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、spring 的拦截器

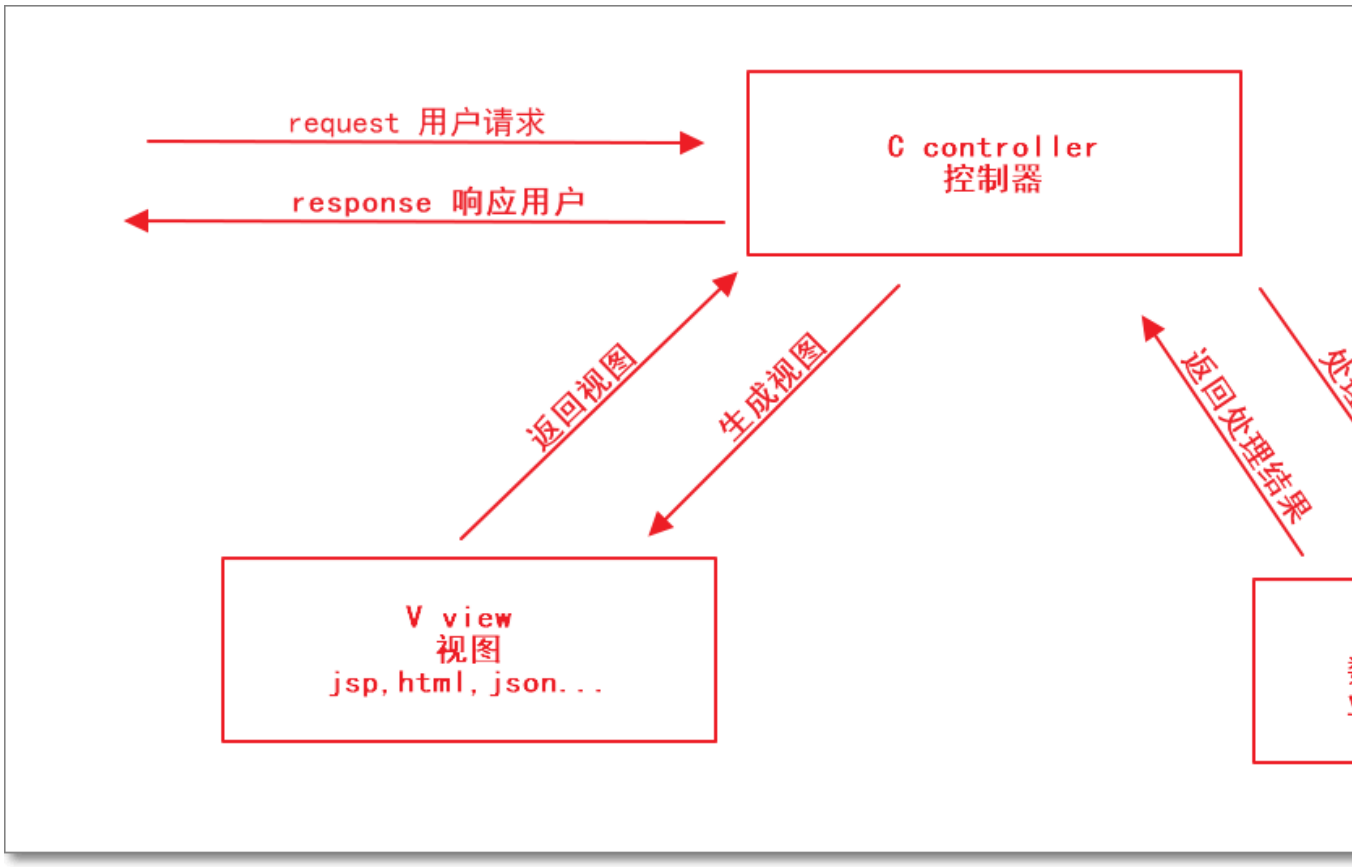
1.认识 Springmvc

1.1. 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



1.2. SpringMVC 介绍

大部分java应用都是web应用，展现层是web应用最为重要的部分。Spring为展现层提供了一个优秀的web框架——Spring MVC。和众多其他web框架一样，它基于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框架更具扩展性和灵活性。

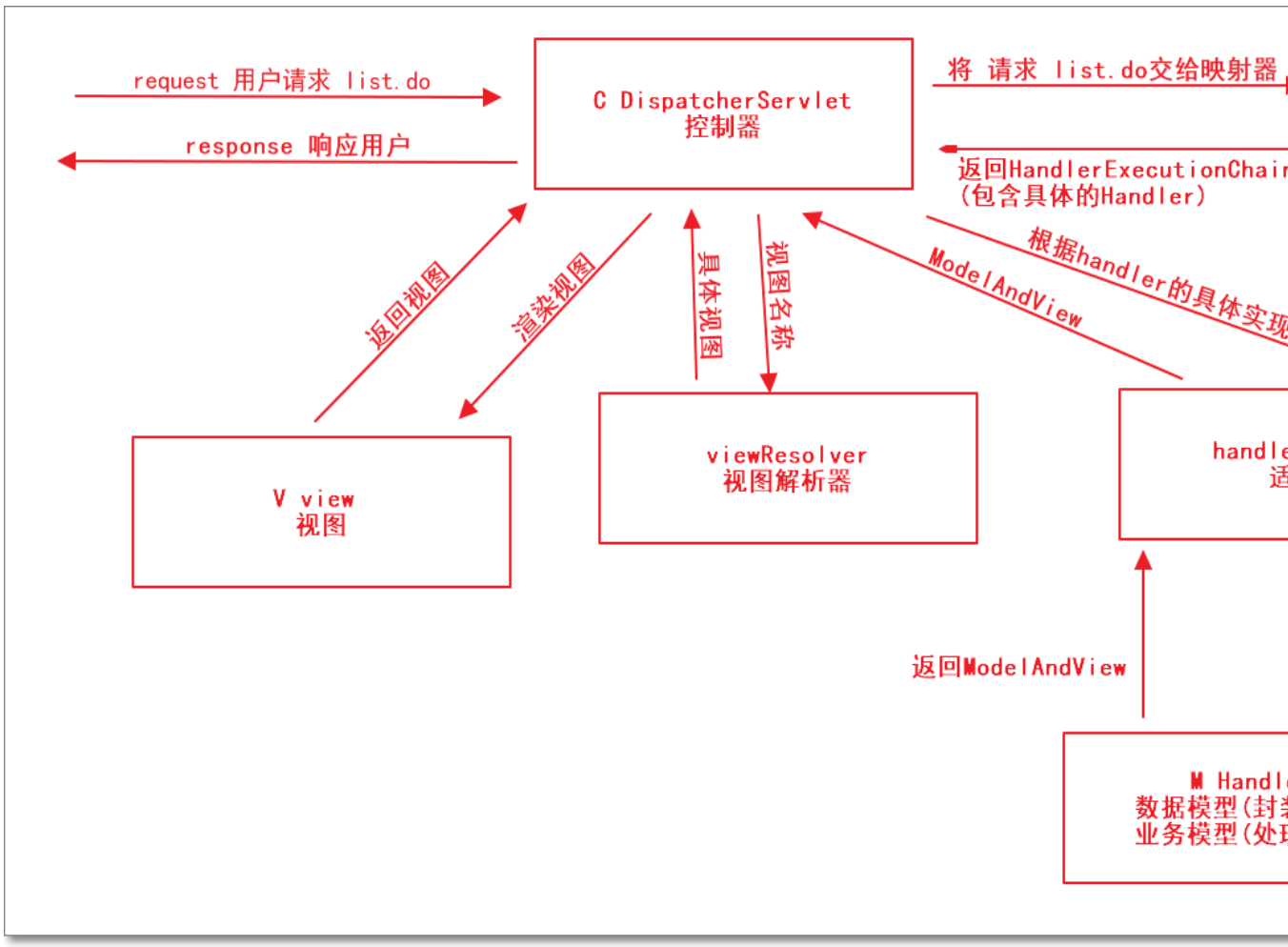
SpringMVC通过一套MVC注解，让POJO成为处理请求的处理器，无需实现任何接口，同时，SpringMVC还支持REST风格的URL请求。

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、灵活性等方面全面超越了Struts、WebWork等MVC框架，从原来的追赶者一跃成为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，它负责截获请求并将其分派给相应的处理器处理。

Springmvc 架构



1. 用户发送请求到 DispatcherServlet 控制器

2. DispatcherServlet 控制器根据请求路径到 HandlerMapping 映射器查询具体的 handler 处理器

3. HandlerMapping 映射器根据用户请求查找与之对应的 HandlerExecutionChain 执行链再回传给 DispatcherServlet 控制器

4. DispatcherServlet 控制器根据 handler 具体的实现方式调用 HandlerAdapter 适配器

5. HandlerAdapter 适配器调用具体的 handler 处理器处理业务并返回 ModelAndView 到 DispatcherServlet 控制器

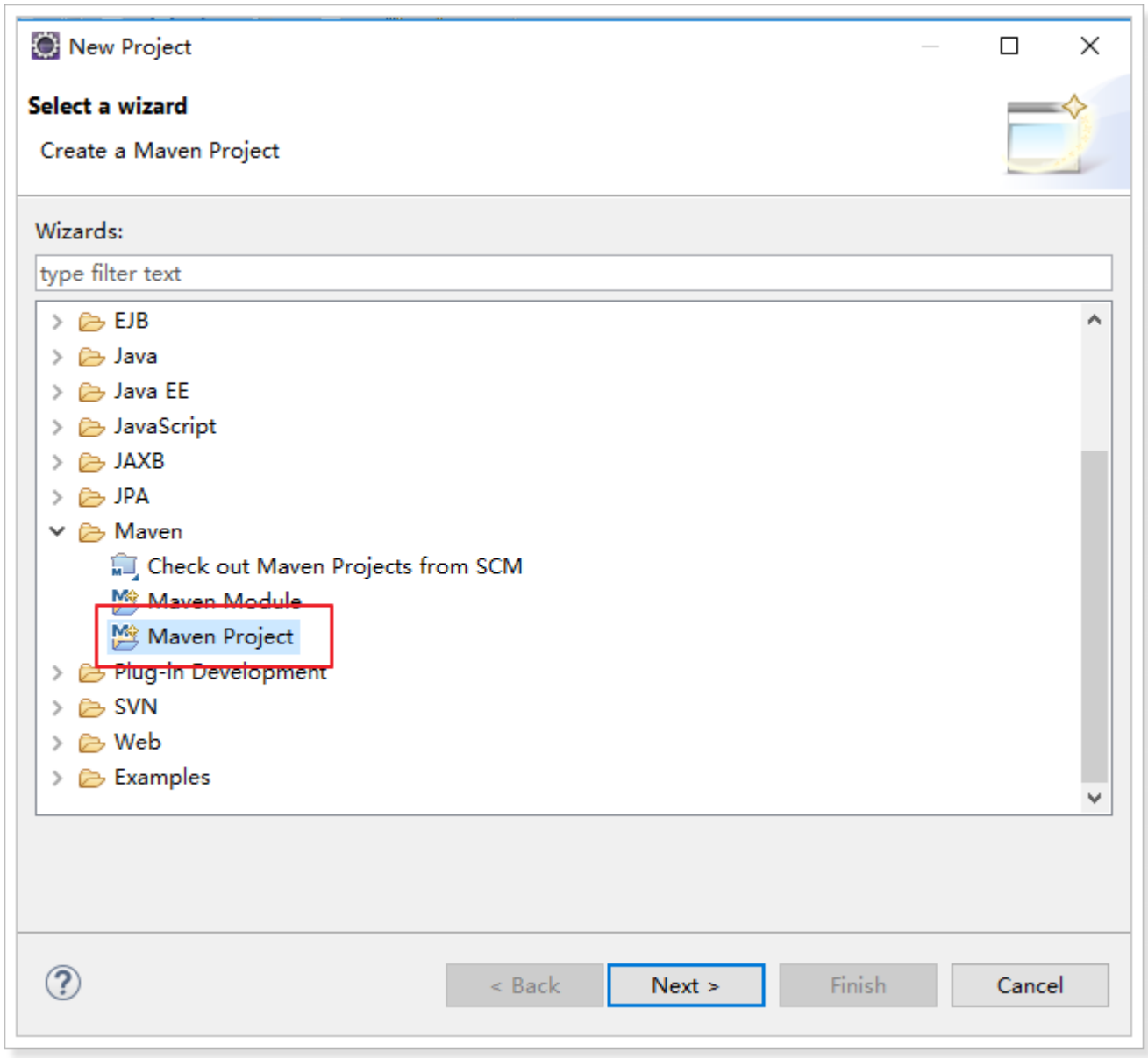
6. DispatcherServlet 控制器将 ModelAndView 专递到 ViewResolver 视图解析器

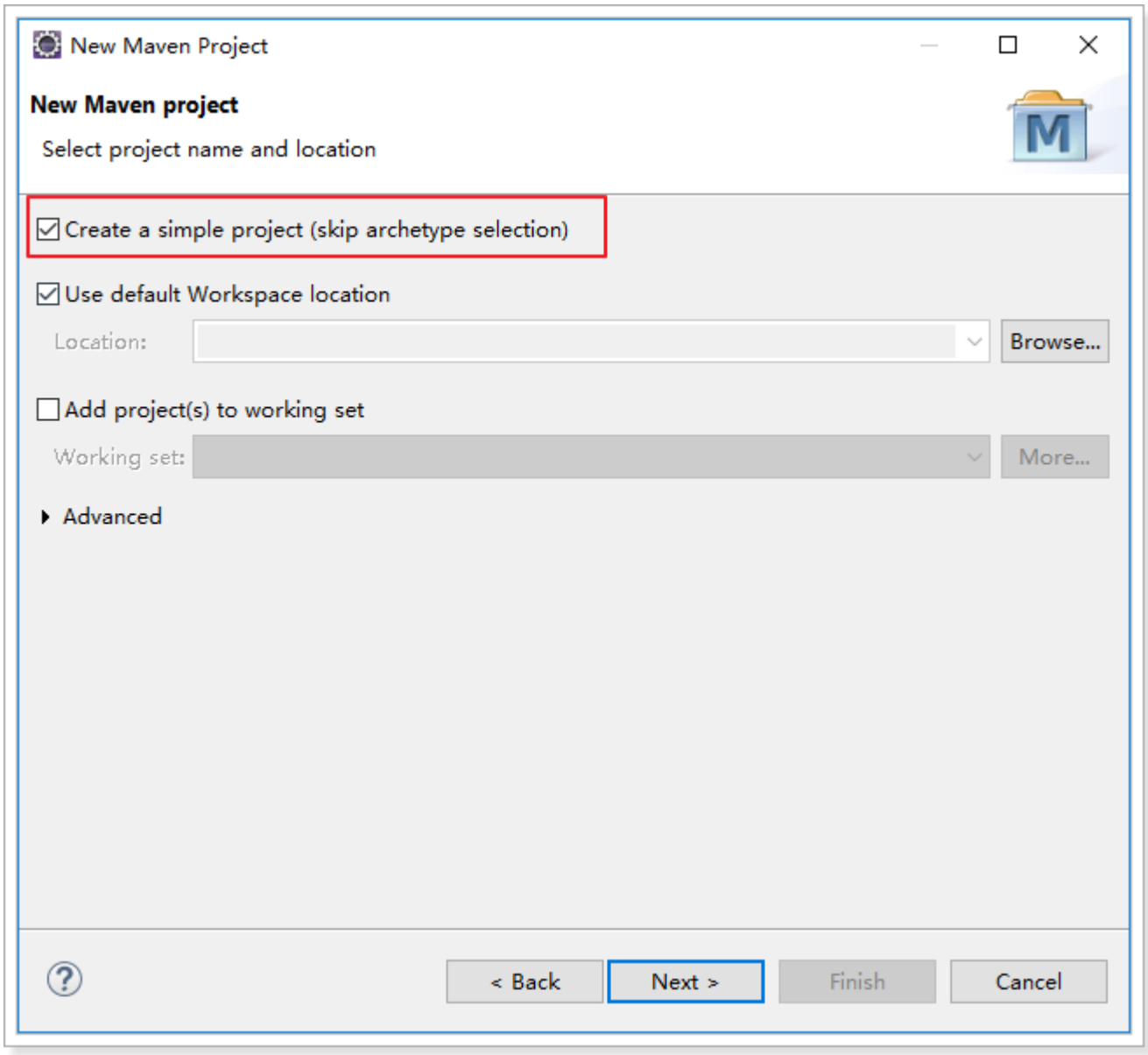
7. ViewResolver 视图解析器 返回具体的视图到 DispatcherServlet 控制器

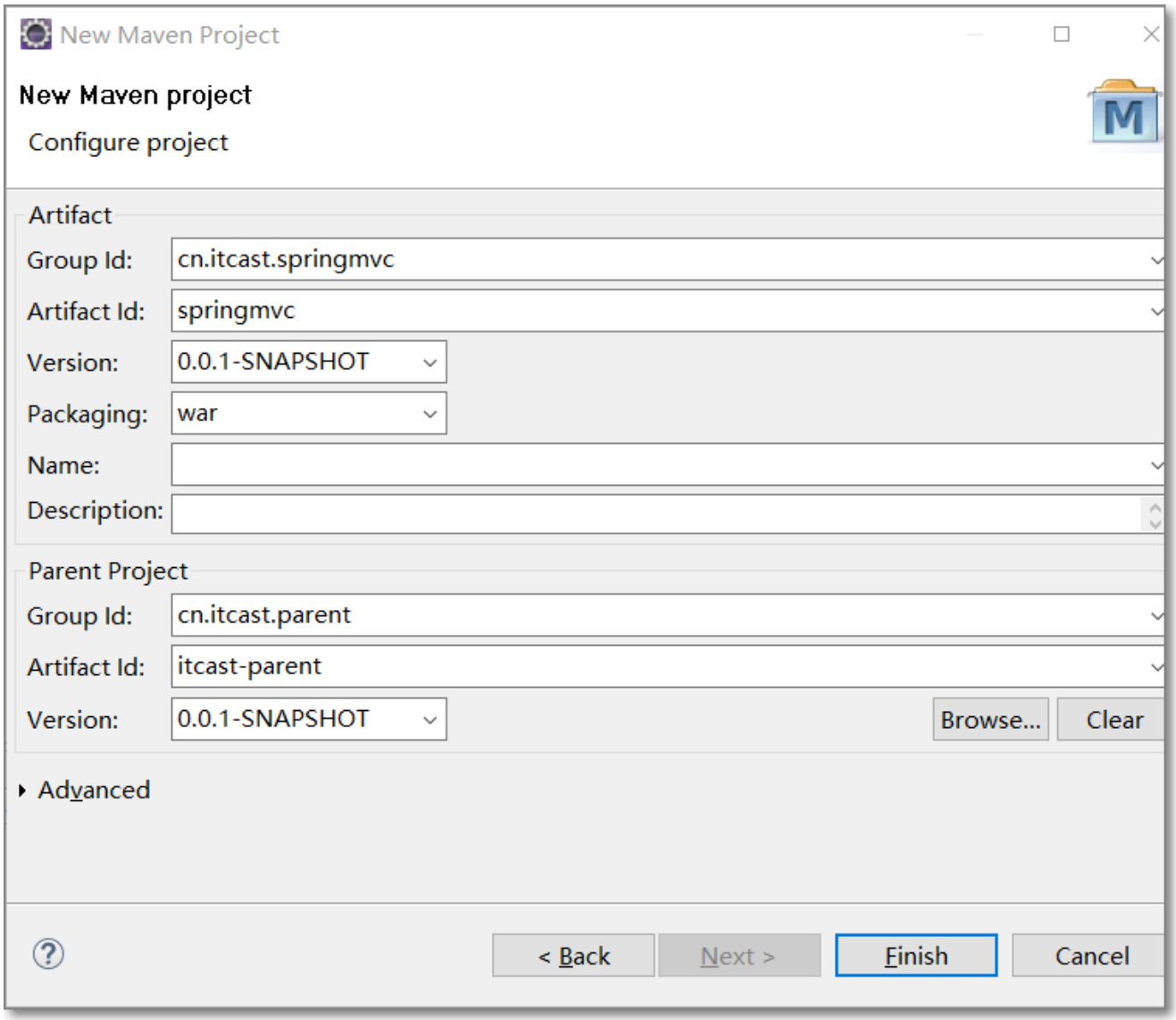
8. DispatcherServlet 控制器渲染视图后响应给用户

2.第一个 springmvc 程序(Hello World)

