

# Spring MVC 课程教案

版本号：JAVAEE1.0

密 级：受控文档

## 课 程 标 准 化

**2018 年 07 月 01 日**



## 目录

1	SpringMVC 架构.....	5
1.1	Spring mvc 介绍 .....	5
1.2	Web MVC.....	5
2	SpringMVC 入门 .....	6
2.1	第一步:建立 Web 项目, 导入 jar 包.....	7
2.2	第二步: 前端控制器配置 .....	7
2.2.1	Servlet 拦截方式 .....	8
2.3	第三步: springmvc 配置文件.....	8
2.4	第四步: 处理器开发.....	9
2.5	第五步: 处理器配置(配置 Handler).....	10
2.6	第六步: 配置视图解析器 .....	10
2.7	第七步: 视图开发 .....	11
2.8	第八步: 部署在 tomcat 测试 .....	11
3	SpringMVC 核心和工作流程 .....	12
3.1	Spring web mvc 架构.....	12
3.1.1	架构图 .....	12
3.1.2	架构流程.....	13
3.2	DispatcherServlet 前端控制器 .....	13
3.3	HandlerMapping 处理器映射器.....	14
3.4	HandlerAdapter 处理器适配器.....	15
3.5	视图解析器 .....	15
4	SpringMVC 注解开发.....	16
4.1	新建 web 项目, 导入 jar, 配置前端控制器 .....	16
4.2	springmvc 配置文件—注解方式.....	16
4.3	注解的 Controller 开发.....	17
4.3.1	注解的 Controller 代码 .....	17
4.3.2	组件扫描器 .....	17
4.4	配置视图解析器.....	17
4.5	<mvc:annotation-driven> 注解驱动.....	18
5	Springmvc 的注解详解.....	18
5.1	@RequestMapping .....	18
5.1.1	注解的使用 .....	18
5.1.2	注解的属性 .....	19
5.2	参数类型和返回类型.....	20
5.2.1	返回 ModelAndView.....	20
5.2.2	返回 void .....	21
5.2.3	返回字符串 .....	22
6	数据绑定 .....	23
6.1	默认支持的数据类型.....	23
6.2	简单数据类型 .....	24
6.3	简单 POJO 类型 .....	25
6.4	包装 POJO 类型 .....	27

---

6.5	自定义参数绑定.....	29
6.6	集合类.....	30
6.7	与 struts2 不同.....	32
7	JSON 数据交互.....	33
7.1	JSON 概述.....	33
7.1.1	JSON 语法规则.....	33
7.1.2	JSON 类型.....	33
7.2	JSON 数据转换.....	34
7.3	JSON 数据转换案例.....	34
7.4	RESTful 支持.....	38
8	拦截器.....	39
8.1	定义.....	39
8.2	拦截器定义.....	40
8.3	拦截器配置.....	40
8.4	拦截器的执行流程.....	41
8.4.1	单个拦截器执行流程.....	41
8.4.2	多个拦截器执行顺序.....	41
9	文件上传下载.....	42
9.1	文件上传.....	42
9.1.1	form 表单.....	42
9.1.2	配置解析器.....	42
9.1.3	依赖包.....	43
9.1.4	文件上传控制类.....	43
9.2	文件下载.....	44

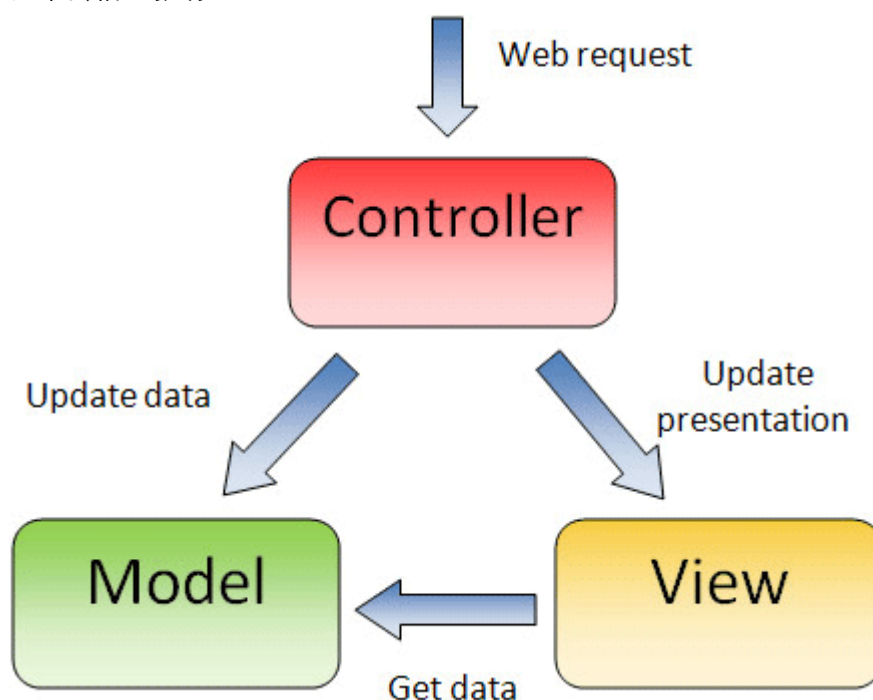
# 1 SpringMVC 架构

## 1.1 Spring mvc 介绍

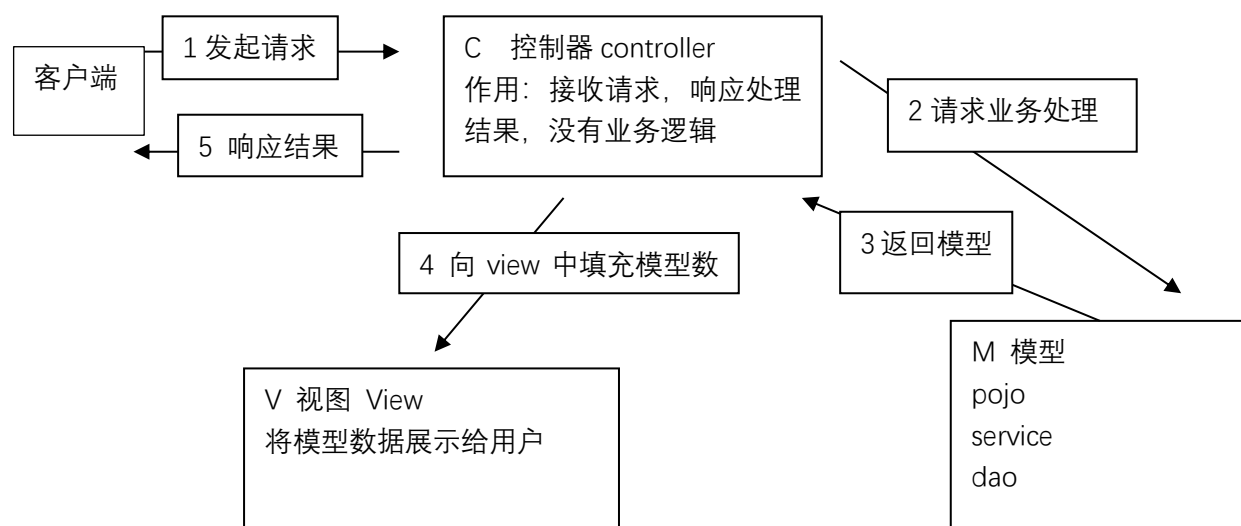
Spring MVC 是一种基于 Java 的实现了 Web MVC 设计模式的请求驱动类型的轻量级 Web 框架，即使用了 MVC 架构模式的思想，将 web 层进行职责解耦，基于请求驱动指的就是使用请求-响应模型，框架的目的就是帮助我们简化开发，Spring Web MVC 也是要简化我们日常 Web 开发的。

