SpringMVC课堂笔记

1、SpringMVC概述

1.1 SpringMVC概念

SpringMVC 也叫 Spring web mvc。是 Spring内置的一个MVC框架，在 Spring3.0 后发布。SpringMVC框架解决了WEB开发中常见的问题(参数接收、文件上传、表单验证等等)，而且使用简单，与Spring无缝集成。支持 RESTful风格的URL请求。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。

1.2 SpringMVC原理

在没有使用SpringMVC之前我们都是使用Servlet在做Web开发。但是使用Servlet开发在接收请求参数，数据共享，页面跳转等操作相对比较复杂。servlet是java进行web开发的标准，既然springMVC是对servlet的封装，那么很显然**SpringMVC底层就是Servlet，SpringMVC就是对Servlet进行深层次的封装。**

1.3 SpringMVC优势

1、基于 MVC 架构，功能分工明确。解决页面代码和后台代码的分离。

2、简单易用。SpringMVC 也是轻量级的，jar 很小。不依赖的特定的接口和类就可以开发一个注解的SpringMVC 项目。

3、作为Spring框架一部分，能够使用Spring的IoC和AOP。方便整合MyBatis,Hiberate,JPA等其他框架。

4、springMVC的注解强大易用。

2、MVC模式回顾

> 模型1：jsp+javabean模型---在jsp页面中嵌入大量的java代码
>
> 模型2：jsp+servlet+javabean模型---jsp页面将请求发送给servlet，由servlet调用javabean，再由servlet将制定jsp页面响应给用户。

模型2一般就是现在的MVC模式，也是我们一直使用的。

Model-View-Controller：模型--视图--控制器

Model： 模型层 javaBean 负责数据访问和业务处理 dao service pojo

View： 视图 JSP技术 负责收集和展示数据

Controller： 控制器 servlet技术 中间调度

控制器的工作：

1、接受客户端的请求（包括请求中携带的数据）

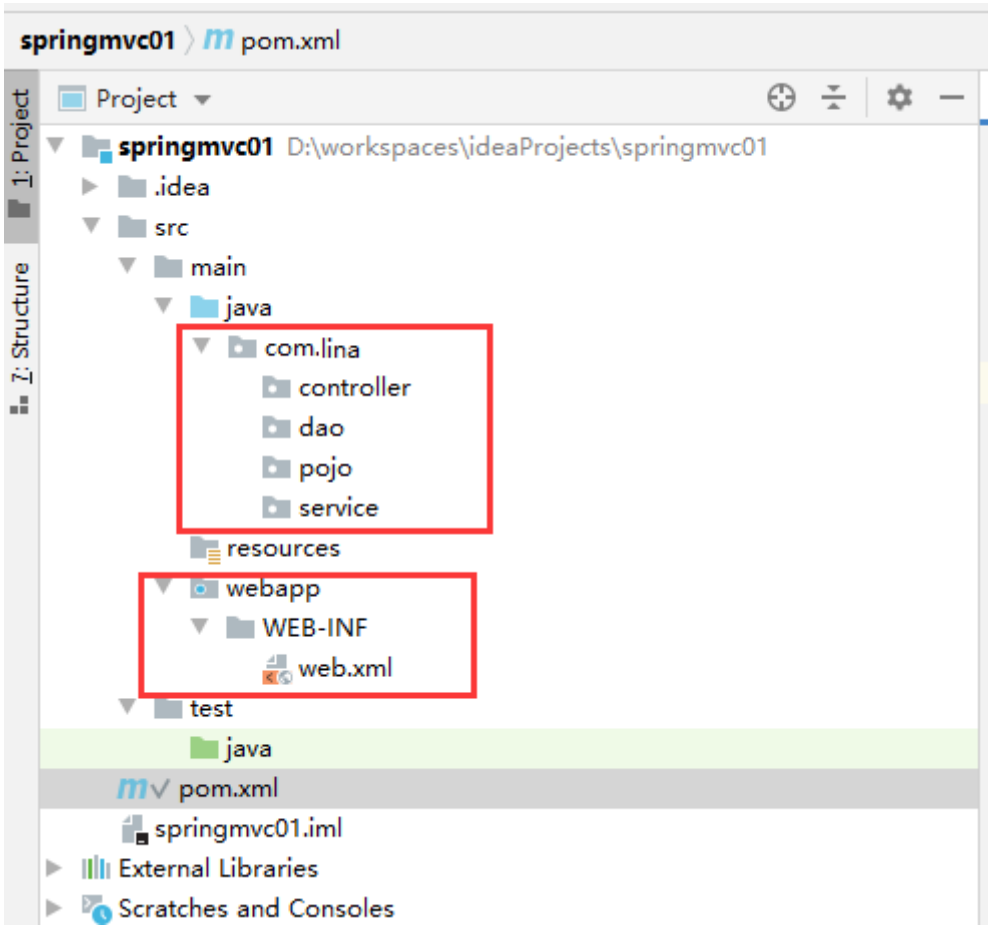
2、处理请求：调用后台的模型层中的业务逻辑

3、页面导航：处理完毕给出响应：JSP页面

3、入门程序

3.1 创建maven项目

创建项目并补齐目录结构



3.2 pom.xml文件添加依赖和插件

```
    <!--web项目-->
    <packaging>war</packaging>

    <dependencies>
        <!--spring-webmvc依赖-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>5.2.13.RELEASE</version>
        </dependency>
        <!--springmvc底层还是servlet，所以必须添加serlvet依赖-->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.1</version>
            <scope>provided</scope><!--插件运行的时候没有scope，插件启动可能会失败-->
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- 编码和编译和JDK版本 -->
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <!--tomcat插件-->
            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <path>/</path>
                    <port>8080</port>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

3.3 创建Spring和SpringMVC的配置文件

我们一般将除了 Controller 之外的所有 Bean 注册到 Spring 容器中，而将 Controller 注册到 SpringMVC 容器中。所以我们在 resources 目录下添加 applicationContext.xml 作为 spring 的配置，添加springmvc.xml作为springmvc的配置文件。

3.3.1 创建Spring配置文件applicationContext.xml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"
       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
        ">

    <!--spring的配置文件:除了控制器之外的bean对象都在这里扫描-->
    <context:component-scan base-package="com.lina.dao,com.lina.service"/>

</beans>
```

3.3.2 创建SpringMVC的配置文件springmvc.xml

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

    <!--springmvc的配置文件:控制器的bean对象都在这里扫描-->
    <context:component-scan base-package="com.lina.controller"/>
</beans>
```

3.4 在web.xml中进行Spring和SpringMVC配置

```
    <!--spring的配置-->
<context-param>
        <!--contextConfigLocation：表示用于加载 Bean的配置文件-->
        <param-name>contextConfigLocation</param-name>
        <!--指定spring配置文件的位置
            这个配置文件也有一些默认规则，它的配置文件名默认就叫 applicationContext.xml ，如果将这个配置文件放在 WEB-INF 目录下，那么这里就可以不用                    指定配置文件位置，只需要指定监听器就可以。
            这段配置是 Spring 集成 Web 环境的通用配置；一般用于加载除控制器层的 Bean（如 dao、service 等），以便于与其他任何web框架集成。
        -->
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
       <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
<!--SpringMVC的配置-->
    <!--
        前端控制器：所有的请求都会经过此控制器，然后通过此控制器分发到各个分控制器.
          前端控制器本质上还是一个Servlet，因为SpringMVC底层就是使用Servlet编写的
    -->
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 创建前端控制器的时候读取springmvc配置文件启动ioc容器 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath*:springmvc.xml</param-value>
        </init-param>
        <!-- Tomcat启动完毕就创建此对象 -->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- 配置拦截路径url，所有以.do结尾的请求都会被前端控制器拦截处理 -->
    <servlet-mapping>
```

```
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
<!--
SpringMVC的配置解析：
    1、servlet-class：前端控制器的完全限定名，在spring-webmvc-5.2.13.RELEASE.jar包中的org.springframework.web.servlet下
    2、load-on-startup：标记是否在Web服务器（这里是Tomcat）启动时会创建这个 Servlet 实例，即是否在 Web 服务器启动时调用执行该 Servlet 的 init()方
法，而不是在真正访问时才创建。 要求取值是整数。
        值大于0：表示容器在启动时就加载并初始化这个 servlet，数值越小，该 Servlet的优先级就越高，其被创建的也就越早
        值小于0或者省略：表示该 Servlet 在真正被使用时才会去创建。
        值相同：容器会自己选择创建顺序
    3、url-pattern：可以写为 / ，可以写为*.do 、*.action、*.mvc等形式，此处先写*.do,以后介绍不同写法的区别。
    4、init-param：表示了springmvc配置文件的名称和位置。如果没有配置，默认在项目的WEB-INF目录下找名称为 Servlet 名称-servlet.xml 的配置文件。
        如果没有配置，启用默认的规则：即如果配置文件放在 webapp/WEB-INF/ 目录下，并且配置文件的名字等于 DispatcherServlet 的名字+ -servlet（即这里的配置文件路径是 webapp/WEB-INF/dispatcherServlet-servlet.xml），如果是这样的话，可以不用添加 init-param 参数，即不用手动配置 springmvc 的配置文件，框架会自动加载。
        而一般情况下，配置文件是放在类路径下，即 resources 目录下。所以，在注册前端控制器时，还需要设置查找 SpringMVC 配置文件路径。

        其中contextConfigLocation属性：来自DispatcherServlet的父类FrameworkServlet，该类中的contextConfigLocation属性用来配置springmvc的路径和名称。
-->
```

3.5 创建控制器

```
package com.lina.controller;

import com.lina.service.TeamService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

/**
 * ClassName: TeamController
 *
 * @author wanglina
 * @version 1.0
 */
@Controller
public class TeamController {

    @Autowired
    private TeamService teamService;

    @RequestMapping("hello.do")
    public ModelAndView add(){
        System.out.println("TeamController----add---");
        teamService.add();
        ModelAndView mv=new ModelAndView();
```

```
        mv.addObject("teamName","湖人");//相当于
request.setAttrubuite("teanName","湖人");
        mv.setViewName("index");//未来经过springmvc的视图解析器处理，转换成物理资源路径，
相当于request.getRequestDispatcher("index.jsp").forward();
        //经过InternalResourceViewResolver对象的处理之后加上前后缀就变为了
/jsp/index.jsp
        return mv;
    }
}
```

```
package com.lina.service;
import org.springframework.stereotype.Service;
@Service
public class TeamService {
    public void add(){
        System.out.println("TeamService---- add-----");
    }
}
```

3.6 配置视图解析器

在springmvc.xml配置文件中添加视图解析器的配置

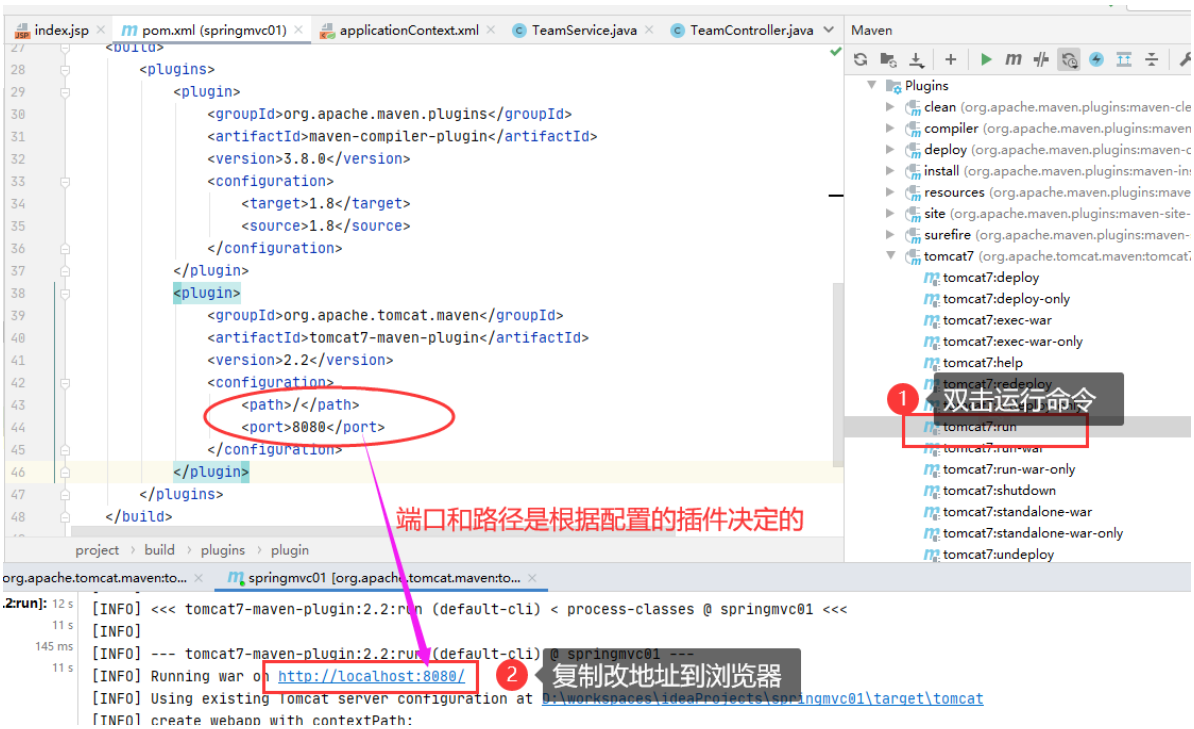
```
<!--视图解析器-->
    <bean id="internalResourceViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>

