SpringMVC

SSM：Mybatis+Spring+SpringMVC **MVC三层架构**

1.什么是MVC

M(Model),V(View),C(controller) DAO,Service,Jsp/Html,Servlet Jsp+javaBean+Servlet

2.复习Servlet

1.先配置实体类继承HttpServlet

```
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String method = req.getParameter( s: "method");
        if(method.equals("add")){
            req.getSession().setAttribute( s: "msg", o: "这个是add方法");
        }else if(method.equals("delete")){
            req.getSession().setAttribute( s: "msg", o: "这个是delete方法");
        }
        req.getRequestDispatcher( s: "index.jsp").forward(req,resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        doGet(req, resp);
    }
}
```

2.在WEB-INF的xml文件中配置相关的servlet的注册

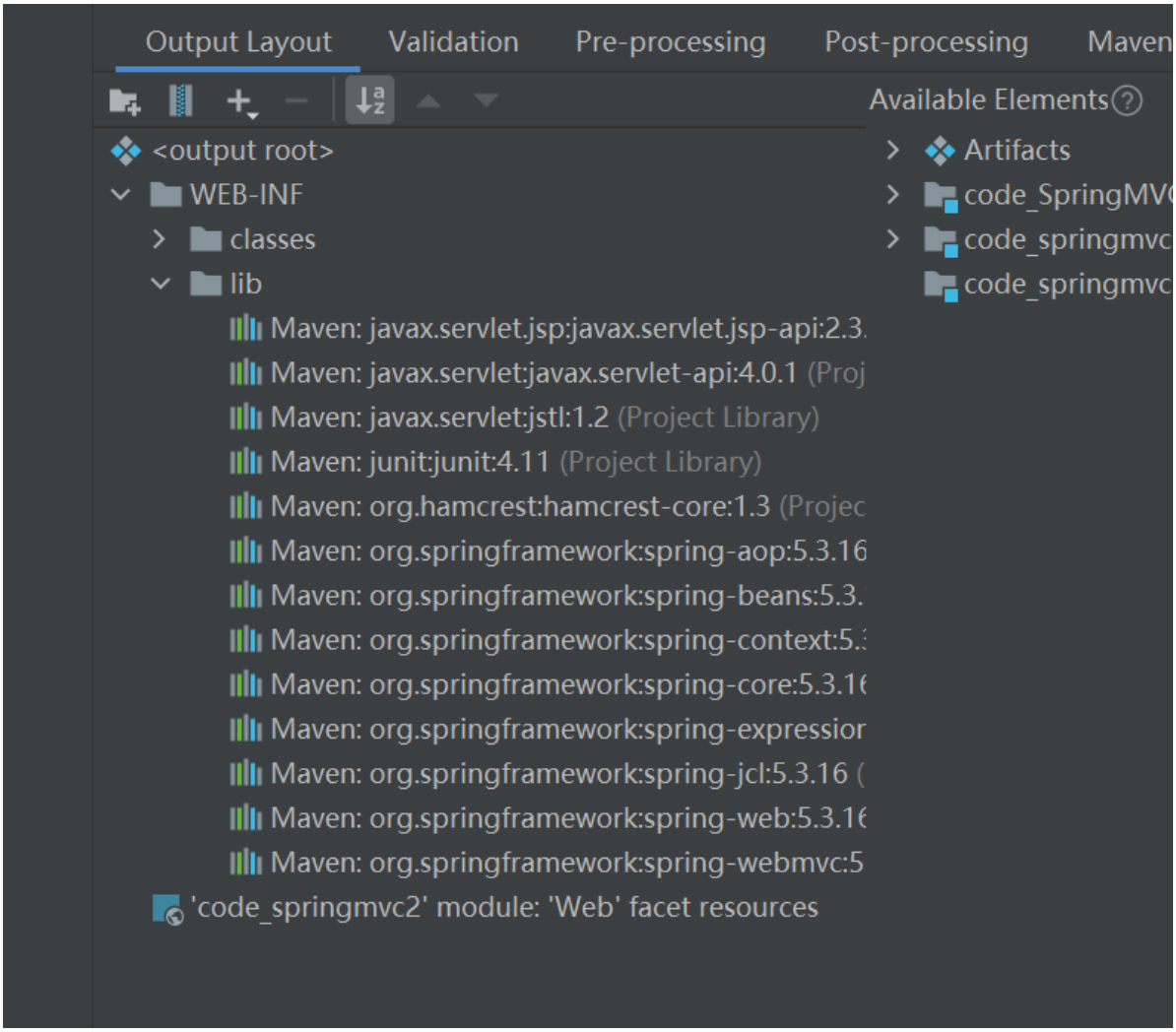
在下面的这张图中有个新的标签这个标签指定的是开始界面的界面是什么jsp

```
<servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>com.feng.controller.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
<welcome-file-list>
    <welcome-file>./jsp/hello.jsp</welcome-file>
</welcome-file-list>
```

3.HelloSpringMVC important 底层原理

提前注意，之前在写jsp的时候就出现的问题

在本文件夹中的project structure的war下，可能当前需要的依赖包没有导入，这个点需要检查



1.先配置相应的Servlet.xml文件

下面的这个包里面有内置的Servlet，所以可以直接使用

```
org.springframework.web.servlet.DispatcherServlet
```

在下面的中配置的变量是相应bean的注册路径，还有名字，这其中的名字不要改动，这个是固定的

这个是声明启动的优先级

当值为0或者大于0时，表示容器在应用启动时就加载这个servlet；

当是一个负数时或者没有指定时，则指示容器在该servlet被选择时才加载。

正数的值越小，启动该servlet的优先级越高。

下面的这个代表的是匹配所有的请求

/ 代表的是匹配所有的请求(不包括.jsp)　**important**

/* 代表的是匹配所有的请求(包括.jsp)

这上面的差别就是用户是从.jsp文件目录直接访问该网页呢，还是靠虚拟路径来实现页面的跳转

```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:spring-servlet.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>
</web-app>
```