## 1.2 Web MVC

模型-视图-控制器 (MVC) 是一个众所周知的以设计界面应用程序为基础的设计模式。它主要通过分离**模型**、**视图**及**控制器**在应用程序中的角色将业务逻辑从界面中解耦。通常，模型负责封装应用程序数据在视图层展示。视图仅仅只是展示这些数据，不包含任何业务逻辑。控制器负责接收来自用户的请求，并调用后台服务 (manager 或者 dao) 来处理业务逻辑。处理后，后台业务层可能会返回了一些数据在视图层展示。控制器收集这些数据及准备模型在视图层展示。MVC 模式的核心思想是将业务逻辑从界面中分离出来，允许它们单独改变而不会相互影响。



mvc 设计模式在 b/s 系统下应用：



- 1、用户发起 request 请求至控制器(Controller)  
控制接收用户请求的数据，委托给模型进行处理
- 2、控制器通过模型(Model)处理数据并得到处理结果  
模型通常是指业务逻辑
- 3、模型处理结果返回给控制器
- 4、控制器将模型数据在视图(View)中展示  
web 中模型无法将数据直接在视图上显示，需要通过控制器完成。如果在 C/S 应用中模型是可以将数据在视图中展示的。
- 5、控制器将视图 response 响应给用户  
通过视图展示给用户要的数据或处理结果。

## 2 SpringMVC 入门

Spring Mvc 具有以下特点：

- 是 pring 框架的一部分，可以方便的利用 spring 所提供的其他功能
- 灵活性强，易于与其他框架集成
- 提供了一个前端控制器 DispatcherServlet,使开发人员无须额外的开发控制器对象
- 可以自动的绑定用户输入，并能正确的转换数据类型
- 内置了常见的效验器，可以效验用户输入
- 支持国际化
- 支持多种视图技术。它支持 JSP/FreeMarker 等
- 使用基于 XML 的配置文件，在编译后，不需要重新编译应用程序。

\*\*\*\*\*

## 2.1 第一步:建立 Web 项目，导入 jar 包

- ▼ 20180718\_springmvc\_01
  - > JAX-WS Web Services
  - > Deployment Descriptor: 20180718\_springmvc\_01
  - > Java Resources
  - > JavaScript Resources
  - > build
  - ▼ WebContent
    - > META-INF
    - ▼ WEB-INF
      - > jsp
      - ▼ lib
        - commons-logging-1.1.1.jar
        - jstl-1.2.jar
        - log4j-1.2.17.jar
        - spring-aop-4.1.3.RELEASE.jar
        - spring-aspects-4.1.3.RELEASE.jar
        - spring-beans-4.1.3.RELEASE.jar
        - spring-context-4.1.3.RELEASE.jar
        - spring-context-support-4.1.3.RELEASE.jar
        - spring-core-4.1.3.RELEASE.jar
        - spring-expression-4.1.3.RELEASE.jar
        - spring-jdbc-4.1.3.RELEASE.jar
        - spring-jms-4.1.3.RELEASE.jar
        - spring-messaging-4.1.3.RELEASE.jar
        - spring-tx-4.1.3.RELEASE.jar
        - spring-web-4.1.3.RELEASE.jar
        - spring-webmvc-4.1.3.RELEASE.jar
      - web.xml

## 2.2 第二步：前端控制器配置

在 WEB-INF\web.xml 中配置前端控制器 DispatcherServlet

```
<servlet>
    <!-- 配置前端控制器 -->
    <servlet-name>springmvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 初始化时加载配置文件 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <!-- 表示容器在启动时 当前Servlet的加载顺序 -->
    <load-on-startup>1</load-on-startup>
```

```

</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.action</url-pattern>
</servlet-mapping>

```

- load-on-startup: 表示servlet随服务启动;
- url-pattern: \*.action的请求交给DispatcherServlet处理。
- contextConfigLocation: 指定springmvc配置的加载位置(配置处理器映射器和适配器), 如果不指定则默认加载WEB-INF/[DispatcherServlet 的Servlet 名字]-servlet.xml。
- url-pattern:
  - 第一种: \*.action, 访问以.action结尾由DispatcherServlet进行解析
  - 第二种: /, 所有访问的地址都有DispatcherServlet进行解析, 对于静态文件的解析不需要DispatcherServlet解析

## 2.2.1 Servlet 拦截方式

1、拦截固定后缀的url, 比如设置为 \*.do、\*.action, 例如: /user/add.action  
此方法最简单, 不会导致静态资源 (jpg, js, css) 被拦截。

2、拦截所有, 设置为/, 例如: /user/add /user/add.action  
此方法可以实现REST风格的url, 很多互联网类型的应用使用这种方式。  
但是此方法会导致静态文件 (jpg, js, css) 被拦截后不能正常显示。需要特殊处理。

3、拦截所有, 设置为/\*, 此设置方法错误, 因为请求到Action, 当action转到jsp时再次被拦截, 提示不能根据jsp路径mapping成功。

## 2.3 第三步: springmvc 配置文件

springmvc.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc

```



```

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

</beans>

```

springmvc默认加载WEB-INF/[前端控制器的名字]-servlet.xml，也可以在前端控制器定义处指定加载的配置文件，如下：

```

<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:springmvc.xml</param-value>
</init-param>

```

如上代码，通过contextConfigLocation加载classpath下的springmvc.xml配置文件。

### 配置处理器适配器

在 springmvc.xml 文件配置如下：

```

<!-- 处理器适配器 -->
<!-- SimpleControllerHandlerAdapter: 简单控制器处理适配器 -->
<!-- 可以执行那种处理器呢？所有实现了Controller接口的处理器 -->
<bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"></bean>

<!-- 处理器映射器 HandlerMapping-->
<!-- 将bean的name作为请求的url: 同时url要以.action结尾 -->
<bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"></bean>

```

- SimpleControllerHandlerAdapter：即简单控制器处理适配器，所有实现了 org.springframework.web.servlet.mvc.Controller 接口的Bean作为 Springmvc的后端控制器。
- BeanNameUrlHandlerMapping：表示将定义的Bean名字作为请求的url，需要将编写的 controller在spring容器中进行配置，且指定bean的name为请求的url，且必须以.action结尾。

## 2.4 第四步：处理器开发

在 src 的 com.hpe.controller 路径下新建 FirstController 对象，该对象需要实现 Controller 接口，如下图

```

public class FirstController implements Controller{

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

```

```

// TODO Auto-generated method stub
//创建ModelAndView 填充数据、设置视图
ModelAndView mv = new ModelAndView();
//向模型对象中填充数据 相当于request.setAttribute()
mv.addObject("msg", "这是我的第一个Spring Mvc程序");
//设置视图
mv.setViewName("user");
return mv;
}
}

```

## 2.5 第五步：处理器配置(配置 Handler)

将编写的 Handler 在 spring 容器中进行加载。

在 springmvc.xml 文件配置如下：

```

<!-- 配置controller的bean -->
<bean          id="firstController"          class="com.hpe.controller.FirstController"
name="/first.action"></bean>

```

name="/first.action": 前边配置的处理器映射器为BeanNameUrlHandlerMapping, 如果请求的URL 为“上下文/first.action”将会成功映射到FirstController控制器。

## 2.6 第六步：配置视图解析器

在 springmvc.xml 文件配置如下：

```

<!-- 视图解析器, 支持jsp -->
<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"></property>
    <property name="suffix" value=".jsp"></property>
</bean>

```

InternalResourceViewResolver: 支持JSP视图解析

viewClass: JstlView表示JSP模板页面需要使用JSTL标签库, 所以classpath中必须包含jstl的相关jar 包;

prefix 和suffix: 查找视图页面的前缀和后缀, 最终视图的址为:

前缀+逻辑视图名+后缀, 逻辑视图名需要在controller中返回ModelAndView指定, 比如逻

辑视图名为hello，则最终返回的jsp视图地址 “WEB-INF/jsp/hello.jsp”

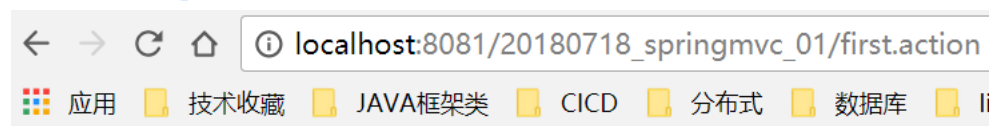
## 2.7 第七步：视图开发

创建/WEB-INF/jsp/user.jsp 视图页面：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    welcome to user.jsp!! <br>
    ${msg }
</body>
</html>
```