2.1. 创建工程




New Maven Project
New Maven project
Select project name and location
Create a simple project (skip archetype selection)
Use default Workspace location
Location:
Browse...
Add project(s) to working set
Working set:
More...
Advanced
< Back
Next >
Finish
Cancel



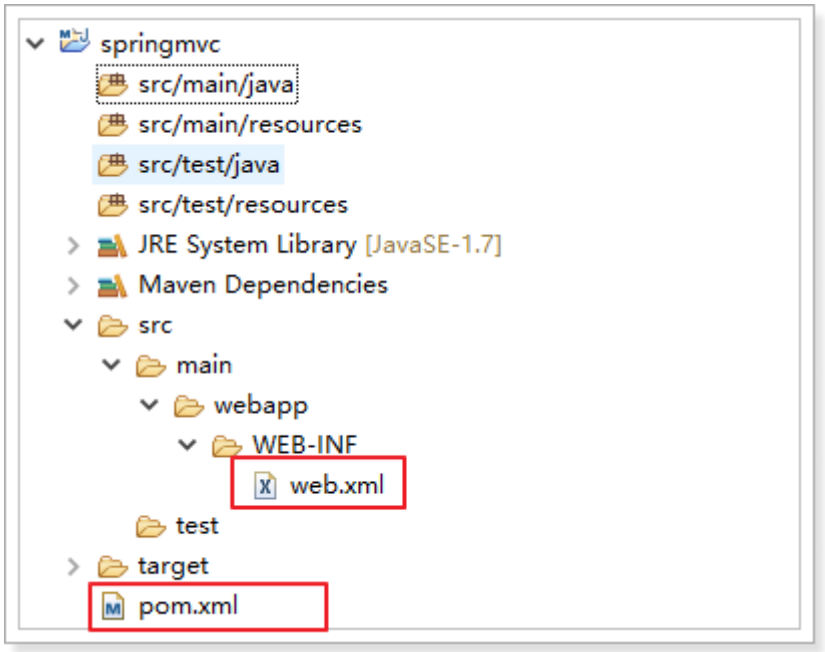
2.2. 引入依赖

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>
```

2.3. 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</servl
et-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>
```

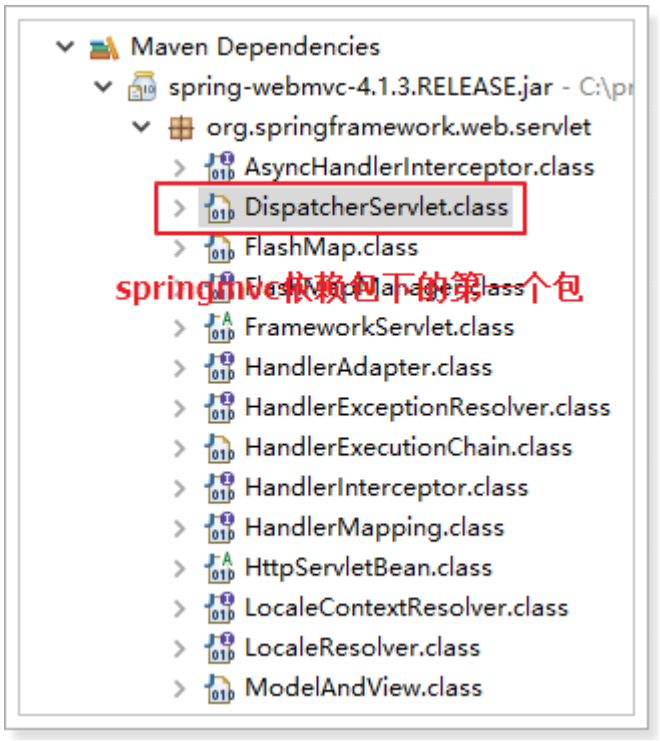
2.4. springmvc 的配置文件

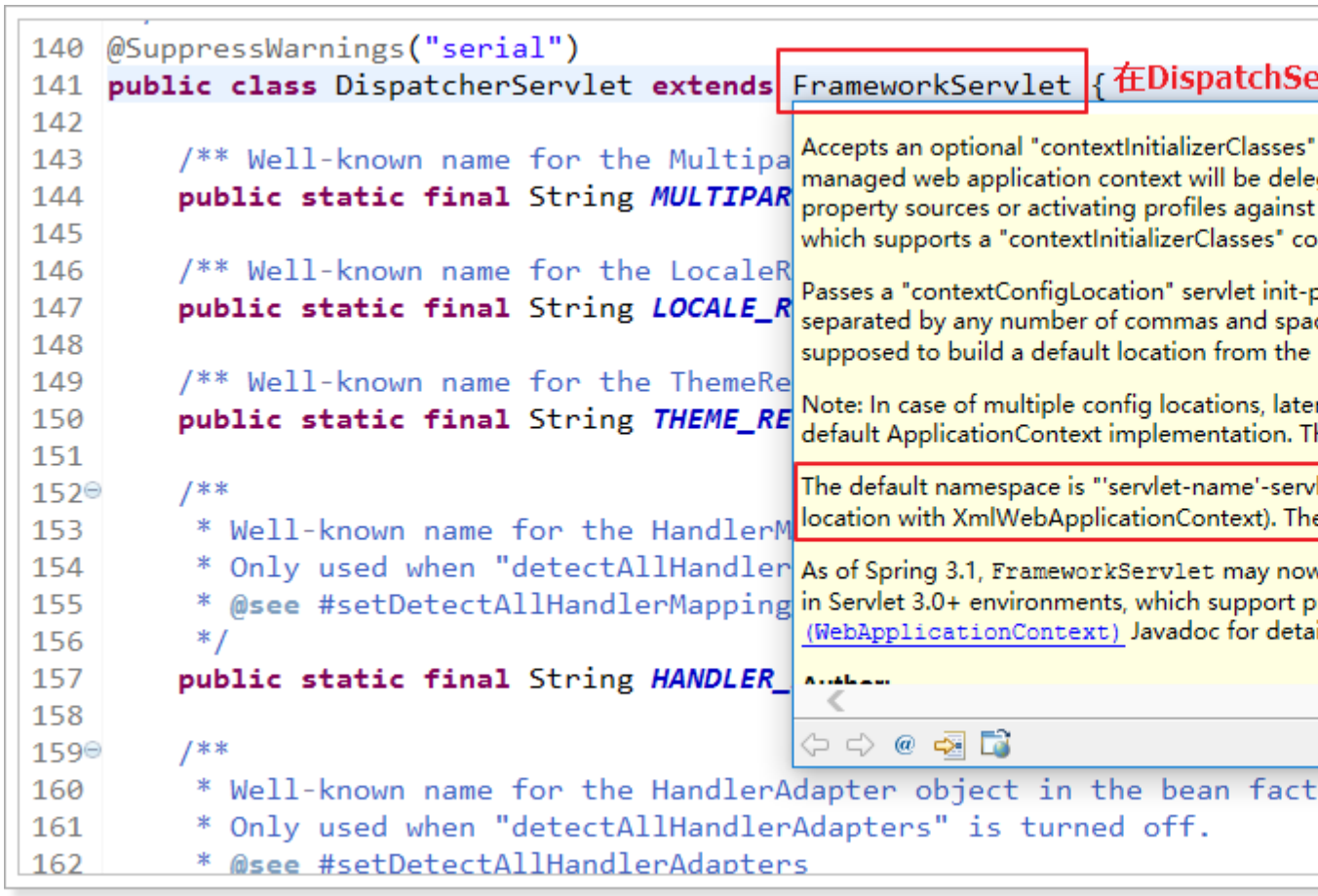
2.4.1. {servlet-name}-servlet.xml

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:

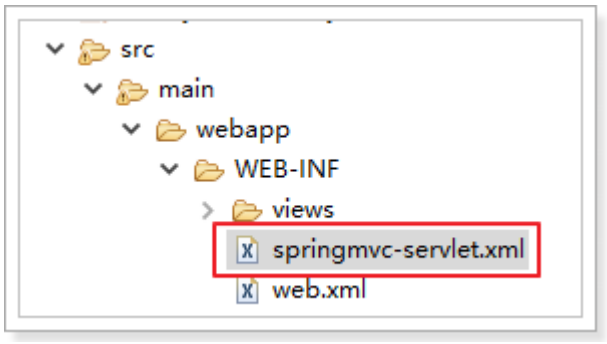




由此知道，springmvc 默认读取/WEB-INF/{servlet-name}-servlet.xml 这个配置文件，因

为我们在 web.xml 中的 servlet-name 配置的是 springmvc，所以在 WEB-INF 目录下创

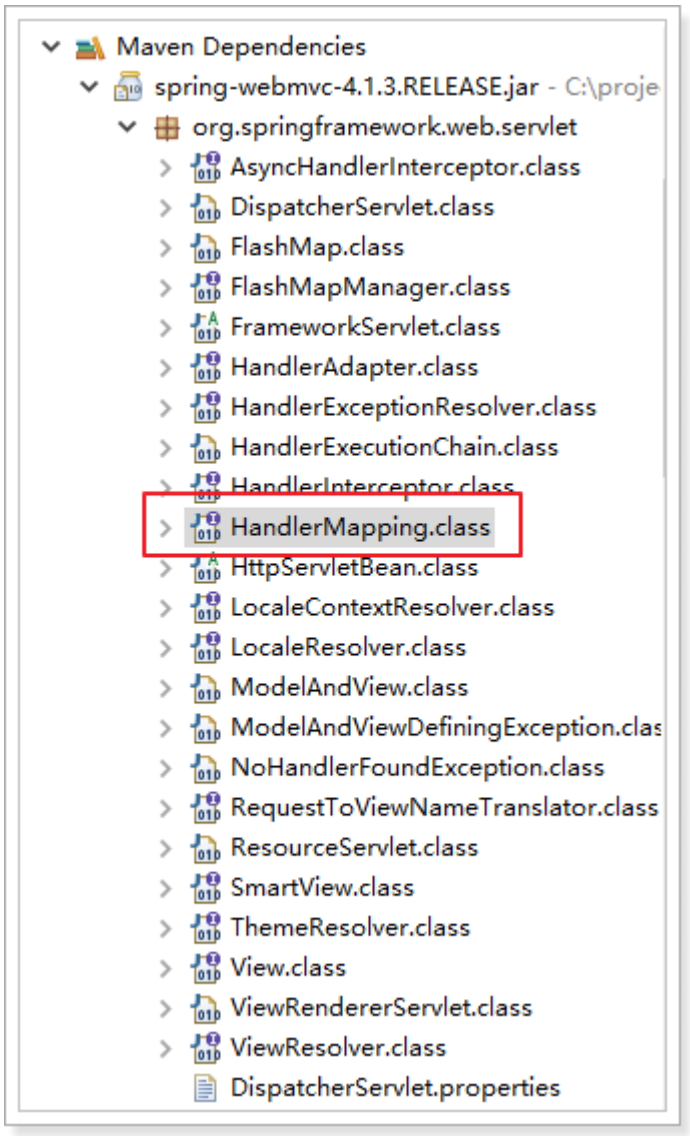
建 springmvc-servlet.xml 文件：



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
   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://ww
       http://www.springframework.org/schema/mvc http://www.springframework.
       http://www.springframework.org/schema/context http://www.springframew

</beans>
```

2.4.2. HandlerMapping 映射器



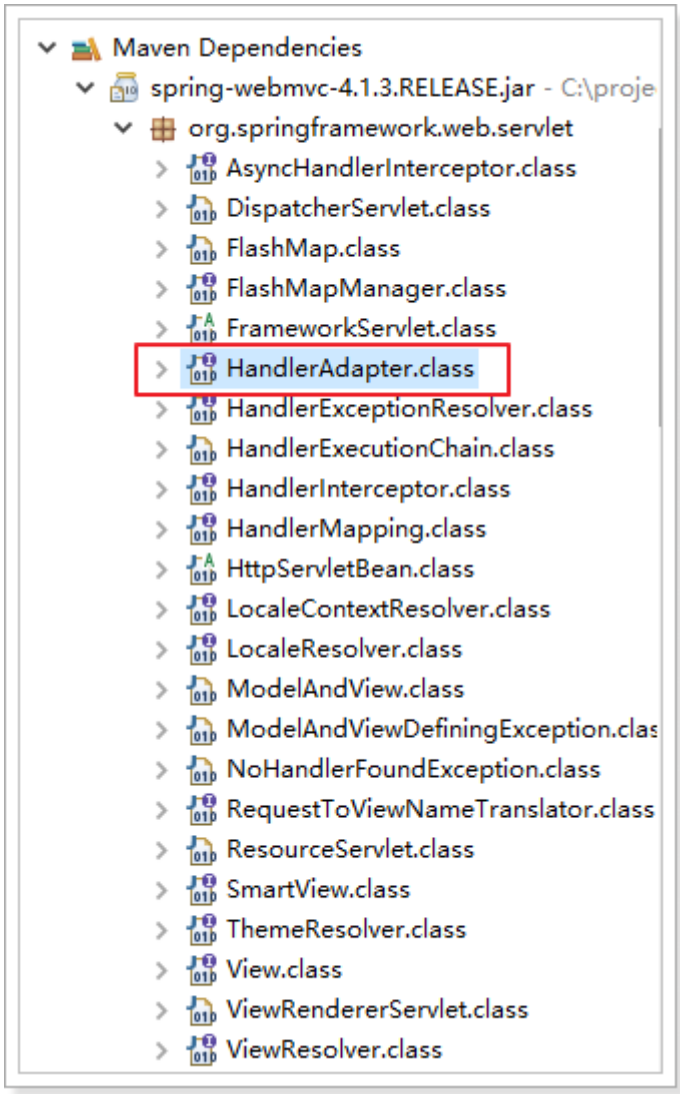
```
54 public interface HandlerMapping {
55
56     /**
57      * Name of th
58      * within the
59      * relevant U
60      * <p>Note: T
61      * HandlerMap
62      * typically
63      * this reque
64      */
65     String PATH_I
66
67     /**
68      * Name of th
69      * best match
70      * <p>Note: T
71      * HandlerMap
72      * typically
73      * this reque
```

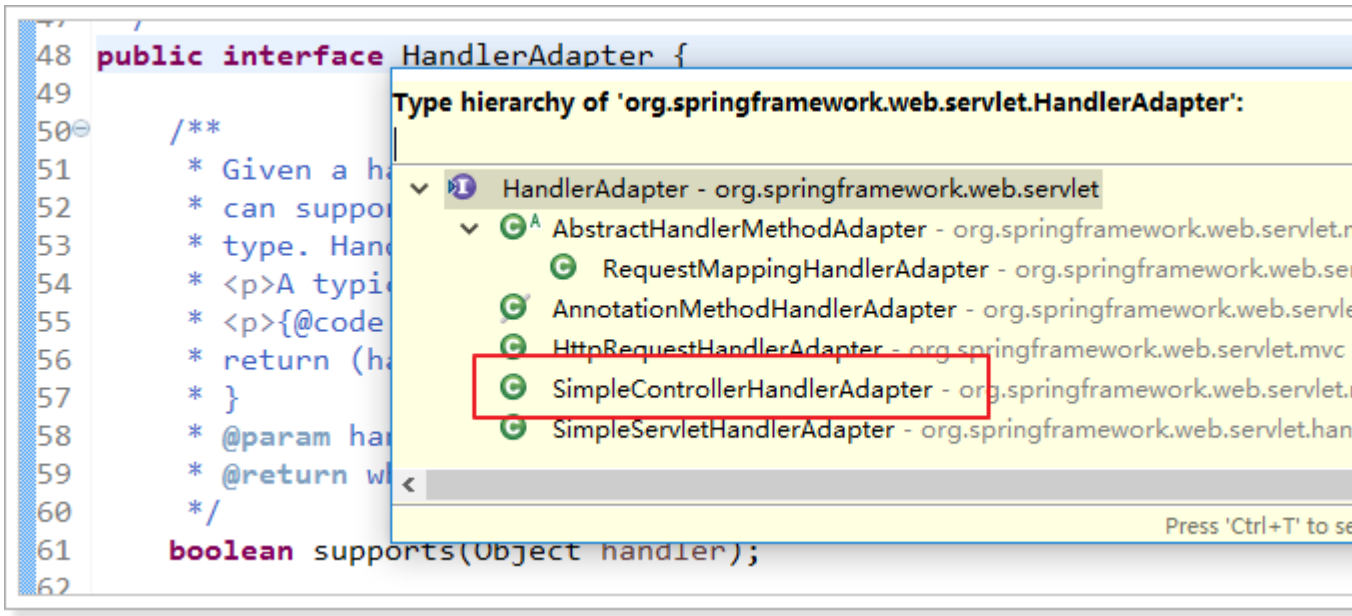
Type hierarchy of 'org.springframework.web.servlet.HandlerMapping':

- HandlerMapping - org.springframework.web.servlet
  - AbstractHandlerMapping - org.springframework.web.servlet.handler
    - AbstractHandlerMethodMapping<T> - org.springframework.we
      - RequestMappingInfoHandlerMapping - org.springframewo
        - RequestMappingHandlerMapping - org.springframewo
    - AbstractUrlHandlerMapping - org.springframework.web.servlet.
      - AbstractDetectingUrlHandlerMapping - org.springframeworo
        - AbstractControllerUrlHandlerMapping - org.springfram
          - ControllerBeanNameHandlerMapping - org.springf
          - ControllerClassNameHandlerMapping - org.springf
          - BeanNameUrlHandlerMapping - org.springframework.
          - DefaultAnnotationHandlerMapping - org.springframewo
      - SimpleUrlHandlerMapping - org.springframework.web.servl
  - EmptyHandlerMapping - org.springframework.web.servlet.confi

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

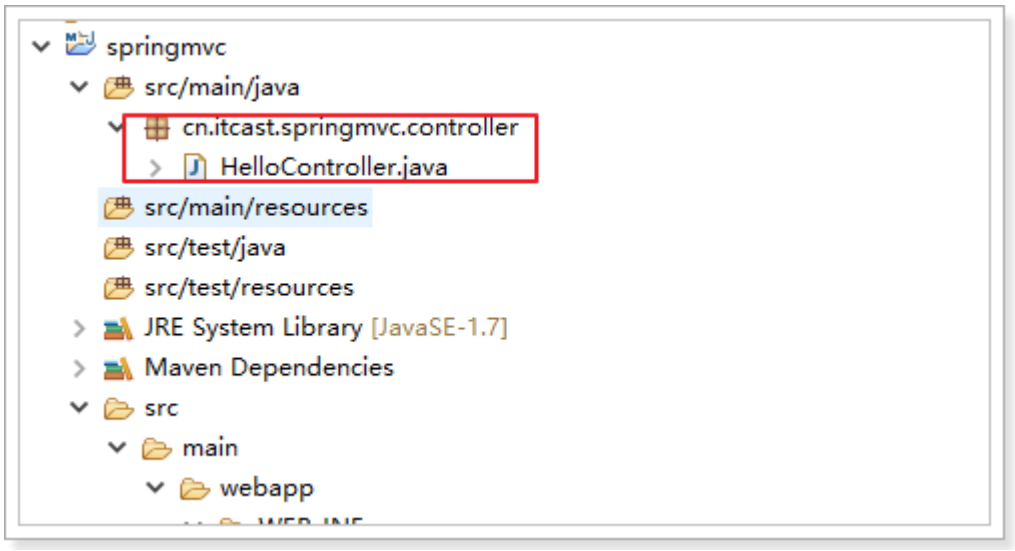
2.4.3. HandlerAdapter 适配器





