

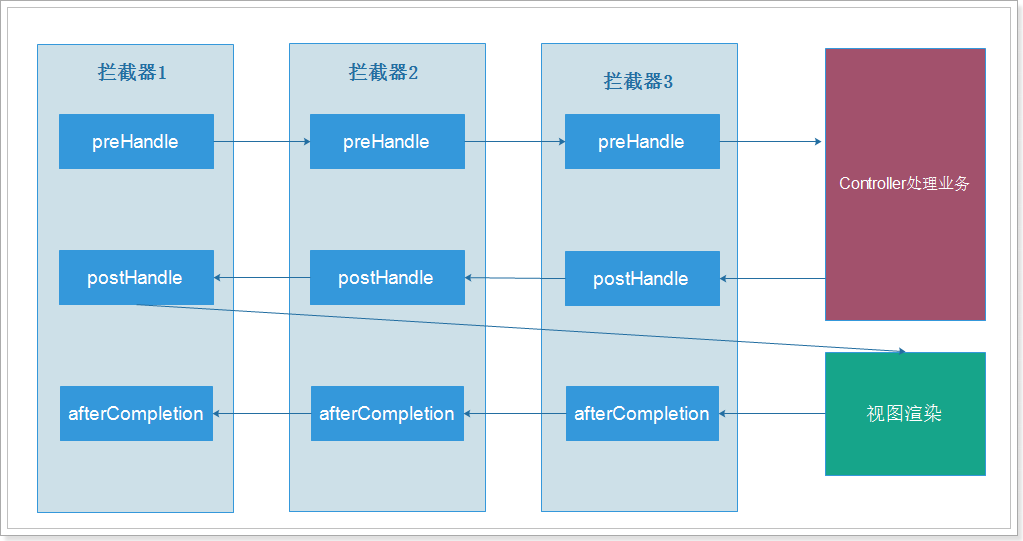
* preHandle调用Handler之前执行，称为前置方法

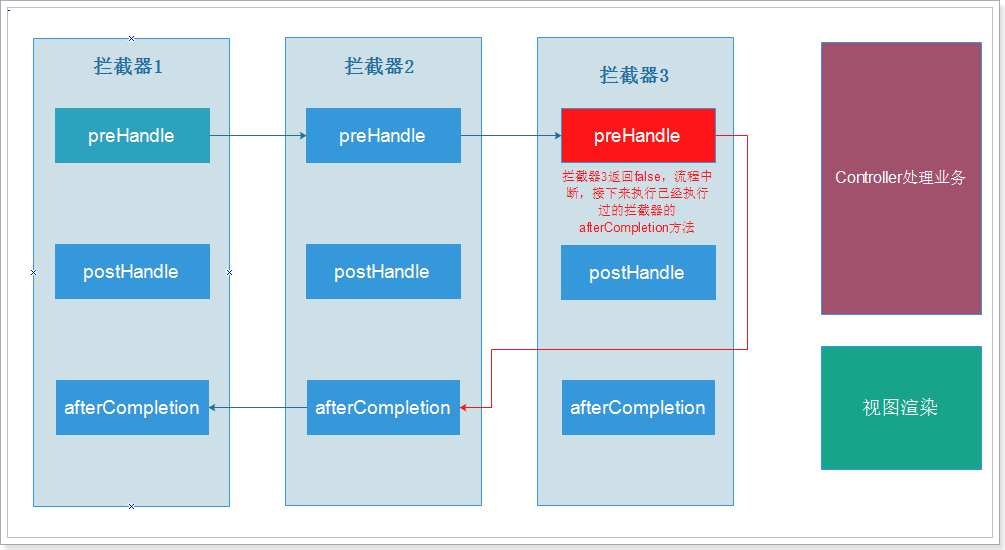
返回值：true表示放行，后续业务逻辑继续执行

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

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

## 拦截器的执行过程





拦截器的配置过程：

1. 编写自定义拦截器实现HandlerInterceptor
2. 在springmvc-servlet.xml中注册自定义拦截器

## 单个自定义拦截器

### 步骤一：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

\*/

@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");

}

}

### 步骤二：配置自定义拦截器

/： 表示绝对路径：

<http://localhost:8080/springmvc>

/\*：表示绝对路径下的任意一级路径：

<http://localhost:8080/springmvc>/xxx

/\*\*：表示绝对路径下的任意多级目录：

<http://localhost:8080/springmvc>/xxx

<http://localhost:8080/springmvc/xxx/xxx/xxx>

注意：之所以请求url中看不到springmvc，是因为在pom.xml中配置了tomcat插件，配置了path为/而省略了项目名称springmvc

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

<!-- 配置自定义拦截器 -->

<mvc:interceptors>

<mvc:interceptor>

<!-- 表示拦截所有请求 -->

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

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

</mvc:interceptor>

</mvc:interceptors>

### 步骤三：编写自定义处理器

@RequestMapping(value="show35")