## 2.8 第八步：部署在 tomcat 测试

通过请求：<http://localhost:8080/项目名/first.action>，页面如下：



welcome to user.jsp!!  
这是我的第一个Spring Mvc程序

处理器映射器根据 url 找不到 Handler，报下边的错误。说明 url 错误

HTTP Status 404 - /20180818\_springmvc\_01/user.action1

处理器映射器根据 url 找到了 Handler，转发的 jsp 页面找到，报下边的错误，说明 jsp 页面地址错误了。

HTTP Status 404 - /20180818\_springmvc\_01/WEB-INF/jsp/user.jsp

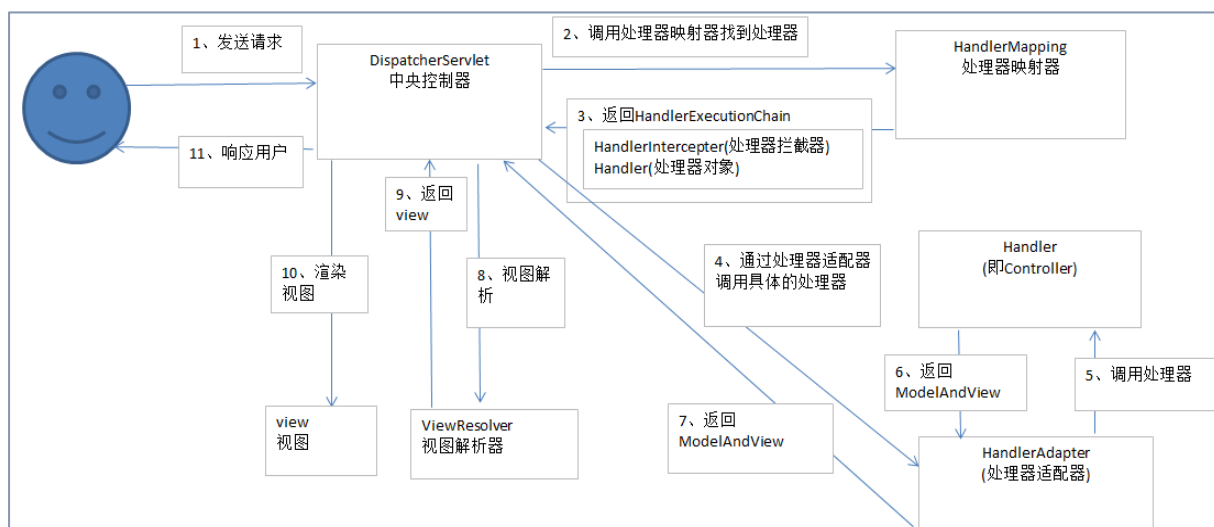
## 3 SpringMVC 核心和工作流程

### 3.1 Spring web mvc 架构

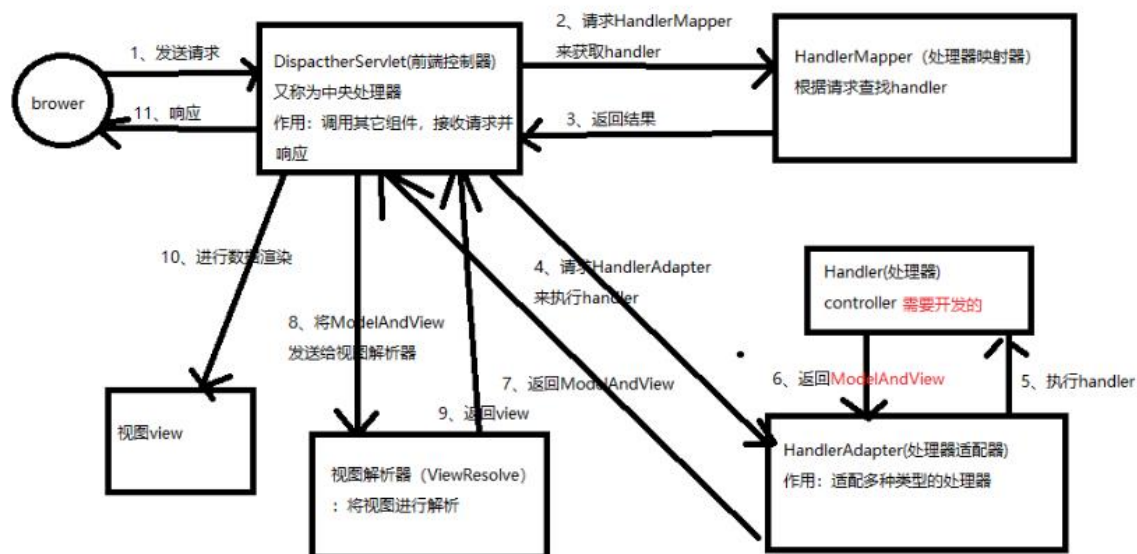
#### 3.1.1 架构图

在最简单的 Spring MVC 应用程序中，控制器是唯一的你需要在 Java web 部署描述文件（即 web.xml 文件）中配置的 Servlet。Spring MVC 控制器——通常称作 Dispatcher Servlet，实现了前端控制器设计模式。并且每个 web 请求必须通过它以便它能够管理整个请求的生命周期。

当一个 web 请求发送到 Spring MVC 应用程序，dispatcher servlet 首先接收请求。然后它组织那些在 Spring web 应用程序上下文配置的（例如实际请求处理控制器和视图解析器）或者使用注解配置的组件，所有的这些都需要处理该请求。



### 3.1.2 架构流程



第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求 HandlerMapping 查找 Handler

可以根据 xml 配置、注解进行查找

第三步：处理器映射器 HandlerMapping 向前端控制器返回 Handler

第四步：前端控制器调用处理器适配器去执行 Handler

第五步：处理器适配器去执行 Handler

第六步：Handler 执行完成给适配器返回 ModelAndView

第七步：处理器适配器向前端控制器返回 ModelAndView

ModelAndView 是 springmvc 框架的一个底层对象，包括 Model 和 view

第八步：前端控制器请求视图解析器去进行视图解析

根据逻辑视图名解析成真正的视图(jsp)

第九步：视图解析器向前端控制器返回 View

第十步：前端控制器进行视图渲染

视图渲染将模型数据(在 ModelAndView 对象中)填充到 request 域

第十一步：前端控制器向用户响应结果

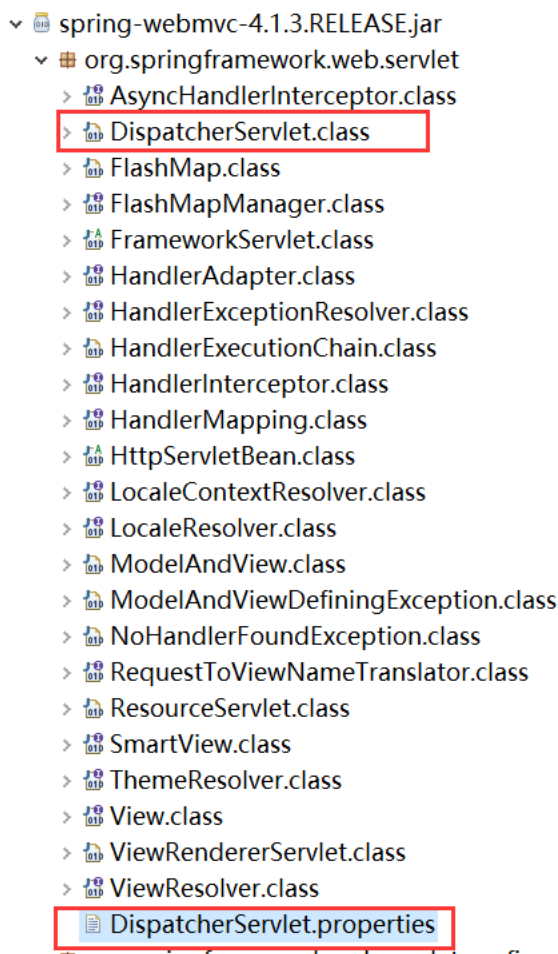
## 3.2 DispatcherServlet 前端控制器

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。作用接收请求，响应结果，相当于转发器，中央处理器

DispatcherServlet 作为 springmvc 的中央调度器存在，DispatcherServlet 创建时会默认从 DispatcherServlet.properties 文件加载 springmvc 所用的各各组件，如果在 springmvc.xml 中配置了组件则以 springmvc.xml 中配置为准，DispatcherServlet 的存在降低了 springmvc

各组件之间的耦合度。

\*\*在 SpringMVC 的 jar 包中含有一个默认配置文件，如果没有在 springmvc.xml 配置，就默认使用 DispatcherServlet.properties 的配置如下图：