 <mvc:annotation-driven/>
<!--annotation-driven是一种简写形式，也可以手动配置替代这种简写形式，简写形式可以让初学者快速
应用默认配置方案。
    该注解会自动注册DefaultAnnotationHandlerMapping与AnnotationMethodHandlerAdapter
两个bean,是springMVC为@Controller分发用户请求所必须的，解决了@Controller注解使用的前提配
置。
同时它还提供了：数据绑定支持，@NumberFormatannotation支持，@DateTimeFormat支持，@Valid支
持，读写XML的支持（JAXB，读写JSON的支持（Jackson）。我们处理响应ajax请求时，就使用到了对json
的支持（配置之后，在加入了jackson的core和mapper包之后，不写配置文件也能自动转换成json）。
-->
```

3.7 编写index.jsp页面

webapp文件夹下面创建文件夹jsp,然后jsp文件夹中添加index.jsp页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>index</title>
</head>
<body>
    <h1>index---------------${teamName}</h1>
</body>
</html>
```

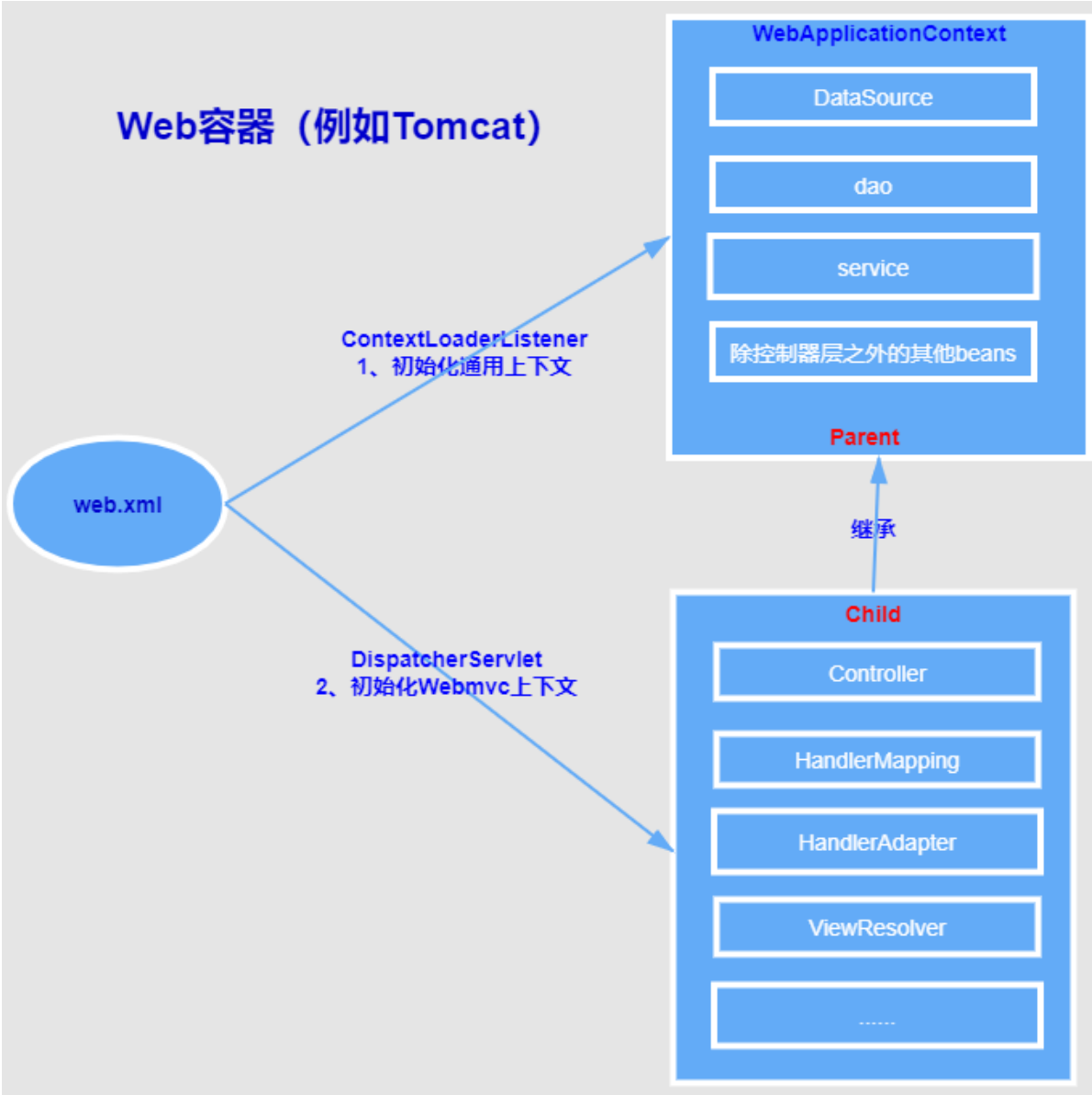
3.8 测试



将刚刚的地址复制到浏览器之后追加控制中add方法的访问路径hello.do,看到如下页面表示运行成功!

3.9 解析

当 Spring 和 SpringMVC 同时出现，我们的项目中将存在两个容器，一个是 Spring 容器，另一个是 SpringMVC 容器，Spring 容器通过 ContextLoaderListener 来加载，SpringMVC 容器则通过 DispatcherServlet 来加载，这两个容器不一样：



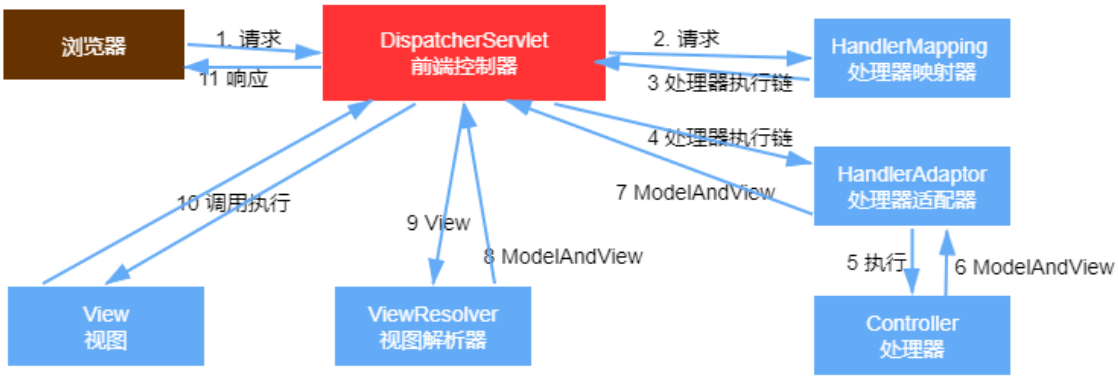
如图所示：

- ContextLoaderListener 初始化的上下文加载的 Bean 是对于整个应用程序共享的，不管是使用什么表现层技术，一般如 dao层、service层 的bean；
- DispatcherServlet 初始化的上下文加载的 bean 是只对 Spring Web MVC 有效的 bean，如 Controller、HandlerMapping、HandlerAdapter 等等，该初始化上下文应该只加载 Web相关组件。

```
1.Spring容器中不能扫描所有Bean嘛？
    不可以。当用户发送的请求达到服务端后，会寻找前端控制器DispatcherServlet去处理，只在SpringMVC容器中找，所以Controller必须在SpringMVC容器中扫描。

2.SpringMVC容器中可以扫描所有Bean嘛？
    可以。可以在SpringMVC容器中扫描所有Bean。但是实际开发中一般不会这么做，原因如下：
    （1）为了方便配置文件的管理
    （2）未来在 Spring+SpringMVC+Mybatis组合中，要写的配置内容很多，一般都会根据功能分开编写
```

4、SpringMVC工作流程

4.1 工作流程分析



（1）用户通过浏览器发送请求到前端控制器DispatcherServlet。

（2）前端控制器直接将请求转给处理器映射器HandleMapping。

（3）处理器映射器HandleMapping会根据请求，找到负责处理该请求的处理器，并将其封装为处理器执行链HandlerExecutionChina后返回给前端控制器DispatcherServlet。

（4）前端控制器DispatcherServlet根据处理器执行链中的处理器，找到能够执行该处理器的处理器适配器HandlerAdaptor。

（5）处理器适配器HandlerAdaptor调用执行处理器Controller。

（6）处理器Controller将处理结果及要跳转的视图封装到一个对象 ModelAndView 中，并将其返回给处理器适配器HandlerAdaptor。

（7）处理器适配器直接将结果返回给前端控制器DispatcherServlet。

（8）前端控制器调用视图解析器，将 ModelAndView 中的视图名称封装为视图对象。

（9）视图解析器ViewResolver将封装了的视图View对象返回给前端控制器DispatcherServlet.

（10）前端控制器DispatcherServlet调用视图对象，让其自己进行渲染，即进行数据填充，形成响应对象。

（11）前端控制器响应浏览器。

4.2 SpringMVC组件

1.DispatcherServlet：前端控制器,也称为中央控制器或者核心控制器。

用户请求的入口控制器，它就相当于 mvc 模式中的c，DispatcherServlet 是整个流程控制的中心，相当于是 SpringMVC 的大脑，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。SpringMVC框架提供的该核心控制器需要我们在web.xml文件中配置。

2.HandlerMapping：处理器映射器

HandlerMapping也是控制器，派发请求的控制器。我们不需要自己控制该类，但是他是springmvc运转历程中的重要的一个控制器。 HandlerMapping负责根据用户请求找到 Handler 即处理器（也就是我们所说的 Controller），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，在实际开发中，我们常用的方式是注解方式。

3.Handler：处理器

Handler 是继 DispatcherServlet 前端控制器的后端控制器，在DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。由于 Handler 涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发 Handler。（这里所说的 Handler 就是指我们的 Controller）

4.HandlAdapter：处理器适配器

通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展处理器适配器，支持更多类型的处理器,调用处理器传递参数等工作。

5.ViewResolver：视图解析器

ViewResolver 负责将处理结果生成 View 视图，ViewResolver 首先根据逻辑视图名解析成物理视图名称，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。 SpringMVC 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

5 @RequestMapping 注解

5.1@RequestMapping出现的位置

过@RequestMapping 注解定义了处理器对于请求的映射规则。该注解可以定义在类上，也可以定义在方法上，但是含义不同。

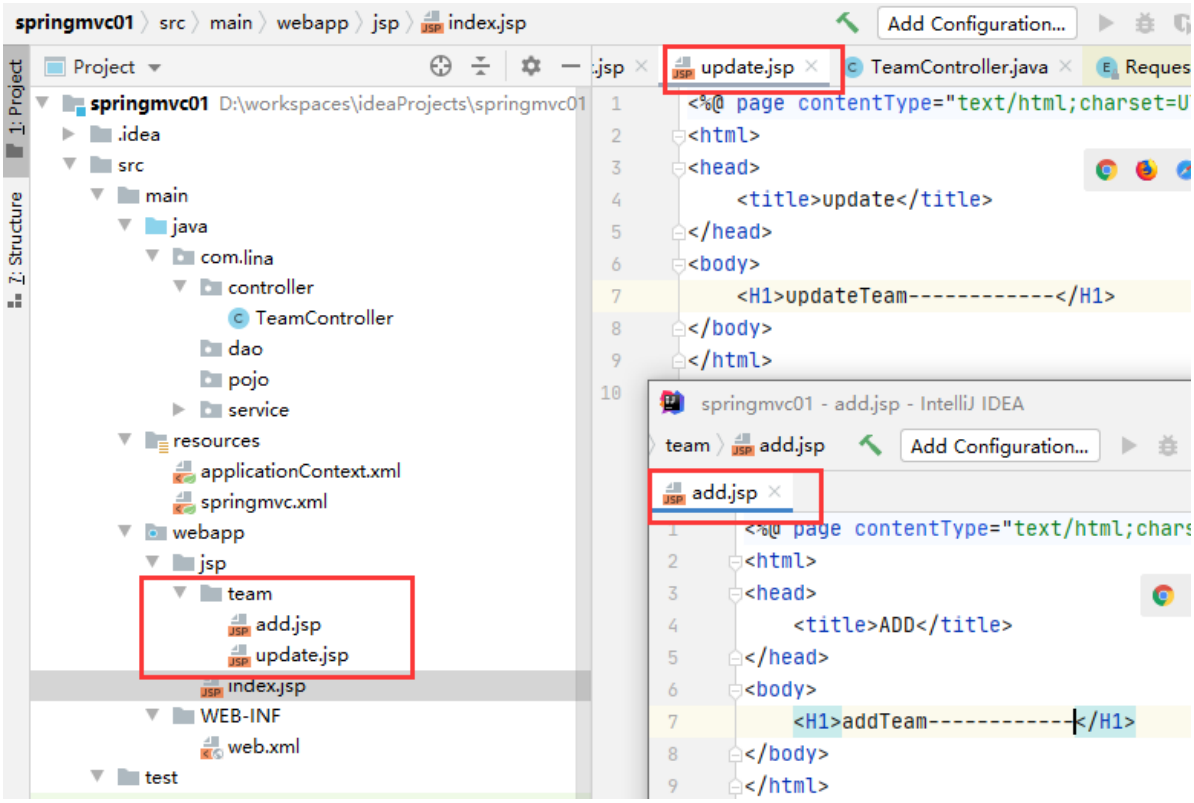
```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Mapping
public @interface RequestMapping {
    String name() default "";

    @AliasFor("path")
    String[] value() default {};

    @AliasFor("value")
    String[] path() default {};

    RequestMethod[] method() default {};

    String[] params() default {};

    String[] headers() default {};

    String[] consumes() default {};

    String[] produces() default {};
}
```