2.配置相应的Spring-Servlet.xml文件

这下面的三个class是十分重要的

1.org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping

这个代表的是给文件项目里注册了处理映射器

2.org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter

这个代表的是给文件项目里注册了处理适配器

3.org.springframework.web.servlet.view.InternalResourceViewResolver

这个代表的是给文件项目里注册了试图解析器，这其中的属性，是和下面的网页访问对应的jsp文件进行拼接的时候用的

以上的三个步骤是固定的，无需修改

下面要修改的是controller的注册

最下面的那个代表的是我注册这个id的网址，类实现的是class的值

注意

第一个property中的属性值

/jsp/ 和 **./jsp/**的区别：

第一个的根基目录就是Web的目录

第二个是相对于当前网页目录的前一个目录下的jsp文件

所以第一个更加好，索引也更加的标准

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

        <bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
        <bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
        <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
               <property name="prefix" value="./jsp/"/>
               <property name="suffix" value=".jsp"/>
        </bean>
        <bean id="/hello" class="com.feng.controller.HelloController"/>
</beans>
```

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

3.配置对应的controller的文件

这个是实现了Controller接口的方法，这个ModelView是模型与视图

下面的方法是将Object对象封装起来放在了页面中

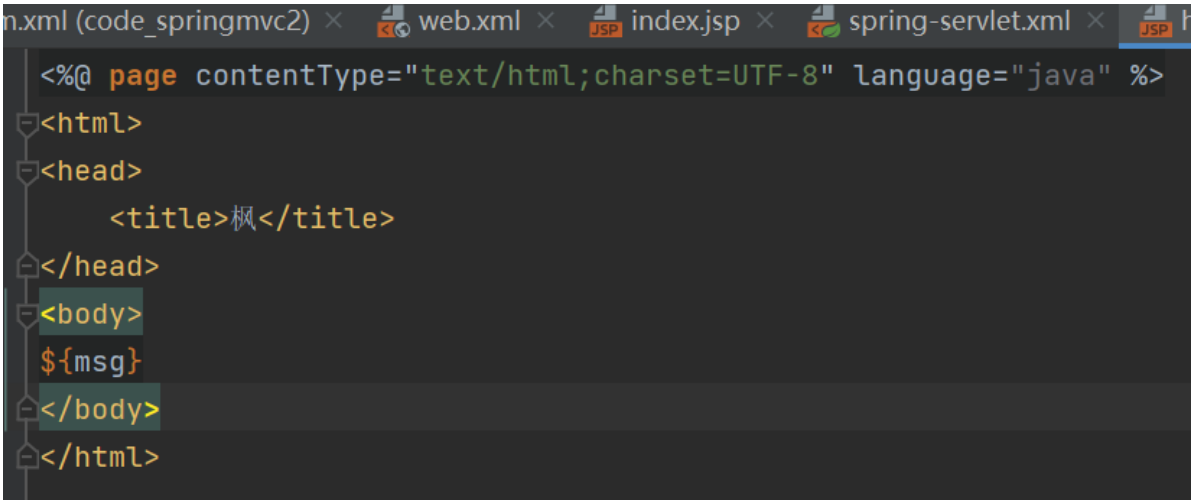
setViewName是和上面的Servlet.xml中的中的中的属性进行拼接，从而实现上面的Spring中注册的界面的访问

在这个文件中可以实现业务的访问和工作

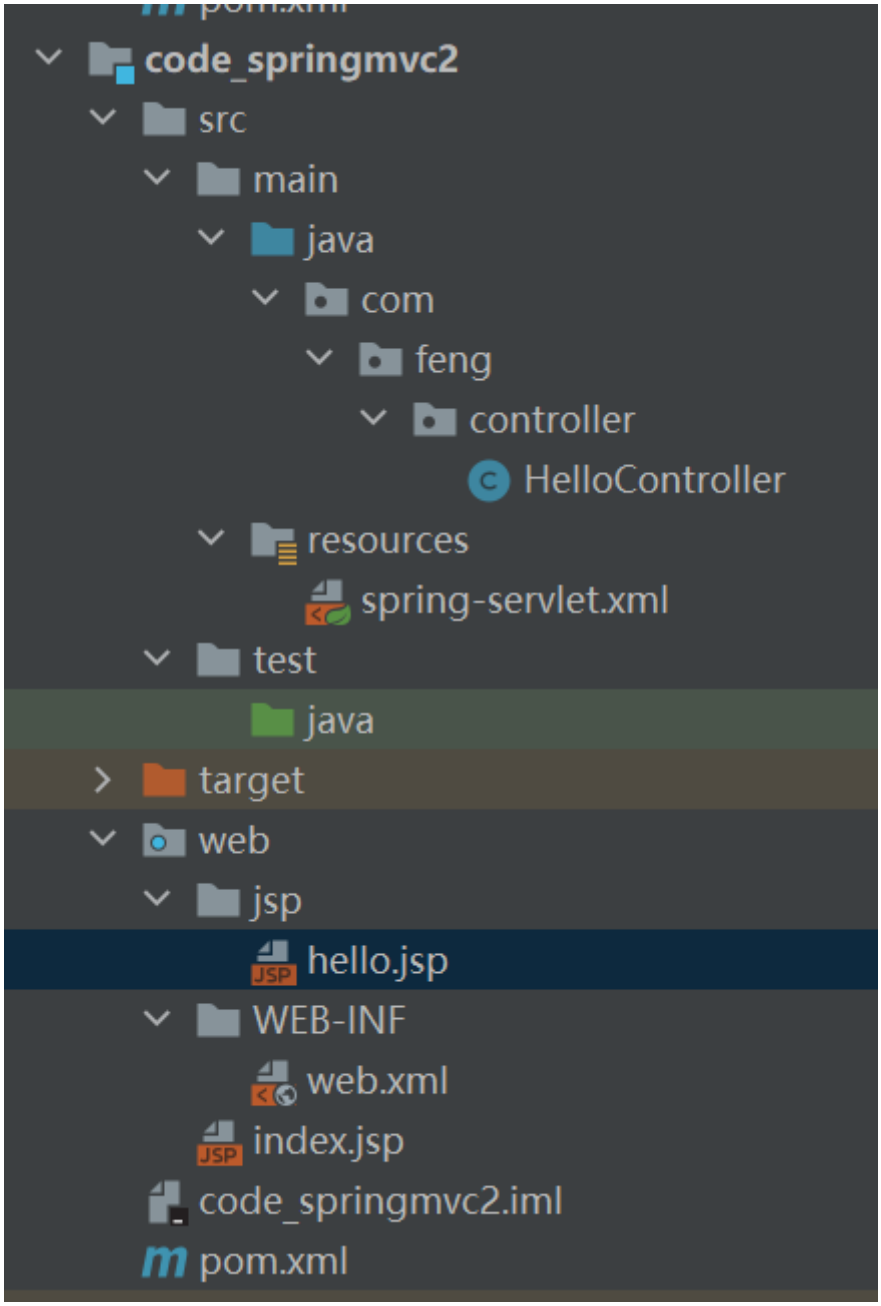
```java
package com.feng.controller;

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

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

public class HelloController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject( attributeName: "msg", attributeValue: "HelloSpringMVC");
        modelAndView.setViewName("hello");
        return modelAndView;
    }
}
```

4.配置对应的jsp文件

在这个文件就实现了msg的读取



5.文件的配置



4.注解实现HelloSpringMVC

1.配置mvc和context在Servlet.xml的引入