### 3.3 HandlerMapping 处理器映射器

HandlerMapping 负责根据 request 请求找到对应的 Handler 处理器及 Interceptor 拦截器，将它们封装在 HandlerExecutionChain 对象中给前端控制器返回。

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。

作用：根据请求的 url 查找 handler

**BeanNameUrlHandlerMapping** 处理器映射器，根据请求的 url 与 spring 容器中定义的 bean 的 name 进行匹配，从而从 spring 容器中找到 bean 实例。

```
<!-- 处理器映射器 HandlerMapping -->
<!-- 将bean的名称作为请求的url: 同时url要以.action结尾 -->
<bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"></bean>
```

## 3.4 HandlerAdapter 处理器适配器

HandlerAdapter 会根据适配器接口对后端控制器进行包装（适配），包装后即可对处理器进行执行，通过扩展处理器适配器可以执行多种类型的处理器，这里使用了适配器设计模式。

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。

作用：按照特定规则 (HandlerAdapter要求的规则) 去执行Handler

注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler

**SimpleControllerHandlerAdapter**简单控制器处理器适配器，所有实现了org.springframework.web.servlet.mvc.Controller 接口的Bean通过此适配器进行适配、执行。

要求编写的Handler需要实现Controller接口。

适配器配置如下：

```
<!-- 处理器适配器 AdapterHandler -->
<!-- SimpleControllerHandlerAdapter: 简单控制器处理适配器 -->
<!-- 可以执行那种处理器呢？所有实现了Controller接口的处理器 -->
<bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"></bean>
>
```

## 3.5 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。

作用：进行视图解析，根据逻辑视图名解析成真正的视图

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <!-- 配置 jsp 路径的前缀和后缀 -->
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

将 modelAndView.setViewName("/WEB-INF/jsp/user.jsp");更改为  
modelAndView.setViewName("user");

## 4 SpringMVC 注解开发

### 4.1 新建 web 项目，导入 jar，配置前端控制器

此步骤和上个案例一致

### 4.2 springmvc 配置文件—注解方式

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

    <!-- 配置注解的处理器映射器 -->
    <bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandle
rMapping"></bean>
    <!-- 配置注解的处理器适配器 -->
    <bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandle
rAdapter"></bean>

</beans>
```

- **RequestMappingHandlerMapping**: 注解式处理器映射器，对类中标记@ResquestMapping的方法进行映射，根据 ResquestMapping 定义的 url 匹配 ResquestMapping 标记的方法，匹配成功返回 HandlerMethod 对象给前端控制器，HandlerMethod 对象中封装 url 对应的方法 Method。
- **RequestMappingHandlerAdapter**: 注解式处理器适配器，对标记@ResquestMapping 的方法进行适配。



## 4.3 注解的 Controller 开发

### 4.3.1 注解的 Controller 代码

在 com.hpe.controller 下新建 FirstController

```
@Controller
public class FirstController {
    // @RequestMapping: 请求的url, 将请求的url与方法对应起来, 一个url对应一个方法
    // .action可加可不加
    @RequestMapping("/first")
    public ModelAndView first() {
        // 创建ModelAndView 填充数据、设置视图
        ModelAndView mv = new ModelAndView();
        // 向模型对象中填充数据 相当于request.setAttribute()
        mv.addObject("msg", "这是我的第一个Spring Mvc程序");
        // 设置视图
        // mv.setViewName("/WEB-INF/jsp/user.jsp");
        mv.setViewName("user");
        return mv;
    }
}
```

与实现了 Controller 接口的方式相比, 使用了注解的方式显然更加简单。同时, Controller 接口的实现类只能处理一个单一的请求动作, 二基于注解的控制器可以同时处理多个请求动作, 在使用上更加的灵活。

**@RequestMapping:** 定义请求 url 到处理器功能方法的映射  
使用: 标注在方法上、标注在类上。

### 4.3.2 组件扫描器

使用组件扫描器省去在 spring 容器配置每个 controller 类的繁琐。使用<context:component-scan 自动扫描标记@Controller 的控制器类, 配置如下:

```
<context:component-scan base-package="com.hpe.controller"></context:component-scan>
```

## 4.4 配置视图解析器

```
<!-- 视图解析器, 支持jsp -->
```

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

配置视图然后运行，测试成功。

## 4.5 <mvc:annotation-driven> 注解驱动

<mvc:annotation-driven> 会自动注册 RequestMappingHandlerMapping 与 RequestMappingHandlerAdapter 两个 Bean,这是 Spring MVC 为@Controller 分发请求所必需的,并且提供了数据绑定支持, @NumberFormatannotation 支持, @DateTimeFormat 支持,@Valid 支持读写 XML 的支持 (JAXB) 和读写 JSON 的支持 (默认 Jackson) 等功能。

```
<!-- 注解驱动, 会自动的注册许多bean, 包括注解映射器和注解适配器 -->
<mvc:annotation-driven></mvc:annotation-driven>
```

**\*\*重点, 常用开发方式**

# 5 Springmvc 的注解详解

## 5.1 @RequestMapping

通过 RequestMapping 注解可以定义不同的处理器映射规则。

### 5.1.1 注解的使用

- 1、标注在方法上。
- 2、标注在类上

当标注在类上时, 该类中所有的方法都将映射到 value 属性值指定的路径下。

示例如下:

```
@Controller
//@RequestMapping注解在类上
@RequestMapping(value="second")
public class SecondController {
    //@RequestMapping注解在方法上
    @RequestMapping("/hello")
```

```

public ModelAndView hello(){
    ModelAndView mv = new ModelAndView();
    mv.addObject("msg", "hello everyone!!!");
    mv.setViewName("hello");
    return mv;
}
}

```

访问路径为: [http://localhost:8081/20180720\\_springmvc\\_02/second/hello.action](http://localhost:8081/20180720_springmvc_02/second/hello.action)

## 5.1.2 注解的属性

属性名	描述
value	指定请求的实际地址。默认属性支持多路径和传值。
method	指定请求的 method 类型, GET、POST、PUT、DELETE 等 不配置的话任何请求类型都支持 如果支持多个请求方式则需要通过{}写成数组的形式
consumes	指定处理请求的提交内容类型 (Content-Type), 例如 application/json, text/html;
produces	指定返回的内容类型, 仅当 request 请求头中的(Accept)类型中包含该指定类型才返回;
params	指定 request 中必须包含某些参数值时, 才让该方法处理。
headers	指定 request 中必须包含某些指定的 header 值, 才能让该方法处理请求。

所有的属性都是可选的, 但是其默认的属性是 value。

### ● value 多路径示例:

```

//value多路径 多个使用{}
@RequestMapping(value={"/hello01","/100"})
public ModelAndView hello01(){
    ModelAndView mv = new ModelAndView();
    mv.addObject("msg", "hello 01 everyone!!!");
    mv.setViewName("hello");
    return mv;
}

```

### ● method 示例:

```

// method 指定method后只能通过指定的请求方式访问 多个使用{}
//可使用@GetMapping和@PostMapping这种简化方式来直接指定请求方式
@RequestMapping(value = "hello02",method=RequestMethod.POST )

```

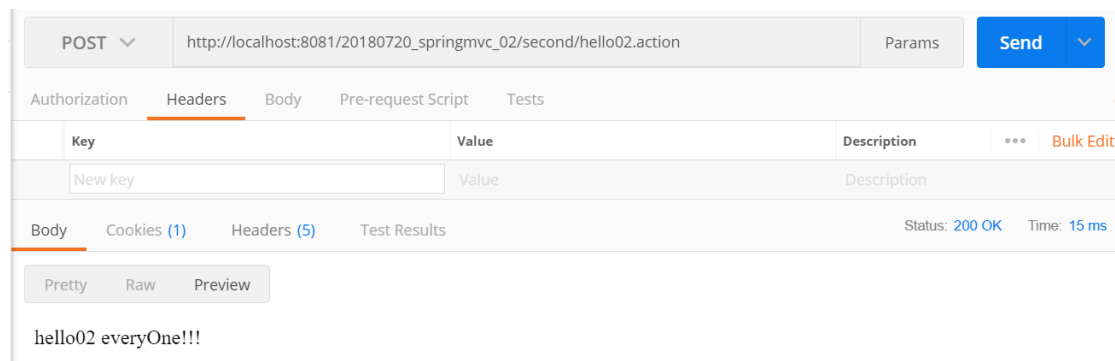
```

public ModelAndView hello02() {
    ModelAndView mv = new ModelAndView();
    mv.addObject("msg", "hello02 everyOne!!!");
    mv.setViewName("hello");
    return mv;
}

```

Get 方式访问：HTTP Status 405 - Request method 'GET' not supported

使用 Postman 采用 post 方法访问：



扩展：可使用 `@GetMapping` 和 `@PostMapping` 这种简化方式来直接指定请求方式。

## 5.2 参数类型和返回类型

举例说明几种常见的参数类型：

- `javax.servlet.http.HttpServletRequest`：
- `javax.servlet.http.HttpServletResponse`
- `org.springframework.ui.Model`
- java对象，包括基本类型
- io对象、注解等等

常用的几个返回类型

- `ModelAndView`、
- `String`、
- `void`、
- `Map`等

### 5.2.1 返回 ModelAndView

controller 方法中定义 `ModelAndView` 对象并返回，对象中可添加 model 数据、指定 view。  
示例一