一个@Controller 所注解的类中，可以定义多个处理器方法。当然，不同的处理器方法所匹配的 URI 是不同的。这些不同的 URI 被指定在注解于方法之上的@RequestMapping 的value 属性中。但若这些请求具有相同的 URI 部分，则这些相同的 URI，可以被抽取到注解在类之上的RequestMapping 的 value 属性中。此时的这个 URI 表示模块的名称。URI 的请求是相对于 Web 的根目录。在类的级别上的注解会将一个特定请求或者请求模式映射到一个控制器之上。之后你还可以另外添加方法级别的注解来进一步指定到处理方法的映射关系。

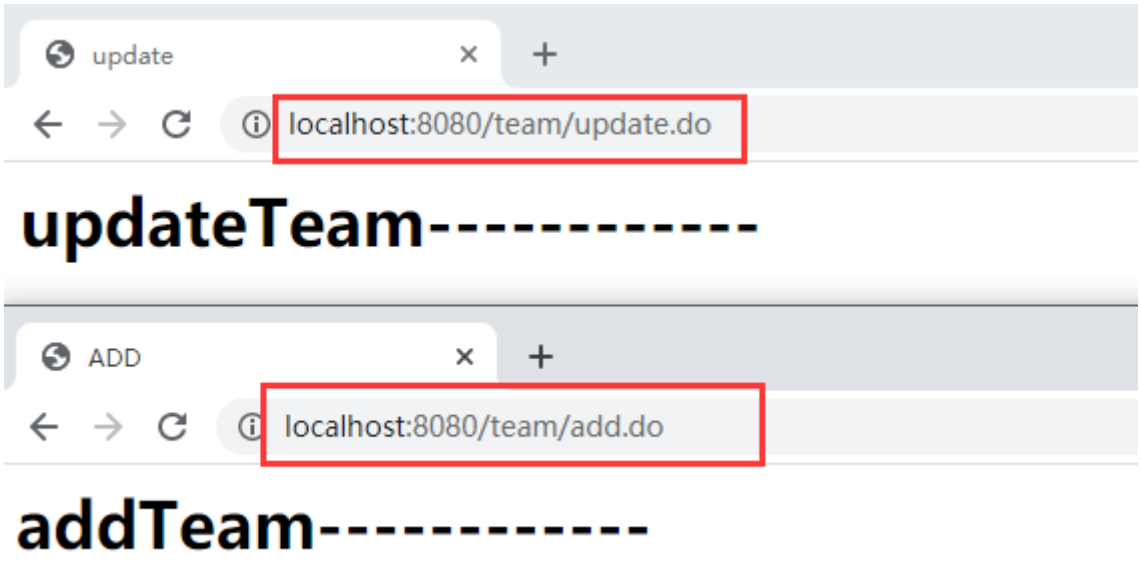
示例：修改TeamController.java 如下

```
package com.lina.controller;
```

```java
import com.lina.service.TeamService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
@Controller
@RequestMapping("team")
public class TeamController {

    @Autowired
    private TeamService teamService;

    @RequestMapping(value = "add.do")
    public ModelAndView addTeam(){
        System.out.println("TeamController----addTeam---");
        ModelAndView mv=new ModelAndView();
        mv.setViewName("team/add");//    映射成为物理资源路径：/jsp/team/add.jsp
        return mv;
    }
    @RequestMapping(value = "update.do")
    public ModelAndView updateTeam(){
        System.out.println("TeamController----updateTeam---");
        ModelAndView mv=new ModelAndView();
        mv.setViewName("team/update");//映射成为物理资源路径：/jsp/team/update.jsp
        return mv;
    }

    @RequestMapping("hello.do")
    public ModelAndView hello(){
        System.out.println("TeamController----add---");
        teamService.add();
        ModelAndView mv=new ModelAndView();
        mv.addObject("teamName","湖人");//相当于
request.setAttrubuite("teanName","湖人");
        mv.setViewName("index");//未来经过springmvc的视图解析器处理，转换成物理资源路径，
相当于request.getRequestDispatcher("index.jsp").forward();
        //经过InternalResourceViewResolver对象的处理之后加上前后缀就变为了
/jsp/index.jsp
        return mv;
    }
}
```



浏览器中访问：



5.2 指定请求提交方式

@RequestMapping的method属性，用来对被注解方法所处理请求的提交方式进行限制，即只有满足method 属性指定的提交方式的请求，才会执行该被注解方法。

Method 属性的取值为 RequestMethod 枚举常量。常用的为 RequestMethod.GET 与

RequestMethod.POST，分别表示提交方式的匹配规则为 GET 与 POST 提交。

```java
@Controller
@RequestMapping("team")
public class TeamController {

    @Autowired
    private TeamService teamService;

    @RequestMapping(value = "add.do",method = RequestMethod.GET)
    public ModelAndView addTeam(){
        System.out.println("TeamController----addTeam---");
        ModelAndView mv=new ModelAndView();
        mv.setViewName("team/add");//     映射成为物理资源路径：/jsp/team/add.jsp
        return mv;
    }
    @RequestMapping(value = "update.do",method = RequestMethod.POST)
    public ModelAndView updateTeam(){
        System.out.println("TeamController----updateTeam---");
        ModelAndView mv=new ModelAndView();
        mv.setViewName("team/update");//映射成为物理资源路径：/jsp/team/update.jsp
        return mv;
    }

    @RequestMapping("hello.do")
    public ModelAndView hello(){
        System.out.println("TeamController----add---");
        teamService.add();
        ModelAndView mv=new ModelAndView();
        mv.addObject( attributeName: "teamName", attributeValue: "湖人");//相当于request.setAttrubuite
        mv.setViewName("index");//未来经过springmvc的视图解析器处理，转换成物理资源路径,相当于request.
        //经过InternalResourceViewResolver对象的处理之后加上前后缀就变为了    /jsp/index.jsp
        return mv;
    }
```

只能处理get方式的请求

只能处理post方式提交的请求

没有指定method属性，都可以处理

update指定了post提交方式，此处用get方式请求就会报出405错误

以下表格列出了常用的提交方式：

请求方式	提交方式
地址栏请求	get请求
超链接请求	get请求
表单请求	默认get，可以指定post
AJAX请求	默认get，可以指定post

5.3 补充url-pattern解析

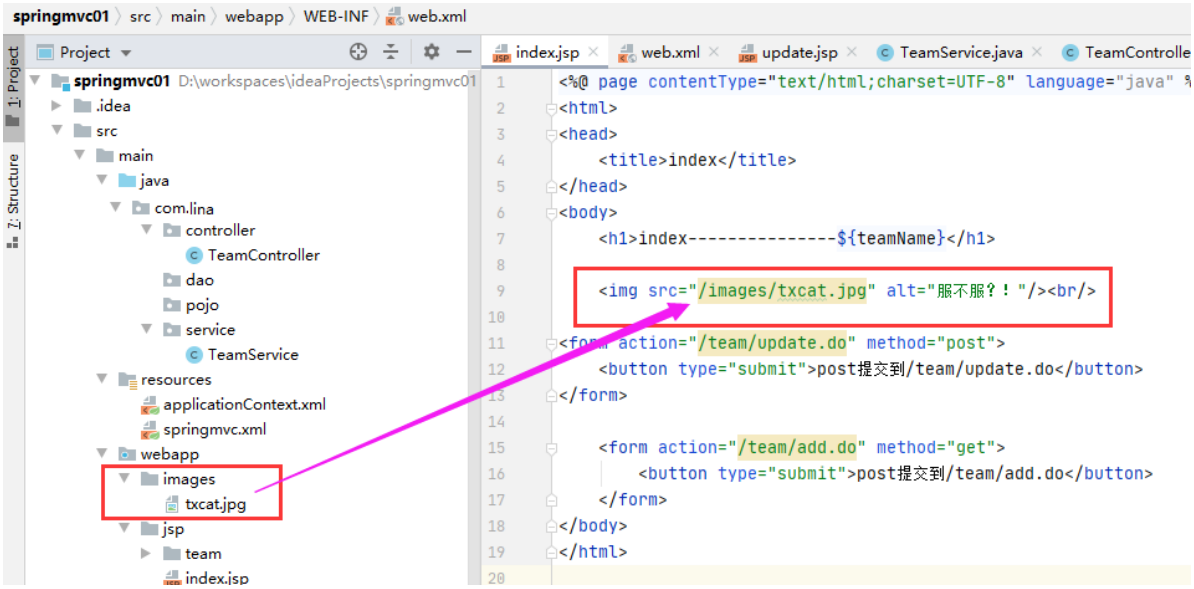
5.3.1 url-pattern解析

在web.xml配置SpringMVC的前端控制器时有这个节点。这个节点中的值一般有两种写法：

> 1、*.do
>
> 在没有特殊要求的情况下，SpringMVC 的前端控制器 DispatcherServlet 的常使用后辍匹配方式，可以写为*.do 或者 *.action, *.mvc 等。
>
> 2、/
>
> 可以写为/，但是 DispatcherServlet 会将向静态内容--例如.css、.js、图片等资源的获取请求时，也会当作是一个普通的 Controller 请求。前端控制器会调用处理器映射器为其查找相应的处理器。肯定找不到啊，所以所有的静态资源获取请求也均会报 404 错误。

案例：在index.jsp页面添加一张图片，如果节点中的值为*.do,图片可以正常访问，但是如果为/就不能访问。

1、项目中添加图片，同时修改index.jsp页面



2、修改web.xml

```
<!--springMVC的前端/核心/中央控制器-->
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name> <!-- dispatcherServlet-servlet.xml-->
        <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

从*.do修改为/

3、此时访问图片无法显示

5.3.2 静态资源访问

如果的值配置为/后，静态资源可以通过以下两种方法解决。

5.3.2.1使用< mvc:default-servlet-handler/ >

在springmvc的配置文件中添加如下内容：

```
 <mvc:default-servlet-handler/>
 <!--声 明 了 <mvc:default-servlet-handler /> 后 ， springmvc框架会在容器中创建
DefaultServletHttpRequestHandler处理器对象。该对象会对所有进入 DispatcherServlet的URL
进行检查。如果发现是静态资源的请求，就将该请求转由Web应用服务器默认的Servlet 处理。
    一般的服务器都有默认的 Servlet。例如咱们使用的Tomcat服务器中，有一个专门用于处理静态资源
访问的 Servlet 名叫 DefaultServlet。其<servlet-name/>为default。可以处理各种静态资源访问
请求。该Servlet注册在 Tomcat 服务器的 web.xml 中。在 Tomcat安装目录/conf/web.xml。-->
```

```
107
108     <servlet>
109         <servlet-name>default</servlet-name>
110         <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
111         <init-param>
112             <param-name>debug</param-name>
113             <param-value>0</param-value>
114         </init-param>
115         <init-param>
116             <param-name>listings</param-name>
117             <param-value>false</param-value>
118         </init-param>
119         <load-on-startup>1</load-on-startup>
120     </servlet>
121
```

5.3.2.2 使用 < mvc:resources/ >

在springmvc的配置文件中添加如下内容：

```
 <mvc:resources location="/images/" mapping="/images/**" />
<!--
    location: 表示静态资源所在目录。当然，目录不要使用/WEB-INF/及其子目录。
    mapping: 表示对该资源的请求。注意，后面是两个星号**。-->
```

在 Spring3.0 版本后，Spring 定义了专门用于处理静态资源访问请求的处理器 ResourceHttpRequestHandler。并且添加了< mvc:resources/ >标签，专门用于解决静态资源无法访问问题。

6、处理器方法的参数

处理器方法可以包含以下四类参数，这些参数会在系统调用时由系统自动赋值.所以我们可以在方法内直接使用。以下是这四类参数:

- HttpServletRequest
- HttpServletResponse
- HttpSession
- 请求中所携带的请求参数

准备工作：创建新的控制器ParamController.java和前端页面hello.jsp页面

```
package com.lina.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("param")
public class ParamController {

    @RequestMapping("hello")
    public  ModelAndView hello(){
        return new ModelAndView("hello);
    }
}
```

6.1 直接使用方法的参数逐个接收

6.2 使用对象接收多个参数

6.3 请求参数和方法名称的参数不一致

6.4 使用HttpServletRequest 对象获取参数

6.5 直接使用URL地址传参

6.6 获取日期类型的参数

6.7 获取数组类型的参数

6.8 获取集合类型的参数

SpringMVC不支持直接从参数中获取对象集合类型，需要将对象集合封装到实体类中。

案例源码：

修改实体类Team.java

```
package com.lina.pojo;
import org.springframework.format.annotation.DateTimeFormat;
import java.util.Date;
public class Team {
    private Integer teamId;
    private String teamName;
    private String location;
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    private Date createTime;

    @Override
    public String toString() {
        return "Team{" +
                "teamId=" + teamId +
                ", teamName='" + teamName + '\'' +
                ", location='" + location + '\'' +
                ", createTime=" + createTime +
                '}';
    }
//省略set get方法
}
```

创建实体类QueryVO.java