```
xmlns:context="http://www.springframework.org/schema/context"
xmlns:mvc="http://www.springframework.org/schema/mvc"

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

在这些都引入的情况下，就可以进行下面的步骤了

先扫描注解需要生效的包，也就是< context:component-scan >的作用

```
<mvc:default-servlet-handler/>
<mvc:annotation-driven/>
```

上面的这些标签分别代表

- 让springmvc不处理静态资源
- 这个功能十分强大，他将上面需要配置的Mapper和Adapter都配置完了，所以下面直接接入Resolver

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

        <context:component-scan base-package="com.feng.controller"/>
        <mvc:default-servlet-handler/>
        <mvc:annotation-driven/>
        <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/jsp/" />
                <property name="suffix" value=".jsp"/>
        </bean>

</beans>
```

2.实体类的配置

@Controller这个声明这个类是一个Controller类，代表是网页控制类

@RequestMapping这个声明这是一个网页的处理方法，默认是post和get方法都能访问

这个方法返回的String字符串就是和上面的需要拼接的字符串的网页jsp文件

```java
package com.feng.controller;

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

@Controller
public class HelloController2 {
    @RequestMapping("/hello")
    public String Hello(Model model){
        model.addAttribute( attributeName: "msg", attributeValue: "HelloSpring");
        return "hello";
    }
}
```

注意

在这种方法下可以实现jsp文件的重复利用性

在这种多次网页切换的情况下，就只需要一个jsp文件就可以实现重复使用性

```java
@Controller
public class HelloController2 {
    @RequestMapping("/hello")
    public String Hello(Model model){
        model.addAttribute( attributeName: "msg", attributeValue: "HelloSpring");
        return "hello";
    }
    @RequestMapping("/test")
    public String Hello2(Model model){
        model.addAttribute( attributeName: "msg", attributeValue: "HelloSpringMVC");
        return "hello";
    }
}
```

在最上面加个**@RequestMapping**可以实现多级目录的效果

```java
@Controller
@RequestMapping("/father")
public class HelloController2 {
    @RequestMapping("/hello")
    public String Hello(Model model){
        model.addAttribute( attributeName: "msg", attributeValue: "HelloSpring");
        return "hello";
    }
    @RequestMapping("/test")
    public String Hello2(Model model){
        model.addAttribute( attributeName: "msg", attributeValue: "HelloSpringMVC");
        return "hello";
    }
}
```

5.RestFul风格

1.基本原理

- 系统上的一切对象都要抽象为资源
- 每个资源对应唯一的资源标识
- 对资源的操作不能改变资源标识本身
- 所有的操作都是无状态的

可以通过不同的请求方式来实现不同的效果！

例子：百度百科就是典型的RestFul风格

2.传统网页的配置

下面需要访问该网页并实现该网页功能的网址是如下图，就像是get方法一样传入参数，这个网页默认的提取方式为get的方式

localhost:8080/feng/add?a=1&b=2

```java
package com.feng.controller;

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

@Controller
public class HelloController3 {
    @RequestMapping("/add")
    public String test1(int a, int b, Model model){
        int res = a+b;
        model.addAttribute( attributeName: "msg",res);
        return "hello";
    }
}
```

2.RestFul的配置

在如下的这张图中直接将a,b的变量变为该网页要访问的路径之一，从而实现该网页的功能，访问该网站的网址为

localhost:8080/feng/add/1/2

```java
@Controller
public class HelloController3 {
    @RequestMapping("/add/{a}/{b}")
    public String test1(@PathVariable int a,@PathVariable int b, Model mod
        int res = a+b;
        model.addAttribute( attributeName: "msg",res);
        return "hello";
    }
}
```

通过下面的方式可以实现相同的网址，但是结果不同，也就是因为访问的方式不同，结果也不同

```java
@Controller
public class HelloController3 {
    @RequestMapping(value = "/add/{a}/{b}",method = RequestMethod.GET)
    public String test1(@PathVariable int a,@PathVariable int b, Model model){
        int res = a+b;
        model.addAttribute( attributeName: "msg",res);
        return "hello";
    }
}
```

HTML中的FORM是传统的网页路径的设置

下面的方法简化上述的@RequestMapping的设置方法的复杂

相应的POST有对应的@PostMapping的注解，其他的也有相应的注解

```java
@Controller
public class HelloController3 {
    @GetMapping("/add/{a}/{b}")
    public String test1(@PathVariable int a,@PathVariable int b, Model model){
        int res = a+b;
        model.addAttribute( attributeName: "msg", attributeValue: "方式一："+res);
        return "hello";
    }

}
```

4.RestFul的好处

还有一点安全

结果：3

思考：使用路径变量的好处？

- 使路径变得更加简洁；
- 获得参数更加方便，框架会自动进行类型转换。
- 通过路径变量的类型可以约束访问参数，如果类型不一样，则访问不到对应的请求方法，如这里访问是的路径是/commit/1/a，则路径与方法不匹配，而不会是参数转换失败。

6.重定向和转发

1.配置视图解析器

ModelAndView 的 setViewName 的方法来实现重定向

通过prefic和suffix的字符串拼接来实现相应页面的跳转

2.通过HttpServlet

上面的那个方法是通过HttpServletResponse的sendRedirect的方法，着个方法不会将当前的response和request中存在的东西传递下去，所以这里还新加了个HttpSession来存在数据在浏览器中，这要就可以达到重定向和传递数据的功能，当然还可以通过ModelAndView来进行存储

下面的那个方法是通过HttpServletRequest的getRequestDispatcher的方法来达到转发的功能，在加上forward的功能来转发数据

注意 第一种方法会改变当前页面的网页，第二种不会改变

```java
@Controller
@RequestMapping("/test")
public class HelloController4 {
    @RequestMapping("/t1")
    public void test1(HttpServletResponse resp, HttpServletRequest req, HttpSession httpSession) throws IOException {
        System.out.println("进入了这个方法");
        httpSession.setAttribute( s: "msg", o: "这个是test1");
        resp.sendRedirect( s: "../index.jsp");

    }
    @RequestMapping("/t2")
    public void test2(HttpServletRequest req,HttpServletResponse resp) throws ServletException, IOException {
        req.setAttribute( s: "msg", o: "这个是test2");
        req.getRequestDispatcher( s: "/index.jsp").forward(req,resp);
    }
}
```

3.通过Return语句

测试前，先将视图解析器去掉

这个方法默认是转发，这个方法就是在return语句中加上forward或者redirect的标签

roller2.java × HelloController3.java × HelloController4.java × HelloController5

```java
package com.feng.controller;

