

SpringMVC

SpringMVC的执行流程

MVC: 模型Model (dao, service) , 视图View (jsp) , 控制器Controller (servlet)

Controller: 控制器→取得表单数据, 调用业务逻辑; 转向指定的页面

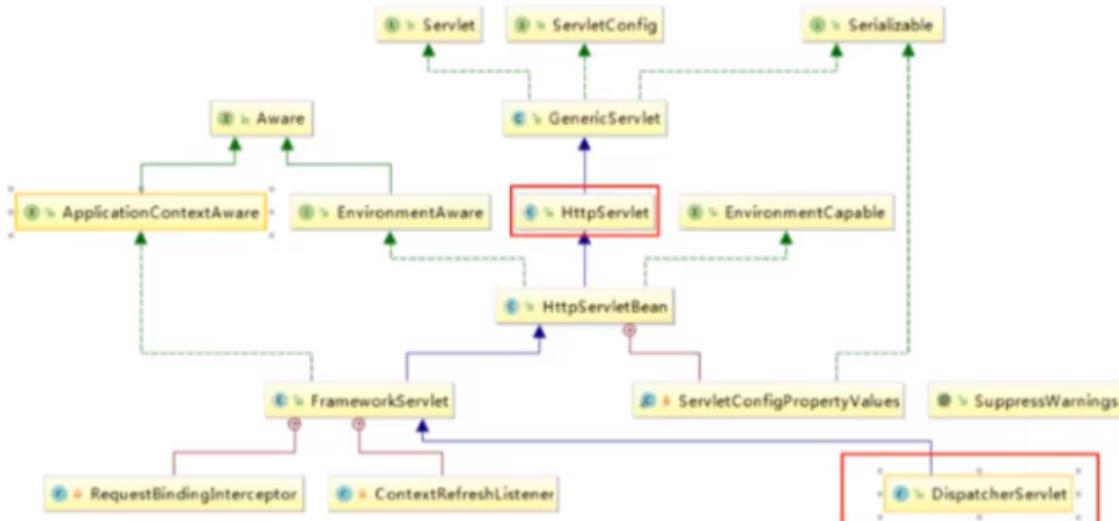
Model: 模型→业务逻辑; 保存数据的状态

View: 视图→显示页面

中心控制器

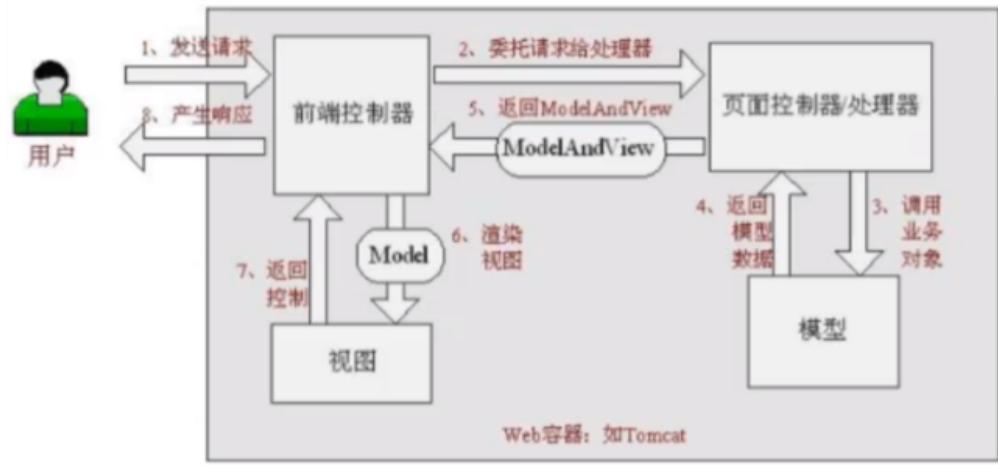
Spring的web框架围绕DispatcherServlet设计。 DispatcherServlet的作用是将请求分发到不同的处理器。从Spring 2.5开始, 使用Java 5或者以上版本的用户可以采用基于注解的controller声明方式。

Spring MVC框架像许多其他MVC框架一样, 以请求为驱动, 围绕一个中心Servlet分派请求及提供其他功能, DispatcherServlet是一个实际的Servlet (它继承自HttpServlet 基类)。

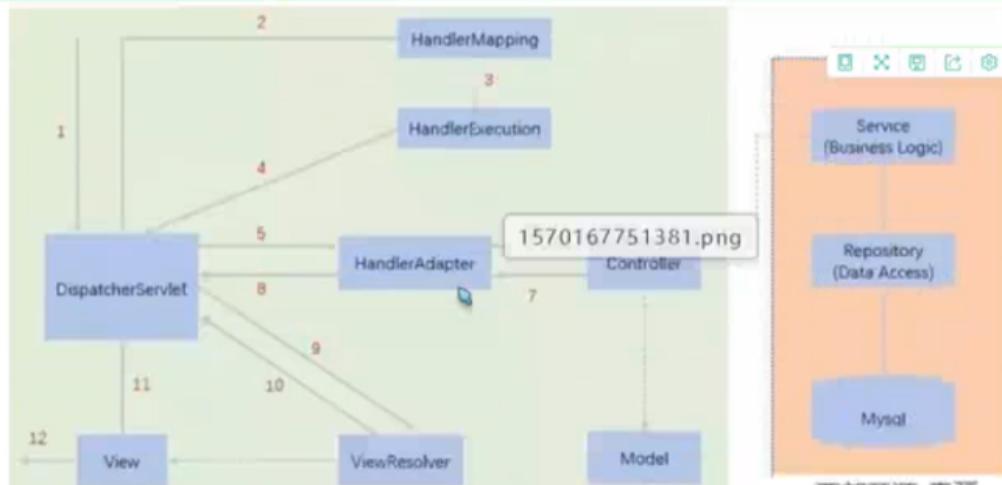


可能遇到的问题: 访问出现404, 排查步骤:

1. 查看控制台输出, 看一下是不是缺少了什么jar包。
2. 如果jar包存在, 显示无法输出, 就在IDEA的项目发布中, 添加lib依赖!
3. 重启Tomcat 即可解决!



Spring MVC 执行原理



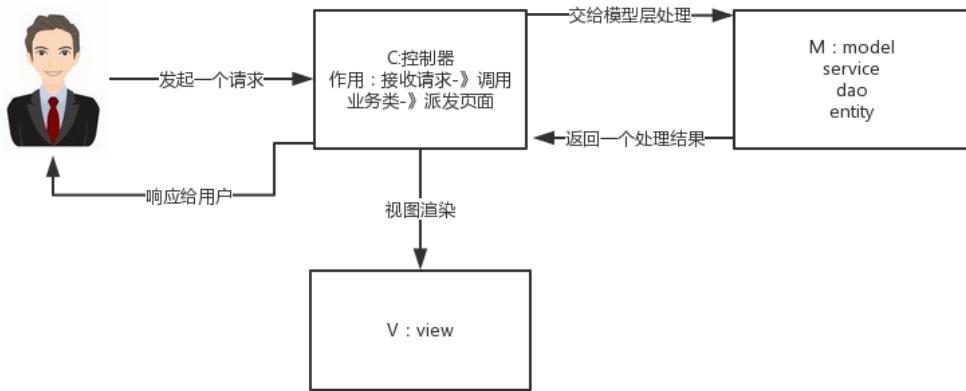
1、MVC工作原理

M:Model(完成业务逻辑 Service/dao/entity/)

V:View(完成界面渲染 jsp/html)

C:Controller(控制器->类似于CPU 接受请求->调用M->返回V)

MVC工作原理图



2、SpringMVC工作原理

Spring和SpringMVC的关系：

SpringMVC是一个MVC的开源框架，SpringMVC是Spring的一个后续产品，其实就是Spring在原有基础上，又提供了web应用的MVC模块，可以简单的把SpringMVC理解为是spring的一个模块（类似AOP，IOC这样的模块），网络上经常会说SpringMVC和Spring无缝集成，其实SpringMVC就是Spring的一个子模块，所以根本不需要同spring进行整合。

SpringMVC中的组件：

前端控制器（DispatcherServlet）：接收请求，响应结果，相当于电脑的CPU。

处理器映射器（HandlerMapping）：根据URL去查找处理器

处理器（Handler）：（需要程序员去写代码处理逻辑的）

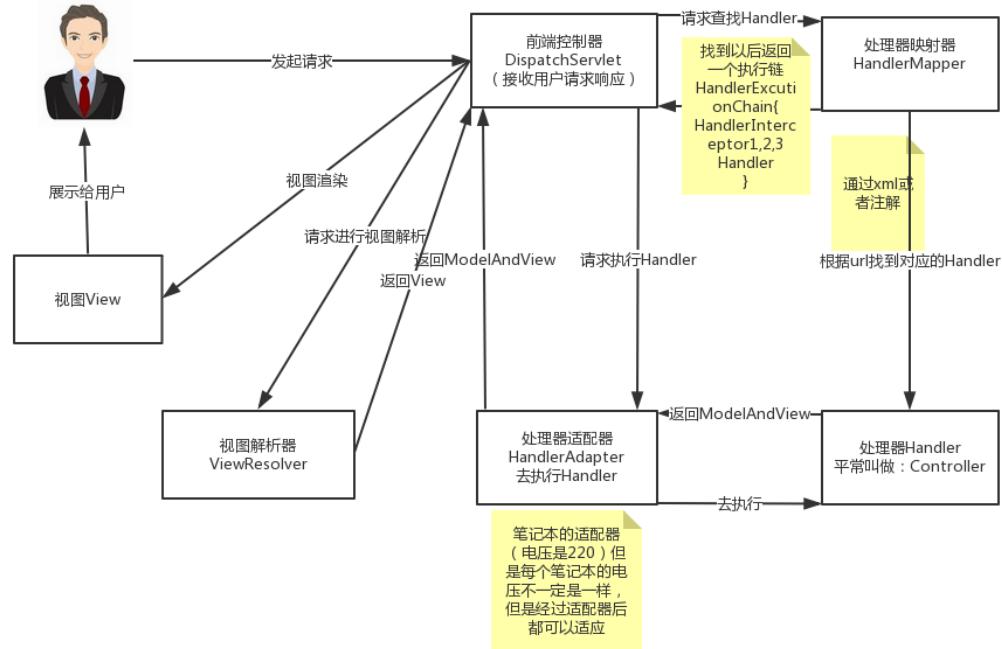
处理器适配器（HandlerAdapter）：会把处理器包装成适配器，这样就可以支持多种类型的处理器，类比笔记本的适配器（适配器模式的应用）