```
<!-- 配置 HandlerAdapter -->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHa
ndlerAdapter" />
```

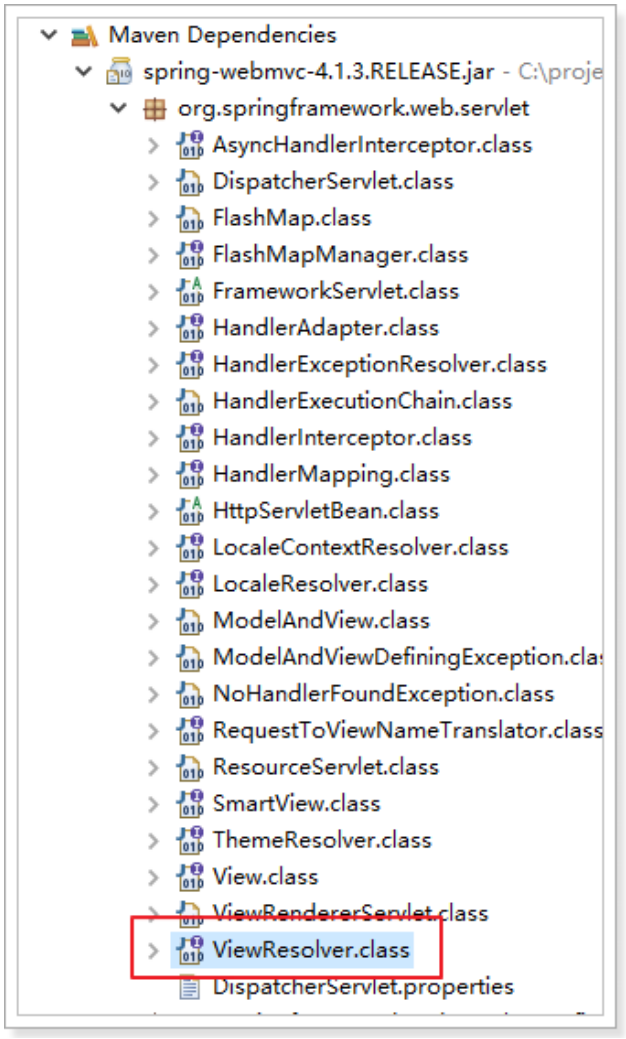
2.4.4. HelloController



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");
         mv.addObject("msg","springmvc的第一个程序");
        return mv;
    }
}
```

2.4.5. ViewResolver 视图解析器



```
public interface ViewResolver {

    /**
     * Resolve t
     * <p>Note:
     * return {@
     * However,
     * to build
     * (rather t
     * @param vi
     * @param lo
     * ViewResol
     * @return t
     * (optional
     * @throws E
     * (typicall
     */
    View resolve

}
```

Type hierarchy of 'org.springframework.web.servlet.ViewResolver':

- ResourceBundleViewResolver - org.springframework.web.servlet.view
- UrlBasedViewResolver - org.springframework.web.servlet.view
  - AbstractTemplateViewResolver - org.springframework.web.servlet.view
    - FreeMarkerViewResolver - org.springframework.web.servlet.view.freemarker
    - GroovyMarkupViewResolver - org.springframework.web.servlet.view.groovy
    - VelocityViewResolver - org.springframework.web.servlet.view.velocity
      - VelocityLayoutViewResolver - org.springframework.web.servlet.view.veloc
  - InternalResourceViewResolver - org.springframework.web.servlet.view
  - JasperReportsViewResolver - org.springframework.web.servlet.view.jasperreports
  - TilesViewResolver - org.springframework.web.servlet.view.tiles3
  - TilesViewResolver - org.springframework.web.servlet.view.tiles2
  - XsltViewResolver - org.springframework.web.servlet.view.xslt
- XmlViewResolver - org.springframework.web.servlet.view
- BeanNameViewResolver - org.springframework.web.servlet.view
- ContentNegotiatingViewResolver - org.springframework.web.servlet.view

Press 'Ctrl+T' to see t

```
public class InternalResourceViewResolver extends UrlBasedViewResolver {

    private static final boolean jstlPre
            "javax.servlet.jsp.jstl.core

    private Boolean alwaysInclude;


    /**
     * Sets the default {@link #setViewC
     * by default {@link InternalResourc
     * is present.
     */
    public InternalResourceViewResolver(
        Class<?> viewClass = requiredVie
        if (viewClass.equals(InternalRes
            viewClass = JstlView.class;
        }
        setViewClass(viewClass);
    }
```

在内部资源视图解析器的文档的注释中

org.springframework.web.servlet.view.freemarker.FreeMarkerView. The view class for all views generated by this resolver can be specified via the "viewClass" property.

View names can either be resource URLs themselves, or get augmented by a specified prefix and/or suffix. Exporting an attribute that holds the RequestContext to all views is explicitly supported.

Example: prefix="/WEB-INF/jsp/", suffix=".jsp", viewname="test" -> "/WEB-INF/jsp/test.jsp"

As a special feature, redirect URLs can be specified via the "redirect:" prefix. E.g.: "redirect:myAction.do" will trigger a redirect to the given URL, rather than resolution as standard view name. This is typically used for redirecting to a controller URL after finishing a form workflow.

Furthermore, forward URLs can be specified via the "forward:" prefix. E.g.: "forward:myAction.do" will trigger a forward to the given URL, rather than resolution as standard view name. This is typically used for controller URLs; it is not supposed to be used for JSP URLs - use logical view names there.

Note: This class does not support localized resolution, i.e. resolving a symbolic view name to different resources depending on the current locale.

Note: When chaining ViewResolvers, a UrlBasedViewResolver will check whether the specified resource actually exists. However, with InternalResourceView it is not

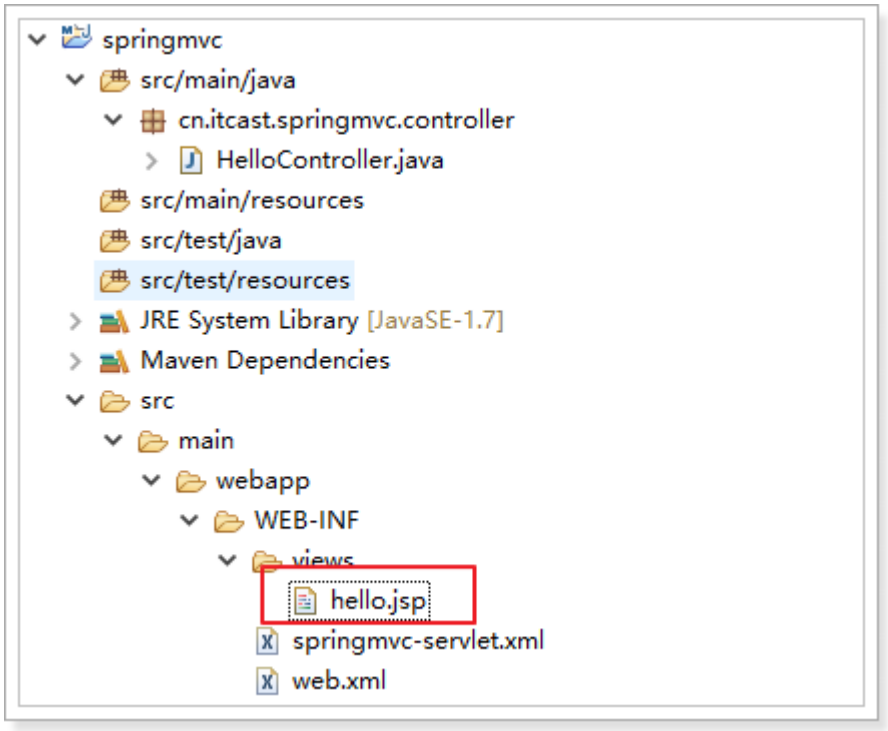
由此可见，视图解析器的规则是：prefix+viewName+suffix

2.4.6. 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
    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.
        http://www.springframework.org/schema/context http://www.springframew
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.BeanNameUrlHandlerMa

    <!-- 配置适配器 -->
    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerA

    <bean name="/hello.do" class="cn.itcast.springmvc.controller.HelloControl

    <!-- 配置视图解析器 -->
    <!-- Example: prefix="/WEB-INF/jsp/", suffix=".jsp", viewname="test" -> "
    <bean class="org.springframework.web.servlet.view.InternalResourceViewRes
       <property name="prefix" value="/WEB-INF/views/"></property>
       <property name="suffix" value=".jsp"></property>
    </bean>

</beans>
```

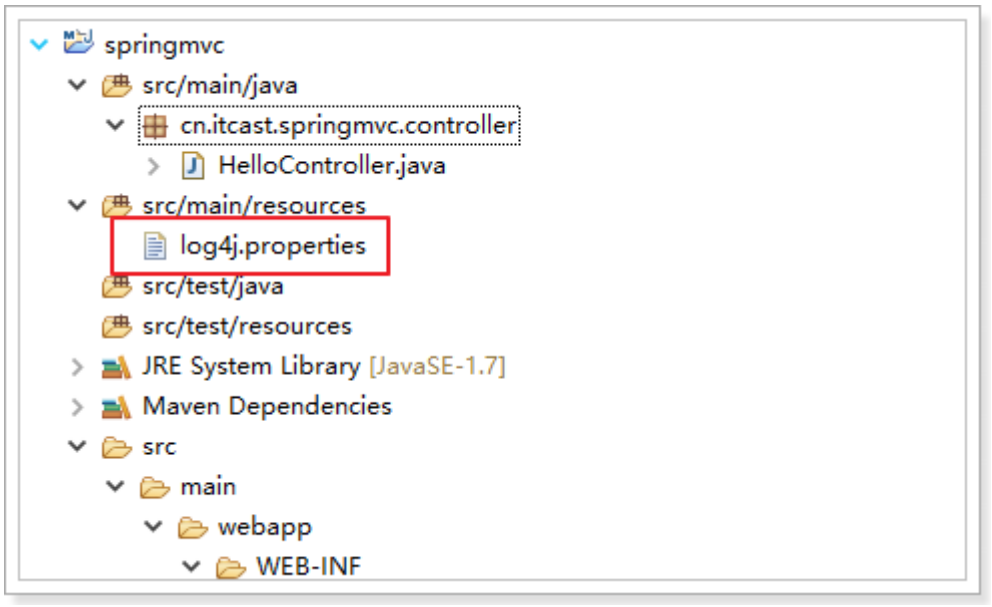
2.5. 添加 jsp 页面（hello.jsp）



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>
```

2.6. 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
```

2.7. 日志打印信息

```
2017-04-13 21:20:48,070 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[D
2017-04-13 21:20:48,082 [http-bio-8080-exec-1] [org.springframework.web.servlet.handler.BeanNameUrlHa
[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]-[D
2017-04-13 21:20:48,089 [http-bio-8080-exec-1] [org.springframework.beans.factory.support.DefaultList
2017-04-13 21:20:48,090 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[D
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
2017-04-13 21:20:48,107 [http-bio-8080-exec-1] [org.springframework.web.servlet.view.JstlView]-[DEBUG
2017-04-13 21:20:48,233 [http-bio-8080-exec-1] [org.springframework.web.servlet.DispatcherServlet]-[D
```

2.8. 流程分析

springmvc-day03/pom.xml | web.xml | springmvc-servlet.xml | HelloController.java

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=
  <display-name>springmvc-day03</display-name>

  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet<
    <!-- The default namespace is "'servlet-name'-servlet", e.g. "tes
    (leading to a "/WEB-INF/test-servlet.xml"  -->
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <!--
        /*: 拦截所有请求，并且拦截jsp （filter servlet）
        /:拦截所有请求，但是不包括jsp（默认的servlet）
        *.do *.action *.html *.htm
     -->
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
```

1 用户发送/hello.do请求，满足*.do的路径
进入springmvc（DispatchServlet）

springmvc-day03/pom.xml | web.xml | springmvc-servlet.xml

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

    <!-- 配置HandlerMapping映射器 -
    <bean class="org.springframework.web.servlet.handler.BeanNameUrlHan

    <!-- 配置HandlerAdapter适配器 -->
    <bean class="org.springframework.web.servlet.mvc.SimpleControllerHa
    <bean name="/hello.do" class="cn.itcast.springmvc.controller.helloC

    <!-- /WEB-INF/views/hello.jsp -->
    <bean
        class="org.springframework.web.servlet.view.InternalResourceVie
        <property name="prefix" value="/WEB-INF/views/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

</beans>
```

2. DispatchServlet根据请求路径到HandlerMapping（BeanNameUrlHan

4.根据Handler的实现方式不同，调用对应的HandlerAdapt

7.Handler给Dispatc

3.根据请求路径（"/hello.do"）找bean的name为"/hello.do"的bean
结合日志得知返回值是一个HandlerExecutionChain（包含HelloController以及

8.DispatchServlet根据视图名称（

9.视图解析器根据prefix+viewName+suffix的解析过程，获得

```java
public class HelloController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request, Http
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg", "springmvc第一个程序! ");
        mv.setViewName("hello");
        return mv;
    }

}
```

5.HandlerAdapter调用HelloController处理具体的业务逻辑

6.HelloController处理完业务逻辑返回一个ModelAndV
（视图名称：hello，数据模型：msg）

2.9. 优化 helloworld 程序

2.9.1. web.xml

```
5   <servlet>
6     <servlet-name>springmvc</servlet-name>
7     <servlet-class>org.springframework.web.servlet.DispatcherServlet</
8     <!-- The default namespace is "'servlet-name'-servlet", e.g. "test
9     (leading to a "/WEB-INF/test-servlet.xml"  -->
10    <load-on-startup>1</load-on-startup>
11  </servlet>
12  <servlet-mapping>
13    <servlet-name>springmvc</servlet-name>
14    <!--
15        /*: 拦截所有请求，并且拦截jsp （filter servlet）
16        /:拦截所有请求，但是不包括jsp（默认的servlet）
17        *.do *.action *.html *.htm
18     -->
19    <url-pattern>*.do</url-pattern>
20  </servlet-mapping>
```

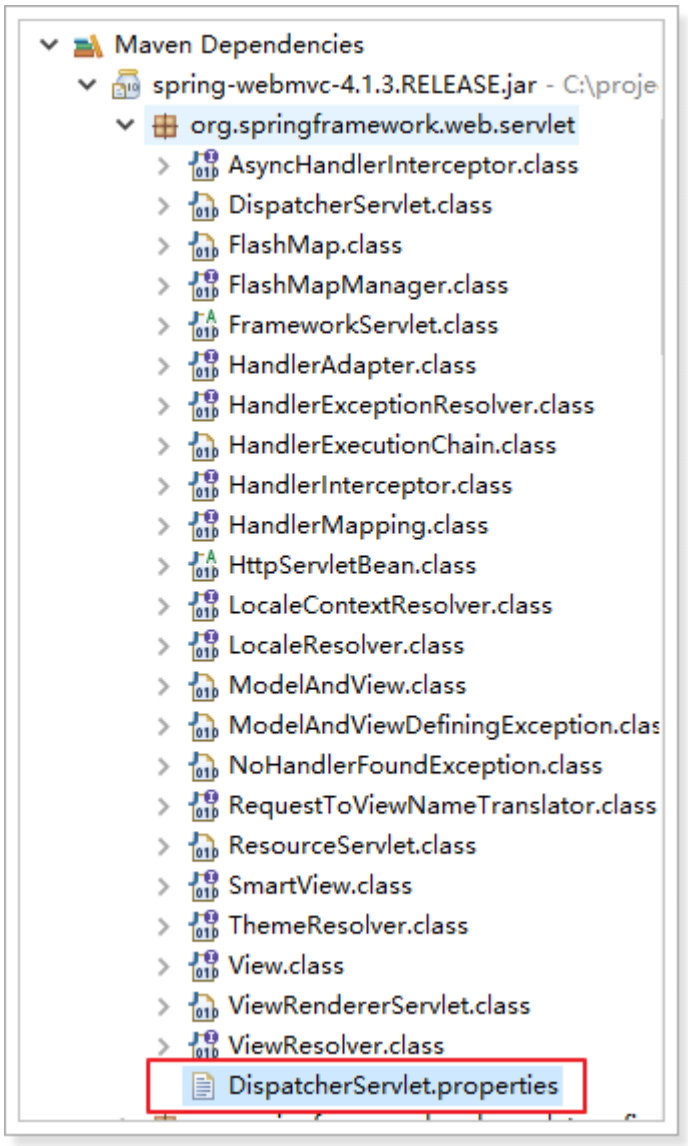
2.9.2. springmvc-servlet.xml

DispatchServlet.class 源码中：