```

//返回类型为ModelAndView 参数类型HttpServletRequest HttpServletResponse
//通过request取值，返回值

```

```

@RequestMapping("hello03")
public ModelAndView hello03(HttpServletRequest request, HttpServletResponse response) {
    ModelAndView mv = new ModelAndView();
    String name = (String) request.getParameter("name");
    System.err.println(name);
    request.setAttribute("msg", "hello03 everyOne!!!");
    request.getSession().setAttribute("user", "jack");
    mv.setViewName("hello");
    return mv;
}

```

通过 项目路径/second/hello03.action?name=rose 的方式访问

## 5.2.2 返回 void

示例二：返回 void 的类型

```

// 返回类型void 通过response指定响应结果
@RequestMapping("hello04")
public void hello04(HttpServletRequest request, HttpServletResponse response) {
    response.setCharacterEncoding("utf-8");
    response.setContentType("application/json;charset=utf-8");
    try {
        response.getWriter().write("hello04 everyOne!!!");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//返回类型void request转发
@RequestMapping("hello06")
public void hello06(HttpServletRequest request, HttpServletResponse response) {
    try {
        request.getRequestDispatcher("/first.action").forward(request, response);
    } catch (ServletException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//返回类型void response重定向
@RequestMapping("hello07")
public void hello07(HttpServletRequest request, HttpServletResponse response) {
    try {

```

```
        response.sendRedirect("hello.action");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## 5.2.3 返回字符串

### 5.2.3.1 逻辑视图名

controller 方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

//指定逻辑视图名，经过视图解析器解析为jsp物理路径

```
// 返回类型String 参数类型Model
@RequestMapping("hello08")
public String hello05(Model model) {
    ModelAndView mv = new ModelAndView();
    model.addAttribute("msg", "hello08 everyone!!!");
    return "hello";
}
```

### 5.2.3.2 Redirect 重定向

```
// redirect 重定向
@RequestMapping("hello10")
public String hello10() {
    return "redirect:/first.action";
}
```

redirect 方式相当于“response.sendRedirect()”，转发后浏览器的地址栏变为转发后的地址，因为转发即执行了一个新的 request 和 response。

由于新发起一个 request 原来的参数在转发时就不能传递到下一个 url，如果要传参数可以 /item/queryItem.action 后边加参数，如下：

/item/queryItem?...&... ..

### 5.2.3.3 forward 转发

```
// forward转发
@RequestMapping("hello09")
```

```
public String hello09() {  
    return "forward:/first.action";  
}
```

forward 方式相当于“request.getRequestDispatcher().forward(request,response)”，转发后浏览器地址栏还是原来的地址。转发并没有执行新的 request 和 response，而是和转发前的请求共用一个 request 和 response。所以转发前请求的参数在转发后仍然可以读取到。

## 6 数据绑定

在执行程序时，Spring MVC 会根据客户端请求参数的不同，将请求消息中的信息以一定的方式转换并绑定到控制类的方法参数中，这种将请求消息数据与后台方法参数建立连接的过程就是 Spring MVC 的数据绑定。

处理器适配器在执行 Handler 之前需要把 http 请求的 key/value 数据绑定到 Handler 方法形参数上。

### 6.1 默认支持的数据类型

处理器形参中添加如下类型的参数处理适配器会默认识别并进行赋值。

- HttpServletRequest：通过 request 对象获取请求信息
- HttpServletResponse：通过 response 处理响应信息
- HttpSession：通过 session 对象得到 session 中存放的对象
- Model/ModelMap：ModelMap 是 Model 接口的实现类，作用是将数据填充到 request 域。

```
/**  
 * @ClassName: ThirdController  
 * @Description: 数据绑定的案例  
 */  
@Controller  
@RequestMapping(value="/third")  
public class ThirdController {  
    @RequestMapping(value="/selectUser")  
    public String selectUser(HttpServletRequest request){  
        String id = request.getParameter("id");  
        System.err.println("id:"+id);  
        return "user";  
    }  
}
```

通过浏览器访问 项目路径/third/selectUser.action?id=1, 后台会打印出 id=1, 说明后台方法已从请求中正确的获取到了 id 的参数信息, 这说明使用默认的 HttpServletRequest 参数类型已经完成了数据绑定。

```
//session类型 取参
@RequestMapping(value="/getSession")
public String getSession(HttpSession session){
    String id = (String) session.getAttribute("id");
    System.err.println("id:"+id);
    return "user";
}

//session类型
@RequestMapping(value="/setSession")
public String setSession(HttpSession session){
    session.setAttribute("id","2");
    return "user";
}
```

## 6.2 简单数据类型

- 当请求的参数名称和处理器形参名称一致时会将请求参数与形参进行绑定。
- 简单数据类型的绑定, 指的 java 中几种基本数据类型的绑定, 如 int, String、double、Boolean 等。

```
//简单数据类型
@RequestMapping("/selectUser2")
public String selectUser2(int id, String name, boolean boo){
    System.err.println("id: "+id+",name is:"+name+",boo is "+boo);
    return "user";
}
```

访问示例 项目/third/selectUser2.action?id=3&name=jack&boo=true

**\*\*注意:** 当前端请求中参数名后天后天控制器类中的形参名不一样时, 无法进行数据绑定。Spring MVC 提供了@RequestParam 来进行间接数据绑定。

@RequestParam 的属性如下:

- **value:** 参数名字, 即入参的请求参数名字, 如value= "item\_id"表示请求的参数区中的名字为item\_id的参数的值将传入;
- **required:** 是否必须, 默认是true, 表示请求中一定要有相应的参数, 否则将报:  
TTP Status 400 - Required Integer parameter 'XXXX' is not present
- **defaultValue:** 默认值, 表示如果请求中没有同名参数时的默认值



```
// @RequestParam的应用
@RequestMapping("/selectUser3")
public String selectUser3(@RequestParam(value="user_id", required=true) int
id ,@RequestParam(value="user_name", defaultValue="rose") String name) {
    System.err.println("id: " + id+ ",name is:" + name);
    return "user";
}
```

/third/selectUser3.action?user\_id=3&user\_name=jack

形参名称为 id，但是这里使用 value="user\_id"限定请求的参数名为 user\_id，所以页面传递参数的名必须为 user\_id。

这里通过 required=true 限定 user\_id 参数为必需传递，如果不传递则报错误，可以使用 defaultvalue 设置默认值，即使 required=true 也可以不传参数值

## 6.3 简单 POJO 类型

请求中传递多个不同类型的参数类型，可使用 POJO 类型进行数据绑定。

一个简单的注册案例如下：

步骤一：新建 pojo 对象。

```
public class User {
    private int id;
    private String username;// 用户姓名
    private String password;// 密码
}
```

步骤二：新建注册的控制器，创建跳转注册页面和接收注册信息的方法

```
@Controller
@RequestMapping("register")
public class RegisterController {
    //跳转到注册页面
    @RequestMapping("/toRegister")
    public String toRegister(){
        return "register";
    }

    //注册，通过pojo 绑定参数
    @RequestMapping("/register")
    public String register(User user){
        System.err.println("注册信息为: "+user.toString());
        return "hello";
    }
}
```

```
<%@ 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>注册</title>
</head>
<body>
    <form action="${pageContext.request.contextPath}/register/register.action"
method="post">
        <tr>
            <td>用户名: </td>
            <td><input type="text" name="username"></td>
        </tr><br>
        <tr>
            <td>密 &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&码: </td>
            <td><input type="text" name="password"></td>
        </tr><br>
        <tr> <td colspan="2"><input type="submit" value="注册"> </td>
        </tr>
    </form>
</body>
</html>
```