```
package com.lina.vo;
import com.lina.pojo.Team;
import java.util.List;

public class QueryVO {
    private List<Team> teamList;

    public List<Team> getTeamList() {
        return teamList;
    }

    public void setTeamList(List<Team> teamList) {
        this.teamList = teamList;
    }
}
```

前端页面hello.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>hello</title>
</head>
<body>

    <h3>8、获取集合类型的参数</h3>
    <form action="/param/test08" method="post">
        球队名称1：<input type="text" name="teamName"/><br/>
        球队名称2：<input type="text" name="teamName"/><br/>
        球队名称3：<input type="text" name="teamName"/><br/>
        <button type="submit">提交</button>
    </form>
    <form action="/param/test09" method="post">
        球队id1：<input type="text" name="teamList[0].teamId"/><br/>
        球队id2：<input type="text" name="teamList[1].teamId"/><br/>
        球队id3：<input type="text" name="teamList[2].teamId"/><br/>
        球队名称1：<input type="text" name="teamList[0].teamName"/><br/>
        球队名称2：<input type="text" name="teamList[1].teamName"/><br/>
        球队名称3：<input type="text" name="teamList[2].teamName"/><br/>
        <button type="submit">提交</button>
    </form>
    <h3>7、获取数组类型的参数</h3>
    <form action="/param/test07" method="post">
        球队名称1：<input type="text" name="teamName"/><br/>
        球队名称2：<input type="text" name="teamName"/><br/>
        球队名称3：<input type="text" name="teamName"/><br/>
        <button type="submit">提交</button>
    </form>
    <h3>6、获取日期类型的参数</h3>
    <form action="/param/test06" method="post">
        球队id：<input type="text" name="teamId"/><br/>
        球队名称：<input type="text" name="teamName"/><br/>
        球队位置：<input type="text" name="location"/><br/>
        创建日期：<input type="text" name="createTime"/><br/>
        <button type="submit">提交</button>
```

```
    </form>
    <h3>5、直接使用URL地址传参</h3>

    <h3>4、使用HttpServletRequest 对象获取参数</h3>
    <form action="/param/testO4" method="post">
        球队id：<input type="text" name="teamId"/><br/>
        球队名称：<input type="text" name="teamName"/><br/>
        球队位置：<input type="text" name="location"/><br/>
        <button type="submit">提交</button>
    </form>
    <h3>3、请求参数和方法名称的参数不一致</h3>
    <form action="/param/testO3" method="get">
        球队ID：<input name="id" /><br/>
        球队name：<input name="name" /><br/>
        球队location：<input name="location" /><br/>
        <button type="submit">提交</button>
    </form>
    <h3>2、使用对象接收多个参数</h3>
    <form action="/param/testO2" method="post">
        球队id：<input type="text" name="teamId"/><br/>
        球队名称：<input type="text" name="teamName"/><br/>
        球队位置：<input type="text" name="location"/><br/>
        <button type="submit">提交</button>
    </form>
    <h3>1、直接使用方法的参数逐个接收</h3>
    <form action="/param/testO1" method="post">
        球队id：<input type="text" name="teamId"/><br/>
        球队名称：<input type="text" name="teamName"/><br/>
        球队位置：<input type="text" name="teamLocation"/><br/>
        <button type="submit">提交</button>
    </form>
</body>
</html>
```

控制器ParamController.java

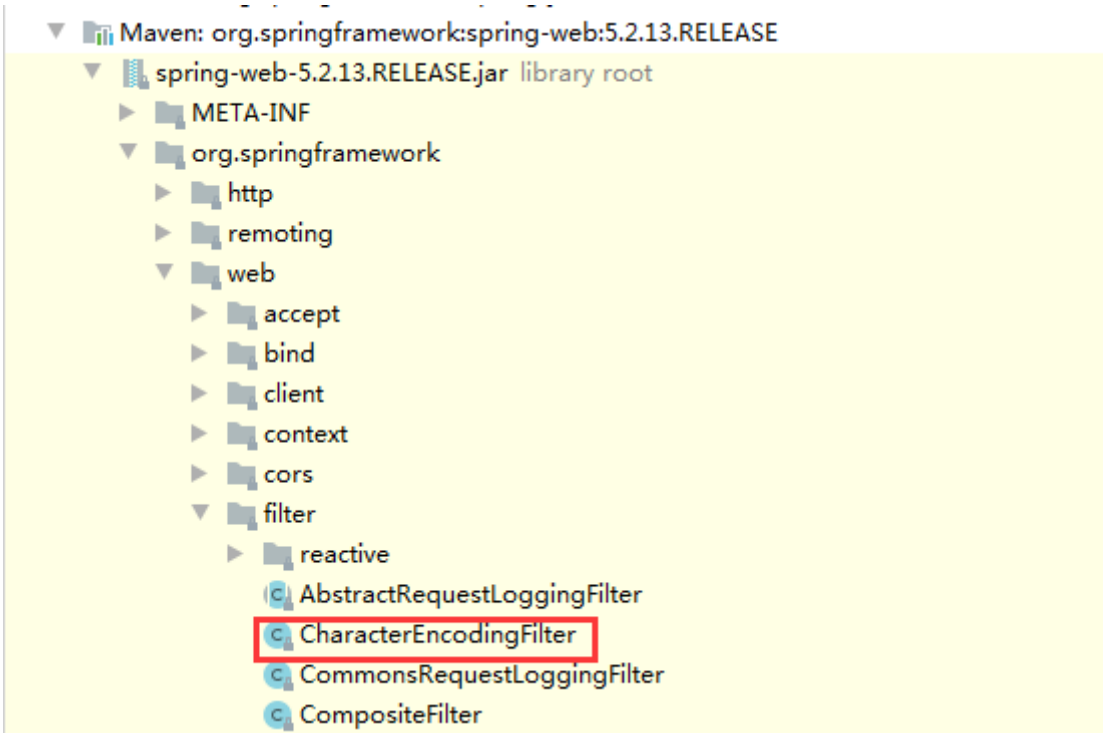
```
package com.lina.controller;
import com.lina.pojo.Team;
import com.lina.vo.QueryVO;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.http.HttpServletRequest;
import java.util.List;

@Controller
@RequestMapping("param")
public class ParamController {

    //8、获取集合类型的参数：简单类型的可以通过@RequestParam注解实现；对象集合不支持直接获取，必须封装在类中，作为一个属性操作
    @RequestMapping("testO8")
    public ModelAndView testO8(@RequestParam("teamName") List<String> nameList){
        System.out.println("testO8-----------------");
```

```
        for (String s : nameList) {
            System.out.println(s);
        }
        return new ModelAndView("ok");
    }
    @RequestMapping("test09")
    public ModelAndView test09(QueryVO vo){
        System.out.println("test09-----------------");
        for (Team team : vo.getTeamList()) {
            System.out.println(team);
        }
        return new ModelAndView("ok");
    }

        //7、获取数组类型的参数
    @RequestMapping("test07")
    public ModelAndView test07(String[] teamName,HttpServletRequest request){
        System.out.println("test07-----------------");
        //方式1:
        for (String s : teamName) {
            System.out.println(s);
        }
        System.out.println("---------------");
        //方式2:
        String[] teamNames = request.getParameterValues("teamName");
        for (String name : teamNames) {
            System.out.println(name);
        }
        return new ModelAndView("ok");
    }
    //6、获取日期类型的参数
    @RequestMapping("test06")
    public ModelAndView test06(Team team){
        System.out.println("test06-----------------");
        System.out.println(team);
        return new ModelAndView("ok");
    }

    //5、直接使用URL地址传参: 借助@PathVariable 注解
    // 例如http://localhost:8080/param/test05/1001/lacker/las
    @RequestMapping("test05/{id}/{name}/{loc}")
    public ModelAndView test05(@PathVariable("id") Integer teamId,
                               @PathVariable("name") String teamName,
                               @PathVariable("loc") String teamLocation){
        System.out.println("test05-----------------");
        System.out.println(teamId);
        System.out.println(teamName);
        System.out.println(teamLocation);
        return new ModelAndView("ok");
    }
    //4、使用HttpServletRequest 对象获取参数:跟原来的javaWeb项目中使用的方式是一样的
    @RequestMapping("test04")
    public ModelAndView test04(HttpServletRequest request){
        System.out.println("test04-----------------");
        String teamId = request.getParameter("teamId");
        String teamName = request.getParameter("teamName");
        String location = request.getParameter("location");
        if(teamId!=null)
```

```
            System.out.println(Integer.valueOf(teamId));
        System.out.println(teamName);
        System.out.println(location);
            return new ModelAndView("ok");
    }

    //3、请求参数和方法名称的参数不一致:使用@RequestParam进行矫正，
    // value属性表示请求中的参数名称
    // required属性表示参数是否是必须的：true:必须赋值，否则报出400错；false：可以不赋值，
结果就是null
    //defaultValue=自定义默认值  如果没有从用户请求中获取到值，使用默认值
    @RequestMapping("testO3")
    public ModelAndView testO3(@RequestParam(value = "id",required =
false,defaultValue = "1") Integer teamId,
                               @RequestParam(value = "name") String teamName,
                               @RequestParam(value = "location") String
teamLocation){
        System.out.println("testO3--------------");
        System.out.println(teamId);
        System.out.println(teamName);
        System.out.println(teamLocation);
        return new ModelAndView("OK");
    }

    //2、使用对象接收多个参数：要求用户请求中携带的参数名称必须是实体类中的属性保持一致，否则就
获取不到
    @RequestMapping("testO2")
    public ModelAndView testO2(Team team){
        System.out.println("testO2-----------------");
        System.out.println(team);
        return new ModelAndView("ok");
    }

    /**
     * 1、直接使用方法的参数逐个接收:方法的参数名称必须与用户请求中携带的参数名称保持一致，否则
就获取不到
     * 好处：不需要类型转换
     */
    @RequestMapping("testO1")
    public ModelAndView testO1(Integer teamId,String teamName,String
teamLocation){
        System.out.println("testO1-----------------");
        System.out.println(teamId);
        System.out.println(teamName);
        System.out.println(teamLocation);
        return new ModelAndView("ok");
    }

    @RequestMapping("hello")
    public  ModelAndView hello(){
        return new ModelAndView("hello");
    }
}
```

7、请求参数中文乱码

对于前面所接收的请求参数，若含有中文，则会出现中文乱码问题。Spring 对于请求参数中的中文乱码问题，给出了专门的字符集过滤器CharacterEncodingFilter 类。如图所示。



7.1 乱码解决方案

在 web.xml 中注册字符集过滤器，推荐将该过滤器注册在其它过滤器之前。因为过滤器的执行是按照其注册顺序进行的。

在web.xml配置文件直接注册字符集

```
<!--注册字符集过滤器：post请求中文乱码问题的解决方案-->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <!--指定字符集-->
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <!--强制request使用字符集encoding-->
        <init-param>
            <param-name>forceRequestEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
        <!--强制response使用字符集encoding-->
        <init-param>
            <param-name>forceResponseEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

7.2 解决方案原理

```
public class CharacterEncodingFilter extends OncePerRequestFilter {

    @Nullable
    private String encoding;   设置的编码格式

    private boolean forceRequestEncoding = false;  是否强制设置请求编码格式为设置的编码格式

    private boolean forceResponseEncoding = false; 是否强制设置返回编码格式为设置的编码格式

    @Override
    protected void doFilterInternal(
            HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String encoding = getEncoding();
        if (encoding != null) {
            if (isForceRequestEncoding() || request.getCharacterEncoding() == null) {
                request.setCharacterEncoding(encoding);
            }
            if (isForceResponseEncoding()) {
                response.setCharacterEncoding(encoding);
            }
        }
        filterChain.doFilter(request, response);
    }
```

8、处理器方法的返回值

使用@Controller 注解的处理器的处理器方法，其返回值常用的有四种类型：

1. ModelAndView
2. String
3. 返回自定义类型对象
4. 无返回值 void

咱们根据实际的业务选择不同的返回值。

案例准备工作：

创建控制器ResultController.java

```
package com.lina.controller;
import com.lina.pojo.Team;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@RequestMapping("result")
public class ResultController {

}
```

在jsp文件夹中添加页面result.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>result</title>
</head>
<body>
<h1>result----------------</h1>

</body>
</html>
```

8.1 返回ModelAndView

如果是前后端不分的开发，大部分情况下，我们返回 ModelAndView，即数据模型+视图：

控制器ResultController.java中添加方法：

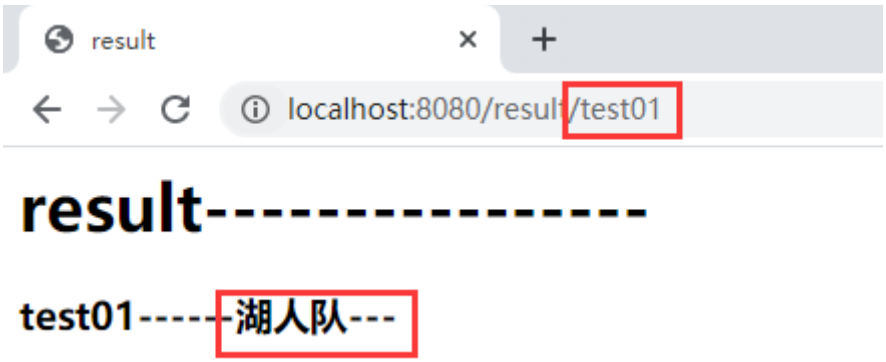
```
//1、返回值是ModelAndView：  这种方式既有数据的携带还有资源的跳转，可以选择该种方式
    @RequestMapping("test01")
    public ModelAndView test01(){
        ModelAndView mv=new ModelAndView();//模型与视图
        //携带数据
        mv.addObject("teamName","湖人队");//相当于request。
setAttribute("teamName","湖人队“);
        mv.setViewName("result");// 经过视图解析器InternalResourceViewResolver的处
理，将逻辑视图名称加上前后缀变为物理资源路径   /jsp/result.jsp
        return mv;
    }
```

Model 中，放我们的数据，然后在 ModelAndView 中指定视图名称。

当处理器方法处理完后，需要跳转到其它资源的同时传递数据，选择返回 ModelAndView 比较好，但是如果只是需要传递数据或者跳转之一，这个时候ModelAndView 就不是最优选择。

result.jsp页面中添加如下内容：

```
<h3>test01------${teamName}---</h3>
```



8.2 返回String

上一种方式中的 ModelAndView 可以拆分为两部分，Model 和 View，在 SpringMVC 中，Model 我们可以直接在参数中指定，然后**返回值是逻辑视图名**，视图解析器解析可以将逻辑视图名称转换为物理视图地址。

```
<!--视图解析器-->
<bean id="internalResourceViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/jsp/"></property>
    <property name="suffix" value=".jsp"></property>
</bean>
```