```
245     /**
246      * Name of the class path resource (relative to the DispatcherS
247      * that defines DispatcherServlet's default strategy names.
248      */
249     private static final String DEFAULT_STRATEGIES_PATH = "DispatcherS
250
```

DispatchServlet会读取一个默认的配置文件
由这个配置的路径得知该配置文件和DispatchS
在同一个目录下

找到 DispatchServlet.properties 文件：


Maven Dependencies
spring-webmvc-4.1.3.RELEASE.jar - C:\proje
org.springframework.web.servlet
AsyncHandlerInterceptor.class
DispatcherServlet.class
FlashMap.class
FlashMapManager.class
FrameworkServlet.class
HandlerAdapter.class
HandlerExceptionResolver.class
HandlerExecutionChain.class
HandlerInterceptor.class
HandlerMapping.class
HttpServletBean.class
LocaleContextResolver.class
LocaleResolver.class
ModelAndView.class
ModelAndViewDefiningException.clas
NoHandlerFoundException.class
RequestToViewNameTranslator.class
ResourceServlet.class
SmartView.class
ThemeResolver.class
View.class
ViewRendererServlet.class
ViewResolver.class
DispatcherServlet.properties

```
1 # Default implementation classes for DispatcherServlet's strategy interface
2 # Used as fallback when no matching beans are found in the DispatcherServle
3 # Not meant to be customized by application developers.
4
5 org.springframework.web.servlet.LocaleResolver=org.springframework.web.serv
6
7 org.springframework.web.servlet.ThemeResolver=org.springframework.web.servl
8
9 org.springframework.web.servlet.HandlerMapping=org.springframework.web.serv
10     org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandler
11
12 org.springframework.web.servlet.HandlerAdapter=org.springframework.web.serv
13     org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
14     org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerA
15
16 org.springframework.web.servlet.HandlerExceptionResolver=org.springframewor
17     org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionR
18     org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionReso
19
20 org.springframework.web.servlet.RequestToViewNameTranslator=org.springframe
21
22 org.springframework.web.servlet.ViewResolver=org.springframework.web.servle
23
24 org.springframework.web.servlet.FlashMapManager=org.springframework.web.ser
```

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

```
<!-- 配置HandlerMapping映射器 -->
<!-- <bean class="org.springframework.web.servlet.handler.BeanNameUrlHar

<!-- 配置HandlerAdapter适配器 -->
<!-- <bean class="org.springframework.web.servlet.mvc.SimpleControllerHa

<bean name="/hello.do" class="cn.itcast.springmvc.controller.HelloContro

<!-- /WEB-INF/views/hello.jsp -->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewReso
    <property name="prefix" value="/WEB-INF/views/"></property>
    <property name="suffix" value=".jsp"></property>
</bean>
```

再次测试：

2.10. 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

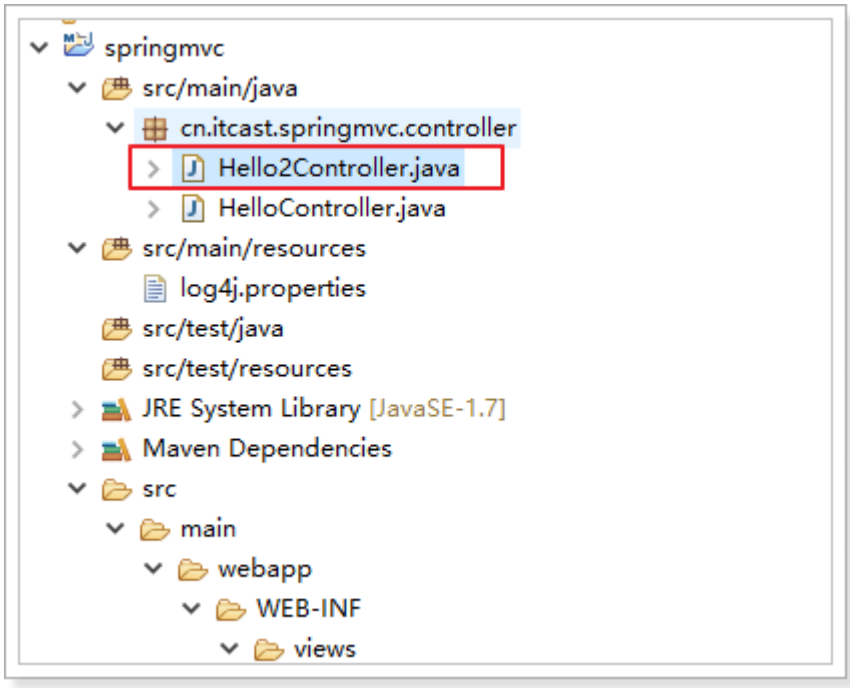
3.注解

3.1. 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

```
1 # Default implementation classes for DispatcherServlet's strategy interfaces.
2 # Used as fallback when no matching beans are found in the DispatcherServlet context
3 # Not meant to be customized by application developers.
4
5 org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.
6
7 org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.
8
9 org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handl
10     org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping
11
12 org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.H
13     org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
14     org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
15
16 org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.ser
17     org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
18     org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver
19
20 org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.
21
22 org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.In
23
24 org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.supp
```

3.1.1. 创建 hello2Controller



内容：

```java
@Controller
public class Hello2Controller {

    @RequestMapping(value="show1")
    public ModelAndView test1(){
        ModelAndView mv = new ModelAndView();
        mv.setViewName("hello");
        mv.addObject("msg", "springmvc的第一个注解程序! ");
        return mv;
    }

}
```

@Controller:表示该类为一个处理器,相当于 HelloController implements Controller

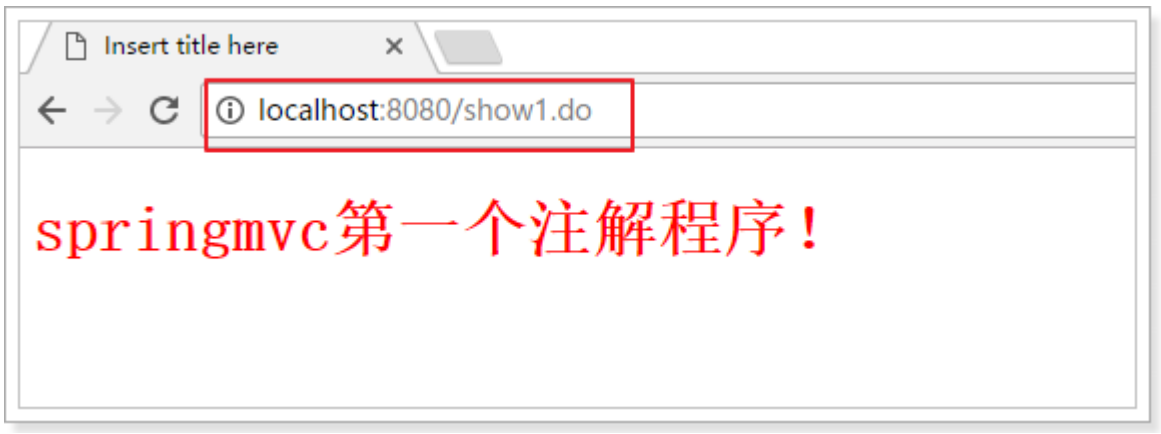
@RequestMapping(value="/show1"),配置方法的对应的 url，默认的 url 后缀和<url-pattern>*.do</url-pattern> 一致

3.1.2. 配置扫描器

在 springmvc-servlet.xml 中，开启注解扫描

```xml
<!-- 开启注解扫描，使用方式和spring一样 -->
<context:component-scan base-package="cn.itcast.springmvc.controller" />
```

3.1.3. 测试

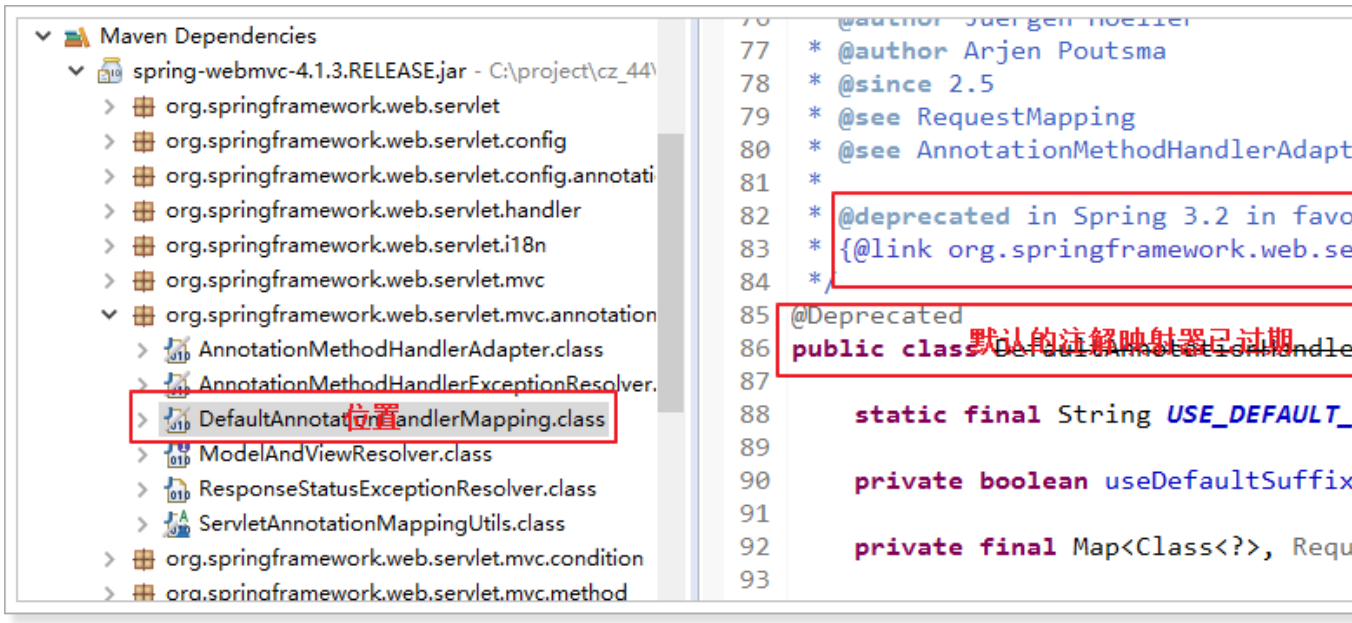


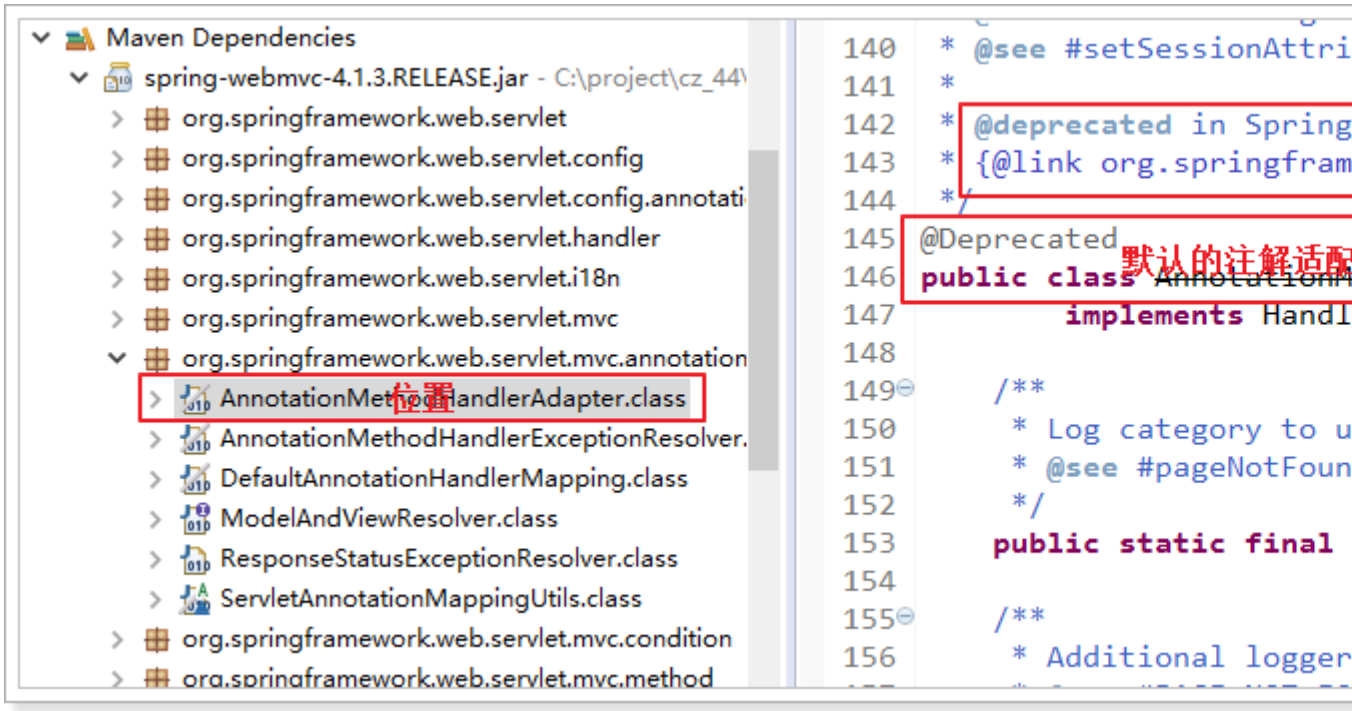
3.1.4. 日志

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.serv
'springmvc' processing GET request for [/hello/show1.do]
2017-04-14 01:10:27,140 [http-bio-8080-exec-1] [org.springframework.web.serv
patterns for request [/hello/show1.do] are [/hello/show1.*]
2017-04-14 01:10:27,142 [http-bio-8080-exec-1] [org.springframework.web.serv
Template variables for request [/hello/show1.do] are {}
2017-04-14 01:10:27,144 [http-bio-8080-exec-1] [org.springframework.web.serv
[/hello/show1.do] to HandlerExecutionChain with handler [cn.itcast.springmvc
2017-04-14 01:10:27,148 [http-bio-8080-exec-1] [org.springframework.web.serv
[/hello/show1.do] is: -1
2017-04-14 01:10:27,169 [http-bio-8080-exec-1] [org.springframework.web.binc
handler method: public org.springframework.web.servlet.ModelAndView cn.itcas
2017-04-14 01:10:27,174 [http-bio-8080-exec-1] [org.springframework.beans.fa
afterPropertiesSet() on bean with name 'hello'
2017-04-14 01:10:27,174 [http-bio-8080-exec-1] [org.springframework.web.serv
[org.springframework.web.servlet.view.JstlView: name 'hello'; URL [/WEB-INF/
2017-04-14 01:10:27,174 [http-bio-8080-exec-1] [org.springframework.web.serv
[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.serv
INF/views/hello.jsp] in InternalResourceView 'hello'
2017-04-14 01:10:27,291 [http-bio-8080-exec-1] [org.springframework.web.serv
```

3.1.5. 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

3.2. 推荐使用的注解配置

3.2.1. springmvc-servlet.xml

```
<!-- 配置推荐使用的映射器 -->
<bean class="org.springframework.web.servlet.mvc.method.annotation.Reque

<!-- 配置推荐使用的适配器 -->
<bean class="org.springframework.web.servlet.mvc.method.annotation.Reque
```

3.2.2. 测试

3.2.3. 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'
2017-04-14 01:06:38,947 [http-bio-8080-exec-1]
[org.springframework.web.servlet.DispatcherServlet]-[DEBUG] Successfully completed
request
```

3.3. 最佳方案（注解驱动）

3.3.1. 注解驱动的配置

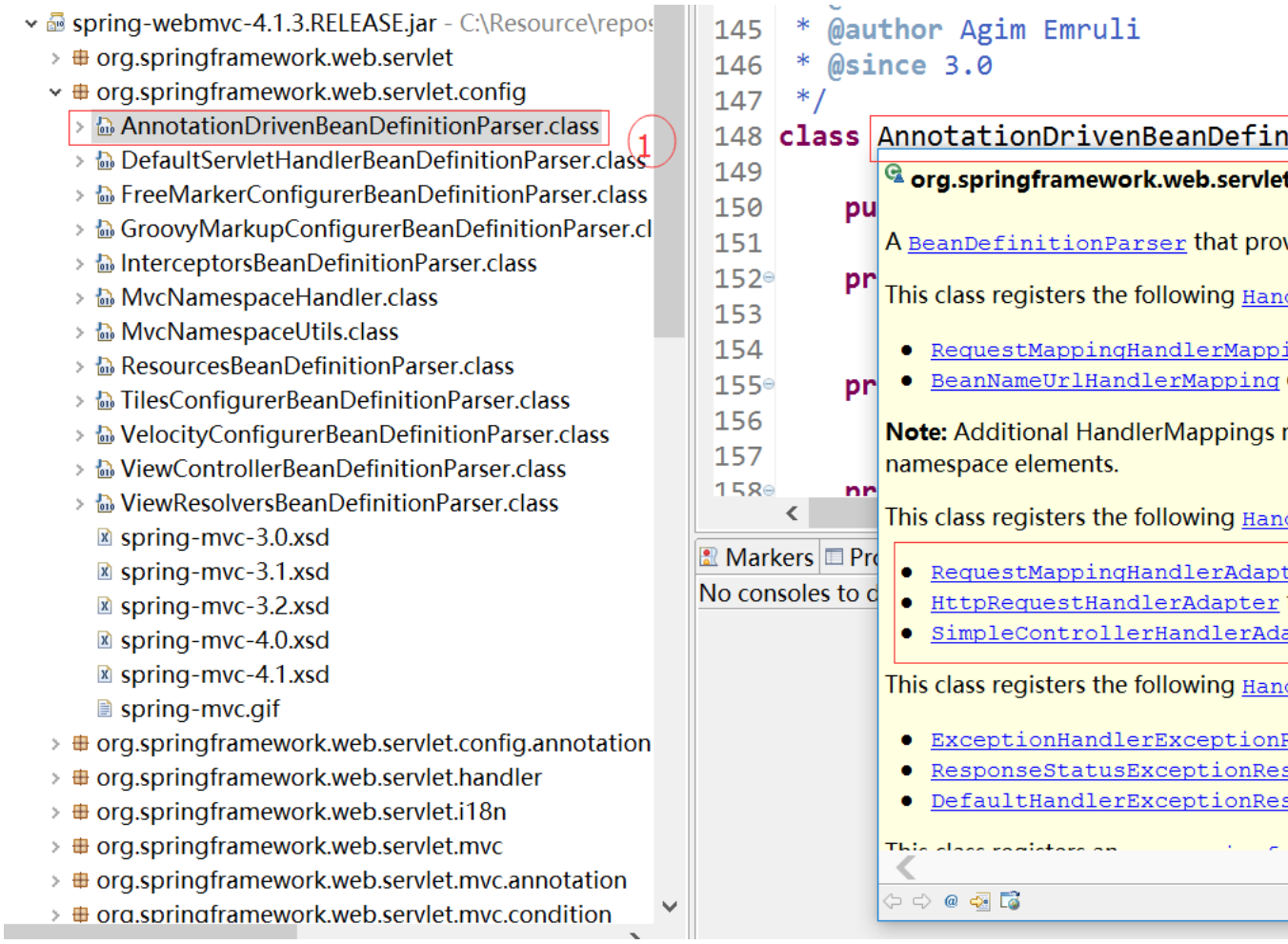
在 springmvc-servlet.xml 中配置注解驱动

<mvc:annotation-driven />