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

@Controller
@RequestMapping("/test1")
public class HelloController5 {
    @RequestMapping("/t1")
    public String test1(){
        return "/index.jsp";
    }
    @RequestMapping("/t2")
    public String test2(){
        return "forward:/index.jsp";
    }
    @RequestMapping("/t3")
    public String test3(){
        return "redirect:/index.jsp";
    }

}
```

4.存在视图解析器

直接通过return相关语句来重定向

```
    }
    @RequestMapping("/test")
    public String Hello2(Model model){
        model.addAttribute( attributeName: "msg", attributeValue: "HelloSpringMVC");
        return "hello";
    }
}
```

7.接受请求参数及数据回显

1.接受请求参数

1.处理提交参数名相同的数据

访问方式为get的方法，访问地址为　localhost:7878/feng/test2/t1?name=feng

```
@Controller
@RequestMapping("/test2")
public class HelloController6 {
    @RequestMapping("/t1")
    public String test1(String name){
        System.out.println(name);
        return "/hello1";
    }
}
```

2.处理提交参数名不同的数据

通过给参数设置@RequestParam的属性设置域名访问的参数，通过这个方法参数就必须被设置

访问地址为　localhost:7878/feng/test2/t2?username=feng

```
@RequestMapping("/t2")
public String test2(@RequestParam("username")String name){
    System.out.println(name);
    return "/hello1";
}
```

3.传入参数是一个实体类

直接就是把实体类中的属性值设置一下就可以将参数传入了，这些参数不是必要的

访问地址为 localhost:7878/feng/test2/t3?id=1&name=feng&age=2

```
@RequestMapping("/t3")
public String test3(User user){
    System.out.println(user);
    return "/hello";
}
```

2.数据显示在前端上

0.前端页面

通过${msg}来访问该属性

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

1.通过ModelAndView

在视图上添加属性，之后在前端访问

```
public class HelloController implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject( attributeName: "msg", attributeValue: "HelloSpringMVC");
        modelAndView.setViewName("hello");
        return modelAndView;
    }
}
```

2.通过ModelMap

```
@RequestMapping("/test")
public String Hello2(ModelMap model){
    model.addAttribute( attributeName: "msg", attributeValue: "HelloSpringMVC");
    return "hello";
}
```

3.通过Model 用的最多

```
@RequestMapping("/hello")
public String Hello(Model model){
    model.addAttribute( attributeName: "msg", attributeValue: "HelloSpring");
    return "hello";
}
```

8.乱码问题

插曲

注意：form表中的action=""的路径一定要设置为相对路径"LoginServlet"，绝对路径"/LoginServlet"是找不到的。

这里直接使用了Spring自带的Filter类，这个是固定的语句

注意url-pattern的 / 和 /* 的区别，前面有讲

```
<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>
```

下面是大神的过滤器的实现，这个是通用过滤器，注意要在web.xml中注册filter

```
package com.kuang.filter;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Map;

/**
* 解决get和post请求 全部乱码的过滤器
*/
public class GenericEncodingFilter implements Filter {

    @Override
    public void destroy() {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
```

```
        //处理response的字符编码
        HttpServletResponse myResponse=(HttpServletResponse) response;
        myResponse.setContentType("text/html;charset=UTF-8");

        // 转型为与协议相关对象
        HttpServletRequest httpServletRequest = (HttpServletRequest) request;
        // 对request包装增强
        HttpServletRequest myrequest = new MyRequest(httpServletRequest);
        chain.doFilter(myrequest, response);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

}

//自定义request对象，HttpServletRequest的包装类
class MyRequest extends HttpServletRequestWrapper {

    private HttpServletRequest request;
    //是否编码的标记
    private boolean hasEncode;
    //定义一个可以传入HttpServletRequest对象的构造函数，以便对其进行装饰
    public MyRequest(HttpServletRequest request) {
        super(request);// super必须写
        this.request = request;
    }

    // 对需要增强方法 进行覆盖
    @Override
    public Map getParameterMap() {
        // 先获得请求方式
        String method = request.getMethod();
        if (method.equalsIgnoreCase("post")) {
            // post请求
            try {
                // 处理post乱码
                request.setCharacterEncoding("utf-8");
                return request.getParameterMap();
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        } else if (method.equalsIgnoreCase("get")) {
            // get请求
            Map<String, String[]> parameterMap = request.getParameterMap();
            if (!hasEncode) { // 确保get手动编码逻辑只运行一次
                for (String parameterName : parameterMap.keySet()) {
                    String[] values = parameterMap.get(parameterName);
                    if (values != null) {
                        for (int i = 0; i < values.length; i++) {
                            try {
                                // 处理get乱码
                                values[i] = new String(values[i]
                                                       .getBytes("ISO-8859-1"),
"utf-8");
                            } catch (UnsupportedEncodingException e) {
                                e.printStackTrace();
```

```
                            }
                        }
                    }
                }
                hasEncode = true;
            }
            return parameterMap;
        }
        return super.getParameterMap();
    }

    //取一个值
    @Override
    public String getParameter(String name) {
        Map<String, String[]> parameterMap = getParameterMap();
        String[] values = parameterMap.get(name);
        if (values == null) {
            return null;
        }
        return values[0]; // 取回参数的第一个值
    }

    //取所有值
    @Override
    public String[] getParameterValues(String name) {
        Map<String, String[]> parameterMap = getParameterMap();
        String[] values = parameterMap.get(name);
        return values;
    }
}
```

9.JSON important

1.json的格式

json是一种键值对的行书存在的，键名用双引号""来包裹，使用冒号：来进行分隔，然后紧接着的就是值，这个本质就是一个字符串

实例

```
'{"a":"hello","b":"SpringMVC"}'
```

javaScript对象，这个本质就是一个对象

实例

```
{a:"hello",b:"SpringMVC"}
```

其中有两个方法：

JSON.stringify() 这个方法是将js对象转化为json字符串

JSON.parse() 这个方法是将json字符串转化为js对象

测试程序如图：

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>枫</title>
</head>
<body>
<script type="text/javascript">
    var user = {
        name:"秦疆",
        age:3,
        sex:"男"
    };
    var str = JSON.stringify(user);
    console.log(str);
    var user2 = JSON.parse(str);
    console.log(user2);
</script>
</body>
</html>
```

结果如下图：第一个是JSON字符串，第二个是javaScript对象

```
{"name":"秦疆","age":3,"sex":"男"}
▼ Object
    age: 3
    name: "秦疆"
    sex: "男"
  ▶ [[Prototype]]: Object
```