视图解析器通过**内部资源视图解析器InternalResourceViewResolver**将字符串与解析器中的prefix和suffix结合形成要跳转的额URI。

控制器ResultController.java中添加方法：

```
    //2、返回字符串
    @RequestMapping("testO2")
    public String testO2(HttpServletRequest request){
        Team team=new Team();
        team.setLocation("迈阿密");
        team.setTeamId(1002);
        team.setTeamName("热火");
        //携带数据
        request.setAttribute("team",team);
        request.getSession().setAttribute("team",team);
        //资源的跳转
        return "result";// 经过视图解析器InternalResourceViewResolver的处理，将逻辑视图名称加上前后缀变为物理资源路径  /jsp/result.jsp
    }
```

result.jsp页面中添加如下内容：

```
<h3>testO2---request作用域获取:---${requestScope.team.teamName}--
-${requestScope.team.teamId}---${requestScope.team.location}</h3>
<h3>testO2---session作用域获取:---${sessionScope.team.teamName}--
-${sessionScope.team.teamId}---${sessionScope.team.location}</h3>
```

8.3 返回对象类型

当处理器方法返回Object对象类型的时候，可以是Integer、String、Map、List，也可以是自定义的对象类型。但是无论是什么类型，都不是作为逻辑视图出现，而是直接作为数据返回然后展示的。一般前端发起Ajax请求的时候都会使用直接返回对象的形式。

返回对象的时候，需要使用@ResponseBody 注解，将转换后的 JSON 数据放入到响应体中。

pom.xml文件中添加两个依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.0</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.0</version>
</dependency>
```

8.3.1 返回基础类型

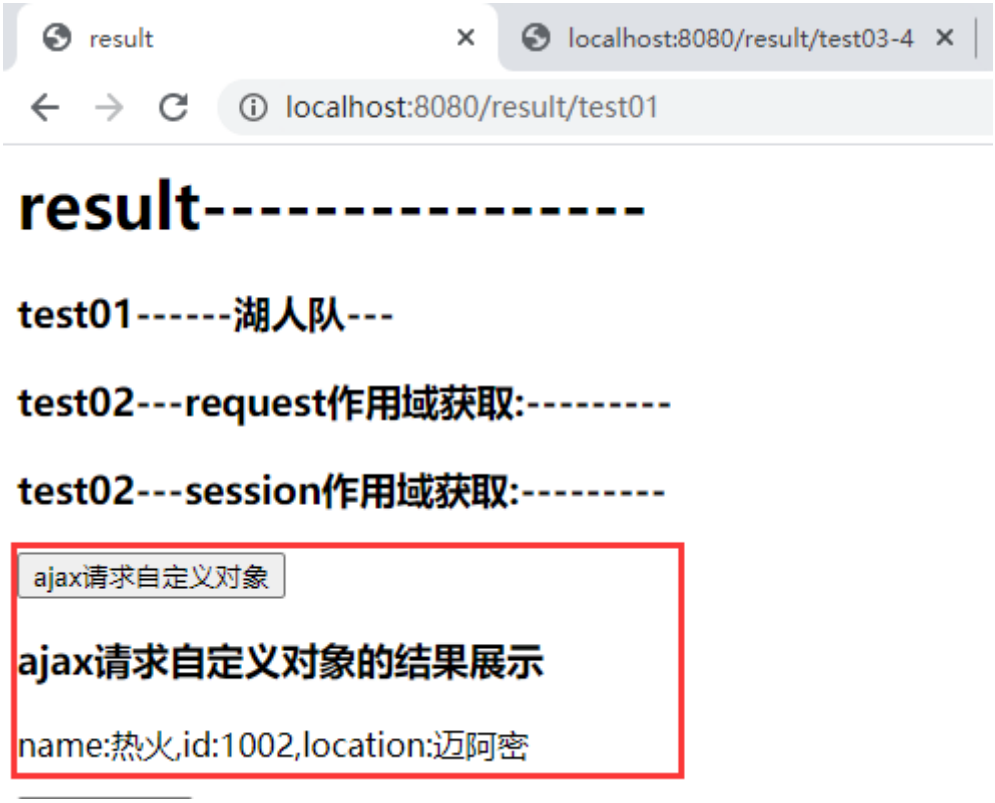
```
 //3、返回对象类型：Integer Double String  自定义类型   List  Map  返回的不是逻辑视图的名称，而直接就是数据返回 ，一般是ajax请求搭配使用 ，将json格式的数据直接返回给响应体
    // 一定要与@ResponseBody
    @ResponseBody
    @RequestMapping("testO3-1")
    public Integer testO31(){
        return 666;
    }

    @ResponseBody
    @RequestMapping("testO3-2")
    public String testO32(){
        return "test";
    }
```

8.3.2 返回自定义对象类型

```
@ResponseBody
    @RequestMapping("testO3-3")
    public Team testO33(){
        Team team=new Team();
        team.setLocation("迈阿密");
        team.setTeamId(1002);
        team.setTeamName("热火");
        return team;
    }
```

```
<div>
    <button type="button" id="btn1">ajax请求自定义对象</button>
    <h3>ajax请求自定义对象的结果展示</h3>
    <p id="res"></p>
</div>
<script>
    $(function(){
        $("#btn1").click(function () {
            $.ajax({
                type: "POST",
                url: "/result/testO3-3",
                data: "",
```

```
                success: function(msg){
                    alert( "Data Saved: " + msg );
                     var name=msg.teamName;
                     var id=msg.teamId;
                     var loc=msg.location;
                    $("#res").html("name:"+name+",id:"+id+",location:"+loc);
                }
            });
        });
</script>
```

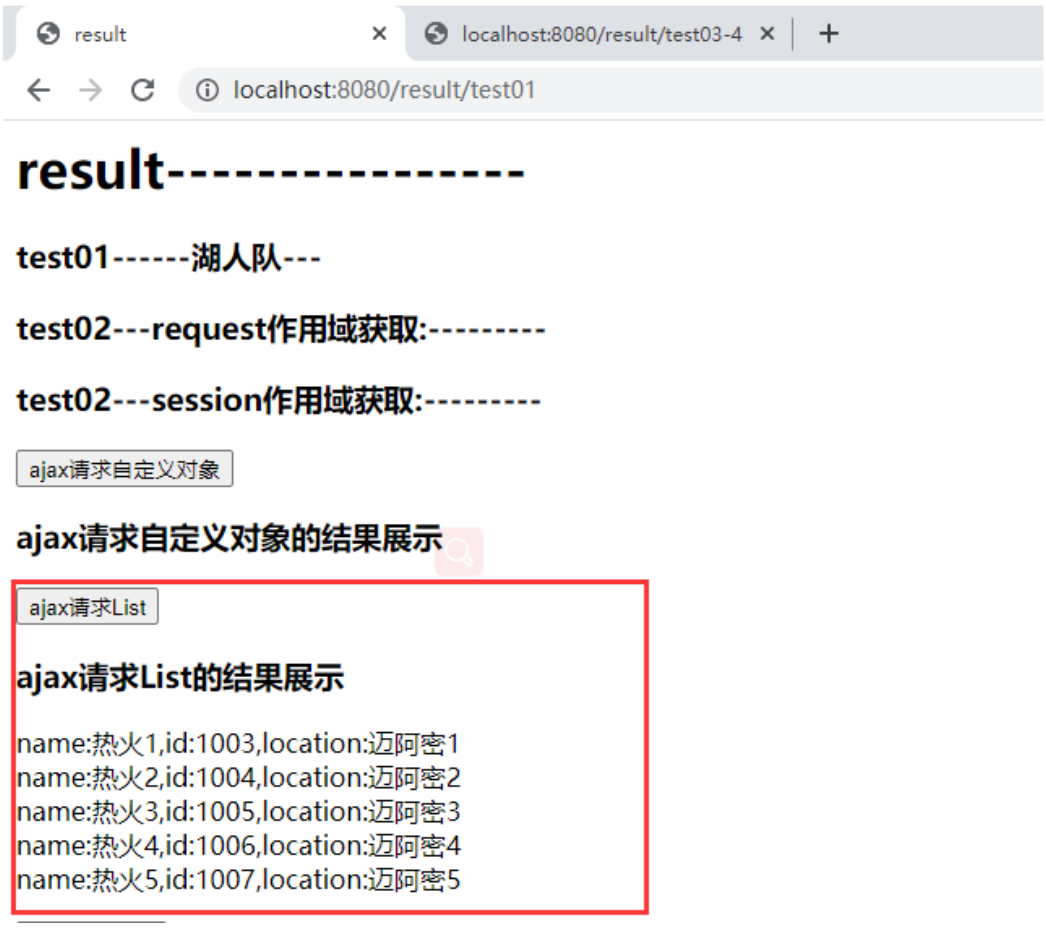


8.3.3 返回集合List

```
@ResponseBody
    @RequestMapping("test03-4")
    public List<Team> test034(){
        List<Team> list=new ArrayList<>(5);
        for(int i=1;i<=5;i++) {
            Team team = new Team();
            team.setLocation("迈阿密"+i);
            team.setTeamId(1002+i);
            team.setTeamName("热火"+i);
            list.add(team);
        }
        return list;
    }
```

```
<div>
        <button type="button" id="btn2">ajax请求对象--List</button>
        <h3>ajax请求自定义对象--list的结果展示</h3>
        <p id="res2"></p>
    </div>
<script>
```

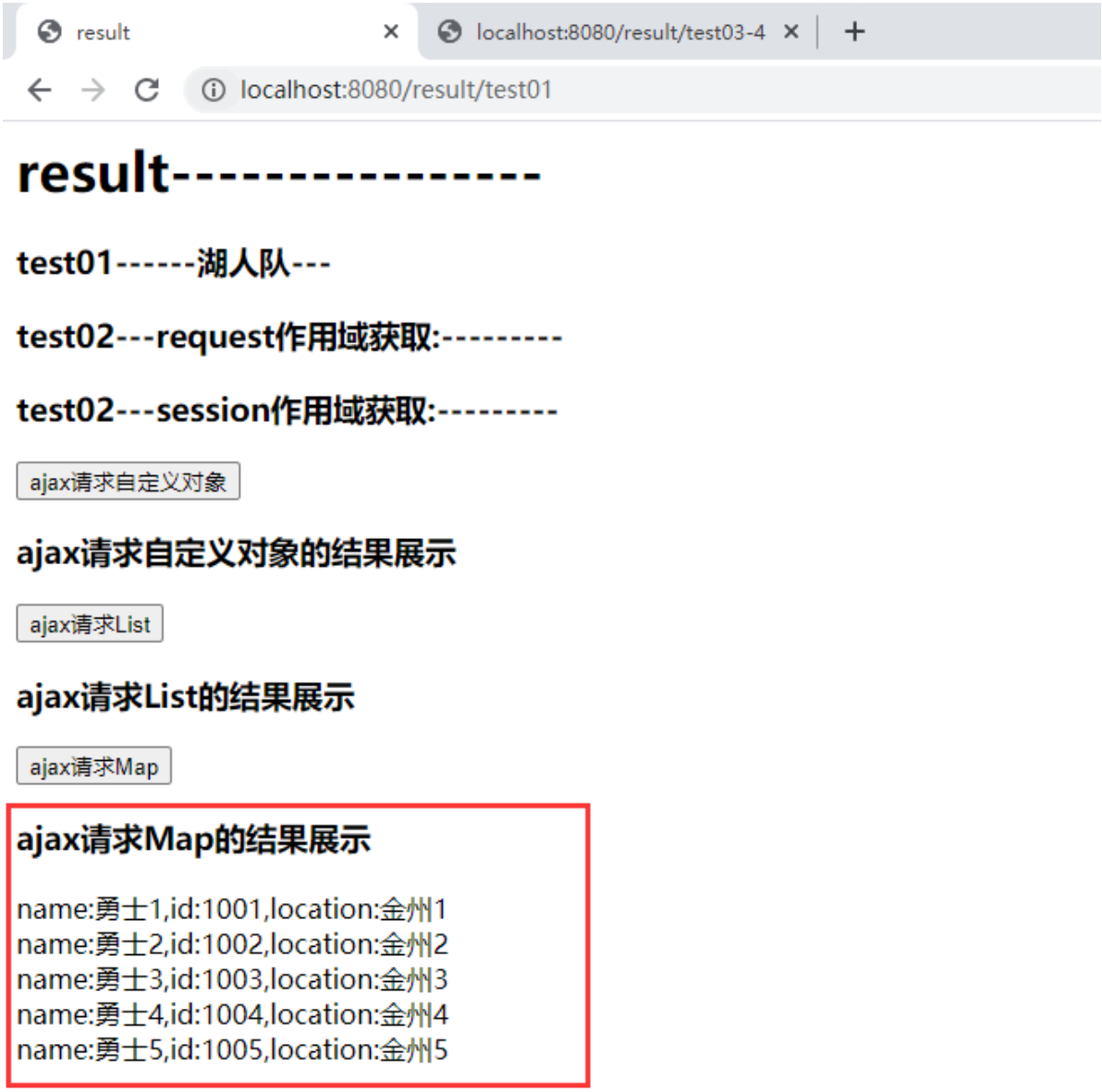
```
    $(function(){
        $("#btn2").click(
            function(){
                $.ajax({
                    type:"POST",
                    url:"/result/test03-4",
                    data:"",
                    success:function (list) {
                        //alert(msg);
                        var str = "";
                        for (var i = 0; i < list.length; i++) {
                            var msg=list[i];
                            var name = msg.teamName;
                            var id = msg.teamId;
                            var location = msg.location;
                            str+="name:" + name + ",id:" + id + ",location:" +
location+"<br/>";
                        }
                        $("#res2").html(str);
                    }
                });
            }
        );
    });
</script>
```



8.3.4 返回集合Map