```
<!-- 注解驱动，替代推荐使用的适配器和映射器，提供对json的支持 -->
<mvc:annotation-driven />
```

3.3.2. 注解驱动的原理

AnnotationDrivenBeanDefinitionParser 的注释



3.4. 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
    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.
```

```
        http://www.springframework.org/schema/context http://www.springframew
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.BeanNameUrlHand
    <!-- 配置推荐使用的映射器 -->
    <!-- <bean class="org.springframework.web.servlet.mvc.method.annotation.R

    <!-- 配置适配器 -->
    <!-- <bean class="org.springframework.web.servlet.mvc.SimpleControllerHar
    <!-- 推荐使用的适配器 -->
    <!-- <bean class="org.springframework.web.servlet.mvc.method.annotation.R

    <!-- 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 -->
    <mvc:annotation-driven />

    <!-- 开启注解扫描 -->
    <context:component-scan base-package="cn.itcast.springmvc.controller"></c

    <!-- <bean name="/hello.do" class="cn.itcast.springmvc.controller.HelloCo

    <!-- 配置视图解析器 -->
    <!-- Example: prefix="/WEB-INF/jsp/", suffix=".jsp", viewname="test" -> "
    <bean class="org.springframework.web.servlet.view.InternalResourceViewRes
        <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

4.RequestMapping（映射请求）

标准 URL 映射

Ant 风格的映射

Rest 风格的映射

限定请求方法的映射

限定参数的映射

4.1. 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

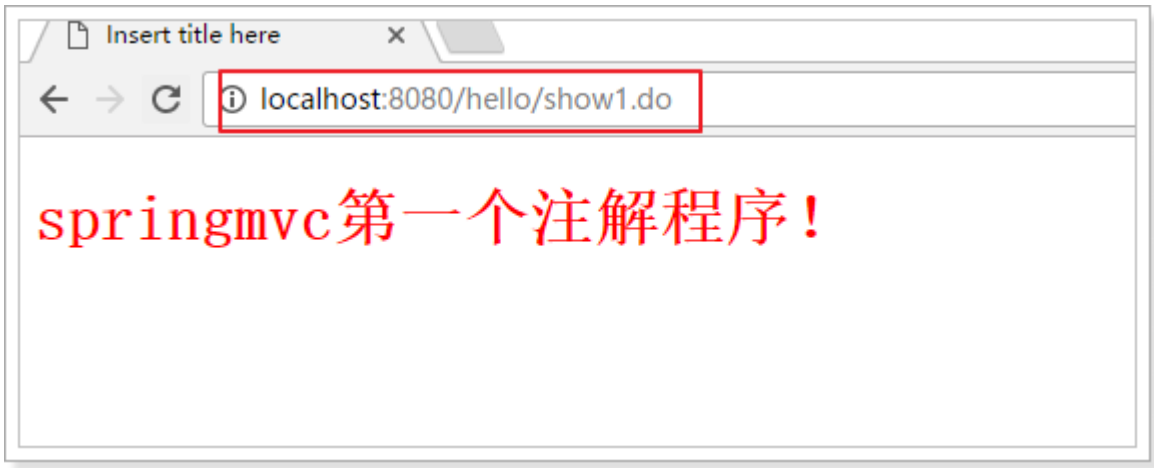
请求映射的规则：

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

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

路径不可重复

```java
@RequestMapping("hello")
@Controller
public class Hello2Controller {

    @RequestMapping(value = "show1")
    public ModelAndView test1() {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("hello");
        mv.addObject("msg", "springmvc第一个注解程序！");
        return mv;
    }
```



4.2. Ant 风格的映射（通配符）

?：通配一个字符

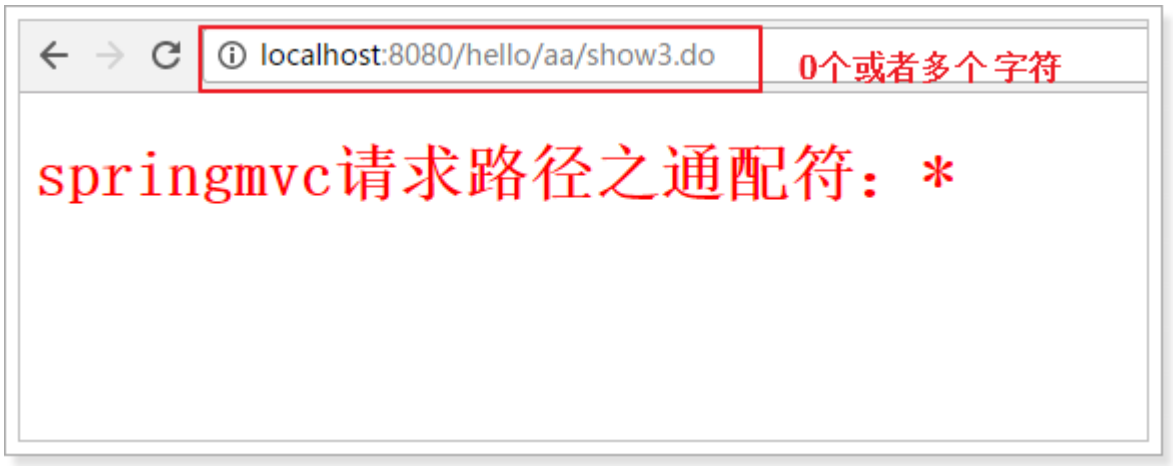
*：通配 0 个或者多个字符

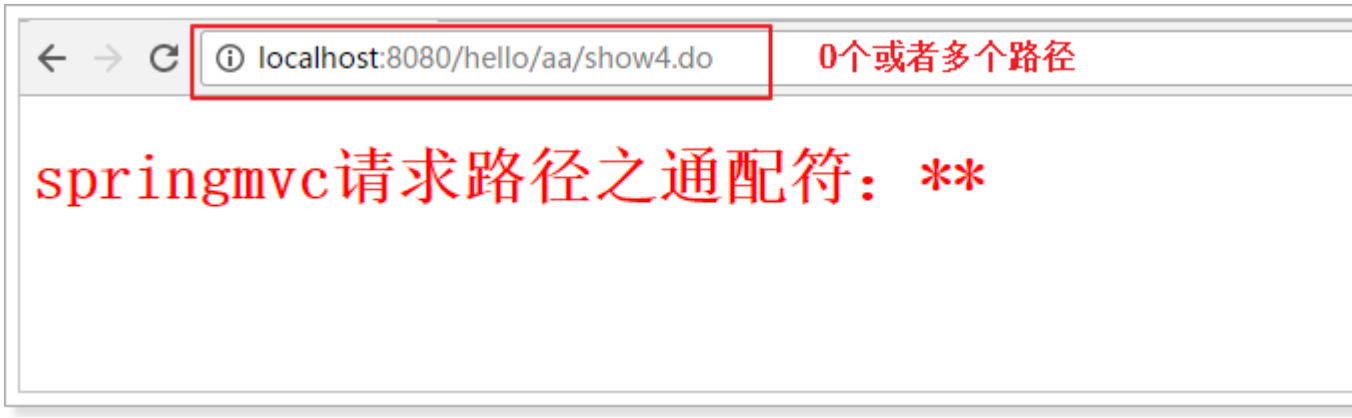
**：通配 0 个或者多个路径

```java
@RequestMapping(value = "/sss?/show2")
public ModelAndView test2() {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("hello");
    mv.addObject("msg", "springmvc请求路径之通配符：？");
    return mv;
}

@RequestMapping(value = "/aa*/show3")
public ModelAndView test3() {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("hello");
    mv.addObject("msg", "springmvc请求路径之通配符：*");
    return mv;
}

@RequestMapping(value = "/**/show4")
public ModelAndView test4() {
    ModelAndView mv = new ModelAndView();
    mv.setViewName("hello");
    mv.addObject("msg", "springmvc请求路径之通配符：**");
    return mv;
}
```

0个或者多个

如果这里配置成/*/show3
匹配0个时则访问路径为
http://localhost:8080/hello//show3.do
注意：路径中的hello后面的双斜杠
在URL路径中和单斜杠效果一样,所以等同于：
http://localhost:8080/hello/show3.do
被认为是0个路径了，而不是0个字符





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?

4.3. 占位符的映射(restful 风格)

示例:@RequestMapping(value=“/user/{userId}/{name}”)

对应请求 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。

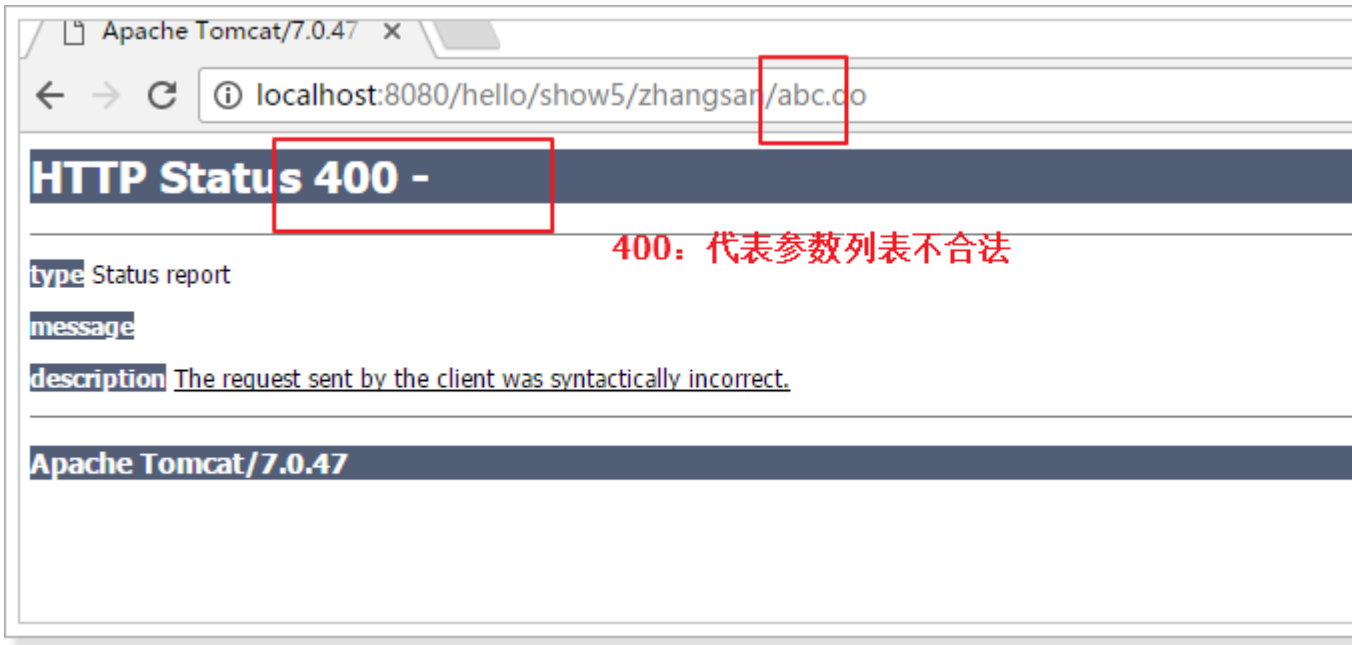
注意：@PathVariable(“key”)中的 key 必须和对应的占位符中的参数名一致，而方法形

参的参数名可任意取

```
@RequestMapping(value = "show5/{name}/{id}")
public ModelAndView test5(@PathVariable("name") String name, @PathVariab
    ModelAndView mv = new ModelAndView();
    mv.setViewName("hello");
    mv.addObject("msg", "springmvc请求路径之占位符name:" + name + ",id=" + id
    return mv;
}
```

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）。



4.4. 限定请求方法的映射

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

```
@RequestMapping(value = "show6", method = RequestMethod.POST)
public ModelAndView test6() {
    ModelAndView mv = new ModelAndView("hello");
    mv.addObject("msg", "springmvc请求路径之限定请求方法: ");
    return mv;
}
```

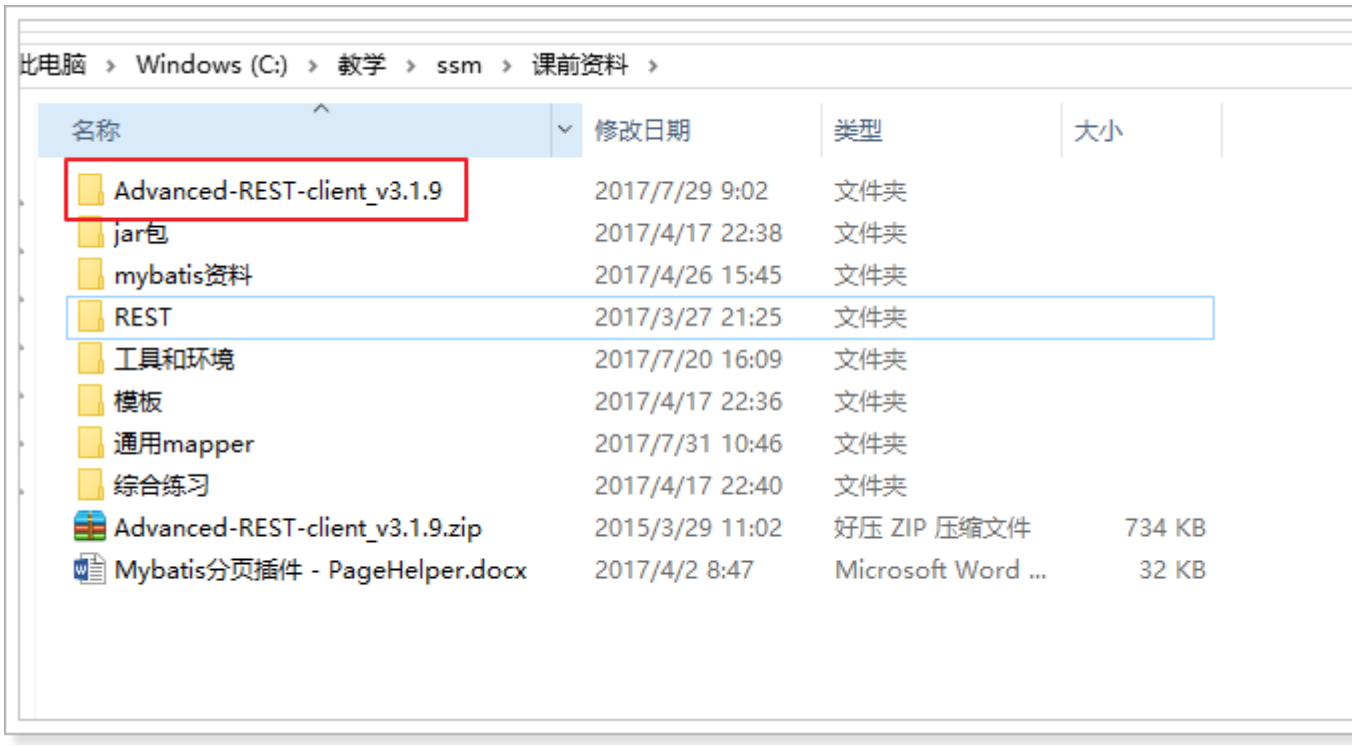
用到了框架提供的 RequestMethod 枚举类，源代码截图：

```
public enum RequestMethod {

    GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE

}
```

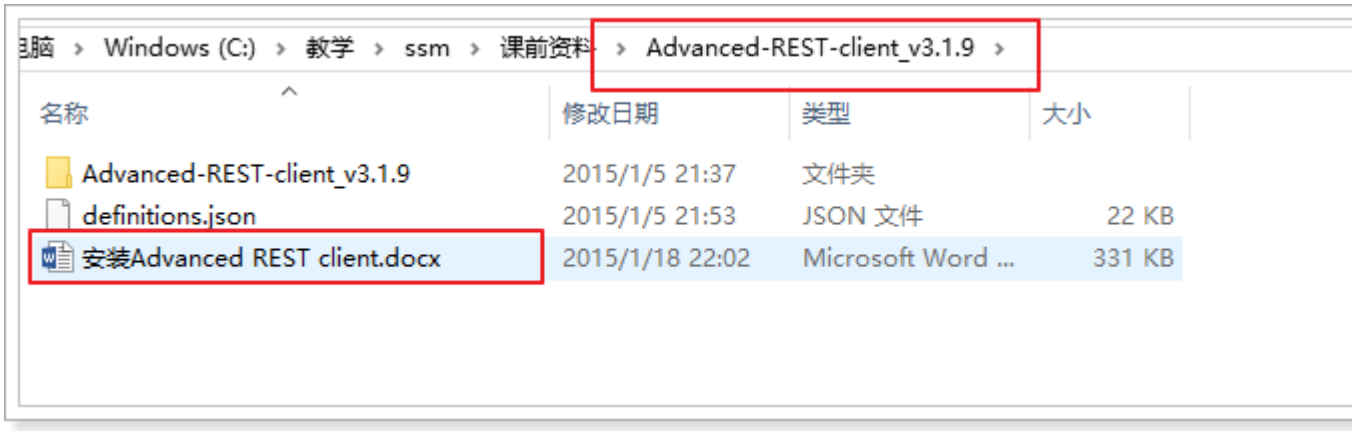
此时 show6 限定请求方法为 POST 请求，如果通过浏览器地址栏输入请求路径（也就是

GET 请求），结果：

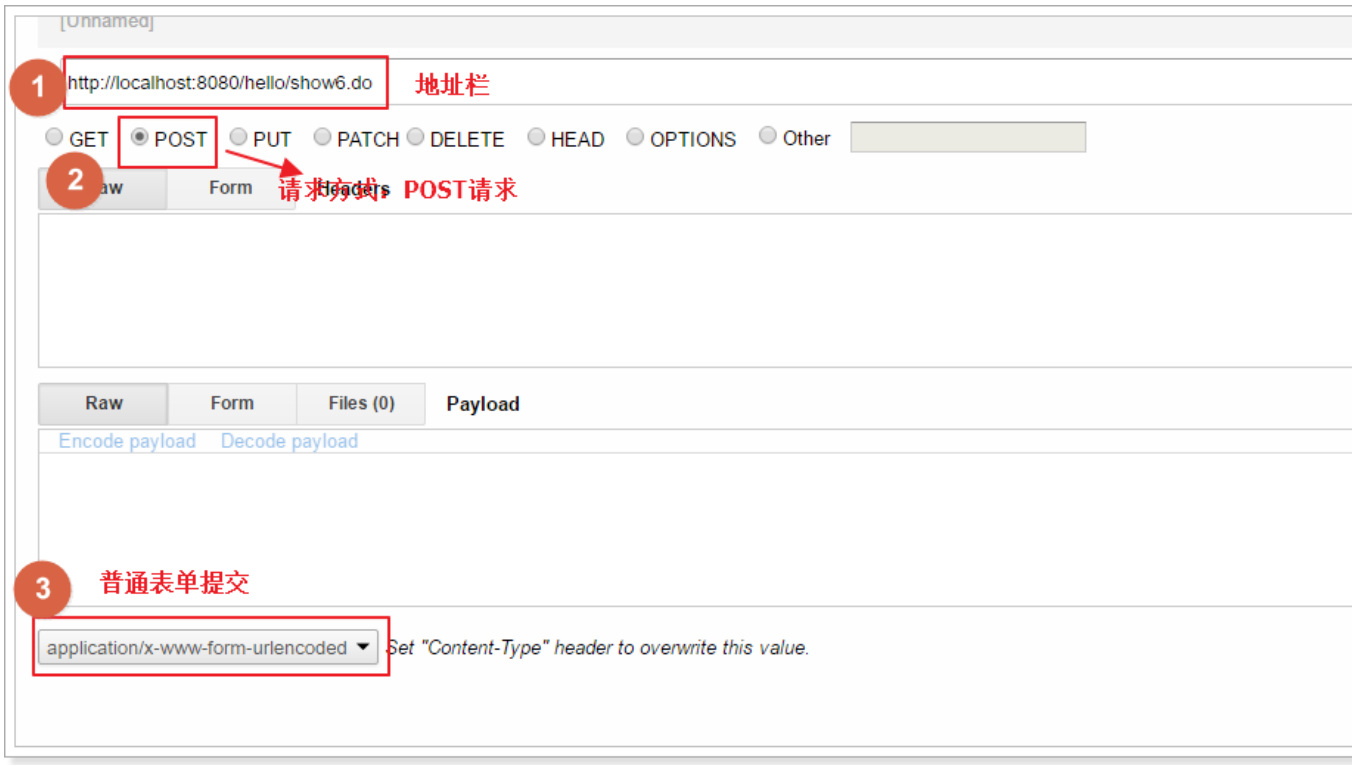
地址栏无法模拟 POST 请求，需要使用浏览器插件，模拟 POST 请求，下面为 chrome 浏览器的模拟插件，参见课前资料



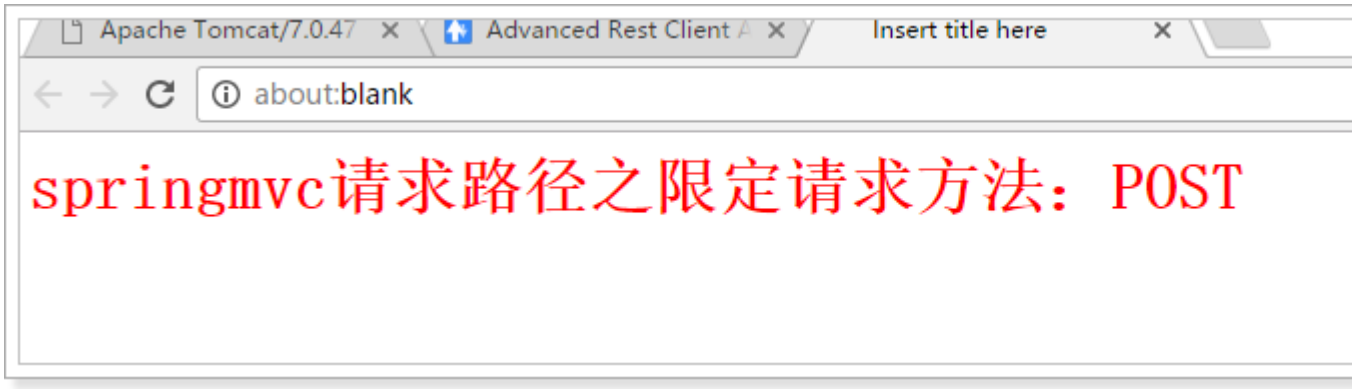
安装过程参见教程：



安装完成后的使用：


[Unnamed]
1
http://localhost:8080/hello/show6.do
地址栏
GET
POST
PUT
PATCH
DELETE
HEAD
OPTIONS
Other
2
Form
请求方式：POST请求
Raw
Form
Files (0)
Payload
Encode payload
Decode payload
3
普通表单提交
application/x-www-form-urlencoded
Set "Content-Type" header to overwrite this value.

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})