#### 步骤四：测试，访问注册页

```
<!-- 配置编码过滤器 -->
```

```

<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>

```

上述代码中，通过<filter-mapping>元素的配置会拦截前端页面中的所有的请求，并且交给名字为 EncodingFilter 的编码过滤器来进行处理。在<filter>中，首先配置了编码过滤器类为 org.springframework.web.filter.CharacterEncodingFilter，然后通过初始化参数设置统一的编码格式为 UTF-8。这样所有的请求信息内容都会以 UTF-8 的编码格式进行解析。

对于 get 请求中文参数出现乱码解决方法有两个：

修改 tomcat 配置文件添加编码与工程编码一致，如下：

```

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

```

另外一种方法对参数进行重新编码：

```

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

```

ISO8859-1 是 tomcat 默认编码，需要将 tomcat 编码后的内容按 utf-8 编码

## 6.4 \*包装 POJO 类型

所谓的包装 POJO，就是在一个 POJO 钟包含另一个简单 POJO。

案例如下：

步骤一：新建 pojo 对象 car，它属于某个 User 用户

```

public class Car {
  private int carId;//id
  private String name;// 汽车品牌
  private User user;//属于哪个人

```

## 步骤二：创建跳转汽车查询页面和接收查询信息的方法

```
// 跳转到汽车查询页面
@RequestMapping("/toCar")
public String toCar() {
    return "car";
}

// 查询汽车信息，通过复杂pojo 绑定参数
@RequestMapping("/selectCar")
public String selectCar(Car car) {
    System.err.println("汽车信息为: " + car.toString());
    return "hello";
}
```

## 步骤三：新建汽车查询页面 car.jsp

```
<form action="${pageContext.request.contextPath}/register/selectCar.action"
method="post">
    <tr>
        <td>汽车品牌: </td>
        <td><input type="text" name="name"></td>
    </tr><br>
    <tr>
        <td>所属用户: </td>
        <td><input type="text" name="user.username"></td>
    </tr><br>
    <tr> <input type="submit" value="查询"> </tr>
</form>
```

注：使用包装 POJO 类型绑定数据时，前端请求的参数名必须符合：

- 如果查询条件参数是包装类的直接基本属性，则参数名直接用对应的属性名，如上述案例中的 name。
- 如果查询条件参数是包装类中 POJO 的子属性，则参数名必须为【对象.属性】，如上述代码中的 user.username

## 步骤四：测试

汽车品牌: 宝马

所属用户: jack

查询

控制台打印信息：

汽车信息为: Car [carId=0, name=宝马, user=User [id=0, username=jack, password=null]]

## 6.5 \*自定义参数绑定

一般情况下,使用基本数据类型和 POJO 类型的参数数据就已经能够满足需求,容纳后有些特殊类型的参数是无法在后台进行直接转换的,例如日期数据 就需要自定义转换器 (Converter) 来进行数据绑定。

**Converter** 用于将一种类型的对象转换为另一种类型的对象。

自定义 Converter 类需要实现 org.springframework.core.convert.converter.Converter 接口,该接口的代码如下所示:

```
public interface Converter<S, T>{
    T convert(S source);
}
```

在上述接口代码中,泛型中的 S 表示源类型, T 表示目标类型,而 convert(S source)表示接口中的方法。

实现方法示例:

步骤一: 新建日期转换类 DateConverter

```
//定义的日期转换器
public class DateConverter implements Converter<String, Date>{
    //定义的日期格式
    private String dataPattern = "yyyy-MM-dd HH:mm:ss";
    @Override
    public Date convert(String source) {
        SimpleDateFormat sdf = new SimpleDateFormat(dataPattern);
        try {
            return sdf.parse(source);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

步骤二: springmvc.xml 配置

```
<!-- 显示的装配自定义类型转换器 -->
<mvc:annotation-driven conversion-
service="conversionService"></mvc:annotation-driven>

<!-- 自定义类型转换器配置 -->
<bean id="conversionService" class="
org.springframework.context.support.ConversionServiceFactoryBean">
```

```

        <property name="converters">
            <set>
                <bean class="com.hpe.convert.DateConverter"></bean>
            </set>
        </property>
    </bean>

```

### 步骤三：新建日期控制类测试

```

@Controller
@RequestMapping("date")
public class DateController {
    // 查询汽车信息，通过复杂pojo 绑定参数
    @RequestMapping("/selectDate")
    public String selectDate(Date date) {
        System.err.println("日期信息为: " + date);
        return "hello";
    }
}

```

地址栏输入 项目路径 /date/selectDate.action?date=2018-07-22 11:33:01.

控制台打印信息：日期信息为：Sun Jul 22 11:33:01 CST 2018

## 6.6 集合类

### ● 数组

应用场景：批量删除、批量导出数据等，前端请求需要传递到后台多个相同名称参数。

页面定义如下：

页面选中多个 checkbox 向 controller 方法传递

```

<form action="${pageContext.request.contextPath}/register/delete.action"
        method="post">
    <table width="25%" border="1">
        <tr>
            <td><input type="checkbox" name="ids" value="1"></td>
            <td>jack</td>
        </tr>
        <tr>
            <td><input type="checkbox" name="ids" value="2"></td>
            <td>tom</td>
        </tr>
    </table>

```

```

        <td><input type="checkbox" name="ids" value="3"></td>
        <td>rose</td>
    </tr>
</table>
<tr>
    <td><input type="submit" value="删除">
</td>
</tr>
</form>

```

Controller 方法中可以用 Integer[] 接收，定义如下：

```

// 批量删除 参数类型为 数组
@RequestMapping("/delete")
public String delete(Integer[] ids) {
    if(ids!=null){
        for(Integer id:ids){
            System.err.println("删除的id为: " + id);
        }
    }
    return "hello";
}

```

## ● \*List

应用场景：批量修改等

List 中存放对象，并将定义 List 放在包装类中，action 层使用包装对象接收。

注意：后台方法中不支持直接使用集合形参进行数据绑定，所以需要使用包装 POJO 作为形参，然后在包装 POJO 中包装一个集合属性。

List 中对象：

用户对象 UserVo

```

public class UserVo {
    private List<User> users;//用户列表
}

```

页面定义如下：

```

<form action="${pageContext.request.contextPath}/register/updateUser.action"
    method="post">
    <table width="25%" border="1">
        <tr>
            <td>选择</td>
            <td>用户名</td>
            <td>密码</td>
        </tr>
        <tr>
            <td><input type="checkbox" name="users[0].id" value="1"></td>

```

```

        <td><input type="text" name="users[0].username"
            value="jack"></td>
        <td><input type="text" name="users[0].password"
            value="123456"></td>
    </tr>
    <tr>
        <td><input type="checkbox" name="users[1].id" value="2"></td>
        <td><input type="text" name="users[1].username"
value="tom"></td>
        <td><input type="text" name="users[1].password"
            value="abcd"></td>
    </tr>
</table>
<tr>
    <td><input type="submit" value="修改">
</td>
</tr>
</form>

```

Controller 方法定义如下:

```

// 批量修改
@RequestMapping("/updateUser")
public String updateUser(UserVo userList) {
    List<User> users = userList getUsers();
    if (userList != null) {
        for (User user: users) {
            System.err.println("修改的用户: " + user);
        }
    }
    return "hello";
}

```

## 6.7 \*与 struts2 不同

- 1、springmvc 的入口是一个 servlet 即前端控制器，而 struts2 入口是一个 filter 过滤器。
- 2、springmvc 是基于方法开发(一个 url 对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例。
- 3、Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，springmvc 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 request 域传输到页面。Jsp 视图解析器默认使用 jstl。



## 7 JSON 数据交互

### 7.1 JSON 概述

[JSON](#)([JavaScript](#) Object Notation, JS 对象简谱) 是一种轻量级的数据交换格式。

它基于 [ECMAScript](#) (欧洲计算机协会制定的 js 规范)的一个子集,

采用完全独立于编程语言的文本格式来存储和表示数据。

简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。易于人阅读和编写,同时也易于机器解析和生成,并有效地提升网络传输效率。

#### 7.1.1 JSON 语法规则

在 JS 语言中,一切都是对象。因此,任何支持的类型都可以通过 JSON 来表示,例如字符串、数字、对象、数组等。但是对象和数组是比较特殊且常用的两种类型:

- 对象表示为键值对,以冒号分割
- 数据由逗号分隔
- 花括号保存对象
- 方括号保存数组

#### 7.1.2 JSON 类型

JSON 键值对是用来保存 JS 对象的一种方式,和 JS 对象的写法也大同小异,键/值对组合中的键名写在前面并用双引号 "" 包裹,使用冒号 : 分隔,然后紧接着值:

例: `{"name": "Json"}`

在 JS 语言中,一切都是对象。因此,任何支持的类型都可以通过 JSON 来表示,例如字符串、数字、对象、数组等。但是对象和数组是比较特殊且常用的两种类型。

- 对象: 对象在 JS 中是使用花括号包裹 {} 起来的内容,数据结构为 {key1: value1, key2: value2, ...} 的键值对结构。在面向对象的语言中, key 为对象的属性, value 为对应的值。键名可以使用整数和字符串来表示。值的类型可以是任意类型。

例: `{"city": "jining", "postcode": "272000"}`

- 数组: 数组在 JS 中是方括号 [] 包裹起来的内容,数据结构为 ["java", "javascript", "vb", ...] 的索引结构

例: `{"people": [{"name": "jack", "sex": "男"}, {"name": "rose", "sex": "女"}]}`

## 7.2 JSON 数据转换

`HttpMessageConverter<T>`接口：用于将请求信息中的数据转换为一个类型为 `T` 的对象，并且将类型为 `T` 的对象绑定到请求方法中的参数中，或者将对象转换为响应信息传递给浏览器显示。

Spring 为 `HttpMessageConverter<T>`接口提供了很多的实现类，这些实现类可以对不同类型的数据信息进行转换。`MappingJackson2HttpMessageConverter` 是 SpringMVC 默认处理 JSON 格式请求响应的实现类。该实现类需要使用 Jackson 开源包读取 JSON 数据，将 java 对象转成 JSON 对象和 XML 文档，同时也可以将 JSON 对象和 XML 文档转换为 java 对象。

Jackson 包，建议版本为 2.8.8




- `jackson-annotations-2.8.8.jar`：JSON 转换注解包
- `jackson-core-2.8.8.jar`：JSON 转换核心包
- `jackson-databind-2.8.8.jar`：JSON 转换的数据绑定包

在使用注解开发时，有两个重要的 JSON 转换注解，如下：

- **@RequestBody**：用于读取 http 请求的内容(字符串)，通过 springmvc 提供的 `HttpMessageConverter` 接口将读到的内容转换为 json、xml 等格式的数据并绑定到 controller 方法的参数上。
- **@ResponseBody**：该注解用于将 Controller 的方法返回的对象，通过 `HttpMessageConverter` 接口转换为指定格式的数据如：json、xml 等，通过 Response 响应给客户端

## 7.3 JSON 数据转换案例

步骤一：新建 web 项目，导入 jar 包

 `jackson-annotations-2.8.8.jar`  
 `jackson-core-2.8.8.jar`  
 `jackson-databind-2.8.8.jar`

步骤二：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_3_0.xsd"
    id="WebApp_ID" version="3.0">
```

```
<display-name>20180723_springmvc_03</display-name>
<welcome-file-list>
    <welcome-file>/jsp/index.jsp</welcome-file>
</welcome-file-list>

<servlet>
    <!-- 配置前端控制器 -->
    <servlet-name>springmvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 初始化时加载配置文件 -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <!-- 表示容器在启动时 当前Servlet的加载顺序 -->
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- 配置编码过滤器 -->
<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>

</web-app>
```

步骤三：创建 Spring MVC 的核心配置文件

```
<?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:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-
package="com.hpe.controller"></context:component-scan>
    <!-- 注解驱动，会自动注册许多bean，包括注解映射器和注解适配器 -->
    <mvc:annotation-driven></mvc:annotation-driven>
    <!-- 配置静态资源的访问映射，此配置中的文件，将不会被前端控制前拦截 -->
    <mvc:resources location="/js/" mapping="/js/**"></mvc:resources>
    <!-- 视图解析器，支持jsp -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>

```

- 注：其中，<mvc:annotation-driven>会自动注册 RequestMappingHandlerMapping 与 RequestMappingHandlerAdapter 两个 Bean，并且提供了读写 XML 和读写 JSON 等功能。
- 静态资源访问<mvc:resources ...>中的两个重要属性 location 和 mapping  
Location:用于定位需要访问的本地静态资源文件路径，具体到某个文件夹  
Mapping:匹配静态资源全路径，其中"/\*\*"表示文件夹及其子文件夹下的某个具体文件

步骤四：创建 User 类，用于封装请求数据

```

public class User {
    private int id;
    private String username; // 用户姓名
    private String password; // 密码
}

```

步骤五：创建 jsp 页面来测试 JSON 数据交互。需要引用 jquery.js

```

<%@ 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>

```



```
}  
</script>  
</html>
```

注：在 ajax 中包含了 3 个重要的属性，如下：

- data: 请求时携带的数据，当使用 JSON 格式时，要注意编写规范。  
\*JSON.stringify() 是把 json 对象解析出 json 字符串，因为@RequestBody 是用来读取请求中的字符串。  
如果传输的为 json 对象，后台读取参数无须使用@RequestBody 注解。
- contentType: 默认为 application/x-www-form-urlencoded，当使用 JSON 格式时，值必须为 **application/json**
- dataType: 当响应数据为 JSON 时，可以定义 dataType 属性，并且值必须为 json。

步骤六：创建控制类 UserController

```
@Controller  
public class UserController {  
    @RequestMapping("testJson")  
    @ResponseBody  
    public User testJson(@RequestBody User user){  
        System.err.println(user);  
        return user;  
    }  
}
```

方法中的@RequestBody 注解用于将请求体中的 JSON 格式数据绑定到形参 user 上，@ResponseBody 注解用于直接返回 User 对象（会默认转换为 JSON 格式数据进行响应）。

步骤七：测试



控制台打印信息：User [id=0, username=jack, password=123456]

\*\*注：实际开发中多使用请求 key/value 数据（传输 JSON 对象，无须@RequestBody 注解），响应 json 结果，方便客户端对结果进行解析。

## 7.4 RESTful 支持

REST（英文：**Representational State Transfer**，简称 **REST**），可以将它理解为一种

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

简单来说，RESTful 风格就是把请求参数变成请求路径的一种风格。

例如，传统的 URL 请求格式为：<http://...../queryUsers?id=1>

采用了 RESTful 风格后，其 URL 请求为：<http://.....queryUsers/1>

案例：采用 RESTful 风格查询用户信息，返回 JSON 格式的数据

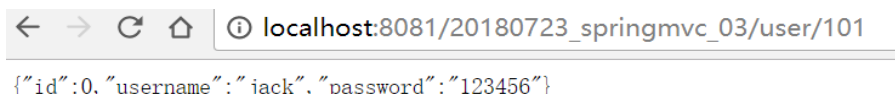
//接收RESTful风格的请求，其接收方式为GET

```
@RequestMapping(value="/user/{id}",method=RequestMethod.GET)
@ResponseBody
public User selectUser(@PathVariable String id){
    System.err.println("id is:"+id);
    User user = new User();
    if("101".equals(id)){
        user.setUsername("jack");
        user.setPassword("123456");
    }
    return user;
}
```

在上述代码中，`@RequestMapping(value="/user/{id}",method=RequestMethod.GET)` 注解用于匹配请求路径（包括参数）和方式。其中 `value="/user/{id}"` 表示可以匹配以 `"/user/{id}"` 结尾的请求，`id` 为请求中的动态参数。

`@PathVariable("id")` 注解用于接收并绑定请求参数，它可以将请求 URL 中的标量映射到方法的形参上。如果请求路径为 `"/user/{id}"`，即请求参数中的 `id` 和方法形参名称 `id` 一样，则 `@PathVariable` 后边 `("id")` 可以省略。