视图解析器（ViewResolver）：进行视图解析，多返回的字符串，进行处理，可以解析成对应的页面

SpringMvc工作原理

了解SpringMvc之前先看看Mvc的工作原理

SpringMvc工作原理图



SpringMvc工作流程：

第一步：用户发起请求到前端控制器（DispatcherServlet）

第二步：前端控制器请求处理器映射器（HandlerMapper）去查找处理器（Handler）：通过xml配置或者注解进行查找

第三步：找到以后处理器映射器（HandlerMapper）向前端控制器返回执行链（HandlerExecutionChain）

第四步：前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）去执行处理器（Handler）

第五步：处理器适配器去执行Handler

第六步：Handler执行完给处理器适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView

第八步：前端控制器请求视图解析器（ViewResolver）进行视图解析

第九步：视图解析器向前端控制器返回View

第十步：前端控制器对视图进行渲染

第十一步：前端控制器向用户响应结果

从宏观角度考虑，DispatcherServlet是整个Web应用的控制器；从微观考虑，Controller是单个Http请求处理过程中的控制器，而ModelAndView是Http请求过程中返回的模型（Model）和视图（View）

SpringMVC 重要组件说明

1、前端控制器DispatcherServlet（不需要工程师开发），由框架提供（重要）

作用：Spring MVC 的入口函数。接收请求，响应结果，相当于转发器，中央处理器。有了 DispatcherServlet 减少了其它组件之间的耦合度。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。

2、处理器映射器HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的url查找Handler。HandlerMapping负责根据用户请求找到Handler即处理器（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。

3、处理器适配器HandlerAdapter

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

4、处理器Handler(需要工程师开发)

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 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，所以一般情况需要工程师根据业务需求开发Handler。

5、视图解析器View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view） 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。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，需要由工程师根据业务需求开发具体的页面。

6、视图View(需要工程师开发)

View是一个接口，实现类支持不同的View类型（jsp、freemarker、pdf...）

注意：处理器Handler（也就是我们平常说的Controller控制器）以及视图层view都是需要我们自己手动开发的。其他的一些组件比如：前端控制器DispatcherServlet、处理器映射器HandlerMapping、处理器适配器HandlerAdapter等等都是框架提供给我们的，不需要自己手动开发。

编写

1. 注册DispatcherServlet

简要分析执行流程

1. DispatcherServlet表示前置控制器，是整个SpringMVC的控制中心。用户发出请求，

DispatcherServlet接收请求并拦截请求。

- 我们假设请求的url为：<http://localhost:8080/SpringMVC/hello>
- 如上url拆分成三部分：
 - <http://localhost:8080>服务器域名
 - SpringMVC部署在服务器上的web站点
 - hello表示控制器
- 通过分析，如上url表示为：请求位于服务器<localhost:8080>上的SpringMVC站点的hello控制器。

2. HandlerMapping为处理器映射。DispatcherServlet调用

HandlerMapping, HandlerMapping根据请求url查找Handler。

3. HandlerExecution表示具体的Handler,其主要作用是根据url查找控制器，如上url被查找控制器为：hello。

4. HandlerExecution将解析后的信息传递给DispatcherServlet,如解析控制器映射等。

5. HandlerAdapter表示处理器适配器，其按照特定的规则去执行Handler。

6. Handler让具体的Controller执行。

7. Controller将具体的执行信息返回给HandlerAdapter,如ModelAndView。

8. HandlerAdapter将视图逻辑名或模型传递给DispatcherServlet。

9. DispatcherServlet调用视图解析器(ViewResolver)来解析HandlerAdapter传递的逻辑视图名。

10. 视图解析器将解析的逻辑视图名传给DispatcherServlet。

11. DispatcherServlet根据视图解析器解析的视图结果，调用具体的视图。

12. 最终视图呈现给用户。

DispatcherServlet是整个Spring MVC的核心。它负责接收HTTP请求组织协调Spring MVC的各个组成部分。其主要工作有以下三项：

(1) 截获符合特定格式的URL请求。

(2) 初始化DispatcherServlet上下文对应的WebApplicationContext，并将其与业务层、持久化层的WebApplicationContext建立关联。

(3) 初始化Spring MVC的各个组成组件，并装配到DispatcherServlet中。

SpringMVC环境搭建：

1. 在web.xml中配置DispatcherServlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
      version="4.0">

    <!--SpringMVC的执行流程：
        (1) 客户端（浏览器）发送请求，直接请求到前端控制器DispatcherServlet。
        (2) DispatcherServlet根据请求信息调用处理器映射器HandlerMapping，解析请求对应的处理器Handler(即Controller)。
        (3) 解析到对应的Handler后，开始由HandlerAdapter适配器处理。-->
```

(4) HandlerAdapter会根据Handler来调用真正的处理器来处理请求，并处理相应的业务逻辑。

(5) 处理器处理完业务后，会返回一个 ModelAndView 对象，Model 返回的是数据对象，View 是个逻辑上的 View。

(6) 前端控制器请求视图解析器（ViewResolver）进行视图解析，ViewResolver 会根据逻辑 View 拼接实际的 View。

(7) 视图解析器向前端控制器返回 View，DispatcherServlet 把返回的数据对象传给实际的 View。

(8) 前端控制器进行视图渲染-->

```
<!--1. 注册DispatcherServlet-->
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!--关联一个springmvc的配置文件: 【servletname】-servlet.xml-->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springmvc-servlet.xml</param-value>
    </init-param>
    <!--启动级别: 1-->    <load-on-startup>1</load-on-startup>    </servlet>
<!-- 匹配所有的请求 (不包括.jsp) -->
<!--/* 匹配所有的请求 (包括.jsp) -->
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

</web-app>
```

编写 springmvc 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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--添加处理器映射器：根据请求的url查找Handler(即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。-->
    <bean
        class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/> <!--把
        URL映射到Controller类-->

    <!--添加处理器适配器：按照特定规则(HandlerAdapter要求的规则)去执行Handler
        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-->
    <bean
        class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

    <!--添加视图解析器：对DispatcherServlet传给它的 ModelAndView 进行解析，
        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
        1. 获取了 ModelAndView 的数据
        2. 解析 ModelAndView 的视图名字
        3. 拼接视图名字，找到对应的视图 /WEB-INF/jsp/hello
        4. 将数据渲染到这个视图上-->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          id="internalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/jsp/" />
        <!--后缀-->
        <property name="suffix" value=".jsp" />
```

```

</bean>

<!--Handler: 会返回一个ModelAndView对象, Model是返回的数据对象, View是个逻辑上的
View-->
<bean id="/hello" class="com.xiang.controller.HelloController"/>

</beans>

```

编写业务处理器

```

package com.xiang.controller;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.Mvc.Controller;

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

public class HelloController implements Controller {
    /*
    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
    在DispatcherServlet的控制下Handler对具体的用户请求进行处理
    */
    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {

        // ModelAndView模型和视图
        ModelAndView mv = new ModelAndView();

        // 封装对象, 放在 ModelAndView 中, Model
        mv.addObject("msg", "HelloSpringMVC! ! ");

        // 封装要跳转的视图, 放在 ModelAndView 中
        mv.setViewName("hello"); // 得到 /WEB-INF/jsp/hello.jsp
        return mv;
    }
}

```

跳转到 /WEB-INF/jsp/hello.jsp 目录下的 hello.jsp

```

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

```

小结

实现步骤其实非常的简单：

1. 新建一个web项目
2. 导入相关jar包
3. 编写web.xml，注册DispatcherServlet
4. 编写springmvc配置文件
5. 接下来就是去创建对应的控制类，controller
6. 最后完善前端视图和controller之间的对应
7. 测试运行调试。

使用springMVC必须配置的三大件：

处理器映射器、处理器适配器、视图解析器

通常，我们只需要手动配置视图解析器，而处理器映射器和处理器适配器只需要开启注解驱动即可，而省去了大段的xml配置

注意导包问题：如果项目的部署目录中没有在WEB-INF/lib目录下导入需要的包会报404错误。

注解实现springmvc

Controller：

```
package com.xiang.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

// @RequestMapping("/hellocontroller")
// @RestController 表示这个类不会被视图解析器解析，返回的就是一个字符串
@Controller
public class HelloController {

    //如果类上面加了请求映射则访问地址变成：
    http://localhost:8080/helloController/hello
    @RequestMapping("/hello")
    public String hello(Model model) {

        // 封装数据
        model.addAttribute("msg", "Hello, SpringMVCAnnotation!!");

        return "hello"; // 相当于setViewName="hello"会被视图解析器处理
    }
}
```

Springmvc-servlet.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
                           https://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           https://www.springframework.org/schema/mvc/spring-mvc.xsd">
```

```

<!--自动扫描包，让指定目录下的注解生效，由IOC容器统一管理-->
<context:component-scan base-package="com.xiang.controller"/>

<!--让SpringMVC不处理静态资源 过滤掉如 .css, .js, .html, .mp3, .mp4等静态资源-->
<mvc:default-servlet-handler/>