```
@RequestMapping(value = "/show7",method={RequestMethod.POST,Requ
public ModelAndView test7() {
    ModelAndView mv = new ModelAndView();

    mv.setViewName("hello");

    mv.addObject("msg", "限定请求方式post和get");

    return mv;
}
```

4.5. 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 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;
    }
```

4.6.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）

```
    @GetMapping(value="show113")
    public ModelAndView test113(){
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("msg", "GetMapping");
```

```
        return mv;
    }
    @PostMapping(value="show114")
    public ModelAndView test114(){
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("msg", "PostMapping");
        return mv;
    }
    @PutMapping(value="show115")
    public ModelAndView test115(){
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("msg", "PutMapping");
        return mv;
    }
    @DeleteMapping(value="show116")
    public ModelAndView test116(){
        ModelAndView mv = new ModelAndView("hello");
        mv.addObject("msg", "DeleteMapping");
        return mv;
    }
```

5.接收数据及数据绑定

a. 接收 servlet 的内置对象

b. 接收占位符请求路径中的参数

c. 接收普通的请求参数

d. 获取 cookie 参数

e. 基本数据类型的绑定

f. Pojo 对象的绑定

g. 集合的绑定

请求参数—>方法形参，方法形参没有顺序。

5.1. 接收 servlet 的内置对象

常用的 servlet 对象,request,response,session

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="show13")
public ModelAndView test13(HttpServletRequest request, HttpServletR
    ModelAndView mv = new ModelAndView("hello");
    StringBuffer sb= new StringBuffer();
    sb.append("request:"+request+"<br>");
    sb.append("response:"+response+"<br>");
    sb.append("session:"+session+"<br>");

    mv.addObject("msg", "springmvc接受参数servlet内置对象:<br>"+sb);
    return mv;
}
```

通过 springMVC 的 Model 对象代替 ModelAndView 对象(推荐!)

```
@RequestMapping(value="show14")
public String test14(Model model,HttpServletRequest request){


    model.addAttribute("msg", "springmvc接受内置对象model");//相当于m

    request.setAttribute("msg", "springmvc接受的request对象");

    return "hello";//返回视图名称,相当于mv.setViewName("hello");
}
```

提示: Model.addAttribute()方法的解析在 request.setAttribute()方法之后,所以当设置的 key 相

同时,不管代码的前后顺序,最终都会采用 model 传递的数据

5.2. 接收请求路径中的占位符

@PathVariable(value=” name”)获取占位符中的参数

```
@RequestMapping(value = "show15/{name}")
public String test15(Model model, @PathVariable("name") String name) {
    model.addAttribute("msg", "springmvc接收参数与数据绑定,占位符: " + name);
    return "hello";
}
```

5.3. 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，自动为 false，如

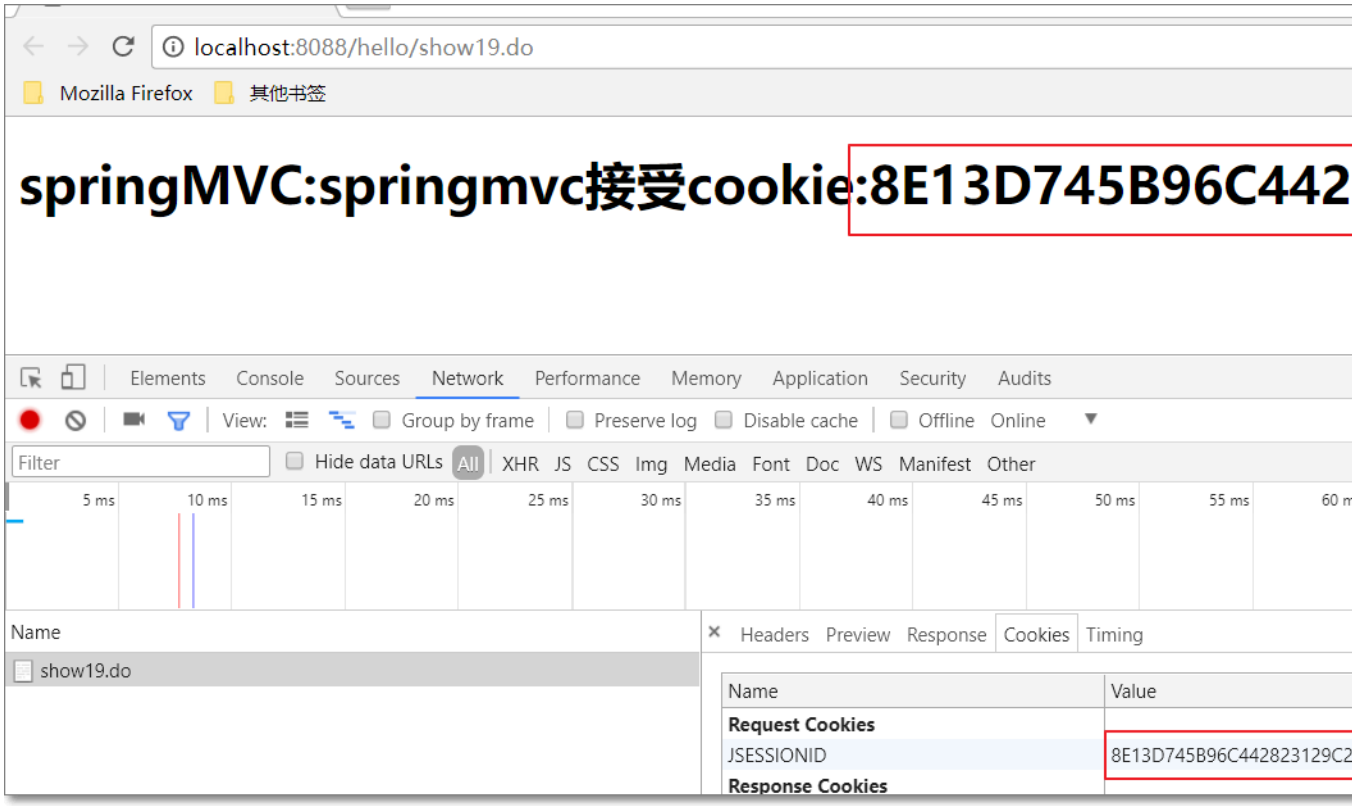
果请求中不包含该参数则使用默认值。

```
@RequestMapping(value="show16")
public String test16(Model model, @RequestParam(value="username")String user
    model.addAttribute("msg", "springmvc接受普通参数username:"+username);
    return "hello";
}


@RequestMapping(value="show17")
public String test17(Model model, @RequestParam(value="username",required=fa
    model.addAttribute("msg", "springmvc接受普通参数username:"+username);
    return "hello";
}

@RequestMapping(value="show18")
public String test18(Model model, @RequestParam(value="username",defaultValu
    model.addAttribute("msg", "springmvc接受普通参数username:"+username);
    return "hello";
}
```

5.4. 获取 cookie

@CookieValue 使用方法同@RequestParam 使用方法一致

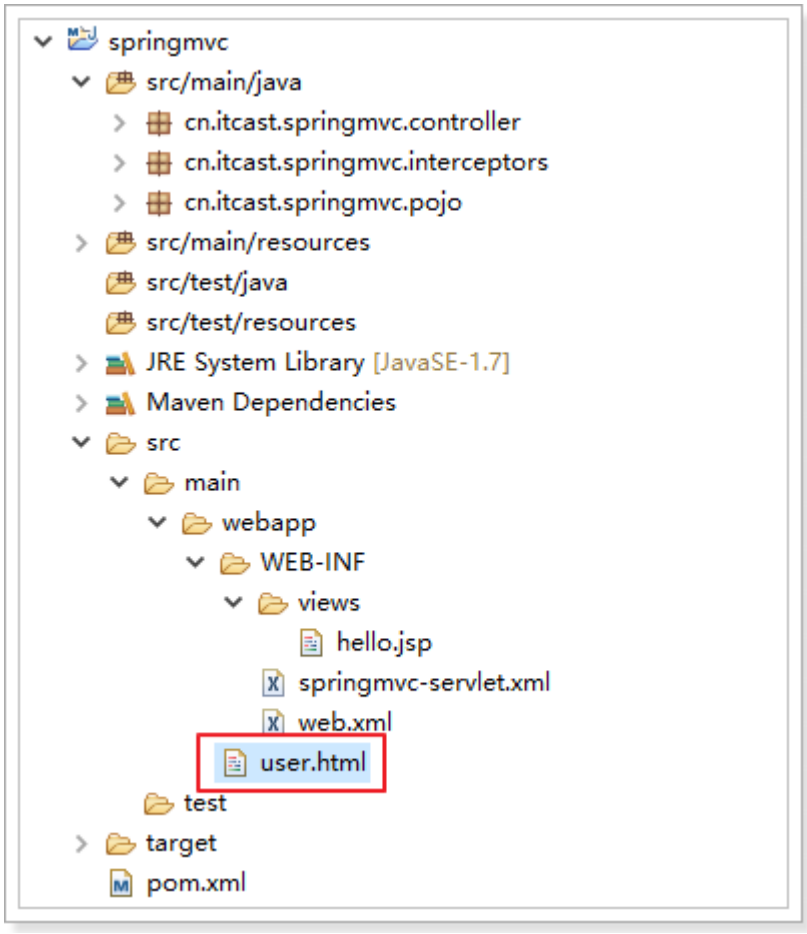
```
@RequestMapping(value="show19")
public String test19(Model model,@CookieValue("JSESSIONID")St
    model.addAttribute("msg", "springmvc接受cookie:"+cookie);
    return "hello";
}
```



5.5. 基本数据类型的绑定

常用基本数据类型:　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/show20.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>
```

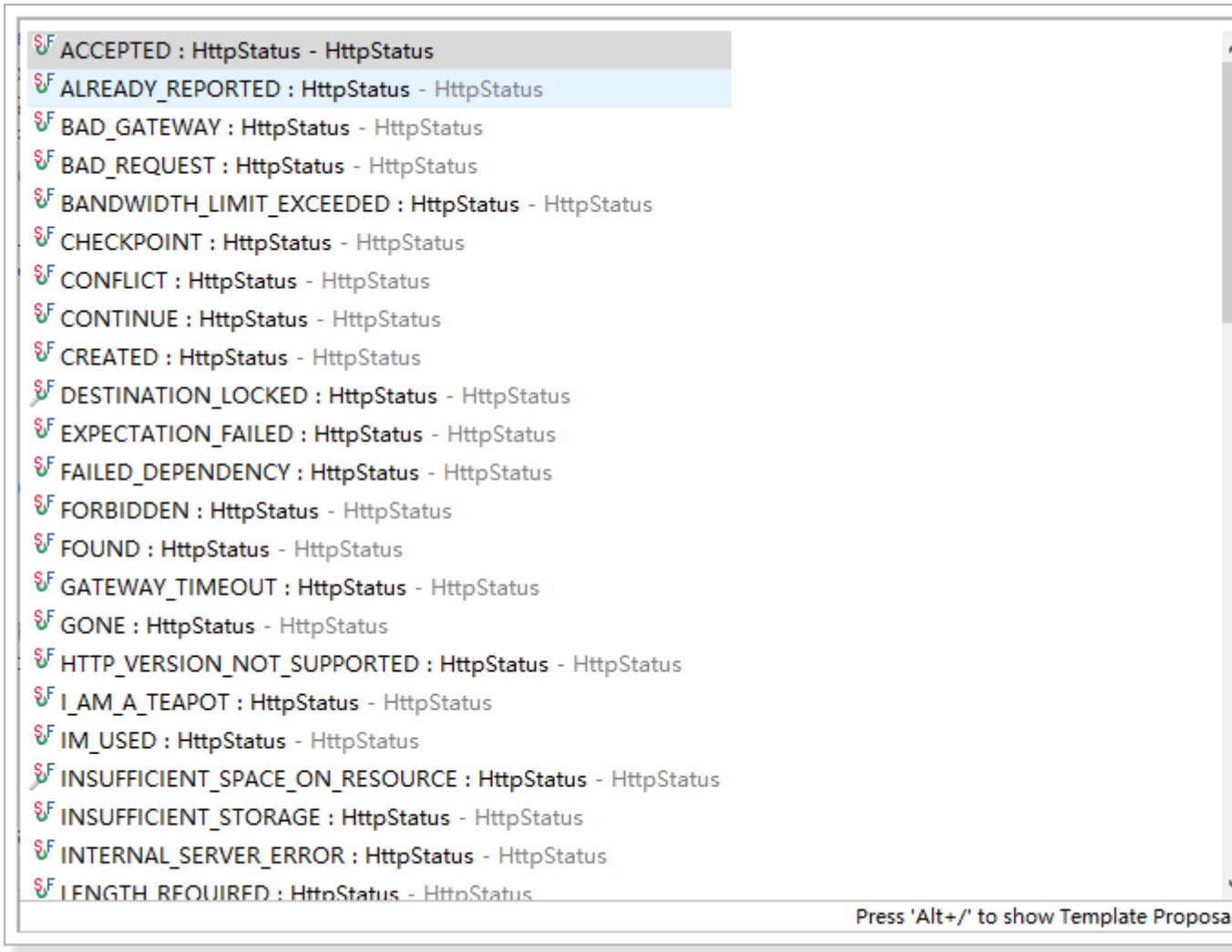
可以通过 localhost:8080/user.html 访问

接受传递的表单数据:如果不想跳转页面方法可无返回值通过@ResponseStatus 指定响应

状态

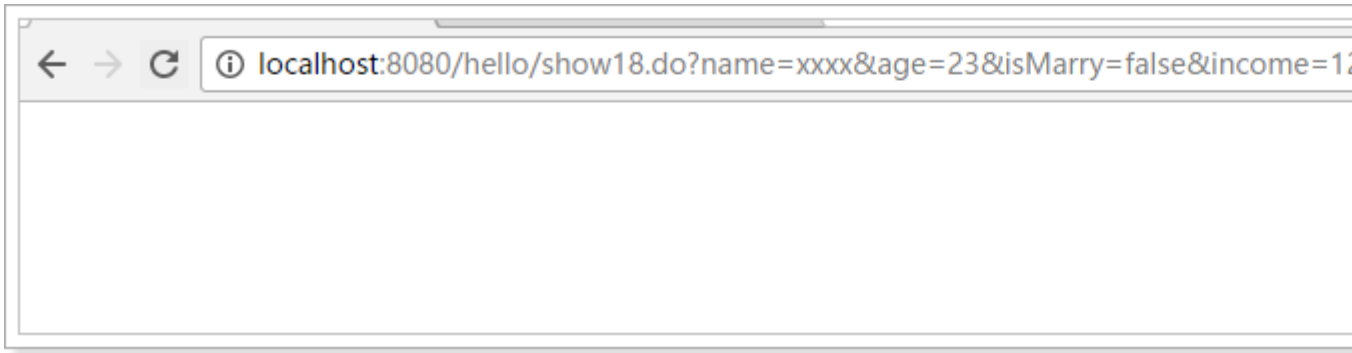
```
@RequestMapping(value="show20")
@ResponseStatus(value=HttpStatus.OK)//返回响应状态OK = 200
public void test20(@RequestParam("name")String name,
        @RequestParam("age")Integer age,
        @RequestParam("isMarry")boolean isMarry,
        @RequestParam("income")Double income,
        @RequestParam("interests")String[] interests){

    StringBuffer sb = new StringBuffer();

    sb.append("name:"+name+"\r\n");
    sb.append("age:"+age+"\r\n");
    sb.append("isMarry:"+isMarry+"\r\n");
    sb.append("income:"+income+"\r\n");
    sb.append("interests:[");
    for (String string : interests) {
        sb.append(string+"  ");
    }
    sb.append("]");
    System.out.println(sb);

}
```