浏览器访问



`{\"id\":0,\"username\":\"jack\",\"password\":\"123456\"}`

## 8 拦截器

### 8.1 定义

Spring MVC 的处理器拦截器类似于 Servlet 开发中的过滤器 Filter，用于对处理器进行预处理和后处理，用于拦截用户请求并做响应的处理。例如通过拦截器可以进行权限验证、记录日志、判断用户是否登录等。

## 8.2 拦截器定义

实现 HandlerInterceptor 接口，如下：

```
public class MyIntercept implements HandlerInterceptor{

    /**
     * 该方法在控制器方法前执行
     * 返回true则继续向下执行，返回false中断执行
     * 适用于登录验证、权限拦截等
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.err.println("MyIntercept ....preHandle");
        return true;
    }

    /**
     * 该方法在控制器方法调用之后，且解析视图之前执行
     * 可对请求域中的模型数据和视图做出修改，比如加入公共数据
     */
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView mv) throws Exception {
        System.err.println("MyIntercept ....postHandle");
    }

    /**
     * 该方法在整个请求完成，即视图渲染结束之后执行
     * 可用于记录日志、清理资源等
     */
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.err.println("MyIntercept ....afterCompletion");
    }
}
```

## 8.3 拦截器配置

Spring MVC的配置文件配置如下：

```
<!-- 配置拦截器 -->
<mvc:interceptors>
```



```

<!-- 直接使用bean定义在<mvc:interceptors>下面的 Interceptor 将拦截所有请求 -
->

<!-- <bean class="com.hpe.intercept.MyIntercept"></bean> -->
<!-- 拦截器 -->
<mvc:interceptor>
    <!-- 配置拦截器作用的路径 -->
    <mvc:mapping path="/**"/>
    <!-- 配置不需要拦截器作用的路径 -->
    <mvc:exclude-mapping path="/" />
    <!-- 定义在<mvc:interceptor>下面的 Interceptor, 表示对匹配路径的请求才进
行拦截 -->

    <bean class="com.hpe.intercept.MyIntercept"></bean>
</mvc:interceptor>
</mvc:interceptors>

```

注: <mvc:interceptor>子元素的顺序必须按照上述顺序编写。

运行项目，测试效果。访问之前工程的json测试方法，控制台输出如下：

```

MyIntercept ....preHandle
User [id=0, username=qwe, password=123]
MyIntercept ....postHandle
MyIntercept ....afterCompletion

```

## 8.4 拦截器的执行流程

### 8.4.1 单个拦截器执行流程

单个拦截器的执行顺序是

1、preHandle 2、handle 3、postHandle 4、afterCompletion

### 8.4.2 多个拦截器执行顺序

当有多个拦截器同时工作时，它们的 preHandle()方法会按照配置文件中的拦截器配置的顺序执行，而它们的 postHandle()方法和 afterCompletion()方法会按照配置顺序的反序执行。

案例：

定义两个拦截器分别为：HandlerInterceptor1 和 HandlerInteptor2，每个拦截器的 preHandler 方法都返回 true。

控制台输出如下：

```
HandlerInterceptor1..preHandle..
```

```
HandlerInterceptor2..preHandle..
```

```
HandlerInterceptor2..postHandle..
```

```
HandlerInterceptor1..postHandle..
```

```
HandlerInterceptor2..afterCompletion..
```

```
HandlerInterceptor1..afterCompletion..
```

## 9 \*文件上传下载

### 9.1 文件上传

#### 9.1.1 form 表单

多数文件上传都是通过表格形式提交给后台服务器的，因此，要实现文件上传功能，就需要提供一个文件上传的表单。需要满足以下 3 个条件

- Form 表单的 method 属性设置为 post
- Form 表单的 enctype 属性设置为 multipart/form-data
- 提供<input type="file" name="filename"/>的文件上传输入框

文件上传的示例代码如下：

```
<form action="**url" method="post" enctype="multipart/form-data">
    <input type="file" name="filename" multiple="multiple">
    <input type="submit" value="上传">
</form>
```

上述代码中，除了满足表单所必须的 3 个条件外，在<input>元素中还增加了一个 multiple 属性。该属性是 HTML5 中的新属性，表示可以同时选择多个文件进行上传。

#### 9.1.2 配置解析器

当客户端 form 表单的 enctype 属性设置为 multipart/form-data 时，浏览器就会采用二级制流的方式来处理表单数据，服务器端就会对文件上传的请求进行解析处理。SpringMVC 为文件上传提供了直接的支持，这种支持是通过 MultipartResolver(多部件解析器)对象实现的。MultipartResolver 是一个接口对象，需要通过它的实现类 CommonsMultipartResolver 来完成文件上传。具体配置如下：

```
<!-- 文件上传 -->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 设置编码格式，也页面一致 -->
    <property name="defaultEncoding" value="UTF-8"></property>
    <!-- 设置允许的上传文件最大值(2MB) 单位为字节 -->
    <property name="maxUploadSize" value="2097152"></property>
</bean>
```

注：bean 的 id 必须为 `multipartResolver`。默认配置的原因

### 9.1.3 依赖包

commons-fileupload-1.\*.\*.jar

commons-io-2.\*.jar

### 9.1.4 文件上传控制类

新建 FileUploadController

```
@Controller
public class FileUploadController {
    @RequestMapping("/toFileUpload")
    public String toFileUpload(){
        return "fileUpload";
    }
    /**
     * 文件上传
     */
    @RequestMapping(value="/fileUpload",method=RequestMethod.POST)
    public String fileUpload(@RequestParam("filename") MultipartFile file,
        HttpServletRequest request){
        try {
            //获取上传文件的原始名称
            String org_filename = file.getOriginalFilename();
            //设置上传文件的保存目录
            String dirPath = //"D:/upload/";
            request.getServletContext().getRealPath("/upload/");
            System.err.println(dirPath);
            File filePath = new File(dirPath);
            //如果保存文件的地址不存在，就先创建目录
```

```

        if(!filePath.exists()){
            filePath.mkdirs();
        }
        //使用UUID进行重命名
        String new_filename = UUID.randomUUID()+"_"+org_filename;
        System.err.println(new_filename);
        //使用MultipartFile 接口的方法 完成上传到指定位置
        file.transferTo(new File(dirPath+"/"+new_filename));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "index";
}
}

```

注意路径问题。到控制台输出的路径下去查找文件是否上传成功。

## 9.2 文件下载

SpringMVC 提供了一个 `ResponseEntity` 类型，使用它可以很方便地定义返回的 `HttpHeaders` 对象和 `HttpStatus` 对象，通过对这两个对象的设置，既可以完成下载文件时所需要的配置信息。

```

/**
 * 文件下载上传
 */
@RequestMapping(value = "/download")
public ResponseEntity<byte[]> download(HttpServletRequest request, String
filename) throws Exception {
    // 指定要下载的文件所在路径
    String path = request.getServletContext().getRealPath("/upload/");
    // 创建该文件对象
    File file = new File(path + "/" + filename);
    // 设置响应头
    HttpHeaders headers = new HttpHeaders();
    // 通知浏览器以下载的方式打开文件
    headers.setContentDispositionFormData("attachment", filename);
    // 定义以流的形式返回文件数据
    headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
    // 使用springmvc框架的ResponseEntity对象封装返回下载数据
    return new ResponseEntity<byte[]>(FileUtils.readFileToByteArray(file),
headers, HttpStatus.OK);
}

```

download 处理方法接收页面传递的文件名 filename 后, 使用 Apache Commons FileUpload 组件的 FileUtils 读取项目的上传文件, 并将其构建成 ResponseEntity 对象返回客户端下载。

使用 ResponseEntity 对象, 可以很方便的定义返回的 HttpHeaders 和 HttpStatus。上面代码中的 MediaType, 代表的是 Internet Media Type, 即互联网媒体类型, 也叫做 MIME 类型。在 Http 协议消息头中, 使用 Content-Type 来表示具体请求中的媒体类型信息。HttpStatus 类型代表的是 Http 协议中的状态。有关 MediaType 和 HttpStatus 类可以参考 Spring MVC 的 API 文档。

在页面中使用超链接进行文件下载, 点击即可下载

```
<a href="{pageContext.request.contextPath}/download?filename=2589e346-cebf-446a-b6a1-4cd57249bd73_one.png">文件下载</a>
```

[文件下载](#)



2589e346-cebf-....png