<!--
支持mvc注解驱动
在spring中一般采用@RequestMapping注解来完成映射关系
要想让@RequestMapping注解生效，必须向上下文注册
DefaultAnnotationHandlerMapping和一个AnnotationMethodHandlerAdopter实例
这两个实例分别在类级别和方法级别处理。而annotation-driven配置帮助我们自动完成上述两个实例的注入
-->
<mvc:annotation-driven/>

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

</beans>

```

Web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">

  <servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <!--通过初始化参数指定SpringMVC配置文件的位置，进行关联-->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc-servlet.xml</param-value>
    </init-param>
    <!--启动级别：数字越小，启动越早-->
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!--所有请求都会被springmvc拦截-->
  <servlet-mapping>
    <servlet-name>SpringMVC</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>

```

定义控制器的方式一：

Controller：

```

package com.xiang.controller;

import org.springframework.web.ModelAndView;
import org.springframework.web.mvc.Controller;

```

```

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

// 实现了Controller接口的类就是一个控制器了，不需要配置映射器和适配器，只需要配置视图解析器，并将这个类注入IOC容器即可
// 实现Controller接口定义控制器是一种较老的方法，缺点是：一个控制器中只能定义一个方法。
// 注意不要导错包，要导入org.springframework.web.servlet.mvc包
public class ControllerTest1 implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        //返回一个模型视图对象
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg", "Test1Controller test");
        mv.setViewName("test");
        return mv;
    }
}

```

Springmvc-servlet.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
                           https://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!--<context:component-scan base-package="com.xiang.controller"/>
    <mvc:default-servlet-handler/>
    <mvc:annotation-driven/>-->

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

    <bean name="/t1" class="com.xiang.controller.ControllerTest1"/>

</beans>

```

Web.xml:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                              http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">

    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!--通过初始化参数指定SpringMVC配置文件的位置，进行关联-->
        <init-param>
            <param-name>contextConfigLocation</param-name>

```

```

        <param-value>classpath:springmvc-servlet.xml</param-value>
    </init-param>
    <!--启动级别: 数字越小, 启动越早-->
    <load-on-startup>1</load-on-startup>
</servlet>

<!--所有请求都会被springmvc拦截-->
<servlet-mapping>
    <servlet-name>SpringMVC</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

</web-app>

```

使用注解@Controller定义控制器方式二:

RequestMapping

@RequestMapping

- @RequestMapping注解用于映射url到控制器类或一个特定的处理程序方法。可用于类或方法上。用于类上, 表示类中的所有响应请求的方法都是以该地址作为父路径。
- 为了测试结论更加准确, 我们可以加上一个项目名测试 myweb
- 只注解在方法上面

```

@Controller
public class TestController {
    @RequestMapping("/h1")
    public String test(){
        return "test";
    }
}

```

访问路径: http://localhost:8080 / 项目名 / h1

- 同时注解类与方法

```

@Controller
@RequestMapping("/admin")
public class TestController {
    @RequestMapping("/h1")
    public String test(){
        return "test";
    }
}

```

访问路径: http://localhost:8080 / 项目名/ admin /h1 , 需要先指定类的路径再指定方法的路径;

Web.xml不变

Springmvc-servlet.xml变为

```

<?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
                           https://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="com.xiang.controller"/>
<!--    <mvc:default-servlet-handler/>-->

```

```

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

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

<!--<bean name="/t1" class="com.xiang.controller.ControllerTest1"/>-->

</beans>

```

加了<context:component-scan base-package="com.xiang.controller"/>

Controller:

```

package com.xiang.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ControllerTest2 {
    @RequestMapping("/t2")
    public String test1(Model model) {
        model.addAttribute("msg", "Controllertest2");
        return "test";
    }

    @RequestMapping("/t3")
    public String test3(Model model) {
        model.addAttribute("msg", "test3");
        return "test";
    }
}

```

Restful 风格

概念

Restful就是一个资源定位及资源操作的风格。不是标准也不是协议，只是一种风格。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。

功能

- 资源：互联网所有的事物都可以被抽象为资源
- 资源操作：使用POST、DELETE、PUT、GET，使用不同方法对资源进行操作。
- 分别对应 添加、删除、修改、查询。

传统方式操作资源：通过不同的参数来实现不同的效果！方法单一，post 和 get

- `http://127.0.0.1/item/queryItem.action?id=1` 查询, GET
- `http://127.0.0.1/item/saveItem.action` 新增, POST
- `http://127.0.0.1/item/updateItem.action` 更新, POST
- `http://127.0.0.1/item/deleteItem.action?id=1` 删除, GET或POST

使用RESTful操作资源：可以通过不同的请求方式来实现不同的效果！如下：请求地址一样，但是功能可以不同！

- `http://127.0.0.1/item/1` 查询, GET
- `http://127.0.0.1/item` 新增, POST
- `http://127.0.0.1/item` 更新, PUT
- `http://127.0.0.1/item` 更新, PUT
- `http://127.0.0.1/item/1` 删除, DELETE

```
package com.xiang.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
public class RestfulController {
    // 原来通过url传参: http://localhost:8080/add?a=1&b=2
    // Restful风格传参: http://localhost:8080/add/1/2
    @RequestMapping(value = "/add/{a}/{b}") // 只有get方式才能请求到/add/a/b 这个注解
    // 等价于@GetMapping("/add/{a}/{b}")
    public String test1(@PathVariable int a, @PathVariable int b, Model model) {
        int res = a + b;
        model.addAttribute("msg", "结果1为: " + res);
        return "test";
    }

    @PostMapping("/add/{a}/{b}) // 等价于 @RequestMapping(value =
    // "/add/{a}/{b}", method = RequestMethod.POST)
    public String test2(@PathVariable int a, @PathVariable int b, Model model) {
        int res = a + b;
        model.addAttribute("msg", "结果2为: " + res);
        return "test";
    }
}
```

学习测试

1. 在新建一个类 RestFulController

```
@Controller  
public class RestFulController {  
}
```



2. 在Spring MVC中可以使用 @PathVariable 注解，让方法参数的值对应绑定到一个 URI 模板变量上。

```
@Controller  
public class RestFulController {  
  
    //映射访问路径  
    @RequestMapping("/commit/{p1}/{p2}")  
    public String index(@PathVariable int p1, @PathVariable int p2, Model model){  
  
        int result = p1+p2;  
        //Spring MVC会自动实例化一个Model对象用于向视图中传值  
        model.addAttribute("msg", "结果: "+result);  
        //返回视图位置  
        return "test";  
  
    }  
  
}
```

使用method属性指定请求类型

用于约束请求的类型，可以收窄请求范围。指定请求谓词的类型如GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE等

我们来测试一下：

- 增加一个方法

```
//映射访问路径,必须是POST请求  
@RequestMapping(value = "/hello",method = {RequestMethod.POST})  
public String index2(Model model){  
    model.addAttribute("msg", "hello!");  
    return "test";  
}
```

Spring MVC 的 @RequestMapping 注解能够处理 HTTP 请求的方法，比如 GET, PUT, POST, DELETE 以及 PATCH。

所有的地址栏请求默认都会是 HTTP GET 类型的。

方法级别的注解变体有如下几个：组合注解

```
@GetMapping  
@PostMapping  
@PutMapping  
@DeleteMapping  
@PatchMapping
```

@GetMapping 是一个组合注解

它所扮演的是 @RequestMapping(method = RequestMethod.GET) 的一个快捷方式。

平时使用的会比较多！

```
package com.xiang.controller;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;
```

```

import org.springframework.web.bind.annotation.*;

@Controller
public class RestfulController {
    // 原来通过url传参: http://localhost:8080/add?a=1&b=2
    // Restful风格传参: http://localhost:8080/add/1/2
    @RequestMapping(value = "/add/{a}/{b}") // 只有get方式才能请求到/add/a/b 这个注解
    // 等价于 @GetMapping("/add/{a}/{b}")
    public String test1(@PathVariable int a, @PathVariable int b, Model model) {
        int res = a + b;
        model.addAttribute("msg", "结果1为: " + res);
        return "test";
    }

    @PostMapping("/add/{a}/{b})" // 等价于 @RequestMapping(value =
    // "/add/{a}/{b}", method = RequestMethod.POST)
    public String test2(@PathVariable int a, @PathVariable int b, Model model) {
        int res = a + b;
        model.addAttribute("msg", "结果2为: " + res);
        return "test";
    }
}

```

SpringMVC: 结果跳转方式

2019-10-19 / Java / 狂神说

位置: Home > Java > 本页

ModelAndView

设置 ModelAndView 对象，根据 view 的名称，和视图解析器跳到指定的页面。

页面 : {视图解析器前缀} + viewName + {视图解析器后缀}

```

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

```

对应的 controller 类

```

public class ControllerTest1 implements Controller {

    public ModelAndView handleRequest(HttpServletRequest httpServletRequest, HttpServletResponse
        // 返回一个模型视图对象
        ModelAndView mv = new ModelAndView();
        mv.addObject("msg", "ControllerTest1");
        mv.setViewName("test");
        return mv;
    }
}

```

ServletAPI

通过设置ServletAPI，不需要视图解析器。

1. 通过HttpServletResponse进行输出
2. 通过HttpServletResponse实现重定向
3. 通过HttpServletResponse实现转发

```
@Controller  
public class ResultGo {  
  
    @RequestMapping("/result/t1")  
    public void test1(HttpServletRequest req, HttpServletResponse rsp) throws IOException  
    {  
        rsp.getWriter().println("Hello, Spring BY servlet API");  
    }  
  
    @RequestMapping("/result/t2")  
    public void test2(HttpServletRequest req, HttpServletResponse rsp) throws IOException  
    {  
        rsp.sendRedirect("/index.jsp");  
    }  
  
    @RequestMapping("/result/t3")  
    public void test3(HttpServletRequest req, HttpServletResponse rsp) throws Exception  
    {  
        //转发  
        req.setAttribute("msg", "/result/t3");  
        req.getRequestDispatcher("/WEB-INF/jsp/test.jsp").forward(req, rsp);  
    }  
}
```