```
@ResponseBody
    @RequestMapping("test03-5")
    public Map<String,Team> test035(){
        Map<String,Team> map=new HashMap();
        for(int i=1;i<=5;i++) {
            Team team = new Team();
            team.setLocation("金州"+i);
            team.setTeamId(1000+i);
            team.setTeamName("勇士"+i);
            //日期类型，在返回的时候是个数字，如果想要按日期格式展示需要在实体类对应属性添加注
解@JsonFormat(pattern = "yyyy-MM-dd")
            team.setCreateTime(new Date());
           map.put(team.getTeamId()+"",team);
        }
        return map;
    }
```

```
<div>
       <button type="button" id="btn3">ajax请求对象-Map</button>
       <h3>ajax请求对象--Map的结果展示</h3>
       <p id="res3"></p>
   </div>
<script>
    $(function(){
        $("#btn3").click(function () {
            $.ajax({
                type: "POST",
                url: "/result/test03-5",
                data: "",
                success: function(map){
                    alert( "Data Saved: " + map );
                    var str="";
                    $.each(map,function (i,obj) {

 str+="name:"+obj.teamName+",id:"+obj.teamId+",location:"+obj.location+"<br/>";
                    });
                    $("#res3").html(str);
                }
            });
        });
</script>
```



8.4 无返回值 void-了解

方法的返回值为 void，并不一定真的没有返回值，我们可以通过其他方式给前端返回。实际上，这种方式也可以理解为 Servlet 中的的处理方案。

```
//通过 HttpServletRequest 做服务端跳转
@RequestMapping("testO4-1")
    public void testO41(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("直接使用HttpServletRequest进行服务器端的转发");
        request.getRequestDispatcher("/jsp/ok.jsp").forward(request,response);
    }
```

```
//通过 HttpServletResponse 做重定向
@RequestMapping("testO4-2")
    public void testO42(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("直接使用HttpServletResponse重定向跳转");
        response.sendRedirect("/jsp/ok.jsp");
    }
```

```
//通过 HttpServletResponse 给出响应
@RequestMapping("test04-3")
    public void test043(HttpServletResponse response) throws ServletException,
IOException {
        response.setCharacterEncoding("UTF-8");
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter writer = response.getWriter();
        writer.write("返回void类型测试---直接返回字符串");
        writer.flush();
        writer.close();
    }
```

```
//也可以自己手动指定响应头去实现重定向:
@RequestMapping("test04-4")
    public void test044(HttpServletResponse response) throws ServletException,
IOException {
        response.setStatus(302);//设置响应码，302表示重定向
       response.setHeader("Location","/jsp/ok.jsp");
    }
```

9、页面导航/资源跳转的方式

页面导航分为两种：1、转发 2、重定向

springMVC有以下两种方式实现页面的转发或重定向：

1、返回字符串

2、使用ModelAndView

在SpringMVC中两种导航进行页面导航的时候使用不同的前缀指定转发还是重定向

前缀： 转发： forward:url 默认

重定向： redirect:url

准备工作：创建一个新的控制器NavigationController.java：

```
package com.lina.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import javax.servlet.http.HttpServletRequest;

/**
 * ClassName: NavigationController
 * SpringMVC导航的方式
 * @author wanglina
 * @version 1.0
 */
@Controller
@RequestMapping("navigation")
public class NavigationController {
}
```

9.1 转发到一个jsp页面

9.1.1 字符串方式转发

```
@RequestMapping("test01-1")
    public String test011(HttpServletRequest request){
        request.setAttribute("teamName","湖人");
        //return "OK";//默认方式:由视图解析器处理之后将逻辑视图转为物理资源路径
        return "forward:/jsp/OK.jsp";//当添加了forward前缀之后，视图解析器中的前后缀就失效了，必须自己编写绝对路径
    }
```

9.1.2 ModelAndView转发

```
@RequestMapping("test01-2")
    public ModelAndView test012(){
        ModelAndView mv=new ModelAndView();
        mv.addObject("teamName","热火");
        //mv.setViewName("OK");//默认方式:由视图解析器处理之后将逻辑视图转为物理资源路径
        mv.setViewName( "forward:/jsp/OK.jsp");//当添加了forward前缀之后，视图解析器中的前后缀就失效了，必须自己编写绝对路径
        return mv;
    }
```

9.2 重定向到一个jsp页面

9.2.1 字符串方式重定向

```
    @RequestMapping("test02-1")
    public String test021(HttpServletRequest request){
        request.setAttribute("teamName","勇士");//页面上无法获取到存储在request作用域中的值，请求中断了
        return "redirect:/jsp/OK.jsp";//当添加了redirect前缀之后，视图解析器中的前后缀就失效了，必须自己编写绝对路径
    }
```

9.2.2 ModelAndView重定向方式

```
@RequestMapping("test02-2")
    public ModelAndView test022(HttpServletRequest request){
        System.out.println("test02-2--------------");
        ModelAndView mv=new ModelAndView();
        //存储在request作用域中的值以参数的形式追加在URL后面
http://localhost:8080/jsp/OK.jsp?teamName=热火队&teamId=1001
        mv.addObject("teamName","热火队");
        mv.addObject("teamId","1001");
        mv.setViewName("redirect:/jsp/OK.jsp");//当添加了redirect前缀之后，视图解析器中的前后缀就失效了，必须自己编写绝对路径
        return mv;
    }
```

9.3 重定向或者转发到控制器

```
//重定向或者转发到控制器
```

```
    @RequestMapping("testO3-1")
   public ModelAndView testO31(){
        System.out.println("testO3-1----转发到控制");
        ModelAndView mv=new ModelAndView();
        mv.addObject("teamName","小牛");
        mv.setViewName("forward:/navigation/testO2-2");
        return mv;
   }

    @RequestMapping("testO3-2")
    public ModelAndView testO32(){
        System.out.println("testO3-2----重定向到控制");
        ModelAndView mv=new ModelAndView();
        mv.addObject("teamName","凯尔特人");
        mv.addObject("teadmId","1OO3");
        mv.setViewName("redirect:/navigation/testO1-1");//参数值直接追加到URL后面
        return mv;
    }
```

10、异常处理

SpringMVC框架常用@ExceptionHandler 注解处理异常。

10.1@ExceptionHandler 注解

@ExceptionHandler 可以将一个方法指定为异常处理方法。

被注解的方法，其返回值可以是 ModelAndView、String，或 void，方法名随意，方法参数可以是 Exception 及其子类对象、HttpServletRequest、HttpServletResponse 等。系统会自动为这些方法参数赋值。

对于异常处理注解的用法，也可以直接将异常处理方法注解于 Controller 之中.

```
@Target(ElementType.METHOD) 用在方法上面的注解
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ExceptionHandler {

    /**
     * Exceptions handled by the annotated method. If empty, will default to any
     * exceptions listed in the method argument list.
     */ 可选属性 value，为一个 Class<?>数组，用于指定该注解的方法所要处理的异常类，即所要匹配的异常。
    Class<? extends Throwable>[] value() default {};

}
```

10.2 实现步骤

10.2.1 自定义异常类

```
package com.lina.exceptions;
/**
 * ClassName: TeamException
 * 自定义的异常类
 * @author wanglina
 * @version 1.0
 */
public class TeamException extends Exception{
```

```
    public TeamException() {
    }

    public TeamException(String message) {
        super(message);
    }
}
```

```
package com.lina.exceptions;
/**
 * ClassName: TeamIdException
 *自定义的异常类
 * @author wanglina
 * @version 1.0
 */
public class TeamIDException extends TeamException{
    public TeamIDException() {
    }

    public TeamIDException(String message) {
        super(message);
    }
}
```

```
package com.lina.exceptions;
/**
 * ClassName: TeamException
 * 自定义的异常类
 * @author wanglina
 * @version 1.0
 */
public class TeamNameException extends  TeamException{
    public TeamNameException() {
    }

    public TeamNameException(String message) {
        super(message);
    }
}
```

编写控制器

```
package com.lina.controller;
import com.lina.exceptions.TeamException;
import com.lina.exceptions.TeamIdException;
import com.lina.exceptions.TeamNameException;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

/**
 * ClassName: ExController
 * 测试异常处理的控制器
```

```
 * @author wanglina
 * @version 1.0
 */
@Controller
@RequestMapping("ex")
public class ExController {

    @RequestMapping("test01/{id}/{name}/{loc}")
    public ModelAndView test01(
            @PathVariable("id") Integer teamId,
            @PathVariable("name")String teamName,
            @PathVariable("loc")String loc) throws TeamException {
        ModelAndView mv=new ModelAndView();
        if(teamId<=1000){
            throw new TeamIDException("teamId不合法！必须在1000之上！");
        }
        if("test".equalsIgnoreCase(teamName)){
            throw new TeamNameException("teamName不合法！不能使用test");
        }
        if("test".equalsIgnoreCase(loc)){
            throw  new TeamException("team出现了异常！");
        }
        //System.out.println(10/0);
        mv.setViewName("OK");
        return mv;
    }

    @ExceptionHandler(value =
{TeamIdException.class,TeamNameException.class,Exception.class})
    public ModelAndView exHandler(Exception ex){
        ModelAndView mv=new ModelAndView();
        mv.addObject("msg",ex.getMessage());
        if(ex instanceof TeamIdException)
             mv.setViewName("idError");
        else if(ex instanceof TeamNameException)
            mv.setViewName("nameError");
        else
            mv.setViewName("error");
        return mv;
    }
}
```

编写error.jsp、idError.jsp、nameError.jsp页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>error</title>
</head>
<body>
    <h1>默认的错误页面--${msg}</h1>
</body>
</html>
```

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>id error</title>
</head>
<body>
    <h1>teamId Error----${msg}</h1>
</body>
</html>
```

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>name error</title>
</head>
<body>
    <h1>teamName error---${msg}</h1>
</body>
</html>
```

测试：

http://localhost:8080/ex/test01/100/test1

http://localhost:8080/ex/test01/1004/test

http://localhost:8080/ex/test01/1001/test1

10.3优化

一般将异常处理方法专门定义在一个类中，作为全局的异常处理类。

使用注解@ControllerAdvice，就是“控制器增强”，是给控制器对象增强功能的。使用@ControllerAdvice 修饰的类中可以使用@ExceptionHandler。

当使用@RequestMapping 注解修饰的方法抛出异常时，会执行@ControllerAdvice 修饰的类中的异常处理方法。

@ControllerAdvice 注解所在的类需要进行包扫描，否则无法创建对象。

```
<context:component-scan base-package="com.lina.exceptions"/>
```

定义全局异常处理类：

```
package com.lina.exceptions;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.servlet.ModelAndView;

/**
 * 自定义的全局的异常处理类
 */
@ControllerAdvice
public class GlobalException {
```

```
    @ExceptionHandler(value = {TeamIDException.class})
    public ModelAndView exHandler1(Exception ex){
        ModelAndView mv=new ModelAndView();
            mv.setViewName("idError");
        mv.addObject("msg",ex.getMessage());
        return mv;
    }
    @ExceptionHandler(value = {TeamNameException.class})
    public ModelAndView exHandler2(Exception ex){
        ModelAndView mv=new ModelAndView();
            mv.setViewName("nameError");
        mv.addObject("msg",ex.getMessage());
        return mv;
    }
    @ExceptionHandler(value ={TeamException.class})
    public ModelAndView exHandler3(Exception ex){
        ModelAndView mv=new ModelAndView();
            mv.setViewName("error");
        mv.addObject("msg",ex.getMessage());
        return mv;
    }
    @ExceptionHandler(value ={Exception.class})
    public ModelAndView exHandler4(Exception ex){
        ModelAndView mv=new ModelAndView();
        mv.setViewName("error");
        mv.addObject("msg",ex.getMessage());
        return mv;
    }
}
```

11、拦截器

SpringMVC 中的 拦截器（Interceptor）是非常重要的，它的主要作用是拦截指定的用户请求，并进行相应的预处理与后处理。

拦截的时间点在“处理器映射器HandlerMapping根据用户提交的请求映射出了所要执行的处理器类，并且也找到了要执行该处理器类的处理器适配器，在处理器适配器HandlerAdaptor执行处理器之前”。