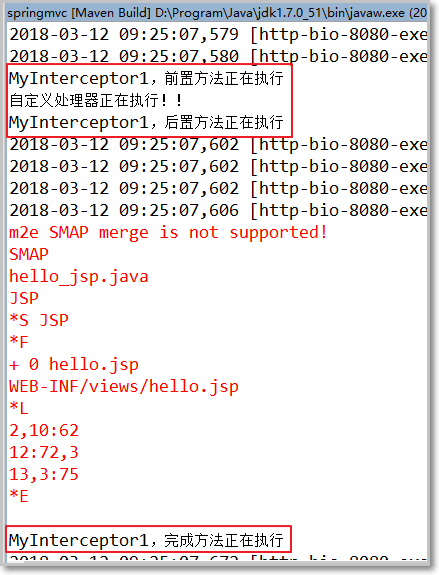
**public** String test35() {

System.***out***.println("自定义处理器正在执行！！");

**return** "hello";

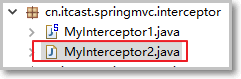
}

### 测试



## 多个自定拦截器

### 步骤一：配置拦截器2



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，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("MyInterceptor2，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

\*/

@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");

}

/\*\*

\* 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("MyInterceptor2，完成方法正在执行");

}

}

### 步骤二：配置拦截器2

<!-- 配置自定义拦截器 -->

<mvc:interceptors>

<mvc:interceptor>

<!-- 表示拦截所有请求 -->

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

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

</mvc:interceptor>

<mvc:interceptor>

<!-- 表示拦截所有请求 -->

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

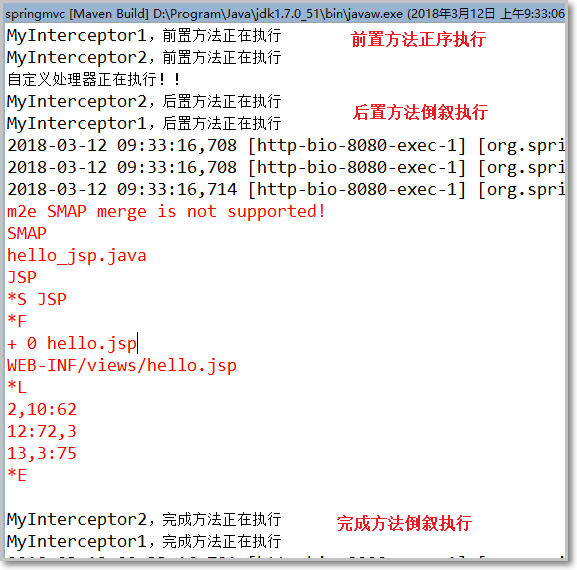
<bean class=*"cn.itcast.springmvc.interceptor.MyInterceptor2"*></bean>

</mvc:interceptor>

</mvc:interceptors>

### 步骤三：执行测试

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。

## 中断拦截器的前置方法

### 中断第一个拦截器的前置方法

将第一个拦截器的前置方法返回值设置为false:

**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** **false**;

}

/\*\*

\* 后置方法，在执行完Handler方法之后执行，倒序执行

\*/

@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)

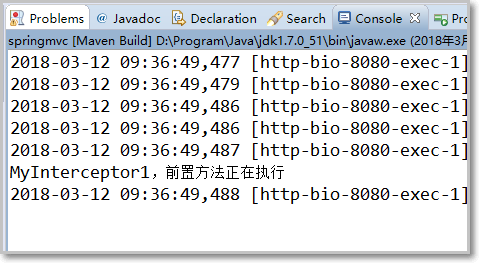
**throws** Exception {

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

}

}

执行测试：后续所有方法不再执行。



### 中断第二个拦截器的前置方法

注意:先将第一个拦截器前置方法的返回值恢复为true;

将第一个拦截器的前置方法返回值设置为false:

**public** **class** MyInterceptor2 **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("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，倒序执行

\*/

@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");

}

/\*\*

\* 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)

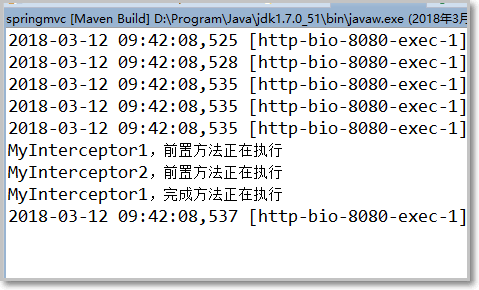
**throws** Exception {

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

}

}

执行测试：执行完连个前置方法后，倒叙执行完成方法，由于第二个连接器的前置方法返回false,因此自定义处理器不再执行，然后开始倒叙执行完成方法，并且没有机会执行第二个拦截器的完成方法，而是从第一个拦截器的完成方法开始执行。