SpringMVC

通过SpringMVC来实现转发和重定向 - 无需视图解析器；

测试前，需要将视图解析器注释掉

```
@Controller  
public class ResultSpringMVC {  
    @RequestMapping("/rsm/t1")  
    public String test1(){  
        //转发  
        return "/index.jsp";  
    }  
  
    @RequestMapping("/rsm/t2")  
    public String test2(){  
        //转发二  
        return "forward:/index.jsp";  
    }  
  
    @RequestMapping("/rsm/t3")  
    public String test3(){  
        //重定向  
        return "redirect:/index.jsp";  
    }  
}
```

通过SpringMVC来实现转发和重定向 - 有视图解析器；

重定向，不需要视图解析器，本质就是重新请求一个新地方嘛，所以注意路径问题。

可以重定向到另外一个请求实现。

```
@Controller
public class ResultSpringMVC2 {
    @RequestMapping("/rsm2/t1")
    public String test1(){
        //转发
        return "test";
    }

    @RequestMapping("/rsm2/t2")
    public String test2(){
        //重定向
        return "redirect:/index.jsp";
        //return "redirect:hello.do"; //hello.do为另一个请求/
    }
}
```

处理提交数据

1、提交的域名称和处理方法的参数名一致

提交数据：<http://localhost:8080/hello?name=kuangshen>

处理方法：

```
@RequestMapping("/hello")
public String hello(String name){
    System.out.println(name);
    return "hello";
}
```

后台输出：kuangshen

2、提交的域名称和处理方法的参数名不一致

提交数据：<http://localhost:8080/hello?username=kuangshen>

处理方法：

```
//@RequestParam("username") : username提交的域的名称 .
@RequestMapping("/hello")
public String hello(@RequestParam("username") String name){
    System.out.println(name);
    return "hello";
}
```

3、提交的是一个对象

要求提交的表单域和对象的属性名一致，参数使用对象即可

1. 实体类

```
public class User {  
    private int id;  
    private String name;  
    private int age;  
    //构造  
    //get/set  
    //toString()  
}
```

2. 提交数据：<http://localhost:8080/mvc04/user?name=kuangshen&id=1&age=15>

3. 处理方法：

```
@RequestMapping("/user")  
public String user(User user){  
    System.out.println(user);  
    return "hello";  
}
```

后台输出：User { id=1, name='kuangshen', age=15 }

说明：如果使用对象的话，前端传递的参数名和对象名必须一致，否则就是null。

数据显示到前端

第一种：通过 ModelAndView

我们前面一直都是如此。就不过多解释

```
public class ControllerTest1 implements Controller {  
  
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest, HttpServletResponse)  
        //返回一个模型视图对象  
        ModelAndView mv = new ModelAndView();  
        mv.addObject("msg", "ControllerTest1");  
        mv.setViewName("test");  
        return mv;  
    }  
}
```

第二种：通过 ModelMap

ModelMap

```
@RequestMapping("/hello")  
public String hello(@RequestParam("username") String name, ModelMap model){  
    //封装要显示到视图中的数据  
    //相当于req.setAttribute("name",name);  
    model.addAttribute("name",name);  
    System.out.println(name);  
    return "hello";  
}
```

对比

就对于新手而言简单来说使用区别就是：

```
Model 只有寥寥几个方法只适合用于储存数据，简化了新手对于Model对象的操作和理解；  
ModelMap 继承了 LinkedHashMap，除了实现了自身的一些方法，同样的继承 LinkedHashMap 的方法和特性；  
ModelAndView 可以在储存数据的同时，可以进行设置返回的逻辑视图，进行控制展示层的跳转。
```

当然更多的以后开发考虑的更多的是性能和优化，就不能单单仅限于此的了解。

用过滤器解决乱码问题

前端输入表单：

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>  
<html>  
<head>  
    <title>Title</title>  
</head>  
<body>  
  
<form action="/e/t1" method="post">  
    <input type="text" name="name">  
    <input type="submit">  
</form>  
  
</body>  
</html>
```

过滤器拦截，解决乱码问题：

```
package com.xiang.filter;  
  
import javax.servlet.*;  
import java.io.IOException;  
  
public class EncodingFilter implements Filter {  
    @Override  
    public void init(FilterConfig filterConfig) throws ServletException {  
    }  
  
    @Override  
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  
        servletRequest.setCharacterEncoding("utf-8");  
        servletResponse.setCharacterEncoding("utf-8");  
        servletResponse.setContentType("text/html; charset=UTF-8");  
        filterChain.doFilter(servletRequest, servletResponse);  
    }  
  
    @Override  
    public void destroy() {  
    }  
}
```

在web.xml中注册过滤器：

```
<!--自己编写的乱码过滤器-->  
<filter>  
    <filter-name>encoding</filter-name>  
    <filter-class>com.xiang.filter.EncodingFilter</filter-class>  
</filter>  
<filter-mapping>
```

```

<filter-name>encoding</filter-name>
<url-pattern>/*</url-pattern> <!--过滤所有请求，包括.jsp文件-->
</filter-mapping>
<!--配置SpringMVC自带的乱码过滤-->
<!--<filter>
    <filter-name>encoding</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>encoding</filter-name>
    <url-pattern>/*</url-pattern> &lt;!&ndash;过滤所有请求，包括.jsp文件&ndash;&gt;;
</filter-mapping>-->

```

Controller接收表单数据并转发：

```

package com.xiang.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class EncodingController {

    @RequestMapping("/e/t1")
    public String test1(@RequestParam("name") String name, Model model) {
        model.addAttribute("msg", name);
        return "test";
    }
}

```

注意：<url-pattern>/*</url-pattern> <!--过滤所有请求，包括.jsp文件-->

前后端分离时代：

后端部署后端，提供接口，提供数据

Json

前端独立部署，接收数据，负责渲染后端的数据

什么是JSON?

- JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式, 目前使用特别广泛。
- 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。

在 JavaScript 语言中, 一切都是对象。因此, 任何 JavaScript 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。看看他的要求和语法格式:

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

JSON 键值对是用来保存 JavaScript 对象的一种方式, 和 JavaScript 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:

```
{"name": "QinJiang"}  
{"age": "3"}  
{"sex": "男"}
```

很多人搞不清楚 JSON 和 JavaScript 对象的关系, 甚至连谁是谁都不清楚。其实, 可以这么理解:

- JSON 是 JavaScript 对象的字符串表示法, 它使用文本表示一个 JS 对象的信息, 本质是一个字符串。

```
var obj = {a: 'Hello', b: 'World'}; //这是一个对象, 注意键名也是可以使用引号包裹的  
var json = '{"a": "Hello", "b": "World"}'; //这是一个 JSON 字符串, 本质是一个字符串
```

JSON 和 JavaScript 对象互转

- 要实现从JSON字符串转换为JavaScript 对象, 使用 `JSON.parse()` 方法:

```
var obj = JSON.parse('{"a": "Hello", "b": "World"}');  
//结果是 {a: 'Hello', b: 'World'}
```

- 要实现从JavaScript 对象转换为JSON字符串, 使用 `JSON.stringify()` 方法:

```
var json = JSON.stringify({a: 'Hello', b: 'World'});  
//结果是 '{"a": "Hello", "b": "World"}'
```

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>Title</title>  
  
<script type="text/javascript">  
  
    // 编写一个JavaScript对象  
    var user = {  
        name: "王五",  
        age: 3,  
        sex: "男"  
    };
```

```

// 将js对象转换为json对象
var json = JSON.stringify(user); // 得到JSON格式的字符串
console.log(json); // {"name": "王五", "age": 3, "sex": "男"}
console.log(user); // {name: '王五', age: 3, sex: '男'}

// 将JSON对象转换为JavaScript对象
var obj = JSON.parse(json);
console.log(obj);
</script>

</head>
<body>
</body>
</html>

```

Controller返回JSON数据

- Jackson应该是目前比较好的json解析工具了
- 当然工具不止这一个，比如还有阿里巴巴的 fastjson 等等。
- 我们这里使用Jackson，使用它需要导入它的jar包；

```

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.8</version>
</dependency>

```

配置web.xml：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">

    <!--1. 注册servlet-->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!--通过初始化参数指定SpringMVC配置文件的位置，进行关联-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-servlet.xml</param-value>
        </init-param>
        <!--启动顺序，数字越小，启动越早-->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!--所有请求都会被springmvc拦截-->
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!--解决乱码问题-->
    <filter>
        <filter-name>encoding</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>

```

```
</init-param>
</filter>
<filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

</web-app>
```

注意: <param-value>classpath:springmvc-servlet.xml</param-value>, 如果springmvc-servlet.xml报红, 可能是因为没有将resources文件夹标记为资源目录

在导出JSON时:

- 发现出现了乱码问题, 我们需要设置一下他的编码格式为utf-8, 以及它返回的类型;
- 通过@RequestMaping的produces属性来实现, 修改下代码

```
//produces:指定响应体返回类型和编码
@RequestMapping(value = "/json1", produces = "application/json;charset=utf-8")
```

- 再次测试, http://localhost:8080/json1 , 乱码问题OK!

==【注意: 使用json记得处理乱码问题】==

代码优化

乱码统一解决

上一种方法比较麻烦, 如果项目中有许多请求则每一个都要添加, 可以通过Spring配置统一指定, 这样就不用每次都去处理了!

我们可以在springmvc的配置文件上添加一段消息StringHttpMessageConverter转换配置!

在springmvc-servlet.xml中加:

```
<mvc:annotation-driven>
    <mvc:message-converters register-defaults="true">
        <bean class="org.springframework.http.converter.StringHttpMessageConverter">
            <constructor-arg value="UTF-8"/>
        </bean>
        <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
            <property name="objectMapper">
                <bean class="org.springframework.http.converter.json.Jackson2ObjectMapperBuilder">
                    <property name="failOnEmptyBeans" value="false"/>
                </bean>
            </property>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

返回json字符串统一解决

```
package com.xiang.controller;

import com.fasterxml.jackson.core.JsonProcessingException;
```

```

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import com.xiang.pojo.User;
import com.xiang.utils.JsonUtils;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

// @Controller
@RequestMapping //表示这个类中方法返回的内容就是Return里的内容，不走视图解析器，  

//RestController是@Controller和@ResponseBody的结合体，两个标注合并起来的作用。
public class UserController {

    // produces: 指定响应体返回类型和编码，只能解决当前方法的JSON乱码问题
    @RequestMapping(value = "/j1") //, produces = "application/json;charset=utf-8"
    // @ResponseBody
    // @ResponseBody注解让这个方法不会走视图解析器，会直接返回一个字符串
    // @ResponseBody的作用：1. 将方法的返回值，以特定的格式写入到response的body区域，  

    //进而将数据返回给客户端。
    // 2. 当方法上面没有写ResponseBody，底层会将方法的返回值封装为 ModelAndView 对象。
    // 3. 如果返回值是字符串，那么直接将字符串写到客户端；如果是一个对象，会将对象转  

    //化为json串，然后写到客户端。
    public String json1() throws JsonProcessingException {

        // 使用 jackson 中的 ObjectMapper
        ObjectMapper mapper = new ObjectMapper();
        User user = new User("王维1号", 100, "男");
        String str = mapper.writeValueAsString(user);
        return str;
    }

    @RequestMapping("/j2")
    public String json2() throws JsonProcessingException {
        // ObjectMapper mapper = new ObjectMapper();
        List<User> userList = new ArrayList<>();

        User user1 = new User("王维1号", 3, "男");
        User user2 = new User("王维2号", 4, "男");
        User user3 = new User("王维3号", 5, "男");
        User user4 = new User("王维4号", 6, "男");

        userList.add(user1);
        userList.add(user2);
        userList.add(user3);
        userList.add(user4);

        // String str = mapper.writeValueAsString(userList);
        // return str;

        // 使用自定义工具类
        return JsonUtils.toJson(userList);
    }

    @RequestMapping("/j3")
    public String json3() throws JsonProcessingException {
        ObjectMapper mapper = new ObjectMapper();
        // ObjectMapper 解析时间后的默认格式为：Timestamp 时间戳 1970-1-1至今的ms数
    }
}

```

```

// 关闭时间戳格式输出Date()
mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);

// 自定义日期格式
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
mapper.setDateFormat(sdf);
Date date = new Date();
return mapper.writeValueAsString(date);

// 使用自定义工具类
// return JsonUtils.getJson(date, "yyyy-MM-dd HH:mm:ss");
}

}

```

自定义工具类

```

package com.xiang.utils;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;

import java.text.SimpleDateFormat;

public class JsonUtils {

    public static String getJson(Object object) {
        return getJson(object, "yyyy-MM-dd HH:mm:ss");
    }

    public static String getJson(Object object, String dateFormat) {
        ObjectMapper mapper = new ObjectMapper();
        // 关闭时间戳
        mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
        // 自定义日期格式
        SimpleDateFormat sdf = new SimpleDateFormat(dateFormat);
        mapper.setDateFormat(sdf);
        try {
            return mapper.writeValueAsString(object);
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        return null;
    }
}

```

FastJson

fastjson.jar是阿里开发的一款专门用于Java开发的包，可以方便的实现json对象与JavaBean对象的转换，实现JavaBean对象与json字符串的转换，实现json对象与json字符串的转换。实现json的转换方法很多，最后的实现结果都是一样的。

fastjson 的 pom 依赖！

```

<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.60</version>
</dependency>

```

fastjson 三个主要的类：

- 【JSONObject 代表 json 对象】
 - JSONObject实现了Map接口, 猜想 JSONObject底层操作是由Map实现的。
 - JSONObject对应json对象, 通过各种形式的get()方法可以获取json对象中的数据, 也可利用诸如size(), isEmpty()等方法获取“键：值”对的个数和判断是否为空。其本质是通过实现 Map接口并调用接口中的方法完成的。
- 【JSONArray 代表 json 对象数组】
 - 内部是有List接口中的方法来完成操作的。
- 【JSON 代表 JSONObject和JSONArray的转化】
 - JSON类源码分析与使用
 - 仔细观察这些方法, 主要是实现json对象, json对象数组, javabean对象, json字符串之间的相互转化。

```
@RequestMapping("/j4")
public String json4() {
    List<User> userList = new ArrayList<>();
    User user1 = new User("王维1号", 3, "男");
    User user2 = new User("王维2号", 4, "男");
    User user3 = new User("王维3号", 5, "男");
    User user4 = new User("王维4号", 6, "男");

    userList.add(user1);
    userList.add(user2);
    userList.add(user3);
    userList.add(user4);

    System.out.println("*****Java对象 转 JSON字符串*****");
    String str1 = JSON.toJSONString(userList);
    System.out.println(str1);
    String str2 = JSON.toJSONString(user);
    System.out.println(str2);

    System.out.println("*****JSON字符串 转 Java对象*****");
    User user = JSON.parseObject(str2, User.class);
    System.out.println(user);

    System.out.println("*****Java对象 转 JSON对象*****");
    JSONObject jsonObject1 = (JSONObject) JSON.toJSONObject(user2);
    System.out.println(jsonObject1.getString("name"));

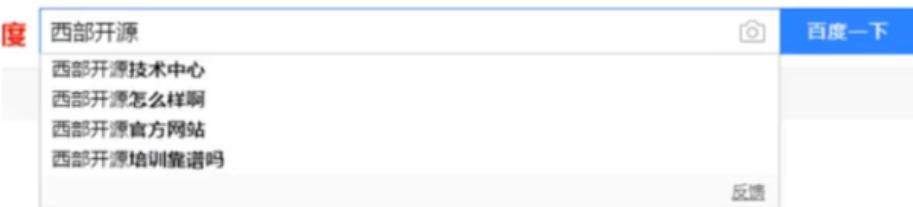
    System.out.println("*****JSON对象 转 Java对象*****");
    User javaUser = JSON.toJavaObject(jsonObject1, User.class);
    System.out.println(javaUser);

    return "hello";
}
```

SpringMVC: Ajax技术

简介

- **AJAX = Asynchronous JavaScript and XML (异步的 JavaScript 和 XML) 。**
- **AJAX 是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。**
- **Ajax 不是一种新的编程语言，而是一种用于创建更好更快以及交互性更强的Web应用程序的技术。**
- 在 2005 年，Google 通过其 Google Suggest 使 AJAX 变得流行起来。Google Suggest能够自动帮你完成搜索单词。
- **Google Suggest 使用 AJAX 创造出动态性极强的 web 界面：当您在谷歌的搜索框输入关键字时，JavaScript 会把这些字符发送到服务器，然后服务器会返回一个搜索建议的列表。**
- 就和国内百度的搜索框一样：



- 传统的网页(即不用ajax技术的网页)，想要更新内容或者提交一个表单，都需要重新加载整个网页。
- 使用ajax技术的网页，通过在后台服务器进行少量的数据交换，就可以实现异步局部更新。
- 使用Ajax，用户可以创建接近本地桌面应用的直接、高可用、更丰富、更动态的Web用户界面。

Ajax：异步刷新，前后端交互

伪造Ajax

我们可以使用前端的一个标签来伪造一个ajax的样子。iframe标签

1. 新建一个module：sspringmvc-06-ajax，导入web支持！
2. 编写一个 ajax-frame.html 使用 iframe 测试，感受下效果

3. 使用IDEA开浏览器测试一下！

利用AJAX可以做：

- 注册时，输入用户名自动检测用户是否已经存在。 ↴
- 登陆时，提示用户名密码错误
- 删除数据行时，将行ID发送到后台，后台在数据库中删除，数据库删除成功后，在页面DOM中将数据行也删除。
-等等

jQuery.ajax

- 纯JS原生实现Ajax我们不去讲解这里，直接使用jquery提供的，方便学习和使用，避免重复造轮子，有兴趣的同学可以去了解下JS原生XMLHttpRequest！
- Ajax的核心是XMLHttpRequest对象(XHR)。XHR为向服务器发送请求和解析服务器响应提供了接口。能够以异步方式从服务器获取新数据。
- jQuery 提供多个与 AJAX 有关的方法。
- 通过 jQuery AJAX 方法，您能够使用 HTTP Get 和 HTTP Post 从远程服务器上请求文本、HTML、XML 或 JSON – 同时您能够把这些外部数据直接载入网页的被选元素中。
- jQuery 不是生产者，而是大自然搬运工。
- jQuery Ajax本质就是 XMLHttpRequest，对他进行了封装，方便调用！

JQuery是一个库：jQuery是一个快速、简洁的[JavaScript](#)框架，是继[Prototype](#)之后又一个优秀的JavaScript代码库（框架）于2006年1月由[John Resig](#)发布。jQuery设计的宗旨是“write Less, Do More”，即倡导写更少的代码，做更多的事情。它封装JavaScript常用的功能代码，提供一种简便的JavaScript[设计模式](#)，优化[HTML](#)文档操作、事件处理、动画设计和[Ajax](#)交互。

jQuery的核心特性可以总结为：具有独特的链式语法和短小清晰的多功能接口；具有高效灵活的[CSS选择器](#)，并且可对[CSS](#)选择器进行扩展；拥有便捷的插件扩展机制和丰富的插件。jQuery兼容各种主流浏览器，如[IE](#) 6.0+、[FF](#) 1.5+、[Safari](#) 2.0+、[Opera](#) 9.0+等。

原生的Ajax：

```

<script type="text/javascript">
var xmlhttp;
function loadXMLDoc(url)
{
xmlhttp=null;
if (window.XMLHttpRequest)
{// code for all new browsers
xmlhttp=new XMLHttpRequest();
}
else if (window.ActiveXObject)
{// code for IEs and IE6
xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
if (xmlhttp!=null)
{
xmlhttp.onreadystatechange=state_Change;
xmlhttp.open("GET",url,true);
xmlhttp.send(null);
}
else
{
alert("Your browser does not support XMLHTTP.");
}
}

function state_Change()
{
if (xmlhttp.readyState==4)
{// 4 = "loaded"
if (xmlhttp.status==200)
{// 200 = OK
// ...our code here...
}
else
{
alert("Problem retrieving XML data");
}
}
}

```

jQuery.ajax(...)