在处理器映射器映射出所要执行的处理器类时，已经将拦截器与处理器组合为了一个处理器执行链HandlerExecutionChain，并返回给了前端控制器。

自定义拦截器，需要实现 HandlerInterceptor 接口。而该接口中含有三个方法：

```java
public interface HandlerInterceptor {

    /** Intercept the execution of a handler. Called after HandlerMapping determined ...*/
    default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {

        return true;
    }

    /** Intercept the execution of a handler. Called after HandlerAdapter actually ...*/
    default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
            @Nullable ModelAndView modelAndView) throws Exception {
    }

    /** Callback after completion of request processing, that is, after rendering ...*/
    default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
            @Nullable Exception ex) throws Exception {
    }

}
```

preHandle(request,response, Object handler)：
    该方法在处理器方法执行之前执行。其返回值为boolean，若为true，则紧接着会执行处理器方法，且会将afterCompletion()方法放入到一个专门的方法栈中等待执行。

postHandle(request,response, Object handler,modelAndView)：
    该方法在处理器方法执行之后执行。处理器方法若最终未被执行，则该方法不会执行。由于该方法是在处理器方法执行完后执行，且该方法参数中包含 ModelAndView，所以该方法可以修改处理器方法的处理结果数据，且可以修改跳转方向。

afterCompletion(request,response, Object handler, Exception ex)：
    当 preHandle()方法返回true时，会将该方法放到专门的方法栈中，等到对请求进行响应的所工作完成之后才执行该方法。即该方法是在前端控制器渲染（数据填充）了响应页面之后执行的，此时对ModelAndView再操作也对响应无济于事。

afterCompletion最后执行的方法，清除资源，例如在Controller方法中加入数据

11.1 自定义拦截器

```java
package com.lina.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * ClassName: MyInterceptor
 * 自定义拦截器
 * @author wanglina
 * @version 1.0
 */
public class MyInterceptor implements HandlerInterceptor {

    //执行时间：  控制器方法执行之前，在ModelAndView返回之前
    //使用场景: 登录验证
    // 返回值 true : 继续执行控制器方法 表示放行     false: 不会继续执行控制器方法，表示拦截
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
```

```
        System.out.println("preHandle-------------------");
        return true;
    }

    //执行时间：  控制器方法执行之hou后，在ModelAndView返回之前，有机会修改返回值
    //使用场景：日记记录，记录登录的ip,时间
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("postHandle-------------------");
    }

    //执行时间：  控制器方法执行之后，在ModelAndView返回之后，没有机会修改返回值
    //使用场景：全局资源的一些操作
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("afterCompletion-------------------");
    }
}
```

```
package com.lina.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * ClassName: MyInterceptor
 * 自定义拦截器
 * @author wanglina
 * @version 1.0
 */
public class MyInterceptor2 implements HandlerInterceptor {

    //执行时间：  控制器方法执行之前，在ModelAndView返回之前
    //使用场景：登录验证
    // 返回值 true : 继续执行控制器方法 表示放行     false: 不会继续执行控制器方法，表示拦截
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("preHandle2-------------------");
        return true;
    }

    //执行时间：  控制器方法执行之hou后，在ModelAndView返回之前，有机会修改返回值
    //使用场景：日记记录，记录登录的ip,时间
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("postHandle-2-------------------");
    }

    //执行时间：  控制器方法执行之后，在ModelAndView返回之后，没有机会修改返回值
```

```
    //使用场景: 全局资源的一些操作
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("afterCompletion2--------------------");
    }
}
```

11.2 配置拦截器

在springmvc.xml中添加如下配置

```
<!-- 配置拦截器 -->
    <mvc:interceptors>
        <!-- 这里可以同时配置多个拦截器，配置的顺序就是拦截器的拦截顺序 -->
        <mvc:interceptor>
            <!-- 拦截器要拦截的请求路径  拦截所有用/** -->
            <mvc:mapping path="/**"/>
            <!-- 指定干活的拦截器 -->
             <bean class="com.lina.interceptor.MyInterceptor2"
id="myInterceptor"></bean>
        </mvc:interceptor>
        <mvc:interceptor>
            <!-- 拦截器要拦截的请求路径  拦截所有用/** -->
            <mvc:mapping path="/**"/>
            <!-- 指定干活的拦截器 -->
             <bean class="com.lina.interceptor.MyInterceptor2"
id="myInterceptor2"></bean>
        </mvc:interceptor>
    </mvc:interceptors>
    如果有多个拦截器的时候：
preHandle：按照配置前后顺序执行
postHandle：按照配置前后逆序执行
afterCompletion：按照配置前后逆序执行
```

12、文件上传和下载

Spring MVC为文件上传提供了直接支持,这种支持是通过即插即用的MultipartResolver实现.

Spring中有一个MultipartResolver的实现类:CommonsMultipartResolver。

在SpringMVC上下文中默认没有装配MultipartResolver,因此默认情况下不能处理文件上传工作。

如果想使用Spring的文件上传功能,则需要先在上下文中配置MultipartResolver。

12.1 文件上传

（1）添加依赖

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

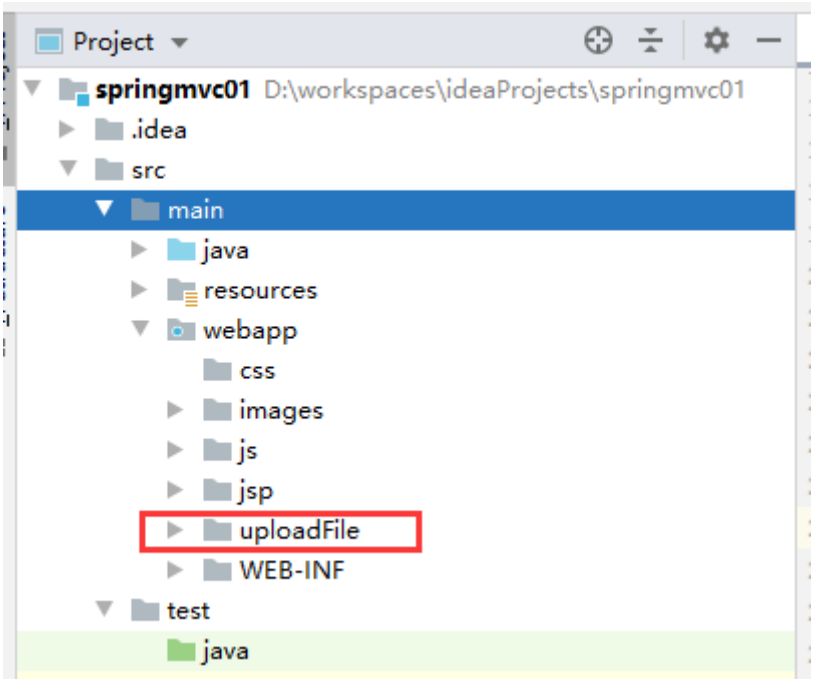
(2)springmvc.xml文件中配置MultipartResolver:

```
<!--文件上传-->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    </bean>
```

(3) fileHandle.jsp页面表单

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>文件操作</title>
</head>
<body>
    <form action="/file/upload" method="post" enctype="multipart/form-data">
        请选择文件：<input type="file" name="myFile" /><br/>
        <button type="submit">上传文件</button>
    </form>
</body>
</html>
```

(4) 配置java代码(注意要创建文件夹保存上传之后的文件)




```
package com.lina.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;

import javax.servlet.http.HttpServletRequest;
import java.io.File;
import java.io.IOException;
import java.util.UUID;
```

```
/**
 * ClassName: FileController
 * 文件操作的控制器
 * @author wanglina
 * @version 1.0
 */
@Controller
@RequestMapping("file")
public class FileController {

    /**
     * 文件上传
     * @param myFile
     * @param request
     * @return
     */
    @RequestMapping("upload")
    public String upload(@RequestParam("myFile") MultipartFile myFile,
HttpServletRequest request){
        //获取文件的原始名称 d:\te.aa\txcat.jpg
        String originalFilename = myFile.getOriginalFilename();
        // 实际开发中，一般都要将文件重新名称进行存储
        // 存储到服务器的文件名称=随机的字符串+根据实际名称获取到源文件的后缀
        String fileName= UUID.randomUUID().toString().replace("-","")
+originalFilename.substring(originalFilename.lastIndexOf("."));
        System.out.println(fileName);
        //文件存储路径
        String realPath =
request.getServletContext().getRealPath("/uploadFile")+"/";
        try {
            myFile.transferTo(new File(realPath+fileName));//真正的文件上传到服务器指
定的位置
            System.out.println("上传成功！"+realPath+fileName);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "OK";
    }

    @RequestMapping("hello")
    public String hello(){
        return "fileHandle";
    }
}
```

优化：如果需要限定文件的大小，可以通过配置文件的方式，但是不好捕捉异常。

```
<!--配置拦截器-->
    <mvc:interceptors>
        ......
        <mvc:interceptor>
            <mvc:mapping path="/**"/>
            <bean class="com.lina.interceptor.FileInterceptor"
id="fileInterceptor"></bean>
        </mvc:interceptor>
```

```
    </mvc:interceptors>
<!--文件上传-->
   <!--文件上传的配置-->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!--限定文件上传的大小 以字节B为单位  1024B=1Kb   1024KB=1M  -->
        <!--<property name="maxUploadSize" value="512000"/>-->
        <property name="defaultEncoding" value="utf-8"/>
        <property name="resolveLazily" value="true"/>
    </bean>
```

提供拦截器的方式：

```
package com.lina.interceptor;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.web.multipart.MultipartHttpServletRequest;
import org.springframework.web.servlet.HandlerInterceptor;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Iterator;
import java.util.Map;
    /**
 * ClassName: FileInterceptor
 * 文件后缀处理的拦截器
 * @author wanglina
 * @version 1.0
 */
public class FileInterceptor implements HandlerInterceptor {
    /**
     * 在文件上传之前判断文件后缀是否合法
     */
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        //判断是否是文件上传的请求
        boolean flag=true;
        if(request instanceof MultipartHttpServletRequest){
            MultipartHttpServletRequest multipartRequest=
(MultipartHttpServletRequest) request;
            Map<String, MultipartFile> fileMap = multipartRequest.getFileMap();
            //遍历文件
            Iterator<String> iterator = fileMap.keySet().iterator();
            while(iterator.hasNext()){
                String key = iterator.next();
                MultipartFile file = multipartRequest.getFile(key);
                String originalFilename = file.getOriginalFilename();
                String hz =
originalFilename.substring(originalFilename.lastIndexOf("."));
                //判断后缀是否合法
                if(!hz.toLowerCase().equals(".png") &&
!hz.toLowerCase().equals(".jpg")){

 request.getRequestDispatcher("/jsp/fileTypError.jsp").forward(request,response)
;
                    flag=false;
                }
            }
```

```
        }
        return flag;
    }
}
```

上传文件类型错误跳转的页面fileError.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>fileError</title>
</head>
<body>
    <h1>文件上传的类型有误！后缀必须是.png或者是.jpg</h1>
</body>
</html>
```

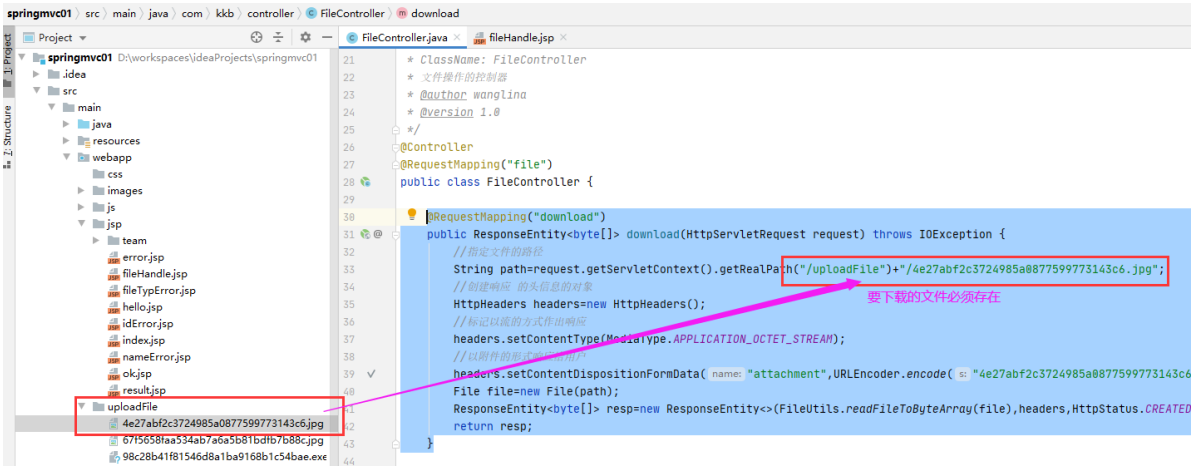
12.2 文件下载

(1）前端页面

```
<form action="/file/download" method="post" enctype="multipart/form-data">
        <button type="submit">下载图片-
-4e27abf2c3724985a0877599773143c6.jpg</button>
    </form>
```

(2)配置处理类方法

```
@RequestMapping("download")
    public ResponseEntity<byte[]> download(HttpServletRequest request) throws
IOException {
        //指定文件的路径-- 要保证指定的路径下有该文件才可以
        String
path=request.getServletContext().getRealPath("/uploadFile")+"/4e27abf2c3724985a0
877599773143c6.jpg";
        //创建响应 的头信息的对象
        HttpHeaders headers=new HttpHeaders();
        //标记以流的方式作出响应
        headers.setContentType(MediaType.APPLICATION_OCTET_STREAM);
        //以附件的形式响应给用户

 headers.setContentDispositionFormData("attachment",URLEncoder.encode("4e27abf2c
3724985a0877599773143c6.jpg","utf-8"));
        File file=new File(path);
        ResponseEntity<byte[]> resp=new ResponseEntity<>
(FileUtils.readFileToByteArray(file),headers,HttpStatus.CREATED);
        return resp;
    }
```



13、RESTful风格

13.1 REST概念

REST(英文：Representational State Transfer，简称**REST**，意思：表述性状态转换，描述了一个架构样式的网络系统，比如web应用)。

它是一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件，它主要用于**客户端和服务端**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。

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

13.2 RESTful概念

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

RESTful的特性：

资源(Resources)：互联网所有的事物都可以被抽象为资源 。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。可以用一个URI（统一资源定位符）指向它，每种资源对应一个特性的URI。要获取这个资源，访问它的URI就可以，因此URI即为每一个资源的独一无二的识别符。

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

状态转换(State Transfer)：每发出一个请求，就代表了客户端和服务器的一次交互过程。HTTP协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转换”(State Transfer)。而这种转换是建立在表现层之上的，所以就是“表现层状态转换”。

具体来说就是HTTP协议里面，四个表示操作方式的动词：**GET、POST、PUT、DELETE**。他们分别对应四种基本操作：GET用来**获取**资源，POST用来**新建**资源，PUT用来**更新**资源，DELETE用来**删除**资源。