响应状态 HttpStatus 是一个枚举类:

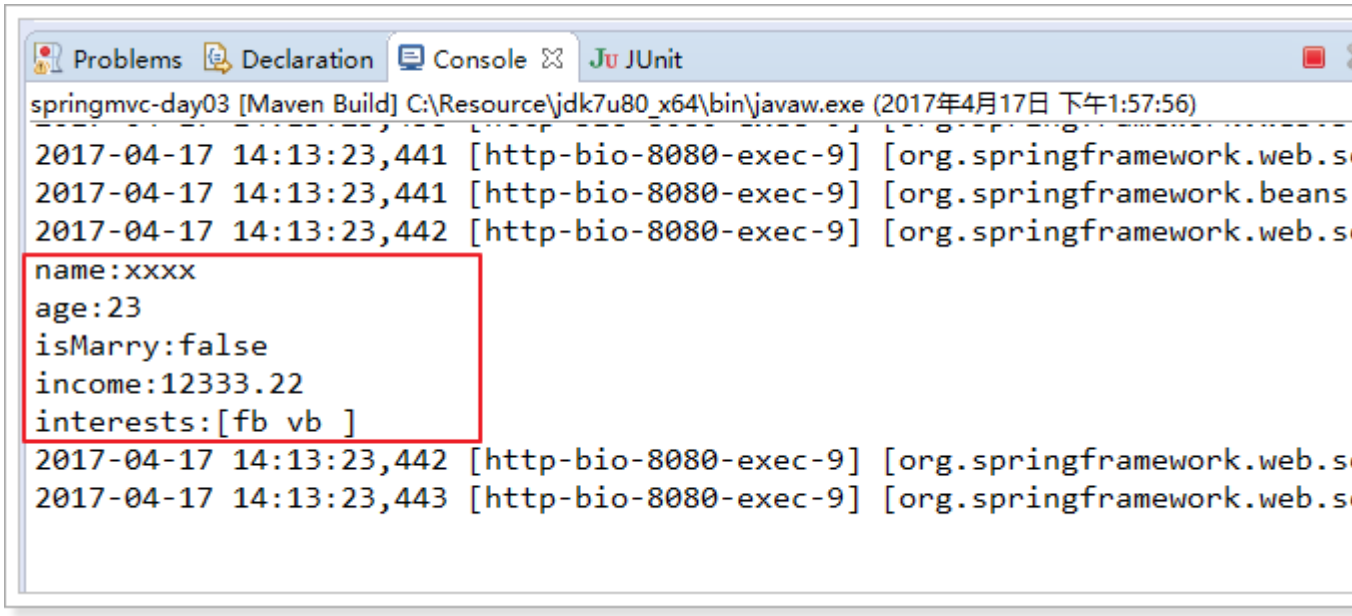


测试：

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



5.6. 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：

```
@RequestMapping(value="show21")
public String test21(Model model,User user){

    model.addAttribute("msg", "springmvc通过pojo封装页面传递参

    return "hello";
}
```

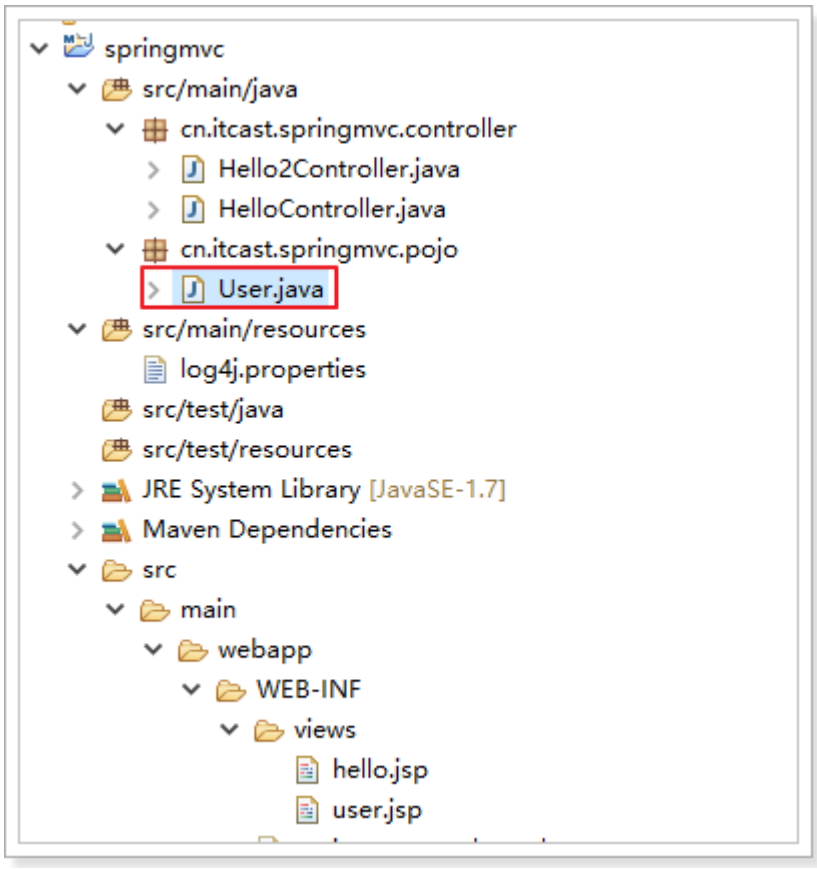
测试：

localhost:8088/hello/show21.do?name=ww&age=22&isMarry=on&income=222&interests=bb&interests=fb&interests=vb

Mozilla Firefox 其他书签

pringMVC:springmvc通过pojo封装页面传递参数:User [name=w
sMarry=true, interests=[bb, fb, vb]]

User 类：



User 内容:

```
public class User {

    private Integer id;
    private String userName;
    private String name;
    private Integer age;
    private boolean isMarry;
    private Double income;
    private String[] interests;


    public Integer getId() {
        return id;
    }

    public void setId(Integer 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 isMarry() {
        return isMarry;
    }

    public void setMarry(boolean isMarry) {
        this.isMarry = isMarry;
    }

    public Double getIncome() {
        return income;
    }

    public void setIncome(Double 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=" + is
              + ", income=" + income + ", interests="
              + Arrays.toString(interests) + "]";
    }



}
```

5.7. 集合的绑定

反例:

```java
@RequestMapping("show22")
public String test21(Model model ,List<User> users) {


    model.addAttribute("msg", users);
    return "hello";

}
```

localhost:8088/hello/show22.do?users[0].name=zhangsan&users[0].age=18&users[0].isMarry=on&users[0].income=1

Mozilla Firefox 其他书签 Advanced Rest Clie 百度一下，你就知道

HTTP Status 500 - Request processing failed; nested exception is org.sprin
Failed to instantiate [java.util.List]: Specified class is an interface

type Exception report

message Request processing failed; nested exception is org.springframework.beans.BeanInstantiationException: Failed to instantiate [java.util.List]: Spec

description The server encountered an internal error that prevented it from fulfilling this request.

exception

```
org.springframework.web.util.NestedServletException: Request processing failed; nested exception is org.springframework.beans.
        org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:982)
        org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:861)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
        org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:846)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:728)
        org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:51)
```

root cause

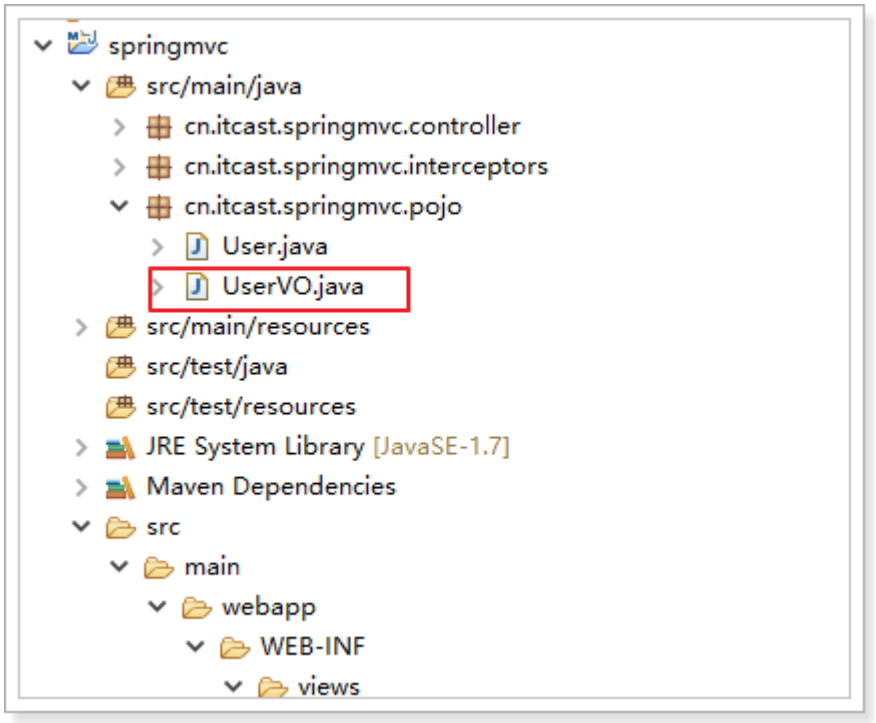
```
org.springframework.beans.BeanInstantiationException: Failed to instantiate [java.util.List]: Specified class is an interface
        org.springframework.beans.BeanUtils.instantiateClass(BeanUtils.java:99)
        org.springframework.web.method.annotation.ModelAttributeMethodProcessor.createAttribute(ModelAttributeMethodProcessor.
        org.springframework.web.servlet.mvc.method.annotation.ServletModelAttributeMethodProcessor.createAttribute(ServletMode
        org.springframework.web.method.annotation.ModelAttributeMethodProcessor.resolveArgument(ModelAttributeMethodProcessor.
        org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentRes
        org.springframework.web.method.support.InvocableHandlerMethod.getMethodArgumentValues(InvocableHandlerMethod.java:158)
```

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:

```
@RequestMapping(value="show22")
public String test22(Model model,UserVO userVO){

    model.addAttribute("msg", "springmvc通过pojo集合封装页面数

    return "hello";
}
```

效果：

localhost:8088/hello/show22.do?users[0].name=ww&users[0].age=22&users[0].isMarry=on&users[0].incom

Mozilla Firefox 其他书签

springMVC:springmvc通过pojo集合封装页面数据:[Use
income=222.0, isMarry=true, interests=[bb, fb, vb]]
income=222.0, isMarry=true, interests=[bb, fb, vb]]

6.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

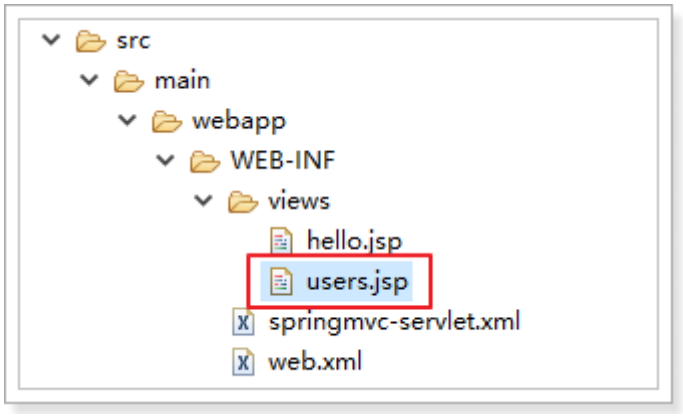
标签。

6.1. 导入 jstl 依赖（pom.xml）

参考父工程：

```
<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

6.2. 静态页面



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>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>
```

6.3. Controller 方法

```
@RequestMapping(value="show23")
public String test23(Model model){

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

    for (int i = 0; i <=10; i++) {
        User user = new User();
        user.setAge(15+i);
        user.setId(1+i);
        user.setUserName("zhangmou"+i);
        user.setName("zhang"+i);
        list.add(user);
    }

    model.addAttribute("userList", list);

    return "users";
}
```

6.4. 引入核心标签库

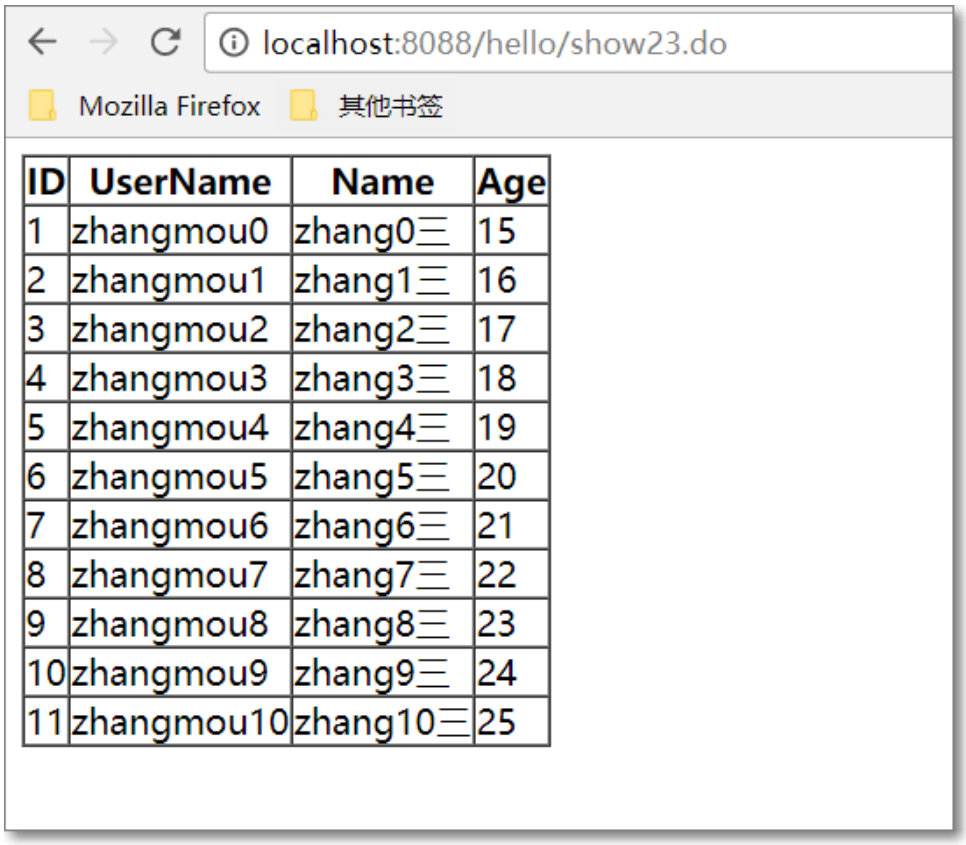
```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
```

6.5. 使用<c:foreach>标签

```
1  <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
4  <!DOCTYPE html>
5  <html>
6  <head>
7  <title>JSTL Demo</title>
8  </head>
9  <body>
10     <table cellpadding=0 cellspacing=0 border="1px solid red;" width="
11         <thead>
12             <tr>
13                 <th>ID</th>
14                 <th>UserName</th>
15                 <th>Name</th>
16                 <th>Age</th>
17             </tr>
18         </thead>
19         <tbody>
20             <c:forEach items="${userList }" var="user">
21             <tr>
22                 <td>${user.id }</td>
23                 <td>${user.userName }</td>
24                 <td>${user.name }</td>
25                 <td>${user.age }</td>
26             </tr>
27             </c:forEach>
28         </tbody>
29     </table>
30 </body>
```

0

6.6. 效果



ID	UserName	Name	Age
1	zhangmou0	zhang0三	15
2	zhangmou1	zhang1三	16
3	zhangmou2	zhang2三	17
4	zhangmou3	zhang3三	18
5	zhangmou4	zhang4三	19
6	zhangmou5	zhang5三	20
7	zhangmou6	zhang6三	21
8	zhangmou7	zhang7三	22
9	zhangmou8	zhang8三	23
10	zhangmou9	zhang9三	24
11	zhangmou10	zhang10三	25

7.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

7.1. 引入依赖

在 Pom.xml 中引入 jackson 依赖，参考父工程：

```
<!-- Jackson Json处理工具包 -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

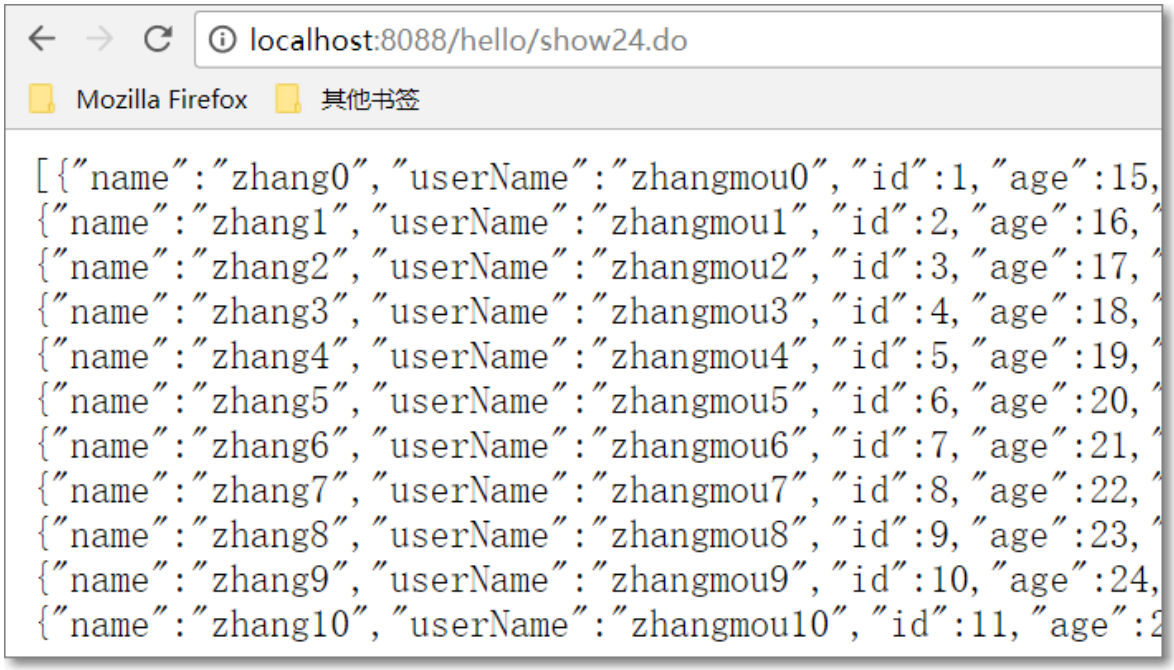
7.2. @ResponseBody

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 通过默认的 json 转化器转化输出

Controller 方法：