部分参数：

url: 请求地址	<input type="button" value="搜索"/>	<input type="button" value="复制"/>
type: 请求方式, GET、POST (1.9.0之后用method)		
headers: 请求头		
data: 要发送的数据		
contentType: 即将发送信息至服务器的内容编码类型(默认：“application/x-www-form-urlencoded”)		
async: 是否异步		
timeout: 设置请求超时时间(毫秒)		
beforeSend: 发送请求前执行的函数(全局)		
complete: 完成之后执行的回调函数(全局)		
success: 成功之后执行的回调函数(全局)		
error: 失败之后执行的回调函数(全局)		
accepts: 通过请求头发送给服务器, 告诉服务器当前客户端可接受的数据类型		
dataType: 将服务器端返回的数据转换成指定类型		
“ xml ”: 将服务器端返回的内容转换成xml格式		
“ text ”: 将服务器端返回的内容转换成普通文本格式		
“ html ”: 将服务器端返回的内容转换成普通文本格式, 在插入DOM中时, 如果包含JavaScript标签		
“ script ”: 尝试将返回值当作JavaScript去执行, 然后再将服务器端返回的内容转换成普通文本格式		
“ json ”: 将服务器端返回的内容转换成相应的JavaScript对象		
“ jsonp ”: JSONP 格式使用 JSONP 形式调用函数时, 如 “myurl?callback=?” jQuery 将自动		

将下载jquery-3.6.0.js并将其放到web/statics/js/目录下。

在application.xml中添加静态资源过滤

1. 配置web.xml 和 springmvc的配置文件，复制上面案例的即可 【记得静态资源过滤和注解驱动配置上】

```
<?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/spring-beans.xsd
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- 自动扫描指定的包，下面所有注解类交给IOC容器管理 -->
    <context:component-scan base-package="com.kuang.controller"/>
    <mvc:default-servlet-handler />
    <mvc:annotation-driven />

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

编写application.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/spring-beans.xsd
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           https://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

    <!-- 静态资源过滤 -->
    <mvc:default-servlet-handler/>

    <!-- 注解驱动 -->
    <mvc:annotation-driven/>

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

    <!-- 使用Jackson解决JSON输出乱码问题 -->
    <mvc:annotation-driven>
        <mvc:message-converters register-defaults="true">
            <bean
                class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean
                class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
                <property name="objectMapper">
```

```

        <bean
class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
    <property name="failOnEmptyBeans" value="false"/>
</bean>
</property>
</bean>
</mvc:messageConverters>
</mvc:annotation-driven>

</beans>

```

编写web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">

    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:application.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <filter>
        <filter-name>encoding</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>encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

</web-app>

```

编写index.xml

```

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

        <script src="${pageContext.request.contextPath}/statics/js/jquery.js"></script>
        <script>/*
            $.等价于jQuery.
        */
        function a() {
            $.post({
                url:"${pageContext.request.contextPath}/a2",
                data:{'name':$('#username').val()}, /*接收前端数据*/
                success:function (data, status) { /*请求成功接收controller返回的数据*/

```

```

        console.log("data=" + data); /*在网站控制台输出*/
        console.log("status=" + status);
    },
    error:function () { /*请求失败*/
        }
    })
}
</script>

</head>
<body>

<%--失去焦点的时候，发送一个请求到后台--%>
用户名: <input type="text" id="username" onblur="a()">

</body>
</html>

```

编写controller

```

package com.xiang.controller;

import com.xiang.pojo.User;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@RestController
public class AjaxController {

    @RequestMapping("/a1")
    public String test(){
        return "hello";
    }

    @RequestMapping("/a2")
    public void a2(String name, HttpServletResponse response) throws IOException {
        System.out.println("a2:param=>" + name);
        if("zhangSan".equalsIgnoreCase(name)){
            response.getWriter().print("true");
        } else {
            response.getWriter().print("false");
        }
    }

    @RequestMapping("/a3")
    public List<User> a3(){
        List<User> userList = new ArrayList<>();
        userList.add(new User("张三",1,"男"));
        userList.add(new User("李四",2,"男"));
        userList.add(new User("王五",3,"男"));
        return userList;
    }

    @RequestMapping("/a4")
    public String a4(String name, String pwd){
        String msg = "";
        if(name != null){
            // admin 应该从数据库中查到
            if("admin".equals(name)){
                msg = "登录成功";
            } else {
                msg = "登录失败";
            }
        }
        return msg;
    }
}

```

```

        msg = "OK";
    }else {
        msg = "用户名有误";
    }
}
if(pwd != null) {
    // pwd 应该从数据库中查到
    if("123456".equals(pwd)) {
        msg = "OK";
    }else {
        msg = "密码有误";
    }
}
return msg;
}
}

```

编写 test.xml 实现不刷新网页加载数据

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
    <script src="${pageContext.request.contextPath}/statics/js/jquery.js"></script>

    <script>
        $(function () { /*jquery的id选择器: $("#id") */
            $("#btn").click(function () {
                /* $.post(url, param可省略, success回调函数 */
                $.post("${pageContext.request.contextPath}/a3", function (data) {
                    // console.log(data);
                    var html="";
                    // 遍历"/a3"中response返回的JSON字符串
                    for (let i = 0; i < data.length; i++) {
                        html += "<tr>" +
                            "<td>" + data[i].name + "</td>" +
                            "<td>" + data[i].age + "</td>" +
                            "<td>" + data[i].sex + "</td>" +
                            "</tr>";
                    }
                    $("#content").html(html);
                });
            })
        })
    </script>
</head>
<body>
<input type="button" value="加载数据" id="btn">
<table>
    <tr>
        <td>姓名</td>
        <td>年龄</td>
        <td>性别</td>
    </tr>
</table>
<tbody id="content">
    <%--从后台获取数据--%>
        </tbody>
</table>
</body>
</html>

```

编写实体类User

```

package com.xiang.pojo;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {

    private String name;
    private int age;
    private String sex;

}

```

编写login.xml实现不刷新网页验证用户输入的登录信息是否正确

```

<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
    <script src="${pageContext.request.contextPath}/statics/js/jquery.js"></script>

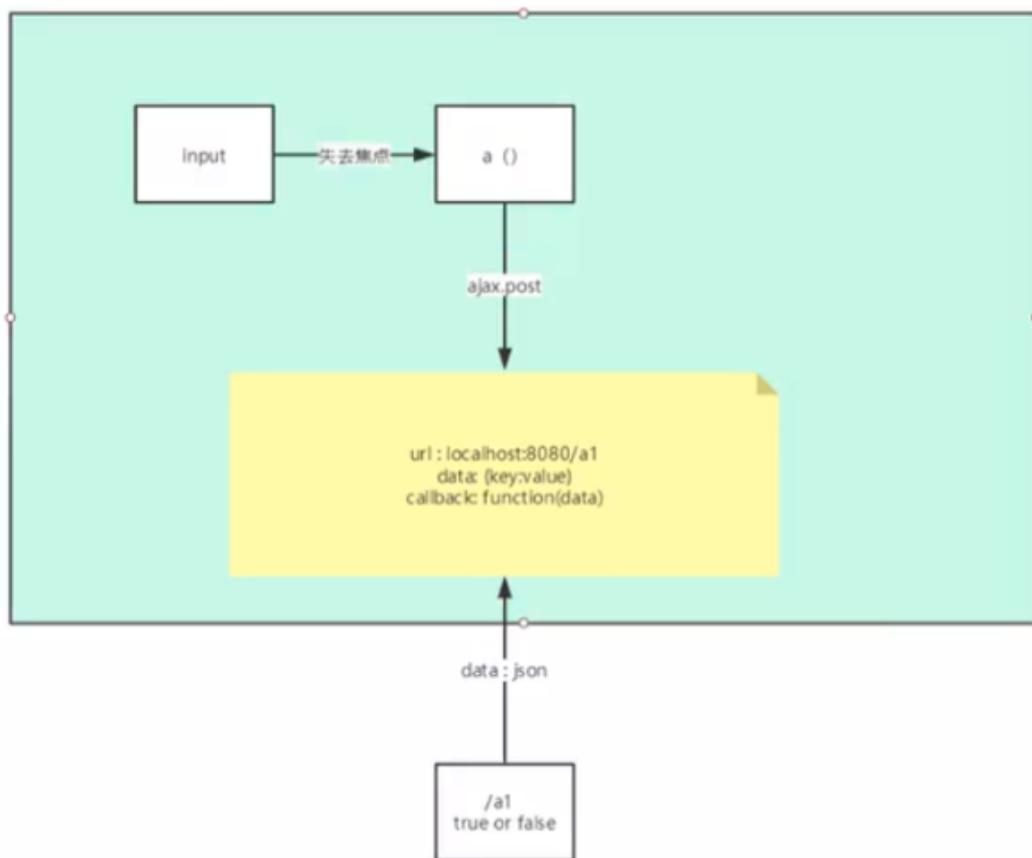
    <script>
        function a1() {
            $.post({
                url:"${pageContext.request.contextPath}/a4",
                data:{'name':$('#name').val()},
                success:function (data) {
                    if(data.toString()=='OK'){
                        $('#userInfo').css("color","green");
                    }else{
                        $('#userInfo').css("color","red");
                    }
                    $('#userInfo').html(data);
                }
            })
        }
        function a2() {
            $.post({
                url:"${pageContext.request.contextPath}/a4",
                data:{'pwd':$('#pwd').val()},
                success:function (data) {
                    if(data.toString()=='OK'){
                        $('#pwdInfo').css("color","green");
                    }else{
                        $('#pwdInfo').css("color","red");
                    }
                    $('#pwdInfo').html(data);
                }
            })
        }
    </script>
</head>
<body>

<p>    用户名: <input type="text" id="name" onblur="a1()">
        <span id="userInfo"></span>
</p>
<p>    密码: <input type="password" id="pwd" onblur="a2()">
        <span id="pwdInfo"></span>
</p>