2.准备工作，引入jackSON

下面的是jackSon的依赖包，实际的效果和导入的maven依赖包有关，直接就下载下面的maven依赖的包

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
```

```
    <artifactId>jackson-databind</artifactId>
    <version>2.9.8</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.8</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.9.8</version>
</dependency>
```

3.Spring-servlet.xml和web.xml的配置

这一部分的配置和之前配置的是一样的，但是spring-servlet.xml中加入了处理json的乱码问题

下面的方法一：

```
<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.MappingJackson2HttpMessageConvert
er">
            <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>
```

方法二：

```
@RequestMapping(value = "/t4",produces = "application/json;charset=utf-8")
```

以上的两种方法都可以有效的解决乱码问题

4.测试类的配置

这里分别给出了两种方法

新的注解

@RestController

@RestController是@Controller和@ResponseBody的结合体，意味着这个标签可以实现两种共功能，这个是返回的是json对象，需要jackson的支持

```java
@RestController
@RequestMapping("/test1")
public class HelloController1 {
    @RequestMapping("/t1")
    public String test1() throws JsonProcessingException {
        ObjectMapper objectMapper = new ObjectMapper();
        User feng = new User( id: 1,  name: "feng",  age: 3);
        String s = objectMapper.writeValueAsString(feng);
        return s;
    }
```

@ReponseBody

使用@ReponseBody时，该方法不会走入视图解析器，所以return语句不会被拼接，就可以直接显示到页面上，这个也是返回json对象的，需要jackson的支持

```java
@RequestMapping("/t2")
@ResponseBody
public boolean test2() { return true; }
@RequestMapping("/t3")
public User test3(){
    User feng = new User( id: 1,  name: "feng",  age: 4);
    return feng;
}
```

方式一：

直接通过return语句来返回值，这里要控制返回值的类型，这种方法会直接调用jackson的json转换的方法

```java
@RequestMapping("/t2")
@ResponseBody
public boolean test2() { return true; }
```

```java
@RequestMapping("/t3")
public User test3(){
    User feng = new User( id: 1,  name: "feng",  age: 4);
    return feng;
}
```

方式二： 首推

通过ObjectMapper的映射器来将对象转化为json对象

```
new ObjectMapper().writeValueAsString(java对象);      将java对象转换为json对象
new ObjectMapper().readValue(json字符串,java对象.class);    json转java对象
```

json转类对象的参数为**java对象.class**

json转集合对象的参数为**new TypeReference<List<java对象类>>(){}**

这个**TypeReference**是jackson中的类，别导包错了

```
@RequestMapping("/t1")
public String test1() throws IOException {
    ObjectMapper objectMapper = new ObjectMapper();
    User feng = new User( id: 1,  name: "feng",  age: 3);
    String s = objectMapper.writeValueAsString(feng);
    User user = objectMapper.readValue(s, User.class);
    System.out.println(user);
    return s;
}
```

```
@RequestMapping(value = "/t4",produces = "application/json;charset=utf-8")
public String test4() throws IOException {
    ObjectMapper objectMapper = new ObjectMapper();
    User feng = new User( id: 1,  name: "枫",  age: 4);
    User feng1 = new User( id: 2,  name: "feng",  age: 5);
    ArrayList<User> users = new ArrayList<>();
    users.add(feng);
    users.add(feng1);
    String s = objectMapper.writeValueAsString(users);
    List<User> ans = objectMapper.readValue(s,new TypeReference<List<User>>(){});
    for (User an : ans) {
        System.out.println(an);
    }
    return s;
}
```

这个方法可以设置为工具类，参数为Object

```
@RequestMapping("/t1")
public String test1() throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    User feng = new User( id: 1,  name: "feng",  age: 3);
    String s = objectMapper.writeValueAsString(feng);
    return s;
}
```

```java
@RequestMapping(value = "/t4",produces = "application/json;charset=utf-8")
public String test4() throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    User feng = new User( id: 1,  name: "枫",  age: 4);
    User feng1 = new User( id: 2,  name: "feng",  age: 5);
    ArrayList<User> users = new ArrayList<>();
    users.add(feng);
    users.add(feng1);
    String s = objectMapper.writeValueAsString(users);
    return s;
}
```

fastjson

这个是阿里巴巴开发的json转换的依赖

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

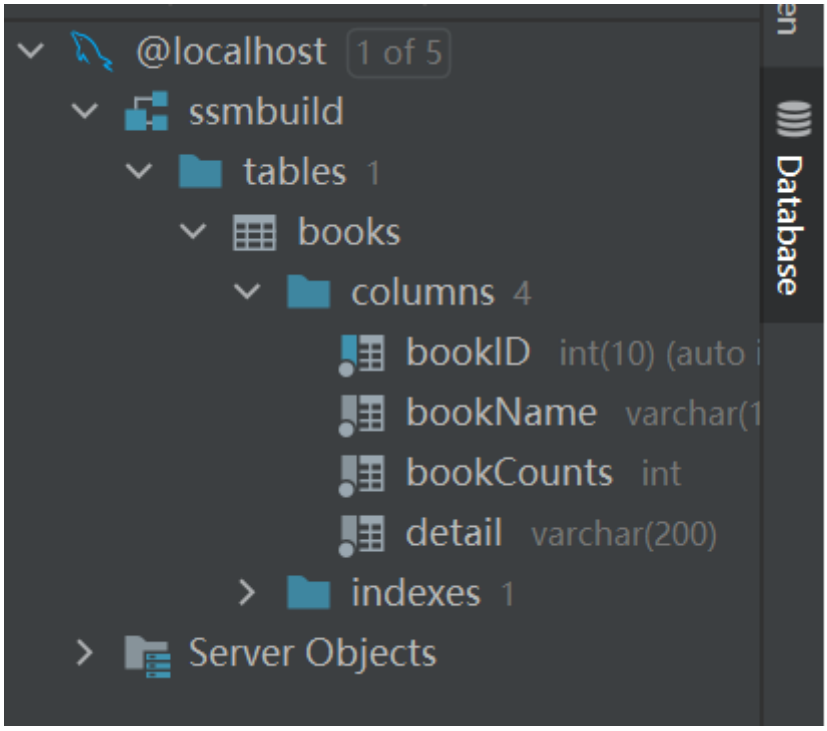
```java
@Controller
@RequestMapping("/test2")
public class HelloController2 {
    @RequestMapping("/t2")
    @ResponseBody
    public String test1(){
        ArrayList<User> users = new ArrayList<>();
        User feng3 = new User( id: 1,  name: "feng",  age: 3);
        User feng2 = new User( id: 2,  name: "feng",  age: 3);
        User feng1 = new User( id: 3,  name: "feng",  age: 3);
        User feng = new User( id: 4,  name: "feng",  age: 3);
        users.add(feng1);
        users.add(feng2);
        users.add(feng3);
        String s1 = JSON.toJSONString(feng);
        User user = JSON.parseObject(s1, User.class);
        System.out.println(user);
        JSONObject o = (JSONObject) JSON.toJSON(user);
        System.out.println(o);
        String s = JSON.toJSONString(users);
        return s;
    }
}
```

他的效果是和前面讲到的jackson是一样的

10.SSM框架整合 important

1.Mybatis层

1.建立相应的数据库



2.先建立实体类