```java
@RequestMapping(value="show24")
@ResponseBody//将该方法的返回值转换为json字符串
public List<User> test24(){

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

    for (int i = 0; i <=10; i++) {
        User user = new User();
        user.setAge(15+i);
        user.setId(1+i);
        user.setUserName("zhangmou"+i);
        user.setName("zhang"+i);
        list.add(user);
    }


    return  list;
}
```

测试：



7.3. @RequestBody

```
@RequestMapping(value = "show23")
public String test23(@RequestBody User user, Model model) {
    model.addAttribute("msg", user.toString());
    return "hello";
}
```

http://localhost:8080/hello/show23.do

GET POST PUT PATCH DELETE HEAD OPTIONS Other

Raw Form Headers

Raw Form Files (0) Payload

Encode payload Decode payload

{"name":"马云0","userName":"mayun0","id":1,"age":20,"income":null,"isMarry":null,"interests":null}

application/json Set "Content-Type" header to overwrite this value.

注意：选择**json**提交

Status 200 OK Loading time: 102 ms

Request headers

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Sa
Origin: chrome-extension://giamknmkmombfhfijfhiagmapcdbnojm
Content-Type: application/json
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.8
Cookie: JSESSIONID=7041B56B98EEE7EDB8D246C78D1B8BDD

8.文件上传

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

8.1. 添加依赖

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

8.2. 文件上传解析器

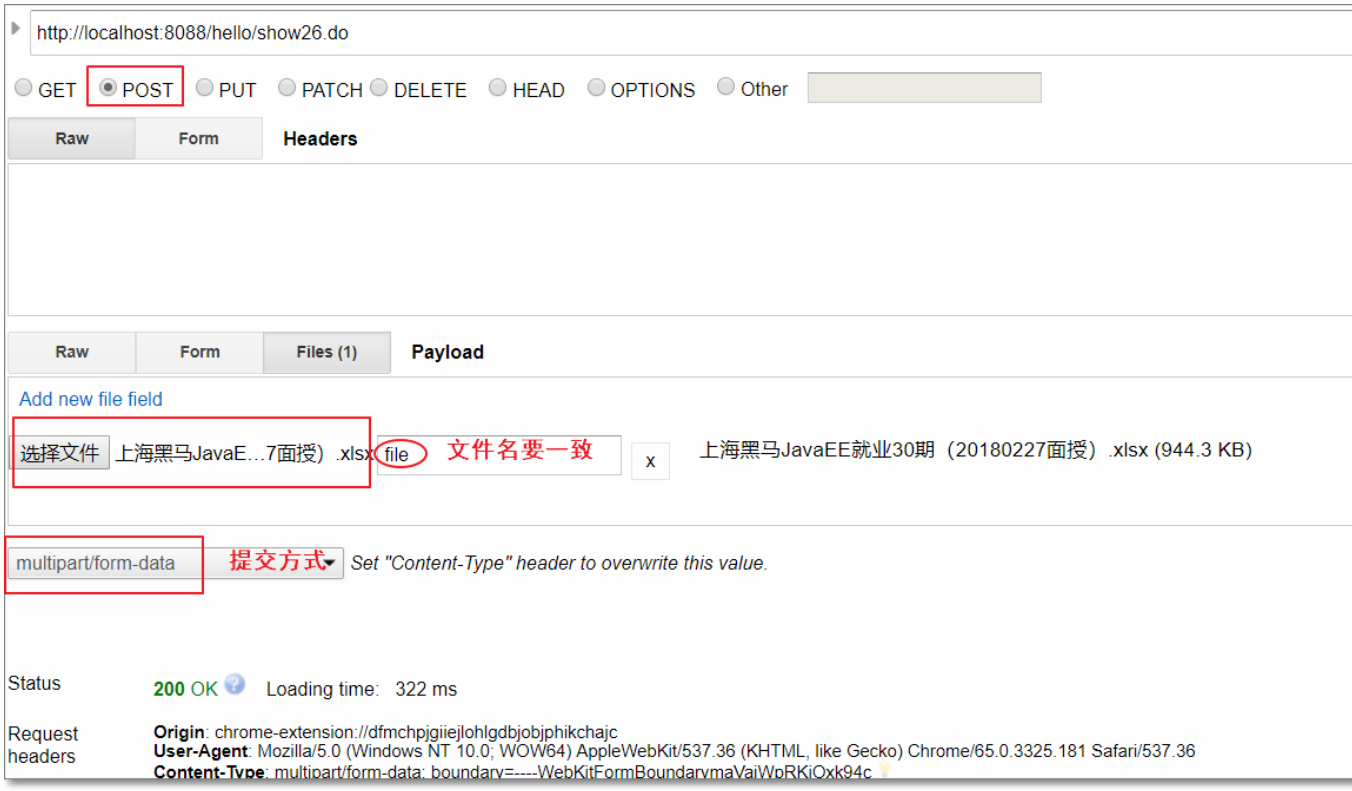
在 springmvc-servlet.xml 中配置

```
<!-- 定义文件上传解析器 -->
   <bean id="multipartResolver"

   class="org.springframework.web.multipart.commons.CommonsMu
ltipartResolver">
      <!-- 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 -->
      <property name="maxUploadSize"
value="5242880"></property>
   </bean>
```

8.3. Controller 方法

```
@RequestMapping(value="show26")
@ResponseStatus(value=HttpStatus.OK)
public void test26(@RequestParam("file")MultipartFile file) throws Illega
    if(file!=null){
        file.transferTo(new File("d:\\tmp\\"+file.getOriginalFilename()))
    }
}
```

8.4. 效果



9.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 或者 forward:show.do

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

```java
@RequestMapping(value="show27")
public String test27(Model model){

    return  "redirect:show29.do?type=redirect";
}


@RequestMapping(value="show28")
public String test28(Model model){

    return  "forward:show29.do?type=forward";
}


@RequestMapping(value="show29")
public String test29(Model model,@RequestParam("type")String type){

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

    return  "hello";
}
```

在浏览器测试重定向（show27.do）： 地址栏发生改变

在浏览器地址栏测试重定向（show28.do）：地址栏未发生改变

10. 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：

a.preHandle 调用 Handler 之前执行，称为前置方法

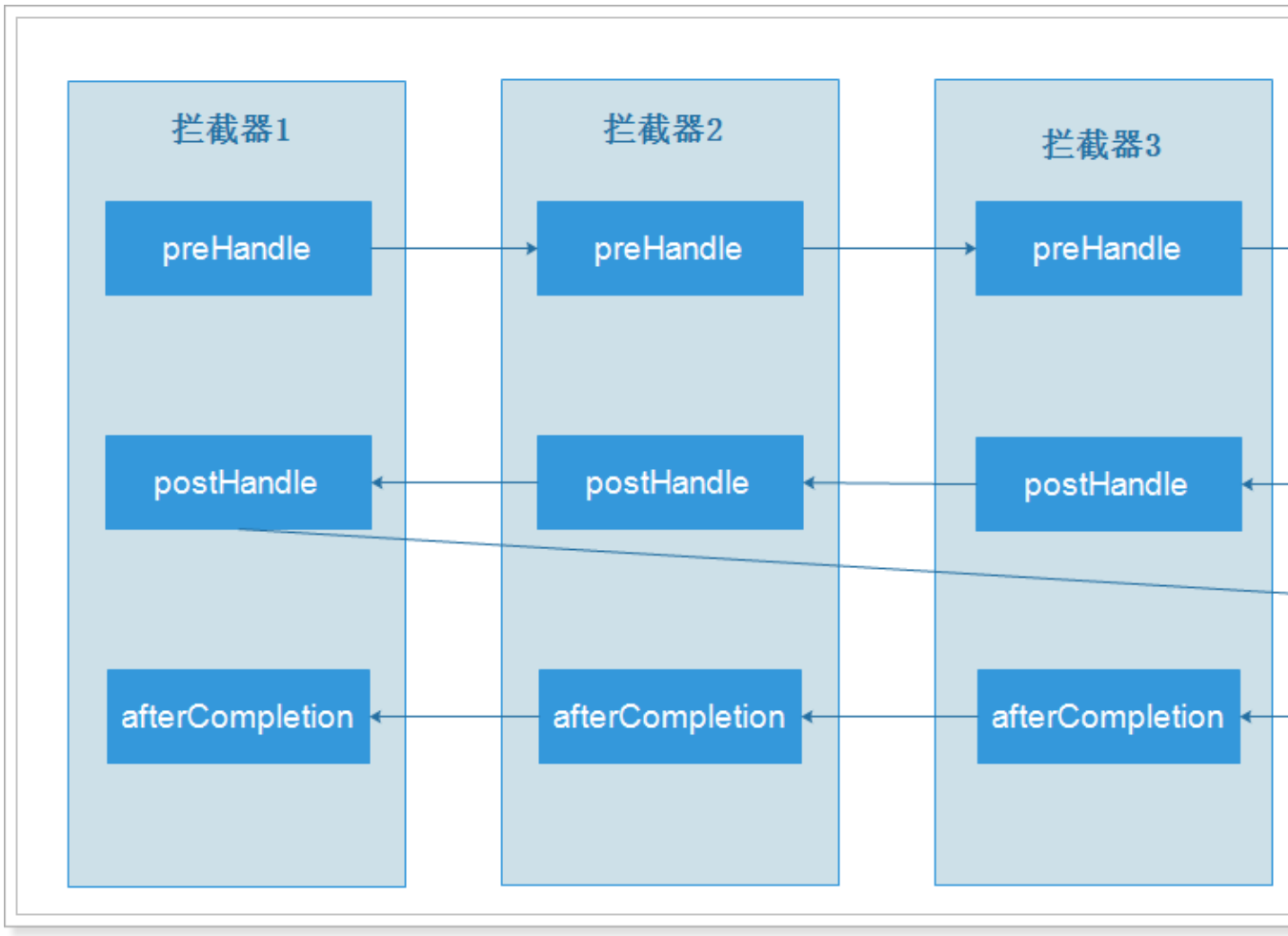
返回值：true 表示放行，后续业务逻辑继续执行

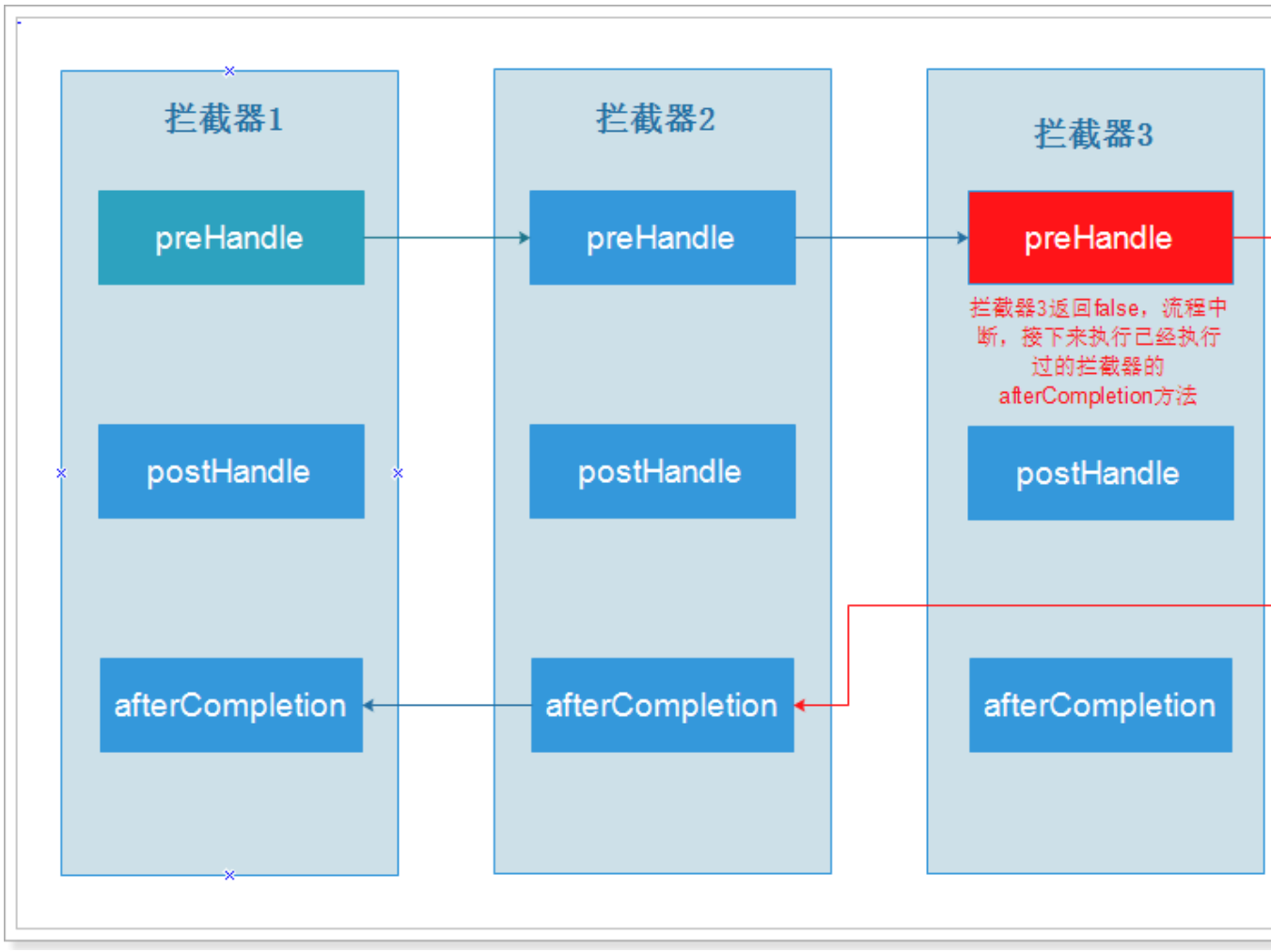
false 表示被拦截，后续业务逻辑不再执行，但之前返回 true 的拦截器的完成方法会倒叙执行

b.postHandle 调用 Handler 之后执行，称为后置方法

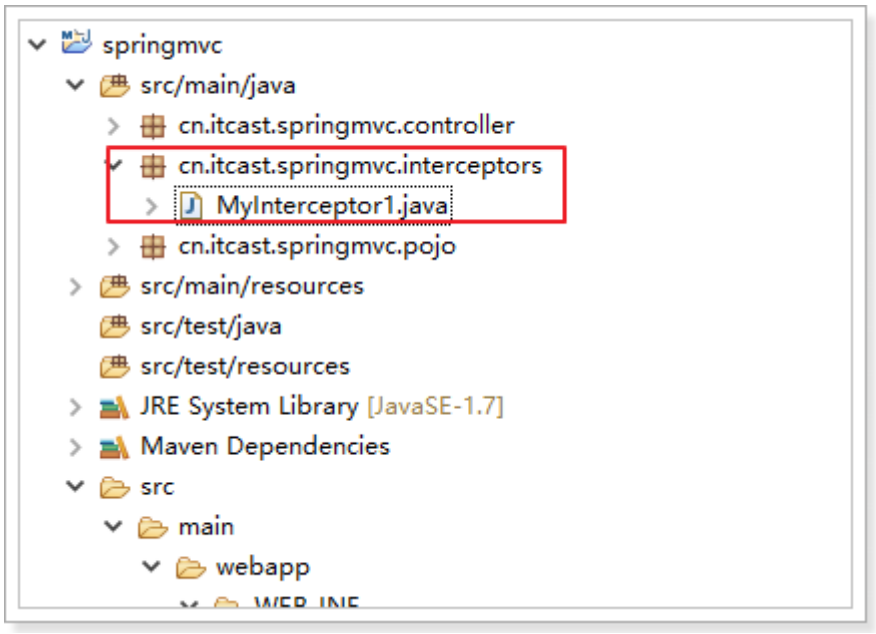
c.afterCompletion 视图渲染完成之后执行

10.1. 拦截器的执行过程




拦截器1
preHandle
postHandle
afterCompletion
拦截器2
preHandle
postHandle
afterCompletion
拦截器3
preHandle
拦截器3返回false，流程中断，接下来执行已经执行过的拦截器的afterCompletion方法
postHandle
afterCompletion

10.2. 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");

    }

}
```

10.3. 配置拦截器

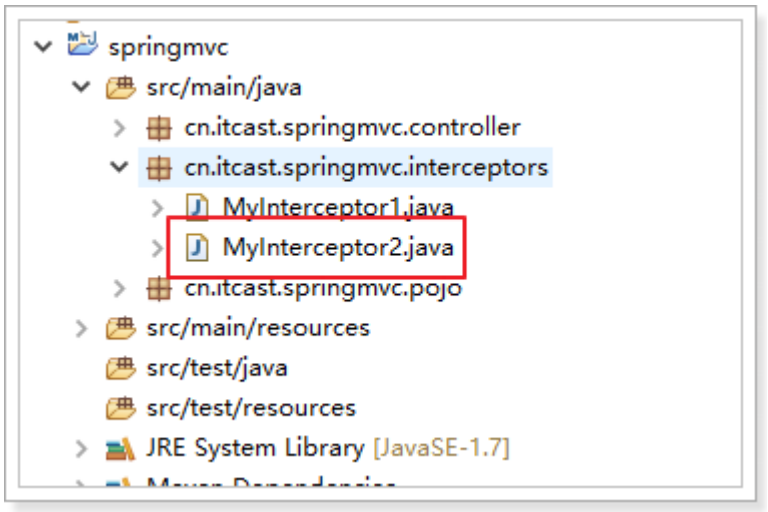
在 springmvc-servlet.xml 中配置自定义的拦截器，/**：拦截所有请求

```
    <!-- 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>
```

10.4. 测试

```
2017-03-13 15:43:40,530 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,536 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,540 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,540 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,544 [http-bio-8080-exec-1] [org.springframe
MyInterceptor, 前置方法执行
handler方法已经执行!
MyInterceptor, 后置方法执行
2017-03-13 15:43:40,556 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,556 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,556 [http-bio-8080-exec-1] [org.springframe
2017-03-13 15:43:40,560 [http-bio-8080-exec-1] [org.springframe
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

MyInterceptor, 完成方法执行
2017-03-13 15:43:40,645 [http-bio-8080-exec-1] [org.springframe
```

10.5. 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");
    }

    /**
     * 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：

```
2017-03-13 15:47:04,572 [http-bio-8080-exec-1] [org.spri
MyInterceptor, 前置方法执行
MyInterceptor2, 前置方法执行
handler方法已经执行!
MyInterceptor2, 后置方法执行
MyInterceptor, 后置方法执行
2017-03-13 15:47:04,584 [http-bio-8080-exec-1] [org.spri
2017-03-13 15:47:04,584 [http-bio-8080-exec-1] [org.spri
2017-03-13 15:47:04,584 [http-bio-8080-exec-1] [org.spri
2017-03-13 15:47:04,592 [http-bio-8080-exec-1] [org.spri
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

MyInterceptor2, 完成方法执行
MyInterceptor, 完成方法执行
2017-03-13 15:47:04,682 [http-bio-8080-exec-1] [org.spri
```

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。