```

```
</p>  
</body>  
</html>
```

Ajax原理剖析：



SpringMVC拦截器：

概述

SpringMVC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。开发者可以自己定义一些拦截器来实现特定的功能。

过滤器与拦截器的区别：拦截器是AOP思想的具体应用。

过滤器

- servlet规范中的一部分，任何java web工程都可以使用
- 在url-pattern中配置了/*之后，可以对所有要访问的资源进行拦截

拦截器

- 拦截器是SpringMVC框架自己的，只有使用了SpringMVC框架的工程才能使用
- 拦截器只会拦截访问的控制器方法，如果访问的是jsp/html/css/image/js是不会进行拦截的

自定义拦截器

那如何实现拦截器呢？

想要自定义拦截器，必须实现 HandlerInterceptor 接口。

1. 新建一个Module，springmvc-07-Interceptor，添加web支持
2. 配置web.xml 和 springmvc-servlet.xml 文件
3. 编写一个拦截器

SpringMvc中的拦截器：

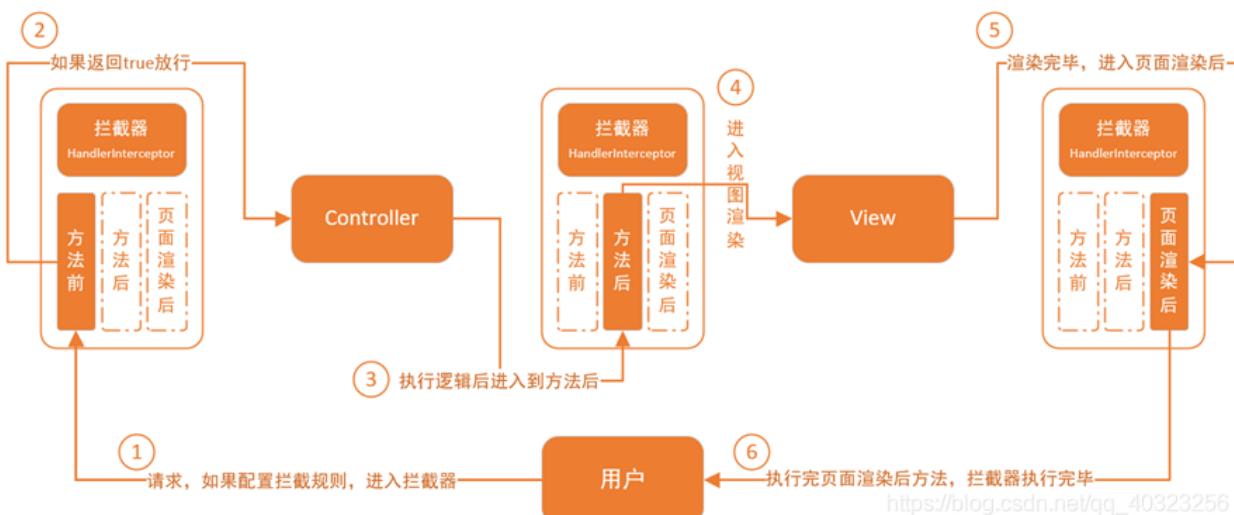
1. SpringMvc拦截器帮我们按照一定规则拦截请求，后根据开发人员自定义的拦截逻辑进行处理；
2. 自定义拦截器需要实现HandlerInterceptor接口；
3. 自定义的拦截器实现类需要在SpringMvc配置文件中配置；
4. 可以配置多个拦截器，配置的顺序会影响到拦截器的执行顺序，配置在前的先执行；
5. HandlerInterceptor有3个方法：

preHandle 预处理：在拦截方法前执行；

postHandle 后处理：在拦截方法后执行；

afterCompletion 渲染后处理：在页面渲染后执行；

6. 拦截器也体现了AOP思想；
7. 拦截器的应用：权限检查，日志记录，性能检测等；



总结的几条拦截器规则：

1. preHandle 预处理：根据拦截器定义的顺序，正向执行。
2. postHandle 后处理：根据拦截器定义的顺序，逆向执行。需要所有的preHandle都返回true时才会调用。
3. afterCompletion 渲染后处理：根据拦截器定义的顺序，逆向执行。preHandle返回true就会调用。

如果定义了MyInterceptor1和MyInterceptor2两个拦截器，则执行顺序为：

1 PreHandle预处理

2 PreHandle预处理

执行请求方法

2 PostHandle后处理

1 PostHandle后处理

2 afterCompletion页面渲染后处理

1 afterCompletion页面渲染后处理

在controller目录下编写LoginController.java

```
package com.xiang.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpSession;

@Controller
@RequestMapping("/user")
public class LoginController {

    @RequestMapping("/main")
    public String main() {
        return "main";
    }

    @RequestMapping("/goLogin")
    public String goLogin() {
        return "login";
    }

    @RequestMapping("/login")
    public String login(HttpSession session, String username, String password, Model model) {
        model.addAttribute("username", username);
        System.out.println("login=>" + username);
        // 把用户的信息存到session中
        session.setAttribute("userLoginInfo", username);
        return "main";
    }

    @RequestMapping("/logout")
    public String logout(HttpSession session) {
        // session.invalidate(); 建议使用下面这种，用户可能还需要访问该服务器，避免
重新创建session
        session.removeAttribute("userLoginInfo");
        return "login";
    }
}
```

```
}
```

在interceptor目录下编写测试拦截器：

```
package com.xiang.interceptor;

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

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

public class MyInterceptor implements HandlerInterceptor {

    @Override // 在拦截方法前执行
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        // return true; 放行，不拦截，执行下一个拦截器
        // return false; 拦截，不执行下一个拦截器和方法
        // return HandlerInterceptor.super.preHandle(request, response, handler);
        System.out.println("=====处理前=====");
        return true;
    }

    // 执行controller下的方法

    @Override // 在拦截方法后执行；可以编写拦截日志，也可以不实现这个方法
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("=====处理后=====");
        // HandlerInterceptor.super.postHandle(request, response, handler,
        modelAndView);
    }

    @Override // 渲染后处理，在页面绚烂后执行
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        System.out.println("=====清理=====");
        // HandlerInterceptor.super.afterCompletion(request, response, handler, ex);
    }
}
```

编写登录拦截器LoginInterceptor.java

```
package com.xiang.interceptor;

import org.springframework.web.servlet.HandlerInterceptor;

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

public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

        HttpSession session = request.getSession();
        // 放行：判断什么情况下登录
        if (session.getAttribute("userLoginInfo") != null) {
            System.out.println("\\"userLoginInfo\\" != null");
            return true;
        }
        // 登录页面也要放行
        if (request.getRequestURI().contains("goLogin")){
```

```

        System.out.println("URI().contains(\"goLogin\")");
        return true;
    }
    if (request.getRequestURI().contains("login")){
        System.out.println("URI().contains(\"login\")");
        return true;
    }

    // 判断什么情况下没有登录
    request.getRequestDispatcher("/WEB-
INF/jsp/login.jsp").forward(request,response);
    return false;
}
}

```

在resources目录编写拦截器配置application.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
    https://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/mvc
    https://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

    <!--静态资源过滤-->
    <mvc:default-servlet-handler/>

    <!--注解驱动-->
    <mvc:annotation-driven/>

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

    <!--使用Jackson解决JSON输出乱码问题-->
    <mvc:annotation-driven>
        <mvc:message-converters register-defaults="true">
            <bean
                class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean
                class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
                <property name="objectMapper">
                    <bean
                        class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
                        <property name="failOnEmptyBeans" value="false" />
                    </bean>
                    </property>
                </bean>
            </mvc:message-converters>
        </mvc:annotation-driven>

        <!--springmvc拦截器配置-->
        <mvc:interceptors>
            <mvc:interceptor>

```

```

<!--"/下面的所有请求都会拦截-->
<mvc:mapping path="/**/login"/> <!--只拦截以/login结尾的请求，不拦截
login.jsp-->
    <bean class="com.xiang.interceptor.MyInterceptor"/> <!--绑定哪个拦截器去
拦截上面的请求-->
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/user/**"/>
        <bean class="com.xiang.interceptor.LoginInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>

</beans>

```

编写前端页面：

在WEB-INF/jsp目录下编写login.jsp和main.jsp:

login.jsp

```

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

<%--WEB-INF目录下的所有页面或者资源，只能通过controller或者servlet进行访问--%>
<h1>登录页面</h1>

<form action="${pageContext.request.contextPath}/user/login" method="post">
    用户名: <input type="text" name="username"/>
    密码: <input type="password" name="password"/>
    <input type="submit" value="提交">
</form>

</body>
</html>

```

main.jsp

```

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

<h1>首页</h1>

<span>欢迎用户 ${username} <a href="${pageContext.request.contextPath}/user/logout">注销</a>
</span>