```
public class Books {
    private int bookID;
    private String bookName;
    private int bookCounts;
    private String detail;
```

3.建立相应的接口类

```java
package com.feng.mapper;

import com.feng.pojo.Books;

import java.util.List;
import java.util.Map;

public interface BooksMapper {
    void addBook(Books books);
    void deleteBook(Map map);
    void updateBook(Map map);
    Books getBook(Map map);
    List<Books> getBooks();
}
```

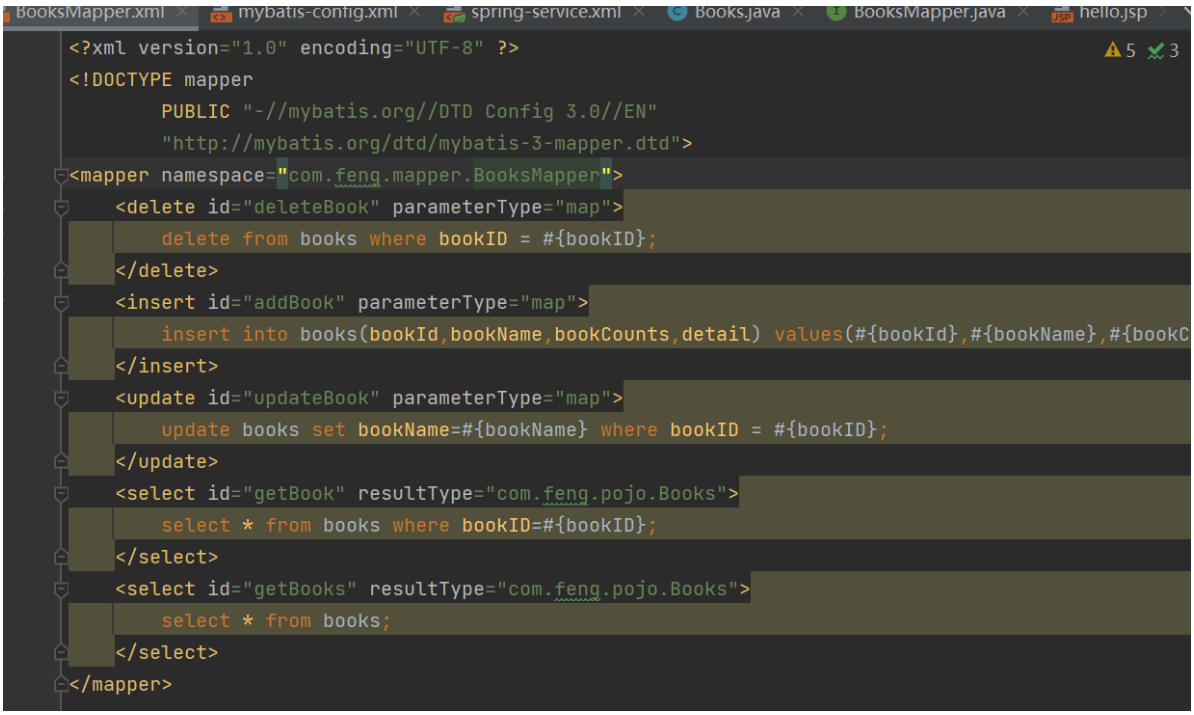
4.配置.xml文件

1先配置对应的mybatis-config.xml文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

</configuration>
```

2.配置对应的mapper文件

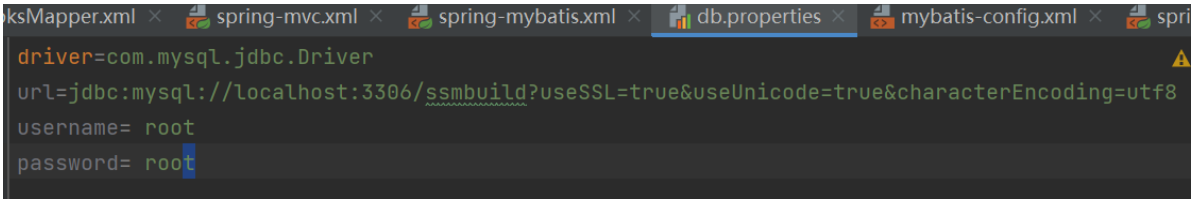
对应的实现方法类的配置



现在第一步被Mybatis-spring给整合了

```
<context:property-placeholder location="classpath:properties/db.properties"/>
```

上面的这个是导入文件，导入对应的properties文件，也就是存有驱动的文件，链接数据库



下面这张图中有新的数据源，也就是c3p0的数据源，这个数据源有很多的属性可以设置

下面的**org.mybatis.spring.mapper.MapperScannerConfigurer**，它会将查找到类路径下的映射器并自动将他们创建成MapperFactoryBean，也就是自动注册了，可以直接使用了，这个点需要多加了解**important**，还有这下面有个参数叫做**autoCommitOnClose**，连接池在回收数据库连接时是否自动提交事务。如果为false，则会回滚未提交的事务，如果为true，则会自动提交事务。这个参数不建议使用。要实现事务提交可以用AOP的事务提交

```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"
       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">
    <context:property-placeholder location="classpath:properties/db.properties"/>
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/ssmbuild?useSSL=true&amp;useUnicode=true&amp;characterEncoding=utf8"/>
        <property name="user" value="root"/>
        <property name="password" value="root"/>
        <property name="maxPoolSize" value="30"/>
        <property name="minPoolSize" value="10"/>
        <property name="autoCommitOnClose" value="true"/>
        <property name="checkoutTimeout" value="20000"/>
        <property name="acquireRetryAttempts" value="2"/>
    </bean>
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="mapperLocations" value="classpath:mapper/BooksMapper.xml"/>
    </bean>
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer" id="configurer">
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
        <property name="basePackage" value="com.feng.mapper"/>
    </bean>
</beans>
```