原来的操作资源的方式：

http://localhost:8080/getExpress.do?id=1

http://localhost:8080/saveExpress.do

http://localhost:8080/updateExpress.do

http://localhost:8080/deleteExpress.do?id=1

原来用的方式当然没有问题，但是如果有更简洁的方式就更好了，此时就是RESTful风格。

使用RESTful操作资源：

GET /expresses #查询所有的快递信息列表

GET /express/1006 #查询一个快递信息

POST /express #新建一个快递信息

PUT /express/1006 #更新一个快递信息(全部更新)

PATCH /express/1006 #更新一个快递信息（部分更新）

DELETE /express/1006 #删除一个快递信息

13.3 API设计/URL设计

13.3.1 动词+宾语

RESTful 的核心思想就是客户端的用户发出的数据操作指令都是"动词 + 宾语"的结构。例如GET /expresses 这个命令，GET是动词，/expresses 是宾语。

```
动词通常就是五种 HTTP 方法，对应 CRUD 操作。
    GET：读取（Read）
    POST：新建（Create）
    PUT：更新（Update）
    PATCH：更新（Update），通常是部分更新
    DELETE：删除（Delete）

PS：1、根据 HTTP 规范，动词一律大写。
    2、一些代理只支持POST和GET方法， 为了使用这些有限方法支持RESTful API，需要一种办法覆盖http原来的方法。使用订制的HTTP头 X-HTTP-Method-Override 来覆盖POST 方法.
```

13.3.2 宾语必须是名词

宾语就是 API 的 URL，是 HTTP 动词作用的对象。它应该是名词，不能是动词。

比如，/expresses 这个 URL 就是正确的。

```
以下这些URL都是不推荐的，因为带上了动词，不是推荐写法。
/getAllExpresses
/getExpress
/createExpress
/deleteAllExpress
```

> ps:不要混淆名词单数和复数，为了保持简单，只对所有资源使用复数。

13.3.3 避免多级URL

如果资源中有多级分类，也不建议写出多级的URL。例如要获取球队中某个队员，有人可能这么写：

GET /team/1001/player/1005

这种写法的语义不够明确，所以推荐使用查询字符串做后缀，改写为

GET /team/1001?player=1005.

```
再例如查询所有的还未取出的快递，你该如何编写呢？
GET /expresses/statu  不推荐
GET /expresses?statu=false  推荐
```

13.4 HTTP状态码

客户端的用户发起的每一次请求，服务器都必须给出响应。响应包括 **HTTP 状态码**和**数据**两部分。

HTTP 状态码就是一个三位数，分成五个类别。这五大类总包含了100多种状态码（不需要全都记住，不用紧张哈，覆盖了绝大部分可能遇到的情况。每一种状态码都有标准的（或者约定的）解释，客户端只需查看状态码，就可以判断出发生了什么情况，所以服务器应该返回尽可能精确的状态码。

```
五类状态码分别如下：
    1xx：相关信息
    2xx：操作成功
    3xx：重定向
    4xx：客户端错误
    5xx：服务器错误
    PS：API 不需要1xx状态码，所以这个类别直接忽略。
```

13.4.1 状态码2xx

`200`状态码表示操作成功，但是不同的方法可以返回更精确的状态码。

```
GET：200 OK  表示一切正常
POST：201 Created  表示新的资源已经成功创建
PUT：200 OK
PATCH：200 OK
DELETE：204 No Content  表示资源已经成功删除
```

13.4.2 状态码3xx

API 用不到`301`状态码（永久重定向）和`302`状态码（暂时重定向，`307`也是这个含义），因为它们可以由应用级别返回，浏览器会直接跳转，API 级别可以不考虑这两种情况。

API 用到的`3xx`状态码，主要是`303 See Other`，表示参考另一个 URL。它与`302`和`307`的含义一样，也是"暂时重定向"，区别在于`302`和`307`用于`GET`请求，而`303`用于`POST`、`PUT`和`DELETE`请求。收到`303`以后，浏览器不会自动跳转，而会让用户自己决定下一步怎么办。

```
我们只需要关注一下304状态码就可以了
304 ：  Not Modified  客户端使用缓存数据
```

13.4.3 状态码4xx

`4xx`状态码表示客户端错误。

```
400 Bad Request：服务器不理解客户端的请求，未做任何处理。
401 Unauthorized：用户未提供身份验证凭据，或者没有通过身份验证。
403 Forbidden：用户通过了身份验证，但是不具有访问资源所需的权限。
404 Not Found：所请求的资源不存在，或不可用。
405 Method Not Allowed：用户已经通过身份验证，但是所用的 HTTP 方法不在他的权限之内。
410 Gone：所请求的资源已从这个地址转移，不再可用。
415 Unsupported Media Type：客户端要求的返回格式不支持。比如，API 只能返回 JSON 格式，但是客户端要求返回 XML 格式。
422 Unprocessable Entity ：客户端上传的附件无法处理，导致请求失败。
429 Too Many Requests：客户端的请求次数超过限额。
```

13.4.4 状态码5xx

`5xx` 状态码表示服务端错误。一般来说，API 不会向用户透露服务器的详细信息，所以只要两个状态码就够了。

```
500 Internal Server Error：客户端请求有效，服务器处理时发生了意外。
503 Service Unavailable：服务器无法处理请求，一般用于网站维护状态。
```

13.5 服务器响应

服务器返回的信息一般不推荐纯文本，而是建议大家选择JSON 对象，因为这样才能返回标准的结构化数据。

所以，服务器回应的 HTTP 头的`Content-Type`属性要设为`application/json`。客户端请求时，也要明确告诉服务器，可以接受 JSON 格式，即请求的 HTTP 头的`ACCEPT`属性也要设成`application/json`。

当发生错误的时候，除了返回状态码之外，也要返回错误信息。所以我们可以自己封装要返回的信息。

13.6 案例

13.6.1RESTful风格的查询

前端页面restful.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>restful</title>
    <script src="/js/jquery-1.11.1.js"></script>
</head>
<body>
    <form id="myForm" action="" method="post">
        球队ID:<input type="text" name="teamId" id="teamId" /><br/>
        球队名称:<input type="text" name="teamName" /><br/>
        球队位置:<input type="text" name="location" /><br/>
        <button type="button" id="btnGetAll">查询所有GET</button>
        <button type="button" id="btnGetOne">查询单个GET</button>
        <button type="button" id="btnPost">添加POST</button>
        <button type="button" id="btnPut">更新PUT</button>
        <button type="button" id="btnDel">删除DELETE</button>
    </form>
    <p id="showResult"></p>
</body>
```

```
</html>
<script>
    //页面加载完毕之后给按钮绑定事件
    $(function () {
        //给 查询所有GET 按钮绑定单击事件
        $("#btnGetAll").click(function () {
            //发起异步请求
            $.ajax({
                type: "GET",
                url: "/teams", //RESTful风格的API定义
                data: "",
                dataType:"json",
                success: function(list){
                    alert( "Data Saved: " + list );
                    var str="";
                    for(var i=0;i<list.length;i++){
                        var obj=list[i];
                        str+=obj.teamId+"----"+obj.teamName+"----
"+obj.location+"<br/>";
                    }
                    $("#showResult").html(str);
                }
            });
        });

        //给 查询单个GET 按钮绑定单击事件
        $("#btnGetOne").click(function () {
            //发起异步请求
            $.ajax({
                type: "GET",
                url: "/team/"+$("#teamId").val(), //RESTful风格的API定义
                data: "",
                dataType:"json",
                success: function(obj){
                    alert( "Data Saved: " + obj );
                    if(obj==""){
                        $("#showResult").html("没有复合条件的数据！");
                    }else
                     $("#showResult").html(obj.teamId+"----"+obj.teamName+"----
"+obj.location+"<br/>");
                }
            });
        });
    });
</script>
```

控制器RestfulController.java

```
package com.lina.controller;

import com.lina.pojo.Team;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
```

```java
import java.util.ArrayList;
import java.util.List;

/**
 * ClassName: RestfulController
 * restful风格的控制器
 * @author wanglina
 * @version 1.0
 */
@Controller
public class RestfulController {

    private static List<Team> teamList;
    static {
        teamList=new ArrayList<>(3);
        for(int i=1;i<=3;i++){
            Team team=new Team();
            team.setTeamId(1000+i);
            team.setTeamName("湖人"+i);
            team.setLocation("洛杉矶"+i);
            teamList.add(team);
        }
    }

    /**
     * 查询所有的球队
     * @return
     */
    @RequestMapping(value = "/teams",method = RequestMethod.GET)
    @ResponseBody
    public List<Team> getAll(){
        System.out.println("查询所有GET--发起的请求---------");
        return teamList;
    }

    /**
     * 根据id 查询单个的球队:
     * @return
     */
    @RequestMapping(value = "/team/{id}",method = RequestMethod.GET)
    @ResponseBody
    public Team getOne(@PathVariable("id")int id){
        System.out.println("查询单个GET--发起的请求---------");
        for (Team team : teamList) {
            if(team.getTeamId()==id){//整数比对值：所以方法参数写为int而不是Integer
                return team;
            }
        }
        return null;
    }

    @RequestMapping("hello")
    public String hello(){
        return "restful";
    }
}
```

13.6.2RESTful风格的添加

前端页面上脚本中添加如下内容：

```
        //给 添加POST 按钮绑定单击事件
        $("#btnPost").click(function () {
            alert($("#myForm").serialize());
            //发起异步请求
            $.ajax({
                type: "POST",
                url: "/team", //RESTful风格的API定义
                data: $("#myForm").serialize(),//表单的所有数据以？&形式追加在URL后面
/team?teamId=1006&teamName=kuaichuan&location=las
                dataType:"json",
                success: function(msg){
                    //alert( "Data Saved: " + msg );
                    $("#showResult").html(msg);
                }
            });
        });
```

控制器中添加方法

```
    /**
     * 添加一个球队
     */
    @RequestMapping(value = "team",method = RequestMethod.POST)
    @ResponseBody
    public String add(Team team){
        System.out.println("添加POST--发起的请求---------");
        teamList.add(team);
        return "201";
    }
```

13.6.3 RESTful风格的更新

前端页面上脚本中添加如下内容：

```
$("#btnPut").click(function(){
            alert($("#myForm").serialize());
            $.ajax({
                type: "POST",
                url: "/team/"+$("#teamId").val(),
                data: $("#myForm").serialize()+"&_method=PUT",
                dataType:"json",
                //headers:{"X-HTTP-Method-Override":"GET"},
                success: function(msg){
                    $("#showResult").html(msg);
                }
            });
        });
```

控制器中添加方法

```
@RequestMapping(value = "/team/{id}",method = RequestMethod.PUT)
    @ResponseBody
    public String update(@PathVariable("id") int id,Team newTeam){
        System.out.println(id);
        for (Team team : teamList) {
            if(team.getTeamId()==id){
                team.setTeamName(newTeam.getTeamName());
                team.setLocation(newTeam.getLocation());
                return "204";
            }
        }
        return "404";
    }
```

13.6.4RESTful风格的删除

前端页面上脚本中添加如下内容:

```
//给 删除DELETE 按钮绑定单击事件
        $("#btnDel").click(function () {
            alert($("#myForm").serialize());
            //发起异步请求
            $.ajax({
                type: "POST",
                url: "/team/"+$("#teamId").val(), //RESTful风格的API定义
                data: "_method=DELETE",
                success: function(msg){
                    //alert( "Data Saved: " + msg );
                    $("#showResult").html(msg);
                }
            });
        });
```

控制器中添加方法

```
/**
     * 根据ID删除一个球队
     */
    @RequestMapping(value = "team/{id}",method = RequestMethod.DELETE)
    @ResponseBody
    public String del(@PathVariable("id")int id){
        System.out.println("删除DELETE--发起的请求---------");
        for (Team team1 : teamList) {
            if(team1.getTeamId()==id){
                teamList.remove(team1);
                return "204";
            }
        }
        return "500";
    }
```

13.6.5 RESTful风格的更新和删除遇到的问题

13.6.5.1 遇到的问题：

在Ajax中，采用Restful风格PUT和DELETE请求传递参数无效，传递到后台的参数值为null

13.6.5.2 产生的原因：

Tomcat封装请求参数的过程：
1.将请求体中的数据，封装成一个map
2.request.getParameter(key)会从这个map中取值
3.SpringMvc封装POJO对象的时候，会把POJO中每个属性的值进行request.getParamter();
AJAX发送PU或者DELETE请求时，请求体中的数据通过request.getParamter()拿不到。
Tomcat一检测到是PUT或者DELETE就不会封装请求体中的数据为map，只有POST形式的请求才封装请求为map。

13.6.5.3 解决方案：

1、前端页面中的ajax发送请求的时候在url中加`&_method="PUT"`或者`&_method="DELETE"`即可

```
$("#btnDelete").click(function(){
    $.ajax({
        type: "POST",
        url: "/team/"+$("#teamId").val(),
        data: "_method=DELETE",
        dataType:"json",
        //headers:{"X-HTTP-Method-Override":"GET"},
        success: function(msg){
            $("#showResult").html(msg);
        }
    });
});

$("#btnPut").click(function(){
    alert($("#myForm").serialize());
    $.ajax({
        type: "POST",
        url: "/team/"+$("#teamId").val(),
        data: $("#myForm").serialize()+"&_method=PUT",
        dataType:"json",
        //headers:{"X-HTTP-Method-Override":"GET"},
        success: function(msg){
            $("#showResult").html(msg);
        }
    });
});
```

2、web.xml中配置

配置的时候多个过滤器需要注意顺序

```
<!-- 使用Rest风格的URI 将页面普通的post请求转为指定的delete或者put请求
原理：在Aajx中发送post请求后，带_method参数，将其修改为PUT，或者DELETE请求-->
    <filter>
        <filter-name>httpMethodFilter</filter-name>
        <filter-class>
            org.springframework.web.filter.HiddenHttpMethodFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>httpMethodFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```