<p>
    <a href="${pageContext.request.contextPath}/user/logout">注销</a>
</p>

</body>
</html>

```

Index.jsp

```

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

```

```
</head>
<body>

<h1><a href="${pageContext.request.contextPath}/user/goLogin">登录</a> </h1>
<h1><a href="${pageContext.request.contextPath}/user/main">首页</a> </h1>

</body>
</html>
```

SpringMVC：文件上传和下载

enctype编码类型：

1. application/x-www-form-urlencoded：默认编码方式，只处理表单域中的value属性值，将表单域中的值处理成URL编码。
2. multipart/form-data：这种编码方式会以二进制的方式处理表单数据，会把文件域指定文件的内容也封装到请求参数中，不会对字符编码
3. text/plain：纯文本的传输。除了把空格转换为“+”号外，其他字符都不做编码处理，适用于直接通过表单发送邮件。

准备工作

文件上传是项目开发中最常见的功能之一，springMVC 可以很好的支持文件上传，但是 SpringMVC上下文中默认没有装配MultipartResolver，因此默认情况下其不能处理文件上传工作。如果想使用Spring的文件上传功能，则需要在上下文中配置MultipartResolver。

前端表单要求：为了能上传文件，必须将表单的method设置为POST，并将enctype设置为 multipart/form-data。只有在这样的情况下，浏览器才会把用户选择的文件以二进制数据发送给服务器；

对表单中的 enctype 属性做个详细的说明：

- application/x-www-form-urlencoded：默认方式，只处理表单域中的 value 属性值，采用这种编码方式的表单会将表单域中的值处理成 URL 编码方式。
- multipart/form-data：这种编码方式会以二进制流的方式来处理表单数据，这种编码方式会把文件域指定文件的内容也封装到请求参数中，不会对字符编码。
- text/plain：除了把空格转换为“+”号外，其他字符都不做编码处理，这种方式适用直接通过表单发送邮件。

```
<form action="" enctype="multipart/form-data" method="post">
    <input type="file" name="file"/>
    <input type="submit">
</form>
```

一旦设置了`enctype`为`multipart/form-data`，浏览器即会采用二进制流的方式来处理表单数据，而对于文件上传的处理则涉及在服务器端解析原始的HTTP响应。在2003年，Apache Software Foundation发布了开源的Commons FileUpload组件，其很快成为Servlet/JSP程序员上传文件的最佳选择。

- Servlet3.0规范已经提供方法来处理文件上传，但这种上传需要在Servlet中完成。
- 而Spring MVC则提供了更简单的封装。
- Spring MVC为文件上传提供了直接的支持，这种支持是用即插即用的MultipartResolver实现的。
- Spring MVC使用Apache Commons FileUpload技术实现了一个MultipartResolver实现类：`CommonsMultipartResolver`。因此，==SpringMVC的文件上传还需要依赖Apache Commons FileUpload的组件==。

文件上传

一、导入文件上传的jar包，`commons-fileupload`，Maven会自动帮我们导入他的依赖包`commons-io`包；

```
<!--文件上传-->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.3</version>
</dependency>
<!--servlet-api导入高版本的-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
</dependency>
```

二、配置bean：multipartResolver

【注意！！！这个bean的id必须为：`multipartResolver`，否则上传文件会报400的错误！在这里栽过坑，教训！】

```
<!--文件上传配置-->
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 请求的编码格式，必须和JSP的pageEncoding属性一致，以便正确读取表单的内容，默认为ISO-8859-1 -->
    <property name="defaultEncoding" value="utf-8"/>
    <!-- 上传文件大小上限，单位为字节（10485760=10M） -->
    <property name="maxUploadSize" value="10485760"/>
    <property name="maxInMemorySize" value="40960"/>
</bean>
```

CommonsMultipartFile 的常用方法：

- `String getOriginalFilename()`: 获取上传文件的原名
- `InputStream getInputStream()`: 获取文件流
- `void transferTo(File dest)`: 将上传文件保存到一个目录文件中

前端编写，`index.xml`:

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

```

<form action="${pageContext.request.contextPath}/upload" enctype="multipart/form-data" method="post">
    <input type="file" name="file"/>
    <input type="submit" value="上传">
</form>

</body>
</html>

```

编写resources目录下的配置文件application.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
                           https://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/mvc
                           https://www.springframework.org/schema/mvc/spring-mvc.xsd">

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

    <!--静态资源过滤-->
    <mvc:default-servlet-handler/>

    <!--注解驱动-->
    <mvc:annotation-driven/>

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

    <!--使用Jackson解决JSON输出乱码问题-->
    <mvc:annotation-driven>
        <mvc:message-converters register-defaults="true">
            <bean
                class="org.springframework.http.converter.StringHttpMessageConverter">
                <constructor-arg value="UTF-8"/>
            </bean>
            <bean
                class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
                <property name="objectMapper">
                    <bean
                        class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
                        <property name="failOnEmptyBeans" value="false"/>
                    </bean>
                </property>
            </bean>
        </mvc:message-converters>
    </mvc:annotation-driven>

    <!--文件上传配置-->
    <bean id="multipartResolver"
          class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!--请求的编码格式，必须和JSP的pageEncoding属性一致，以便正确读取表单的内容，默认为ISO-8859-1-->
        <property name="defaultEncoding" value="utf08"/>
    
```

```

<!--上传文件大小上限，单位为字节（10485760=10M）-->
<property name="maxUploadSize" value="10485760"/>
<property name="maxInMemorySize" value="40960"/>
</bean>

</beans>

```

在controller目录下编写FileController.java:

```

package com.xiang.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.commons.CommonsMultipartFile;

import javax.servlet.http.HttpServletRequest;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

@RestController
public class FileController {

    // @RequestParam("file") 将name=file的控件得到的文件封装成CommonsMultipartFile对象
    // 若为批量上传，将CommonsMultipartFile对象初始化为数组即可
    @RequestMapping("/upload")
    public String upload(@RequestParam("file") CommonsMultipartFile file,
                         HttpServletRequest request) throws IOException {

        // 获取文件名: file.getOriginalFilename()
        String uploadFileName = file.getOriginalFilename();

        // 如果文件名为空，直接回到首页
        if(".".equals(uploadFileName)) {
            return "redirect:/index.jsp";
        }
        System.out.println("上传文件的文件名为: " + uploadFileName);

        // 创建上传文件的保存路径
        String filePath = request.getServletContext().getRealPath("/upload");
        File uploadPath = new File(filePath);
        if (!uploadPath.exists()) {
            uploadPath.mkdirs(); // 目录不存在，创建目录
        }
        System.out.println("上传文件的保存路径为: " + uploadPath);

        InputStream inputStream = file.getInputStream(); // 文件输入流
        FileOutputStream fileOutputStream = new FileOutputStream(new
File(uploadPath, uploadFileName)); // 文件输出流

        //读取写出
        int len = 0;
        byte[] buffer = new byte[1024];
        while ((len = inputStream.read(buffer)) > 0) {
            fileOutputStream.write(buffer, 0, len);
            fileOutputStream.flush();
        }
        fileOutputStream.close();
        inputStream.close();

        return "redirect:/index.jsp";
    }
}

```

```

@RequestMapping("/upload2") // file.transferTo(new File())来保存上传的文件
public String upload2(@RequestParam("file") CommonsMultipartFile file,
HttpServletRequest request) throws IOException {
    // 创建上传文件的保存路径
    String path = request.getServletContext().getRealPath("/upload"); // 获取“.../项目目录/upload”的绝对路径
    File realPath = new File(path);
    if (!realPath.exists()) {
        realPath.mkdirs();
    }
    // 上传文件保存路径
    System.out.println("上传文件的保存路径为: " + realPath);

    // 通过CommonsMultipartFile提供的方法直接写文件
    file.transferTo(new File(realPath + "/" + file.getOriginalFilename()));

    return "redirect:/index.jsp";
}

```

采用file.Transto 来保存上传的文件

1. 编写Controller

```

/*
 * 采用file.Transto 来保存上传的文件
 */
@RequestMapping("/upload2")
public String fileUpload2(@RequestParam("file") CommonsMultipartFile file, HttpServletRequest request) {
    //上传路径保存设置
    String path = request.getServletContext().getRealPath("/upload");
    File realPath = new File(path);
    if (!realPath.exists()){
        realPath.mkdir();
    }
    //上传文件地址
    System.out.println("上传文件保存地址: "+realPath);

    //通过CommonsMultipartFile的方法直接写文件（注意这个时候）
    file.transferTo(new File(realPath + "/" + file.getOriginalFilename()));

    return "redirect:/index.jsp";
}

```

2. 前端表单提交地址修改

3. 访问提交测试，OK！

文件下载:

```

@RequestMapping(value = "/download")
public String download(HttpServletResponse response, HttpServletRequest request)
throws IOException {
    // 要下载的图片地址
    String path = request.getServletContext().getRealPath("/upload");
    String fileName = "内存图.png";

    // 1. 设置response响应头

```

```
response.reset(); // 设置页面不缓存, 清空buffer
response.setCharacterEncoding("utf-8"); // 字符编码
response.setContentType("multipart/form-data"); // 二进制流的方式传输数据

// 设置响应头
response.setHeader("Content-Disposition", "attachment;fileName=" +
URLEncoder.encode(fileName, "UTF-8"));

File file = new File(path, fileName);
// 2. 读取文件--输入流
InputStream inputStream = new FileInputStream(file);
// 3. 写出文件--输出流
OutputStream outputStream = response.getOutputStream();

byte[] buffer = new byte[1024];
int len = 0;
// 4. 执行写出操作
while ((len=inputStream.read(buffer)) > 0) {
    outputStream.write(buffer, 0, len);
    outputStream.flush();
}
outputStream.close();
inputStream.close();
return "下载成功";
}
```