5.配置接口类的实现类

在这个类中可以实现别的业务，也就是加入别的功能，所以在service中配置，经过下面的xml文件的配置，就可以直接调用对应的方法来实现功能了

```java
package com.feng.service;

import com.feng.mapper.BooksMapper;
import com.feng.pojo.Books;

import java.util.List;
import java.util.Map;

public class BooksMapperImpl implements BooksMapper {
    private BooksMapper booksMapper;

    public void setBooksMapper(BooksMapper booksMapper) { this.booksMapper = booksMapper; }

    @Override
    public void addBook(Books books) { this.booksMapper.addBook(books); }

    @Override
    public void deleteBook(Map map) { this.booksMapper.deleteBook(map); }

    @Override
    public void updateBook(Map map) { this.booksMapper.updateBook(map); }

    @Override
    public Books getBook(Map map) { return this.booksMapper.getBook(map); }

    @Override
    public List<Books> getBooks() { return this.booksMapper.getBooks(); }
}
```

6.配置对应的service.xml文件

通过注册实现接口类的实体类，并将相应的属性给赋值，这个bookMapper就是之前自动生成的mapper，所以可以调用对应的方法

```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"
       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">
    <context:component-scan base-package="com.feng.service"/>
    <bean class="com.feng.service.BooksMapperImpl" id="mapper">
        <property name="booksMapper" ref="booksMapper"/>
    </bean>
    <bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager" id="manager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

2.SpringMVC层

1.建立对应的controller包

```java
package com.feng.controller;

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


@Controller
@RequestMapping("/test1")
public class HelloController {
    @RequestMapping("/t1")
    public String test1() { return "/hello"; }
}
```

2.建立对应的xml文件

这个文件和之前的没有区别

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

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

3.Spring层

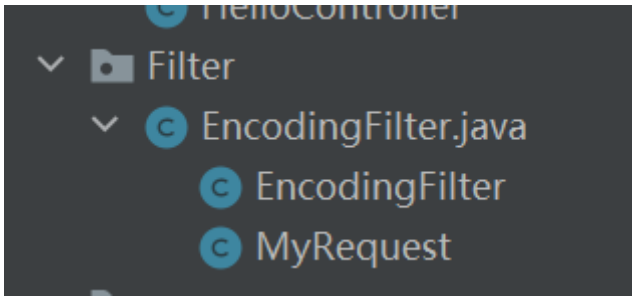
建立一个全部整合的文件

```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"
       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">
    <import resource="classpath*:/service/spring-mvc.xml"/>
     <import resource="classpath*:/service/spring-mybatis.xml"/>
     <import resource="classpath*:/service/spring-service.xml"/>
</beans>
```

4.整合SSM框架

jsp页面出现乱码，使用filter过滤器来实现字符转码，这个可以使用spring提供的Filter，还可以自己做一个Filter来实现该功能，记得要在web.xml文件中注册该filter

Filter文件中的Filter



web.xml

```xml
<filter>
    <filter-name>encoding</filter-name>
    <filter-class>com.feng.Filter.EncodingFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>encoding</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

最后就是通过jsp文件，也就是前端的界面来整合mapper接口中的方法来实现增删改查的操作

注意

在这个项目中还有一个功能没实现，就是删除的时候，编号自动变化

11.Ajax

1.使用iframe实现Ajax的效果

iframe就是一个页面内嵌技术，可以达到一部分页面的重复使用的效果，这个只是一个伪Ajax

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>iframe测试</title>
    <script type="text/javascript">
        function goto(){
            let value = document.getElementById("getUrl").value;
            document.getElementById("iframe1").src = value;
        }
    </script>
</head>
<body>
<p>
    <input type="text" value="" id="getUrl">
    <input type="button" value="提交" onclick="goto()">
</p>
<iframe id="iframe1" style="width: 100%;height:300px"></iframe>
</body>
</html>
```

2.真正的Ajax

真正的Ajax可以实现异步刷新的功能

实例1：

1.配置对应的操作界面

这里用到的onblur的意思是失去鼠标焦点的时候就会去调用方法

这里的方法调用了Ajax的post的方法，但是这个方法的实质还是$.Ajax

在这个方法中有四个参数url，data(可选)，callback(可选)，type(可选)

- url：是回调函数要实现的时候访问的url的地址
- data：是要传递的参数
- callback：是实现该方法的时候的回调函数，也就是接下来的操作

下面的success是回调函数，这个回调函数将data的数据传到了controller中，让其处理

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
    <script src="${pageContext.request.contextPath}/statics/jquery-3.6.0.js"></script>
    <script type="text/javascript">
        function a(){
            $.post({
                url:"${pageContext.request.contextPath}/a1",
                data:{"name":$("#username").val()},
                success:function (data,status){
                    console.log(data);
                    console.log(status);
                }
            })
        }
    </script>
</head>
<body>
<input type="text" id="username" onblur="a()">
</body>
</html>
```

2.配置对应当Controller文件

在这个controller中，加了一个@RestController，这个注解的含义是不会去调用视图解析器，也就不会去拼接路径

```java
}
@RequestMapping("/a1")
public void test2(String name, HttpServletResponse response) throws IOException {
    if("feng".equals(name)){
        response.getWriter().println("true");
    }else{
        response.getWriter().println("false");
    }
}
```

实例2：数据回显

1.配置对应的操作界面

```
<body>
<input type="button" id="btn" value="获取数据"/>
<table style="width: 80%;">
    <thead>
    <tr>
        <td>编号</td>
        <td>姓名</td>
        <td>年龄</td>
    </tr>
    </thead>
    <tbody id="content">
    </tbody>
</table>
</body>
```

2.配置对应的js的方法

这段是经典的jQuery的写法

回调函数将返回值放到这里，也就是data参数，这里在经过拼接技术，将界面恢复到了对应的标签下

```
<%--})--%>
$(document).ready(function (){
    $("#btn").click(function (){
        $.ajax({
            url:"${pageContext.request.contextPath}/a2",
            success:function (data){
                let html="";
                for (var i = 0; i <data.length ; i++) {
                    html+= "<tr>" +
                        "<td>"+data[i].id+"</td>"+
                        "<td>" + data[i].name + "</td>" +
                        "<td>"+data[i].age+"</td>"+
                        "</tr>"
                }
                $("#content").html(html);
            }
        })
    })
})
```

3.对应的controller文件

这里有个实体类，就是User的实体类

```
}
@RequestMapping("/a2")
public List<User> test3(){
    List<User> users = new ArrayList<User>();
    users.add(new User( id: 1, name: "feng", age: 3));
    users.add(new User( id: 2, name: "feng", age: 3));
    users.add(new User( id: 3, name: "feng", age: 4));
    return users;
}
```

实例3：账户验证

1.配置jsp文件

```
<body>
<label>用户名：</label>
<input type="text" id="username" onblur="a1()"/>
<span id="userinfo"></span>
<br>
<label>密码：</label>
<input type="text" id="password" onblur="a2()"/>
<span id="pwdinfo"></span>
</body>
```