# 其他：配置Post乱码过滤器

友情提示：记得将拦截器放行或者注释掉。

在使用Post提交请求参数的时候，一旦参数是中文的，那么controller中获取到的数据数据就会乱码

## 步骤一：编写自定义处理器

@RequestMapping("show36")

**public** String test36(@RequestParam("name") String name) {

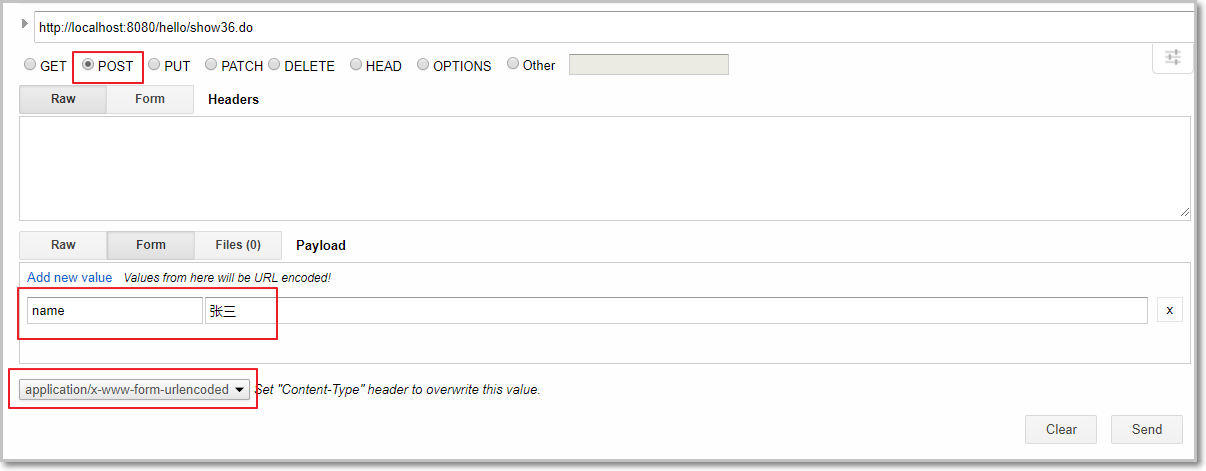
System.***out***.println(name);

**return** "hello";

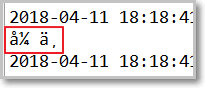
}

## 步骤二：测试发送请求

问题：如果此时提交的数据是中文，那么在后台接收到的数据就是乱码。



后台打印如下:乱码了



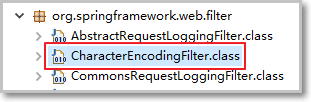
解决方案:

1、可以配置一个全站乱码过滤器。

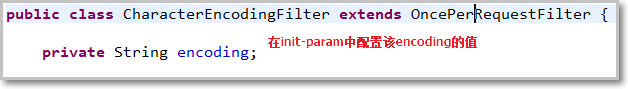
2、手动配置post请求编码过滤器

Spring-web给我们提供了解决post请求编码的过滤器：





编码过滤器：



<!--配置post请求的编码过滤器-->

<filter>

<filter-name>encodingFilter</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>

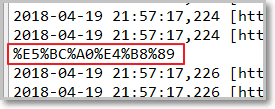
<filter-name>encodingFilter</filter-name>

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

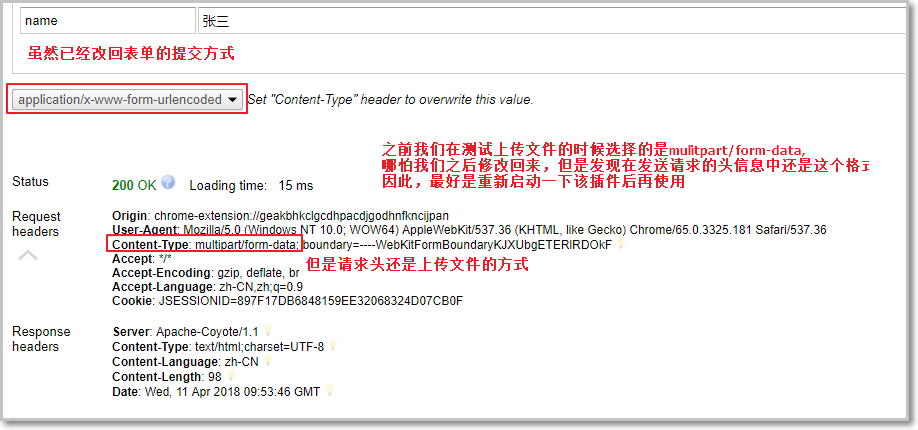
</filter-mapping>

## 注意事项

如果有的同学发现打印出来的乱码如下：



那是因为chrome插件的一个bug引起的：重新启动chrome插件即可



2、