2.配置对应的jQuery的方法

```
<script type="text/javascript">
    function a1(){
        $.ajax({
            url:"${pageContext.request.contextPath}/a3",
            data:{"name":$("#username").val()},
            success:function (data){
                if(data.toString()==="ok"){
                    $("#userinfo").css("color","green");
                }else{
                    $("#userinfo").css("color","red");
                }
                $("#userinfo").html(data);
            }
        })
    }
```

3.配置对应的controller方法

这个和if的顺序没有关系，因为前端的时候就设置了onblur的属性，所以只有失去焦点的时候才会调用方法

```java
@RequestMapping("/a3")
public String test4(String name,String pwd){
    String msg = "";
    if(name!=null){
        if(name.equals("feng")){
            msg = "ok";
        }else{
            msg = "用户名有误";
        }
    }
    if(pwd!=null){
        if(pwd.equals("123")){
            msg = "ok";
        }else{
            msg = "密码错误";
        }
    }
    return msg;
```

12.拦截器

1.过滤器和拦截器的区别

过滤器：

- 过滤是servlet的一部分，在任何javaweb的工程都可以使用
- 在url-pattern中配置了**/***之后，就要对所有的资源进行过滤(包括js,jsp等文件)

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等是不会拦截的

2.拦截器的实现

1.配置对应的实现类

通过继承**HandlerInterceptor**来实现拦截器的功能，但是这个类不需要重载方法，但是要实现拦截器的话，要重写这个类的第一个方法**preHandle**，这个类实现了AOP的方法

这个方法，当**return true**的时候，代表可以继续执行下去，当**return false**的时候，代表不会继续执行下去

```java
public class interceptorTest implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        return true;
    }
```

2.配置spring-mvc.xml文件

这个配置，也就是注册这个拦截器

这里的mapping要注意，`/**`代表的是包括包括路径及其子路径，而`\*`代表的只是下一个路径

```xml
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean class="com.feng.interceptor.interceptorTest"/>
    </mvc:interceptor>
</mvc:interceptors>
```

13.文件的上传和下载

1.设置要求

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

2.对表单中的enctype属性做个说明

- **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**：这种编码方式会议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，其他字符都不会做编码处理，这种方式适用直接通过表单发送邮件

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

一旦将**enctype**这个属性设置为**multipart/form-data**，浏览器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。

3.文件上传

SpringMVC提供了直接的支持，是通过MultipartResolver的类接口来实现的

1.导入**Apache Commons FileUpload**依赖

```
<!--文件上传支持-->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
<!--服务器支持-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
</dependency>
```

2.注册**multipartResolver**

- org.springframework.web.multipart.commons.CommonsMultipartResolver

这个文件名是upload的关键文件，通过注册这个文件，可以使用SpringMVC提供的文件下载的包，**注意这个id必须是multipartResolver，要不然会报错**

下面这段是固定的，可以直接使用

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="defaultEncoding" value="utf-8"/>
    <property name="maxUploadSize" value="10485760"/>
    <property name="maxInMemorySize" value="40960"/>
</bean>
```

3.写相关的controller类

下面的这段代码也是可以直接使用

```
package com.feng.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.commons.CommonsMultipartFile;

import javax.servlet.http.HttpServletRequest;
import java.io.*;

@Controller
public class HelloController3 {
    //@RequestParam("file") 将name=file控件得到的文件封装成CommonsMultipartFile 对象
    //批量上传CommonsMultipartFile则为数组即可
    @RequestMapping("/upload")
    public String fileUpload(@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("上传文件名 : "+uploadFileName);

        //上传路径保存设置
        String path = request.getServletContext().getRealPath("/upload");
        //如果路径不存在，创建一个
        File realPath = new File(path);
        if (!realPath.exists()){
            realPath.mkdir();
        }
        System.out.println("上传文件保存地址："+realPath);

        InputStream is = file.getInputStream(); //文件输入流
        OutputStream os = new FileOutputStream(new
File(realPath,uploadFileName)); //文件输出流

        //读取写出
        int len=0;
        byte[] buffer = new byte[1024];
        while ((len=is.read(buffer))!=-1){
            os.write(buffer,0,len);
            os.flush();
        }
        os.close();
        is.close();
        return "redirect:/index.jsp";
    }
}
```

4.相关jsp页面的配置

这里要注意enctype的类型还有method的方法类型

```
<%@ 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="upload">
        </form>
    </body>
</html>
```

4.文件下载

方式一：

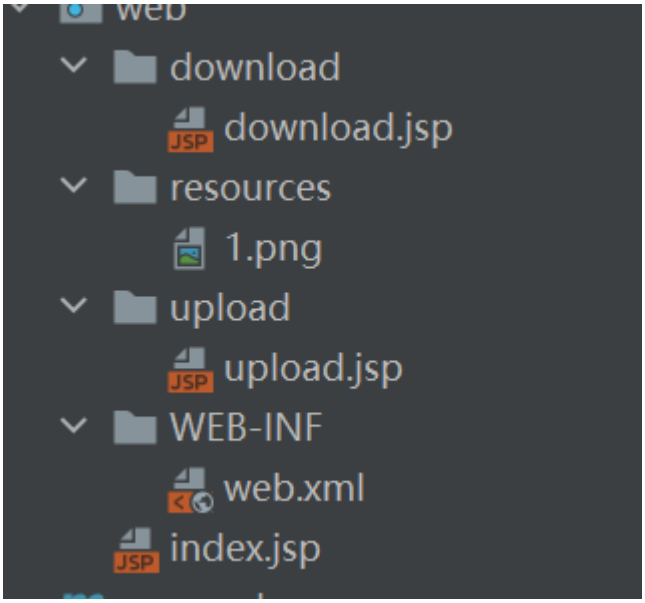
直接使用一个来进行资源定向，让浏览器来替我们做这个文件下载的事情

方式二：
配置一个controller类

代码可以直接使用，不过要改文件路径，因为这些都是静态资源所以可以固定下载路径

```java
package com.feng.controller;

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URLEncoder;

@Controller
public class HelloController4 {
    @RequestMapping(value="/download")
    public String downloads(HttpServletResponse response , HttpServletRequest
request) throws Exception{
        //要下载的图片地址
        String  path = request.getServletContext().getRealPath("/resources");
        String  fileName = "1.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 input=new FileInputStream(file);
        //3、 写出文件--输出流
        OutputStream out = response.getOutputStream();

        byte[] buff =new byte[1024];
        int index=0;
        //4、执行 写出操作
        while((index= input.read(buff))!= -1){
            out.write(buff, 0, index);
            out.flush();
        }
        out.close();
        input.close();
        return null;
    }
}
```

文件结构




```
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.3.3</version>
</dependency>
```

记得排查lib的文件的存在